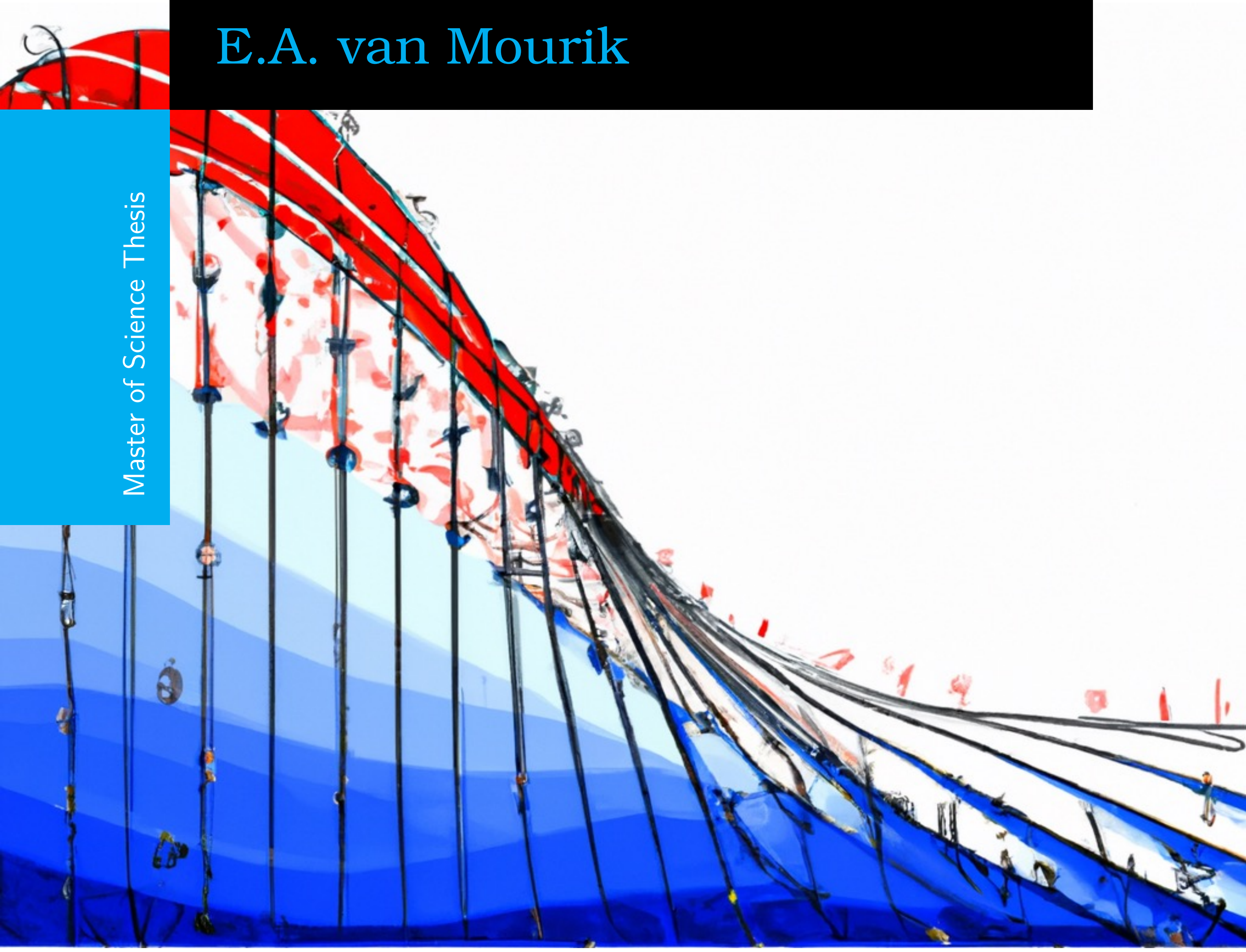
All-at-once optimization for kernel machines with canonical polyadic decompositions

Enabling large scale learning for kernel machines

E.A. van Mourik

Master of Science Thesis



Delft Center for Systems and Control

All-at-once optimization for kernel machines with canonical polyadic decompositions

Enabling large scale learning for kernel machines

Master of Science Thesis

For the degree of Master of Science in Systems and Control at Delft University of Technology

E.A. van Mourik

October 31, 2022

Faculty of Mechanical, Maritime and Materials Engineering (3mE) · Delft University of Technology

Abstract

This thesis studies the Canonical Polyadic Decomposition (CPD) constrained kernel machine for large scale learning, i.e. learning with a large number of samples. The kernel machine optimization problem is solved in the primal space, such that the complexity of the problem scales linearly in the number of samples as opposed to scaling cubically in the dual space. Product feature maps are applied to transform the input data. The weights are constrained to be a CPD, so the number of weights scales linearly in the number of features. The CPD introduces a nonlinearity, so nonlinear optimization must be applied.

It is studied in which situation it is more advantageous to apply iterative all-at-once optimization compared to Alternating Least Squares (ALS) to solve the CPD constrained kernel machine problem. Specifically, all-at-once gradient descent methods are studied. An efficient analytical algorithm for the all-at-once gradient is derived. Furthermore, it is shown that automatic differentiation (AD) can also be applied, but it is found to be slower than the analytical method.

The selection of a step size is found to be challenging. It is shown that the magnitude of the gradient of the mean squared error (MSE) term decreases for an increasing number of features. As a result, selecting the step size becomes more difficult for more features. To overcome this, the Line search and the Adam method are studied. A general expression for the exact line search solution is derived. It can be applied to compute the optimal step size for any step direction and any number of features. However, the Adam method performs better in terms of loss after training, convergence and the training run time. The mini-batch Adam method is used to evaluate the performance of all-at-once optimization for large scale learning.

It is found that the Adam method no longer performs well for data sets with around 16 features or more, likely due to the decrease in the magnitude of the gradient of the MSE term. On large scale data sets with fewer features, the Adam method outperforms ALS in terms of run time until convergence while achieving similar training and validation losses. The Adam method reached convergence on a data set with 11 million samples within ten minutes. Furthermore, it is shown that the scaling of the run time of the Adam method in terms of the feature map order and the CP-rank is more than an order lower than the scaling of ALS when the methods are run on a GPU. This makes to Adam method more suitable for more complex models.

Table of Contents

Preface & Acknowledgements

Machine learning is transforming the world we live. Research pushes the boundaries both in terms of machine learning applications as well as novel methods and algorithms. For example, the picture on the cover of this thesis was created by the Artificial Intelligence system DALL·E when it was prompted to create an image of a tensor decomposition. Machine learning is a topic that has always interested me, so it has been a pleasure to study it and, hopefully, make a small contribution towards something greater.

I would like to thank my supervisor dr.ir. Kim Batselier for introducing me to the world of tensors, tensor decompositions and their application in machine learning. Moreover, I would to thank him and ir. Frederiek Wesel for all the guidance and feedback they have given my during this thesis. The bi-weekly meetings that we had were not only interesting and helpful, but also nice and fun. Additionally, I would like to thank my friends and family for their support and encouragement.

Finally, I would like to thank my mom for everything. I know you would have loved to read my thesis and see my presentation. Although you are not here now, you will always be with us.

Delft, University of Technology E.A. van Mourik
October 31, 2022

Voor mama

Chapter 1

Introduction

Machine learning is a hot topic nowadays, both in the public as well as in the research community. New approaches and applications are researched constantly. Machine learning has entered the daily life of humankind and finds billions of users at every moment. As a result, there is a continuous desire for newer and more advanced models.

A popular trend within machine learning is to increase the number of weights in a machine learning model to more complex models for more complex problems. For example, two years ago a model with 175 billion parameters was trained for natural language processing [1]. A major downside of such gargantuan models is that they require an enormous amount of training and, as a result, an tremendous amount of computation time. This requires huge amounts of energy to run servers, so training these models comes at a cost.

Additionally, more and more data is available to be used for training. Just like humans, when a model has more examples to learn from this generally improves its accuracy. However, using more data also implies a longer training time. How the training time scales with the number of data samples can have a critical impact and differs per learning method. A cubic scaling entails that training with 10 times more samples takes 1000 times as long. For example, when training with one million samples takes one hour, this implies that training with 10 million samples takes 42 days. An increase in data from one to ten million is reasonable nowadays. So, to save time and energy, efficient learning methods need to be designed that scale favorable in the number of training samples.

This thesis studies the supervised kernel machine, a class of machine learning models. Supervised machine learning is a category of machine learning in which the objective is to predict an output based on input data. Examples include determining whether an image contains a cat or predicting a household's electricity consumption. A supervised machine learning model consists of weights that map the input data to a prediction. The model is trained by solving an optimization problem. The model weights are updated based on an optimization method and training data to minimize the error between the prediction and the true output. The supervised kernel machine is a general class of machine learning models that has been successfully applied in various domains, including natural language processing [2] and the annotation of DNA sequences [3]. The Support Vector Machine (SVM) [4] is a well-known example of a supervised kernel machine.

This thesis focuses on large scale learning which is learning with large data sets, e.g. data sets with over a million samples. The common optimization approach for kernel machines scales poorly in the number of samples, so it is less suitable for a large data sets. Therefore, this thesis studies methods that enable large scale learning for kernel machines.

In the next section, the problem is introduced in more detail as well as the goals of this thesis. Then the outline and the contributions of this thesis are presented. Lastly, the implementation of the algorithms is shortly discussed.

1-1 Problem introduction

A supervised kernel machine makes a prediction, y_{pred}, based on its model weights $\mathbf{w}$ and the input $\mathbf{x}$ which is transformed with feature map $\mathbf{z}$. So, the prediction is given by

$$y_{pred} = \langle \mathbf{w}, \mathbf{z}(\mathbf{x}) \rangle .$$

Learning entails optimizing the model weights based on the training data set $\{\mathbf{x}_n, y_n\}_{n=1}^{N}$, where y_n denotes the true output value. The training set contains N samples and each sample has D features, $\mathbf{x}_n \in \mathbb{R}^D$. Kernel machines are commonly solved in the dual space with the kernel trick [4]. However, the computational complexity of this method scales with order $\mathcal{O}(N^3)$. This makes the kernel trick practically infeasible for large scale learning.

Alternatively, the problem can be solved in the primal space where the computation of the solution scales linearly with the number of samples. So-called product feature maps are used to transform the data. However, due to this feature map, the number of model weights, and as a result the complexity, scales exponentially with the number of features. So, for a larger number of features this problem also becomes infeasible. To overcome this, the model weights are constrained to be a tensor decomposition. A tensor decomposition is similar to a matrix decomposition such as the QR-decomposition, but for higher dimensions. In this thesis, the Canonical Polyadic Decomposition (CPD) [5] is studied in particular. The advantage of using the CPD to constrain the model weights is that the resulting number of weights scales linearly in the number of features.

This results in the CPD constrained kernel machine optimization problem. In this thesis, the Kernel Ridge Regression (KRR) optimization problem is studied specifically. It can be applied for both classification and regression tasks. The CPD introduces nonlinearity in the optimization problem which makes it harder to solve. Currently, the state-of-the-art optimization method for the CPD constrained kernel machine is Alternating Least Squares (ALS) [6, 7]. With ALS one factor of the CPD is optimized at a time.

A potential alternative is to optimize all the factors of the CPD simultaneously, i.e. all-at-once optimization. This thesis studies whether all-at-once optimization can effectively be applied to the CPD constrained kernel machine. Particularly, several all-at-once gradient descent methods are studied. Furthermore, since ALS and all-at-once optimization are inherently different methods, it is examined how the methods compare to each other.

Ultimately, the goal of this thesis is to answer the following:

> ***In which situation is it more advantageous to apply all-at-once optimization compared to ALS to solve the CPD constrained kernel machine problem?***

Three sub-goals are formulated to achieve this:

- *Create a proof of concept to show that all-at-once optimization can be applied to the CPD constrained kernel machine and study different all-at-once optimization methods.*
- *Identify challenges of all-at-once optimization for CPD weights and propose solutions to overcome them.*
- *Study how all-at-once optimization and ALS compare to each other in terms of method characteristics, loss after training and convergence.*

1-2 Thesis outline and contributions

The outline of the thesis will be presented here. Additionally, for each chapter the contributions are listed.

In Chapter 2, the notion of tensors, tensor decompositions in general and the Canonical Polyadic Decomposition (CPD) are introduced. Additionally, basic linear algebra, tensor and CPD operations are introduced. Finally, it is shown how a CPD can, under certain assumptions, be stored as a tensor in computer memory.

Contributions Chapter 2

- It is shown that, under assumptions, a CPD can be represented as a tensor and this can significantly speed up computations.

Kernel machines in general are discussed in Chapter 3. The Kernel Ridge Regression (KRR) problem is presented which forms the basis for the optimization problem used in this thesis. It is shown how this problem can be approached in the primal space with product feature maps. The CPD constrained kernel machine is introduced and challenges for solving the CPD constrained kernel machine are discussed. Nonlinear optimization in general is discussed as well as Alternating Least Squares (ALS). Finally, an overview of state-of-the-art learning with CPDs is given.

Contributions Chapter 3

- It is shown that, under assumptions, a product feature map can be represented as a tensor which can be combined with the tensor representation of a CPD for efficient computations.

In Chapter 4, all-at-once optimization of the CPD constrained kernel machine is discussed. The iterative all-at-once optimization framework and the Steepest Gradient Descent (SteGD) method are introduced. The algorithm for computing the all-at-once gradient that was derived for this thesis is presented and automatic differentiation (AD) is introduced as an alternative for the analytical algorithm. In a proof of concept, it is shown with experiments that SteGD works on relatively small data sets. Furthermore, challenges of SteGD are identified. Lastly, all-at-once optimization and ALS are compared based on the characteristics of the methods.

Contributions Chapter 4

- It is shown that SteGD can be applied to the CPD constrained kernel machine.
- An efficient algorithm for the computation of the all-at-once gradient is derived.
- It is shown that AD can be applied to the CPD constrained kernel machine.
- It is shown that the theoretical computational and memory complexity of SteGD scales favorably compared to ALS in terms of the feature map order and CP-rank. This is verified with experiments.

The challenges identified in Chapter 4 are analyzed in more detail in Chapter 5. The selection of the step size is a challenge, so two methods to tackle this are presented. The step size, also called the learning rate, determines how much the weights are updated per iteration. First, Line search is studied. An expression for the exact line search solution of the studied optimization problem is derived for this thesis and presented here. The performance of the Line search method is evaluated with experiments. Secondly, the Adam method [8] is studied. Finally, at the end of the chapter, the three all-at-once optimization methods, SteGD, Line search and the Adam method, are compared. The Adam method is identified as the best all-at-once optimization method and is used in the next chapter for a more in-depth comparison with ALS.

Contributions Chapter 5

- It is found that the absolute values of the gradient of the mean squared error (MSE) term decrease for an increasing number of features. For the regularization term this is not the case. This has implications for the step size and the regularization parameter.
- A general expression for the exact line search solution is derived. It is general, because it can be used with any step direction and number of features.
- It is shown that the Line search method can speed up to training convergence compared to SteGD, but does not improve the performance in terms of the final loss. Furthermore, it is found that the current implementation is relatively slow.
- It is shown that the Adam method can achieve similar performance in terms of final losses as ALS on small data sets.
- It is concluded that the Adam method outperforms the other two all-at-once optimization methods in terms of loss performance, convergence speed and computational and memory complexity.

In Chapter 6 the performance of ALS and all-at-once optimization in the form of the Adam method are compared. The resulting losses and training run times are used as performance criteria. Experiments are conducted on different data sets, including large data sets to investigate the large scale learning performance. First, it is shown that the Adam method does not work well for a larger number of features while ALS does. Then, the general performance of the two methods is compared on data sets with fewer features. The performance is evaluated in terms of the final training and validation loss as well as the training and convergence run time. Thirdly, the scaling of the training run time of both methods as a function of the feature map order and CP-rank is studied. Finally, it is investigated whether the Adam method and ALS can be combined to exploit the best of both methods.

Contributions Chapter 6

- It is found that the Line search and Adam method no longer perform well for around 16 features or more while ALS does.
- It is shown that on data sets with a relatively small number of samples ALS is superior to the Adam method in terms of loss performance and has similar convergence.
- It is shown that on large scale data sets, i.e. data with a large number of samples, the Adam method achieves similar performance as ALS in terms of final losses, but the Adam method has a significantly lower run time until convergence.
- It is shown that the training run time of the Adam method scales favorably compared to ALS for the feature map order and CP-rank. Hence, for more complex models on large scale data sets it is more advantageous in terms of the training run time to apply the Adam method.
- It is shown that the Adam method and ALS can be combined to improve the convergence. It does, however, likely not improve the final loss performance compare to the ALS method alone.

The main body of this thesis finishes with Chapter 7 in which a conclusion and recommendations for future research are given.

1-3 Implementation

The implementation of the different algorithms used in this thesis was done in Python. In particular, the JAX library [9] is extensively used. JAX is a high-performance numerical computing library that has several functionalities that were useful for this thesis. JAX offers AD functionality and allows the jitting of functions. Jitting, short for Just-In-Time compilation, is shortly introduced in Section C-1. Jitting generally results in a significant reduction of the execution time of functions and, as a result, of training run time. Furthermore, the jit functionality of JAX allows for code to be run on a GPU or TPU[1], enabling even faster

[1]GPU: Graphical Processing Unit. TPU: Tensor Processing Unit.

training. Therefore, it was possible to run many of the experiments that were conducted for this thesis on a GPU of a server of the Delft Center for Systems and Control.

On several occasions in this thesis a reference is made to a Numpy/JAX functionality. This implies that this is a standard Numpy functionality that is also implemented in JAX. So, it can be used without JAX if this is desired.

All the code developed for this thesis can be found in the GitHub repository at `https://github.com/ewoudvm/all-at-once-cpd-learning`.

Chapter 2

Tensors and the Canonical Polyadic Decomposition

In this thesis the Canonical Polyadic Decomposition (CPD) is used extensively. The CPD is a so-called tensor decomposition. The notion of tensors will first be introduced, as well as some basic tensor operations and tensor decompositions in general. In the second part of this chapter the Canonical Polyadic Decomposition is introduced followed by some CPD operations that will be used later in this thesis. All the introduced notation is summarized at the end of this chapter in Table 2-1.

2-1 Tensors

Tensors are a generic format to represent data using a given number of dimensions. Well known forms of tensors are the zero-dimensional scalar, the one-dimensional vector and the two-dimensional matrix. In general, a tensor has D dimensions, also called a D-way tensor. A tensor is said to be higher-order when it has 3 or more dimensions.

Tensors can be used as a natural representation for a wide variety of data. For example, an image with an x, y and color dimension can naturally be seen as a 3-way tensor. Furthermore, a video is simply a sequence of images and can thus be represented as a 4-way tensor with time as the fourth dimension.

In this section the preliminaries and basic operations for tensors will be introduced. Additionally, tensor decompositions in general will be discussed. The following section is based on [10] and [11].

2-1-1 Preliminaries

Throughout this thesis the following notation will be used which is consistent with the literature. Higher-order tensors are represented by a bold calligraphic capital letter, $\boldsymbol{\mathcal{A}}$, matrices are represented by a bold capital letter, $\mathbf{A}$, vectors are represented by a bold small letter, $\mathbf{a}$ and scalars are represented by an italic letter, a or A. In this thesis it is assumed that all tensors are real-valued. Therefore, a tensor with D dimensions is given by $\mathcal{A} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_D}$. Here I_d is the size of the dth dimension of tensor $\mathcal{A}$. An element of the tensor is denoted as $\mathcal{A}(i_1, i_2, \ldots, i_D) = a_{i_1,i_2,\ldots,i_D}$.

The entries of a tensor can be rearranged into a vector which is called vectorization. The vectorization operation, denoted as $\text{vec}(\cdot)$, converts a D-way tensor into a vector with one dimension of size $I_1 I_2 \cdots I_D$. In [10] it is discussed that the order in which the tensor elements are stacked in the vectorization can differ in the literature and that the specific order does not matter in general as long as the order is consistent. However, in this thesis the row-major order is used specifically, such that a useful property holds. This will be further clarified in Eq. (2-11).

Rank-one tensor

The rank-one tensor is a specific kind of tensor. A rank-one tensor $\mathcal{A} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_D}$ is defined as the outer product of D vectors where the dth vector is of size I_d:

$$\mathcal{A} = \mathbf{a}^{(1)} \circ \mathbf{a}^{(2)} \circ \cdots \circ \mathbf{a}^{(D)}, \quad \mathbf{a}^{(d)} \in \mathbb{R}^{I_d}. \tag{2-1}$$

Here $\circ$ represents the vector outer product. Each individual element $a_{i_1,i_2,\ldots,i_D}$ of the rank-one tensor is a product of the vectors elements $a^{(d)}_{i_d}$:

$$a_{i_1,i_2,\ldots,i_D} = a^{(1)}_{i_1} a^{(2)}_{i_2} \ldots a^{(D)}_{i_D}. \tag{2-2}$$

In Figure 2-1 a three dimensional rank-one tensor is illustrated. The rank-one tensor is introduced here, since it is used extensively throughout this thesis. It forms the basis for the CPD and it will be used in Section 3-1-2 for product feature maps.

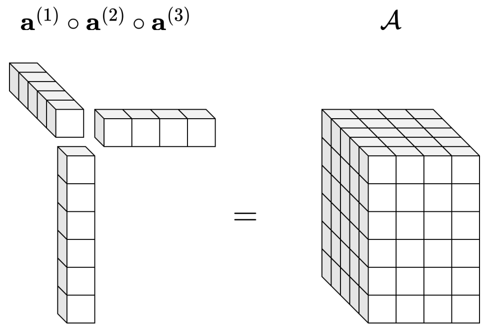

$\mathbf{a}^{(1)} \circ \mathbf{a}^{(2)} \circ \mathbf{a}^{(3)}$
$\mathcal{A}$
$=$

Figure 2-1: Visualization of a 3-way rank-one tensor.

2-1-2 Basic operations

In this section several basic matrix and tensor operations will be introduced. For a more extensive overview of tensor operations, the reader is advised to read [12].

Matrix operations

Besides the ordinary matrix-matrix and matrix-vector products, the Kronecker product, the Hadamard product and the Khatri-Rao product are used in this thesis.

For the Kronecker product the symbol $\otimes$ is used. Given the matrices $\mathbf{A} \in \mathbb{R}^{I\times J}$ and $\mathbf{B} \in \mathbb{R}^{K\times L}$, the Kronecker product is defined as

$$\mathbf{A}\otimes\mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & a_{12}\mathbf{B} & \dots & a_{1J}\mathbf{B} \\ a_{21}\mathbf{B} & a_{22}\mathbf{B} & \dots & a_{2J}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ a_{I1}\mathbf{B} & a_{I2}\mathbf{B} & \dots & a_{IJ}\mathbf{B} \end{bmatrix} \in \mathbb{R}^{IK\times JL}, \tag{2-3}$$

where a_{ij} is an element of matrix $\mathbf{A}$. A sequence of D Kronecker products denoted as $\bigotimes_{d=1}^{D}\mathbf{A}^{(d)}$.

The Hadamard product, also known as the element-wise matrix product, is denoted as $*$. For two matrices of the same size, $\mathbf{A},\ \mathbf{B} \in \mathbb{R}^{I\times J}$, it is defined as follows:

$$\mathbf{A}*\mathbf{B} = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} & \dots & a_{1J}b_{1J} \\ a_{21}b_{21} & a_{22}b_{22} & \dots & a_{2J}b_{2J} \\ \vdots & \vdots & \ddots & \vdots \\ a_{I1}b_{I1} & a_{I2}b_{I2} & \dots & a_{IJ}b_{IJ} \end{bmatrix} \in \mathbb{R}^{I\times J}. \tag{2-4}$$

For a sequence of D Hadamard products $\circledast$ is used:

$$\mathbf{A}^{(1)} * \mathbf{A}^{(2)} * \dots * \mathbf{A}^{(D)} = \underset{d=1}{\overset{D}{\circledast}} \mathbf{A}^{(d)}. \tag{2-5}$$

The Khatri-Rao product, $\odot$, in this thesis refers to the column-wise Khatri-Rao product. For the matrices $\mathbf{A} \in \mathbb{R}^{I\times J},\ \mathbf{B} \in \mathbb{R}^{K\times J}$, it is defined as

$$\mathbf{A}\odot\mathbf{B} = [\mathbf{a}_1\otimes\mathbf{b}_1\ \mathbf{a}_2\otimes\mathbf{b}_2\ \cdots\ \mathbf{a}_J\otimes\mathbf{b}_J] \in \mathbb{R}^{IK\times J}, \tag{2-6}$$

where $\mathbf{a}_j$ and $\mathbf{b}_j$ are the columns of $\mathbf{A}$ and $\mathbf{B}$ respectively.

Although the symbols $\odot$ and $*$ are used in this thesis for the Khatri-Rao and Hadamard product respectively, it is noted that in the literature these symbols can be switched. Additionally, the vector outer product is in the literature sometimes denoted with $\otimes$ as well, but here only $\circ$ will be used.

The matrix products have several useful properties. The Hadamard and Khatri-Rao product have the following properties [13]:

$$\begin{aligned} \mathbf{A}*\mathbf{B} &= \mathbf{B}*\mathbf{A}, \\ (\mathbf{A}*\mathbf{B})*\mathbf{C} &= \mathbf{A}*(\mathbf{B}*\mathbf{C}), \\ \mathbf{A}\odot(\mathbf{B}\odot\mathbf{C}) &= (\mathbf{A}\odot\mathbf{B})\odot\mathbf{C}. \end{aligned} \tag{2-7}$$

Moreover, there are also some properties that involve both products, so-called mixed-product properties [10]. One property that is particularly useful within the context of this thesis involves the Khatri-Rao and the Hadamard product:

$$(\mathbf{A}\odot\mathbf{A})^\top(\mathbf{B}\odot\mathbf{B}) = \mathbf{A}^\top\mathbf{A}*\mathbf{B}^\top\mathbf{B}. \tag{2-8}$$

In this thesis the row-major vectorization is used. This is used, since it results in convenient relations between vectorization and the Kronecker product. In the row-major order the rows of a matrix are stacked vertically to form the vectorized matrix. The following example illustrates this:

$$\operatorname{vec}(\begin{bmatrix} a_1b_1 & a_1b_2 \\ a_2b_2 & a_2b_2 \\ a_3b_1 & a_3b_2 \end{bmatrix}) = \begin{bmatrix} a_1b_1 & a_1b_2 & a_2b_2 & a_2b_2 & a_3b_1 & a_3b_2 \end{bmatrix}^\top. \tag{2-9}$$

With the row-major vectorization, the Kronecker product and the vector outer product have the following relationship:

$$\mathbf{a} \otimes \mathbf{b} = \operatorname{vec}(\mathbf{a} \circ \mathbf{b}). \tag{2-10}$$

This relationship can be demonstrated as follows:

$$\begin{gathered} \mathbf{a} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}, \quad \mathbf{a} \circ \mathbf{b} = \begin{bmatrix} a_1b_1 & a_1b_2 \\ a_2b_2 & a_2b_2 \\ a_3b_1 & a_3b_2 \end{bmatrix}, \\ \mathbf{a} \otimes \mathbf{b} = \begin{bmatrix} a_1b_1 & a_1b_2 & a_2b_2 & a_2b_2 & a_3b_1 & a_3b_2 \end{bmatrix}^\top = \operatorname{vec}(\mathbf{a} \circ \mathbf{b}). \end{gathered} \tag{2-11}$$

The Kronecker product can be used in combination with the standard matrix-vector product to obtain a vectorization of a product or the inverse of a vectorization [14]:

$$\operatorname{vec}(\mathbf{A}\mathbf{X}\mathbf{B}^\top) = (\mathbf{A} \otimes \mathbf{B}) \operatorname{vec}(\mathbf{X}). \tag{2-12}$$

This is particularly useful to switch between a vector representation and a matrix representation. Note that this is a different formulation than the one that is commonly used, since this relation is normally formulated for column-major ordering.

Lastly, a special kind of outer product is used in this thesis. This outer product can be thought of as a batch outer product. Namely, it is used to compute the standard vector outer product for a batch of vectors simultaneously. Given the matrices $\mathbf{A} = [\mathbf{a}^{(1)}\ \mathbf{a}^{(2)}\ \ldots\ \mathbf{a}^{(D)}] \in \mathbb{R}^{I \times D}$ and $\mathbf{B} = [\mathbf{b}^{(1)}\ \mathbf{b}^{(2)}\ \ldots\ \mathbf{b}^{(D)}] \in \mathbb{R}^{J \times D}$, the batch outer product, denoted as $\tilde{\circ}$, is given by

$$\mathcal{C} = \mathbf{A} \mathbin{\tilde{\circ}} \mathbf{B} \in \mathbb{R}^{I \times J \times D}. \tag{2-13}$$

Furthermore,

$$\mathcal{C}_{:,:,d} = \mathbf{a}^{(d)} \circ \mathbf{b}^{(d)} \in \mathbb{R}^{I \times J}, \ d = 1, \ldots, D.$$

This concept can be extended to higher dimensions when series of batch outer products are used. The axis that contains contains the batch is then always the last axis. For example, this can be used in a 4-way case as follows:

$$\begin{gathered} \mathbf{A}_i = [\mathbf{a}_i^{(1)}\ \mathbf{a}_i^{(2)}\ \ldots\ \mathbf{a}_i^{(D)}] \in \mathbb{R}^{I_i \times D}, \ i = 1, 2, 3, 4, \\ \mathcal{C} = \mathbf{A}_1 \mathbin{\tilde{\circ}} \mathbf{A}_2 \mathbin{\tilde{\circ}} \mathbf{A}_3 \mathbin{\tilde{\circ}} \mathbf{A}_4 \in \mathbb{R}^{I_1 \times I_2 \times I_3 \times I_4 \times D}, \\ \mathcal{C}_{:,:,:,:,d} = \mathbf{a}_1^{(d)} \circ \mathbf{a}_2^{(d)} \circ \mathbf{a}_3^{(d)} \circ \mathbf{a}_4^{(d)} \in \mathbb{R}^{I_1 \times I_2 \times I_3 \times I_4}, \ d = 1, \ldots, D. \end{gathered}$$

It is noted that this batched outer product is a tensorized version of the Khatri-Rao product. The Khatri-Rao product $\mathbf{A}_1 \odot \mathbf{A}_2 \odot \mathbf{A}_3 \odot \mathbf{A}_4$ results in a $I_1I_2I_3I_4 \times D$ matrix of which the dth column is the vectorized version of $\mathcal{C}_{:,:,:,:,d}$.

Tensor operations

The most general tensor operation is the tensor contraction. It is the sum over a repeated index of two tensors. For example, for tensors $\mathcal{A} \in \mathbb{R}^{I_1 \times I_2 \times I_3}$ and $\mathcal{B} \in \mathbb{R}^{I_3 \times I_4 \times I_5}$ the tensor contraction over index i_3 is defined as follows [11]:

$$\mathcal{C}(i_1, i_2, i_4, i_5) = \sum_{i_3=1}^{I_3} \mathcal{A}(i_1, i_2, i_3)\mathcal{B}(i_3, i_4, i_5). \tag{2-14}$$

A matrix-matrix product is an example of a tensor contraction for two 2-way tensors. A specific kind of tensor contraction is the mode-d vector product. Given the tensor $\mathcal{A} \in \mathbb{R}^{I_1 \times \cdots \times I_D}$ and the vector $\mathbf{b} \in \mathbb{R}^{I_d}$, the mode-$d$ vector product is given by

$$(\mathcal{A} \times_d \mathbf{b})_{i_1,\ldots,i_{d-1},i_{d+1},\ldots,i_D} = \sum_{i_d=1}^{I_d} a_{i_1,i_2,\ldots,i_D} b_{i_d}. \tag{2-15}$$

Note that the result is a tensor of size $D-1$ where the dth dimension has be removed.

For two tensors of the same size, $\mathcal{A}$, $\mathcal{B} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_D}$, the Frobenius inner product is defined as the contraction over all the indices:

$$\langle \mathcal{A}, \mathcal{B} \rangle_F = \sum_{i_1=1}^{I_2} \sum_{i_2=1}^{I_2} \cdots \sum_{i_D=1}^{I_D} a_{i_1,i_2,\ldots,i_D} b_{i_1,i_2,\ldots,i_D}. \tag{2-16}$$

Alternatively, the Frobenius inner product can be computed by first vectorizing both tensors and subsequently computing the inner product of the resulting vectors:

$$\langle \mathcal{A}, \mathcal{B} \rangle_F = \langle \mathrm{vec}(\mathcal{A}), \mathrm{vec}(\mathcal{B}) \rangle = \mathrm{vec}(\mathcal{A})^\top \mathrm{vec}(\mathcal{B}). \tag{2-17}$$

In the special case where the two tensor are rank-one tensors, the Frobenius inner product can be computed more efficiently. Given the two rank-one tensors $\mathcal{A} = \mathbf{a}^{(1)} \circ \mathbf{a}^{(2)} \circ \cdots \circ \mathbf{a}^{(D)}$ and $\mathcal{B} = \mathbf{b}^{(1)} \circ \mathbf{b}^{(2)} \circ \cdots \circ \mathbf{b}^{(D)}$, the Frobenius inner product can be computed without constructing the full tensor as follows [12]:

$$\langle \mathcal{A}, \mathcal{B} \rangle_F = \prod_{d=1}^{D} \left\langle \mathbf{a}^{(d)}, \mathbf{b}^{(d)} \right\rangle. \tag{2-18}$$

The ability to compute the inner product without constructing the full tensor generally reduces the computational and memory complexity of the operation. Assuming all dimensions are of size I, $I_1 = \ldots = I_d = I$, the complexity for a full tensor inner product is $\mathcal{O}(I^D)$, since each tensor has I^D elements. $\mathcal{O}(\cdot)$ refers to the Big O notation[1]. However, for the inner product given in (2-18) the complexity is $\mathcal{O}(ID)$. For each tensor D vectors of size I need to be stored and D inner products with vectors of size I need to be computed, hence the complexity is $\mathcal{O}(ID)$. Consequently, the complexity does not scale exponentially in the number of dimensions anymore. This thus makes it possible to work with larger dimensions and especially a larger number of dimensions.

[1]See https://xlinux.nist.gov/dads/HTML/bigOnotation.html for more information.

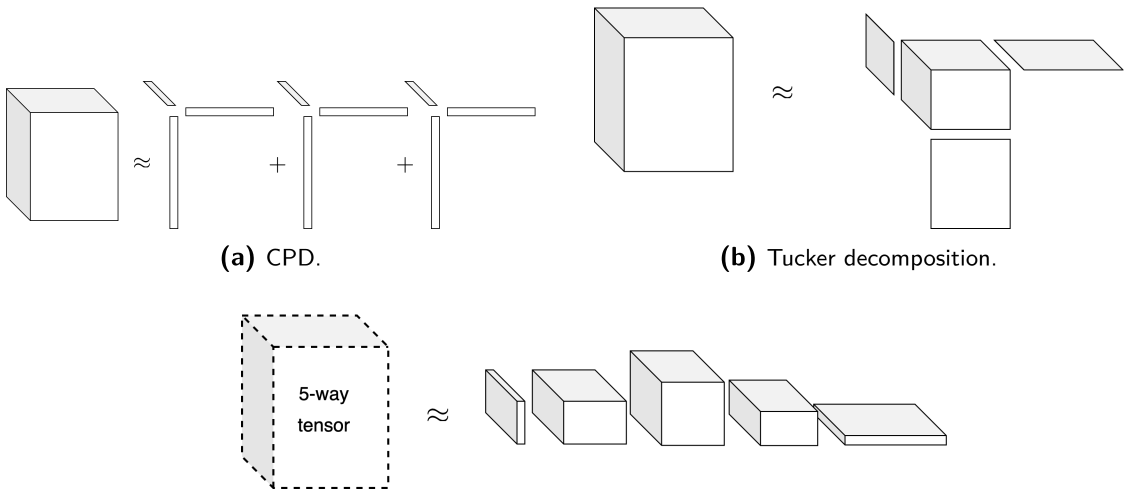
The Frobenius norm of a tensor is defined as

$$\|\mathcal{A}\|_F = \sqrt{\sum_{i_1=1}^{I_1} \sum_{i_2=1}^{I_2} \cdots \sum_{i_D=1}^{I_D} a^2_{i_1,i_2,\ldots,i_D}}. \tag{2-19}$$

From this it can easily be deduced that $\langle \mathcal{A}, \mathcal{A} \rangle_F = \|\mathcal{A}\|_F^2$.

2-1-3 Tensor decompositions

In standard linear algebra there are several methods to decompose matrix, such as the singular value decomposition (SVD) or the QR-decomposition. Each decomposition has different characteristics and applications. Similarly, tensors can be decomposed into tensor decompositions or tensor networks. Three well known tensor decompositions are the Canonical Polyadic Decomposition (CPD), the Tucker decomposition [15] and the Tensor Train (TT) decomposition [16]. The decompositions are sketched in Figure 2-2. In short, for a D-way tensor the CPD consists of a sum of D-way rank-one tensors. The Tucker decompositions consists of a smaller D-way core and D matrices and the TT consists of D 3-way cores. Each decomposition has its advantage and disadvantages. For example, a major disadvantage of the Tucker decomposition is that the number of elements, and thus the complexity of operations with the decomposition, still scales exponentially in D due to the core in the decomposition. For an overview of these tensor decompositions and their applications the reader is advised to read [10].



(a) CPD. **(b)** Tucker decomposition.

(c) TT.

Figure 2-2: Visualization of three different tensor decompositions. For the first two, the decomposition of a 3-way tensor is shown. For the TT the decomposition of a 5-way tensor is shown. Since a 5-way tensor cannot be visually represented it is shown as a box with dashed lines.

Tensor decompositions can in general be applied in two ways. A known tensor can be approximated using a tensor decomposition, for example to reduce the storage of tensor or to extract information from of the tensor. Tensor decompositions have, for instance, been applied to the blind source separation problem in which the objective is to separate signals from an observation [17]. The second option is to use the tensor decomposition to impose a certain structure into a problem, for example by constraining the weights of an optimization problem to be a tensor decomposition. In this case the full tensor that the tensor decomposition represents is unknown. The latter usage of tensor decompositions is used in this thesis.

In this thesis the CPD in particular will be used, since it is the simplest decomposition of the three. The operations on the CPD can easily be expressed with linear algebra. As a result, the required components needed for all-at-once optimization that will be discussed later in this thesis can be derived easily.

2-2 Canonical Polyadic Decomposition

In this section the Canonical Polyadic Decomposition (CPD) will be introduced. First some preliminaries will be presented and then several basic CPD operations are introduced. Lastly, a CPD can under certain assumptions be represented as a tensor. It is briefly discussed how this can be done and why this is beneficial.

2-2-1 Preliminaries

The CPD has been introduced by separate authors under different names, such as PARAFAC [18] or CANDECOMP [19]. See Table 3-1 in [10] for a more complete overview of the different names of the CPD. A CPD represents a tensor that is factorized as a sum of R rank-one tensors, where R is the so-called CP-rank. A CPD with CP-rank R that represents a D-way tensor $\mathcal{A} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_D}$ is defined as follows:

$$\mathcal{A} = \sum_{r=1}^{R} \mathbf{a}_r^{(1)} \circ \mathbf{a}_r^{(2)} \circ \ldots \circ \mathbf{a}_r^{(D)}. \tag{2-20}$$

The vectors $\mathbf{a}_r^{(d)} \in \mathbb{R}^{I_d}$ can be stacked into matrices, $\mathbf{A}^{(d)} = [\mathbf{a}_1^{(d)} \ \mathbf{a}_2^{(d)} \ \ldots \ \mathbf{a}_R^{(d)}]$. The matrices $\mathbf{A}^{(d)} \in \mathbb{R}^{I_d \times R}$, $d = 1, \ldots, D$, are called the factor matrices of the CPD. The Kruskal operator $[\![\cdot]\!]$ [20] can be used to represent a CPD:

$$\mathcal{A} = [\![\mathbf{A}^{(1)}, \ \mathbf{A}^{(2)}, \ \ldots, \mathbf{A}^{(D)}]\!] = \sum_{r=1}^{R} \mathbf{a}_r^{(1)} \circ \mathbf{a}_r^{(2)} \circ \ldots \circ \mathbf{a}_r^{(D)}. \tag{2-21}$$

In the definition of a CPD given in Eq. (2-20), the CPD is given as the exact representation of $\mathcal{A}$. However, a CPD is also often used to approximate a known tensor $\mathcal{T}$, such that $\mathcal{T} \approx [\![\mathbf{A}^{(1)}, \ \mathbf{A}^{(2)}, \ \ldots, \mathbf{A}^{(D)}]\!]$. It is in general an approximation in that case, since it might not be straightforward to find a CPD that is exactly equal to $\mathcal{T}$. Numerous algorithms can be found in the literature for fitting a CPD to a known tensor. An extensive overview of algorithms is given in [21], but this was published in 2017 and several new methods have been proposed since.

It is often desired to limit R, such that the CPD becomes a low-rank approximation of the tensor. One of the important advantages of using a low-rank CPD, is that the number of elements that are needed to represent the tensor decreases. Assuming $I_1 = I_2 = \ldots = I_D = I$, the tensor $\mathcal{T}$ has I^D elements, while the CPD $[\![\mathbf{A}^{(1)}, \ \mathbf{A}^{(2)}, \ \ldots, \mathbf{A}^{(D)}]\!]$ only has IRD elements.

A challenge of the CPD are the scaling and permutation indeterminacy [10]. A rank-one tensor that is given by $\mathbf{a}^{(1)} \circ \mathbf{a}^{(2)} \circ \ldots \circ \mathbf{a}^{(D)}$ can also be represented by $\mathbf{b}^{(1)} \circ \mathbf{b}^{(2)} \circ \ldots \circ \mathbf{b}^{(D)}$ where $\mathbf{b}^{(d)} = \rho_d \mathbf{a}^{(d)}$ for any $\rho_i \quad i = 1, \ldots, D$ given that $\prod_{i=1}^{D} \rho_i = 1$. A CPD consists of a sum of rank-one tensors and thus a CPD also has this scaling indeterminacy problem. Additionally, the rank-one tensors of the CPD can be summed in one order or any other order. As a result, multiple combinations of factor matrices exists that all represent that same CPD, but where the factors matrices have a different ordering of their columns. This is a permutation indeterminacy. This is a challenge, because this implies that there is no truly unique solution when fitting a CPD in an optimization problem. In [10] it is mentioned that there are uniqueness conditions for a CPD, but here unique is defined as up to scaling and permutation.

To overcome the scaling indeterminacy a CPD can be normalized. In a normalized CPD the vectors of the rank-one tensor are normalized to be unit vectors. The norm is contained in a separate vector $\boldsymbol{\mu} \in \mathbb{R}^R$. Hence, a normalized CPD is given by

$$\mathcal{A} = \sum_{r=1}^{R} \mu_r \hat{\mathbf{a}}_r^{(1)} \circ \hat{\mathbf{a}}_r^{(2)} \circ \ldots \circ \hat{\mathbf{a}}_r^{(D)}, \tag{2-22}$$

where $\hat{\mathbf{a}}_r^{(d)}$ is normalized vector. In the literature $\boldsymbol{\lambda}$ is often used to represent the norm, but this symbol will be used later in this thesis for other purposes. Therefore, $\boldsymbol{\mu}$ is used here. Since the norm is represented separately and the vector are normalized, the normalized CPD does not suffer from the scaling indeterminacy anymore.

In the following section several basic operations involving a CPD will be presented. In that section and following chapters the not-normalized CPD will be used. The not-normalized CPD is used, since it is the simplest representation of a CPD and will therefore serve as the starting point in this thesis. In Section 5-3 the effect of normalization is studied in more detail. All the expressions that are derived in this thesis can easily be adapted to apply to a normalized CPD.

2-2-2 Basic operations

Due to the special structure of the CPD, several of the tensor operations defined in Section 2-1-2 can be computed more efficiently for a CPD.

First, the vectorization of a CPD can be formulated as a sequence of Khatri-Rao products [14]:

$$\operatorname{vec}(\mathcal{A}) = \left(\mathbf{A}^{(1)} \odot \cdots \odot \mathbf{A}^{(D)}\right) \mathbf{1}_R. \tag{2-23}$$

Here $\mathbf{1}_R$ is a vector of ones of size R. Note that this is a row-major vectorization and may thus differ from the CPD vectorization used in other literature.

With the mixed-product property given in (2-8) that relates the Khatri-Rao and Hadamard product, the squared Frobenius norm of a CPD $\mathcal{A}$ can be rewritten as follows:

$$\begin{aligned} \|\mathcal{A}\|_F^2 &= \langle \mathcal{A}, \mathcal{A} \rangle_F = \operatorname{vec}(\mathcal{A})^\top \operatorname{vec}(\mathcal{A}) \\ &= \mathbf{1}_R^\top \left(\mathbf{A}^{(1)} \odot \cdots \odot \mathbf{A}^{(D)}\right)^\top \left(\mathbf{A}^{(1)} \odot \cdots \odot \mathbf{A}^{(D)}\right) \mathbf{1}_R \qquad (2\text{-}24) \\ &= \mathbf{1}_R^\top \left(\mathbf{A}^{(1)\top}\mathbf{A}^{(1)} * \mathbf{A}^{(2)\top}\mathbf{A}^{(2)} * \cdots * \mathbf{A}^{(D)\top}\mathbf{A}^{(D)}\right) \mathbf{1}_R. \qquad (2\text{-}25) \end{aligned}$$

This formulation can reduce the memory complexity the computation of the inner product. The sequence of Khatri-Rao products in (2-24) can result in a very large matrix. Assuming that all the dimensions of the tensor are of size I and the CPD has rank R, $\mathbf{A}^{(d)} \in \mathbb{R}^{I\times R}$, $d = 1, \ldots,\ D$, the Khatri-Rao product sequence would result in a matrix of size $I^D \times R$. As a result, the memory complexity for this computation is of order $\mathcal{O}(I^D R)$. The product $\mathbf{A}^{(1)\top}\mathbf{A}^{(1)} \in \mathbb{R}^{R\times R}$, so the memory complexity of (2-25) is of order $\mathcal{O}(R^2)$. Thus, the memory complexity does not scale exponentially in D anymore.

Lastly, since a rank-one tensor is simply a CPD with a CP-rank of one, it is possible to compute the Frobenius inner product between a rank-one tensor and a CPD in a similar fashion. Given a CPD $\mathcal{A} = [\![\mathbf{A}^{(1)},\ \mathbf{A}^{(2)},\ \ldots, \mathbf{A}^{(D)}]\!]$ and a rank-one tensor $\mathcal{B} = \mathbf{b}^{(1)} \circ \mathbf{b}^{(2)} \circ \cdots \circ \mathbf{b}^{(D)}$ the Frobenius inner product can be computed as follows:

$$\langle \mathcal{A}, \mathcal{B} \rangle_F = \left(\mathbf{b}^{(1)\top}\mathbf{A}^{(1)} * \mathbf{b}^{(2)\top}\mathbf{A}^{(2)} * \cdots * \mathbf{b}^{(D)\top}\mathbf{A}^{(D)}\right)\mathbf{1}_R. \tag{2-26}$$

Compared to naively fully constructing both tensors, this formulation again reduces the complexity such that it does not scale exponentially anymore.

2-2-3 CPD represented as tensor

A straightforward way to represent a CPD as a variable in a software program is to have a list of the factor matrices $\mathbf{A}^{(d)}$. To perform computations with the CPD the program would then needs to loop over this list to access the individual factors. Alternatively, with the assumption that each dimension of the tensor has the same size, $I_1 = I_2 = \ldots = I_D = I$, the CPD can also be represented as a tensor by stacking the factor matrices. This results in a tensor of size $D \times I \times R$. This has several computational benefits.

However, the needed assumption is a strong assumption. As will be discussed in Section 3-1-2, it is a valid assumption in many circumstances for learning with a CPD. Nevertheless, it must be noted that all the approaches and provided software presented in this thesis could be rewritten without this assumption as well. In other words, the assumption only enables a faster implementation, but does not limit the applied methods. Therefore, all the mathematical formulations and derivations mentioned in this thesis can be applied regardless of this assumption.

In Section C-2 it is discussed in more detail how a CPD can be represented as a tensor in computer memory. Furthermore, it is shown with an experiment that this tensor representation significantly reduces the run time of a CPD computation. Without jitting[2] the run time decreases by more than two orders of magnitude. With jitting the reduction is less drastic but still considerable.

2-3 Summary

In this chapter the notion of tensors was introduced. A tensor is a multidimensional representation of data. A tensor can be decomposed into a tensor decomposition, similar to how

[2]Jitting is discussed in Section 1-3.

a matrix can be decomposed by a matrix decomposition. A tensor decomposition can be applied to reduce the number of elements that are needed to represent the tensor.

The Canonical Polyadic Decomposition (CPD) is an example of such a decomposition. A CPD decomposes a tensor into a sum of R rank-one tensors where R is the CP-rank. A rank-one tensor is a tensor that is formed by the outer product of a series of vectors. A CPD is also commonly denoted by its factor matrices. A factor matrix contains the R vectors of the rank-one tensors for one particular dimension.

Additionally, basic linear algebra operations, tensor operations and CPD operations were introduced. The structure of the CPD allows for efficient operations compared to working with the full tensor version of the CPD.

Finally it was shown how a CPD can be stored as a tensor in computer memory when it is assumed that all the dimensions of the tensor are equal. This can be a valid assumption for the CPD constrained kernel machine that is introduced later. It is shown that this tensors representation can significantly speed up the run time of a CPD function.

Summary of contributions

- It is shown that, under certain assumptions, a CPD can be stored as a tensor in computer memory instead of a list of factor matrices. This can significantly reduce the run time of CPD functions.

Table 2-1: Nomenclature codes.

Notation	Definition
$\mathcal{A}$	Tensor
$\mathbf{A}$	Matrix
$\mathbf{a}$	Vector
$a,\ A$	Scalar
i_d	Index of dimension d of a tensor
I_d	Size of dimension d of a tensor
$\mathcal{A}(i_1, i_2, \ldots, i_D) = a_{i_1,i_2,\ldots,i_D}$	Element of tensor
$\text{vec}(\mathcal{A})$	Vectorization of tensor $\mathcal{A}$
$\circ,\ \tilde{\circ}$	Vector outer product and batch vector outer product
$*,\ \circledast_{d=1}^{D}$	Hadamard product and sequence of Hadamard products
$\otimes,\ \bigotimes_{d=1}^{D}$	Kronecker product and sequence of Kronecker products
$\odot$	Khatri-Rao product
$\langle \cdot, \cdot \rangle_F$	Frobenius inner product
$\|\cdot\|_F$	Frobenius norm
$\mathbf{I}_M$	Identity matrix of size $M \times M$
$\mathbf{1}_M$	Vector of ones of length M
$\times_d$	Mode-d vector product
$\mathcal{O}(\cdot)$	Big O notation

Chapter 3

Kernel machines with CPD

In this chapter the concept of supervised kernel machines is introduced. Kernel machines are a general class of machine learning models. First, supervised kernel machines in general will be introduced as well as the Kernel Ridge Regression (KRR) problem. The KRR model forms the basis for the optimization problem that is studied in this thesis. The optimization problem is solved in the primal space in this thesis, such that the complexity of the problem scales favorably in the number of samples. Product feature maps are introduced, since they are used to transform the input data. However, they result in a large number of weights. A Canonical Polyadic Decomposition (CPD) is used to reduce the number of weights, resulting in the CPD constrained kernel machine. This is the optimization problem that this thesis studies. The CPD constrained optimization problem is a nonlinear least squares (NLS) problem, so NLS optimization in general is introduced. Additionally, challenges of the CPD constrained problem are discussed. The Alternating Least Squares (ALS) method is introduced, because it is commonly used for CPD optimization problems. Finally, an overview of state-of-the-art learning with CPDs is given.

3-1 Kernel machines

Kernel machines are a class of machine learning algorithms that are based on kernels. Examples of kernel machines are the Support Vector Machine (SVM) [4] and Kernel Ridge Regression (KRR) [22] which are supervised learning algorithms, and kernelized Principal Component Analysis [23], an unsupervised machine learning algorithm. Kernel machines have been used for a variety of applications, for example natural language processing [2] and the annotation of DNA sequences [3].

First, the general setting of supervised kernel machines is introduced. Secondly, product feature maps, a particular kind of feature map, are discussed. Moreover, the specific product feature map that is used in this thesis is introduced. For this overview [24] and [6] were used.

3-1-1 Optimization problem

A general class of models can be described by the following equation:

$$y = \langle \mathbf{w}, \mathbf{z}(\mathbf{x}) \rangle + \varepsilon, \tag{3-1}$$

where $\mathbf{x} \in \mathbb{R}^D$ is the input with D features and y is the recorded output of the model which disturbed by zero mean i.i.d. noise ε. The input is mapped to a feature space by the feature map $\mathbf{z} : \mathbb{R}^D \to \bar{M}$. The model weights are contained in the vector $\mathbf{w} \in \bar{M}$. The inner product $\langle \mathbf{w}, \mathbf{z}(\mathbf{x}) \rangle$ is the prediction of the model; $y_{pred} = \langle \mathbf{w}, \mathbf{z}(\mathbf{x}) \rangle$.

The goal is to learn the weights such that the model predictions match the recorded outputs as closely as possible. The learning is done with a training data set $\{\mathbf{x}_n, y_n\}_{n=1}^N$ containing N samples. The weights can be learned from the training data by solving an optimization problem for $\mathbf{w}$:

$$\min_{\mathbf{w}} \quad \sum_{n=1}^{N} L(y_n, \langle \mathbf{w}, \mathbf{z}(\mathbf{x}_n) \rangle) + \lambda \langle \mathbf{w}, \mathbf{w} \rangle_p . \tag{3-2}$$

Here $L : \mathbb{R} \times \mathbb{R} \to \mathbb{R}_+$ is a loss function and $\langle \cdot, \cdot \rangle_p$ is a regularization term that is scaled with the hyperparameter λ. Different choices for the loss function and regularization term lead to different machine learning problems. For example, when hinge loss is used this corresponds to a SVM. For Kernel Ridge Regression (KRR) a square loss function and regularization term are used and the optimization problem becomes:

$$\min_{\mathbf{w}} \quad \sum_{n=1}^{N} (y_n - \langle \mathbf{w}, \mathbf{z}(\mathbf{x}_n) \rangle)^2 + \lambda \|\mathbf{w}\|_2^2 . \tag{3-3}$$

This problem is a so-called linear least squares (LLS) problem which is a convex quadratic program (QP) in the variable $\mathbf{w}$. The LLS problem has a global optimal solution. Furthermore, a closed-from expression for this solution exists called the normal equations. The computational complexity for computing the solution scales with the size of $\mathbf{w}$. To be more precise, the complexity is $\mathcal{O}(\bar{M}^3)$ where $\bar{M}$ is the size of $\mathbf{w}$. In this thesis the KRR problem is studied specifically.

Although Eq. (3-3) is formulated as a regression problem, it is also possible to solve a binary classification task with the same optimization problem. In a classification task the goal is to predict a discrete label instead of a continuous value. It is shown in [25] that when the class labels are set to -1 and 1, $y_n \in [-1,\ 1]$, a binary classification problem can be solved with the KRR problem. After the optimization problem is solved, the model classification response can be evaluated by taking the sign of the model prediction.

A solution of the kernel machine optimization problem can be computed in multiple ways. Which method is more advantageous depends on the training data. Namely, the number of samples and the number of features of the training data determine whether a primal or dual space approach is be preferred. Commonly, kernel machines are solved in the dual space with the so-called kernel trick [4]. However, the computational and memory complexity of the kernel trick scale $\mathcal{O}(N^3)$ and $\mathcal{O}(N^2)$ respectively [26, 27]. As a result, this method is not suitable for data sets with a large number of samples. For a detailed overview of the dual space approach and the kernel trick, see [24, Ch. 2].

Since this thesis focuses on data with a large number of samples, the problem is solved in the primal space. In the primal space, the complexity of solving the optimization problem scales linearly in the number of samples. A downside of the primal space is that the complexity can scale unfavorably with the number of features. This is deliberated in more detail later. So, this thesis studies methods to find a solution for the primal optimization problem given in Eq. (3-3).

3-1-2 Product feature maps

To solve the primal optimization problem the data needs to be transformed with feature map $\mathbf{z}$. In this thesis the product feature map, or multidimensional multiplicative feature map, is used which is specific kind of feature map. A product feature map can be efficiently combined with the CPD structure. First, the product feature map in general will be introduced. Then the application of product feature maps to kernel machines will be discussed as well as the mapping that is used in this thesis.

Product feature map

As described in [28], when a feature map $\tilde{\mathbf{z}} : \mathbb{R}^D \rightarrow \mathbb{R}^{M_1 M_2 \cdots M_D}$ corresponds to a multidimensional kernel it can be written as a combination of scalar kernels:

$$\tilde{\mathbf{z}}(\mathbf{x}) = \bigotimes_{d=1}^{D} \mathbf{z}_d(x^{(d)}). \tag{3-4}$$

Here $x^{(d)}$ is the dth feature of $\mathbf{x}$ and $\mathbf{z}_d : \mathbb{R} \rightarrow \mathbb{R}^{M_d}$ is a scalar feature map for the dth. The dimension M_d is referred to as the feature map order. A multidimensional kernel in this case means that the input of the kernel is multidimensional, so $\mathbf{x}$ is a vector. As mentioned in Eq. (2-10), there is a relationship between the Kronecker product and the vector outer product. So, the feature map can alternatively be formulated as the outer product of the vectors $\mathbf{z}_d(x^{(d)})$. The result is a D-way tensor instead of a vector. The feature map is then given by

$$\mathcal{Z}(\mathbf{x}) = \mathbf{z}_1(x^{(1)}) \circ \mathbf{z}_2(x^{(2)}) \circ \cdots \circ \mathbf{z}_D(x^{(D)}), \tag{3-5}$$

where $\mathcal{Z} : \mathbb{R}^D \rightarrow \mathbb{R}^{M_1 \times M_2 \times \cdots \times M_D}$ results in a D-way tensor. This product feature map formulation is, for instance, used in [6] and [29]. Since $\text{vec}(\mathcal{Z}(\mathbf{x})) = \tilde{\mathbf{z}}(\mathbf{x})$, it can be shown that $\langle \tilde{\mathbf{z}}(\mathbf{x}), \tilde{\mathbf{z}}(\mathbf{x}') \rangle = \langle \mathcal{Z}(\mathbf{x}), \mathcal{Z}(\mathbf{x}') \rangle_F \ \forall \mathbf{x},\ \mathbf{x}' \in \mathbb{R}^D$. Therefore, both formulations of the feature map can be used. From the structure of $\mathcal{Z}(\mathbf{x})$ it can be concluded that it is a rank-one tensor. This structure is useful, because it can effectively be combined with a CPD as demonstrated, for example, in Eq. (2-26). Therefore, the rank-one tensor version of the product feature map will be used.

For this thesis it will be assumed that the individual feature maps $\mathbf{z}_d$ in the product feature map are of the same order, so $M_1 = M_2 = \ldots = M_D = M$. This allows for the tensor representation of the CPD to be used. Furthermore, the transformed data $\mathcal{Z}(\mathbf{X})$ can then be stored as a tensor as well. This will be discussed later. So, this assumption results in efficient implementations. However, it is important to note that any derived expression in this thesis can also be implemented without this assumption.

Moreover, for convenience of notation it is assumed that all the feature maps are the same, $\mathbf{z}_d = \mathbf{z} : \mathbb{R} \to \mathbb{R}^M,\ d = 1, \ldots,\ D$. Therefore, the subscripts will be dropped.

The feature map can also be computed in batches. This is useful, since in general multiple samples are used. So, it can be beneficial to compute the feature map for all the samples at once. For a batch of N samples the batch feature map is defined as follows:

$$\begin{aligned} &\mathbf{X} \in \mathbb{R}^{N\times D},\ \mathbf{X}^{(d)} \in \mathbb{R}^N, \\ &\mathbf{Z}(\mathbf{X}^{(d)}) = [\mathbf{z}(x_1^{(d)})\ \mathbf{z}(x_2^{(d)}) \ldots \mathbf{z}(x_N^{(d)})]\ \in \mathbb{R}^{M\times N}. \end{aligned} \tag{3-6}$$

Finally, the batch feature map given in Eq. (3-6) can be stored as a 3-way tensor similarly to how the CPD can be stored as a tensor. The sub-mappings $\mathbf{Z}(\mathbf{X}^{(d)})$ are stacked along the first axis to create the tensor $\mathcal{Z}(\mathbf{X})$:

$$\mathcal{Z}(\mathbf{X})_{d,:,:} = \mathbf{Z}(\mathbf{X}^{(d)}),\ d = 1, \ldots,\ D, \quad \mathcal{Z}(\mathbf{X})\ \in \mathbb{R}^{D\times M\times N}. \tag{3-7}$$

It is beneficial to use this representation, because it can be efficiently combined with the tensor representation of a CPD. This will prove to be useful for the computation of the all-at-once gradient.

Product feature maps in kernel machines

Several feature maps can be expressed as product feature maps. In [30] and [31] polynomial functions are expressed by a product feature map and [32] multivariate B-splines are expressed this way. In [26] it is shown that a product feature map can be used to approximate the radial basis function (RBF) kernel.

In this thesis the product feature map that approximates the RBF kernel is used in particular. It is called the Fourier feature map. For input $\mathbf{x} \in \mathbb{R}^D$ that is centered in the hyper box $[-U_1,\ U_1] \otimes \cdots \otimes [-U_D,\ U_D]$, the mth element of $\mathbf{z}(x^{(d)})$ is given by [26, 6]

$$\left(\mathbf{z}(x^{(d)})\right)_m = \frac{1}{\sqrt{U_d}}\sqrt{S}\sin\left(\frac{\pi m(x^{(d)} + U_d)}{2U_d}\right), \quad S = \sqrt{2\pi} \cdot l \cdot \exp\left(\frac{-\pi^2 m^2 l^2}{8U_d^2}\right), \tag{3-8}$$

where l is the length-scale of the RBF kernel that is approximated. In Section D-1 the approximation of the kernel is discussed in more detail. An increase in the feature map order M improves the approximation exponentially [26, Theorem 8]. Therefore, it can be beneficial to use large values for M. In this thesis values for M ranging from 10 to 40 will be mostly used, since similar values were used in [6]. The maximum magnitude of the output of this feature map is less than one when $U_d = 1$. Furthermore, the output is in general considerably smaller. This is discussed in more detail in Section D-2. The fact that the outputs are smaller than one is important in Section 5-1-1. The assumption that $U_d = 1$ implies that the input lies in the interval $[-1,\ 1]$ which is commonly the case in machine learning due to preprocessing of the input data.

Although this Fourier feature map is used, the aim of this thesis is to compare to optimization methods that make use of the same transformed data. It is thus assumed that the choice of feature map does not influence the comparison. There is no specific reason why this particular feature map was used other than that the RBF kernel is often used in kernel machines [24]. Additionally, the precise value of the length-scale is also assumed not to be of importance due to this. Moreover, any other product feature map can be used with the methods that are derived throughout this thesis.

Since the product feature map results in a tensor, the weights also become a tensor. The new prediction model is

$$y_{pred} = \langle \mathcal{W}, \mathcal{Z}(\mathbf{x}) \rangle_F \,, \tag{3-9}$$

and the optimization problem of Eq. (3-3) becomes

$$\min_{\mathcal{W}} \quad \sum_{n=1}^{N} \left(y_n - \langle \mathcal{W}, \mathcal{Z}(\mathbf{x}_n) \rangle_F\right)^2 + \lambda \langle \mathcal{W}, \mathcal{W} \rangle_F \,. \tag{3-10}$$

The resulting problem can be solved by vectorizing $\mathcal{W}$ and $\mathcal{Z}(\mathbf{x}_n)$. This would result in the standard KRR form given in Eq. (3-3). So, Eq. (3-10) is still a LLS problem for which, theoretically, the global optimal solution can be found with the normal equations. However, the weight tensor $\mathcal{W}$ is a D-way tensor that has M^D elements and thus there are M^D weights. As a result, the complexity of the optimization problem scales exponentially with the number of features D. It is only beneficial to use this approach when $M^D \ll N$, because otherwise it would be more efficient to solve the problem in the dual space using the kernel trick. Therefore, the use of Eq. (3-10) is limited to data with a small number of features. However, it is possible to reduce the number of weights in Eq. (3-10) by constraining $\mathcal{W}$ to be a CPD.

3-2 CPD constrained kernel machine

As mentioned in Section 2-2, when a CPD is used to represent a tensor, the number of elements that is needed to represent the tensor decreases. Consequently, when the tensor contains weights, the number of weights can be reduced by constraining the tensor to be a CPD. When this is applied to Eq. (3-10) it results in the following optimization problem:

$$\begin{aligned} \min_{\mathcal{W}} \quad & f(\mathcal{W}) = \frac{1}{N} \sum_{n=1}^{N} \left(y_n - \langle \mathcal{W}, \mathcal{Z}(\mathbf{x}_n) \rangle_F\right)^2 + \lambda \langle \mathcal{W}, \mathcal{W} \rangle_F \\ \text{s.t.} \quad & \mathcal{W} = [\![\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(D)}]\!]. \end{aligned} \tag{3-11}$$

Although the optimization problem includes a constraint, this constraint can be substituted into the loss function and thereby removed. Furthermore, it is possible to use the rank-one structure of the feature map $\mathcal{Z}(\mathbf{x}_n)$ combined with the CPD structure of the weights to write the Frobenius inner product as in Eq. (2-26). Finally, the Frobenius inner product of the CPD with itself can be written as in Eq. (2-25). This results in the following new formulation of the loss function:

$$\begin{aligned} f(\mathcal{W}) &= \frac{1}{N} \sum_{n=1}^{N} \left(y_n - \left(\mathbf{z}(x_n^{(1)})^\top \mathbf{W}^{(1)} * \mathbf{z}(x_n^{(2)})^\top \mathbf{W}^{(2)} * \dots * \mathbf{z}(x_n^{(D)})^\top \mathbf{W}^{(D)} \right) \mathbf{1}_R \right)^2 + \\ &\quad + \lambda \mathbf{1}_R^\top \left(\mathbf{W}^{(1)\top} \mathbf{W}^{(1)} * \mathbf{W}^{(2)\top} \mathbf{W}^{(2)} * \dots * \mathbf{W}^{(D)\top} \mathbf{W}^{(D)} \right) \mathbf{1}_R \\ &= \frac{1}{N} \sum_{n=1}^{N} \left(y_n - \left(\mathop{\circledast}_{d=1}^{D} \mathbf{z}(x_n^{(d)})^\top \mathbf{W}^{(d)} \right) \mathbf{1}_R \right)^2 + \lambda \mathbf{1}_R^\top \left(\mathop{\circledast}_{d=1}^{D} \mathbf{W}^{(d)\top} \mathbf{W}^{(d)} \right) \mathbf{1}_R. \end{aligned} \tag{3-12}$$

Note that the formulation of $f(\mathcal{W})$ differs from the original KRR problem presented in Eq. (3-3), due to the $\frac{1}{N}$ coefficient. Instead of the summed squared error, the mean squared error (MSE) is used here. This is a common loss function within machine learning. The MSE is convenient compared to the summed squared error, because it implies that the balance between the error term and the regularization term is independent of the number of samples.

As mentioned in the previous section, the unconstrained weight tensor $\mathcal{W}$ has M^D elements which makes it infeasible for learning for even small values of M and especially D. The CPD constrained weight tensor has only DMR elements where R is the CP-rank of the CPD. The number of elements now scales linearly with the order of the feature map M and the number of features in the data D. So, larger feature maps and data with more features can be used. Thus, the CPD constrained kernel machine is better suited for large scale learning.

A downside of the CPD constraint is the added complexity. A CPD is the product of its factor matrices. As a result, the CPD constraint introduces nonlinearity into the optimization problem. Therefore, the problem is no longer a linear least squares problem, but has become a NLS problem.

In the rest of this thesis the CPD constrained problem will solely be used, so the constraint will not be specifically mentioned anymore. Furthermore, $\mathcal{W}$ will refer to CPD weight tensor.

3-2-1 Nonlinear least squares optimization

The nonlinear least squares optimization problem is particular nonlinear problem in which the goal is to find the weights such that a squared residual error is minimized. In the context of data fitting such as supervised machine learning, the residual error is the error between the recorded output y_n and a model prediction $m(\mathbf{w}, \mathbf{x}_n)$. The model is a nonlinear function of the model weights $\mathbf{w}$ and the input $\mathbf{x}_n$. The residual for sample $\{\mathbf{x}_n, y_n\}$ is thus given by $r_n = y_n - m(\mathbf{w}, \mathbf{x}_n)$. In the context of the CPD kernel machine the model is thus $\langle \mathcal{W}, \mathcal{Z}(\mathbf{x}_n) \rangle_F$.

While for the LLS problem a closed-form solution exists, this is not the case for the NLS problem. The problem is therefore solved iteratively, with so-called iterative optimization. For a general overview of iterative optimization see [33, 34]. In iterative optimization a solution is found by updating the weights at each iteration. It basically consists of three parts:

- **Weights initialization:** At the start of the optimization initial values for all the weights need to be selected.
- **Step direction:** At each iteration the direction in which to update the weights needs to be computed.
- **Step size:** At each iteration it needs to be determined how much to update the weights for the computed step direction.

Although step direction and step size are listed here as two separate parts of the weight update, they can be linked to each other as well.

Overall, there are two options for updating the weights. Either a subsection of the weight is updated, or all the weights are updated at once. Updating a subsection of the weights can be done with ALS which has already been applied for a CPD in the literature. Alternatively, all factors of the CPD can be updated at once with so-called all-at-once optimization. In this thesis it is studied whether all-at-once optimization can be applied to a CPD constrained kernel machine and when this is more advantageous to use compared to ALS.

In the next section challenges of the CPD constrained kernel machine in general will be discussed first. Then the ALS method will be introduced as well as a literature overview of learning with CPDs. In the next chapter all-at-once CPD learning will be studied.

3-2-2 Challenges of the CPD constrained kernel machine

The most obvious challenge of the CPD constrained kernel machine is the NLS loss function that is in general not convex. Besides the complexity of finding a solution, the validity of a solution is also a challenge.

The not-convex optimization problem can have numerous local minima and thus a found solution is not necessarily the global minimum. Several different weight initializations can be tried to find different minima, but it cannot be guaranteed that a global optimum has been found. Moreover, the scaling and permutation indeterminacy of the CPD that were discussed in Section 2-2-1 complicate this further. Due to these indeterminacies there is an infinite number of CPDs that correspond to the same solution of the optimization problem. Consequently, there is infinite number of minima of the optimization problem.

The abundance of local minima has several implications. First, the weight initialization becomes more crucial due to this, because it can affect which local minimum is found. Several initializations might have to be tried to investigate whether a different initialization leads to a more optimal minimum. Secondly, the indeterminacies make it challenging to compare two CPDs with each other. The factor matrices cannot be compared directly. The difference between two CPDs can be analyzed, but this does not reveal any information about the individual factor matrices. Comparing two solutions could be useful to analyze whether they correspond to the same minimum with the indeterminacies taken into account or whether they correspond to different minima. Furthermore, comparing factor matrices might reveal which features are more important for the model than others. In this thesis the solutions were not compared, but this could be studied in future research.

Additionally, the CPD constrained kernel machine has several parameters that influence the complexity of the optimization, but are important for the model. Most significantly, the feature map order M and the CP-rank R. The feature map order influences the complexity of the optimization problem negatively, but affects the kernel approximation accuracy positively. As mentioned earlier, this is a trade-off that needs to be taken into account. The CP-rank determines the expressiveness of the model. A larger CP-rank allows for a more complicated model. On the other hand, it does increase the complexity of the optimization problem.

Furthermore, a large CP-rank might lead to so-called swamps or bottlenecks [35, 36]. Swamps are defined as periods in the optimization process where convergence is very slow, followed by a period where convergence is fast again. This is undesirable behaviour, since it leads to uncertainty whether convergence is slow due to a near local optimum or due to a swamp. A swamp can occur when two rank-one tensors of the CPD cancel each other out [37]. As a result, the values of these terms can be updated at each iteration, but as long as they cancel each other out no improvement is made. Swamps have been identified for problems of fitting a CPD to a known tensor. No literature was found for swamps in relation to CPD learning. Nevertheless, the phenomenon might occur in this context as well. It is reasoned that when the CP-rank is larger than necessary, it is more likely to occur. In that case not all rank-one terms are needed to obtain the required expressiveness for a particular learning problem. As a result, the 'left over' terms can start cancelling other out. So, the selection of a suitable value for R is a trade-off between the expressiveness of the model, the avoidance of swamps and the complexity of the optimization problem.

3-3 Alternating least squares

In this section Alternating Least Squares (ALS) in general will be discussed as well as ALS applied to the CPD kernel machine. Optimization using ALS is of interest, since it is used extensively in the literature for learning with CPDs.

3-3-1 Alternating least squares optimization

ALS is a form of block coordinate descent and cyclic coordinate minimization [38]. It can be used to solve NLS problems. The general idea behind coordinate descent is to minimize a loss function with respect to one weight while keeping the others constant. Next, a different weight is chosen to be optimized and the rest is held constant and this is repeated for all variables [39]. For block coordinate descent not a single weight, but a block of weights is optimized while the rest remains constant.

ALS is block coordinate descent approach that is applied to a NLS loss function. Particularly, the loss function is nonlinear when all the weights are considered, but it is linear when a block of coordinates, or weights, is optimized and then rest is held constant. As a result, a linear least squares subproblem arises, hence the name Alternating Least Squares. It is important that the block of coordinates is chosen in such a way that linearity of subproblem holds. Due to the linearity, it is possible to efficiently compute an solution for the subproblem and thus to update the block of coordinates.

3-3-2 CPD ALS

The ALS method can easily be applied to optimize a CPD by using the factors of the CPD as blocks of coordinates. Therefore, it can be used for finding a solution to the CPD constrained kernel machine. For the optimization problem given in Eq. (3-11) the ALS subproblem can be formulated as

$$\min_{\mathbf{W}^{(d)}} \quad \frac{1}{N}\sum_{n=1}^{N}\left(y_n - \left(\mathop{\circledast}_{d=1}^{D} \mathbf{z}(x_n^{(d)})^\top \mathbf{W}^{(d)}\right)\mathbf{1}_R\right)^2 + \lambda \mathbf{1}_R^\top \left(\mathop{\circledast}_{d=1}^{D} \mathbf{W}^{(d)\top}\mathbf{W}^{(d)}\right)\mathbf{1}_R, \tag{3-13}$$

where $\mathbf{W}^{(d)}$ is one of the factor matrices of the CPD. In [6] it is shown that the loss function can be rewritten such that it becomes a LLS problem in vec($\mathbf{W}^{(d)}$):

$$
\begin{aligned}
\min_{\operatorname{vec}(\mathbf{W}^{(d)})} \quad & \frac{1}{N}\sum_{n=1}^{N}\left(y_n - \left\langle \operatorname{vec}(\mathbf{W}^{(d)}), \mathbf{z}(x_n^{(d)}) \otimes \left(\mathop{\circledast}_{p=1,p\neq d}^{D} \mathbf{z}(x_n^{(p)})^\top \mathbf{W}^{(p)} \right) \right\rangle \right)^2 \\
& + \lambda \left\langle \operatorname{vec}(\mathbf{W}^{(d)\top}\mathbf{W}^{(d)}), \operatorname{vec}(\left(\mathop{\circledast}_{p=1,p\neq d}^{D} \mathbf{W}^{(p)\top}\mathbf{W}^{(p)} \right)) \right\rangle .
\end{aligned} \tag{3-14}
$$

Note that this is a slightly different formulation than used in [6], since the MSE is used here instead of the summed squared error as in [6].

The factor matrix $\mathbf{W}^{(d)}$ can be updated by solving the LLS problem, for example with the normal equations. By iteratively updating the factor matrices using their respective subproblem the full optimization problem can be solved. The ALS algorithm based on [6] is presented in Algorithm A.1 and Algorithm A.2.

In Section 3-2-1 it was mentioned that the determination of the step direction and the step size are two important aspects within iterative optimization. With ALS these two are combined and consequently the step size does not need to be determined explicitly. In Section 4-4-2 it will be discussed in more detail why this can be beneficial.

Finally, the ALS method essentially solves the main optimization problem by solving a series of smaller optimization problems. As a result, ALS is monotonically decreasing [6]. Due to the nature of the method the training loss cannot increase during the optimization. Since each weight update is an optimization itself, it is not possible that the training loss increases during a weight update. As a result, the training loss decreases during training until the difference in loss becomes so small that the training has practically converged. This does not necessarily hold for any validation loss, because overfitting might occur.

3-4 State-of-the-art for learning with CPDs

In this section an literature overview will be given of learning with CPDs. In general there are two aspects of the CPD constrained kernel machine that are of interest. Firstly, the representation of weights as a CPD and, secondly, the use of a product feature map that has a rank-one tensor structure. In the literature the combination of both has been used as well as only the first aspect.

In [7] a CPD decomposition is used for the weights of a multidimensional Fourier series that is represented by a rank-one tensor. In [6] a CPD is used for the weights of a KRR problem in combination with a product feature map that approximates a RBF kernel. In [40] the weights of a multivariate polynomial are represented as a symmetric CPD. The symmetric structure of the CPD combined with the rank-one and symmetrical structure of their polynomial feature map are exploited for efficient learning. The approach is used in [41] for the identification of nonlinear Volterra systems. In [42] an optimization method is described for solving a linear system of equations, $\mathbf{Ax} = \mathbf{b}$ in a least-squares sense, where $\mathbf{x}$ is constrained to be a vectorized CPD. It could be adapted to be applicable to Eq. (3-11) when the regularization term is omitted. However, the proposed method does not take the special rank-one structure of the transformed data into account.

Two of the mentioned papers use ALS or another method that updates one factor at a time [6, 7]. However, others apply all-at-once optimization [42, 40]. The optimization method described in [42] assumes the data has no special structure In [40] a symmetrical rank-one structure of the feature map is taken into account, but the CPD is also limited to be symmetric. This greatly reduces the number of parameters and is thus a less complex problem to solve.

From this overview it can be deduced that there is still a gap for the application of all-at-once optimization for a general CPD in combination with a product feature map.

3-5 Summary

In this chapter the kernel machine was introduced. The kernel machine is a versatile nonlinear machine learning model that has various applications. It is commonly solved in the dual space with the kernel trick. However, the complexity of the kernel trick scales cubically with the number of samples. This makes the kernel trick practically infeasible for large scale learning.

So, the kernel machine is solved in the primal space in this thesis. Product feature maps are used to transform the input data. The rank-one tensor structure of the product feature map can efficiently be combined with the CPD structure. The product feature map has as downside that it increases the number of weights, such that it scales exponentially in the number of features.

To overcome this, the weights are constrained to be a CPD. This results in the CPD constrained kernel machine, the optimization problem that this thesis studies. Specifically, the KRR kernel machine with MSE loss is studied. The number of weights scales linearly in the number of features due to CPD constraint. However, the CPD introduces a nonlinearity into the optimization problem. Hence, nonlinear optimization must be applied.

ALS is a form of nonlinear optimization that is commonly used for CPDs. In ALS one factor of the CPD is optimized at a time by solving a LLS subsystem. Alternatively, all the factors of the CPD can be optimized simultaneously with so-called all-at-once optimization. In this thesis it is studied whether all-at-once optimization can be applied to the CPD constrained kernel machine and how it compares to ALS.

In a literature overview it is shown that the CPD constrained kernel machine has previously been solved with ALS. All-at-once optimization has also been applied to CPD learning, but only under certain assumptions. All-at-once optimization for a general CPD in combination with a product feature map was not found in the literature.

Summary of contributions

- It is shown that, under certain assumptions, a batch feature map can be stored as a tensor instead of a list in computer memory. It can then be combined with the tensor representation of a CPD for efficient computations.

Chapter 4

All-at-once CPD learning

In this chapter the application of all-at-once iterative optimization for the Canonical Polyadic Decomposition (CPD) constrained kernel machine is studied. First, all-at-once optimization in general is discussed. The Steepest Gradient Descent (SteGD) method is introduced as the basis for all-at-once optimization. One key component of this method is the gradient of the Kernel Ridge Regression (KRR) loss function with respect to the CPD. The algorithm for the all-at-once gradient that was derived during this thesis is presented. It is then shown in a proof of concept with experiments that SteGD can be applied for CPD learning. From these initial experiments challenges of all-at-once optimization are identified that will be discussed in the next chapter. Finally, all-at-once optimization is compared to Alternating Least Squares (ALS) based on the method characteristics such as the theoretical complexity and the differences in the weight update process.

4-1 All-at-once optimization

As the name suggests, in the all-at-once iterative optimization scheme all the weights are updated at the same time at each iteration. Given a weight vector $\mathbf{w}_k$ at iteration k, the general weight update rule is given by

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \alpha_k \mathbf{p}_k, \tag{4-1}$$

where α_k is the step size and $\mathbf{p}_k$ is the step direction. When the weights are represented by a CPD the same update rule can still be applied by vectorizing the CPD to create $\mathbf{w}$. Alternatively, each factor matrix $\mathbf{W}^{(d)}$ can be updated by

$$\mathbf{W}_{k+1}^{(d)} = \mathbf{W}_k^{(d)} + \alpha_k \mathbf{P}_k^{(d)}, \quad d = 1, \ldots,\ D, \tag{4-2}$$

where $\mathbf{P}_k^{(d)}$ corresponds to the correct entries in $\mathbf{p}_k$.

Numerous optimization methods are based upon Eq. (4-1). See [43] for a concise overview of well known iterative optimization methods. This thesis focuses on a particular subset of optimization methods where the step direction is the negative gradient of the loss function, so-called gradient descent methods. In [44] an overview is presented of gradient descent methods that are commonly used in machine learning. The simplest method is so-called Steepest Gradient Descent (SteGD). This method is, therefore, used as the first approach for applying all-at-once optimization for the CPD learning problem. The SteGD method will be discussed in the next section, but first the general all-at-once CPD optimization framework is introduced.

4-1-1 All-at-once optimization framework

In short, the all-at-once optimization framework consists initialization the weights and then applying the weight update rule a set number of times. The number of iterations is also referred to as the number of epochs. Alternatively, the training may be stopped earlier with an early stopping criteria. An early stopping criteria measures the convergence of the loss. A criteria could, for example, be the difference in loss between two subsequent iterations. In this thesis early stopping was not used.

A powerful option of the gradient descent methods is to compute the gradient based on a single sample or a batch of samples instead of the full data set. This is called stochastic gradient descent and mini-batch gradient descent respectively [44]. In the literature the name stochastic gradient descent is also used for mini-batch descent, but in this thesis only the term mini-batch gradient descent will be used.

Mini-batch gradient descent is a powerful option, because it enables multiple weight updates before the full data set has been used once. More updates implies that the training can converge faster. Therefore, the mini-batch method is often used in machine learning research [44, 8]. A downside is that a mini-batch gradient might be a bad estimate for the gradient of the whole data set. Hence, a weight update in the wrong direction could be carried out. Especially for small batches this can be the case. On the other hand, a smaller mini-batch results in more updates before all the data has been used. So, the selection of the mini-batch size poses a trade-off. In addition, due to the possibility to step in a sub-optimal direction, a mini-batch descent method has the potential to escape local minimum which could be beneficial. In this thesis both the full- and mini-batch gradient descent method will be studied.

The optimization framework has several hyperparameters. Hyperparameters are parameters of the optimization problem and can thus influence the optimization performance. The hyperparameters can generally be grouped in two classes. There are hyperparameters of the optimization problem and hyperparameters of the optimization method. When comparing different optimization methods, the hyperparameters of the optimization problem should be of less importance, since both methods work with the same problem.

The hyperparameters for the all-at-once optimization method are the step size α, the batch size in case of mini-batch descent and the number of iterations. The ALS method only has the number of iterations as a hyperparameter. The optimization problem has as hyperparameters the CP-rank R and the regularization parameter λ. Additionally, the Fourier feature map of Eq. (3-8) introduces the feature map order M, length-scale l and input bound U as hyperparameters.

The complete CPD all-at-once optimization framework is given in Algorithm 4.1. The number of training samples might not be a multiple of the batch size. Therefore, the training data is shuffled at each iteration and the remainder of the training samples that does not fit in a multiple of the batch size is left out. Due to the shuffling a random portion is left out at each time. When the batch size is chosen equal to the number of samples this corresponds to the full-batch method. Due to this implementation the same number of samples is always used to compute the gradient. This was done because it makes jitted code more efficient compared to using batches of different sizes. In general, any weight update can be used with this framework.

Algorithm 4.1 CPD all-at-once optimization framework for a full- or mini-batch method.

```
function BATCH_TRAINING($\mathbf{X}$, $\mathbf{y}$, $R$, $\lambda$, $\alpha$, $\mathcal{Z}$, $batch\ size$, $number\ of\ iterations$)
    ▻ Parameters
    $D \leftarrow \mathbf{X}$.shape[1]
    $N \leftarrow \mathbf{X}$.shape[0]
    $M \leftarrow \mathcal{Z}$.order() ▻ Get order of feature map $\mathcal{Z}$
    $\mathcal{W} \leftarrow$ initial_weights($D, M, R$)

    $number\ of\ batches$ = int($N/batch\ size$)

    for $i \leftarrow 1, number\ of\ iterations$ do
        ▻ Shuffle data to leave out random part when $N$ is not a multiple of batch size
        $\mathbf{X}_s, \mathbf{y}_s$ = shuffle($\mathbf{X}, \mathbf{y}$)
        for $b \leftarrow 0, number\ of\ batches - 1$ do
            $\mathbf{X}_{batch} \leftarrow \mathbf{X}_s[b \cdot batch\ size : (b+1) \cdot batch\ size, :]$
            $\mathbf{y}_{batch} \leftarrow \mathbf{y}_s[b \cdot batch\ size : (b+1) \cdot batch\ size]$
            $\mathcal{Z}_{batch} \leftarrow \mathcal{Z}(\mathbf{X}_{batch})$
            $\mathcal{W} \leftarrow$ weight_update($\mathcal{W}$, $\mathcal{Z}_{batch}$, $\mathbf{y}_{batch}$, $\lambda$, $\alpha$)
        end for
    end for
    return $\mathcal{W}$
end function
```

4-1-2 Steepest gradient descent

The idea behind the Steepest Gradient Descent (SteGD) method is to take steps in the opposite direction of the gradient of the loss function for the current weights. Namely, this step direction is the steepest descent direction. Given the loss function $f(\mathcal{W})$ the step direction for the current weights $\mathcal{W}_k$ is given by

$$\mathbf{p}_k = -\frac{\partial f(\mathcal{W}_k)}{\partial \operatorname{vec}(\mathcal{W}_k)}. \tag{4-3}$$

For the CPD specifically, the step direction can be computed for each factor matrix as follows:

$$\mathbf{P}_k^{(d)} = -\frac{\partial f(\mathcal{W}_k)}{\partial \mathbf{W}^{(d)}},\ d = 1, \dots,\ D. \tag{4-4}$$

So, to perform weight updates, the gradient of the KRR loss function given in Eq. (3-12) with respect to the factor matrices of the CPD is needed. There are in general three options to obtain a value of a gradient:

- **Analytical:** If an analytical expression for the gradient is available, this can be used.
- **Finite difference:** The gradient can be estimated using finite difference methods.
- **Automatic differentiation:** The value of the gradient can be computed by a software program using automatic differentiation (AD). Note that this is not an estimation.

This thesis focuses on the analytical expression and AD, since these methods provide true values for the gradient and not estimations. Both methods will be discussed in more detail in the next sections.

For the second aspect of the update method, the step size, there is also a variety of options. The step size can be selected up front and held fixed, the step size can be decreased each iteration or the step size can be a function of the weights at the current and previous iterations. In this thesis the SteGD method will correspond to a fixed step size, the most basic step size variant. In Section 5-1 the selection of the step size is analyzed in more detail.

4-2 All-at-once CPD gradient

In this section the gradient of loss function with respect to the CPD weights is studied. During this thesis an analytical expression was derived for the all-at-once gradient. First, this analytical expression is presented. Secondly, automatic differentiation (AD) as an alternative for obtaining a value for the gradient is discussed.

4-2-1 Analytical gradient

During this thesis an analytical expression for the all-at-once gradient was derived. The resulting expressions will be presented here. In Section A-2-1 the derivations are presented in more detail. First, the partial derivative of the loss function with respect to one factor matrix is given. Instead of computing the partial derivative with respect to each factor matrix separately, an algorithm is derived that efficiently reuses intermediate results. This is presented in the second part of this section.

Partial derivative of loss function with respect to one factor

The partial derivative of the loss function with respect to a single factor matrix is defined as

$$\mathbf{G}^{(d)} = \frac{\partial f(\mathcal{W})}{\partial \mathbf{W}^{(d)}} \in \mathbb{R}^{M \times R}, \tag{4-5}$$

and consists of a mean squared error (MSE) part and a regularization part.

For a batch of samples of size N, $\{\mathbf{x}_n, y_n\}_{n=1}^N$ the MSE term is given by

$$\begin{aligned} mse = \frac{1}{N}\sum_{n=1}^{N}(y_n - \langle \mathcal{W}, \mathcal{Z}(\mathbf{x}_n)\rangle_F)^2 &= \frac{1}{N}\sum_{n=1}^{N} r_n^2 \\ &= \frac{1}{N}\|\mathbf{r}\|_2^2, \quad \mathbf{r} = [r_1\ r_2\ \dots\ r_N]^\top \\ &= \frac{1}{N}\left\|\mathbf{y} - \mathbf{y}_{pred}\right\|_2^2, \quad \mathbf{y} = [y_1\ y_2\ \dots y_N]^\top. \end{aligned} \tag{4-6}$$

In Eq. (3-6) the batch feature map $\mathbf{Z}(\mathbf{X}^{(d)})$ is defined. With the batch feature map a batch of predictions, $\mathbf{y}_{pred}$, can be computed as follows:

$$\mathbf{y}_{pred} = \left(\mathop{\circledast}_{d=1}^{D} \mathbf{Z}(\mathbf{X}^{(d)})^\top \mathbf{W}^{(d)} \right) \mathbf{1}_R \in \mathbb{R}^N. \tag{4-7}$$

Subsequently, the partial derivative of the batch MSE with respect to factor matrix $\mathbf{W}^{(d)}$ is given by

$$\frac{\partial mse}{\partial \mathbf{W}^{(d)}} = \frac{-2}{N}\left(\mathbf{Z}(\mathbf{X}^{(d)}) \,\tilde{\circ}\, \left(\mathop{\circledast}_{p=1,p\neq d}^{D} \mathbf{Z}(\mathbf{X}^{(p)})^\top \mathbf{W}^{(p)} \right)^\top \right) \times_3 (\mathbf{y} - \mathbf{y}_{pred}) \in \mathbb{R}^{M\times R}, \tag{4-8}$$

where $\times_3$ is the mode-3 vector product defined in Eq. (2-15) and $\tilde{\circ}$ is the batch outer product defined in Eq. (2-13).

The partial derivative with respect to the regularization term is given by

$$\frac{\partial \lambda \langle \mathcal{W}, \mathcal{W}\rangle_F}{\partial \mathbf{W}^{(d)}} = 2\lambda \mathbf{W}^{(d)} \mathbf{\Gamma}_d, \tag{4-9}$$

where $\mathbf{\Gamma}_d$ is defined as follows:

$$\mathbf{\Gamma}_d = \mathop{\circledast}_{p=1,p\neq d}^{D} \mathbf{W}^{(p)\top}\mathbf{W}^{(p)}.$$

Hence, the total partial derivative of the loss function with respect to a factor matrix is

$$\mathbf{G}^{(d)} = \frac{-2}{N}\left(\mathbf{Z}(\mathbf{X}^{(d)}) \,\tilde{\circ}\, \left(\mathop{\circledast}_{p=1,p\neq d}^{D} \mathbf{Z}(\mathbf{X}^{(p)})^\top \mathbf{W}^{(p)} \right)^\top \right) \times_3 (\mathbf{y} - \mathbf{y}_{pred}) + 2\lambda \mathbf{W}^{(d)} \mathbf{\Gamma}_d. \tag{4-10}$$

All-at-once gradient

With the expression for the partial derivative for a single factor matrix the all-at-once gradient can be derived. The all-at-once gradient is defined as follows:

$$\mathcal{G}_{d,:,:} = \mathbf{G}^{(d)} = \frac{\partial f(\mathcal{W})}{\partial \mathbf{W}^{(d)}} \in \mathbb{R}^{M\times R}, \quad \mathcal{G} \in \mathbb{R}^{D\times M\times R}. \tag{4-11}$$

The gradient is defined as a tensor, since the CPD weights are also represented as a the tensor $\mathcal{W}$. As a result, the gradient can easily be subtracted from the weights in a weight update.

Furthermore, it allows for an efficient computation of the gradient without looping over the factors.

The strategy for computing the gradient with respect to all factors at once, is to reuse as much as possible. In particular, the Hadamard product sequences that appear in the MSE term and the regularization term can be reused. Furthermore, Numpy/JAX functions are used to efficiently compute components for all the factors at once using broadcasting[1]. The tensor representation of a CPD, $\mathcal{W}$, and the tensor representation of the batch feature map, $\mathcal{Z}(\mathbf{X})$ are used to enable this. They were introduced in Section 2-2-3 and Section 3-1-2 respectively.

In Algorithm A.3 the full algorithm for computing the all-at-once gradient is presented. The derivation is given in Section A-2-1. An important aspect of the algorithm is that no loops are required[2]. This makes it an efficient and fast algorithm.

Finally, it is noted that the computation of the all-at-once gradient easily allows for parallelization. The mapping of the data, $\mathbf{Z}(\mathbf{X}^{(d)})$, can be done in parallel as well as the computations of $\mathbf{Z}(\mathbf{X}^{(d)})^\top \mathbf{W}^{(d)}$ and $\mathbf{W}^{(d)\top}\mathbf{W}^{(d)}$. Once these individual products have been combined into the full Hadamard sequences $\mathbf{ZW}$ and $\mathbf{\Gamma}$, and the residual $\mathbf{r}$ is computed, the gradient with respect to each factor matrix can be computed in parallel. Even the weight update can also be done in parallel. So, it is expected that it is possible to achieve a significant decrease in run time for the SteGD with parallelization. This was not implemented for this thesis, but could be studied in future research.

4-2-2 Automatic differentiation

An alternative to using an analytical expression is so-called automatic differentiation (AD). AD can be used to obtain values for derivatives of functions without the need for explicit expressions. Before it is introduced what AD is, it is also important to know what AD is not. As mentioned in [45], AD is not symbolic differentiation of numerical differentiation. Rather, numerical values of derivatives are computed by using the symbolic rules of differentiation. Numerical values of derivatives are of interest, because they can be used in the iterative optimization schemes.

Conceptually AD can be thought of as augmenting an existing programmed numerical function, such that the derivative of that function can be calculated when the function itself is evaluated. Each function is in essence an combination of basic elements for which the derivatives are known, such as addition and multiplication, or more complex operations like the exponents and sines. Using the chain rule, the derivative of the total function can be computed based on the derivatives of the function building blocks. Naturally, AD relies heavily on the chain rule. Moreover, AD can also compute derivatives when control flow is used, such as loops or conditional statements, making it a very versatile tool.

AD has been applied previously to tensors networks [46]. The authors describe how AD can be applied to compute gradients for tensor networks. The tensor networks they focus on are

[1]See https://numpy.org/doc/stable/user/basics.broadcasting.html for more information about Numpy broadcasting.

[2]Under the hood loops will be needed, but for Numpy these are written and optimized in C-code. So, it is beneficial to use this instead of loops in Python. See https://numpy.org/doc/stable/user/whatisnumpy.html#why-is-numpy-fast.

more complicated than the earlier introduced tensor decompositions, but similar operations are used to construct them. No reference was found in the literature of the combination of AD and CPDs.

For an overview related to the programming implications of AD in general the reader is advised to read [47]. For a general introduction and an overview of AD in machine learning the reader is advised to read [45]. In this thesis the JAX library and its AD functionality are used. This decision is discussed in Section C-3.

To demonstrate the ease of using AD an simplified example is given in Listing 4.1. It is noted that this simplicity is not due to the use of a CPD as the weights or the nonlinear least squares loss function. Any loss function and weight format can be used. It would, therefore, also be possible to use a CPD in combination with a more complex loss function for a which an analytical expression of the gradient is not readily available. Furthermore, another tensor decomposition could be used as well, for example a Tensor Train.

Listing 4.1 Illustration of simplicity of using AD with JAX for a simplified Python example.

```python
from jax import grad

def loss_function(weights, X, y, lambda_reg):
    ...
    return loss

def weight_update(weights, X, y, lambda_reg, learning_rate):
    gradient = grad(loss_function)(weights, X, y, lambda_reg)
    return weights - learning_rate * gradient
```

4-3 Proof of concept

In this section a proof of concept will be presented. For the proof of concept several experiments are conducted in which all-at-once optimization is applied on two relatively small data set. It is demonstrated that all-at-once gradient descent can be applied to CPD learning, but there are challenges that need further investigation. First, the setup of the experiments is described. Then the results are presented and points of interest are identified which will be analyzed in more detail in the next chapter.

4-3-1 Experiment setup

For both data sets the same experiment set up was used. First the two data sets are introduced as well as the preprocessing of the data. Then it is described in more detail how the earlier introduced methods are applied and the definition of one iterations that is used in this thesis is given. Finally, the hyperparameters of the optimization problem are discussed.

Data sets Two data sets were used to study the applicability of the SteGD method. The Banana data set has relatively larger number of samples, 5300, but only two features. The Airfoil data set has less samples, 1502, but has five features. Moreover, the Banana data set is a classification task and the Airfoil data set is a regression problem. See Section B-7 for more information about the data sets.

For both data sets the input data is preprocessed in the same way. The input data is scaled to the range $[0, 1]$ for each feature separately. This is done as follows:

$$\mathbf{X}_p^{(d)} = \frac{\mathbf{X}^{(d)} - \min(\mathbf{X}^{(d)})}{\max(\mathbf{X}^{(d)}) - \min(\mathbf{X}^{(d)})}, \tag{4-12}$$

where $\mathbf{X}_p^{(d)}$ is the preprocessed data.

For the output data the preprocessing differs due to the different tasks. For the Banana data set one of the class labels is set to -1 and the other label to 1. The is needed for classification tasks as described in Section 3-1-1. For the Airfoil data set the output values are normalized around zero by subtracting the mean output and dividing by the standard deviation of the output:

$$\mathbf{y}_p = \frac{\mathbf{y} - \text{mean}(\mathbf{y})}{\text{std}(\mathbf{y})}. \tag{4-13}$$

The data is split up into a training data set and a validation data set. The validation split determines which portion of the whole data set becomes the validation data set. A validation split of 0.1 is used.

Methods The ALS method itself has no hyperparameters. For SteGD there are two hyperparameters, the batch size and the step size. The objective in this proof of concepts is to investigate whether all-at-once optimization can be applied and not yet how it can be applied optimally. The values for these hyperparameters were, therefore, chosen naively. The step size is set to one for all iterations, $\alpha_k = \alpha = 1$. For the batch size the performance of both full-batch and mini-batch SteGD are compared. A mini-batch size of 100 is used. Finally, both the analytical expression and AD are used to compute the values for the gradient to analyze whether they produce the same results.

Due to the inherent difference between ALS and SteGD, there are multiple ways to define one iteration of the training process. Furthermore, the usage of mini-batches further complicates this definition. In this thesis the following definition of a iteration will be used.

Definition 4.1 (Iteration/Epoch). *One iteration, or epoch, is defined as any number of updates until all the training data has been used once.*

This means that for the ALS method one iteration is equal to an update of one factor. For the full-batch method one iteration is equal to one update of all factors. For the mini-batch method one iteration implies multiple updates depending on the batch size. It is noted that different definitions of an iteration are also possible. Furthermore, the choice of definition might influences the perspective the training results offer. This is further deliberated in Section B-1. This definition of an iteration allows for a detailed description of the training progress of both methods. It is, for example, more detailed than the case where one update of all factors is one iteration. In the other hand, there will not be an information overload which can occur when one weight update is one iteration and a mini-batch method leads to tens of thousands of updates. So, it is believed that this definition of an iteration allows for a fair comparison of the methods.

After each iteration the training and validation losses are computed. For the training loss the full training data set is used even when the mini-batch method is used. Additionally, the training and validation loss are determined before any updates are performed, so that the loss at initialization is also taken into account.

Occasionally the loss after a weight update will be used, but this will be clearly stated. In that case the loss is reported for the data that was used in that update. Hence, for the mini-batch method the loss over that mini-batch is reported.

Finally, the ALS updates are performed in sweeps. One sweep is defined as updating the factors in the order $1 \rightarrow D$. Note that this is different from the definition of a sweep in [6]. With this definition of a sweep all factor matrices have been updated once in one sweep. One sweep is thus equal to D iterations.

Weight initialization As mentioned in Section 3-2-1, the weight initialization is an important aspect of iterative optimization algorithms. For this proof of concept it was naively opted to use random normalized uniform initial weights. Each vector in the CPD is drawn from an uniform distribution in the range $[-1,\ 1]$ and then normalized to unit length. Note that normalized initial weights are used, but not a normalized CPD as described in Eq. (2-22). The procedure for initializing random normalized uniform weights is given in Algorithm A.6. To create a fair comparison between the different methods, in an experiment the same initial weights are used for each method.

Hyperparameters of the optimization problem Besides the hyperparameters of the optimization methods, the optimization problem itself also has hyperparameters. The hyperparameters are summarized in Table 4-1.

An important parameter of the CPD, and thus also the problem, is the CP-rank R. It was chosen to use a CP-rank of 5 for both data sets. This was deemed an appropriate value, since similar values worked well with in [6].

The Fourier feature map given Eq. (3-8) is used. For its parameters the values $M = 12$, $l = 0.1$ and $U = 1$ are used.

Finally, a number of parameters related to the training itself need to be chosen. A different number of ALS sweeps are used for the Banana data set and the Airfoil data set, because the Banana data set only has two features. As a result, only a few updates are needed to reach convergence with ALS. For the Airfoil data set twenty sweeps were used for ALS and a hundred iterations for SteGD such that the methods use an equal number of iterations.

4-3-2 Results

In this section the results of the proof of concept are presented and analyzed to identify potential challenges of all-at-once optimization.

Table 4-1: Hyperparameters used for training on two different data sets.

	Data set	
Parameter	Banana	Airfoil
CP-rank R	5	5
Feature map size M	12	12
Feature map bound U	1.0	1.0
Feature map length scale l	0.1	0.1
Step size α	1	1
Regularization λ	0.00001	0.00001
Mini-batch size	100	100
Validation split	0.1	0.1
ALS sweeps	10	20
Number of iterations for SteGD	100	100

Banana data set

In Figure 4-1a the training loss during training on the Banana data set is shown for the different methods. The final training and validation loss are given in Table 4-2. Additionally, since the Banana data set only has two features, it is possible to plot the decision boundaries of the resulting models. These are shown in Figure 4-1b.

It is first noted that two lines coincide in Figure 4-1a. As expected and desired, the full-batch SteGD using the analytical expression, called plainly 'SteGD', and method using AD have the same training loss trajectory. Both methods should produce the same gradient values and thus the same training progress. This is confirmed by Figure 4-1a and the fact that the final losses are equal. Although only three decimals are given in Table 4-2 the losses were in fact identical for additional decimals.

Secondly, the mini-batch SteGD converges significantly faster than the full-batch method. However, the definition of one iteration, Definition 4.1, needs to be taken into account. For the mini-batch method the weights are updated 47 times in the first iteration, while for the full-batch method only one update is performed. There are 47 updates, since the validation split is 0.1, there are 5300 samples in the full data set and the batch size is 100, so $(5300 \cdot 0.9)/100 = 47.7 \rightarrow 47$ updates. This explains why the convergence of the mini-batch method can be so much faster. Moreover, it illustrates the effect of the definition of one iteration on the depiction of the results. Nevertheless, this result suggests that is possible to use mini-batches for efficient training. The ALS method works well as already shown in [6].

Finally, Figure 4-1b demonstrates that the CPD constrained kernel machine produces a complex nonlinear decision boundary. It is concluded that the three different methods, counting the two full-batch methods as one, result in different final models.The decision boundaries produced by the final models are overall different. Interestingly, the mini-batch SteGD decision boundaries seem closer to the ALS boundaries than the full-batch method. However, in the areas where the two classes meet and the decision boundary is truly important, the produced decision boundaries overlap mostly for all the methods.

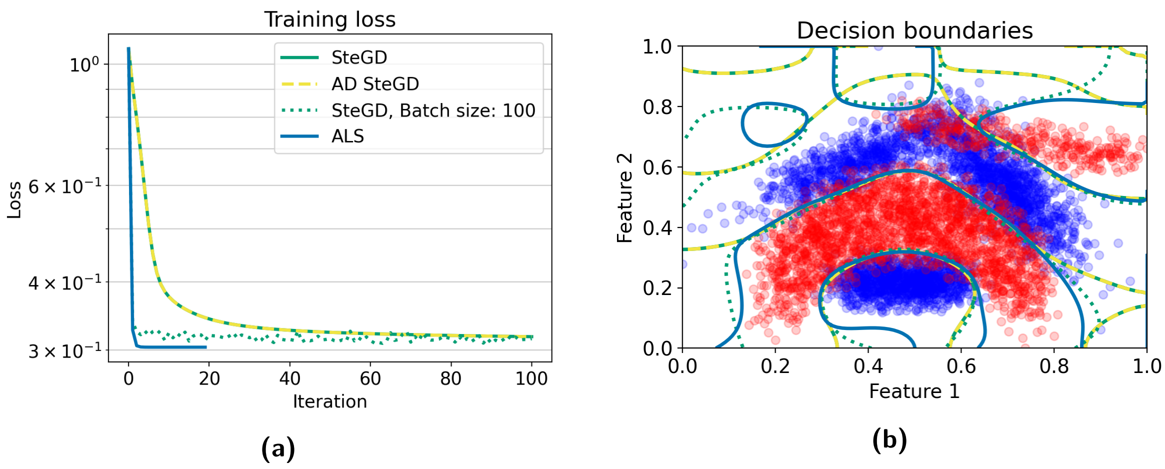


Figure 4-1: Training loss and the final decision boundaries on the Banana data set for different methods. Analytical and AD SteGD overlap as expected. Figure 4-1a shows that SteGD works and mini-batch descent converges faster. Figure 4-1b demonstrates that the CPD constrained kernel machine produces a complex nonlinear decision boundary. Furthermore, the models overlap in the critical regions which reflects their similar loss performance.

Table 4-2: Training and validation losses for different methods after training on the Banana and Airfoil data set. The results show that the SteGD method can achieve similar final losses as ALS, but it depends on the data set.

	Banana		Airfoil	
Method	Training	Validation	Training	Validation
SteGD	0.317	0.321	0.554	0.467
AD SteGD	0.317	0.321	0.554	0.464
Mini-batch SteGD	0.315	0.311	**0.550**	0.464
ALS	**0.303**	**0.296**	0.551	**0.463**

Airfoil data set

The training progress on the Airfoil data set is shown in Figure 4-2 and the final losses are given in Table 4-2.

First, it is noted that in Figure 4-2a the training failed for the full-batch SteGD. For the mini-batch SteGD the training did succeed, but for the first fifteen iterations there is little progress. Fifteen iterations correspond to $1502 \cdot 0.9/100 \cdot 15 = 202$ weight updates. The same experiment was run for more iterations and the results are shown in Figure 4-2b. Indeed, after more than 200 iterations the full-batch method does start to make progress as well. Furthermore, it can be seen that the full-batch method becomes unstable around 500 iterations. The instability could be due to the indeterminacies of the CPD. Alternatively, a local minimum or saddle point could have been overshot. The instability was further investigated in Section B-2 and it is likely that the spike in loss is indeed due to an overshoot. After the period of instability the full-batch method converges to a similar loss as the other methods.

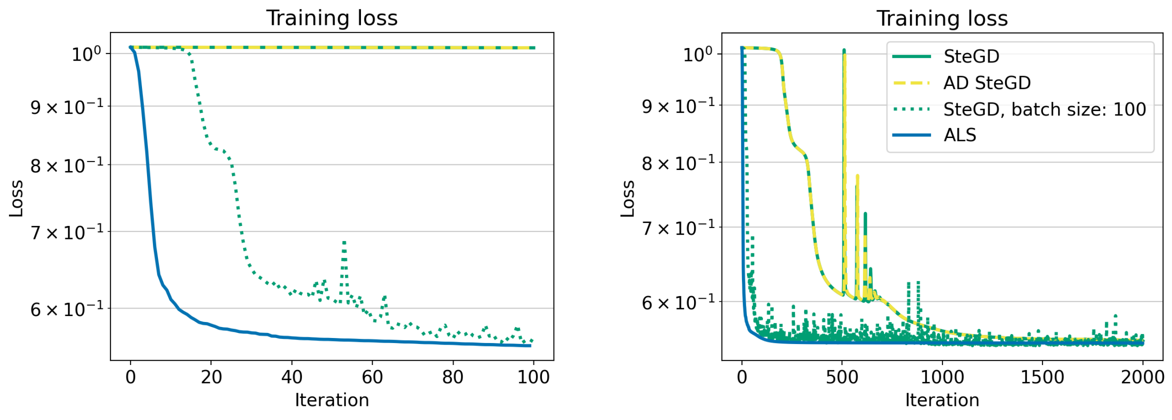
Again the full-batch method using the analytical expression and the AD method have almost the same result. In Table 4-2 it can be seen that the final losses differ slightly. This is most likely due to the instabilities during training, since the gradient values are significantly larger there. As a result, a small numerical difference likely occurred which caused a small difference in the end. It was verified that before the instability the losses were still identical. Due the results on the Banana data set and these results it is reasoned that the methods can be used interchangeably.

To determine which method is better, the run time for both methods was evaluated. For both methods it was timed how long it takes to complete a 100 iterations on the Airfoil data set with the full-batch method. The results are summarized in Table 4-3. From these results it can be concluded that the analytical expression is, at least on the Airfoil data set, faster. This is also expected. Naturally, using the known explicit expression is expected to be faster than letting the computer perform the derivation. Since faster training with the same performance in terms of less is in general beneficial, it was opted to use the analytical expression for the rest of this thesis.

Note that a comparison based on the run time is only one perspective. Alternatively, the memory complexity could, for example, have been used as the comparison criterion. Nevertheless, it was concluded that the analytical expression is more advantageous.

Table 4-3: Mean training run time and one standard deviation on the Airfoil data set for two gradient computation methods. The analytical run time is significantly faster than AD, so it is in that regard more advantageous to use the analytical gradient computation.

	Training run time
Analytical	**0.582** (0.115)
Automatic differentiation	0.855 (0.192)



(a) Normal run. **(b)** Long run.

Figure 4-2: Training loss on Airfoil data set for different methods. Analytical and AD SteGD overlap as desired. Full-batch SteGD takes longer to convergence. Furthermore, it can be unstable which is likely due to an overshoot of a local minimum or saddle point. The mini-batch implementation speeds up the training.

The effect of using mini-batches can be nicely seen when the training loss per weight update is plotted. In Figure 4-3 the training loss is shown for full-batch and mini-batch SteGD for two different batch sizes. Note that the loss per weight update is given, so not per iteration. In total a 100 iterations were used for each of the three runs. It can be seen that the trend in the training loss is similar for the full-batch method as for the mini-batch methods. Especially with a batch size of 500 the trend is very similar. The full training data set has $1502 \cdot 0.9 = 1352$ samples, so that means that 37% of the training data is used per weight update for a batch size of 500. For a smaller batch size there are more oscillations in the loss. This is expected, since it is more likely for smaller batches of data that the pattern in the batch deviates from the overall pattern in whole data set. As a result, the mini-batch MSE deviates more from the full-batch MSE and thus the loss deviates more as well. Nevertheless, the trend is still comparable.

These results show why the mini-batch method works well. It can achieve a similar learning curve as the full-batch method, but can speed up training by performing more weight updates per iteration. So, mini-batch SteGD is deemed a viable and efficient alternative to the full-batch method.

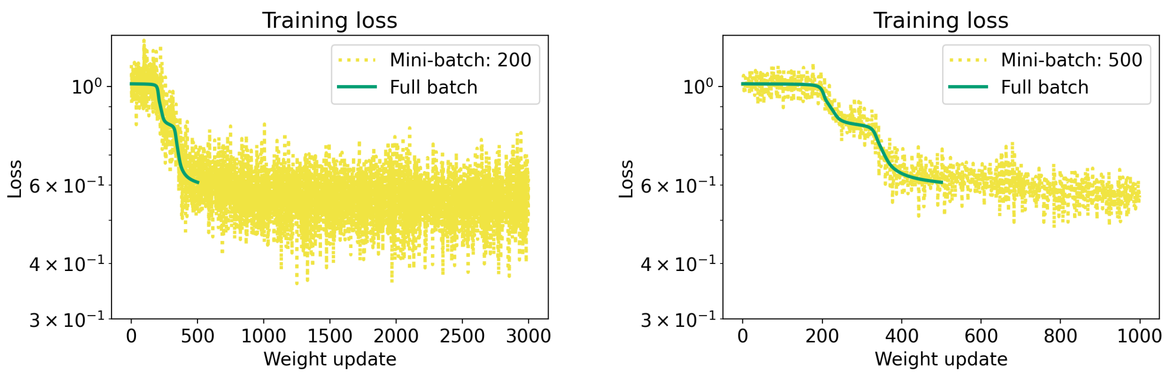


Figure 4-3: Training loss for full-batch and mini-batch SteGD for different batch sizes on the Airfoil data set. The loss is given per weight update, so not per iteration. The losses of the mini-batch method follow a similar trend as the full-batch loss, but the mini-batch method performs more updates for a set amount of iterations which speeds up training.

Another option to speed up the training would be to increase the step size α. Since a step size of one, $\alpha = 1$, was already used, it is investigated whether an even larger step size would be beneficial. To test this, models are trained again on the Airfoil data set using full-batch SteGD with different step sizes. The step sizes $\alpha \in [1, 2, 3, 4, 5]$ are used. For the other hyperparameters the same values as previously are used. Both the training and validation losses during training are shown in Figure 4-4. For larger step sizes the losses reduce quicker. However, it can also be seen that training becomes unstable for both the training and validation loss for larger step sizes. This demonstrates that selecting an adequate step size can be challenging.

As mentioned earlier, a main difference between the Banana and Airfoil data set is the number of features. The challenge of selecting a proper step size and its relation to the number of features is analyzed in more detail in the next chapter, but first all-at-once optimization will be compared with ALS based on the characteristics of the methods.

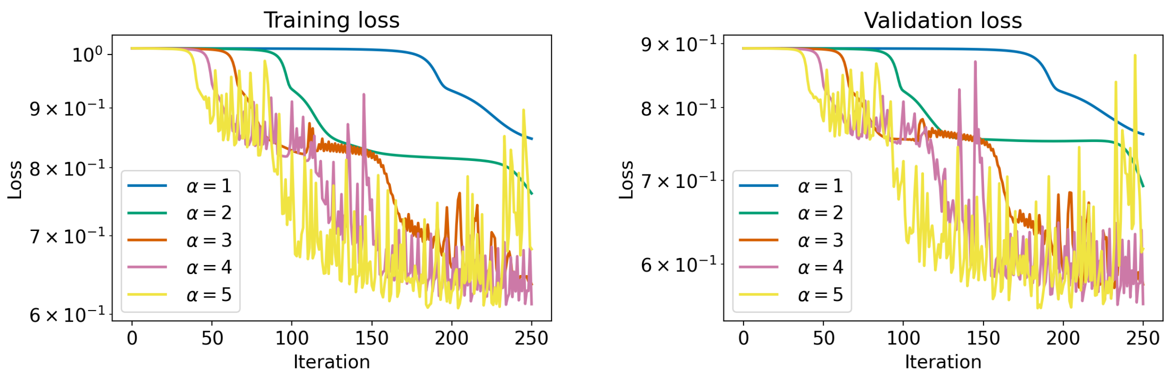


Figure 4-4: Losses for different step sizes α during training with full-batch SteGD on the Airfoil data set. For larger step sizes the training initially converges faster, but later becomes unstable in both the training and validation loss. So, applying a larger fixed step size is not a straightforward option.

4-4 All-at-once optimization characteristics compared to ALS

The SteGD and ALS method are inherently different methods that use different approaches to solve an optimization problem. In this section the method will be compared based on their characteristics. First, the theoretical computational and memory complexity of the methods are compared. The computational complexity is then studied in more detail by timing the run times of the two methods. Secondly, the implications of the difference in the weight update approach are studied in more detail.

4-4-1 Theoretical complexity

The theoretical complexity of an algorithm or set of operations describes theoretically how many elementary operations are needed to run the algorithm, the computational complexity, or how many elements need to be stored, the memory complexity. Often not the exact numbers are determined, but rather the order of magnitude as a function of the inputs of the algorithm is used. This is denoted by the Big O notation $\mathcal{O}(\cdot)$.

To compare ALS with SteGD the computational and memory complexity of both methods are analyzed. The complexity of weight updates with N samples is used. The complexity of SteGD is derived in Table A-2 and for ALS in Table A-1.

Computational complexity

For SteGD the computational complexity of one weight update is $\mathcal{O}(DMRN)$, assuming that $N \gg R$. The complexity is dominated by the computation of the gradient of the MSE term, since this complexity depends on the number of samples. Most importantly, the complexity scales linear in all input parameters, D, M, R and N. Since the complexity scales linear in the number of samples, this makes it a suitable method for large scale learning.

The computational complexity of one update with ALS is $\mathcal{O}(M^2R^2N)$, assuming that $N \gg MR$. The complexity is dominated by the computation of the components of the normal equation that is used to solve the linear least squares (LLS) system. Thus, the complexity of updating all factors once is $\mathcal{O}(DM^2R^2N)$. Again the complexity scales linearly in the number of samples N, so this method is also suitable for large scale learning.

So, the complexity of updating all the factors with SteGD is of a lower order than updating all the factors with ALS. The difference is of order $\mathcal{O}(MR)$. Hence, from that perspective it is more efficient to use SteGD. However, when the complexity of the update of one factor with ALS is used, it depends on the relation between D and MR. Common values for M and R are 20 and 5, respectively, or higher, while a hundred features is quite a lot. So, generally $D \ll MR$. Thus, the computational complexity of one weight update with ALS is also higher than the complexity of one update of all factors with SteGD.

As mentioned earlier, there are multiple ways to compare these methods. The decrease in loss is, for example, not taken into account in this comparison. It is expected that the ALS method has learned more after all factors are updated in one sweep than SteGD has after one update. This can also be seen in the results presented in the previous section.

Since the complexity scales linearly with the number of samples, the complexity of η updates with N/η samples is the same as one update with N samples. Hence, a mini-batch method can be very efficient, because it can, theoretically, perform multiple updates in the same time as one full-batch update. In practice, the mini-batch method might be even faster, because the computer can more easily handle smaller batches of data due to the limited size of the processors caches.

Memory complexity

While the computational complexity can easily be defined as the required operations needed in an algorithm, this is more complicated for the theoretical memory complexity. Mainly, the memory complexity can be defined for a given implementation of an algorithm or the most memory efficient implementation. For example, the implementation of the all-at-once gradient given in Algorithm A.3 computes the gradient in batches. Thus, its memory complexity is in practice related to the number of samples N. Alternatively, the gradient can be computed for each sample individually and summed. The memory complexity would then no longer scale with N. The memory complexity of both versions is taken into consideration in the following analysis.

The memory complexities for SteGD and ALS are derived in Table A-2 and Table A-1 respectively per step of the algorithm. The memory complexity of a step is defined as the highest order memory complexity of any intermediate results and the output of that particular step. The input of the step is not considered, since this is taken into account in the previous step.

For the most memory efficient implementation the memory complexity is SteGD is $\mathcal{O}(DMR)$. This is the same memory required to store the CPD weights. The memory complexity of the batch implementation is $\mathcal{O}(DMRN)$ for a batch size of N. This illustrates a problem of the batch implementation. For large data sets it can become infeasible to compute that gradient for the whole data set in one batch due to limited memory. Alternatively, the gradient could be computed in smaller subbatches and summed to obtain the total gradient. This approach would benefit from the efficiency of a batch implementation while requiring a smaller amount of memory.

The memory complexity of the ALS method is $\mathcal{O}(M^2R^2)$ for the most memory efficient implementation. Assuming $N \gg MR$, the complexity is $\mathcal{O}(MRN)$ for a batch implementation with batch size N. The ALS memory complexity does not scale in D, since only one factor is updated the time.

So, assuming again that $D \ll MR$, the SteGD method has a better memory complexity than ALS. Batch implementations increase the memory complexity of both methods, but can decrease the run time. This is a trade-off than needs to be taken into account.

Run time comparison

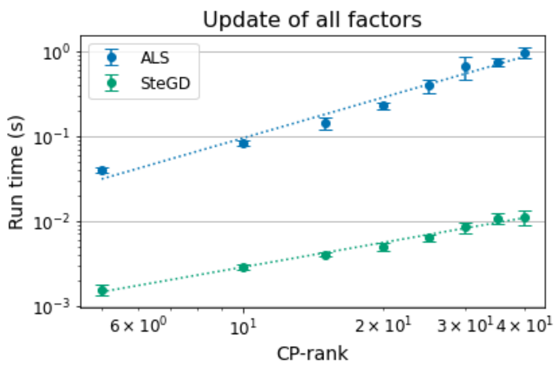
The earlier derived theoretical computational complexity indicates how the two methods scale for different inputs and how they relate to each other. In this section the weight updates of the two methods will be timed to investigate how their run times scale in practice.

The run time of one update of all factors is analyzed. This corresponds to D ALS updates and one full-batch SteGD update. In the experiment one ALS update is timed and this is multiplied by D. It is assumed that an update will take equally long for each factor. The execution time is evaluated for different values of M and R. These parameters are varied, since the two methods theoretically scale differently for them. A randomly generated data set with $N = 10000$ samples with $D = 10$ features is used. Random data is used, since the value of the resulting weight update is not relevant. A relatively large number of samples was used to ensure $N \gg MR$. When the CP-rank is varied, the feature map of order $M = 20$ is used. For the CP-rank the values $[5,\ 10,\ 15,\ 20,\ 25,\ 30,\ 40]$ are used. When the feature map order is varied, the CP-rank $R = 10$ is used. For the feature map order the values $[10,\ 20,\ 30,\ 40,\ 50, 60]$ are used.

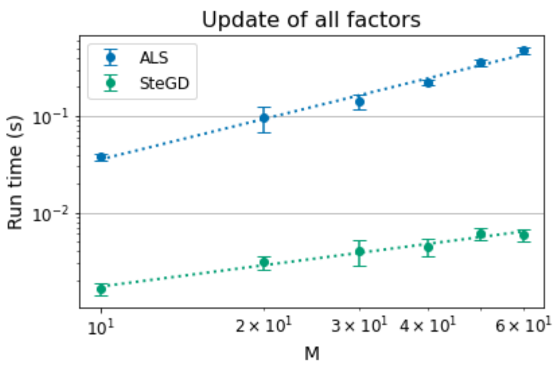
Per variation the run time is evaluated 11 times, but the first timing is discarded. This is due to additional overhead for the first run which is not taken into account. For each variation the same randomly initialized weights are used. The other hyperparameters are the same for each experiment; $\lambda = 0.00001,\ \alpha = 0.1,\ l = 0.1,\ U = 1$. They are assumed to be irrelevant for the run time. Finally, the jitted versions of the batch implementations are compared, since these are also used for the other experiments in this thesis. Jitting can reduce the run time, but should not decrease the order of the complexity. The experiments are conducted on a CPU.

The results are shown in Figure 4-5 in log-logs plots. As can be seen, one update of all the weights is significantly faster with SteGD than with ALS for all values of M and R. Additionally, it was estimated how the run times scale as a function of the varied parameter in practice. Since log-log plots are used, the scaling can be determined by fitting a linear line through measurement points. The mean run times are used here as the measurement points. The estimated scaling parameters are given in Table 4-4. The fits are plotted Figure 4-5. The estimated parameter b is particularly of interest, because this parameter determines how the run time scales for a given input.

From the estimated scaling parameters it can be seen that they are different from the expected values of one for SteGD and two for ALS. An explanation for this could be that the computer



(a) $M = 20$.



(b) $R = 10$.

Figure 4-5: Mean run time of one update of all factors with one standard deviation. Additionally, a fit (dotted) through the mean run times is shown. The difference in the slope of the fit shows that the SteGD update time scales favorably compared to ALS in terms of both the feature map order M and the CP-rank R. This is expected based on the theoretical computational complexity.

is able to perform some computations in parallel. For example, in matrix-matrix products computations could be done in parallel which would explain the lower scaling parameters.

Nevertheless, the estimated scaling parameter b for the input M is almost twice as large for ALS as for SteGD. In other words, the scaling is twice the order which is also the case for the theoretical computational complexity. For the CP-rank R ALS also scales worse than SteGD, but the difference is of order 1.5 instead of two. This could be due to the complexity of the regularization term which theoretically scales quadratically in R for both methods. Although this term is not the decisive factor for the theoretical computational complexity, it could dampen the difference in the practical complexity with regard to R between ALS and SteGD.

Table 4-4: Estimated scaling parameters for the weight update run time (y) for two different methods and inputs. The results for b verify that the SteGD update time scales favorably compared to ALS in terms of M and R, like the theoretical computational complexity suggests.

		$y = ax^b$	
Input (x)	Method	a	b
CP-rank R	ALS	$2.4 \cdot 10^{-3}$	1.60
	SteGD	$3.1 \cdot 10^{-4}$	0.97
Feature map order M	ALS	$1.4 \cdot 10^{-3}$	1.40
	SteGD	$3.2 \cdot 10^{-4}$	0.73

4-4-2 Weight update comparison

The weight update method for SteGD and ALS are inherently different. The ALS method involves solving a linear system of equations, while the SteGD requires the computation of the gradient and the determination of a step size. In this section the implications that arise from this are discussed.

Stochastic updates

A considerable difference between ALS and SteGD is the possibility of applying stochastic updates or mini-batch updates. Stochastic updates allow for multiple updates before the full data set is used. In the proof of concept it was shown that the mini-batch method was able to boost the performance of SteGD significantly.

In future research it could be investigated whether a stochastic version of ALS can be effective. In [48] and [49] the authors propose stochastic ALS, but it is in the context of fitting a CPD to a known tensor. For the CPD learning problem, the ALS algorithm could be adapted such that it updates all the factors based on a mini-batch of the data and subsequently uses a different mini-batch for the next D updates. It is reasoned that the mini-batch batch size would need to be substantially larger than for SteGD. ALS minimizes the loss function in each weight update which requires computational effort. It is, therefore, more important that this update is performed in the correct direction. So, a larger mini-batch size is most likely needed to ensure that the update direction is a good enough estimate of the update direction for the whole data set. For SteGD only a small step is taken, so it is less costly if the step is taken in the wrong direction.

Step size

An advantage of the ALS method is that the step direction and step size are computed at once by finding the solution to the LLS subproblem. As a result, it is not necessary to select a step size for ALS. Furthermore, ALS takes to most optimal step at each weight update. So, the decrease in loss per weight update is expected to be in general much larger than for SteGD.

In the proof of concept it was already shown that it is a challenge to select an suitable step size for the SteGD method. So, it is beneficial that this is not needed with ALS. In the next chapter different solutions for selecting a suitable step size for the gradient descent method are studied.

4-5 Summary

In this chapter all-at-once optimization of the CPD constrained kernel machine was studied. The general all-at-once optimization framework was introduced first. This thesis focuses on all-at-once gradient descent methods in which the step direction is the negative gradient of the loss function. The Steepest Gradient Descent (SteGD) method is the simplest of these methods and is used to study the applicability of all-at-once optimization. SteGD can be applied in an full-batch manner in which the full training data set is used to compute the gradient. Alternatively, a mini-batch method can be used in which the gradient is computed for a batch of the training data. As a result, multiple updates can be performed before the full training data set has been used.

Two methods were discussed for obtaining a value for the gradient. First, an analytical expression for the all-at-once gradient which was derived for this thesis is presented. It is an algorithm that efficiently makes use of the tensor representation of the CPD and batch feature map. Furthermore, it reuses intermediate results as much as possible to compute the gradient for the different factor matrices. Finally, it makes use of Numpy/JAX functionalities to efficiently compute multiple terms at once.

Secondly, automatic differentiation (AD) was studied as an alternative to the analytical expression. It is shown that with JAX AD can be applied easily in Python. Moreover, AD can be applied to compute the gradient of the CPD constrained kernel machine.

In a proof of concept it is shown with experiments on two small data sets that the SteGD method can applied for CPD learning. It is shown that the analytical gradient expression and AD have the same results. However, it is found that the run time of the analytical expression is considerably shorter than that of AD. As a result, it is concluded that the analytical expression for the gradient will be used for the rest of this thesis. It is shown that the mini-batch version of SteGD can speed up the training and improve the final losses.

It is found in the experiments that the SteGD method does not perform as well for a larger number of features, because the training is slower. It is shown that the selection of an appropriate size, which could speed up the training, is difficult. This is studied in more detail in the next chapter.

Finally, the characteristics of all-at-once optimization and ALS are compared. The theoretical computational complexity of one SteGD update is $\mathcal{O}(DMRN)$, while the complexity of an update of one factor with ALS is $\mathcal{O}(M^2R^2N)$. Generally, $D \ll MR$, so the computational complexity of SteGD scales favorable compared to ALS. This is verified with experiments for the run time of one update of all the factors. It is found that the SteGD run time scales favorably compared to ALS as a function of M and R, just like the theoretical computational complexity suggests.

Furthermore, it is reasoned that ability of SteGD to use mini-batch updates can given SteGD an edge over ALS. On the other hand, the absence of a step size in the ALS method can be an advantage.

Summary of contributions

- An analytical expression for the all-at-once gradient is presented. It is an efficient algorithm that computes the gradient for all the factor matrices at once.
- It is shown in a proof of concept that all-at-once optimization in the form of SteGD can be applied to the CPD constrained kernel machine. However, for a larger number of features SteGD takes significantly longer to converge.
- It is shown that AD can be applied for the CPD constrained kernel machine.
- It is shown that mini-batch SteGD can be applied to speed up training.
- It is shown that the run time of an update of all the factors scales favorable for SteGD compared ALS for the feature map order and the CP-rank.

Chapter 5

Challenges of all-at-once optimization

In this chapter several challenges of all-at-once optimization for the Canonical Polyadic Decomposition (CPD) constrained kernel machine will be analyzed. First, the determination of a suitable step size is studied. It is shown that this depends on the number of features. Two methods for determining a step size, Line search and the Adam method, are introduced. An general expression for the exact line search solution was derived for this thesis, so this is presented. The performance in terms of the final losses and convergence of both methods is evaluated with experiments. Secondly, the effect of the weight initialization on the performance is studied. Thirdly, the normalization of the CPD is analyzed. In Section 2-2-1 it was discussed that a normalized CPD does not have a scaling indeterminacy. Thus, it is investigated whether normalization influences the training performance. Finally, it is concluded which of the studied all-at-once gradient descent methods is the best and will be used in the next chapter as final method.

5-1 Step size

One of the hyperparameters in the Steepest Gradient Descent (SteGD) method is the step size. So far, a fixed step size has been used during training. In Section 4-3-2 a potential problem of using a fixed step size was identified. When the fixed step size is too small, the training converges so slow that it is deemed unsuccessful. It is undesirable to need thousands of iterations, since this implies that the training takes excessively long. On the other hand, when the step size is too large the training becomes unstable. Selecting a suitable step size, or learning rate, is a challenge in many machine learning problems. However, in combination with a CPD an additional challenge arises.

In this section it will first be shown that a suitable value for the step size depends on the number of features. Subsequently, two methods will be presented that can be used to compute a suitable variable step size. Line search is first considered. An expression for the exact line search solution was derived for this thesis and is presented here. Secondly, the Adam method [8] is discussed. For both methods the training performance is studied using the same proof of concept set up as in the previous chapter.

5-1-1 Dependency on the number of features

The step size and the step direction are listed in Section 3-2-1 as two separate parts of the general iterative optimization scheme, but the step direction does influence the step size. The step direction is a direction, but it does have a magnitude. This magnitude affects what suitable values for the step size are. Namely, when the magnitude is large it is likely more sensible to select a smaller step size, while for a very small magnitude a larger step size might be better. This relation is further complicated by the nature of the gradient. Near a minimum the gradient is expected to be smaller. So, a small gradient could mean that the gradients are small in general or that the current weights are near a minimum.

For SteGD the magnitude of the step direction is equal to the magnitude of the gradient. The challenge for all-at-once optimization with a CPD is that the magnitude of the gradient depends on the number of features. In particular, the gradient becomes smaller for a larger number of features.

Without loss of generality, the gradient of the loss function for a single sample will be analyzed. This gradient, as derived in Section A-2-1, is given by

$$\frac{\partial f_n(\mathcal{W})}{\partial \mathbf{W}^{(d)}} = -\frac{2}{N} r_n \mathbf{z}(x_n^{(d)}) \circ \left(\mathop{\circledast}_{p=1,p\neq d}^{D} \mathbf{z}(x_n^{(p)})^\top \mathbf{W}^{(p)} \right) + 2\lambda \mathbf{W}^{(d)} \left(\mathop{\circledast}_{p=1,p\neq d}^{D} \mathbf{W}^{(p)\top} \mathbf{W}^{(p)} \right).$$

The gradient of the mean squared error (MSE) term for a single sample is an estimate for the gradient for a batch of samples. The order of magnitude of the gradient is, thus, not expected to be different for a single or multiple samples. So, the following analysis of the gradient of the MSE for a single sample should hold for multiple samples as well.

In Section 3-1-2 it was shown that the values of the Fourier feature map, which is used in this thesis, are always smaller than one and in general considerably smaller. Furthermore, for the random weight initialization, as described in Section 4-3-1, values are drawn from the uniform distribution on $[-1, 1]$. The normalization step in the weight initialization further diminishes the values. So, the values of the initial weights are also smaller than one.

During the normalization step the individual vectors of the rank-one tensors of the CPD are divided by their norm. The vectors are of size M and their norm thus depends on the feature map order M. Namely, a larger M implies that the vector contains more elements and, as a result, will have a larger norm. Since the vectors are divided by their norm, a larger M implies smaller elements in the initial factor matrices.

So,

$$|\left(\mathbf{z}(x_n^{(d)})\right)_m| \ll 1, \quad |\left(\mathbf{W}^{(d)}\right)_{mr}| \ll 1,$$

where $m = 1, \dots M$, $r = 1, \dots, R$, $d = 1 \dots, D$. Therefore, it is reasoned that the values of the product of these terms are also smaller than one:

$$|\left(\mathbf{z}(x_n^{(d)})^\top \mathbf{W}^{(d)}\right)_r| \ll 1, \quad r = 1, \dots, R, \;\; d = 1 \dots, D.$$

Consequently, the Hadamard products of this term result in even smaller values. Given

$$\max(|\left(\mathbf{z}(x_n^{(d)})^\top \mathbf{W}^{(d)}\right)_r|) = a \ll 1, \quad r = 1, \dots R, \; d = 1, \dots, D,$$

then

$$\max(|\left(\underset{p=1,p\neq d}{\overset{D}{\circledast}} \mathbf{z}(x_n^{(p)})^\top \mathbf{W}^{(p)}\right)_r|) \leqslant a^{D-1}, \quad r = 1, \ldots, R. \tag{5-1}$$

So, the number of features affects the maximum values in this Hadamard sequence and thus influences magnitude the gradient of the MSE term. Since the feature map order influences the values in $\mathbf{W}^{(d)}$, the number of features D and the feature map order M are expected to influence the magnitude of the gradient of the MSE term.

The regularization term is less dependent on the number of features when the initial weights are normalized. The regularization term is related to the norm of the CPD and this norm is in turn determined by the normalization in the weight initialization. The norm of the CPD is related to the CP-rank, since it determines the number of rank-one tensors. Due to the normalization, the norm is not related to the number of features or the feature map order. Therefore, it is expected that the gradient of the regularization term is less related to D and M.

To confirm the presence of this issue the norm of the gradient is analyzed for different values of M and D. The norm of the gradient of the MSE term and the norm of the gradient of the regularization are analyzed separately. The norm of the gradient is used, since it gives a good indication of the magnitude of the values in the gradient. A downside of the norm is that it does not reveal information about the relative difference between values. The norm is dominated by the largest elements and does not reveal the magnitude of the smallest elements. However, the objective of this experiment is to illustrate the order of magnitude of the gradient in general, so this downside is taken for granted.

The Fourier feature map with bound $U = 1$ and length-scale $l = 0.1$ is used for all values of M. For the CP-rank $R = 10$ is used. The weights are initialized randomly from a uniform distribution on $[-1, 1]$ and then normalized as described in Section 4-3-1.

The Frobenius norm of the partial derivative of the particular term with respect to each factor matrix $\mathbf{W}^{(d)}$ is computed. The mean of these norms is the result of one experiment:

$$\begin{aligned} t(\mathcal{W}) &= mse \text{ or } t(\mathcal{W}) = \lambda \langle \mathcal{W}, \mathcal{W} \rangle, \\ \text{mean norm} &= \text{mean}\left(\left\| \frac{\partial t(\mathcal{W})}{\partial \mathbf{W}^{(1)}} \right\|_F, \left\| \frac{\partial t(\mathcal{W})}{\partial \mathbf{W}^{(2)}} \right\|_F, \ldots, \left\| \frac{\partial t(\mathcal{W})}{\partial \mathbf{W}^{(D)}} \right\|_F \right). \end{aligned} \tag{5-2}$$

For each experiment for the MSE term a randomly generated data set with 100 samples and D dimensions is used as input data and a randomly generated vector of size 100 is used as output data. Random data is used, since the objective is to compute the magnitude of the gradient that is not influenced by the presence of a local minimum. It is assumed that for random input and output data this is indeed not the case. Both the input and output data are drawn form a uniform distribution on the range [0, 1]. For the regularization parameter a value of one is used, $\lambda = 1$, such that the magnitude of the gradient is not influenced by this. For M the values [10, 20, 40] are used and for D the values [2, 4, 8, 10, 20] are used. These values were selected, because they are representative for values that occur in the studied learning problems.

It is noted that the number of samples can affect the magnitude of the MSE term due to the $1/N$ factor. Nevertheless, when a constant number of samples is used it should not influence the trend.

The results of the experiments are shown in Figure 5-1. The results show that the norm of the MSE and the regularization term clearly behave differently.

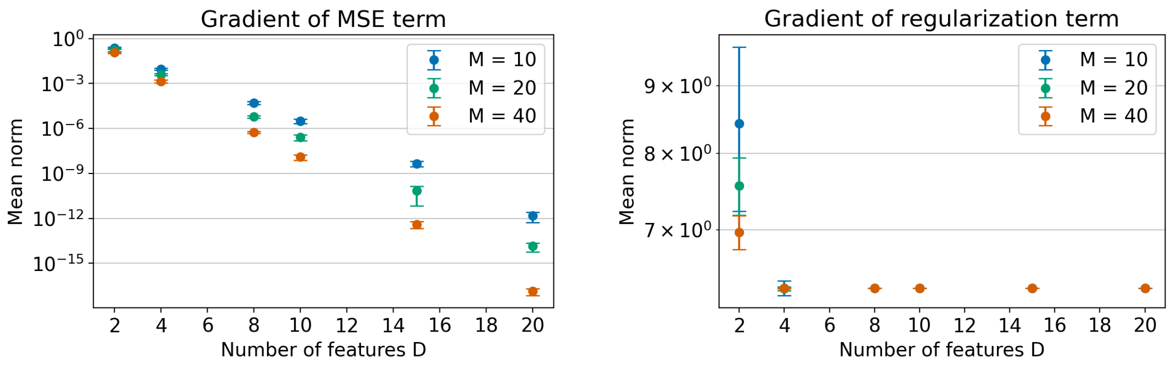


Figure 5-1: Mean norm of the partial derivatives with respect to the factor matrices for the two terms of the loss function, see Eq. (5-2). The dot is the mean of the mean norms and the error bar depict the standard deviation of the different experiments. The magnitude of the gradient of the MSE term is heavily dependent on the number of features, but this is not the case for the regularization term.

For the MSE term, the mean norm of the gradient decreases significantly when the number of features increases. In addition, the feature map order has an influence as well, but this influence relatively smaller. For double the number of features the decrease is substantially larger than for double the feature map order. This is expected, as D has an exponential contribution as shown in Eq. (5-1). M just influences the values of the individual components and thus has a smaller impact.

It can be seen that for a larger number of features the mean norm of the MSE gradient becomes extremely small. Moreover, this is only the norm of the gradient, so the gradient itself likely contains much smaller values. Furthermore, only up to twenty features were used for this experiment while a data set could have a lot more features. Given the visible trend, the MSE gradient would then be even smaller.

For the regularization term the number of features has no influence, except in the case of two dimensions. This conforms with the expectation that M and D have less influence on the regularization term due to the normalization of the initial weights. Furthermore, it was found in additional tests that the CP-rank does influence the mean norm.

There is likely still an influence in the case of two features, because in that case there are only two factor matrices. As a result, when the gradient is computed the Hadamard sequence only contains a single term. It is reasoned that the initialization, therefore, has a larger influence. The influence of the random initialization likely leads to the larger variance in the mean norm. Furthermore, it would explain why M, which affects the initial weights, has a substantial effect when only two features are used.

It is noted that a different initialization could be applied which might lessen the issue. For example, a different initialization could result in a reduced dependency on M. Nevertheless, it is expected that this will not remove the issue, since it is inherent to the model structure. The model consists of a large number of products, so when values smaller than one are used this leads to exceedingly small final values. On the other hand, when values larger than one are used the final values will blow up. Here values smaller than one are used, for example in the feature map. Therefore, it is reasoned that this issue of diminishing gradient values is inherent to the model structure.

Since the gradient can contain such small values, it was checked whether the floating point precision of the implementation has an effect. By default JAX uses 32-bit floating point precision, so it was studied if the training results were different for 64-bit precision. This analysis is presented in Section C-4. It was found that for the all-at-once optimization methods there is a notable difference, but not for Alternating Least Squares (ALS). Moreover, in some cases the 32-bit precision resulted in NaN's[1], so the training failed. Therefore, it was decided to use 64-bit precision throughout this thesis at the expense of an increase in computation and memory load.

Besides the influence on the step size, another noteworthy outcome of this result is the relation between the number of features and a suitable value for λ. The total gradient consists of the combination of the MSE term and the regularization term. The first determines the step direction to minimize the error and the latter enforces a smaller norm. As a result, the gradient of the MSE term should be of similar or larger order of magnitude than the regularization term. Otherwise, gradient will solely point in the direction that minimizes the norm and not take error minimization into account. To achieve this, the regularization term has to be scaled appropriately with the regularization parameter λ. The required amount of scaling is determined by the order of magnitude of the MSE gradient. Thus, a suitable value for λ is linked to the number of features.

Nevertheless, an appropriately scaled regularization gradient does not increase the overall magnitude of the gradient. Hence, a suitable step size still needs to be determined. Since the value of the gradient decreases with the number of features, naturally a large step size might be required for data that has more features. However, this is not as straightforward as simply using a larger step size. When the weights are near an minimum smaller steps are desired as to not overshoot the optimum. In Figure 4-4 it was shown that a too large step size can lead to instability. A solution could be to choose a varying step size that is larger at first and then decreases as a function of the number of iterations. Alternatively, at each iteration the optimal step size can be computed. This is called line search.

5-1-2 Line search

In the previous chapter the all-at-once optimization framework was introduced as well as the SteGD method. Here it was assumed that a fixed step size α is used for all the iterations. In the previous section it was shown that this can be problematic due to small gradient values.

Alternatively, the optimal step size for the current iteration could be computed. In other words, given the current weights and step direction the optimal step size can be computed by minimizing the loss function with respect to the step size. This is called line search or specifically exact line search, since an exact solution is determined. With the CPD weight update rule from Eq. (4-2) and the loss function given in Eq. (3-12) the following optimization

[1]NaN: Not a Number; this means that the training has failed

problem can be formulated:

$$\min_{\alpha} \quad h(\alpha) = \frac{1}{N}\sum_{n=1}^{N}\left(y_n - \left(\mathop{\circledast}_{d=1}^{D}\left(\mathbf{z}(x_n^{(d)})^\top \mathbf{W}^{(d)} + \alpha \mathbf{z}(x_n^{(d)})^\top \mathbf{P}^{(d)}\right)\right)\mathbf{1}_R\right)^2 \\ + \lambda \mathbf{1}_R^\top \left(\mathop{\circledast}_{d=1}^{D}\left(\mathbf{W}^{(d)} + \alpha \mathbf{P}^{(d)}\right)^\top \left(\mathbf{W}^{(d)} + \alpha \mathbf{P}^{(d)}\right)\right)\mathbf{1}_R. \tag{5-3}$$

For this thesis an expression was derived for the exact solution of this optimization problem.

First, a short literature overview of line search applied to a CPD will be presented. Then, the expression for the solution of the line search problem will be presented. It is shown with experiments in a proof of concept that the Line search method in combination with gradient descent can be applied for learning with CPDs. Finally, the downsides of the Line search method, in the particular context of learning with CPDs, will be discussed and demonstrated with experiments.

CPD line search in the literature

Line search has been applied to CPD optimization before in the literature. It is noted that in the literature the term line search is occasionally used to refer to updates with a fixed step size as well, as in [50, 51]. However, in this thesis line search will only refer to exact line search where an optimal step size is computed. In all the cases that were found in the literature, line search is applied to the problem of fitting a CPD to a known tensor. In [50] line search is combined with ALS to approximate real-valued tensors and in [52] this is extended to complex-valued tensors. The Line search method derived in [50] is combined with the Gauss-Newton step direction in [53].

The expressions for the line search solution that are presented in the cited papers are all for a 3-way CPD. No general expression for the line search solution for any number of dimensions is given. In [50] line search is also applied to a 4 dimensional problem, but no formulation for the solution of the line search problem is presented.

So, line search has so far only been applied in the context of fitting a CPD to a known tensor and within in that context no general formulation of the line search solution has been presented.

Expression for line search solution

Here the expression for the solution of the optimization problem of Eq. (5-3) will be presented. As it turns out, $h(\alpha)$ is a $2D$ order polynomial in α:

$$h(\alpha) = \sum_{i=0}^{2D} h_i \alpha^i. \tag{5-4}$$

As a result, the solution can be obtained by computing the roots of the derivative of $h(\alpha)$ and evaluating the function h for these roots to see which root results in the minimal value. Taking the derivative of a polynomial is straightforward. The expression of the solution therefore focuses on the computation of the coefficients h_i.

The computation of the coefficients is split up into two parts. The coefficients of the MSE term $\omega(\alpha)$ and the regularization term $k(\alpha)$ are computed separately and then combined such that $h(\alpha) = \omega(\alpha) + k(\alpha)$.

The coefficients are computed by recursively expanding a polynomial that consists of many terms. To illustrate this approach as small example is first given. Then the expressions for the coefficients of $k(\alpha)$ and $\omega(\alpha)$ are presented. The full derivation of the line search solution is given in Section A-3-1.

Recursive expansion of a polynomial A polynomial of several terms can be expanded recursively. To demonstrate this a toy-example will be used. Given two second order polynomials s_1 and s_2

$$s_1 = a_0 + a_1\alpha + a_2\alpha^2, \quad s_2 = b_0 + b_1\alpha + b_2\alpha^2, \tag{5-5}$$

the multiplication of these polynomials results in a fourth order polynomial t. Since t is of order four, $t = \sum_{i=0}^{4} u_i\alpha^i$, there are five coefficients that need to be computed:

$$\begin{aligned} t &= s_1 s_2 \\ &= (a_0 + a_1\alpha + a_2\alpha^2)(b_0 + b_1\alpha + b_2\alpha^2) \\ &= a_0b_0 + (a_0b_1 + a_1b_0)\alpha + (a_0b_2 + a_1b_1 + a_2b_0)\alpha^2 + (a_1b_2 + a_2b_1)\alpha^3 + a_2b_2\alpha^4 \\ &= u_0 + u_1\alpha + u_2\alpha^2 + u_3\alpha^3 + u_4\alpha^4. \end{aligned} \tag{5-6}$$

The coefficients u_i are a function of the coefficients a_i and b_i. This is called the Cauchy product of power series [54]:

$$u_i = \sum_{m+n=i} a_m b_n. \tag{5-7}$$

To compute the coefficients of a polynomial $q = s_1s_2s_3$ where $s_3 = c_0 + c_1\alpha + c_2\alpha^2$, the coefficients of t can be reused:

$$\begin{aligned} q &= s_1s_2s_3 = ts_2 \\ &= (u_0 + u_1\alpha + u_2\alpha^2 + u_3\alpha^3 + u_4\alpha^4)(c_0 + c_1\alpha + c_2\alpha^2) \\ &= u_0c_0 + (u_0c_1 + u_1c_0)\alpha + (u_0c_2 + u_1c_1 + u_2c_0)\alpha^2 + (u_1c_2 + u_2c_1 + u_3c_0)\alpha^3 \\ &\quad + (u_2c_2 + u_3c_1 + u_4c_0)\alpha^4 + (u_3c_2 + u_4c_1)\alpha^5 + u_4c_2\alpha^6. \end{aligned} \tag{5-8}$$

Again the same pattern appears. The subscripts of the multiplied coefficients are equal to the corresponding order of α. Note that for the boundary coefficients, the first two and last two, the pattern is the same, but less terms need to be added. This illustrates how coefficients of a polynomial can be computed by recursively expanding the polynomial one term at the time.

Regularization term The objective is to compute the coefficients k_i such that $k(\alpha)$ can be expressed as

$$k(\alpha) = \sum_{i=0}^{2D} k_i\alpha^i. \tag{5-9}$$

The regularization term $k(\alpha)$ is given by

$$k(\alpha) = \lambda \mathbf{1}_R^\top \left(\mathop{\circledast}_{d=1}^{D} \left(\mathbf{W}^{(d)} + \alpha \mathbf{P}^{(d)}\right)^\top \left(\mathbf{W}^{(d)} + \alpha \mathbf{P}^{(d)}\right)\right) \mathbf{1}_R \tag{5-10}$$

$$= \lambda \mathbf{1}_R^\top \left(\mathop{\circledast}_{d=1}^{D} \left(\mathbf{A}_0^{(d)} + \mathbf{A}_1^{(d)}\alpha + \mathbf{A}_2^{(d)}\alpha^2\right)\right) \mathbf{1}_R, \tag{5-11}$$

$$\mathbf{A}_0^{(d)} = \mathbf{W}^{(d)\top}\mathbf{W}^{(d)}, \quad \mathbf{A}_1^{(d)} = \mathbf{W}^{(d)\top}\mathbf{P}^{(d)} + \mathbf{P}^{(d)\top}\mathbf{W}^{(d)}, \quad \mathbf{A}_2^{(d)} = \mathbf{P}^{(d)\top}\mathbf{P}^{(d)}. \tag{5-12}$$

The coefficients k_i are computed as follows:

$$k_i = \lambda \mathbf{1}_R^\top \mathbf{K}_i \mathbf{1}_R, \quad i = 0, \ldots, 2D, \tag{5-13}$$

where the matrices $\mathbf{K}_i$ are computed by recursively expanding the Hadamard product sequence as illustrated in the previous section. The only difference is that coefficients during the expansion are not scalars, but rather matrices.

As with any recursive formula, initial values are needed:

$$\begin{aligned}
\mathbf{K}_0 &= \mathbf{W}^{(1)\top}\mathbf{W}^{(1)} = \mathbf{A}_0^{(1)},\\
\mathbf{K}_1 &= \mathbf{P}^{(1)\top}\mathbf{W}^{(1)} + \mathbf{W}^{(1)\top}\mathbf{P}^{(1)} = \mathbf{A}_1^{(1)},\\
\mathbf{K}_2 &= \mathbf{P}^{(1)\top}\mathbf{P}^{(1)} = \mathbf{A}_2^{(1)}.
\end{aligned} \tag{5-14}$$

Then, the Hadamard product of the sequence can be expanded at a time using the coefficients of the current factor given by Eq. (5-14) and the coefficients of the polynomial, $\mathbf{K}_i$, that have been worked out so far. This corresponds to recursively looping over d from $d = 2$ to $d = D$. At each loop the new coefficients are computed as follows:

$$\begin{aligned}
\mathbf{K}_0^{new} &= \mathbf{K}_0 * \mathbf{A}_0^{(d)},\\
\mathbf{K}_1^{new} &= \mathbf{K}_0 * \mathbf{A}_1^{(d)} + \mathbf{K}_1 * \mathbf{A}_0^{(d)},\\
\mathbf{K}_{2d-1}^{new} &= \mathbf{K}_{2d-3} * \mathbf{A}_2^{(d)} + \mathbf{K}_{2d-2} * \mathbf{A}_1^{(d)},\\
\mathbf{K}_{2d}^{new} &= \mathbf{K}_{2d-2} * \mathbf{A}_2^{(d)},\\
\mathbf{K}_i^{new} &= \mathbf{K}_{i-2} * \mathbf{A}_2^{(d)} + \mathbf{K}_{i-1} * \mathbf{A}_1^{(d)} + \mathbf{K}_i * \mathbf{A}_0^{(d)}, \quad i = 2, \ldots, 2d-2.
\end{aligned} \tag{5-15}$$

Note again that for the first two and last two coefficients separate formulas are used, since less terms are involved. The result of this loop is a $2D$ polynomial with matrix coefficients. The final coefficients of $k(\alpha)$ are computed with Eq. (5-13).

MSE term The coefficients of the MSE term, $\omega(\alpha)$, can be computed in a similar fashion. Since the MSE term is the averaged sum over all the samples, the coefficients for each individual sample can be computed individually and then added:

$$\begin{aligned}
\omega(\alpha) &= \sum_{i=0}^{2D} \omega_i \alpha^i = \sum_{n=1}^{N} \omega_n(\alpha) = \sum_{n=1}^{N}\sum_{i=0}^{2D} \omega_{i,n}\alpha^i,\\
\omega_n(\alpha) &= \sum_{i=0}^{2D} \omega_{i,n}\alpha^i.
\end{aligned} \tag{5-16}$$

For a single sample $\{\mathbf{x}_n,\ y_n\}$, $\omega_n(\alpha)$ can be expanded as follows:

$$\begin{aligned}\omega_n(\alpha) = \frac{1}{N}y_n^2 - \frac{2}{N}y_n \left(\mathop{\circledast}_{d=1}^{D} \left(\mathbf{z}(x_n^{(d)})^\top \mathbf{W}^{(d)} + \alpha \mathbf{z}(x_n^{(d)})^\top \mathbf{P}^{(d)} \right) \right) \mathbf{1}_R \\ + \frac{1}{N} \left(\left(\mathop{\circledast}_{d=1}^{D} \left(\mathbf{z}(x_n^{(d)})^\top \mathbf{W}^{(d)} + \alpha \mathbf{z}(x_n^{(d)})^\top \mathbf{P}^{(d)} \right) \right) \mathbf{1}_R \right)^2 .\end{aligned} \tag{5-17}$$

The computation of each individual coefficient $\omega_{i,n}$ is split up into a linear part $l_{i,n}$ and quadratic part $q_{i,n}$:

$$\begin{aligned}\omega_{0,n} &= \frac{1}{N}(y_n^2 - 2y_n l_{0,n} + q_{0,n}), \\ \omega_{i,n} &= \frac{1}{N}(-2y_n l_{i,n} + q_{i,n}), \quad i = 1, \dots, D, \\ \omega_{i,n} &= \frac{1}{N}q_{i,n}, \quad i = D+1, \dots, 2D.\end{aligned} \tag{5-18}$$

The coefficients of the second term of Eq. (5-17) are denoted by $l_{i,n}$:

$$\sum_{i=0}^{D} l_{i,n}\alpha^i = \left(\mathop{\circledast}_{d=1}^{D} \left(\mathbf{z}(x_n^{(d)})^\top \mathbf{W}^{(d)} + \alpha \mathbf{z}(x_n^{(d)})^\top \mathbf{P}^{(d)} \right) \right) \mathbf{1}_R. \tag{5-19}$$

These coefficients are computed first, because they can be used in turn to compute the coefficients $q_{i,n}$. The coefficients are given by

$$l_{i,n} = \mathbf{l}_{i,n}\mathbf{1}_R, \quad i = 0, \dots, D, \tag{5-20}$$

where the vectors $\mathbf{l}_{i,n}$ are computed recursively. The components of the recursive formula are

$$\mathop{\circledast}_{d=1}^{D} \left(\mathbf{z}(x_n^{(d)})^\top \mathbf{W}^{(d)} + \alpha \mathbf{z}(x_n^{(d)})^\top \mathbf{P}^{(d)} \right) = \mathop{\circledast}_{d=1}^{D} \left(\mathbf{b}_{0,n}^{(d)} + \mathbf{b}_{1,n}^{(d)}\alpha \right), \tag{5-21}$$

$$\mathbf{b}_{0,n}^{(d)} = \mathbf{z}(x_n^{(d)})^\top \mathbf{W}^{(d)}, \quad \mathbf{b}_{1,n}^{(d)} = \mathbf{z}(x_n^{(d)})^\top \mathbf{P}^{(d)}, \tag{5-22}$$

and the initial coefficients are given by

$$\begin{aligned}\mathbf{l}_{0,n} &= \mathbf{z}(x_n^{(1)})^\top \mathbf{W}^{(1)} = \mathbf{b}_{0,n}^{(1)}, \\ \mathbf{l}_{1,n} &= \mathbf{z}(x_n^{(1)})^\top \mathbf{P}^{(1)} = \mathbf{b}_{1,n}^{(1)}.\end{aligned} \tag{5-23}$$

The recursion is similar to Eq. (5-15), only the coefficients are now vectors instead of matrices. The new coefficients are computed by looping over $d = 2, \dots, D$:

$$\begin{aligned}\mathbf{l}_{0,n}^{new} &= \mathbf{l}_{0,n} * \mathbf{b}_{0,n}^{(d)}, \\ \mathbf{l}_{d,n}^{new} &= \mathbf{l}_{d-1,n} * \mathbf{b}_{1,n}^{(d)}, \\ \mathbf{l}_{i,n}^{new} &= \mathbf{l}_{i-1,n} * \mathbf{b}_{1,n}^{(d)} + \mathbf{l}_{i,n} * \mathbf{b}_{0,n}^{(d)}, \quad i = 1, \dots, d-1.\end{aligned} \tag{5-24}$$

For the coefficients $q_{i,n}$ of the last term of Eq. (5-17) the coefficients $l_{i,n}$ are reused:

$$\begin{aligned}\sum_{i=0}^{2D} q_{i,n}\alpha^i &= \left(\left(\mathop{\circledast}_{d=1}^{D} \mathbf{z}(x_n^{(d)})^\top \mathbf{W}^{(d)} + \alpha \mathbf{z}(x_n^{(d)})^\top \mathbf{P}^{(d)}\right) \mathbf{1}_R\right)^2 \\ &= \left(\sum_{i=0}^{D} l_{i,n}\alpha^i\right)^2 .\end{aligned} \tag{5-25}$$

The coefficients $q_{i,n}$ can be computed by taking the convolution of the polynomial with coefficients $l_{i,n}$ with itself[2]. This is done in Numpy/JAX with the `convolve` command.

Total solution The coefficients k_i and ω_i are summed to obtain the coefficients h_i:

$$h_i = \omega_i + k_i, \quad i = 0, \ldots, 2D. \tag{5-26}$$

The solution to the line search problem can then be determined by taking the derivative of $h(\alpha)$ with respect to α and computing the roots. Since $h(\alpha)$ is a polynomial in α, its derivative can easily be computed. Furthermore, finding the roots of the resulting $2D - 1$ polynomial is a straightforward operation. It can be done with the `roots` command in Numpy/JAX.

After the roots have been found it needs to be evaluated which root does in fact minimize $h(\alpha)$. Since only real step sizes are valid, only the real roots need to be tested. Using the previously calculated coefficients h_i, the value of $h(\alpha)$ for each real root can quickly be computed. The real root that results in the lowest value of $h(\alpha)$ is the optimal step size at the current iteration. So, it is used to update the weights. The full algorithm of the Line search CPD weight update method is given in Algorithm A.4.

Note that throughout the computation of the coefficients a general step direction is used. Thus, it is possible to apply the Line search algorithm with any step direction assuming that this direction can result in an improvement. Furthermore, the derived expression can be used with any number of features.

Proof of concept for Line search with gradient descent

To investigate the applicability of the Line search method for learning with a CPD, the Line search method is combined with Steepest Gradient Descent (SteGD). So, the negative all-at-once gradient derived in Section 4-2 is used as the step direction.

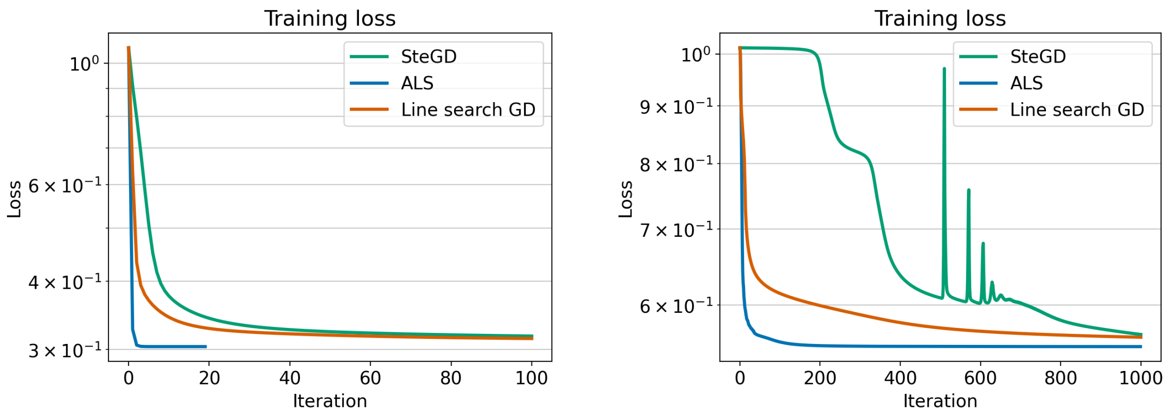
For this proof of concept the same data set, experiment setup and hyperparameters are used as described in Section 4-3-1. Only full-batch methods are used in this case. For the Airfoil data set a large number of iterations is used to allow for convergence. The training losses are shown in Figure 5-2.

The Line search method works well compared to the standard SteGD in both cases. It must be noted that for the latter the step size was chosen naively. Still, the Line search method has removed the problem of selecting a suitable step size. Moreover, the Line search method

[2]See https://nl.mathworks.com/help/matlab/ref/conv.html.

does not show any instabilities during training like SteGD which is an advantage. This is expected, since the Line search method cannot increase the training loss due to the inherent working of the method. A step size that increases the loss will never be selected. The Line search method is in that sense similar to ALS, because for each weight update an optimization problem is solved.

Nevertheless, in both cases the Line search method is not able to reach the same performance as the ALS method. Moreover, the Line search method takes long to converge, especially for the Airfoil data set. It might not be possible to run an experiment for 1000 iterations, especially for large data sets.



(a) Banana data set. **(b)** Airfoil data set.

Figure 5-2: Training loss for different methods on two data sets. The Line search method speeds up the training compared to SteGD.

The final training and validation losses are shown in Table 5-1. It can also seen that the Line search method achieves a similar final performance as SteGD. Thus, it is likely not able to improve the performance in terms of loss, but simply speed up the training.

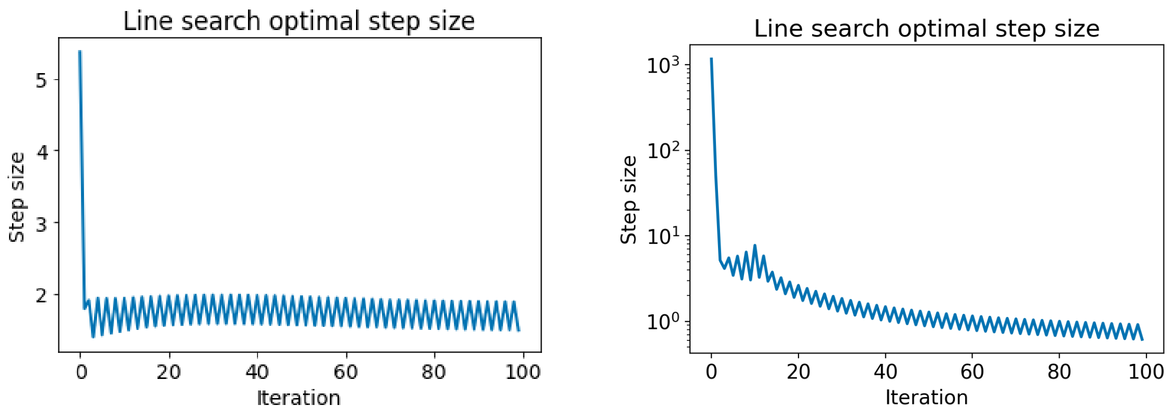
Table 5-1: Final losses for different methods after training on the Banana and Airfoil data set. The Line search method does not notably improve the final losses of the SteGD method.

	Banana		Airfoil	
Method	Training	Validation	Training	Validation
SteGD	0.317	0.321	0.565	0.476
ALS	**0.303**	**0.296**	**0.551**	**0.463**
Line search Gradient Descent	0.314	0.312	0.562	0.476

To further analyze the Line search method, the computed optimal step sizes are investigated. The optimal step sizes during training on the two data sets are shown in Figure 5-3. It can be seen that the step size is significantly larger at first and then decreases rapidly. F or the Airfoil data set the difference between the first and last optimal step size is especially large. These results further suggest that selecting and using a fixed step size can be complicated and inefficient.

The sharp decline in the optimal step size magnitude might have an undesired effect. Since the first step size is so large, the first step and thus the first gradient have a big impact. The first gradient is determined by the initial weights. The large initial step size can, therefore, indicate that the long term performance of the Line search method is heavily influenced by the initial weights. This and other challenges of the Line search method will be discussed in the next section.

Finally, it was also tested whether Line search can be combined with mini-batch gradient descent. The results for these experiments are shown in Section B-3. These results indicate that it is possible to use the Line search method with mini-batch gradient descent. The mini-batch method is able to converge faster at the start of the training. However, later in the training process does the training loss of the mini-batch Line search method oscillate more. This is most likely because the Line search method overfits on the mini-batch data and this, thus, leads to a worse overall. Using mini-batches with Line search could be beneficial to escape local minima which the full-batch method is not be able to do. In future research the mini-batch Line search method could be further investigated.



(a) Banana data set. **(b)** Airfoil data set.

Figure 5-3: Optimal step size determined by the Line search method while training on two different data sets. For the Airfoil data set only the first 100 iterations are shown. The optimal step size sharply decreases in magnitude, so the initial step is of great importance. Furthermore, it indicates that a fixed step size is inefficient.

Challenges of line search

In this section two challenges of the Line search method are discussed. First, the influence of the weight initialization is studied. Secondly, the complexity for the computation of the optimal step size is derived theoretically and tested in practice.

Initial weights Due to the large initial step size for the Line search method, initialization of the weights can have a large effect. While the initialization always has an effect, a large effect would be undesirable. It would be necessary to train the model more times to ensure that the achieved performance is not hindered by the choice initial weights. When the performances for different initial weights are close to each other, this would not be so necessary.

To investigate the dependence on the initial weights, the Line search method is trained ten times on the same data set for different initial weights. Additionally, the ALS method is trained using the same initial weights to asses the potential performance is for those particular initial weights. The training is done on the Airfoil data set with the hyperparameters described in Section 4-3. The weights are initialized as described in Section 4-3 in all the experiments.

The training losses are shown in Figure 5-4. The final training and validation losses are shown in Table 5-2. The standard deviation of the training losses of the Line search method is an order of magnitude larger than for ALS. This suggests that the performance of the Line search method does indeed depend more on the initial weights. The standard deviation for the validation loss is similar to that of ALS. This indicates that the Line search method finds different local minima in the training data depending on the initial weights, but these local minima generalize to a similar validation loss. Overall, the Line search method does seem to indeed depend more on the initial weights, but when the validation loss is the only performance criteria this does not have to be a problem. In future research this could be studied in more detail. For example, it could be studied whether the dependence on the initialization is similar for different initialization methods.

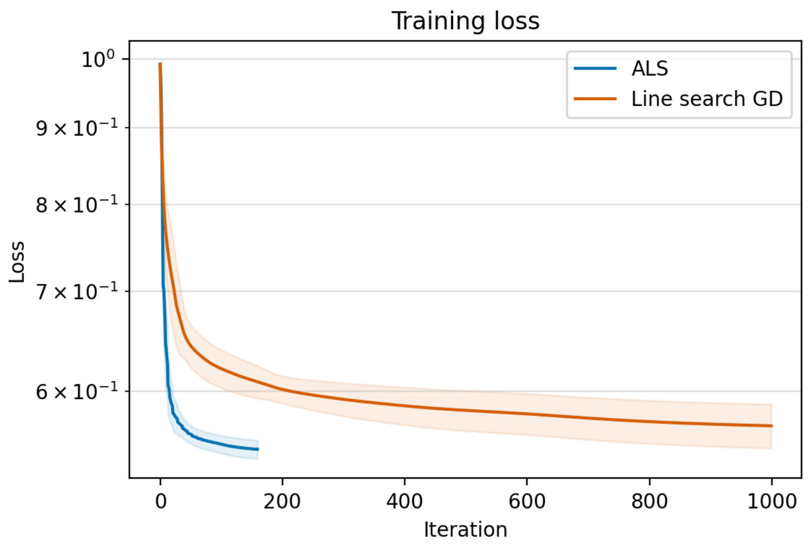


Figure 5-4: Mean training loss (line) and one standard deviation (shaded area) during training on the Airfoil data set for different methods. The larger standard deviation indicates that the Line search method is more dependent on the initial weights than ALS.

Complexity The computation of the optimal step size adds both computational and memory complexity to the weight update. The derivation of the additional complexity is given in Section A-3-2. For the most memory efficient implementation the added memory complexity is of order $\mathcal{O}(DR^2)$. This is due to the memory required to store the intermediate values of the coefficients during the recursion. The added computational complexity is of order $\mathcal{O}(DMRN)$ assuming $M \gg D$, $N \gg R$. Since the complexity of the computation of the gradient is also of order $\mathcal{O}(DMRN)$, the overall theoretical computational complexity does not increase.

Table 5-2: Mean final loss and one standard deviation for different methods on the Airfoil data set. The validation standard deviations of the Line search method and ALS are of similar magnitude. This suggests that the validation performance of the Line search method is not as dependent on the initial weights as the training performance.

	Airfoil	
Method	Training	Validation
ALS	**0.549** (0.0077)	**0.581** (0.0086)
Line search Gradient Descent	0.569 (0.0192)	0.590 (0.0096)

However, it was found in practice that the Line search method slows down the training considerably. To illustrate this the run times of one update with the SteGD method and the Line search gradient descent method are evaluated. This is done for the parameters $R = 10$, $M = 20$ and $D = 10$ for randomly generated data and two different numbers of samples. The run time is evaluated 11 times, but the first timing is discarded as was done previously. The results are shown in Table 5-3. It can be seen that it takes significantly longer to perform one Line search update compared to one SteGD update. It must be noted that Line search run time includes the computation of the gradient which is needed for the update. So, it is expected that the Line search update takes longer.

Although the theoretical computational complexity is of the same order, the Line search method takes longer in practice. Given the two algorithms it is reasoned that this is due to difference in number of operations. The SteGD update is composed of simple matrix products for which the required number of operations is close to the corresponding theoretical complexity. On the other hand, for the Line search method this deviates more. For example, the computation of $\mathbf{K}_i^{new}$ in Eq. (5-15) has a theoretical complexity of $\mathcal{O}(R^2)$, but $3 \cdot R^2$ operations are required for the Hadamard products and another $\cdot R^2$ for the two additions. This phenomenon occurs in several equations and accumulates due to the recursion. As a result, the total number of operations is far larger in practice, but the theoretical complexity still scales with $\mathcal{O}(DMRN)$.

Table 5-3: Mean run time and on standard deviation for one weight update for the two methods for two different data set sizes. The Line search weight update takes significantly longer compared to SteGD. This implies that the Line search method is too inefficient to be applied to large scale learning problems.

	Weight update time (s)	
	$N = 1000$	$N = 10000$
SteGD	**0.00170** (0.000169)	**0.0407** (0.00830)
Line search Gradient Descent	0.0120 (0.00220)	0.137 (0.0085)

Additionally, the training run time for SteGD and Line search were timed and it was found that on the Airfoil data set Line search took 13 times longer than SteGD. For the Banana data set this was more than a 100 times. This illustrates that the Line search method does indeed slow down the training considerably. Furthermore, it means that the Line search method can be infeasible for large scale learning due to the long run time.

In future research it could be investigated whether the Line search run time can be improved. A more efficient computation of the coefficients could be studied. Another bottleneck is that the Line search method could not easily be run on a GPU, since the `roots` JAX command is not yet implemented for a GPU. When it becomes possible to use a GPU it could be analyzed whether this reduces the run time enough to make large scale Line search learning feasible.

5-1-3 Adam

Besides a fixed step size and line search various others step size schemes that are combined with gradient descent have been derived by the machine learning community. One scheme that is very popular at the moment is the Adam method [8]. The name Adam comes from adaptive moment estimation. Adaptive means that the step size is not fixed, but variable. Contrary to the Line search method, the step size is not based on the loss function, but is a function of the gradients.

First, the Adam method is shortly presented and its characteristic are analyzed. Then, the results of a proof of concept for the Adam method are presented.

The Adam method

The most significant difference between the Adam method and the two previously used methods, is that the Adam method determines a step size for each individual weight. Based on the first and second moment of the gradients a step size is determined for each individual weight at each iteration. By taking these moments into account the Adam method determines an appropriate step size based on the change in the gradient compared to previous iterations. When the gradient is similar for several iterations the step sizes increase. However, when gradient changes the step sizes decrease. The Adam method algorithm is shown in Algorithm A.5.

The Adam method introduces three new hyperparameters. The step size α_{adam} is not new, but has a different role compared to the SteGD method. The parameters β_1 and β_2 are new and they determine the influence of previous gradients on the current step. The authors recommend the values $\beta_1 = 0.9$ and $\beta_2 = 0.999$. For the parameter ϵ the authors recommend the value 10^{-8}.

With $\epsilon = 0$ the authors define the effective step as $\Delta_k = \alpha_{adam}\hat{\mathbf{m}}_k/\sqrt{\hat{\mathbf{v}}_k}$. When the gradient has remained similar for long, $\hat{\mathbf{m}}_k \approx \pm\sqrt{\hat{\mathbf{v}}_k}$. The parameter α_{adam} determines in this case thus the magnitude of the effective step. Moreover, it can be seen that the Adam method performs a sort of scaling and can thus likely handle small gradients well.

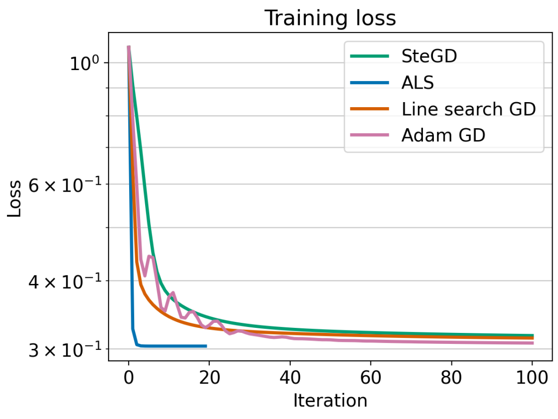
The Adam method uses the gradient, but this gradient can be computed with the full-batch or a mini-batch method. It is expected that the Adam method will also be effective with mini-batch gradient descent, since this also applied in [8]. Moreover, all the operations in the Adam method are element-wise. As a result, it is possible to work with the CPD factor matrices directly without vectorizing them first.

The theoretical computational complexity of the Adam method without the computation of the gradient is $\mathcal{O}(DMR)$, since only element-wise products of the gradient need to be computed. The memory complexity is also $\mathcal{O}(DMR)$ which corresponds to the memory required to store $\mathbf{m}_k$ and $\mathbf{v}_k$. Thus, the Adam method barely adds any computational or memory complexity compared to SteGD which makes it a very efficient method.

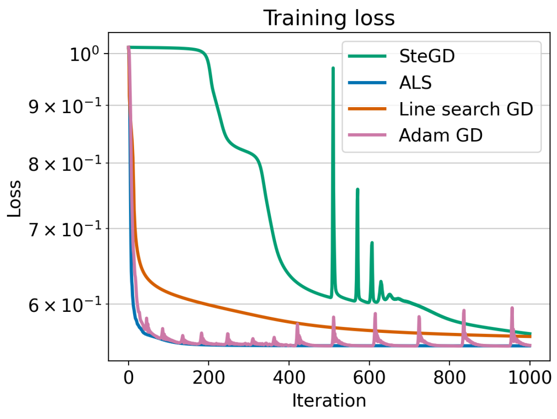
Proof of concept for the Adam method

In this section it is studied whether the Adam method can be effectively applied to the CPD learning problem. Again the same proof of concept setup as described in Section 4-3-1 is used. Only for the Adam method a different step size is used. The base step size parameter $\alpha_{adam} = 0.1$ is used, while for the SteGD $\alpha = 1$ is used. For the other hyperparameters of the Adam method the default values suggested by the authors of the method are used. Furthermore, full-batch gradient descent is used. In Figure 5-5 the training losses for the Banana and Airfoil data sets are shown and in Table 5-4 the final losses are given. From the results it concluded that the Adam method can achieve a similar performance as the ALS method and can outperform the line search method. Moreover, the Adam method is able to converge quickly, even when full-batches are used.

The Adam method also show instabilities during training, similar to SteGD. It seems that they especially occur when the training has almost converged. In Section B-2 it is shown that SteGD likely can overshoot a minimum, so it is probable that the same happens with the Adam method. To prevent this an early stopping criteria could be implemented.



(a) Banana data set.



(b) Airfoil data set.

Figure 5-5: Training loss for different methods on two data sets. The Adam method reaches a similar convergence and final loss as ALS.

Table 5-4: Final losses for different methods after training on the Banana and Airfoil data set. For the Adam method with the Airfoil data set the minimum validation loss in last 150 iterations is given, since the training was cut-off when the loss for the Adam method had just jumped as can be seen in Figure 5-5b. The results show that the Adam method can reach similar final losses as ALS.

	Banana		Airfoil	
Method	Training	Validation	Training	Validation
SteGD	0.317	0.321	0.565	0.476
ALS	**0.303**	**0.296**	**0.551**	0.463
Line search Gradient Descent	0.314	0.312	0.562	0.476
Adam	0.307	0.302	**0.551**	**0.462**

The Adam method is studied in more detail by analyzing the step sizes that it produces. The effective step was previously defined as $\Delta_k = \alpha_{adam}\hat{\mathbf{m}}_k/\sqrt{\hat{\mathbf{v}}_k}$, but now ϵ will be added, since this is also added in practice. So, the new effective step is defined as $\hat{\Delta}_k = \alpha_{adam}\hat{\mathbf{m}}_k/(\sqrt{\hat{\mathbf{v}}_k}+\epsilon)$. Note that the effective step is in this case a tensor, because the gradient is as a tensor, $\hat{\Delta}_k \in \mathbb{R}^{D\times R\times M}$. Additionally, the step size $\hat{\alpha}_{d,m,r}$ for each weight is defined as the effective step for that particular weight divided by the gradient value for that weight:

$$\left(\hat{\Delta}_k\right)_{d,m,r} = \hat{\alpha}_{d,m,r}\mathcal{G}_{d,m,r} \rightarrow \hat{\alpha}_{d,m,r} = \left(\hat{\Delta}_k\right)_{d,m,r}/\mathcal{G}_{d,m,r},$$
$$d = 1,\ldots,D,\ m = 1,\ldots,M,\ r = 1,\ldots,R.$$

It is analyzed how the effective step and step size differ per weight during training on the Airfoil data set. The smallest and largest effective step and step size are determined at each weight update:

$$\min(|\hat{\Delta}_k|),\quad \max(|\hat{\Delta}_k|),\quad \min(|\hat{\boldsymbol{\alpha}}|),\quad \max(|\hat{\boldsymbol{\alpha}}|).$$

Here $\boldsymbol{\alpha} \in \mathbb{R}^{D\times R\times M}$ has $\hat{\alpha}_{d,m,r}$ as elements.

The results are shown in Figure 5-6. It can be seen that the difference between the smallest and largest values is multiple orders of magnitude. This further supports the conclusion that selecting a fixed step size is inefficient and ineffective. Furthermore, it shows that the Adam method is most likely so effective, because it can determine a step size for each weight individually.

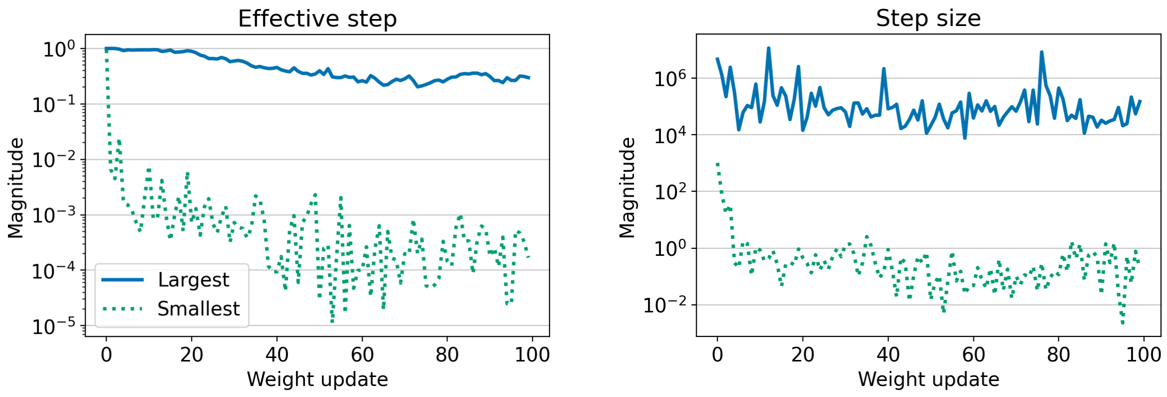


Figure 5-6: The largest (solid) and smallest (dotted) effective step and step size magnitude per weight update during training on the Airfoil data set. The difference between largest and smallest magnitudes indicates why the Adam method works so well and that a fixed step size is inefficient.

Finally, the Adam method combined mini-batch gradient descent is studied. The results are shown in Section B-3. From these results it is concluded that the Adam method can be effectively combined with mini-batch gradient descent. Moreover, using mini-batch gradient descent can speed up training significantly. Additionally, mini-batch gradient descent can be combined well with large scale learning, since the number of samples in a data set is irrelevant for mini-batch descent.

5-2 Weight initialization

The initialization of the weights can influence the training. So far, the weights have been initialized from a uniform distribution and were normalized. In this section other possible initialization methods will be analyzed. First, the methods will be introduced and then it is investigated what their effect on the training is for the different optimization methods.

5-2-1 Initialization methods

There are countless possible ways to initialize weights. For example, initial values can be drawn from any random distribution or the initial weights can be based on an initial estimate of the solution. Four possible methods will be studied; initial weights drawn from an uniform distribution with and without normalization, and initial weights drawn from a normal distribution with and with normalization. In Section 6-2-4 it is investigated whether the solution of one method can be used as the initial weights for the next method and thus to combine methods.

The uniform weight initialization was described earlier in Section 4-3-1. Random uniform weights can simply be initialized in the same fashion by skipping the normalization step. When the normalization is not applied the bounds of the uniform distribution become a hyperparameter. Since the values are no longer scaled, the bounds determine the highest and lowest values the initial weights can attain.

The normal weight initialization procedure is very similar to the uniform initialization procedure. Instead of a uniform distribution, the weights are drawn from a normal distribution $\mathcal{N}(\mu, \sigma^2)$. The total initialization procedure including the normalization is given in Algorithm A.7. So, this initialization method has two hyperparameters, μ and σ. Zero mean, $\mu = 0$, will be used, because negative and positive weights are assumed to be equally likely. So, during the initialization there is no preference for either one. The standard deviation does, therefore, determine what are likely the highest and lowest values the initial weights attain.

Finally, it is noted that initialization with a normal distribution is more logical. It is assumed that the true weights of the model do not contain a certain cut-off like the uniform distribution has. Thus, an initialization with a normal distribution would most likely better mimic the distribution of the true model weights.

5-2-2 Effect on performance

The effect of the initialization method on the performance is investigated by analyzing the effect on the training loss during training and the final losses. This is done for the ALS method, the Line search method and the Adam method. It was chosen not to test this on the SteGD method, because it was already identified that this method works only well for a limited number of features.

A model is trained on the Airfoil data set ten times for each initialization method. The same hyperparameters as in Table 4-1 are used. For the uniform initialization without normalization the bounds $[-0.25,\ 0.25]$ were used and for the normal initialization without normalization

$\sigma = 0.5$ was used. These bounds were used since they create a reasonable spread around zero to draw the weights from, but don't result in a CPD with a very large norm. A smaller norm is desired, because otherwise the regularization term is dominating at the beginning of the training. The first iteration would then only be an update to reduce the norm.

The results are shown Figure 5-7 and Table 5-5. The performance of the Line search method is the most affected by the choice of the initialization method. In Section 5-1-2 it was already discussed that the Line search method is prone to be influenced by the initial weights. These results further confirm this. The final losses show that the uniform weight distribution results in poorer performance for the Line search method.

For ALS and the Adam method the initialization method has little influence. There is little difference during the training process. Furthermore, the final losses also do not differ significantly. So, the choice of initialization method is less important for them.

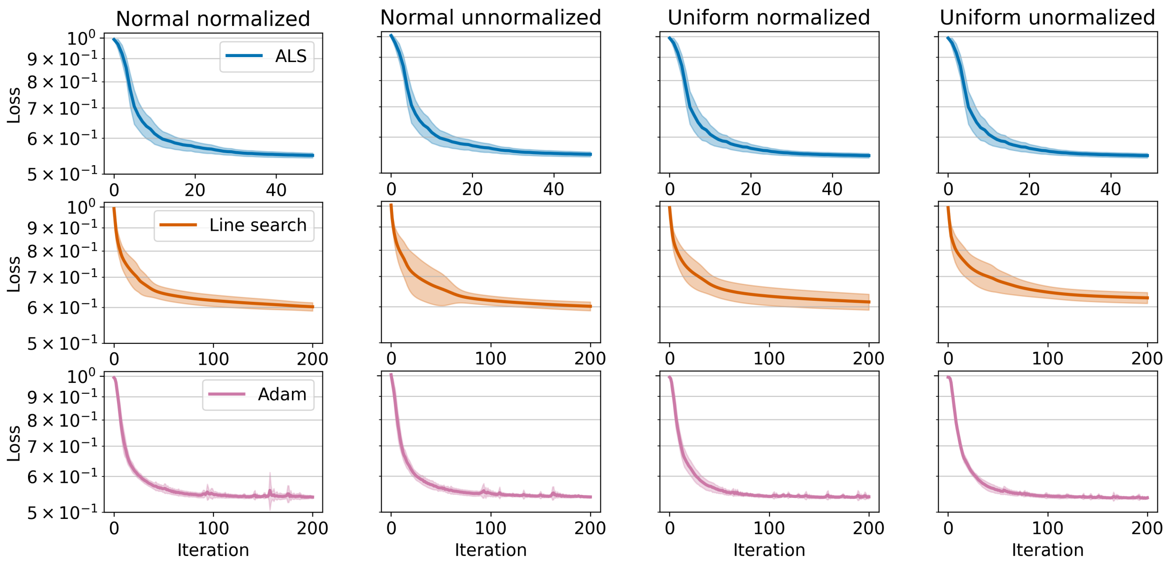


Figure 5-7: Mean training loss (line) and one standard deviation (shaded area) during training on the Airfoil data set for different optimization methods using different initialization methods. The subfigures share the same y-axis. The larger standard deviation indicates that the Line search method is the most influenced by the initialization method. For the other two methods the initialization seems to have no noteworthy impact.

5-3 Normalization of weights

In Section 2-2-1 it was mentioned that a CPD can be normalized which removes the scaling indeterminacy. It is thus possible that the normalization improves the training performance. First, the normalization approach will be discussed and then the effect on the performance is studied.

Table 5-5: Mean final loss and one standard deviation for different optimization and initialization methods. norm. stands for normalized, unnorm. stands for unnormalized. For the Line search method the mean final losses differ considerably, so the initialization has a large impact. For the other two methods the difference is less than five thousandths, thus the initialization is of less importance.

	ALS		Line search	
Initialization method	Training	Validation	Training	Validation
Normal norm.	0.550 (0.0005)	0.575 (0.0081)	0.602 (0.0485)	**0.612** (0.0107)
Normal unnorm.	0.550 (0.0083)	0.575 (0.0081)	**0.601** (0.0368)	**0.612** (0.0140)
Uniform norm.	**0.546** (0.0005)	**0.574** (0.0092)	0.614 (0.0499)	0.626 (0.0250)
Uniform unnorm.	**0.546** (0.0000)	**0.574** (0.0092)	0.627 (0.0410)	0.635 (0.0247)

	Adam	
Initialization method	Training	Validation
Normal norm.	0.540 (0.0166)	**0.568** (0.0038)
Normal unnorm.	0.540 (0.0377)	**0.568** (0.0053)
Uniform norm.	0.540 (0.0162)	0.571 (0.0100)
Uniform unnorm.	**0.538** (0.0044)	0.569 (0.0057)

5-3-1 Normalization method

A CPD can be normalized by extracting the norm of the rank-one tensor components and storing them in the separate vector $\boldsymbol{\mu}$, see Eq. (2-22). To use a normalized CPD for CPD learning it has to be ensured that the CPD stays normalized after each weight update. One option would be to impose constraints such that the factors remain normalized. The vector $\boldsymbol{\mu}$ would then become weights of the optimization problem as well and would be updated to achieve the needed norm. This, however, would in a constrained optimization problem in which the constraints cannot be substituted into the loss function. This is outside the scope of this thesis.

Alternatively, the weights can be updated as before. The weights are then normalized again after each update and the norm is transferred to $\boldsymbol{\mu}$. In this case the vector $\boldsymbol{\mu}$ is not a weight, so it is not updated like the other weights. This is done to prevent a double update of $\boldsymbol{\mu}$, namely a gradient based update and the update due to normalization. It was found by experiment that the values of $\boldsymbol{\mu}$ explode with a double update. The general weight update with normalization is shown in Algorithm A.8 and the normalization procedure itself is given in Algorithm A.9.

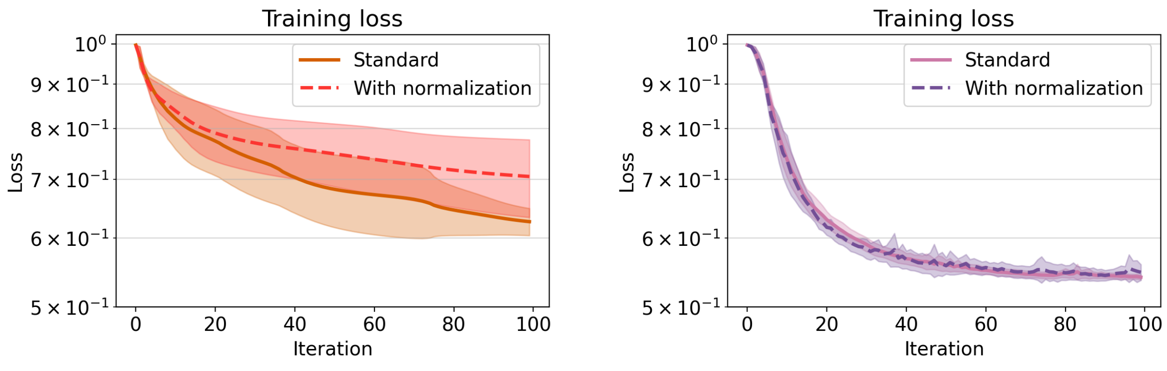
To compute the gradient of the loss function with $\boldsymbol{\mu}$ as an additional parameter automatic differentiation (AD) was used. This was done to quickly investigate the effect of the normalization without having to derive the analytical expression for the gradient with $\boldsymbol{\mu}$ taken into account. The computation of the optimal step size with Line search also needs to be adjusted to take $\boldsymbol{\mu}$ into account. Theses adjustments are given in Section A-6.

Overall, this normalization approach is more of a brute force normalization approach. It ensures that after a weight update the CPD is normalized, but it does not guarantee so during the weight updates. Since this approach is constraint-free and can thus be applied with the earlier introduced methods, it was opted to use this approach to study the effect of normalization.

5-3-2 Effect on performance

The effect of intermediate normalization was investigated for the Line search and the Adam method. Each method is trained on the Airfoil data sets tens times with and without normalization. The same hyperparameters as in Section 4-3 are used. The results are shown in Figure 5-8. The Line search method with normalization performs significantly worse than the method without normalization. For the Adam method there is no notable difference. It is therefore concluded that it is not beneficial to apply normalization in between weight updates.

The large difference for the Line search method could be due to the optimal step size. Since the Line search method takes an optimal step at each iteration, it could be more thrown off by the normalization step in between iterations.



(a) Line search Gradient Descent. **(b)** Adam Gradient Descent.

Figure 5-8: Mean training loss (line) and one standard deviation (shaded area) during training on the Airfoil data set for different optimization methods without (standard) or with normalization in between weight updates. The difference in the mean training loss indicates that the Line search method performs worse with normalization. For the Adam method there is not notable difference. So, overall the application of normalization is not preferred.

5-4 Conclusion: Best all-at-once optimization method

In this and the previous chapter various all-at-once optimization methods based on gradient descent have been presented. In this section one method is selected that is applied to various learning problems in the next chapter to compare all-at-once optimization with ALS. Additionally, the main findings of this chapter are summarized.

In the previous chapter the SteGD method was introduced. It was found that the SteGD method works, but it is difficult to use due to the fixed step size. In this chapter the step size was further analyzed. It was found that the magnitude of the gradient of the MSE term decreases significantly for an increasing number of features. For the regularization term this is not the case. Therefore, finding a suitable step size is linked to the number of features. Moreover, the regularization parameters needs to be selected carefully to ensure an appropriate balance between the gradient of the MSE and regularization term and to avoid that the regularization term dominates.

To overcome the step size challenge, the Line search method and the Adam method were introduced. The expression for the exact line search solution that was derived for this thesis was presented. It was shown that the Line search method can increase the performance compared to SteGD. However, the Line search method is limited by its longer run time which makes it, at the moment, unsuitable for large scale learning.

The Adam method barely adds any complexity. Furthermore, in the proof of concept it was shown that the Adam method can reach similar performance as the ALS method. Moreover, it was shown that the Adam method can be used with mini-batch gradient descent. Not only does the mini-batch method speed up training, it can easily be applied to large scale learning. Namely, the number of samples in a data set is irrelevant for the mini-batch method.

To conclude, the mini-batch Adam method is found to be the most effective and efficient all-at-once gradient descent optimization method. It can be applied to large scale learning problem and has shown to achieve similar performance as ALS in the first experiments. Since the initialization method was shown to not have a significant effect for the Adam method, normal normalized initialization will be used. It eliminates one hyperparameters, σ, and a normal distribution is more logical than a uniform distribution. Moreover, normalization in between weight updates also has no noteworthy effect and will therefore also not be used.

Summary of contributions

- It is shown that the norm of the gradient of the MSE terms, and therefore the values of this gradient term, decreases for an increasing number of features. For the regularization term this is not the case. The implications for this on the step size and regularization parameter are deliberated.
- An general expression for the exact solution of the CPD constrained kernel machine line search problem is derived.
- It is shown that the Line search method converges faster than SteGD but reaches a similar final loss. Moreover, it is found that the required additional computations slow down the Line search method to such an extent that it is infeasible for large scale learning.
- It is shown that the Adam method can reach a similar performance as ALS, both in terms of convergence and final loss. Furthermore, it is shown that the Adam method can effectively be combined with mini-batch gradient descent for the CPD learning problem.
- It is found that the Adam method is the best all-at-once optimization method in terms of convergence and the final training and validation loss compared to SteGD and the Line search method.

- It is shown that the Line search method is impacted by the weight initialization method. For the Adam method and ALS there is no notable difference for initialization methods that are studied
- It is shown that the normalization of the CPD in between weight updates has an averse effect for the Line search method and has no notable effect for the Adam method.

Chapter 6

All-at-once learning performance

In this chapter the performance of all-at-once optimization is studied. The performance of the mini-batch Adam method, in rest of this chapter simply called the Adam method, and Alternating Least Squares (ALS) are compared on several learning problems. First, the experiment set up is discussed, including the performance criteria that are used. Afterwards, the results of the experiments are presented. During the experiments it was found that the Adam method no longer works for a larger number of features. The results of this limiting effect is first presented. Then the general performance of the Adam method and ALS is compared. Thirdly, the training run times of the two methods for different CP-ranks and feature maps orders are studied. Finally, the performance of a combination of the methods is analyzed.

6-1 Experiment setup

The setup used for the following experiments is briefly described. First, the used data sets are introduced. Secondly, the experiment procedure is discussed. Thirdly, the performance criteria are introduced that are used to compare the two methods. Finally, the hyperparameters are discussed.

6-1-1 Data sets

Nine data sets are used for the different experiments. They are summarized in Table 6-1. The data sets have different characteristics and thus allow for different perspectives on the methods. In particular there are both relatively small and large data sets and they have different numbers of features. Moreover, both classification and regression tasks are included.

The SUSY data set [55] appears three times in the table. The full data set has 18 dimensions of which 8 represent low level features and 10 are high level features. The data sets with only low or high level features are used as well as the data set with all the features. For the HIGGS data set [55] only the high level features are used to limit the number of features.

The data is preprocessed in the same way as described in Section 4-3-1. The preparation of the data sets is discussed in more detail in Section B-7. Moreover, it is listed where the data can be obtained.

Table 6-1: Dimensions and learning task of the used data sets.

Data set	Number of samples N	Number of features D	Learning task
Elevators	16599	18	Regression
Protein	45730	9	Regression
Precipitation	628474	3	Regression
HouseElectric	2049280	10	Regression
SUSY Low	5000000	8	Classification
SUSY High	5000000	10	Classification
SUSY Full	5000000	18	Classification
Airline	5929413	8	Regression
Higgs High	11000000	7	Classification

6-1-2 Experiment procedure

For each data set and hyperparameters combination the models are trained ten times with different initial weights. The weights are initialized from a normal distribution and normalized. For each experiment the same initial weights are used for both methods. Furthermore, the data set is split up into a training and validation set based on the validation split. The methods use the same training and validation set, but for each experiment a different random split is made.

In each experiment a set number of iterations is. One iteration, as defined in Definition 4.1, is any number of updates until the full training data set is used once. Since mini-batch gradient descent is used with the Adam method, this implies that numerous updates are performed within a single iteration. For ALS one iteration corresponds to the update of one factor. Early stopping based on a convergence criterion is not used.

The experiments are run on the `dcscgpuserver4` of the Delft Center for Systems and Control (DCSC). The server has 4 Intel Xeon E7-4809 8-core CPUs, 256 GB memory, 2 TB SSD, 4 14 TB hard disks and a NVIDIA GeForce RTX3090 GPU. The implementations of both methods make use of jitting where possible and all experiments are run on the GPU with the help of JAX. The floating-point precision is set to 64-bits.

6-1-3 Performance criteria

Different performance criteria are used to compare the methods. Naturally, the training and validation loss are performance criteria. The training loss is evaluated over the whole training data set, so not a mini-batch. The validation loss is evaluated on the validation data set. The stochasticity of the Adam method can lead to an increase in the losses after reaching a minimum. It was, therefore, decided to use the minimum of the mean validation loss during training as the validation loss performance. Furthermore, the mean training loss at the iteration where the mean validation loss is minimal is used as the training loss performance. The validation loss is used to determine the final loss, since it represents the performance of the model on unseen data. It is a metric about the real-world performance of the model and is, therefore, in the machine learning community commonly used to compare models and methods. The training loss is used to compare the training progress of the two optimization methods, since the training loss reflects the ability of the method to solve the optimization problem.

The total training run time is also evaluated. The training run time is the run time it has taken to perform all the iterations. Additionally, the run time until convergence is also determined. The convergence is determined after training based on the mean training loss. Since early stopping is not implemented in the algorithms, the convergence is used as an indicator when an early stop could have happened. This notion of convergence is used to analyze how fast the methods reach convergence and thus which method is more efficient in terms of training run time until convergence.

The convergence is defined as follows:

Definition 6.1 (Convergence of mean training loss)**.** *The mean training loss of a method has converged at a particular iteration when the difference between the minimum mean training loss and the mean training loss at the current iteration is less than* $\varepsilon = 0.0005$*.*

This definition of convergence is used for several reasons. First, it can deal with plateaus in the loss during training, because it takes the minimum training loss into account. Secondly, the stochasticity of the Adam method is less of a problem with this definition. A difference in loss between consecutive iterations was also considered. However, this does not work well when the losses oscillate like with a stochastic method. By using the relation with the minimum loss the convergence indicates when the stochastic method has reached a similar loss as the minimum loss. Finally, the value $\varepsilon = 0.0005$ is used, because the losses are reported in thousandths. So, this value means that the minimum loss and loss at convergence are the same up to a thousandths without rounding taken into account. It is noted that the minimum losses do not need to coincide with the losses at convergence. Moreover, it is noted that this definition of convergence can only be applied after the experiments have been completed. So, although it can indicate when early stopping might have happened, this definition can in itself not be applied as an early stopping criterion. Finally, it is noted that other definitions of convergence could also be used and that they result in a different perspective of the results. Nevertheless, it was found that this definition of the convergence, and the resulting the training run times until convergence, is a good measure to illustrate the differences between the two methods.

The training run time until convergence is computed by multiplying the total training run time with the number of iterations until convergence and dividing by the total number of iterations:

$$\text{time}_{convergence} = \text{time}_{total} \cdot \frac{\text{iter}_{convergence}}{\text{iter}_{total}}. \tag{6-1}$$

The implementation of the methods has an effect on the run time. For example, the training loss can easily be evaluated in the ALS method by reusing the components of the algorithm. For the Adam method, on the other hand, the full training data set needs to be evaluated from scratch which enlarges the run time. So, the effect of the implementation should be taken into account when analyzing the run time.

6-1-4 Hyperparameters

The hyperparameters can affect the learning performance. However, for several hyperparameters it is assumed that they will not influence the relative performance of the methods. The same values are, therefore, used for all the experiments. This applies to the bound and the length-scale of the feature map and the validation split. The values $l = 0.1$, $U = 1$ and a validation split of 0.1 are used.

For the Adam method the learning rate, $\alpha_{adam} = 0.05$ was used in all cases, since this worked well. Furthermore, for β_1, β_2 and ϵ the values suggested by the authors of the method are used. ALS has no hyperparameters.

The other hyperparameters, the feature map order M, the CP-rank R, the batch size and the regularization parameter λ are varied per data set. The used values are listed in Table 6-2. For the batch size a smaller batch size is used for the smaller data sets and a large batch size for the larger data sets. Other than that, the batch size value was not tuned.

The parameters M and R affect the model, but both methods use the same model. Thus, their values should not affect the relative performance in terms of loss. It is therefore reasoned that a hyperparameter search was not necessary for M and R to compare the methods in terms of loss. The values $M \in [20, 40]$ and $R \in [10, 20]$ are used, since similar values are used in [6]. It was found that for the data sets with five million or more samples $M = 40$ could not be used with the current implementation due to a memory limit. Therefore, $M = 20$ was used for these data sets and a larger value of R was used. The theoretical complexity of the methods scales different for M and R, so their values do affect the performance in terms of run time. This is further investigated in Section 6-2-3.

The regularization parameter λ is tuned with a rudimentary search to ensure that the training succeeds for both methods. This is explained in more detail in Section 6-2-1. However, this parameter is again used by both methods. So, when training succeeds, it should not affect the relative performs of the methods. In Section 5-1 the relation between the regularization parameter and the number of features was discussed. It can be seen in Table 6-2 that the chosen regularization parameters are related to the number of features.

Table 6-2: Hyperparameters that are used for each data set for the experiments in this chapter.

Data set	N	D	R	M	λ	Batch size
Elevators	16599	18	10	20	10^{-16} [1]	500
Protein	45730	9	10	20	10^{-7}	500
Precipitation	628474	3	10	20	10^{-5}	500
HouseElectric	2049280	10	10	40	10^{-10}	5000
SUSY Low	5000000	8	20	20	10^{-10}	5000
SUSY High	5000000	10	20	20	10^{-10}	5000
SUSY Full	5000000	18	20	20	10^{-17} [1]	5000
Airline	5929413	8	20	20	10^{-10}	5000
Higgs High	11000000	7	20	20	10^{-10}	5000

[1] See Section 6-2-1.

6-2 Results

Numerous experiments are conducted to investigate the performance of the Adam method and compare it to the performance of ALS. The results are shown in this section. First, it is shown that the Adam method can no longer train successfully for a larger number of features. Thus, there is a limiting effect of the number of features. Then, the general performance the Adam method and ALS on a number of data sets is studied. Thirdly, it is investigated how the training run time of both methods scale for various values of the CP-rank and the feature map order on the HouseElectric data set. Finally, it is analyzed whether the Adam method and ALS can be combined to obtain the best of both methods.

6-2-1 Limiting effect of number of features

In Section 5-1-1 it was shown that the number of features affects the magnitude of the gradient which in turn influences the choice of the step size and the regularization parameter. When the number of features increases, the magnitude of the gradient of the mean squared error (MSE) decreases significantly. For the gradient of the regularization term this is not the case. Since the MSE gradient determines direction to take to reduce the error, there should be a reasonable balance between the gradient of the MSE term and the regularization term. To achieve this the value of the regularization parameter is paramount. Furthermore, an exceedingly small gradient could lead to additional problems.

To study the effect of a large number of features in more detail, the Elevators and SUSY Full data sets are used. A data set with fewer features can artificially be created by leaving out features of a data set. This was done with the Elevators data set to create data sets with 15, 16 and 17 features and with the SUSY Full data set for 16 and 17 features. It is assumed that it does not matter which feature is left out, so the last features are left out.

For each data set it is attempted to tune the regularization parameter λ such that training succeeds. The regularization parameter is tuned by trying different orders of magnitude. When the training has made no progress after a 100 iterations is it deemed to have failed. In case the ALS method succeeds, but the Adam method does not for a particular regularization parameter, a regularization parameter of one and two orders smaller is also tried. So, when, for example, $\lambda = 10^{-16}$ succeeds for ALS but not the Adam method, 10^{-17} and 10^{-18} are also tried. For the other hyperparameters the values given in Section 6-2 are used.

The results are shown in Table 6-3. For a larger number of features ALS still succeeds, but the Adam method does not. These results suggests that around 16-17 features the Adam method does not perform well anymore. Since the Elevators data set and the SUSY Full data set have a significant difference in number of samples it is reasoned that the number of samples is not causing this. Furthermore, they are a regression and classification task, so the learning task is most likely not the issue.

In Figure 6-1 the training loss and Canonical Polyadic Decomposition (CPD) norm during training on the SUSY Full data set with only 16 features are shown. The results show a distinct difference between the Adam method and ALS. The norm increases rapidly with ALS at the start of the training. Moreover, it was found that for the final ALS weights the norm of the first factor is significantly larger than the norm of the other factors. This corresponds to the large jump in the CPD norm after the first iteration.

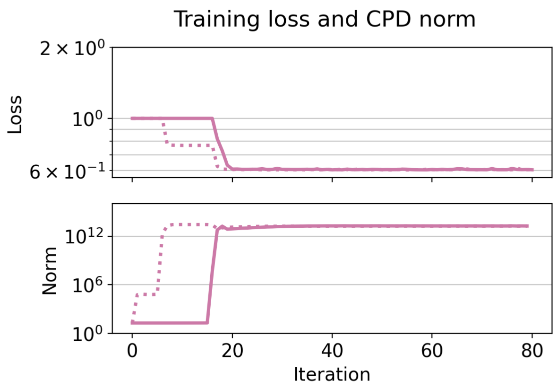
Table 6-3: Training success and corresponding regularization parameter for two data sets. A data set with fewer features is artificially created by leaving out the last feature(s). The Adam method can no longer train successfully for a larger number of features while ALS can.

(a) Elevators.

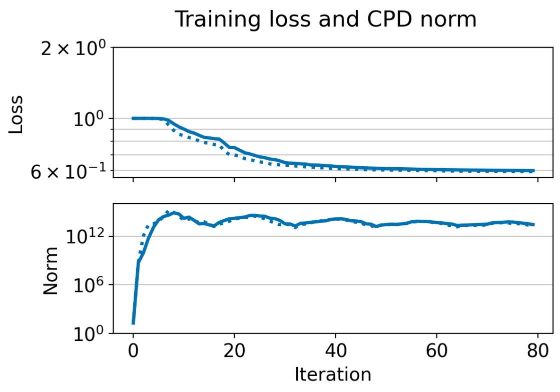
		D		
		15	16	17
λ		10^{-16}	10^{-16}	10^{-16}
Succeeded	Adam	✓	✓	
	ALS	✓	✓	✓

(b) SUSY full.

		D		
		16	17	18
λ		10^{-17}	10^{-18}	10^{-19}
Succeeded	Adam	✓	✓	
	ALS	✓	✓	✓



(a) Adam.



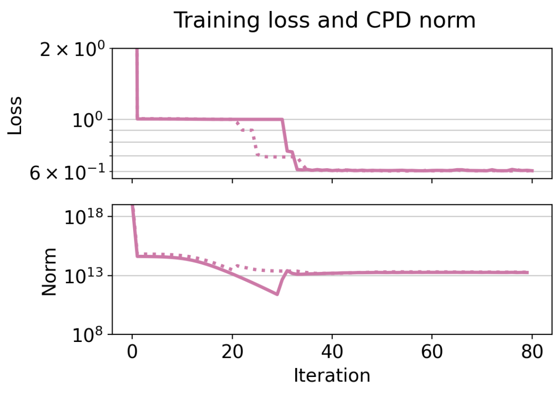
(b) ALS.

Figure 6-1: Training loss and CPD norm on the SUSY Full data set with only 16 features for two runs (solid/dotted) for different methods. The difference in the training progress for the two Adam method runs indicates that the Adam method struggles more with the training. For ALS there is no noteworthy difference, so this suggests that it can still train well. Interestingly, the downward jump in the training loss for the Adam method coincides with an increase of the CPD norm.

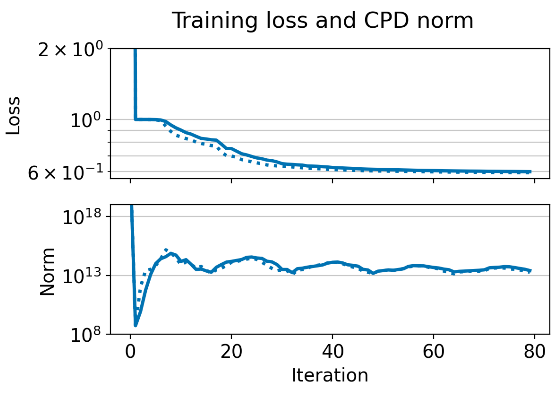
The Adam method requires several iterations before the training practically starts. In the worse case this is after 17 iterations, so there have been $5000000 \cdot 0.9/5000 \cdot 17 = 15300$ weight updates before then. Thus, the value of the gradient is most likely so small that all-at-once gradient descent becomes inefficient and for more features probably infeasible. The downward jumps in training loss are accompanied by a significant increase of the CPD norm. Moreover, the factors of final weights of the Adam method all have a similar norm. Since all factors are updated simultaneously with the same regularization it is logical that the norms of the factors are of similar order.

The observed behaviour is similar to the swamps mentioned in Section 3-2-2 where a region of slow convergence is followed by a region of fast convergence. However, swamps are usually due to a wrongly chosen CP-rank or an unfortunate initialization. Since these are the same for both methods it is reasoned that they should suffer from this equally, but this is not the case. It is therefore unclear whether this corresponds to swamp.

To investigate this further the same experiment is redone without the normalization at the initialization of the weights. The results are shown in Figure 6-2. The weights are initialized



(a) Adam.



(b) ALS.

Figure 6-2: Training loss and CPD norm on the SUSY Full data set with only 16 features for two runs (solid/dotted). The weights initialized identically as in Figure 6-1, but without normalization. So, the initial CPD factors have the same direction but a much larger norm. The norm with the Adam reaches a order of magnitude close to the final one, but this does not speed up the training. To the contrary, training takes longer, so an appropriate initial norm does not have to speed up the training. The training progress of ALS after the first iteration is very similar to Figure 6-1b which suggest that ALS can quickly recover from an initial bad scaling.

identically as before only without the normalization. For the initialization from the normal distribution $\sigma = 1$ is used, since this was also used for the initialization with normalization. Since the initialization is identical, the direction of the factors is the same, only the norm is different. Without the normalization, the initial norm is much larger which can also be seen in Figure 6-2.

The results show that both methods are able to reduce the initial norm significantly in the first iteration. Moreover, for ALS the results for the experiment with and without normalization are almost identical after the first iteration. Since both experiments start with weights that point in the same direction, but that have a different norm, this suggests that ALS is unaffected by the scaling of the initial weights. Subsequently, it can easily overcome bad initial scaling. Additionally, it shows that the absence of a step size hyperparameter for ALS can be a big advantage.

Although the CPD norm of the Adam method reaches an order closer to the final norm quicker than before, this does not improve the training. On the contrary, it takes more iterations before the downward jump in training loss happens. Thus, the initial weights are most likely not the only factor that cause this slow training.

As mentioned earlier, the gradient becomes smaller for an increasing number of features. So, this is probably an important cause of the slow training. Due to the small gradient the step size becomes crucial. Although the Adam method performs a sort of scaling of the gradient, this seems not to be enough anymore. Moreover, in Section B-4 it is shown that the Line search gradient descent method is also not able to learn on a data set with 18 features. It is shown that even with an optimal step size there is almost no training progress due to the small gradient. Additionally, the outcomes indicate that the floating-point precision might be an issue. It could be studied in future research whether the gradient can be scaled before the step size is determined in order to maintain its direction but increase its magnitude. For example, the gradient could be normalized.

The results suggests that ALS is not, or at least less, limited by the number of features. In [6] data sets with a significantly larger number of features were also used, but it is noted that their loss function differs slightly from the one used in this thesis. It could be due to the weight update method. The ALS method solves a linear least squares (LLS), which is done with the Numpy/JAX `solve` command. This method and its implementation might be better equipped to deal with the exceedingly small values in the components of the LLS, which are caused by the Hadamard sequences and the large number of features. For example, it might apply some sort of scaling. Moreover, where the Line search method has to find one optimal step size to update all DMR weights, the ALS update optimization only has to take MR weights into account. This could explain why ALS is able to perform effective updates, but the Line search method is not.

Overall, it is concluded that the studied all-at-once optimization methods do not perform well anymore for a larger number of features. The ALS method does still works, so this method is preferred in those cases.

In future research this limiting effect of the number of features for all-at-once optimization could be investigated further. First, it could be studied more precisely for which number of features this effect truly starts to come into effect. In this thesis it is only shown that for 10 features the Adam method still performs nicely, which is done in the next section, and that for 16 features problems arise. In future research numbers of features in between 10 and 16 could be analyzed. Secondly, the origin of the swamp-like behaviour could be studied in more detail. Thirdly, it could be investigated whether a different initialization method could prevent the problem. A new initialization method should not focus on the norm as illustrated with the experiment, but an initialization from a different distribution or an initialization based on the data might be a solution. Finally, it could be studied whether other all-at-once methods do not suffer from this limiting effect.

In the next section the Elevators data set will no longer be used. For the SUSY Full data set, the artificial variant with only 16 features will be used to evaluated the performance for a large scale data set with a relatively large number of features.

6-2-2 General performance

In this section the performance of the Adam method and ALS on the earlier introduced data sets is analyzed. In Figure 6-3 the training losses on a selection of the data sets are shown. The training losses for all the data sets are shown in Figure B-5. In Table 6-4 the training and validation performance losses are presented. The training run times including the convergence run times are given in Table 6-5.

Protein and Precipitation

For the two smaller data sets, Protein and Precipitation, ALS performs similar in terms of convergence and has a lower final loss than the Adam method. Figure 6-3a illustrates this. Furthermore, for both data sets the total training run time and the convergence run time are significantly lower for ALS. This suggests that for relatively small data sets it is more beneficial to use ALS compared to the Adam method.

HouseElectric, Airline, SUSY Low and SUSY High

For the large scale data sets HouseElectric, Airline, SUSY Low and SUSY High the training trajectories are noticeably different from the smaller data set. Figure 6-3b and Figure 6-3d illustrate typical training progresses for these data sets. They show that the Adam method results in a remarkably sharp decline in the training loss in the first few iterations. As a result, the Adam method reaches its convergence faster than ALS. This is due to the mini-batch aspect of the Adam method. As a result, it can perform thousands of updates within a few iterations leading to the rapid decrease in the training loss. For the ALS method this is not the case, as one iteration corresponds to one update. As a result, multiple iterations are required to reach convergence. However, the decrease in loss per update is significantly larger compared to the Adam method. This is also expected, since each ALS update is optimal. So, the Adam method thanks its performance to the ability to perform a huge amount of updates and ALS works so well due to its update effectiveness.

Table 6-4: Mean loss and one standard deviation for the Adam method and ALS on different data sets. The losses for a method are reported for the iteration where the mean validation loss for that particular method is minimal. In case training was not possible a dash is used. The results show that for all the data sets except for the Protein and SUSY Full data set, the Adam method can reach similar performance as ALS in terms of both the training and validation loss. The difference is at most two thousandths.

		Training loss		Validation loss	
Data set	D	Adam	ALS	Adam	ALS
Protein	9	0.545 (0.0113)	**0.521** (0.0031)	0.552 (0.0151)	**0.530** (0.0158)
Precipitation	3	0.955 (0.0029)	**0.953** (0.0033)	0.954 (0.0281)	**0.952** (0.0279)
HouseElectric	10	**0.155** (0.0007)	**0.155** (0.0004)	**0.155** (0.0009)	**0.155** (0.0009)
SUSY Low	8	0.600 (0.0005)	**0.599** (0.0008)	0.601 (0.0011)	**0.600** (0.0011)
SUSY High	10	0.600 (0.0009)	**0.599** (0.0007)	0.601 (0.0014)	**0.600** (0.0013)
SUSY Full*	16[1]	0.621 (0.0337)	**0.594** (0.0019)	0.621 (0.0338)	**0.594** (0.0023)
Airline	8	0.854 (0.0012)	**0.852** (0.0010)	0.855 (0.0069)	**0.853** (0.0074)
Higgs High	7	**0.804** (0.0012)	-	**0.804** (0.0014)	-

[1] An artificial data set with only 16 features is created by leaving out the last two features of the SUSY Full data set.

This difference is also reflected in the training run times. It can be seen in Table 6-5 that the total training run times of ALS are all lower than those of the Adam method. However, for the run time until convergence the opposite is true. The Adam method is able to reach convergence significantly faster than ALS. So, the Adam method is beneficial when the time to convergence is important. The hyperparameters M and R can affect the run time which is further investigated in the next section.

Moreover, it can be seen in Table 6-4 that the Adam method achieves a very similar performance in terms of loss as ALS. The differences in the losses are only one or two thousandths and the standard deviation are of similar order. Therefore, one method is not regarded as better than the other in terms of loss performance. Additionally, for both methods the variance of the loss is small, especially for the training loss. Thus, for both methods the loss

Table 6-5: Mean training run times and one standard deviation for the Adam method and ALS on different data sets. The total run time is the time to complete all iterations. The convergence is defined in Definition 6.1 and the convergence run time in Eq. (6-1). Note that different numbers of iterations are used per data set, so the performance of different data sets cannot be directly compared. For the data sets with over a million samples the Adam method reaches converge significantly faster than ALS.

				Convergence			
		Total run time (s)		Iteration		Run time (s)	
Data set	D	Adam	ALS	Adam	ALS	Adam	ALS
Protein	9	23.4 (1.06)	**2.77** (1.226)	169	174	22.0	**2.68**
Precipitation	3	90.0 (2.86)	**8.90** (2.217)	9	29	13.5	**4.30**
HouseElectric	10	259.0 (4.42)	**143.9** (3.31)	25	80	**64.8**	115.1
SUSY Low	8	621.1 (7.22)	**276.2** (5.96)	14	74	**90.6**	212.9
SUSY High	10	612.6 (4.59)	**346.6** (5.73)	33	90	**202.2**	311.9
SUSY Full*	16[1]	699.6 (7.26)	**340.3** (5.51)	32	92	**233.2**	326.1
Airline	8	748.0 (6.57)	**391.7** (7.43)	12	89	**124.7**	693.5
Higgs High	7	**1297.9** (7.17)	-	42	-	**548.1**	-

[1] An artificial data set with only 16 features is created by leaving out the last two features of the SUSY Full data set.

performance is not really affected by the particular initialization. As a result, for similar data sets it is most likely not necessary to conduct many experiments with different initializations to ascertain that the performance is not due to an unfortunate initialization.

For both methods the mean validation loss and the mean training loss are quite close and differ only a few thousandths. So, the regularization of the model works well. This is due to the regularization term, but the CPD structure can also have a regularizing effect.

Interestingly, with the Adam method the training loss can increase again after reaching a minimum. This it most noticeable for the Airline data set in Figure 6-3d due to the scaling of the plot. This might be due to the stochastic nature of the Adam method, but it is unexpected that this upward trend seems to keep going. So, it seems not to be just a temporary bump. An increase in validation loss would be due to overfitting, but an increase in training loss is happening here. The difference between the minimum and the final loss is only a few thousandths though. It could be investigated in future research why this is happening. Due to this increase in loss it could be beneficial for the Adam method to implement early stopping, although this can be difficult due to the stochasticity of the method.

HIGGS High

For the HIGGS High data set the ALS method was not able to train. In the current implementation of the ALS algorithm the feature mapping is applied for one feature on all the data samples simultaneously. Due to the large number of sample in the HIGGS High data set this resulted in a memory overload. The Adam method was able to train, because it uses batches of data instead of the full data set at once.

The training progress of the Adam method is very similar to the other large scale data sets. Moreover, the Adam method is able to reach convergence in less than 10 minutes on a data set with 11 million samples. Based on Figure B-5d, it could even be reasoned that the method converges quicker. This truly shows that the Adam method can be applied effectively and efficiently to large scale learning. In future research the implementation of ALS could be adapted to work regardless of the number of samples and then be compared with the Adam method on such large data sets.

SUSY Full

The performance of the SUSY Full data set with 16 features is more difficult to analyze, particularly for the Adam method. It can be seen in Figure 6-3c that the variance during the during is significantly larger for the Adam method compared to the other experiments. In Figure 6-1 it was already shown that there can be a big difference in terms of training progress on this data set with the Adam method for different initializations. This is also why there is such a large variance in the results of the ten experiments. Furthermore, one experiment did not convergence to a loss similar to ALS while the other nine did. As a result, the mean loss is higher and the variance larger than ALS. Nevertheless, this shows the downside of using the Adam method for data sets with a larger number of features. Then multiple experiments will have to be conducted to ensure that a seemingly converged training loss is not due to an unfortunate initialization. The ALS method is not affected by the large number of features and works similarly as for the other data sets. Note that due to the large number of features each factor is updated only $96/18 = 6$ times with ALS. This is why the training progress seems to converge slower than for the data sets with fewer features.

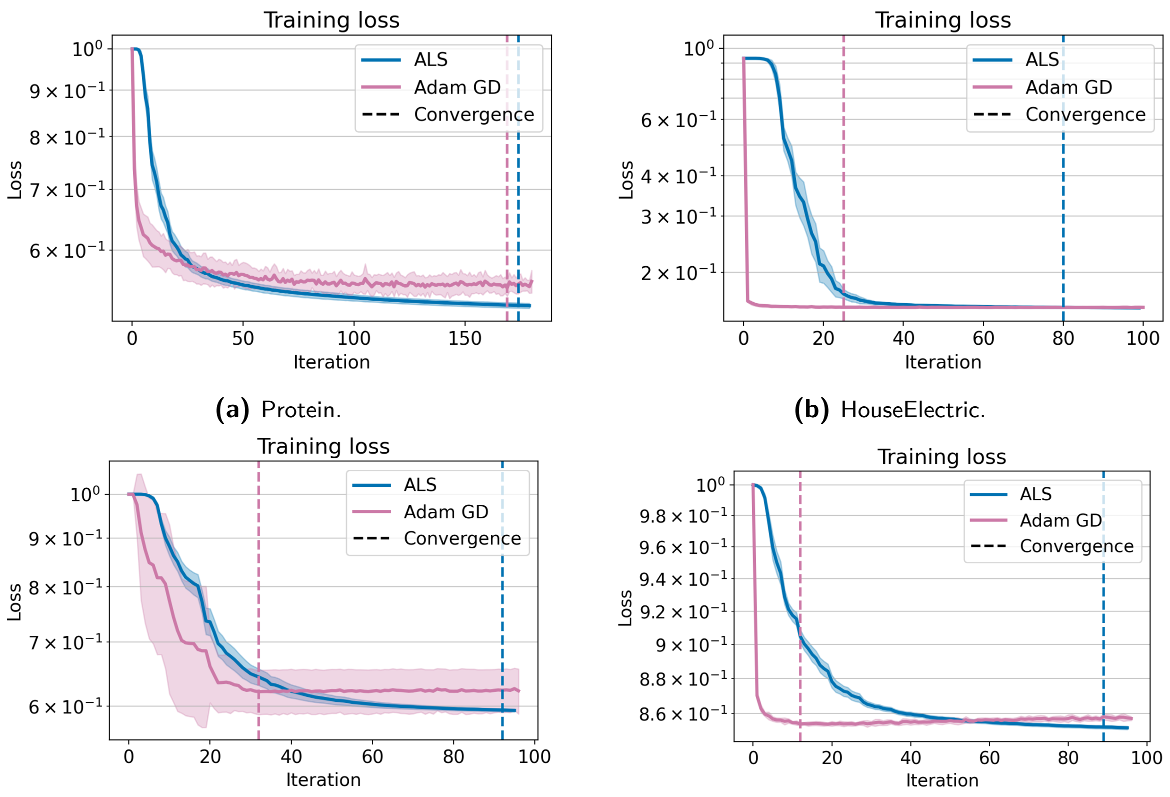
Conclusion

From these results it can be concluded that ALS is superior to the Adam method for relatively small data sets. For large scale learning with a moderate number of features, around 10 or less, the Adam method and ALS perform similar in terms of loss, but the Adam method can reach convergence more quickly. The Adam method should thus be preferred in those cases. For a larger number of features ALS is preferred, since the Adam method performs less robustly and thus more training experiments will be required. Moreover, the Adam method might not succeed at all for a larger number of features.

6-2-3 Training time for varying CP-ranks and feature map orders

It was shown in Section 4-4-1 that theoretical computational complexity of Steepest Gradient Descent (SteGD) and ALS scale differently for R and M. Since the computational complexity of the Adam method is equal to the complexity of SteGD, the effect of R and M on the total training run time for the Adam method and ALS is investigated in more detail.

To study this, both methods are trained on the HouseElectric data set for different values of M and R. The HouseElectric data set is used, because it is relatively large scale which corresponds to the focus of this thesis. Furthermore, it was found that for the current implementation of the ALS method it is not possible to use larger values of M and R for the larger data sets due to a memory limit.



(a) Protein. **(b)** HouseElectric. **(c)** SUSY Full with 16 features. **(d)** Airline.

Figure 6-3: Mean training loss (line) and one standard deviation (shaded area) on different data sets with Adam and ALS. The identified convergence points for each method are also drawn. Figure 6-3a shows that for smaller data sets ALS works better in terms of final loss and has similar convergence. Figure 6-3b demonstrates that for large data sets the Adam method converges significantly faster than ALS while achieving a similar final training loss. The large standard deviation in Figure 6-3c shows that for a larger number of features the Adam method struggles with training while ALS does not. Figure 6-3d shows again the fast convergence of the Adam method, but it also shows that the training loss of the Adam method can increase again.

The hyperparameters given in Table 6-2 are used. For each combination of M and R ten experiments are conducted. The loss performance of the method is practically equal for in all combinations, like in Figure 6-3b. They are reported in Section B-6.

In Table 6-6 the total run times are reported for the different combinations. It can be seen that when MR is the same value, the resulting run time is also similar. This corresponds to the theoretical complexity which scales the same in M and R for both methods. It can be seen that for MR of 40 or smaller the Adam method is slower than the ALS method. Once MR becomes larger, the Adam method becomes faster.

In Figure 6-4 the training run times are plotted as a function of MR. Furthermore, a fit through the mean run times is plotted. The estimated parameters of the fit are given in Table 6-7. The results show that the Adam method and ALS have a significantly different scaling. The scaling of the ALS method is of a similar order as the weight update time scaling determined in Table 4-4. The Adam method, however, scales an order smaller. This suggests that for more complex models where MR is larger, it is more advantageous to apply the Adam method for large scale learning compared to ALS.

The notable small scaling parameter of the Adam method is most likely due to the use of mini-batches and the batch implementation in combination with the application of a GPU. The computation of the all-at-once gradient and Adam update are both implemented in a batch manner and require relatively simple operations that are repeated often. Furthermore, they are applied on relative small batches due to the mini-batch method. It is reasoned that the Adam method can, therefore, make excellent use of a GPU to speed up the training. The experiment presented in Table 4-4 was not run with a GPU, so this could explain the difference in scaling. Since the ALS method still scales similarly as in Table 4-4 it is reasoned that the ALS implementation can only make sub-optimal use of the GPU, likely due to the more complex operation of solving a linear system.

It is noted that the convergence has not been taken into account in this analysis. Moreover, it was earlier mentioned that the computation of the training loss slows down the Adam method more than ALS. So, the Adam method could already be faster than ALS smaller values of MR when a record of the training progress is not needed and the training is stopped at convergence.

Here the mini-batch Adam method is compared to ALS. In future research the run time of the full-batch Adam method could also be compared to ALS. Additionally, it could be studied whether the full-batch Adam method scales similarly as the mini-batch method which it should do theoretically.

Table 6-6: Mean total training run time and one standard deviation for training on the HouseElectric data set for different values of M and R. ALS is faster for smaller values of MR. For more complex models with larger values for M and R the Adam method is faster which can make it more advantageous to use.

	$M = 10$		$M = 20$		$M = 40$	
R	Adam	ALS	Adam	ALS	Adam	ALS
10	245 (2.7)	**23.7** (3.18)	245 (3.6)	**47.2** (2.78)	259 (2.4)	**144** (3.2)
20	249 (3.6)	**46.0** (2.23)	254 (4.7)	**142** (3.0)	**272** (6.7)	481 (2.1)
40	259 (3.8)	**144** (3.2)	**279** (3.1)	482 (2.0)	**320** (2.0)	1858 (2.9)

Table 6-7: Estimated scaling parameters for the training run time (y) on the HouseElectric data set as a function of MR, the product of the feature map order M and CP-rank R. The difference in the found values for b is a factor 20 instead of 2 as is expected from the theoretical computational complexity. It is reasoned that the Adam method is be able make more efficient use of a GPU than ALS.

		$y = ax^b$	
Input (x)	Method	a	b
MR	ALS	$1.01 \cdot 10^{-2}$	1.61
	Adam	154	0.0891

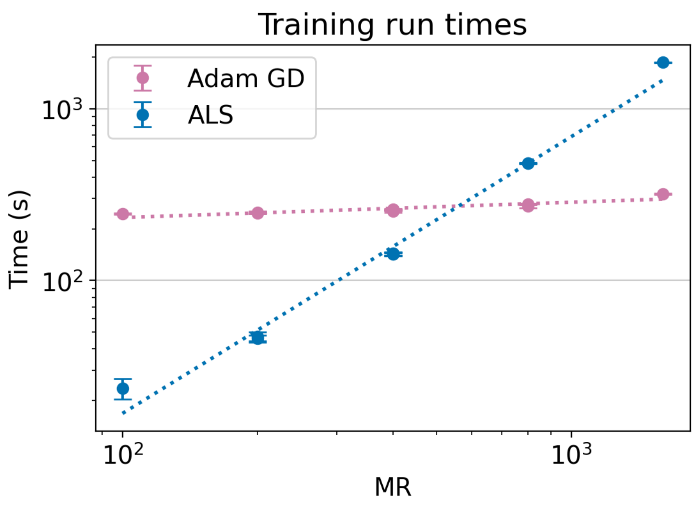


Figure 6-4: Mean total training run time on the Household data set with one standard deviation. Additionally, a fit (dotted) through the mean run times is shown. The difference in the slopes of the fit demonstrates that the run time of the Adam method scales favorable compared to ALS in terms of MR.

6-2-4 Combination of methods

In this thesis the different characteristics of the Adam method and ALS have been discussed multiple times and have been illustrated with experiments. Since the methods both have advantages and disadvantages, it is shortly studied whether they can be combined to get the best of both worlds. The mini-batch implementation of the Adam method enables quick learning, but due to the stochasticity it does not converge unless the learning rate is decreased. ALS is monotonically decreasing and thus converges, but the convergence can be slower, especially for large scale learning.

Combination approach

Based on these characteristics and the results shown in Figure 6-3 it is reasoned that it would be beneficial to use the Adam method at first to quickly reduce the training loss in a few iterations and then switch ALS to achieve convergence.

This approach was tried for the HouseElectric data set and the Airline data set. The combination of methods is compared to the performance of each method separately for a similar number of iterations. The same hyperparameters as listed in Table 6-2 and the same initialization as before are used. As a result, for the individual methods the results will be the same as in the previous section. The iteration to switch methods at is determined based on the training loss of the Adam method using the results of the previous section. So, it is determined beforehand. For the HouseElectric data set the methods switch after 10 iterations, for the Airline data set this is at 25 iterations.

Results

The training results are shown in Figure 6-5. The losses are given in Table 6-8. The results show that for the HouseElectric data set there is no advantage to using the combination of methods compared to solely using the Adam method. Only the standard deviation of the losses is decreased slightly. Furthermore, the combination of methods does not reach the minimal validation loss earlier than the combination of methods. So, for this particular case the combination is not useful.

For the Airline data set the performance does improve compared to solely using the Adam method. By switching to the ALS method the increase of the training loss of the Adam method is prevented. Hence, the combination of methods makes use of the monotonically-decrease property of ALS as desired. Moreover, the ALS combo is able to improve loss compared to the minimal loss that the Adam method achieves. The final training and validation loss of ALS are similar to that of the Adam-ALS combination. Furthermore, given the training loss trajectory of the ALS method alone, it is expected that it can converge to the same losses as the combination. Nevertheless, the combination of methods reaches the minimal loss significantly quicker.

From these two experiments it is concluded that the combination of first using the Adam method and then ALS can be beneficial for large scale learning to reach the final loss performance quicker. The substantial initial decrease due to the Adam method can effectively be combined with the monotonic decrease in the training loss of ALS to improve the final convergence. However, it seems that the combination of methods is not able to achieve a better performance in terms of the training and validation loss compared to the ALS method alone. So, the combination of methods can be used to speed up training, but does not improve the loss performance.

In these experiments the switch moment was set beforehand based on the training trajectories of the individual methods. In practice it will be unknown beforehand what the training trajectories of the individual methods are. Thus, it cannot be determined from these trajectories when to switch or whether a combination of methods is beneficial at all. An implementation of a combination of methods in practice, therefore, needs to switch during training. It is suggested to incorporate an earlier stopping criterion for the Adam method and switch to ALS after this criterion is met. In future research this can be investigated further.

Finally, in future research it could be investigated whether it is beneficial to apply ALS first and then the Adam method. For example, when the number of features is large as in Section 6-2-1 it might be beneficial to use ALS first.

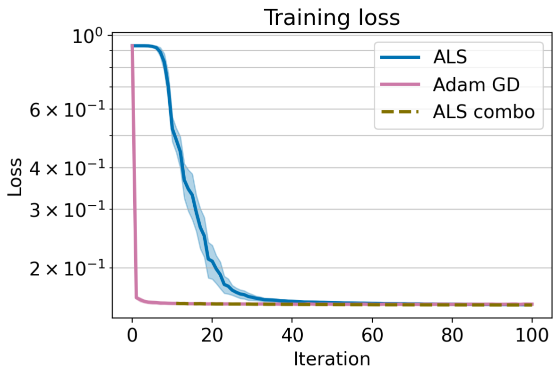
6-3 Summary

In this chapter the performance of all-at-once optimization was studied and compared to ALS. The Adam method is used as the all-at-once update method. To evaluate the performance experiments are conducted on different data sets. Several data sets with over a million samples are used to test the large scale learning performance.

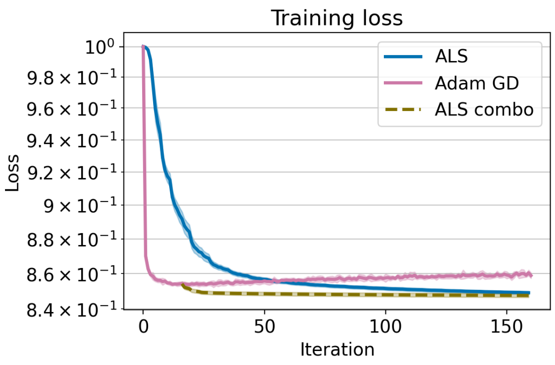
Different performance criteria are applied to compare the methods. Naturally, the training loss and validation loss are criteria. The total training run time, the run time to complete all

Table 6-8: Mean loss and one standard deviation for the Adam method, ALS and a combination of the methods for different data sets. The losses for a method are reported for the iteration where the mean validation loss for that particular method is minimal. The combination of methods does not lead to a higher final loss, but the amount of added benefit depends on the data set.

	Loss			
	HouseElectric		Airline	
Method	Training	Validation	Training	Validation
Adam	**0.155** (0.0007)	**0.155** (0.0009)	0.849 (0.0009)	0.850 (0.0073)
ALS	**0.155** (0.0004)	**0.155** (0.0009)	0.854 (0.0012)	0.860 (0.0069)
Combo	**0.155** (0.0002)	**0.155** (0.0008)	**0.848** (0.0011)	**0.849** (0.0071)



(a) HouseElectric. ALS combo starts after 10 iterations.



(b) Airline. ALS combo starts after 15 iterations.

Figure 6-5: Mean training loss (line) and one standard deviation (shaded area) on two data sets for the Adam method, ALS, and a combination. For the combination the Adam method is applied first and then, after a set number of iterations, ALS is used which is denoted as ALS combo. The results show that the combination of methods does likely not improve the final loss compared to ALS alone, but it can converge faster to the final loss.

iterations, is also used. Additionally, the run time until convergence is used. Convergence is determined based on the mean validation after the experiments have been conducted.

It is found that the Adam method does not perform well on data sets with around 16 features or more while ALS does. This is most likely due to exceedingly small gradient values that are caused by the large number of features. Additionally, it is shown that the Line search method also is not able to train on a data set with 18 features. This further confirms that the small gradient is most likely the issue. It is reasoned that ALS still works well due to the inherent difference in the update method between ALS and all-at-once optimization.

On small data sets ALS is found to be superior compared to the Adam method, both in terms of loss as well as run time. On the large scale data sets the Adam method achieves similar performance in terms of loss as ALS. The training progress of the Adam method shows a sharp decline in the first few iterations. As a result, the Adam method is able to reach convergence quicker than ALS. So, while the total run time of ALS is lower, the Adam method has a considerably lower run time until convergence. For example, the Adam method was able to reach convergence in less than 10 minutes on a data set with 11 million samples.

Furthermore, it is found that the training run time of the Adam methods scales favorable compared to ALS. The scaling for the feature map order and CP-rank of the Adam method is more than an order of magnitude lower than ALS. It is reasoned that this is because the Adam method can profit more from the GPU capabilities compared to ALS. This shows that for more complex models the Adam method is more advantageous.

Finally, it was investigated whether the methods could be combined to obtain the best of both worlds; the rapid initial decline in loss of the Adam method and the monotonic decrease of ALS. The Adam method is used for the first few iterations and then it switches to ALS. It is found that this combination can reach convergence quicker than the individual methods. However, the results suggest that it does not improve the performance in terms of loss compare to ALS alone. Moreover, it was found that the added benefit differed for the two data sets this was tested on. For one data set there was no additional benefit while for the other there was.

Overall, it is concluded that for smaller problems and problems with around 16 or more features it is beneficial to use ALS. For large scale learning with fewer features the Adam method is preferred. Moreover, for more complex models with larger feature map orders and CP-ranks it is more advantageous to use the Adam method as well. A combination of methods can be applied, but this does not necessarily perform better than the individual methods. Nevertheless, it should not perform worse. So, all-at-once optimization in the form the Adam method can effectively and efficiently be applied to solve the CPD constrained kernel machine and thus to enable large scale learning for kernel machines.

Summary of contributions

- It is shown that the Adam and Line search method do no longer perform well for data sets with around 16 or more features while ALS does.
- It is shown that on small data sets ALS is superior to the Adam method in terms of loss performance and has similar convergence.
- It is shown that on large scale data sets the Adam method achieves similar performance as ALS in terms of the final losses, but it reaches convergence significantly quicker.
- It is shown that the training run time of the Adam method scales favorably compared to ALS for the feature map order and CP-rank. So, for more complex models on large scale data set it is more advantageous in terms of the training run time to apply the Adam method.
- It is shown that the Adam method and ALS can be combined by first applying the Adam method and then switching to ALS. Does not seem to improve the performance in terms of loss, but can allow for quicker convergence compared to the individual methods.
- It is shown that all-at-once optimization in the form of the Adam method can effectively and efficiently be applied for learning with a large number of samples with the CPD constrained kernel machine.

Chapter 7

Conclusion and Recommendations

7-1 Conclusion

In this thesis it has been investigated whether all-at-once optimization can be applied to the Canonical Polyadic Decomposition (CPD) constrained Kernel Ridge Regression (KRR) optimization problem. Moreover, the goal was to enable learning with a large number of samples for kernel machines. The KRR problem is solved in the primal space such that the complexity of the solution does not scale cubically with the number of samples, which the kernel trick in the dual space does. Product feature maps are used to represent the kernel in the primal space. To reduce the number of weights of the optimization problem, a CPD is used to represent the weights. As a result, the number of weights scales linearly in the number of features of the input data instead of exponentially. However, the CPD introduces nonlinearity into the optimization problem and thus complicates the computation of a solution.

The Alternating Least Squares (ALS) method is the state-of-the-art optimization method for this problem. ALS updates one factor of the CPD at a time. In this thesis the application of all-at-once optimization in the form of gradient descent has been studied as an alternative for ALS. The main goal of this thesis was:

> ***In which situation is it more advantageous to apply all-at-once optimization compared to ALS to solve the CPD constrained kernel machine problem?***

An efficient algorithm for the computation of the all-at-once gradient was derived for this thesis. The CPD and the product feature map are stored as tensors, so the gradient can be computed without looping over the factors in Python. This significantly speeds up the computation. It was shown that automatic differentiation (AD) can also be used to compute the gradient. However, the computation time for the analytical algorithm was found to be considerably lower, so this was used.

With experiments that served as a proof of concept, it was found that Steepest Gradient Descent (SteGD) can be used to find a solution to the CPD constrained optimization problem. However, it was also found that the selection of a suitable step size is a challenge of all-at-once optimization. It was shown that the magnitude of the gradient of the mean squared error (MSE) term decreases with an increase in the number of features. As a result, determining a good step size becomes very challenging for a larger number of features. It was found that SteGD with a fixed step size already becomes ineffective for five features. Since most data sets have more than five features, SteGD is generally ineffective for large scale learning. Moreover, the gradient of the regularization term does not decrease for more features, while the gradient of the MSE term does. As a result, this needs to be taken into account when determining an appropriate value for the regularization parameter.

To overcome the challenge of selecting a step size, the use of line search was studied. An expression for the exact line search solution was derived for this thesis. By recursively expanding the Hadamard sequences in the loss function, an exact value for the optimal step size for a given step direction can be computed. It is a general expression, so it can be used for any number of features and any step direction. It was found that the Line search method in combination with gradient descent can be used to speed up training compared to SteGD with a fixed step size. However, it was also found that the Line search method is more sensitive to the weight initialization due to the optimality of the Line search method and, consequently, the inability to escape a local minimum. Moreover, the Line search method requires significantly more run time for training which makes the Line search method ineffective for large scale learning.

The Adam method was investigated as another alternative for the determination of the step size. The Adam method determines a step size for each weight individually depending on the evolution of the gradient for that particular weight. It was found that the Adam method reaches similar performance in terms of loss as ALS and barely increases the complexity compared to SteGD. Moreover, the Adam method can be combined with mini-batch gradient descent in which batches of the data set are used to compute the gradient, instead of the full training data set. This can speed up the convergence and make the Adam method effective and efficient for large scale learning. Thus, it was concluded that the mini-batch Adam method is the best all-at-once optimization method for large scale learning of the methods researched in this thesis.

Finally, the performance of all-at-once optimization in the form of the Adam method for large scale learning was studied and compared to the performance of ALS. For performance, both the training and validation losses were taken into account as well as the total training run time and training run time till convergence. The convergence was determined after the experiments were conducted, and was used to indicate when early stopping could have happened if it had been used.

It was found that the Adam method does not longer perform well for data sets with around 16 features or more while ALS does. This is most likely due to the exceedingly small gradient values which are a result of the large number of features. It was verified with a different data set that the Line search method is also not able to train successfully on a data set with 18 features. It is reasoned that ALS still works well due to the inherent difference in the update method between ALS and all-at-once optimization.

For large scale data sets with 10 features or less, it was found that the Adam method can achieve similar performance as ALS for both the training and validation loss. Due to the mini-batch implementation, the Adam method can converge within a few passes over the whole data set for large data sets. As a result, the training run time until convergence is significantly lower for the Adam method than ALS for these large data sets. For small data sets the ALS method is quicker.

Furthermore, the total training run time for the Adam method scales favorable compared to ALS for the feature map order and CP-rank. The scaling parameter of the Adam method was found to be an order smaller when both methods are run on a GPU. It is thus concluded that for larger, more complex models where the CP-rank and feature map order are large, it is advantageous to use the Adam method.

Finally, it was investigated whether the Adam method and ALS can be combined to exploit the benefits of both methods. The Adam method can be used for the first iterations to make use of its initial rapid decrease in loss and ALS can be used afterward for its monotonically decreasing property to achieve convergence. This was tried on two large scale data sets and it was shown that the combination worked. The combination of methods did not lead to improved performance in terms of loss compared to solely using ALS. However, it was able to reach the final convergence loss faster than the individual methods could. The added benefits differed for two data sets that this was tried on. When the Adam method was able to converge on its own, it did not help to switch to the ALS method. Nevertheless, it did not have an adverse effect.

To conclude and to answer the goal of this thesis, all-at-once optimization in the form of the mini-batch Adam method can be effectively and efficiently applied for large scale learning with kernel machines. The ALS method is superior on smaller problems, but on data sets with over a million samples it was found that the Adam method can outperform ALS in terms of training run time until convergence and can achieve similar performance in terms of loss. Furthermore, for more complex models the Adam method is favorable compared ALS due to its more advantageous scaling for the training run time in terms of the feature map order and the CP-rank.

7-2 Recommendations for future research

In this thesis a first study into all-at-once optimization for the CPD constrained kernel machine is done. Many aspects of the all-at-optimization method and the optimization procedure in general were analyzed, but a lot can still be studied. Given the numerous choices and assumptions that are made and used for the implementation of a machine learning algorithm, there are many areas that could be researched further.

First, potential future research areas for all-at-once optimization in general are presented. Secondly, the optimization methods studied in this thesis in specific are considered. Finally, suggestions that reach beyond the focus of this thesis are discussed.

7-2-1 All-at-once optimization in general

Several aspects of all-at-once optimization in general could be studied in more detail.

Methods In this thesis only three all-at-once optimization methods were analyzed, SteGD, Line search and the Adam method. In future research other methods could be studied such as the Gauss-Newton method or the Levenberg-Marquardt method, or other gradient based methods similar to the Adam method.

Effect of a large number of features In this thesis it was shown that the number of features has an influence on the performance of all-at-once optimization. Although this was investigated briefly, a conclusive answer why this is happening was not yet determined. Therefore, this could be studied in more detail. Furthermore, it was shown that for the Adam method this influence occurs for more than 10 features and likely starts to occur around 16-17 features. It could be studied in more detail when exactly this starts happening.

Additionally, it could be studied whether a different initialization method could help prevent this problem.

Finally, it could be studied whether the gradient can be scaled before the step size is determined. The direction of the gradient would then be maintained while its magnitude increases. For example, the gradient could be normalized.

Parallelization As mentioned earlier, the algorithm for the computation of the all-at-once gradient lends itself for parallelization. This could be implemented in the future and its effect on the training run time could be investigated. With the expected speed up, the application of all-at-once optimization on even larger data sets could be studied.

7-2-2 Methods discussed in this thesis

Line search An expression for the exact Line search solution was derived for this thesis. However, it was found the Line search algorithm is too slow in its current implementation to be applicable to large data sets. In future research the implementation could be improved. Furthermore, it could be studied whether the Line search algorithm could be implemented with JAX in such a way that it can be run on a GPU to speed it up.

The derived Line search expression was combined with the gradient in this thesis. In future research a different step direction could be tried, for example the Gauss-Newton step. Alternatively, the Line search method could be combined with ALS as in [50] and it could be studied whether this method works well for the CPD constrained kernel machine. The combination of the Line search method with mini-batch gradient descent was briefly investigated for this thesis, but it could be studied in more detail. In particular, it could be studied whether the mini-batch version of the Line search method is able the escape local minima and is, therefore, less dependent on the weight initialization.

Adam The Adam method was found to be the most effective all-at-once optimization method studied in this thesis. However, no hyperparameter search for the Adam method itself was conducted. For β_1, β_2 and ϵ the standard values recommended by the authors were used. In future research it could be investigated whether different values lead to better results.

Secondly, the experimental results indicated that the training loss of the Adam method could start increasing again after reaching a minimum. Although this increase was only a few thousandths, it did not seem like a temporary bump due to stochasticity. It could be analyzed further why this is happening.

Combination of methods The combination of the Adam method and ALS was studied where they were used in that order. However, the moment to switch from Adam to ALS was determined based on the training results of the individual methods. In practice it would be desirable to switch without knowing the results of the individual methods. A switching criteria and its effects could be studied.

Secondly, a different order of the methods and its effect could be studied. It might be beneficial to start with ALS, for example when there are a larger number of features, and then use the Adam method.

Comparison of methods Lastly, the optimization methods were compared with each other. In this thesis the losses and training run times were mostly used to compare to the methods. In future research the resulting models could be studied. It could be analyzed whether the resulting models are close to each other where the definition of close has yet to be determined.

Besides that, the current implementation of ALS limited the number of samples that could be used. The implementation could be adapted to allow for any number of samples such that ALS can be compared to the Adam method on the largest data sets.

7-2-3 Beyond this thesis

Finally, there are possibilities for future research that go beyond the focus of this thesis.

Other loss function In this thesis the KRR problem with an MSE term was the optimization problem that was studied. This problem had a convenient loss function, since it was relatively straightforward to derive its gradient. The choice of loss function affects the results. For example, the MSE error is susceptible to outliers. In future research the use of a different loss function could be studied. Its gradient could be obtained with AD which was shown to work well in this thesis. In that case, it is not necessary to have an expression for the gradient of the loss function, so a more complex loss function could be applied.

Other tensor decompositions In this thesis the CPD was used to constrain the weights. However, other tensor decompositions also could be used. It was already quickly investigated whether a Tensor Train (TT) could be combined with AD and it was found that the gradient can be computed and that all-at-once SteGD can be applied for a TT constrained kernel machine as well. Other tensor decompositions have different characteristics which could be more beneficial than those of the CPD. Therefore, the application of other tensor decomposition to the constrained kernel machine in combination with all-at-once optimization could be studied in future research.

Appendix A

Algorithms and Derivations

In this chapter of the appendix the algorithms that are used in this thesis are presented in more detail. Furthermore, several derivations of expressions that are used in the thesis are presented. First, Alternating Least Squares (ALS) for the Canonical Polyadic Decomposition (CPD) constrained kernel machine is discussed. Secondly, all-at-once optimization with gradient descent is discussed. Then, the Line search and Adam method are discussed. Lastly, the CPD weight initialization and the normalization of the CPD are discussed.

A-1 CPD ALS

In this section the CPD ALS algorithm is derived first. Secondly, the theoretical complexity of the ALS algorithm is discussed.

A-1-1 CPD ALS algorithm

The ALS algorithm for the CPD constrained kernel machine is based on [6]. However, in [6] the summed squared error is used while in this thesis the mean squared error (MSE) loss is used, so there are slight differences.

The ALS optimization problem that is used to update factor $\mathbf{W}^{(d)}$ is given by

$$\begin{aligned}\min_{\text{vec}(\mathbf{W}^{(d)})} \quad & \frac{1}{N}\sum_{n=1}^{N}\left(y_n - \left\langle \text{vec}(\mathbf{W}^{(d)}), \mathbf{z}(x_n^{(d)}) \otimes \left(\mathop{\circledast}_{p=1,p\neq d}^{D} \mathbf{z}(x_n^{(p)})^\top \mathbf{W}^{(p)}\right)\right\rangle\right)^2 \\ & + \lambda \left\langle \text{vec}(\mathbf{W}^{(d)\top}\mathbf{W}^{(d)}), \text{vec}(\left(\mathop{\circledast}_{p=1,p\neq d}^{D} \mathbf{W}^{(p)\top}\mathbf{W}^{(p)}\right))\right\rangle.\end{aligned} \tag{A-1}$$

For ease of notation the following expressions are introduced:

$$\mathbf{zW} = \mathop{\circledast}_{d=1}^{D} \mathbf{z}(x_n^{(d)})^\top \mathbf{W}^{(d)}, \qquad \mathbf{zW}_d = \mathop{\circledast}_{p=1,p\neq d}^{D} \mathbf{z}(x_n^{(p)})^\top \mathbf{W}^{(p)}, \tag{A-2}$$

$$\mathbf{\Gamma} = \mathop{\circledast}_{d=1}^{D} \mathbf{W}^{(d)\top}\mathbf{W}^{(d)}, \qquad \mathbf{\Gamma}_d = \mathop{\circledast}_{p=1,p\neq d}^{D} \mathbf{W}^{(p)\top}\mathbf{W}^{(p)}. \tag{A-3}$$

So, the optimization problem becomes

$$\begin{aligned}\min_{\mathrm{vec}(\mathbf{W}^{(d)})} \quad & \frac{1}{N}\sum_{n=1}^{N}\left(y_n - \left\langle \mathrm{vec}(\mathbf{W}^{(d)}), \mathbf{z}(x_n^{(d)}) \otimes \mathbf{zW}_d \right\rangle\right)^2 \\ & + \lambda \left\langle \mathrm{vec}(\mathbf{W}^{(d)\top}\mathbf{W}^{(d)}), \mathrm{vec}(\mathbf{\Gamma}_d)\right\rangle .\end{aligned} \tag{A-4}$$

This optimization problem is a linear least squares (LLS) in $\mathrm{vec}(\mathbf{W}^{(d)})$ and is thus convex. Its exact solution can be found by setting the gradient with respect to $\mathrm{vec}(\mathbf{W}^{(d)})$ to zero.

The gradient with respect to one factor in matrix format is given by

$$\frac{\partial f(\mathcal{W})}{\partial \mathbf{W}^{(d)}} = \frac{-2}{N}\left(\mathbf{Z}(\mathbf{X}^{(d)}) \mathbin{\tilde{\circ}} \left(\mathop{\circledast}_{p=1,p\neq d}^{D} \mathbf{Z}(\mathbf{X}^{(p)})^\top \mathbf{W}^{(p)}\right)^\top\right) \times_3 (\mathbf{y} - \mathbf{y}_{pred}) + 2\lambda \mathbf{W}^{(d)}\mathbf{\Gamma}_d, \tag{A-5}$$

which is derived in Section A-2-1.

To be able to set the gradient to zero this expression needs to be reformulated in terms of $\mathrm{vec}(\mathbf{W}^{(d)})$. The solution can then be obtained by solving a linear system.

For the regularization term this can be done as follows by making use of the Kronecker property given in Eq. (2-12):

$$\begin{aligned}2\lambda\,\mathrm{vec}(\mathbf{W}^{(d)}\mathbf{\Gamma}_d) &= 2\lambda\,\mathrm{vec}(\mathbf{I}_M \mathbf{W}^{(d)}\mathbf{\Gamma}_d) \\ &= 2\lambda(\mathbf{I}_M \otimes \mathbf{\Gamma}_d^\top)\,\mathrm{vec}(\mathbf{W}^{(d)}) \\ &= 2\lambda(\mathbf{I}_M \otimes \mathbf{\Gamma}_d)\,\mathrm{vec}(\mathbf{W}^{(d)}).\end{aligned}$$

For the MSE term it is noted that only the term $\mathbf{y}_{pred}$ contains $\mathrm{vec}(\mathbf{W}^{(d)})$. Furthermore, the batch outer product can be rewritten with the Khatri-Rao product as mentioned at Eq. (2-13):

$$\left(\mathbf{Z}(\mathbf{X}^{(d)}) \mathbin{\tilde{\circ}} \left(\mathop{\circledast}_{p=1,p\neq d}^{D} \mathbf{Z}(\mathbf{X}^{(p)})^\top \mathbf{W}^{(p)}\right)^\top\right) \rightarrow \left(\mathbf{Z}(\mathbf{X}^{(d)}) \odot \left(\mathop{\circledast}_{p=1,p\neq d}^{D} \mathbf{Z}(\mathbf{X}^{(p)})^\top \mathbf{W}^{(p)}\right)^\top\right),$$

such that

$$\left(\mathbf{Z}(\mathbf{X}^{(d)}) \mathbin{\tilde{\circ}} \left(\mathop{\circledast}_{p=1,p\neq d}^{D} \mathbf{Z}(\mathbf{X}^{(p)})^\top \mathbf{W}^{(p)}\right)^\top\right) \times_3 \mathbf{y} = \left(\mathbf{Z}(\mathbf{X}^{(d)}) \odot \left(\mathop{\circledast}_{p=1,p\neq d}^{D} \mathbf{Z}(\mathbf{X}^{(p)})^\top \mathbf{W}^{(p)}\right)^\top\right)\mathbf{y}.$$

Furthermore, the predictions can also be formulated as follows:

$$\mathbf{y}_{pred} = \left(\mathbf{Z}(\mathbf{X}^{(d)}) \odot \left(\mathop{\circledast}_{p=1,p\neq d}^{D} \mathbf{Z}(\mathbf{X}^{(p)})^\top \mathbf{W}^{(p)}\right)^\top\right)^\top \mathrm{vec}(\mathbf{W}^{(d)}).$$

For convenience the matrix $\mathbf{C}$ is defined as follows:

$$\mathbf{C} = \left(\mathbf{Z}(\mathbf{X}^{(d)}) \odot \left(\underset{p=1,p\neq d}{\overset{D}{\circledast}} \mathbf{Z}(\mathbf{X}^{(p)})^\top \mathbf{W}^{(p)}\right)^\top\right)^\top \in \mathbb{R}^{N\times MR}. \tag{A-6}$$

Hence, the gradient given in Eq. (A-5) can be rewritten as:

$$\frac{\partial f(\mathcal{W})}{\partial \mathbf{W}^{(d)}} = \frac{-2}{N}(\mathbf{C}^\top \mathbf{y} - \mathbf{C}^\top \mathbf{C}\,\mathrm{vec}(\mathbf{W}^{(d)})) + 2\lambda(\mathbf{I}_M \otimes \mathbf{\Gamma}_d)\,\mathrm{vec}(\mathbf{W}^{(d)}). \tag{A-7}$$

To solve for $\mathrm{vec}(\mathbf{W}^{(d)})$ such that the gradient becomes zero the following linear system needs to be solved:

$$\left(\mathbf{C}^\top \mathbf{C} + \lambda N(\mathbf{I}_M \otimes \mathbf{\Gamma}_d)\right)\mathrm{vec}(\mathbf{W}^{(d)}) = \mathbf{C}^\top \mathbf{y}. \tag{A-8}$$

Since one factor is updated at the time, the ALS algorithm can reuse the Hadamard sequences that are needed to compute $\mathbf{C}$ and $\mathbf{\Gamma}_d$. To that extent the full Hadamard sequences $\mathbf{ZW}$ and $\mathbf{\Gamma}$ as defined in Eq. (A-30) and Eq. (A-31) respectively are computed first. The needed Hadamard sequences are then obtained by dividing the full Hadamard sequences by the appropriate term. After a factor has been updated the full Hadamard sequence can be computed efficiently by using the Hadamard sequence without that factor and the newly updated factor.

The full ALS algorithm for optimizing the CPD constrained kernel machine is given in Algorithm A.1 and Algorithm A.2.

A-1-2 Theoretical complexity

The theoretical computational and memory complexity of the CPD ALS algorithm are given in Table A-1. For the memory complexity both the complexity is given for the batch implementation as well as the most memory efficient implementation.

Only the complexity of the weight update step is included, since these operations are conducted multiple times. The complexity for each step in Algorithm A.2 is given. The computation of the full Hadamard sequences at the startup of the algorithm are left out.

In general, the computational complexity for a matrix multiplication of a matrix of size $I_1 \times I_2$ times a matrix of size $I_2 \times I_3$ is $\mathcal{O}(I_1 I_2 I_3)$. The computational complexity of solving a linear system $\mathbf{Ax} = \mathbf{b}$ with matrix $\mathbf{A}$ of size $I \times I$ is $\mathcal{O}(I^3)$.

A-2 All-at-once CPD learning

In this section the algorithms and derivations for all-at-once gradient descent are presented. First, the derivation of the all-at-once gradient is presented. Then the total algorithm is given. Finally, the theoretical complexity of computing the gradient is discussed.

Algorithm A.1 CPD ALS algorithm based on [6].

```
function ALS_TRAINING($\mathbf{X}$, $\mathbf{y}$, $R$, $\lambda$, $\mathcal{Z}$, $number\ of\ sweeps$)
    ▷ Parameters
    $D \leftarrow \mathbf{X}$.shape[1]
    $N \leftarrow \mathbf{X}$.shape[0]
    $M \leftarrow \mathcal{Z}$.order() ▷ Get order of feature map $\mathcal{Z}$
    $\mathcal{W} \leftarrow$ initial_weights($D, M, R$)

    ▷ Compute full $\mathbf{ZW}$ and $\mathbf{\Gamma}$ sequence
    $\mathbf{ZW} \leftarrow 1$
    $\mathbf{\Gamma} \leftarrow 1$
    for $d \leftarrow 1, D$ do
        $\mathbf{ZW} \leftarrow \mathbf{ZW} * \mathbf{Z}(\mathbf{X}^{(d)})^\top \mathcal{W}_{d,:,:}$
        $\mathbf{\Gamma} \leftarrow \mathbf{\Gamma} * \mathcal{W}_{d,:,:}^\top \mathcal{W}_{d,:,:}$
    end for

    ▷ Update loop
    for $i \leftarrow 1, number\ of\ sweeps$ do
        for $d \leftarrow 1, D$ do
            $\mathcal{W}$, $\mathbf{ZW}$, $\mathbf{\Gamma} \leftarrow$ als_weight_update($\mathcal{W}, \mathbf{Z}(\mathbf{X}^{(d)}), \mathbf{y}, \mathbf{ZW}, \mathbf{\Gamma}, \lambda, d$)
        end for
    end for
    return $\mathcal{W}$
end function
```

A-2-1 All-at-once analytical gradient derivation

During this thesis an analytical expression for the all-at-once gradient was derived. The derivations are presented here in more detail. First, the partial derivative with respect to one factor matrix $\mathbf{W}^{(d)}$ will be derived and then it will be shown how the gradient with respect to all factor matrices can be computed efficiently. The derivation starts at Eq. (3-12) and for simplicity the derivation will first be done for a single sample. The loss function can be split up into the MSE term and the regularization term. The partial derivatives for each term are derived separately. Finally, the all-at-once gradient expression will be presented.

MSE term gradient

The MSE term for a single sample is given by:

$$mse_n = \frac{1}{N}\left(y_n - \left(\mathop{\circledast}_{d=1}^{D} \mathbf{z}(x_n^{(d)})^\top \mathbf{W}^{(d)}\right)\mathbf{1}_R\right)^2 = \frac{1}{N} r_n^2, \tag{A-9}$$

where r_n is the residual for sample $\{\mathbf{x}_n,\ y_n\}$ for the current weights. The partial derivative of this term with respect to the vectorized factor matrix $\text{vec}(\mathbf{W}^{(d)})$ can be expanded with the

Algorithm A.2 CPD ALS weight update based on [6].

```
function ALS_WEIGHT_UPDATE($\mathcal{W}$, $\mathbf{Z}(\mathbf{X}^{(d)})$, $y$, $\mathbf{ZW}$, $\mathbf{\Gamma}$, $\lambda, d$)
    $\mathbf{ZW}_d \leftarrow \mathbf{ZW}/(\mathbf{Z}(\mathbf{X}^{(d)})^\top \mathcal{W}_{d,:,:})$
    $\mathbf{\Gamma}_d \leftarrow \mathbf{\Gamma}/(\mathcal{W}_{d,:,:}^\top \mathcal{W}_{d,:,:})$
    $\mathbf{C} \leftarrow \left(\mathbf{Z}(\mathbf{X}^{(d)}) \odot \left(\circledast_{p=1,p\neq d}^{D} \mathbf{Z}(\mathbf{X}^{(p)})^\top \mathbf{W}^{(p)}\right)^\top\right)^\top$
    $\mathbf{A}_{reg} \leftarrow \lambda \cdot (\mathbf{I}_M \otimes \mathbf{\Gamma}_d)$
    $\mathbf{A} \leftarrow \mathbf{C}^\top \mathbf{C} + N \cdot \mathbf{A}_{reg}$
    $\mathbf{b} \leftarrow \mathbf{C}^\top y$
    $\mathbf{w}_{new} = \texttt{solve}(\mathbf{A}, \mathbf{b})$ ▷ Solve linear system
    $\mathbf{W}_{new} = \texttt{reshape}(\mathbf{w}_{new}, (M, R))$ ▷ Reshape vector back into factor matrix
    ▷ Update full Hadamard sequences
    $\mathbf{ZW} \leftarrow \mathbf{ZW}_d * \mathbf{Z}(\mathbf{X}^{(d)})^\top \mathbf{W}_{new}$
    $\mathbf{\Gamma} \leftarrow \mathbf{\Gamma}_d * \mathbf{W}_{new}^\top \mathbf{W}_{new}$
    $\mathcal{W}[d,:,:] \leftarrow \mathbf{W}_{new}$
    return $\mathcal{W}$, $\mathbf{ZW}$, $\mathbf{\Gamma}$
end function
```

chain-rule as follows:

$$\begin{aligned}\frac{\partial mse_n}{\partial \operatorname{vec}(\mathbf{W}^{(d)})} &= -\frac{2}{N}\left(y_n - \left(\underset{d=1}{\overset{D}{\circledast}} \mathbf{z}(x_n^{(d)})^\top \mathbf{W}^{(d)}\right)\mathbf{1}_R\right)\frac{\partial \langle \mathcal{W}, \mathcal{Z}(\mathbf{x})\rangle_F}{\partial \operatorname{vec}(\mathbf{W}^{(d)})} \\ &= -\frac{2}{N} r_n \frac{\partial \langle \mathcal{W}, \mathcal{Z}(\mathbf{x})\rangle}{\partial \operatorname{vec}(\mathbf{W}^{(d)})}.\end{aligned} \tag{A-10}$$

The prediction term $\langle \mathcal{W}, \mathcal{Z}(\mathbf{x})\rangle_F$ can be written as an inner product of $\operatorname{vec}(\mathbf{W}^{(d)})$. Hence, the partial derivative can easily be derived:

$$\begin{aligned}\langle \mathcal{W}, \mathcal{Z}(\mathbf{x})\rangle_F &= \left\langle \operatorname{vec}(\mathbf{W}^{(d)}), \mathbf{z}(x_n^{(d)}) \otimes \mathbf{zW}_d \right\rangle, \\ \frac{\partial \langle \mathcal{W}, \mathcal{Z}(\mathbf{x})\rangle_F}{\partial \operatorname{vec}(\mathbf{W}^{(d)})} &= \mathbf{z}(x_n^{(d)}) \otimes \mathbf{zW}_d \quad \in \mathbb{R}^{MR}.\end{aligned} \tag{A-11}$$

Here $\mathbf{zW}_d$ is defined in Eq. (A-3).

In Eq. (2-10) it is shown that the Kronecker product can be matricized using the vector outer product. Subsequently, the partial derivative can also be computed directly with respect to $\mathbf{W}^{(d)}$ without vectorizing:

$$\frac{\partial \langle \mathcal{W}, \mathcal{Z}(\mathbf{x})\rangle_F}{\partial \mathbf{W}^{(d)}} = \mathbf{z}(x_n^{(d)}) \circ \mathbf{zW}_d \quad \in \mathbb{R}^{M\times R}. \tag{A-12}$$

Regularization term gradient

Following Eq. (A-4) the regularization term $\langle \mathcal{W}, \mathcal{W}\rangle_F$ can be written as follows:

$$\begin{aligned}\langle \mathcal{W}, \mathcal{W}\rangle_F &= \left\langle \operatorname{vec}(\mathbf{W}^{(d)\top}\mathbf{W}^{(d)}), \operatorname{vec}(\mathbf{\Gamma}_d)\right\rangle \\ &= \operatorname{vec}(\mathbf{W}^{(d)\top}\mathbf{W}^{(d)})^\top \operatorname{vec}(\mathbf{\Gamma}_d),\end{aligned} \tag{A-13}$$

Table A-1: Theoretical complexity of the ALS method. The individual steps correspond to the steps given in Algorithm A.2.

	Complexity			
Operation	Computational	Memory impl.	Memory eff.	Note
$\mathbf{Z}(\mathbf{X}^{(d)})$	$\mathcal{O}(MN)$	$\mathcal{O}(MN)$	$\mathcal{O}(M)$	
$\mathbf{ZW}_d$	$\mathcal{O}(MRN)$	$\mathcal{O}(RN)$	$\mathcal{O}(R)$	
$\mathbf{\Gamma}_d$	$\mathcal{O}(MR^2)$	$\mathcal{O}(R^2)$	$\mathcal{O}(R^2)$	
$\mathbf{C}$	$\mathcal{O}(MRN)$	$\mathcal{O}(MRN)$	$\mathcal{O}(MR)$	
$\mathbf{A}_{reg}$	$\mathcal{O}(M^2R^2)$	$\mathcal{O}(M^2R^2)$	$\mathcal{O}(M^2R^2)$	
$\mathbf{A}$	$\mathcal{O}(M^2R^2N)$	$\mathcal{O}(M^2R^2)$	$\mathcal{O}(M^2R^2)$	
$\mathbf{b}$	$\mathcal{O}(MRN)$	$\mathcal{O}(MR)$	$\mathcal{O}(MR)$	
$\mathbf{w}_{new}$	$\mathcal{O}(M^3R^3)$	$\mathcal{O}(MR)$	$\mathcal{O}(MR)$	
$\mathbf{W}_{new}$	$\mathcal{O}(1)$	$\mathcal{O}(MR)$	$\mathcal{O}(MR)$	Computer operation
$\mathbf{ZW}$	$\mathcal{O}(MRN)$	$\mathcal{O}(RN)$	$\mathcal{O}(R)$	
$\mathbf{\Gamma}$	$\mathcal{O}(MR^2)$	$\mathcal{O}(R^2)$	$\mathcal{O}(R^2)$	
Overall	$\mathcal{O}(M^2R^2N)$	$\mathcal{O}(MRN)$	$\mathcal{O}(M^2R^2)$	Assuming $N \gg MR$
For D updates	$\mathcal{O}(DM^2R^2N)$	$\mathcal{O}(MRN)$	$\mathcal{O}(M^2R^2)$	

where $\mathbf{\Gamma}_d$ is defined in Eq. (A-3). With the Kronecker matrix-vector product property introduced in Eq. (2-12) this can be rewritten as:

$$\langle \mathcal{W}, \mathcal{W} \rangle_F = \operatorname{vec}(\mathbf{W}^{(d)})^\top (\mathbf{W}^{(d)} \otimes \mathbf{I}_M) \operatorname{vec}(\mathbf{\Gamma}_d). \tag{A-14}$$

Therefore, the partial derivative with respect to $\operatorname{vec}(\mathbf{W}^{(d)})$ is:

$$\begin{aligned} \frac{\partial \langle \mathcal{W}, \mathcal{W} \rangle_F}{\partial \operatorname{vec}(\mathbf{W}^{(d)})} &= 2(\mathbf{W}^{(d)} \otimes \mathbf{I}_M) \operatorname{vec}(\mathbf{\Gamma}_d) \\ &= 2 \operatorname{vec}(\mathbf{W}^{(d)} \mathbf{\Gamma}_d). \end{aligned} \tag{A-15}$$

The factor two in the partial derivative is due to the chain-rule. Since $\mathbf{W}^{(d)}$ occurs twice in Eq. (A-14) the chain rule results in two identical terms, hence the factor two.

The partial derivative with respect to the factor matrix itself is:

$$\frac{\partial \langle \mathcal{W}, \mathcal{W} \rangle_F}{\partial \mathbf{W}^{(d)}} = 2\mathbf{W}^{(d)} \mathbf{\Gamma}_d. \tag{A-16}$$

Overall the partial derivative of the loss function for one sample, $f_n(\mathcal{W})$, with respect to factor matrix $\mathbf{W}^{(d)}$ is:

$$\frac{\partial f_n(\mathcal{W})}{\partial \mathbf{W}^{(d)}} = -\frac{2}{N} r_n \mathbf{z}(x_n^{(d)}) \circ \mathbf{z}\mathbf{W}_d + 2\lambda \mathbf{W}^{(d)} \mathbf{\Gamma}_d. \tag{A-17}$$

Batch MSE gradient

In the previous section the partial derivative of the MSE term for a single sample was derived. To compute the partial derivative for a batch of samples the derivatives for the individual samples can be summed. Alternatively, the partial derivative for a batch of samples can be computed at once by adding the samples as an extra dimension in each computation. This is only necessary for the MSE term, since the regularization term is independent of the samples. In the derivation of the following equations a batch size N is assumed. Following the notation in this thesis this corresponds to the full batch of samples. However, any batch size can be used.

The batch MSE term can be given as the sum of the individual residuals, but alternatively it can be formulated in vector notation:

$$\begin{aligned} mse = \frac{1}{N}\sum_{n=1}^{N}(y_n - \langle \mathcal{W}, \mathcal{Z}(\mathbf{x}_n)\rangle_F)^2 &= \frac{1}{N}\sum_{n=1}^{N} r_n^2 \\ &= \frac{1}{N}\|\mathbf{r}\|_2^2, \quad \mathbf{r} = [r_1\ r_2\ \dots\ r_N]^\top \\ &= \frac{1}{N}\left\|\mathbf{y} - \mathbf{y}_{pred}\right\|_2^2, \quad \mathbf{y} = [y_1\ y_2\ \dots y_N]^\top. \end{aligned} \tag{A-18}$$

The vector $\mathbf{y}_{pred}$ contains the individual predictions:

$$\mathbf{y}_{pred} = [y_{1,pred}\ \dots\ y_{N,pred}]^\top = [\langle \mathcal{W}, \mathcal{Z}(\mathbf{x}_1)\rangle_F\ \dots\ \langle \mathcal{W}, \mathcal{Z}(\mathbf{x}_N)\rangle_F]^\top. \tag{A-19}$$

In Eq. (3-6) the batch feature map $\mathbf{Z}(\mathbf{X}^{(d)})$ is defined. It contains the transformed data for all the samples for one feature:

$$\mathbf{Z}(\mathbf{X}^{(d)}) = [\mathbf{z}(x_1^{(d)})\ \mathbf{z}(x_2^{(d)}) \dots \mathbf{z}(x_N^{(d)})] \ \in \mathbb{R}^{M\times N}. \tag{A-20}$$

Subsequently, the product $\mathbf{z}(x_n^{(d)})^\top \mathbf{W}^{(d)}$ can be computed for all the samples at once:

$$\mathbf{Z}(\mathbf{X}^{(d)})^\top \mathbf{W}^{(d)} = [\mathbf{z}(x_1^{(d)})^\top \mathbf{W}^{(d)}\ \mathbf{z}(x_2^{(d)})^\top \mathbf{W}^{(d)} \dots \mathbf{z}(x_N^{(d)})^\top \mathbf{W}^{(d)}] \in \mathbb{R}^{N\times R}. \tag{A-21}$$

By taking the Hadamard products of $\mathbf{Z}(\mathbf{X}^{(d)})^\top \mathbf{W}^{(d)}$ for $d = 1, \dots, D$ a batch of predictions can be computed as follows:

$$\begin{aligned} \mathbf{y}_{pred} &= \left(\mathop{\circledast}_{d=1}^{D} \mathbf{Z}(\mathbf{X}^{(d)})^\top \mathbf{W}^{(d)} \right) \mathbf{1}_R \in \mathbb{R}^N \\ &= \left[\left(\mathop{\circledast}_{d=1}^{D} \mathbf{z}(x_1^{(d)})^\top \mathbf{W}^{(d)} \right) \mathbf{1}_R\ \left(\mathop{\circledast}_{d=1}^{D} \mathbf{z}(x_2^{(d)})^\top \mathbf{W}^{(d)} \right) \mathbf{1}_R\ \dots\ \left(\mathop{\circledast}_{d=1}^{D} \mathbf{z}(x_N^{(d)})^\top \mathbf{W}^{(d)} \right) \mathbf{1}_R \right]^\top. \end{aligned} \tag{A-22}$$

The partial derivative of the batch prediction with respect to one factor follows Eq. (A-12) and is given by:

$$\frac{\partial \mathbf{y}_{pred}}{\partial \mathbf{W}^{(d)}} = \mathbf{Z}(\mathbf{X}^{(d)}) \tilde{\circ} \left(\mathop{\circledast}_{p=1, p\neq d}^{D} \mathbf{Z}(\mathbf{X}^{(p)})^\top \mathbf{W}^{(p)} \right) \ \in \mathbb{R}^{M\times R\times N}, \tag{A-23}$$

such that

$$\left(\frac{\partial \mathbf{y}_{pred}}{\partial \mathbf{W}^{(d)}} \right)_{:,:,n} = \mathbf{z}(x_n^{(d)}) \circ \mathbf{z}\mathbf{W}_d \quad \in \mathbb{R}^{M\times R}. \tag{A-24}$$

To compute the final partial derivative, the partial derivatives of each sample need to be multiplied with the corresponding residual and summed. This can be done by contracting the tensor of Eq. (A-23) with the error vector $\mathbf{y} - \mathbf{y}_{pred}$:

$$\frac{\partial mse}{\partial \mathbf{W}^{(d)}} = \frac{-2}{N}\mathbf{Z}(\mathbf{X}^{(d)})\tilde{\circ}\left(\mathop{\circledast}_{p=1,p\neq d}^{D}\mathbf{Z}(\mathbf{X}^{(p)})^\top\mathbf{W}^{(p)}\right)\times_3(\mathbf{y}-\mathbf{y}_{pred}) \in \mathbb{R}^{M\times R}. \tag{A-25}$$

Here $\times_3$ means that tensor contraction is computed along the third axis of the left tensor which in this case corresponds to the axis that contains the samples. Since the right side of the contraction is a vector, the tensor loses one dimension.

So, the total partial derivative of the loss function with respect to a factor matrix for a batch of samples is

$$\mathbf{G}^{(d)} = \frac{-2}{N}\left(\mathbf{Z}(\mathbf{X}^{(d)})\ \tilde{\circ}\ \left(\mathop{\circledast}_{p=1,p\neq d}^{D}\mathbf{Z}(\mathbf{X}^{(p)})^\top\mathbf{W}^{(p)}\right)^\top\right)\times_3(\mathbf{y}-\mathbf{y}_{pred}) + 2\lambda\mathbf{W}^{(d)}\mathbf{\Gamma}_d. \tag{A-26}$$

All-at-once gradient

With the expression for the partial derivative for a single factor matrix the all-at-once gradient can be derived. The all-at-once gradient is defined as follows:

$$\mathcal{G}_{d,:,:} = \mathbf{G}^{(d)} = \frac{\partial f(\mathcal{W})}{\partial \mathbf{W}^{(d)}} \in \mathbb{R}^{M\times R}, \quad \mathcal{G} \in \mathbb{R}^{D\times M\times R}. \tag{A-27}$$

The gradient is defined as a tensor, since the CPD weights are also represented as a the tensor $\mathcal{W}$. As a result, the gradient can easily be subtracted from the weights in a weight update. Furthermore, it allows for an efficient computation of the gradient without looping over the factors.

The strategy for computing the gradient with respect to all factors at once, is to reuse as much as possible. In particular the Hadamard product sequences that appear in the MSE term and the regularization term can be reused. Furthermore, Numpy/JAX functions are used to efficiently compute multiple parts at once using broadcasting[1]. The tensor representation of a CPD, $\mathcal{W}$, and the tensor representation of the batch feature map, $\mathcal{Z}(\mathbf{X})$ are used. They were introduced in Section 2-2-3 and Section 3-1-2 respectively.

First, the relevant parts that are needed for the all-at-once algorithm will be defined. Then the whole algorithm is presented.

The tensors $\mathcal{ZW}$ and $\mathcal{T}$ are defined for $d = 1, \ldots, D$ as follows:

$$\mathcal{ZW}_{d,:,:} = \mathbf{Z}(\mathbf{X}^{(d)})^\top\mathbf{W}^{(d)} \in \mathbb{R}^{N\times R}, \quad \mathcal{ZW} \in \mathbb{R}^{D\times N\times R}, \tag{A-28}$$

$$\mathcal{T}_{d,:,:} = \mathbf{W}^{(d)\top}\mathbf{W}^{(d)} \in \mathbb{R}^{R\times R}, \quad \mathcal{T} \in \mathbb{R}^{D\times R\times R}. \tag{A-29}$$

[1]See https://numpy.org/doc/stable/user/basics.broadcasting.html for more information about Numpy broadcasting.

They can be used to compute the Hadamard product sequences:

$$\mathbf{ZW} = \mathop{\circledast}_{d=1}^{D} \mathcal{ZW}_{d,:,:} = \mathop{\circledast}_{d=1}^{D} \mathbf{Z}(\mathbf{X}^{(d)})^{\top}\mathbf{W}^{(d)}, \tag{A-30}$$

$$\mathbf{\Gamma} = \mathop{\circledast}_{d=1}^{D} \mathcal{T}_{d,:,:} = \mathop{\circledast}_{d=1}^{D} \mathbf{W}^{(d)\top}\mathbf{W}^{(d)}. \tag{A-31}$$

The tensors $\mathcal{ZW}$ and $\mathcal{T}$ can be computed at once in Numpy/JAX using $\mathcal{W}$ and $\mathcal{Z}(\mathbf{X})$. The `matmul` command is used to perform the matrix multiplications simultaneously. The `swapaxes` command can be used to perform the transpose of a batch of matrices that are stacked to form a 3-way tensor.

$$\begin{aligned} \mathcal{W}^{\top} &= \text{swapaxes}(\mathcal{W},\ 1,\ 2), & \mathcal{T} &= \text{matmul}(\mathcal{W}^{\top}, \mathcal{W}), \\ \mathcal{Z}(\mathbf{X})^{\top} &= \text{swapaxes}(\mathcal{Z}(\mathbf{X}),\ 1,\ 2), & \mathcal{ZW} &= \text{matmul}(\mathcal{Z}(\mathbf{X})^{\top}, \mathcal{W}). \end{aligned}$$

Note that the transpose sign in $\mathcal{W}^{\top}$ and $\mathcal{Z}(\mathbf{X})^{\top}$ is a slight abuse of the notion of a transpose. Here it means that matrices that are stacked along the first dimension are transposed. Furthermore, zero-indexing of the axes is used, since this is also used in Python. So `swapaxes`$(\cdot, 1, 2)$ corresponds to swapping the second and third axis of the 3-way tensor.

The full Hadamard sequences of Eq. (A-30) and Eq. (A-31) can be used to compute each Hadamard sequences where one factor is left out. This is, for instance, needed in Eq. (4-10). These sequences are represented by $\mathcal{ZW}_d \in \mathbb{R}^{D\times N\times R}$ and $\mathcal{T}_d \in \mathbb{R}^{D\times R\times R}$ for the MSE and regularization term respectively:

$$\begin{aligned} (\mathcal{ZW}_d)_{d,:,:} &= \left(\mathop{\circledast}_{p=1}^{D} \mathbf{Z}(\mathbf{X}^{(p)})^{\top}\mathbf{W}^{(p)}\right) / \left(\mathbf{Z}(\mathbf{X}^{(d)})^{\top}\mathbf{W}^{(d)}\right) \\ &= \mathop{\circledast}_{p=1, p\neq d}^{D} \mathbf{Z}(\mathbf{X}^{(p)})^{\top}\mathbf{W}^{(p)}, \end{aligned} \tag{A-32}$$

$$\begin{aligned} (\mathcal{T}_d)_{d,:,:} &= \left(\mathop{\circledast}_{p=1}^{D} \mathbf{W}^{(p)\top}\mathbf{W}^{(p)}\right) / \left(\mathbf{W}^{(d)\top}\mathbf{W}^{(d)}\right) \\ &= \mathop{\circledast}_{p=1, p\neq d}^{D} \mathbf{W}^{(p)\top}\mathbf{W}^{(p)}. \end{aligned} \tag{A-33}$$

The divisions are done element-wise here.

The full tensors $\mathcal{ZW}_d$ and $\mathcal{T}_d$ can be computed at once using the Numpy/JAX broadcasting[2] functionality as follows:

$$\mathcal{ZW}_d = \mathbf{ZW}/\mathcal{ZW}, \tag{A-34}$$

$$\mathcal{T}_d = \mathbf{\Gamma}/\mathcal{T}. \tag{A-35}$$

The division / in these equations follows the broadcasting rules, since, $\mathbf{ZW}$ and $\mathcal{ZW}_d$ are not the same shape. It is thus also an slight abuse of the division operator.

For the regularization term, the gradient can now be computed as in Eq. (A-16) for all factors at once:

$$\mathcal{G}_{reg} = 2\lambda \cdot \text{matmul}(\mathcal{W}, \mathcal{T}_d). \tag{A-36}$$

[2]See https://numpy.org/doc/stable/user/basics.broadcasting.html for more information.

To compute the gradient of the MSE term as given in Eq. (A-25) for all factors at once, the batch outer product and mode-d vector product need to be computed with an extra dimension in mind. Let the intermediate tensor $\mathcal{V} \in \mathbb{R}^{D\times M\times R\times N}$ be defined as follows:

$$\mathcal{V}_{d,:,:,:} = \left(\mathcal{Z}(\mathbf{X})_{d,:,:} \,\tilde{\circ}\, (\mathcal{Z}\mathcal{W}_d)^\top_{d,:,:}\right) \in \mathbb{R}^{M\times R\times N}, \quad d = 1, \dots,\ D. \tag{A-37}$$

The tensor $\mathcal{V}$ can be computed at once with Numpy/JAX as follows:

$$\begin{aligned} \mathcal{Z}\mathcal{W}_d^\top &= \text{swapaxes}(\mathcal{Z}\mathcal{W}_d, 1, 2), \\ \mathcal{V} &= \mathcal{Z}(\mathbf{X})[:, :, \text{jnp.newaxis}, :] * \mathcal{Z}\mathcal{W}_d^\top[:, \text{jnp.newaxis}, :, :] \end{aligned} \tag{A-38}$$

This expression again makes use of Numpy/JAX broadcasting by adding an extra axis with `jnp.newaxis` where appropriate.

The MSE gradient can be computed at once by

$$\mathcal{G}_{mse} = \frac{-2}{N}\mathcal{V} \times_4 (\mathbf{y} - \mathbf{y}_{pred}). \tag{A-39}$$

The total all-at-once gradient is simply the sum of the two components and can be used to compute the step direction of Eq. (4-4):

$$\mathcal{G} = \mathcal{G}_{reg} + \mathcal{G}_{mse}, \quad \mathbf{P}^{(d)} = -\mathcal{G}_{d,:,:}, \quad d = 1, \dots D. \tag{A-40}$$

In Algorithm A.3 the full algorithm for computing the all-at-once gradient is presented.

A-2-2 Theoretical complexity

In this section the theoretical complexity of the Steepest Gradient Descent (SteGD) weight update is presented. The theoretical complexity determined for the individual steps for computing the gradient and then the overall complexity is determined. The results are shown in Table A-2.

As mentioned previously in Section A-1-2, the computational complexity for a matrix multiplication of a matrix of size $I_1 \times I_2$ times a matrix of size $I_2 \times I_3$ is $\mathcal{O}(I_1 I_2 I_3)$. The computational complexity of solving a linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$ with matrix $\mathbf{A}$ of size $I \times I$ is $\mathcal{O}(I^3)$.

A-3 Line search

The Line search method computes an optimal step size at each iteration given the current weights and a step direction. The corresponding optimization problem in given by

$$\begin{aligned} \min_{\alpha} \quad h(\alpha) = \frac{1}{N}\sum_{n=1}^{N}\left(y_n - \left(\mathop{\circledast}_{d=1}^{D}\left(\mathbf{z}(x_n^{(d)})^\top \mathbf{W}^{(d)} + \alpha \mathbf{z}(x_n^{(d)})^\top \mathbf{P}^{(d)}\right)\right)\mathbf{1}_R\right)^2 \\ + \lambda \mathbf{1}_R^\top \left(\mathop{\circledast}_{d=1}^{D}\left(\mathbf{W}^{(d)} + \alpha\mathbf{P}^{(d)}\right)^\top \left(\mathbf{W}^{(d)} + \alpha\mathbf{P}^{(d)}\right)\right)\mathbf{1}_R. \end{aligned} \tag{A-41}$$

It turns out that $h(\alpha)$ is a polynomial in α. An expression for $h(\alpha)$ was derived for this thesis which can be used to compute the exact solution to the line search problem. The solution is presented in Section 5-1-2. Here the derivation will be presented in some more detail. Then, the theoretical computational and memory complexity of the computation of the solution is discussed.

Algorithm A.3 All-at-once gradient for the CPD Kernel Ridge Regression (KRR) problem. The algorithm makes use of the Numpy/JAX functions `matmul` and `swapaxes` to compute the matrix-matrix products and the transpose for batches of matrices. Moreover, broadcasting is used for the division of tensors of different shape.

```
function GRADIENT($\mathcal{W}$, $\mathcal{Z}_{batch}$, $\mathbf{y}_{batch}$, $\lambda$)
    ▷ Regularization gradient
    $\mathcal{W}^\top \leftarrow$ swapaxes($\mathcal{W}$, 1, 2)
    $\mathcal{T} \leftarrow$ matmul($\mathcal{W}^\top, \mathcal{W}$)
    $\mathbf{\Gamma} \leftarrow \circledast_{d=1}^{D} \mathcal{T}_{d,:,:}$
    $\mathcal{T}_d \leftarrow \mathbf{\Gamma}/\mathcal{T}$
    $\mathcal{G} \leftarrow 2\lambda \cdot$ matmul($\mathcal{W}$, $\mathcal{T}_d$)

    ▷ MSE gradient
    $N_{batch} \leftarrow \mathbf{y}_{batch}$.shape[0] ▷ Batch size
    $\mathcal{Z}_{batch}^\top \leftarrow$ swapaxes($\mathcal{Z}_{batch}$, 1, 2)
    $\mathcal{ZW} \leftarrow$ matmul($\mathcal{Z}_{batch}^\top, \mathcal{W}$)
    $\mathbf{ZW} \leftarrow \circledast_{d=1}^{D} \mathcal{ZW}_{d,:,:}$
    $\mathbf{y}_{pred} \leftarrow \mathbf{ZW}\ \mathbf{1}_R$
    $\mathbf{r} \leftarrow \mathbf{y}_{batch} - \mathbf{y}_{pred}$
    $\mathcal{ZW}_d \leftarrow \mathbf{ZW}/\mathcal{ZW}$
    $\mathcal{ZW}_d^\top \leftarrow$ swapaxes($\mathcal{ZW}_d$, 1, 2)
    $\mathcal{V} = \mathcal{Z}_{batch}$[:, :, jnp.newaxis, :] $*$ $\mathcal{ZW}_d^\top$[:, jnp.newaxis, :, :]

    ▷ Compute final gradient
    $\mathcal{G} \leftarrow \mathcal{G} - 2/N_{batch} \cdot (\mathcal{V} \times_4 \mathbf{r})$
    return $\mathcal{G}$
end function
```

A-3-1 Line search solution derivation

The loss function $h(\alpha)$ is a $2D$ order polynomial in α, so it can be written as follows:

$$h(\alpha) = \sum_{i=0}^{2D} h_i \alpha^i. \tag{A-42}$$

As a result, the solution can be obtained by computing the roots of the derivative of $h(\alpha)$ and evaluating the function h for these roots to see which root results in the minimal value. Taking the derivative of a polynomial is straightforward. The expression of the solution therefore focuses on the computation of the coefficients h_i.

The computation of the coefficients is split up into two parts. The coefficients of the MSE term $\omega(\alpha)$ and the regularization term $k(\alpha)$ are computed separately and then combined such that $h(\alpha) = \omega(\alpha) + k(\alpha)$.

In Section 5-1-2 it was illustrated how a polynomial with scalar factors can be expanded recursively. The same approach is used to expand the Hadamard sequences in Eq. (A-41).

Table A-2: Theoretical complexity of one weight update for SteGD. The individual steps correspond to the steps given in Algorithm A.3 and the SteGD weight update. The memory complexity of each step is defined as the memory complexity of the intermediate values and output of that step, not the input. Both the memory complexity of used implementation and the most memory efficient possibility are given.

	Complexity			
Operation	Computational	Memory impl.	Memory eff.	Note
Reg. gradient				
$\mathcal{W}^\top$	$\mathcal{O}(1)$	$\mathcal{O}(DMR)$	$\mathcal{O}(DMR)$	Computer operation
$\mathcal{T}$	$\mathcal{O}(DMR^2)$	$\mathcal{O}(DR^2)$	$\mathcal{O}(DR^2)$	
$\mathbf{\Gamma}$	$\mathcal{O}(DR^2)$	$\mathcal{O}(R^2)$	$\mathcal{O}(R^2)$	
$\mathcal{T}_d$	$\mathcal{O}(DR^2)$	$\mathcal{O}(DR^2)$	$\mathcal{O}(DR^2)$	
$\mathcal{G}$	$\mathcal{O}(DMR^2)$	$\mathcal{O}(DMR)$	$\mathcal{O}(DMR)$	
Overall	$\mathcal{O}(DMR^2)$	$\mathcal{O}(DMR)$	$\mathcal{O}(DMR)$	Assuming $M \gg R$
MSE gradient				
$\mathcal{Z}_{batch}$	$\mathcal{O}(DMN)$	$\mathcal{O}(DMN)$	$\mathcal{O}(DM)$	
$\mathcal{Z}_{batch}^\top$	$\mathcal{O}(1)$	$\mathcal{O}(DMN)$	$\mathcal{O}(DM)$	Computer operation
$\mathcal{Z}\mathcal{W}$	$\mathcal{O}(DMRN)$	$\mathcal{O}(DRN)$	$\mathcal{O}(DR)$	
$\mathbf{ZW}$	$\mathcal{O}(DRN)$	$\mathcal{O}(RN)$	$\mathcal{O}(R)$	
$\mathbf{y}_{pred}$	$\mathcal{O}(RN)$	$\mathcal{O}(N)$	$\mathcal{O}(1)$	
$\mathbf{r}$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(1)$	
$\mathcal{Z}\mathcal{W}_d$	$\mathcal{O}(DRN)$	$\mathcal{O}(DRN)$	$\mathcal{O}(DR)$	
$\mathcal{Z}\mathcal{W}^\top$	$\mathcal{O}(1)$	$\mathcal{O}(DRN)$	$\mathcal{O}(DR)$	Computer operation
$\mathcal{V}$	$\mathcal{O}(DMRN)$	$\mathcal{O}(DMRN)$	$\mathcal{O}(DMR)$	
$\mathcal{G}$	$\mathcal{O}(DMRN)$	$\mathcal{O}(DMRN)$	$\mathcal{O}(DMR)$	
Overall	$\mathcal{O}(DMRN)$	$\mathcal{O}(DMRN)$	$\mathcal{O}(DMR)$	
Weight update	$\mathcal{O}(DMR)$	$\mathcal{O}(DMR)$	$\mathcal{O}(DMR)$	
Overall	$\mathcal{O}(DMRN)$	$\mathcal{O}(DMRN)$	$\mathcal{O}(DMR)$	Assuming $N \gg R$

Regularization term The regularization term $k(\alpha)$ is given by

$$k(\alpha) = \lambda \mathbf{1}_R^\top \left(\mathop{\circledast}_{d=1}^{D} \left(\mathbf{W}^{(d)} + \alpha \mathbf{P}^{(d)}\right)^\top \left(\mathbf{W}^{(d)} + \alpha \mathbf{P}^{(d)}\right) \right) \mathbf{1}_R \tag{A-43}$$

$$= \lambda \mathbf{1}_R^\top \left(\mathop{\circledast}_{d=1}^{D} \left(\mathbf{W}^{(d)\top}\mathbf{W}^{(d)} + (\mathbf{W}^{(d)\top}\mathbf{P}^{(d)} + \mathbf{P}^{(d)\top}\mathbf{W}^{(d)})\alpha + \mathbf{P}^{(d)\top}\mathbf{P}^{(d)}\alpha^2\right) \right) \mathbf{1}_R \tag{A-44}$$

$$= \lambda \mathbf{1}_R^\top \left(\mathop{\circledast}_{d=1}^{D} \left(\mathbf{A}_0^{(d)} + \mathbf{A}_1^{(d)}\alpha + \mathbf{A}_2^{(d)}\alpha^2\right) \right) \mathbf{1}_R, \tag{A-45}$$

$$\mathbf{A}_0^{(d)} = \mathbf{W}^{(d)\top}\mathbf{W}^{(d)}, \quad \mathbf{A}_1^{(d)} = \mathbf{W}^{(d)\top}\mathbf{P}^{(d)} + \mathbf{P}^{(d)\top}\mathbf{W}^{(d)}, \quad \mathbf{A}_2^{(d)} = \mathbf{P}^{(d)\top}\mathbf{P}^{(d)}. \tag{A-46}$$

The goal is now to compute the coefficients k_i, such that $k(\alpha)$ can be expressed as:

$$k(\alpha) = \sum_{i=0}^{2D} k_i \alpha^i. \tag{A-47}$$

The coefficients k_i can be computed by expanding the Hadamard sequence recursively. The only difference is that the coefficients are now not scalars, but matrices and the product is a Hadamard product. For example, for three dimensions, $D = 3$, this would look as follows:

$$\begin{aligned}
&\mathop{\circledast}_{d=1}^{3} \left(\mathbf{A}_0^{(d)} + \mathbf{A}_1^{(d)}\alpha + \mathbf{A}_2^{(d)}\alpha^2\right) \\
&\quad = \left(\mathbf{A}_0^{(1)} + \mathbf{A}_1^{(1)}\alpha + \mathbf{A}_2^{(1)}\alpha^2\right) * \left(\mathbf{A}_0^{(2)} + \mathbf{A}_1^{(2)}\alpha + \mathbf{A}_2^{(2)}\alpha^2\right) * \left(\mathbf{A}_0^{(3)} + \mathbf{A}_1^{(3)}\alpha + \mathbf{A}_2^{(3)}\alpha^2\right) \\
&\quad = \left(\left(\mathbf{A}_0^{(1)} * \mathbf{A}_0^{(1)}\right) + \left(\mathbf{A}_1^{(1)} * \mathbf{A}_0^{(0)} + \mathbf{A}_0^{(1)} * \mathbf{A}_1^{(0)}\right)\alpha \right. \\
&\qquad + \left(\mathbf{A}_2^{(1)} * \mathbf{A}_0^{(0)} + \mathbf{A}_1^{(1)} * \mathbf{A}_1^{(0)} + \mathbf{A}_0^{(1)} * \mathbf{A}_2^{(0)}\right)\alpha^2 + \left(\mathbf{A}_2^{(1)} * \mathbf{A}_1^{(0)} + \mathbf{A}_1^{(1)} * \mathbf{A}_1^{(0)}\right)\alpha^3 \\
&\qquad \left. + \mathbf{A}_2^{(1)} * \mathbf{A}_2^{(0)}\alpha^4\right) * \left(\mathbf{A}_0^{(3)} + \mathbf{A}_1^{(3)}\alpha + \mathbf{A}_2^{(3)}\alpha^2\right) \\
&\quad = \left(\mathbf{K}_0^{new} + \mathbf{K}_1^{new}\alpha + \mathbf{K}_2^{new}\alpha^2 + \mathbf{K}_3^{new}\alpha^3 + \mathbf{K}_4^{new}\alpha^4\right) * \left(\mathbf{A}_0^{(3)} + \mathbf{A}_1^{(3)}\alpha + \mathbf{A}_2^{(3)}\alpha^2\right) \\
&\quad = \ldots.
\end{aligned}$$

This illustrates how the Hadamard sequences can be expanded recursively. The recursive formula starts with initial values. For the first dimension:

$$\begin{aligned}
\mathbf{K}_0 &= \mathbf{W}^{(1)\top}\mathbf{W}^{(1)}, \\
\mathbf{K}_1 &= \mathbf{P}^{(1)\top}\mathbf{W}^{(1)} + \mathbf{W}^{(1)\top}\mathbf{P}^{(1)}, \\
\mathbf{K}_2 &= \mathbf{P}^{(1)\top}\mathbf{P}^{(1)}.
\end{aligned} \tag{A-48}$$

Then, one multiplication at a time of the Hadamard product sequence can be worked out using the coefficients of the current factor given by Eq. (A-48) and the coefficients of the polynomial $\mathbf{K}_i$ that are worked out so far. This corresponds to recursively looping over d from $d = 2$ to $d = D$. At each loop the new coefficients are given by:

$$\begin{aligned}
\mathbf{K}_0^{new} &= \mathbf{K}_0 * \mathbf{A}_0^{(d)}, \\
\mathbf{K}_1^{new} &= \mathbf{K}_0 * \mathbf{A}_1^{(d)} + \mathbf{K}_1 * \mathbf{A}_0^{(d)}, \\
\mathbf{K}_{2d-1}^{new} &= \mathbf{K}_{2d-3} * \mathbf{A}_2^{(d)} + \mathbf{K}_{2d-2} * \mathbf{A}_1^{(d)}, \\
\mathbf{K}_{2d}^{new} &= \mathbf{K}_{2d-2} * \mathbf{A}_2^{(d)}, \\
\mathbf{K}_i^{new} &= \mathbf{K}_{i-2} * \mathbf{A}_2^{(d)} + \mathbf{K}_{i-1} * \mathbf{A}_1^{(d)} + \mathbf{K}_i * \mathbf{A}_0^{(d)}, \quad i = 2, \ldots, 2d-2.
\end{aligned} \tag{A-49}$$

Note again that for the first two and last two coefficients separate formulas are used, since less terms are involved. The result of this loop is a $2D$ polynomial with matrix coefficients. The final scalar coefficients can then be computed by taking the sum:

$$k_i = \lambda \mathbf{1}_R^\top \mathbf{K}_i \mathbf{1}_R, \quad i = 0, \ldots, 2D. \tag{A-50}$$

MSE term The coefficients of the MSE term, $\omega(\alpha)$, can be computed in a similar fashion. Since the MSE term is the averaged sum over all the samples, the coefficients for each individual sample can be computed individually and then added:

$$\begin{aligned}\omega(\alpha) &= \sum_{i=0}^{2D} \omega_i \alpha^i = \sum_{n=1}^{N} \omega_n(\alpha) = \sum_{n=1}^{N}\sum_{i=0}^{2D} \omega_{i,n}\alpha^i, \\ \omega_n(\alpha) &= \sum_{i=0}^{2D} \omega_{i,n}\alpha^i.\end{aligned} \tag{A-51}$$

For a single sample $\{\mathbf{x}_n,\ y_n\}$, $\omega_n(\alpha)$ can be expanded as follows:

$$\begin{aligned}\omega_n(\alpha) = \frac{1}{N}y_n^2 - \frac{2}{N}y_n \left(\mathop{\circledast}_{d=1}^{D} \left(\mathbf{z}(x_n^{(d)})^\top \mathbf{W}^{(d)} + \alpha \mathbf{z}(x_n^{(d)})^\top \mathbf{P}^{(d)} \right) \right) \mathbf{1}_R \\ + \frac{1}{N} \left(\left(\mathop{\circledast}_{d=1}^{D} \left(\mathbf{z}(x_n^{(d)})^\top \mathbf{W}^{(d)} + \alpha \mathbf{z}(x_n^{(d)})^\top \mathbf{P}^{(d)} \right) \right) \mathbf{1}_R \right)^2.\end{aligned} \tag{A-52}$$

The computation of each individual coefficient $\omega_{i,n}$ is split up into a linear part $l_{i,n}$ and quadratic part $q_{i,n}$:

$$\begin{aligned}\omega_{0,n} &= \frac{1}{N}(y_n^2 - 2y_n l_{0,n} + q_{0,n}), \\ \omega_{i,n} &= \frac{1}{N}(-2y_n l_{i,n} + q_{i,n}), \quad i = 1, \ldots, D, \\ \omega_{i,n} &= \frac{1}{N} q_{i,n}, \quad i = D+1, \ldots, 2D.\end{aligned} \tag{A-53}$$

In Section 5-1-2 it is presented how the coefficient $l_{i,n}$ can be computed, so this is noted repeated here. For the coefficients $q_{i,n}$ of the last term of Eq. (A-52) the coefficients $l_{i,n}$ are reused:

$$\begin{aligned}\sum_{i=0}^{2D} q_{i,n}\alpha^i &= \left(\left(\mathop{\circledast}_{d=1}^{D} \mathbf{z}(x_n^{(d)})^\top \mathbf{W}^{(d)} + \alpha \mathbf{z}(x_n^{(d)})^\top \mathbf{P}^{(d)} \right) \mathbf{1}_R \right)^2 \\ &= \left(\sum_{i=0}^{D} l_{i,n}\alpha^i \right)^2.\end{aligned} \tag{A-54}$$

The coefficients $q_{i,n}$ can be computed by multiplying the polynomial with coefficients $l_{i,n}$ with itself:

$$\sum_{i=0}^{2D} q_{i,n}\alpha^i = \left(\sum_{i=0}^{D} l_{i,n}\alpha^i \right) \cdot \left(\sum_{i=0}^{D} l_{i,n}\alpha^i \right). \tag{A-55}$$

The values for $q_{i,n}$ can be computed with the Cauchy product of power series:

$$\begin{aligned}q_{i,n} &= \sum_{i=a+b} l_{a,n} l_{b,n} \\ &= \sum_{i=0}^{a} l_{a,n} l_{i-a,n}.\end{aligned} \tag{A-56}$$

The last formulation of the series is similar to the convolution operation. The convolution of two discrete signals f and g is given by

$$(f \circledast g)[i] = \sum_{a=-\infty}^{\infty} f[a]g[i-a], \tag{A-57}$$

where $\circledast$ is the convolution operator [56]. The similarities between this and the Cauchy product of power series can already be seen. The coefficients $q_{i,n}$ can be computed by taking the convolution of the signal with $l_{i,n}$ as its elements with itself. The signal is padded with zeros outside the coefficient $l_{i,n}$.

Line search algorithm The complete Line search method algorithm is given in Algorithm A.4. It can be used in combination with any step direction.

Algorithm A.4 CPD line search weight update for KRR.

```
function WEIGHT_UPDATE($\mathcal{W}$, $\mathcal{Z}_{batch}$, $\mathbf{y}_{batch}$, $\lambda$, $\alpha$)
    ▷ Note that input $\alpha$ is not used, but passed to be consistent with other weight updates
    ▷ Compute step direction, can be any step direction in general
    $\mathcal{P} \leftarrow$ step_direction($\mathcal{W}$, $\mathcal{Z}_{batch}$, $y_n$, $\lambda$)

    ▷ Compute coefficients and roots
    $\mathbf{k} \leftarrow$ regularization_coefficients($\mathcal{W}$, $\mathcal{P}$, $\lambda$) ▷ Eq. (5-13).
    $\boldsymbol{\omega} \leftarrow \mathbf{0}$
    for $n \leftarrow 1,$ number of samples do
        $\boldsymbol{\omega} \leftarrow \boldsymbol{\omega} +$ residual_coefficients($\mathcal{W}$, $\mathcal{P}$, $\mathcal{Z}_{batch}$, $\mathbf{y}_{batch}$) ▷ Eq. (5-16).
    end for
    $\mathbf{h} \leftarrow \mathbf{k} + \boldsymbol{\omega}$
    $\mathbf{dh} \leftarrow \frac{d\mathbf{h}}{d\alpha}$
    roots $\leftarrow$ compute_roots($\mathbf{dh}$) ▷ Roots of polynomial that has dh as coefficients.

    ▷ Find best root
    $\alpha_{best} \leftarrow 0$
    best loss $\leftarrow$ compute_loss($\mathcal{W}$, $\mathcal{Z}_{batch}$, $\mathbf{y}_{batch}$, $\lambda$)
    for root in roots do
        if is_real(root) then
            loss $\leftarrow h(root)$ ▷ Using Eq. (5-4).
            if loss < best loss then
                $\alpha_{best} \leftarrow root$
                best loss $\leftarrow$ loss
            end if
        end if
    end for
    $\mathcal{W}_{new} \leftarrow \mathcal{W} + \alpha_{best}\mathcal{P}$
    return $\mathcal{W}$
end function
```

A-3-2 Theoretical complexity

For each weight update the optimal step size needs to be computed. This adds computational and memory complexity to the weight update. The complexity per step in computation of the optimal step size is determined. The results are shown in Table A-3.

Due to the recursive nature of the method the complexity is harder to determine. The amount of products that need to be compute increases as the recursion progresses. For the regularization term, the complexity of computing a new term $\mathbf{K}_i^{new}$ is R^2 which corresponds to the element-wise operations. The term $\mathbf{K}_i^{new}$ is defined in Eq. (A-49). For the first loop of the recursion, $d = 1$, 3 new terms need to be computed, for the second loop, five terms, for the third loop seven terms. After the recursion is completed $\mathcal{O}(D^2)$ terms have been computed, which is given by the sum of odd numbers[3]. So, the computational complexity of the whole recursion is $\mathcal{O}(D^2R^2)$. The memory complexity is $\mathcal{O}(DR^2)$, since each term is of size R^2 and there are $2D$ terms in the end.

For the coefficients of the MSE term $l_{i,n}$, the complexity can be determined in a similar fashion. For the first loop there are 2 terms, for the second loop there are three terms, for the third loop four terms. After the recursion is completed $\mathcal{O}(D^2)$ terms have been computed, which is given by the sum of the natural numbers[4]. The computational complexity of a single term is R and this needs to be computed for N samples. So, the overall computational complexity of the recursion is $\mathcal{O}(D^2RN)$. The memory complexity is $\mathcal{O}(RN)$ for a batch implementation where the coefficient of each sample is computed simultaneously. For a memory efficient implementation the memory complexity is $\mathcal{O}(R)$.

For the evaluation of the loss for each root the Horner's method is used[5]. Since the derivative polynomial is of order $2D - 1$, there are generally $\mathcal{O}(D)$ real roots.

A-4 The Adam method

The Adam method algorithm is shown in Algorithm A.5.

A-5 Weight initialization

In this section the algorithms are presented for the two weight initialization methods that are used for this thesis. In Algorithm A.6 the procedure for initializing random uniform CPD weights is given with the option to normalize or not. The procedure for initializing weights from a normal distribution is given in Algorithm A.7 including the option to normalize or not.

[3]See https://www.cuemath.com/algebra/sum-of-odd-numbers/.
[4]See https://www.cuemath.com/sum-of-natural-numbers-formula/.
[5]See https://eli.thegreenplace.net/2010/03/30/horners-rule-efficient-evaluation-of-polynomials.

Table A-3: Theoretical complexity of the computation of the optimal step size with the Line search method. The individual steps correspond to the steps given in Algorithm A.4. The memory complexity of each step is defined as the memory complexity of the intermediate values and output of that step, not the input. Both the memory complexity of used implementation and the most memory efficient possibility are given. For some steps more explanation is given in the text.

	Complexity			
Operation	Computational	Memory impl.	Memory eff.	Note
Reg. coeff.				
$\mathbf{A}_0^{(d)}, \mathbf{A}_1^{(d)}, \mathbf{A}_2^{(d)}$	$\mathcal{O}(DMR^2)$	$\mathcal{O}(R^2)$	$\mathcal{O}(R^2)$	Computed $2D$ times
Recursion	$\mathcal{O}(D^2R^2)$	$\mathcal{O}(DR^2)$	$\mathcal{O}(DR^2)$	Explanation in the text
k_i	$\mathcal{O}(DR^2)$	$\mathcal{O}(DR^2)$	$\mathcal{O}(DR^2)$	
Overall	$\mathcal{O}(DMR^2)$	$\mathcal{O}(DR^2)$	$\mathcal{O}(DR^2)$	Assuming $M \gg D$
MSE coeff.				
$\mathbf{b}_{0,n}, \mathbf{b}_{1,n}$	$\mathcal{O}(DMRN)$	$\mathcal{O}(RN)$	$\mathcal{O}(R)$	Computed D times
Recursion	$\mathcal{O}(D^2RN)$	$\mathcal{O}(DRN)$	$\mathcal{O}(DR)$	Explanation in the text
l_i	$\mathcal{O}(DRN)$	$\mathcal{O}(D)$	$\mathcal{O}(D)$	
q_i	$\mathcal{O}(D^2)$	$\mathcal{O}(D)$	$\mathcal{O}(D)$	Standard convolution
Overall	$\mathcal{O}(DMRN)$	$\mathcal{O}(DRN)$	$\mathcal{O}(DR)$	Assuming $M \gg D$
Derivative	$\mathcal{O}(D)$	$\mathcal{O}(D)$	$\mathcal{O}(D)$	
Compute roots	$\mathcal{O}(D^3)$	$\mathcal{O}(D^2)$	$\mathcal{O}(D^2)$	
Evaluate roots	$\mathcal{O}(D^2)$	$\mathcal{O}(D)$	$\mathcal{O}(D)$	$\mathcal{O}(D)$ roots
Overall	$\mathcal{O}(DMRN)$	$\mathcal{O}(DRN)$	$\mathcal{O}(DR^2)$	Assuming $N \gg R$

A-6 Normalization

In Section 5-3 it is studied whether the normalization of the CPD improves the learning performance. In this section the weight update with normalization and the normalization method itself are presented first. The computation of the gradient for the normalized CPD was done with automatic differentiation (AD), so no analytical expression was derived. As a result, the SteGD method and Adam method can be readily applied. For the computation of the step size with the Line search method, the step size loss function has to be adapted, so this is presented here.

Weight update with normalization The algorithm for the general CPD all-at-once weight update with the normalization step is given in Algorithm A.8. The algorithm to renormalize a CPD after a weight update is given in Algorithm A.9.

Line search solution with normalization In Section 5-1-2 the solution of the line search optimization problem is derived for a CPD that is not normalized. The solution can easily be adapted to apply for a normalized CPD.

Algorithm A.5 Adam method [8]. All vector operations are element-wise.

```
Require: $\alpha_{adam}$ ▷ Step size hyperparameter
Require: $\beta_1,\ \beta_2 \in [0,1)$ ▷ Exponential decay rates of moment estimates
Require: $f(\mathbf{w})$ ▷ Loss function for weight vector $\mathbf{w}$
Require: $\mathbf{w}_0$ ▷ Initial values
  $\mathbf{m}_0 \leftarrow \mathbf{0}$ ▷ Initialize first moment vector
  $\mathbf{v}_0 \leftarrow \mathbf{0}$ ▷ Initialize second moment vector
  $k \leftarrow 0$
  while $\mathbf{w}_k$ not converged do
      $k \leftarrow k+1$
      $\mathbf{g}_k \leftarrow \nabla f_k(\mathbf{w}_{k-1})$ ▷ Compute gradient for current weights
      $\mathbf{m}_k \leftarrow \beta_1 \cdot \mathbf{m}_{k-1} + (1-\beta_1) \cdot \mathbf{g}_k$
      $\mathbf{v}_k \leftarrow \beta_2 \cdot \mathbf{v}_{k-1} + (1-\beta_2) \cdot \mathbf{g}_k * \mathbf{g}_k$
      $\hat{\mathbf{m}}_k \leftarrow \mathbf{m}_k/(1-\beta_1^k)$
      $\hat{\mathbf{v}}_k \leftarrow \mathbf{v}_k/(1-\beta_2^k)$
      $\mathbf{w}_k \leftarrow \mathbf{w}_k - \alpha_{adam}\hat{\mathbf{m}}_k/(\sqrt{\hat{\mathbf{v}}_k}+\epsilon)$
  end while
  return $\mathbf{w}_k$
```

An not-normalized CPD is in essence a normalized CPD of which the factors are not normalized and the norm vector $\boldsymbol{\mu}$ is a vector full of ones. With this in mind, the loss function can easily be adapted to include the norm vector $\boldsymbol{\mu}$ by simply replacing any $\mathbf{1}_R$ with $\boldsymbol{\mu}$. So, the general loss function of the CPD constrained kernel machine given in Eq. (3-12) for a normalized CPD is given by

$$f(\mathcal{W}) = \frac{1}{N}\sum_{n=1}^{N}\left(y_n - \left(\mathop{\circledast}_{d=1}^{D} \mathbf{z}(x_n^{(d)})^\top \mathbf{W}^{(d)}\right)\boldsymbol{\mu}\right)^2 + \lambda\boldsymbol{\mu}^\top\left(\mathop{\circledast}_{d=1}^{D}\mathbf{W}^{(d)\top}\mathbf{W}^{(d)}\right)\boldsymbol{\mu}. \tag{A-58}$$

Hence, the line search optimization problem of Eq. (5-3) becomes:

$$\begin{aligned}\min_{\alpha} \quad h(\alpha) = \frac{1}{N}\sum_{n=1}^{N}&\left(y_n - \left(\mathop{\circledast}_{d=1}^{D} \mathbf{z}(x_n^{(d)})^\top \mathbf{W}^{(d)}\right)\boldsymbol{\mu}\right)^2 \\ &+ \lambda\boldsymbol{\mu}^\top\left(\mathop{\circledast}_{d=1}^{D}\mathbf{W}^{(d)\top}\mathbf{W}^{(d)}\right)\boldsymbol{\mu}.\end{aligned} \tag{A-59}$$

The computation of the coefficients $\mathbf{K_i}$ and $\mathbf{l}_{i,n}$ does not change. Only the computation of the final scalar coefficients must be adapted. Again, the products with $\mathbf{1}_R$ are replaced with $\boldsymbol{\mu}$:

$$\begin{aligned} k_i &= \lambda\boldsymbol{\mu}^\top\mathbf{K}_i\boldsymbol{\mu}, \quad i = 0,\dots,2D, \\ l_{i,n} &= \mathbf{l}_{i,n}\boldsymbol{\mu}, \quad i = 1,\dots,D. \end{aligned} \tag{A-60}$$

These are the only adaptations that are needed to compute the line search solution for the normalized CPD.

Algorithm A.6 Uniform CPD weight initialization with the option to normalize or not.

```
procedure UNIFORM_WEIGHTS($D,\ M,\ R,\ bounds,\ normalize = True$)
    $low,\ high = bounds$ ▷ Get high and low bounds
    $\mathcal{W} \leftarrow$ zeros($D, M, R$) ▷ Initialize CPD in tensor format
    ▷ Loop of factor matrix and then individual vectors
    for $d \leftarrow 1, D$ do
        $\mathbf{W}^{(d)} \leftarrow$ zeros($M, R$)
        for $r \leftarrow 1, R$ do
            $\mathbf{w}_r^{(d)} \leftarrow$ uniform($M$) ▷ Draw from standard $[0, 1)$ uniform distribution.
            $\mathbf{w}_r^{(d)} \leftarrow \mathbf{w}_r^{(d)} \cdot (high - low) + low$ ▷ Shift bounds of distribution
            if $normalize$ then
                $\mathbf{w}_r^{(d)} \leftarrow \frac{\mathbf{w}_r^{(d)}}{\|\mathbf{w}_r^{(d)}\|}$ ▷ Normalize
            end if
            $\mathbf{W}_{:,r}^{(d)} \leftarrow \mathbf{w}_r^{(d)}$
        end for
        $\mathcal{W}_{d,:,:} \leftarrow \mathbf{W}^{(d)}$
    end for
    return $\mathcal{W}$
end procedure
```

Algorithm A.7 Normal CPD weight initialization with the option to normalize or not.

```
procedure UNIFORM_WEIGHTS($D,\ M,\ R,\ \mu,\ \sigma,\ normalize = True$)
    $low,\ high = bounds$ ▷ Get high and low bounds
    $\mathcal{W} \leftarrow$ zeros($D, M, R$) ▷ Initialize CPD in tensor format
    ▷ Loop of factor matrix and then individual vectors
    for $d \leftarrow 1, D$ do
        $\mathbf{W}^{(d)} \leftarrow$ zeros($M, R$)
        for $r \leftarrow 1, R$ do
            $\mathbf{w}_r^{(d)} \leftarrow \mathcal{N}(\mu, \sigma^2)$ ▷ Draw from normal distribution.
            if $normalize$ then
                $\mathbf{w}_r^{(d)} \leftarrow \frac{\mathbf{w}_r^{(d)}}{\|\mathbf{w}_r^{(d)}\|}$ ▷ Normalize
            end if
            $\mathbf{W}_{:,r}^{(d)} \leftarrow \mathbf{w}_r^{(d)}$
        end for
        $\mathcal{W}_{d,:,:} \leftarrow \mathbf{W}^{(d)}$
    end for
    return $\mathcal{W}$
end procedure
```

Algorithm A.8 General CPD all-at-once weight update with normalization step.

```
function WEIGHT_UPDATE($\mathcal{W}$, $\boldsymbol{\mu}$, $\mathcal{Z}_{batch}$, $\mathbf{y}_{batch}$, $\lambda$, $\alpha$)
    $\mathcal{G}$ = gradient($\mathcal{W}$, $\boldsymbol{\mu}$, $\mathcal{Z}_{batch}$, $\mathbf{y}_{batch}$, $\lambda$)
    $\alpha \leftarrow ...$  ▷ Determine step size: fixed/Line search/Adam
    $\mathcal{W}_{new} = \mathcal{W} - \alpha\mathcal{G}$
    $\mathcal{W}_{new}$, $\boldsymbol{\mu}_{new}$ = normalize_cpd($\mathcal{W}_{new}$, $\boldsymbol{\mu}$)
    return $\mathcal{W}_{new}$, $\boldsymbol{\mu}_{new}$
end function
```

Algorithm A.9 Normalization of CPD.

```
function NORMALIZE_CPD($\mathcal{W}$, $\boldsymbol{\mu}$)
    $D \leftarrow \mathcal{W}$.shape[0]
    $R \leftarrow \mathcal{W}$.shape[2]
    for $d \leftarrow 1, D$ do
        for $r \leftarrow 1, R$ do
            $\mathbf{w}_r^{(d)} \leftarrow \mathcal{W}_{d,:,:}$
            $norm \leftarrow \left\|\mathbf{w}_r^{(d)}\right\|$
            $\mathcal{W}_{d,:,:} \leftarrow \mathbf{w}_r^{(d)}/norm$ ▷ Update factor matrix
            $\boldsymbol{\mu}_r \leftarrow \boldsymbol{\mu}_r \cdot norm$ ▷ Move norm to norm vector
        end for
    end for
    return $\mathcal{W}$, $\boldsymbol{\mu}$
end function
```

Appendix B

Training and results

In this chapter of the Appendix more experiment results are presented. Furthermore, several aspects of the training are discussed in more detail. First, it is shown with a small experiment what the effect of the definition of an iteration is. Secondly, the instability of the full-batch Steepest Gradient Descent (SteGD) method during training on the Airfoil data set is investigated. Thirdly, the mini-batch Line search and Adam method are studied. Fourthly, it is shown that the Line search method cannot train successfully on a data set with a large number of features. Fifthly, additional results for the experiments conducted in Section 6-2-2 are presented. Then, the training losses for the run time experiments conducted in Section 6-2-3 are shown. Lastly, for the used data sets it is listed where they can be accessed and what preparation of the data was applied.

B-1 The perspective of one iteration

In Definition 4.1 an iteration is defined. Moreover, it is mentioned that the definition of an iteration can have an effect on the perspective of the results. This is briefly illustrate here. For convenience the definition of an iteration is repeated here:

Definition B.1 (Iteration/Epoch). *One iteration, or epoch, is defined as any number of updates until all the training data has been used once.*

Alternative definitions of an iteration could be:

1. One iteration is defined as any number of updates until all the factors have been updated and all the training data has been used once.

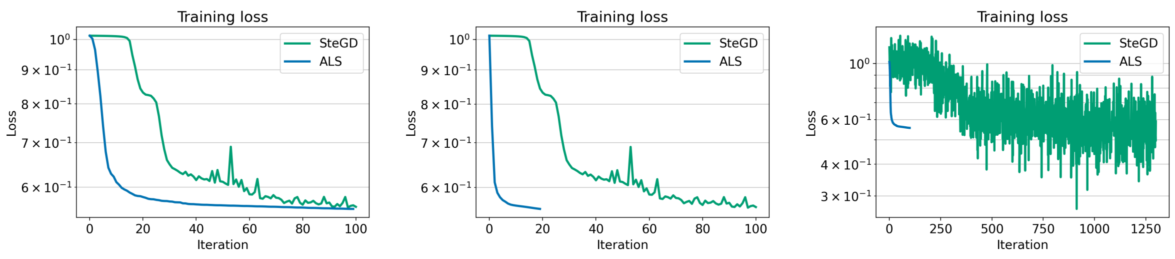
2. One iteration is defined as one weight update.

More definitions are possible, but these two are used as examples. To demonstrate how this influences the perspective on the results, the results of an experiment are shown for the used definition and the two alternatives. The mini-batch SteGD method and Alternating Least Squares (ALS) are applied on the Airfoil data set with the hyperparameters given in Section 4-3-1. These two methods are used, since they lead to a clear demonstration. For Alternative 2 only, the training loss for the mini-batch method is evaluated for the mini-batch that was used to perform the weight update.

The results are shown in Figure B-1. It can be seen that Alternative 1 has similar results as the used definition, only the ALS training loss is squeezed together. It was chosen not to use this definition, since detail is lost due to this squeeze. Furthermore, with ALS often 5, 10 or 20 twenty sweeps are performed. A sweep is D updates such that all the factors are updated once. On the other hand, for SteGD around a hundred passes over the training data are usually performed which implies a hundred iterations. As a result, Alternative 1 leads to a difference between the amount of iterations that are performed by the two methods. This could make it more difficult to compare the methods.

Alternative 2 also leads to a difficult comparison of the methods. Since the loss of the mini-batch method is evaluated on a batch, the training loss shows a lot of oscillations. This makes is difficult to see a clear trend in the loss, especially when the batch size is small compared to the training set size. Moreover, the mini-batch method performs significantly more updates than ALS, so has a lot more iterations with Alternative 2 as iteration definition. This further complicates the comparison. The latter would also hold when the training loss would have been evaluated on the whole training set.

So, the used definition allows for enough detail to compare the methods, but does not result in an information overload. Furthermore, with the used definition both methods converge within the same order of iterations. As a result, an equal amount of iterations can be used.



(a) Used definition. **(b)** Alternative 1. **(c)** Alternative 2.

Figure B-1: Illustration of the effect of the definition of an iteration for the training loss on the Airfoil data set with the mini-batch SteGD method and ALS. The definition used in this thesis is shown as well as the two listed alternatives.

B-2 Instability during training

It was found in Section 4-2 that the SteGD method could exhibit instabilities. The instability is briefly investigated here. The full-batch SteGD method is applied to the Airfoil data set with the parameters listed in Section 4-3-1. The norm of the Canonical Polyadic Decomposition (CPD) is registered as well as the gradients. The angle between two gradients ,$\mathcal{G}_1$, $\mathcal{G}_2$, is computed by vectorizing the tensor gradient:

$$\chi = \frac{\langle \text{vec}(\mathcal{G}_1), \text{vec}(\mathcal{G}_2) \rangle}{\|\mathcal{G}_2\|_F \, \|\mathcal{G}_2\|_F}. \tag{B-1}$$

The angle between the gradient vector at iteration 475 and the gradient that particular iteration is computed. It was chosen to use an absolute angle instead of a relative angle so that the change in gradient after the instability can also be seen.

The results are shown in Figure B-2. It can be seen that the spike in training loss is accompanied by a change in the gradient direction and sharp decrease in the CPD norm. The gradient direction first starts changing and then the spike loss follows. The change in direction is almost half a pi, which implies that the gradient is perpendicular to what it was before. Moreover, after the spike the gradient is still direction to what it was before, because there is still an absolute difference. The results suggest that it is likely that the spike is an overshoot. It is imagined that there might be an overshoot occurring in sort of a saddle point. After the overshoot has been correct the gradient points in a different direction, since it is continuing in the other direction of the saddle point. The progress of the gradient direction gives an interesting glance at the CPD optimization landscape.

This instability is only shortly studied here, but could be studied in more detail in future research.

B-3 Mini-batch training

In this section it is shown that the Line search method and the Adam method can be used with mini-batch gradient descent. In Chapter 5 it was already shown that the methods work with full-batch gradient descent.

The same experiments as in Section 5-1-2 and Section 5-1-3 are conducted, but now with both full-batch and mini-batch gradient descent. The results are shown in Figure B-3 and Table B-1. Note that with the used definition of one iteration, see Definition 4.1, one iteration in with a mini-batch method corresponds to multiple weight updates.

From the results it can be seen that both the Line search method and the Adam method converge faster at first. For the Line search method the loss oscillates more once the training has almost converged. This is most likely because the method overfits on the mini-batch due to the optimality of the method. As a result, the overall loss worsens. Nevertheless the mini-batch version of the Line search method allows the method to escape local minima which the full-batch method cannot. This could be a benefit and could be investigated further in future research.

For the Adam method the full-batch and mini-batch convergence is similar later in the training process. There are oscillations, but they are less severe than with the Line search method. Moreover, the authors of the Adam method also suggest to use mini-batch gradient descent. So, it is concluded that the Adam method can effectively be combined with mini-batch gradient descent to speed up the training. In Chapter 4 it is shown that the mini-batch Adam method performs well for large scale learning.

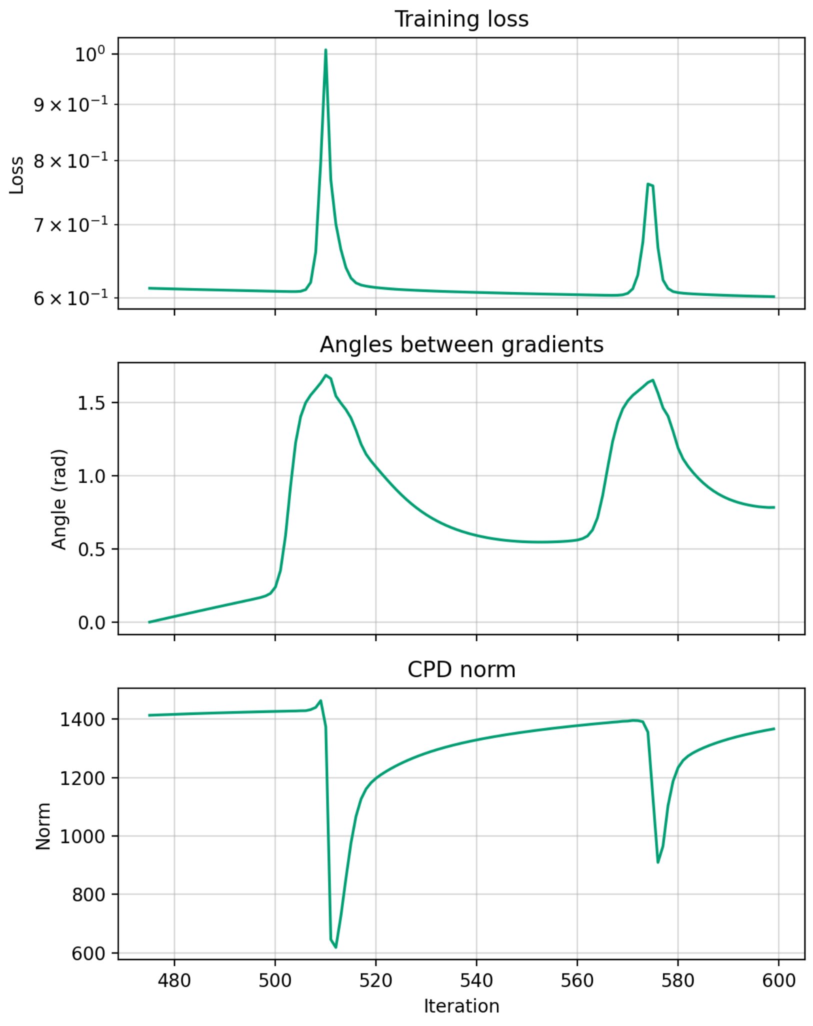


Figure B-2: Training progress during training on the Airfoil data set with the full-batch SteGD method. For the angles between gradients, the angle between the gradient at iteration 475 and that particular iteration is shown.

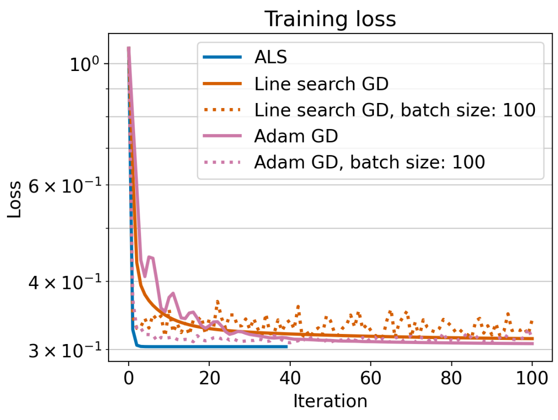
B-4 Line search with a large number of features

In Section 6-2-1 it is shown that the Adam method no longer performs well when the data set has around 16 or more features. It is suggested that this is due to the exceedingly small gradient. To further investigate this, the Line search method is applied to a data set with 18 features. A different data set with less samples is used to limit the training run time of the Line search method.

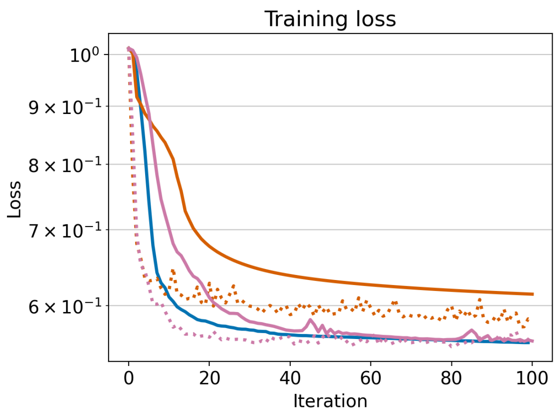
The Statlog Vehicle data set is an UCI data set and originally has 846 samples and four classes. To create a binary classification task, the samples of two classes are discarded. This results in data set with 435 samples and 18 features. For the hyperparameters the values $R = 5$, $M = 20$, $U = 1$, $l = 0.1$ and a validation split of 0.1 are used. For the Adam method $\alpha_{adam} = 0.05$ is used and for its other hyperparameters the values suggested by its authors are used.

Table B-1: Final losses for different methods after training on the Banana and Airfoil data set for full-batch and mini-batch methods.

	Banana		Airfoil	
Method	Training	Validation	Training	Validation
ALS	**0.303**	**0.296**	**0.557**	**0.465**
Line search GD full-batch	0.314	0.312	0.614	0.523
Line search GD mini-batch	0.340	0.349	0.582	0.493
Adam full-batch	0.307	0.302	0.558	0.473
Adam mini-batch	0.314	0.303	0.566	0.483



(a) Banana data set.



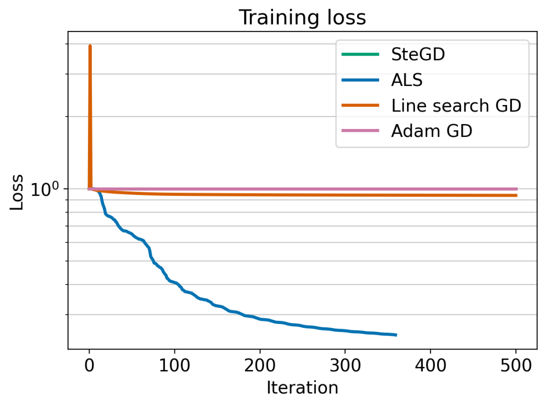
(b) Airfoil data set.

Figure B-3: Training loss for different methods during training on the Banana and Airfoil data set.

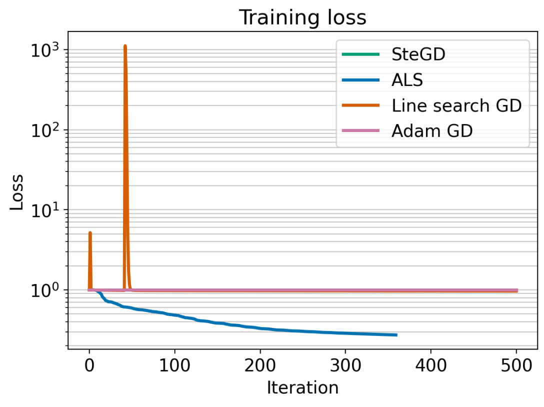
For the regularization parameter λ different orders of magnitude were tried to investigate whether successful training could be achieved. The values $\lambda \in [10^{-15}, 10^{-17}, 10^{-19}, 10^{-21}]$ are used. The intermediate orders are also tried, but their results show a similar trend, so they are not shown.

The results are shown in Figure B-4. The training of the ALS method succeeds in all cases. The training of the Adam method makes no progress at all. In all cases does the Line search method not train successfully. For $\lambda = 10^{-15}$ some progress is made, but very little compared to ALS. Moreover, the Line search method shows strange behaviour. The training loss of the Line search method should not be able to increase due to the optimality of the method. A step size that increases the loss cannot be chosen. It is, therefore, peculiar that it does. This could suggest that there are floating-point issues that result in the increases. The small gradient values combined with the recursive nature of the computation of the step size could result in such small values that floating-point errors become dominating. Alternatively, it could of course be an issue in the implementation. It is implemented that no improvement is also possible, but the implementation did not indicate that it found no improvement in any of the cases.

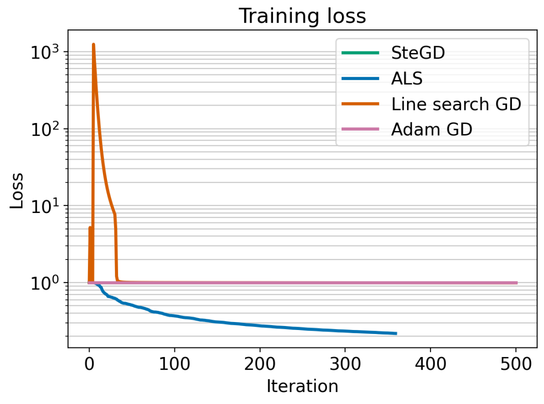
It is noted that the ALS method, and the Line search method for $\lambda = 10^{-15}$, resulted high validation losses. This is likely due to the small data set size and consequently the ease of overfitting.



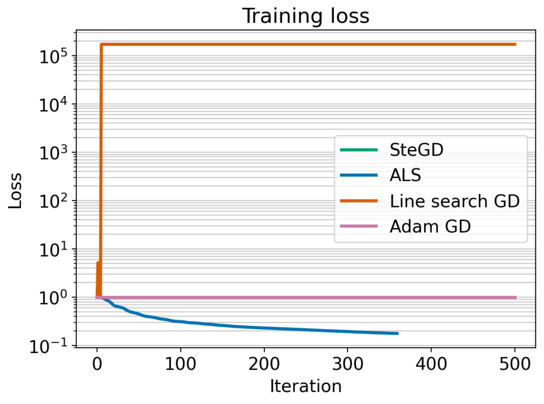
(a) $\lambda = 10^{-15}$.



(b) $\lambda = 10^{-17}$.



(c) $\lambda = 10^{-19}$.



(d) $\lambda = 10^{-21}$.

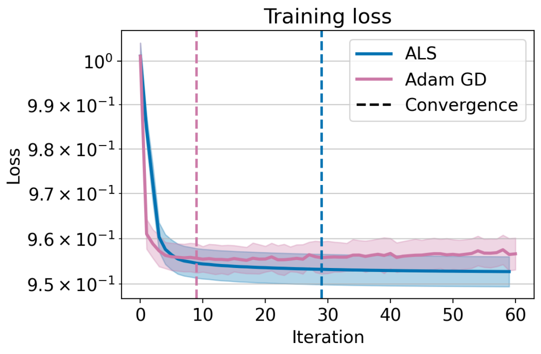
Figure B-4: Training loss on the Vehicle data set for different methods and regularization parameters.

B-5 General performance results

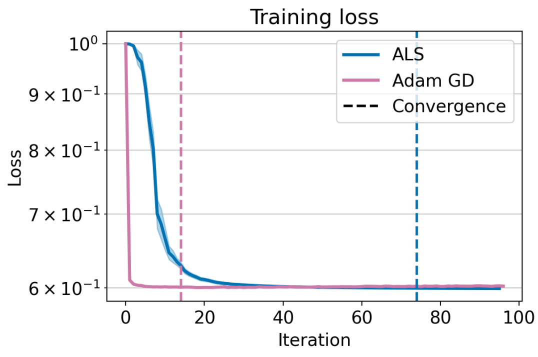
In this section the results of the experiments conducted in Section 6-2-2 are shown for the data sets that were not shown in Section 6-2-2. So, the results for the Precipitation, SUSY Low, SUSY High and HIGGS Highs data set are shown.

B-6 Results for HouseElectric data set for different CP-ranks and feature map orders

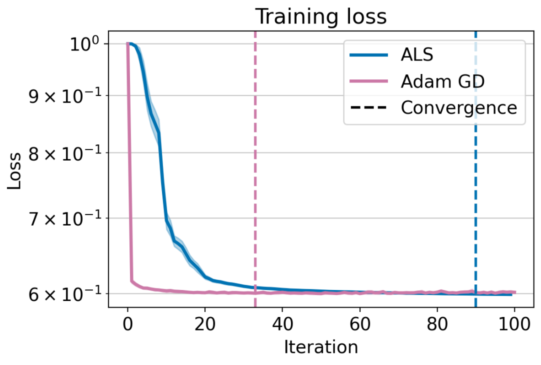
In Section 6-2-3 the Adam method and ALS are applied to the HouseElectric data set for different values of the feature map order M and CP-rank R. In that section only the training run times are reported. Here the losses are given. The experiment setup is given in Section 6-2-3.



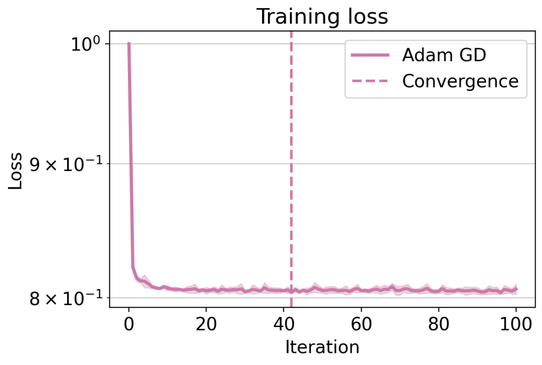
(a) Precipitation.



(b) SUSY Low.



(c) SUSY High.



(d) HIGGS High.

Figure B-5: Mean training loss (line) and one standard deviation (shaded area) on different data sets with Adam and ALS. The identified convergence points for each method are also drawn.

The training losses are given in Figure B-6 and Figure B-7. It can be seen that the relative progress is similar for the different combinations of M and R. Only the loss that the methods converge to changes notably. The final losses are given in Table B-2. It can be seen that the mean losses are identical or differ at most a two thousandths for all combinations. So, the Adam method and ALS have a similar performance in terms of loss for all combinations of M and R.

B-7 Data preparation

Several data sets are used in this thesis. In this section it will be shown where the data sets can be accessed. Furthermore, any data preparation is also discussed. This is, for example, the splitting or removing of features. However, it does not include preprocessing of the data itself, such as the normalization of the output.

Vehicle data set The Vehicle data set is an UCI data set which can be accessed at `https://archive.ics.uci.edu/ml/datasets/Statlog+%28Vehicle+Silhouettes%29`. The data set on the UCI website contains 9 separate files that need to be merged. Furthermore, the data set contains four classes. All the samples that correspond to the 'Van' and 'Opel' class are discarded. The remaining data set has 435 samples.

		Training loss		Validation loss	
R	M	Adam	ALS	Adam	ALS
10	10	**0.197** (0.0004)	**0.197** (0.0005)	**0.197** (0.0008)	**0.197** (0.0009)
	20	**0.163** (0.0002)	**0.163** (0.0003)	**0.163** (0.0008)	**0.163** (0.0006)
	40	0.156 (0.0007)	**0.155** (0.0004)	**0.155** (0.0008)	**0.155** (0.0009)
20	10	0.195 (0.0002)	**0.194** (0.0002)	0.196 (0.0008)	**0.195** (0.0008)
	20	**0.162** (0.0003)	**0.162** (0.0002)	**0.162** (0.0009)	**0.162** (0.0007)
	40	0.154 (0.0003)	**0.153** (0.0002)	0.154 (0.0007)	**0.153** (0.0008)
40	10	0.194 (0.0002)	**0.192** (0.0003)	0.195 (0.0009)	**0.193** (0.0007)
	20	0.162 (0.0003)	**0.161** (0.0001)	0.162 (0.0009)	**0.161** (0.0007)
	40	**0.153** (0.0002)	**0.153** (0.0001)	**0.153** (0.0007)	**0.153** (0.0007)

Table B-2: Mean final loss and one standard deviation for the Adam method and ALS on the HouseElectric data set for different values of M and R.

Banana data set The Banana data set can be accessed at https://www.kaggle.com/code/saranchandar/standard-classification-with-banana-dataset/data. The class labels are already $[-1,\ 1]$, so no additional preparation is needed.

Airfoil data set The Airfoil data set is an UCI data set and can be accessed at https://archive.ics.uci.edu/ml/datasets/airfoil+self-noise. No preparation is needed.

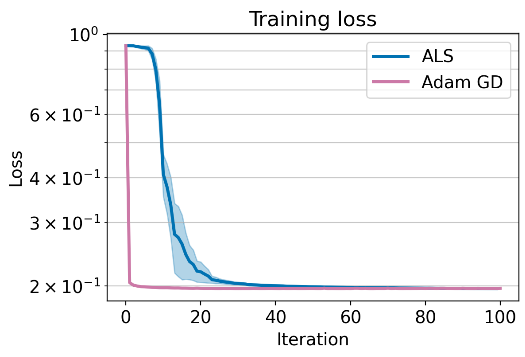
Elevators data set The Elevators data set seems to be an old UCI data set. It can be accessed at https://github.com/treforevans/uci_datasets/tree/master/uci_datasets/elevators. No preparation is needed.

Protein data set The Protein data set refers to the UCI data set ‘Physicochemical Properties of Protein Tertiary Structure Data Set’. It can be accessed at https://archive.ics.uci.edu/ml/datasets/Physicochemical+Properties+of+Protein+Tertiary+Structure.

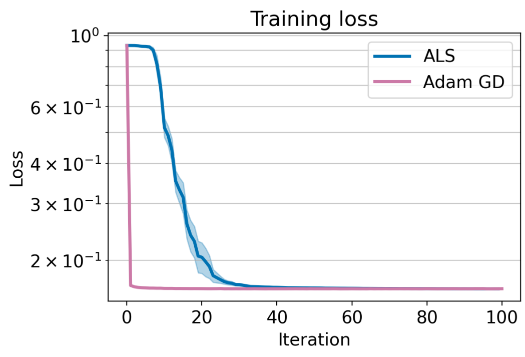
Precipitation data set The Precipitation data set was obtained from Frederiek Wesel, the author of [6]. The preparation had already been done.

HouseElectric data set The HouseElectric data set refers to the UCI data set ’Individual household electric power consumption Data Set’ and can be accessed at https://archive.ics.uci.edu/ml/datasets/Individual+household+electric+power+consumption. This data set requires some preprocessing.

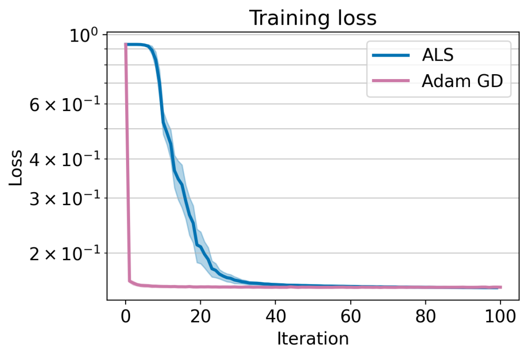
The first two columns/features of the data set are the date and time stamp respectively. The date stamp is split into a day and a month. The time stamp is split into an hour and a minute. These are included as features, but the date stamp and time stamp itself are discarded.



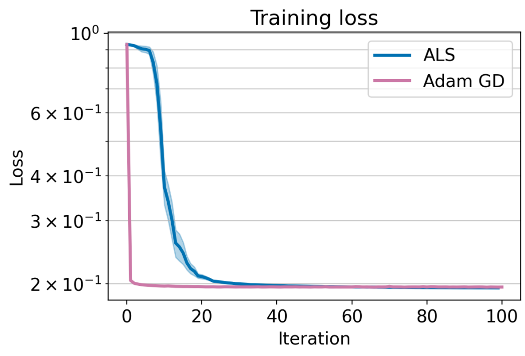
(a) $R = 10,\ M = 10$.



(b) $R = 10,\ M = 20$.



(c) $R = 10,\ M = 40$.



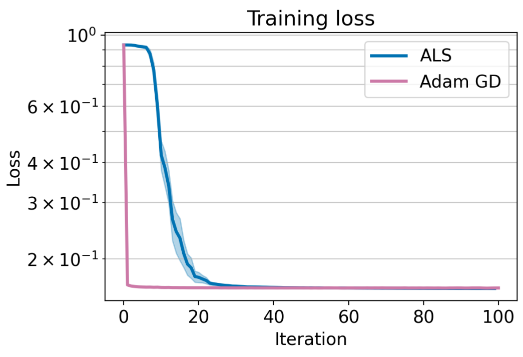
(d) $R = 20,\ M = 10$.

Figure B-6: Mean training loss (line) and one standard deviation (shaded area) on HouseElectric data set with Adam and ALS for different values of the feature map order M and CP-rank R.

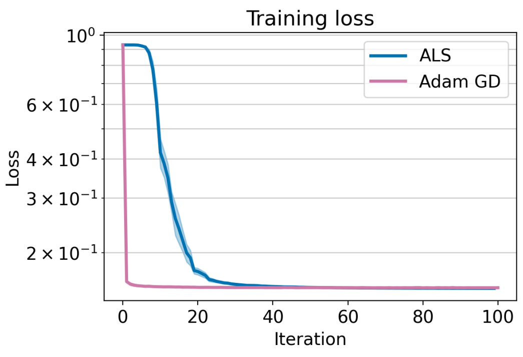
SUSY Low, High and Full The SUSY data set [55] can be accessed at https://archive.ics.uci.edu/ml/datasets/SUSY. It has 8 low level features and 10 high level features. To create SUSY Low, the first 8 features are used, for SUSY High the last 10 are used and for SUSY Full all the features are used.

Airline The Airline data set was obtained from Frederiek Wesel, the author of [6]. The preparation had already been done.

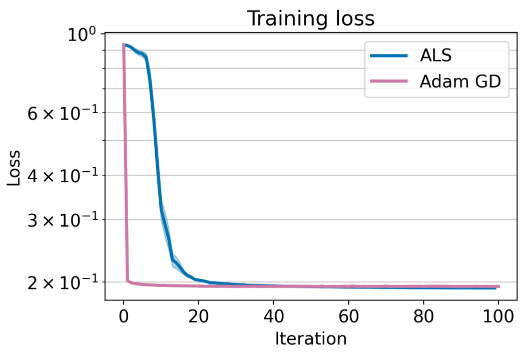
HIGGS High The HIGGS High data set is obtained from the HIGGS data set [55] which can be accessed at https://archive.ics.uci.edu/ml/datasets/HIGGS. The full data set has 28 features. In this thesis only the high level features are used which results in the HIGGS High data set with 7 features.



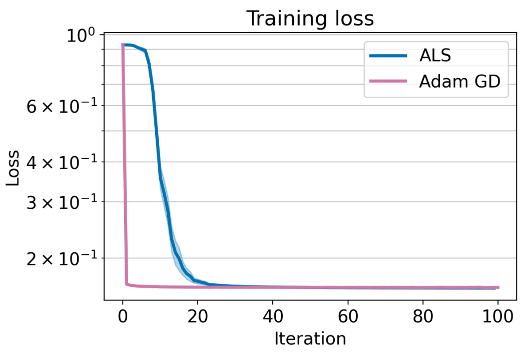
(a) $R = 20,\ M = 20.$



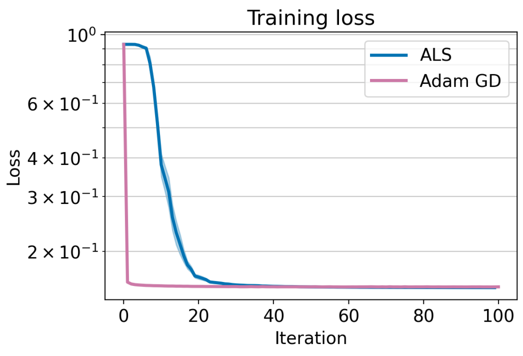
(b) $R = 20,\ M = 40.$



(c) $R = 40,\ M = 10.$



(d) $R = 40,\ M = 20.$



(e) $R = 40,\ M = 40.$

Figure B-7: Mean training loss (line) and one standard deviation (shaded area) on HouseElectric data set with Adam and ALS for different values of the feature map order M and CP-rank R.

Appendix C

Python implementations

In this chapter of the appendix details related to the Python implementations are discussed in more detail. First, Just-In-Time compilation is discussed. Then it is shown how a Canonical Polyadic Decomposition (CPD) can be represented as a tensor under certain assumptions. It is demonstrated with an experiment that the representation can significantly decrease the run time of CPD functions. Thirdly, automatic differentiation (AD) in Python is discussed. Finally, the effect of the floating point precision on the training is investigated with experiments.

C-1 Just-In-Time compilation

Just-In-Time compilation, jitting in short, means that a function is compiled during the execution of the program such that the compiled function can be reused for additional calls to the function. The execution of the compiled function is in general much faster, which makes jitting useful.

As long as similar inputs are used a function only needs to be compiled once. Similar inputs are, for example, arrays of the same size and data type. The compilation takes time, so it is important that similar data is used. The weight update function in an iterative optimization scheme is an example of such a function. When input data of the same size and shape is used, the weight update only needs to be compiled once and can then be reused for the entire training process. Thus, the benefit of jitting code is that it can result in significant reduction of the training time.

An additional major benefit of the jit functionality in JAX, is that it automatically allows for execution of the code on a GPU or TPU. This results in an additional significant speed up of the training time. No extra adjustments of the code are needed. It is only, at the moment, not possible to compute the eigenvalues of a non-symmetric matrix on a GPU with JAX. As a result, the Line search method which uses this functionality, can not be run on a GPU.

A downside of the JAX package and its jit functionality is that it requires functionally pure code. So, any function that is to be jitted needs to follow the functional-programming paradigm. This entails that writing code with JAX can be tricky at times. However, almost all the functionality required for this thesis could be implemented with JAX.

For more information about jitting and JAX, see the JAX documentation and tutorials.

C-2 CPD as a tensor

A straightforward way to represent a CPD as a variable in a software program is to have a list of the factor matrices $\mathbf{A}^{(d)}$. To perform computations with the CPD the program would then needs to loop over this list to access the individual factors. Alternatively, with the assumption that each dimension of the tensor has the same size, $I_1 = I_2 = \ldots = I_D = I$, the CPD can also be represented as a tensor.

When the dimensions of the tensor all have the same size I, all factors have the same size, $\mathbf{A}^{(d)} \in \mathbb{R}^{I \times R},\ d = 1, \ldots,\ D$. The CPD can then be represented as a tensor by stacking all factors along a new dimension. Note that this is not possible when the factors have different sizes. The tensor representation of a CPD is illustrated in Figure C-1. It can be seen that the factors are stacked along the first axis. As a result, the first axis is of size D. Overall, the tensor representation of the CPD is a tensor of size $D \times I \times R$. This particular order of axes, number of dimensions, tensor dimension and CP-rank, will be used throughout this report.

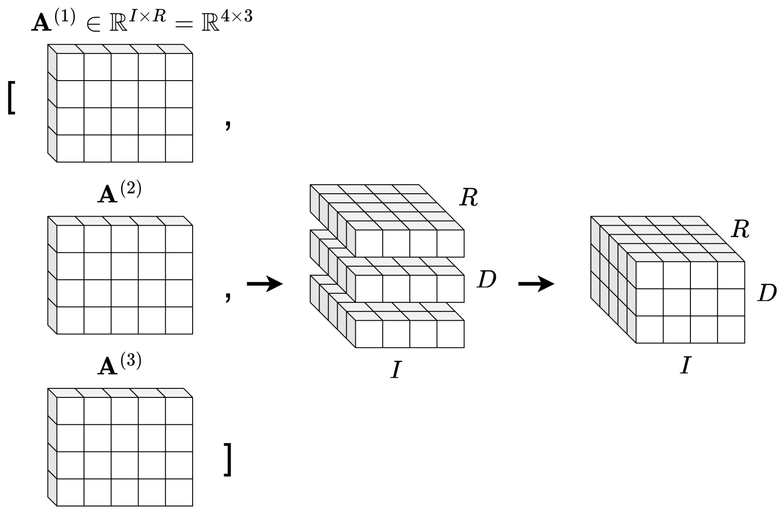

$\mathbf{A}^{(1)} \in \mathbb{R}^{I\times R} = \mathbb{R}^{4\times 3}$
$\mathbf{A}^{(2)}$
$\mathbf{A}^{(3)}$
R
D
I
R
D
I

Figure C-1: The representation of a CPD as a $D \times I \times R$ tensor. It is assumed that the factors of the CPD all have the same size.

The benefit of this representation is that the CPD can now to be stored as a Numpy/JAX `ndarray`. Consequently, the standard Numpy/JAX functions can be used to perform the operations and no loops are required. Since these functions are written and optimized in C code, this results in faster operations[1]. The Python implementation of the Frobenius norm of (2-25) is shown in Listing C.1 to illustrate how the basic Numpy/JAX functions can be used.

[1]See https://numpy.org/doc/stable/user/whatisnumpy.html#why-is-numpy-fast.

Listing C.1 Python implementation of the Frobenius norm of a CPD that is represented as a $D \times I \times R$ tensor.

```python
# factors is DxIxR ndarray
def cpd_norm(factors):
    # Transpose factors by swapping axis I and R axis (with zero-indexing)
    factors_T = jnp.swapaxes(factors, 1, 2)

    # Compute matrix products
    gamma = jnp.matmul(factors_T, factors)

    # Hadamard product along D axis
    gamma_full = jnp.prod(gamma, axis=0)

    # Compute squared norm by summing
    norm = jnp.sum(gamma_full)
    return norm
```

To illustrate the effect of representing the CPD as a tensor, the run time of Listing C.1 is compared with the run time of the same function that uses the CPD as a list. The implementation of the latter is given in Listing C.2. Both functions are executed a 1000 times for the same random CPD and the total run time is recorded. In every case factors of the same size $I = 20$, $R = 10$ where used, but a different number of dimension D was used. A different number of dimensions was used specifically, because this is the axis that a list representation has to loop over. For each CPD size ten experiments are conducted. The mean run time and one standard deviation are given in Table C-1. Besides the implementations given in Listing C.1 and Listing C.2, the jitted[2] version of these function are also timed. The experiments are run on a CPU.

It can be seen that there is a serious reduction in the run time when the CPD is represented as a tensor compared to the list implementation for the normal Python implementation. Additionally, it can be concluded that jitting the function speeds up the implementation significantly for both implementations. Nevertheless, even after jitting the tensor implementation is still faster. Especially for a larger number of dimensions this is the case. This is in line with what was expected, because for a larger number of dimensions the list implementation needs to loop longer.

Since the computation of the Frobenius norm of a CPD is only one of several operations used in this thesis that involve looping over the factors, the tensor representation of the CPD is used in this thesis to speed up the run time.

C-3 Automatic differentiation in Python

Since AD is used for functions implemented in a software program, AD itself is naturally also implemented as a software program. As mentioned Section 1-3 Python is used in this

[2]Using the JAX jit functionality. See Section 1-2 for more details

Table C-1: Mean run time and one standard deviation for computing the squared Frobenius norm of a CPD a 1000 times for two representations of a CPD.

	Run time (s)		
Implementation	$D = 5$	$D = 10$	$D = 20$
CPD as tensor	0.0473 (0.00574)	0.0530 (0.00281)	0.0765 (0.0255)
CPD as list	1.10 (0.148)	1.88 (0.263)	3.31 (0.220)
CPD as tensor, jitted	**0.0162** (0.00153)	**0.0227** (0.00337)	**0.0320** (0.00419)
CPD as list, jitted	0.0191 (0.00137)	0.0300 (0.00392)	0.0432 (0.00403)

Listing C.2 Python implementation of the squared Frobenius norm of a CPD that is represented as a list of D factors.

```python
# factors is list of D IxR ndarrays
def cpd_norm(factors_list):
    gamma = 1

    # Loop over factors
    for factor in factors_list
        gamma = gamma * jnp.matmul(factor.T, factor)

    # Compute squared norm by summing
    norm = jnp.sum(gamma)
    return norm
```

thesis and especially the JAX library. The JAX library offers AD functionality and that is one of the reasons it was opted to use it. Nevertheless, several other Python libraries offer AD functionality. A short overview of TensorFlow [57], Autograd/JAX [58, 9] and PyTorch [59, 60] and their AD functionality is given in the next paragraphs. For a more general overview of AD tools for different programming languages the reader is advised to read [45].

TensorFlow TensorFlow uses a graph to represent a model. When implementing a model in TensorFlow the user uses Python as a metalanguage to create this model out of existing nodes [61]. To use AD with TensorFlow, the TensorFlow syntax has to be used. As a result, any existing code based on Numpy cannot be directly used and has to be rewritten. The basic operations such as matrix multiplication and the dot product are available as TensorFlow functions, so the objective function of Eq. (3-11) can be implemented in TensorFlow. Only the feature map $\mathcal{Z}$ might be more difficult to implement in TensorFlow. However, this could possibly be done in Numpy and then it can be casted to a TensorFlow variable, since no derivatives with respect to the data are needed. Additionally, by default TensorFlow has eager execution, but it is possible to use graphs for custom functions as well. This can be advantageous for the performance. Lastly, using TensorFlow has the additional benefit that many iterative optimization schemes are already implemented in TensorFlow. As a result, they can be used directly.

Autograd/JAX Autograd, developed at the Harvard Intelligent Probabilistic Systems Group (HIPS) is a tool that enables AD for standard Python and Numpy code. So with Autograd, it is possible to use most of the functionalities of Numpy and apply AD directly. Functions written using Numpy can in most cases be used directly by using Autograd's Numpy wrapper instead of Numpy itself. Most Numpy functions are implemented, but there are some things that need to be taken into account. See the Autograd documentation on Github for more details. For example, in-place operations such as `a -= b` as should not be used. The explicit version `a = a - b` should be used instead. However, besides these technicalities it is possible to directly obtain gradients without having to adapt already existing Numpy code.

Currently, the main developers of Autograd are working on JAX, which combines a newer version of Autograd with XLA [62]. XLA is compiler specifically designed for linear algebra. Besides the AD functionality, JAX can be especially useful for speeding up execution times and allows for code to be run on a GPU and a TPU as well. To make use of these functionalities additional programming constraints need to be taken into account.

PyTorch Similarly as with TensorFlow, to use AD with PyTorch [59, 60] the PyTorch syntax and functions needs to be used. PyTorch has the basic operations implemented. The objective function can therefore be implemented and its gradient can be computed. Furthermore, the many iterative optimization schemes are also readily available in PyTorch.

C-4 Effect of floating point precision

In Section 5-1 it is discussed that the gradient values can be exceedingly small. It is, therefore, investigated whether there is a difference between using a 32-bits or a 64-bits representation. By default JAX uses 32-bit precision, but this can be adjusted to 64-bits precision. The downside of using 64-bit precision is that it requires more memory and computations can take longer.

To study the difference between 32-bit and 64-bit precision training is performed with both precisions. This was tested on the Airfoil data set for the Steepest Gradient Descent (SteGD) method, the Line search method, the Adam method and Alternating Least Squares (ALS). Additionally, it is tested on the Protein data set for the Adam method and the ALS method. For the hyperparameters, the values listed in Table 4-1 and Table 6-2 are used for the Airfoil and Protein data set respectively. In all cases the full-batch method was used. For each method ten experiments were run.

The results are shown in Figure C-2 and Figure C-3. It can be seen that for ALS the differences are relatively small. However, for the Line search method and SteGD there is quite a significant difference. For the Adam method the difference is smaller. However, on the Protein data set 60% of the experiments with 32-bit precision resulted in NaN's[3]. They are left out of the plot, but this implies that with this precision the training can fail.

From these results, it is concluded that 64-bit precision must be used to fairly evaluate the methods and to ensure that the training does not fail. The added computational and memory load are taken for granted.

[3]NaN: Not a Number; this means that the training has failed

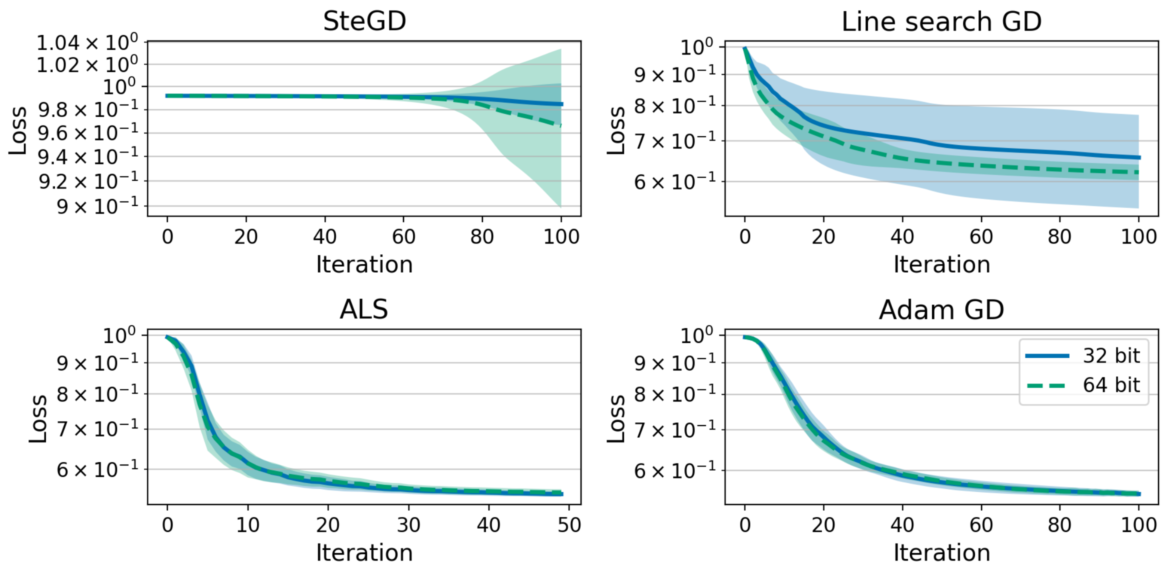


Figure C-2: Mean training loss (line) and one standard deviation (shaded area) on Airfoil data set for different methods and floating point precisions.

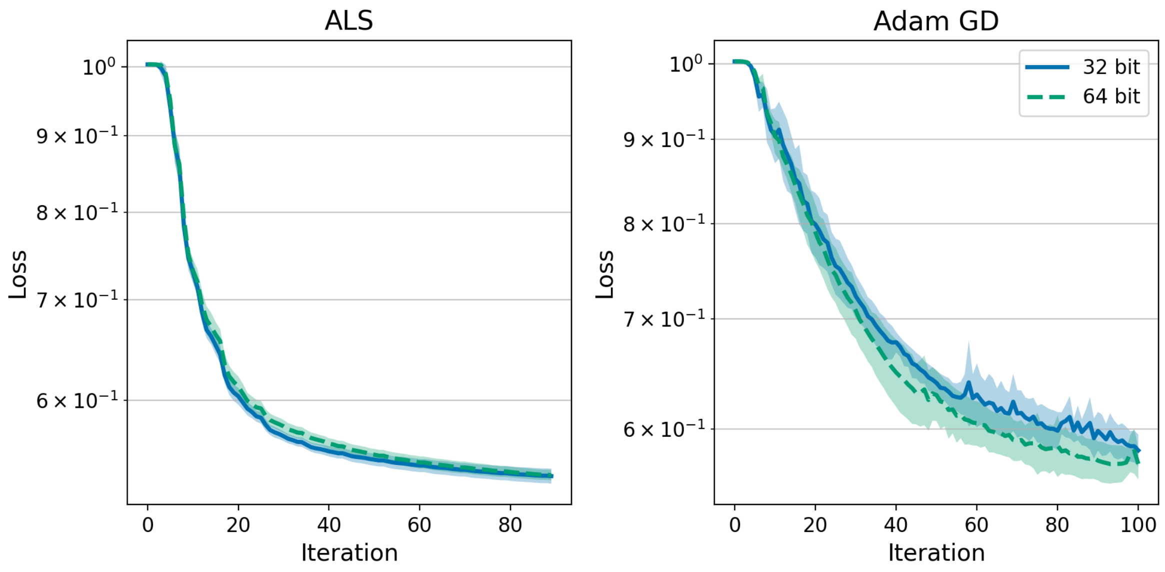


Figure C-3: Mean training loss (line) and one standard deviation (shaded area) on Protein data set for different methods and floating point precisions. Experiments that contained NaN losses were omitted.

Appendix D

Feature map

In this section several aspects of the Fourier feature map are discussed in more detail. For convenience the Fourier feature map that is used throughout this thesis is repeated here [26, 6]:

$$\begin{aligned}\left(\mathbf{z}(x^{(d)})\right)_m &= \frac{1}{\sqrt{U_d}}\sqrt{S}\sin\left(\frac{\pi m(x^{(d)}+U_d)}{2U_d}\right),\\ S &= \sqrt{2\pi}\cdot l\cdot\exp\left(\frac{-\pi^2 m^2 l^2}{8U_d^2}\right).\end{aligned} \tag{D-1}$$

In this thesis the same values for U_d are used for all dimensions, so $U_d = U,\ d = 1,\dots,D$.

First, it is shown that this feature map can approximate the radial basis function (RBF) kernel and it is visualized what the influence of the feature map parameter M is on this approximation. Secondly, the output values of the feature map are discussed. Specifically, the magnitude of the output is analyzed.

D-1 Kernel approximation

In Section 3-1-2 the concept of product feature maps is introduced. Product feature maps are useful, because their structure can be combined with the Canonical Polyadic Decomposition (CPD) structure. However, product feature maps are also useful, because they can be applied for kernel approximation. This is not discussed in the main body of this thesis, since it is not the focus of this thesis. It will shortly be deliberated here.

Kernel approximation entails that a feature map is used primal space such that it approximates a given kernel were the problem to be solved in the dual space. So, a feature map can be used to a resemble the behaviour of a kernel and thus benefit from its useful properties. For more information about kernel approximation, see [63].

The product feature map $\mathbf{z}(\cdot)$ given in Eq. (D-1) can be used to approximate the RBF kernel [26]. This feature map is especially useful for kernel approximation, because the approximation error bound decreases exponential in M [26, Theorem 8]. This is another reason to specifically use product feature maps.

So, accuracy of the approximation depends on the feature map order M. For the one dimensional case this can be visualized. The RBF kernel is given by [25]

$$k(x, x') = e^{\frac{-\|x-x'\|^2}{2l^2}}. \tag{D-2}$$

To visualize the value of the kernel the given formula is evaluated for $x' = 0$ and different values of x. Furthermore, the kernel approximation is also computed using $\langle \mathbf{z}(x), \mathbf{z}(x') \rangle$. This is done for different values of M. For the length-scale l the value 0.1 is used and a bound of one, $U = 1$, is used. In Figure D-1 it can be seen that for a higher value of M approximation becomes better. For this length-scale the approximation is already very close for $M = 16$.

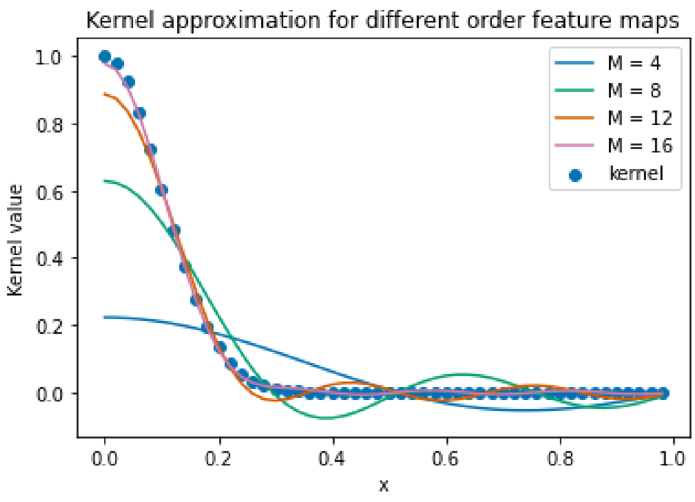


Figure D-1: Approximation of the RBF kernel for feature maps with a different order M. The kernel is evaluated for $x,\ x'$ where $x' = 0$.

D-2 Feature map value

In Section 3-1-2 it is mentioned that the value of the Fourier product feature map is smaller than one when $U = 1$. This is further worked out here. The bound $U = 1$ is reasonable, since this entails that the input $x^{(d)}$ is scaled to to lay within -1 and 1 for each feature $d = 1, \ldots, D$. This is a common practice within the machine learning community. As a result, the amplitude of the feature map is given by $\sqrt{(S)}$ with S defined in Eq. (D-1). From the expression for S it can be deduced that for $m = 1$ the amplitude is the largest. Hence, the largest amplitude is given by

$$\sqrt{\sqrt{2\pi} \cdot l \cdot \exp\left(\frac{-\pi^2 l^2}{8}\right)}. \tag{D-3}$$

In Figure D-2 the value of $\sqrt{S}$ is plotted as a function of l. The maximum value occurs at $l = 0.64$, $\sqrt{S} = 0.9838$.

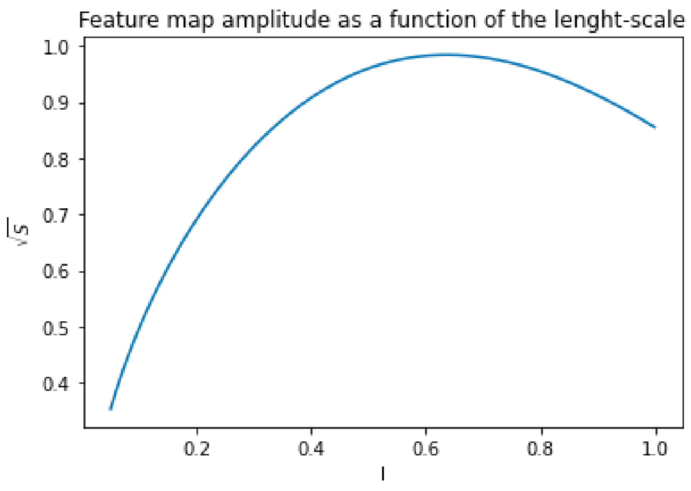
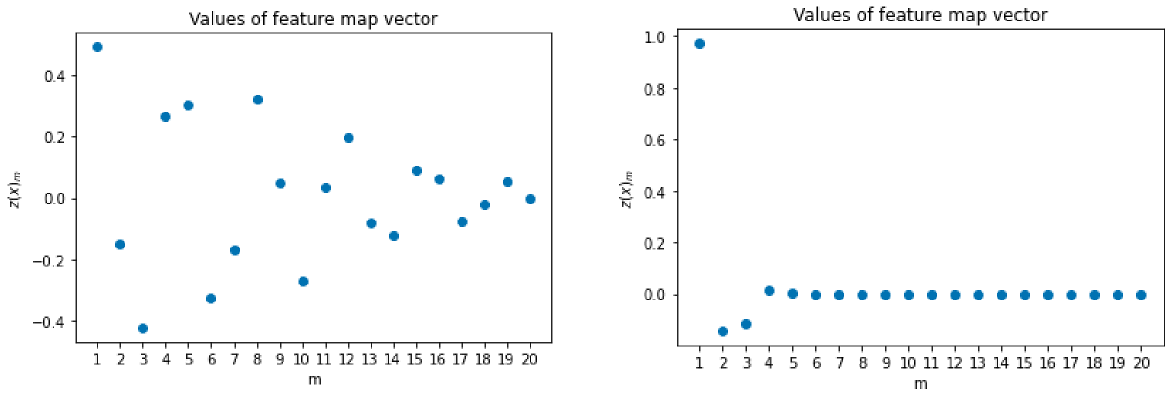


Figure D-2: Amplitude $\sqrt{S}$ of feature map as a function of the length-scale l for $U = 1$ and $m = 1$.

To analyze the values of the feature vector $\mathbf{z}(x)$, the values are plotted for two values of l assuming that $U = 1$. In Figure D-3 the values of the feature map are shown for $l = 0.1$ and $l = 0.64$ for the input $x = 0.1$. The input $x = 0.1$ is used for this illustration, because for $x = 0$ the sine in the feature map is zero half of the time. It can be seen that the values of the feature vector are always smaller than one. For $l = 0.1$ the values are in between -0.5 and 0.5. When l is larger, a large part of the feature vector is close to zero. A value of $l = 0.1$ is used, because the input bounds U_d are one. So, the length-scale is one tenth of the bound which is deemed reasonable.



(a) $l = 0.1$.

(b) $l = 0.64$.

Figure D-3: Values of feature vector $\mathbf{z}(x)$ for two different length-scales for the input $x = 0.1$ with $U = 1$.

Bibliography

[1] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language Models are Few-Shot Learners," 5 2020.

[2] M. Collins and N. Duffy, "Convolution Kernels for Natural Language," in *Advances in Neural Information Processing Systems* (T. Dietterich, S. Becker, and Z. Ghahramani, eds.), vol. 14, MIT Press, 2001.

[3] T. Hofmann, B. Schölkopf, and A. J. Smola, "Kernel methods in machine learning," *The Annals of Statistics*, vol. 36, no. 3, p. 1171 – 1220, 2008.

[4] B. E. Boser, I. M. Guyon, and V. N. Vapnik, "Training algorithm for optimal margin classifiers," *Proceedings of the Fifth Annual ACM Workshop on Computational Learning Theory*, pp. 144–152, 1992.

[5] F. L. Hitchcock, "The Expression of a Tensor or a Polyadic as a Sum of Products," *Journal of Mathematics and Physics*, vol. 6, pp. 164–189, 4 1927.

[6] F. Wesel and K. Batselier, "Large-Scale Learning with Fourier Features and Tensor Decompositions," in *Advances in Neural Information Processing Systems* (A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, eds.), 2021.

[7] N. Kargas and N. D. Sidiropoulos, "Supervised Learning and Canonical Decomposition of Multivariate Functions," *IEEE Transactions on Signal Processing*, vol. 69, pp. 1097–1107, 2021.

[8] D. P. Kingma and J. L. Ba, "Adam: A Method for Stochastic Optimization," *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, 12 2014.

[9] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, “JAX: composable transformations of Python+NumPy programs,” 2018.

[10] T. G. Kolda and B. W. Bader, “Tensor Decompositions and Applications,” *SIAM Review*, vol. 51, no. 3, pp. 455–500, 2009.

[11] C. Chen, K. Batselier, W. Yu, and N. Wong, “Kernelized support tensor train machines,” *Pattern Recognition*, vol. 122, p. 108337, 2 2022.

[12] T. G. Kolda, “SANDIA REPORT Multilinear operators for higher-order decompositions,” tech. rep., Sandia National Laboratories, Albuquerque, 4 2006.

[13] D. Cheng, H. Qi, and Y. Zhao, “Multi-Dimensional Data,” in *An Introduction to Semi-Tensor Product of Matrices and its Applications*, p. 22, World Scientific Publishing Co., 1 2012.

[14] J. E. Cohen, “About Notations in Multiway Array Processing,” 2016.

[15] L. R. Tucker, “Some mathematical notes on three-mode factor analysis,” *Psychometrika*, vol. 31, pp. 279–311, 9 1966.

[16] I. V. Oseledets, “Tensor-Train Decomposition,” *SIAM Journal on Scientific Computing*, vol. 33, no. 5, pp. 2295–2317, 2011.

[17] M. Bousse, O. Debals, and L. De Lathauwer, “A Tensor-Based Method for Large-Scale Blind Source Separation Using Segmentation,” *IEEE Transactions on Signal Processing*, vol. 65, pp. 346–358, 1 2017.

[18] R. A. Harshman, “Foundations of the PARAFARC procedure: Models and conditions for an“explanatory” multi-modal factor analysis,” *UCLA Working Papers in Phonetics*, vol. 16, pp. 1–84, 12 1970.

[19] J. D. Carroll and J. J. Chang, “Analysis of individual differences in multidimensional scaling via an n-way generalization of “Eckart-Young” decomposition,” *Psychometrika 1970 35:3*, vol. 35, pp. 283–319, 9 1970.

[20] J. B. Kruskal, “Three-way arrays: rank and uniqueness of trilinear decompositions, with application to arithmetic complexity and statistics,” *Linear Algebra and its Applications*, vol. 18, pp. 95–138, 1 1977.

[21] N. D. Sidiropoulos, L. De Lathauwer, X. Fu, K. Huang, E. E. Papalexakis, and C. Faloutsos, “Tensor Decomposition for Signal Processing and Machine Learning,” *IEEE Transactions on Signal Processing*, vol. 65, pp. 3551–3582, 7 2017.

[22] C. Saunders, A. Gammerman, V. Vovk, and R. Holloway, “Ridge Regression Learning Algorithm in Dual Variables,” *Proceedings of the 15th International Conference on Machine Learning*, 1998.

[23] B. Schölkopf, A. Smola, and K.-R. Müller, “Kernel principal component analysis,” in *Artificial Neural Networks — ICANN’97* (W. Gerstner, A. Germond, M. Hasler, and J.-D. Nicoud, eds.), (Berlin, Heidelberg), pp. 583–588, Springer Berlin Heidelberg, 1997.

[24] J. A. K. Suykens, T. Van Gestel, J. De Brabanter, B. De Moor, and J. P. L. Vandewalle, *Least Squares Support Vector Machines.* Singapore: World Scientific Publishing Company, 2002.

[25] J. A. K. Suykens and J. Vandewalle, "Least Squares Support Vector Machine Classifiers," *Neural Processing Letters*, vol. 9, no. 3, pp. 293–300, 1999.

[26] A. Solin and S. Särkkä, "Hilbert space methods for reduced-rank Gaussian process regression," *Statistics and Computing*, vol. 30, pp. 419–446, 3 2020.

[27] T. Dao, C. M. De Sa, and C. Ré, "Gaussian Quadrature for Kernel Features," *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[28] A. Vedaldi and A. Zisserman, "Efficient additive kernels via explicit feature maps," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 34, no. 3, pp. 480–492, 2012.

[29] E. Stoudenmire and D. J. Schwab, "Supervised Learning with Tensor Networks," *Advances in Neural Information Processing Systems*, vol. 29, 2016.

[30] A. Novikov, M. Trofimov, and I. Oseledets, "Exponential machines," *Bulletin of the Polish Academy of Sciences: Technical Sciences*, vol. 66, no. 6, pp. 789–797, 2018.

[31] Z. Chen, K. Batselier, J. A. Suykens, and N. Wong, "Parallelized Tensor Train Learning of Polynomial Classifiers," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 29, pp. 4621–4632, 10 2018.

[32] R. Karagoz and K. Batselier, "Nonlinear system identification with regularized Tensor Network B-splines," *Automatica*, vol. 122, p. 109300, 12 2020.

[33] A. Björck, "Numerical Methods for Least Squares Problems," *Numerical Methods for Least Squares Problems*, 1 1996.

[34] C. T. Kelley, "Iterative Methods for Optimization," *Iterative Methods for Optimization*, 1 1999.

[35] B. C. Mitchell and D. S. Burdick, "Slowly converging parafac sequences: Swamps and two-factor degeneracies," *Journal of Chemometrics*, vol. 8, pp. 155–168, 3 1994.

[36] P. Comon, X. Luciani, and A. L. de Almeida, "Tensor decompositions, alternating least squares and other tales," *Journal of Chemometrics*, vol. 23, no. 7-8, pp. 393–405, 2009.

[37] L. Sorber, M. Van Barel, and L. De Lathauwer, "Optimization-Based Algorithms for Tensor Decompositions: Canonical Polyadic Decomposition, Decomposition in Rank-$(L_r,L_r,1)$ Terms, and a New Generalization," *SIAM Journal on Optimization*, vol. 23, pp. 695–720, 4 2013.

[38] A. Saha and A. Tewari, "On the Finite Time Convergence of Cyclic Coordinate Descent Methods," 2010.

[39] S. J. Wright, "Coordinate descent algorithms," *Mathematical Programming*, vol. 151, pp. 3–34, 6 2015.

[40] S. Hendrikx, M. Bousse, N. Vervliet, and L. De Lathauwer, "Algebraic and Optimization Based Algorithms for Multivariate Regression Using Symmetric Tensor Decomposition," *2019 IEEE 8th International Workshop on Computational Advances in Multi-Sensor Adaptive Processing, CAMSAP 2019 - Proceedings*, pp. 475–479, 12 2019.

[41] Z. Li, *Symmetric Canonical Polyadic Decomposition And Gauss-Newton Optimizer For Nonlinear Volterra System Identification.* PhD thesis, Delft University of Technology, Delft, 7 2022.

[42] M. Boussé, N. Vervliet, I. Domanov, O. Debals, and L. De Lathauwer, "Linear systems with a canonical polyadic decomposition constrained solution: Algorithms and applications," *Numerical Linear Algebra with Applications*, vol. 25, p. e2190, 12 2018.

[43] K. Madsen, H. B. Nielsen, and O. Tingleff, "Methods for non-linear least squares problems," tech. rep., Informatics and Mathematical Modelling Technical University of Denmark, Denmark, 2004.

[44] S. Ruder, "An overview of gradient descent optimization algorithms," 9 2016.

[45] A. B. Güneş, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, "Automatic Differentiation in Machine Learning: a Survey," *Journal of Machine Learning Research*, vol. 18, pp. 1–43, 2018.

[46] H. J. Liao, J. G. Liu, L. Wang, and T. Xiang, "Differentiable Programming Tensor Networks," *Physical Review X*, vol. 9, p. 031041, 9 2019.

[47] U. Naumann, "The Art of Differentiating Computer Programs," *The Art of Differentiating Computer Programs*, 1 2011.

[48] T. Maehara, K. Hayashi, and K. I. Kawarabayashi, "Expected Tensor Decomposition with Stochastic Gradient Descent," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 30, pp. 1919–1925, 2 2016.

[49] Y. Cao, S. Das, L. Oeding, and H.-W. van Wyk, "Analysis of the Stochastic Alternating Least Squares Method for the Decomposition of Random Tensors," 4 2020.

[50] M. Rajih, P. Comon, and R. A. Harshman, "Enhanced line search: A novel method to accelerate parafac," *SIAM Journal on Matrix Analysis and Applications*, vol. 30, no. 3, pp. 1128–1147, 2008.

[51] J. Luo and B. Dai, "Joint Line Search and ALS Channel Estimation for Intelligent Reflective Surface Assisted MIMO System," *2021 7th International Conference on Computer and Communications, ICCC 2021*, pp. 1885–1889, 2021.

[52] D. Nion and L. De Lathauwer, "An enhanced line search scheme for complex-valued tensor decompositions. Application in DS-CDMA," *Signal Processing*, vol. 88, pp. 749–755, 3 2008.

[53] P. Tichavský and Z. Koldovský, "Simultaneous search for all modes in multilinear models," *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, pp. 4114–4117, 2010.

[54] C. Canuto and A. Tabacco, "Series of functions and power series," in *Mathematical Analysis II*, vol. 85, pp. 33–74, Springer Cham, 2015.

[55] P. Baldi, P. Sadowski, and D. Whiteson, "Searching for exotic particles in high-energy physics with deep learning," *Nature Communications 2014 5:1*, vol. 5, pp. 1–9, 7 2014.

[56] M. H. Hayes, *Statistical digital signal processing and modeling.* John Wiley & Sons, 2009.

[57] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A System for Large-Scale Machine Learning," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, (USA), pp. 265–283, USENIX Association, 2016.

[58] D. Maclaurin, D. Duvenaud, and R. P. Adams, "Autograd: Effortless Gradients in Numpy," in *ICML AutoML Workshop*, 2020.

[59] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d Alché-Buc, E. Fox, and R. Garnett, eds.), vol. 32, Curran Associates, Inc., 2019.

[60] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in PyTorch," 2017.

[61] B. van Merrienboer, O. Breuleux, A. Bergeron, and P. Lamblin, "Automatic differentiation in ML: Where we are and where we should be going," in *Advances in Neural Information Processing Systems* (S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, eds.), vol. 31, Curran Associates, Inc., 2018.

[62] Google, "XLA: Optimizing Compiler for Machine Learning | TensorFlow."

[63] F. Liu, X. Huang, Y. Chen, and J. A. Suykens, "Random Features for Kernel Approximation: A Survey on Algorithms, Theory, and Beyond," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 4 2020.

Glossary

List of Acronyms

DCSC	Delft Center for Systems and Control
SVD	singular value decomposition
SVM	Support Vector Machine
QP	quadratic program
RBF	radial basis function
LLS	linear least squares
NLS	nonlinear least squares
ALS	Alternating Least Squares
TT	Tensor Train
CPD	Canonical Polyadic Decomposition
KRR	Kernel Ridge Regression
AD	automatic differentiation
MSE	mean squared error
SteGD	Steepest Gradient Descent

List of Symbols

$*$, $\circledast_{d=1}^{D}$	Hadamard product and sequence of Hadamard products
$\langle \cdot, \cdot \rangle_F$	Frobenius inner product
$\mathbf{A}$	Matrix
$\mathbf{I}_M$	Identity matrix of size $M \times M$
$\|\cdot\|_F$	Frobenius norm
$\odot$	Khatri-Rao product

$\mathcal{O}(\cdot)$	Big O notation
$\otimes,\ \bigotimes_{d=1}^{D}$	Kronecker product and sequence of Kronecker products
$\circ,\ \tilde{\circ}$	Vector outer product and batch vector outer product
$\mathcal{A}$	Tensor
$\mathcal{A}(i_1, i_2, \ldots, i_D) = a_{i_1, i_2, \ldots, i_D}$	Element of tensor
$\times_d$	Mode-d vector product
$\mathbf{1}_M$	Vector of ones of length M
$\mathbf{a}$	Vector
$\mathrm{vec}(\mathcal{A})$	Vectorization of tensor $\mathcal{A}$
$a,\ A$	Scalar
I_d	Size of dimension d of a tensor
i_d	Index of dimension d of a tensor