Approximative Sparse Factorization of Neural Network Weight Matrices

Michael Brandner

Master's thesis

Approximative Sparse Factorization of Neural Network Weight Matrices

Michael Brandner

January 25, 2022

Chair of Data Processing
Technische Universität München

Michael Brandner. *Approximative Sparse Factorization of Neural Network Weight Matrices.* Master's thesis, Technische Universität München, Munich, Germany, 2022.

Supervised by Prof. Dr.-Ing. Klaus Diepold and Matthias Kissel; submitted on January 25, 2022 to the Department of Electrical and Computer Engineering of the Technische Universität München.

Chair of Data Processing, Technische Universität München, 80290 München, Germany, http://www.ldv.ei.tum.de/.

Abstract

In modern image processing applications, Convolutional Neural Networks (CNNs) are indispensable. Especially in the domains of object classification and face recognition, CNNs achieve impressive results. However, the increasingly accurate predictions are accompanied by ever-larger networks and consequently more computations. The number of operations required to compute the matrix vector product of a dense matrix $M \in \mathbb{N}^{N \times N}$ in the fully connected layer scales with $O(N^2)$, mainly responsible that networks like VGG19 require 19.6 billion Floating-point Operations (FLOPs) to evaluate a single image. This thesis investigates the factorization of weight matrices, of the fully connected layer, into a product of sparse matrices, which potentially reduces the order of operations needed to the subquadratic domain. Consequently, the number of operations required for inference and thus, resource consumption is reduced. I examine three approximation algorithms, namely Butterfly factorization, sparse EigenGame, and Flexible Approximate MUlti-layer Sparse Transform ($FA\mu ST$). The approaches are compared regarding the sparseness of their approximation and the approximation error. Furthermore, weight matrices of pre-trained Convolutional Neural Networks are factorized and compared regarding their prediction accuracy after approximation.

The best performance in terms of approximation error of the matrix and subsequent prediction accuracy was achieved by $FA\mu ST$. $FA\mu ST$ was able to make sufficiently accurate predictions with only 20% of the parameters. Where sufficient accurate means that the prediction accuracy drops by only 1%. For similar results, the other algorithms needed 3% (Butterfly) and 18% (sparse EigenGame) more computations than the original matrix-vector product.

The experiments show that Approximative Sparse Factorization (ASF) of the weight matrices can significantly decrease resources consumption without deteriorating the accuracy of the predictions too much. This can enable complex computer vision algorithms to be used on devices with low computational resources or time-critical systems.

Contents

1 Introduction

Neural Networks (NNs) are a powerful tool and a central component of modern Artificial Intelligence (AI) algorithms. CNNs in particular are an indispensable part of modern Computer Vision (CV), the task of extracting information from images or videos. Since their introduction, CNNs were continuously improved, especially in terms of prediction accuracy and the number of predictable objects.

1.1 History of Convolutional Neural Networks

The development of CNNs was mainly inspired by research on the visual cortex [19], which revealed that cells in the visual cortex have a small receptive field [19]. Furthermore, they can be divided into simple and complex cells. The simple cells process the input, whereas the complex cells process the output of the simple cells [19]. The receptive field of the complex cells is, therefore, an integration of the receptive fields of many simple cells.
With the support of these results, in 1980, the Neocognitron was developed [9]. This predecessor of modern CNNs consisted of stacked s-cells (simple cells) for feature extraction and c-cells (complex cells) to compensate for position changes [9]. The introduction of these concepts revolutionized modern image processing, as it was possible to detect shifted objects. With its seven layers and 11320 parameters, it was able to achieve good classification results on the NIST data (predecessor of MNIST). However, it was pointed out that the Neocognitron would achieve better results if it had more parameters [9]. This insight determined the course for further development.

In the years between 1989 [24] and 1998 [25] the term CNN was introduced by LeCun. The proposed LeNet models are the cornerstone of all modern CNNs. They consist of three basic components :

- convolutional layer
- pooling layer
- fully connected layer.

The great success of this architecture can be attributed mainly to the convolutional layer, which, like the visual cortex, processes information only in a small receptive field. By stacking these, information is assembled layer by layer. Since this adds the

a priori information that objects consist of locally linked components, it leads to better generalization [24].
With seven layers and $60,000$ parameters, LeNet-5 achieved an error rate of 0.95% on the MNIST test data.

However, CNNs breakthroughs were archived between 2010 and 2017, made possible primarily by the annual ImageNet Large Scale Visual Recognition Challenge (ILSVRC). In the first two years, traditional methods won the competition with an error rate (top-5 error) of 28.2% (2010) and 26.3% (2011) [38]. In 2012, a CNN won the competition for the first time. The network used, AlexNet, reduced the error by almost 10% to 15.3% [21]. More importantly, however, was the conclusion that large and deep NNs are required to achieve record-breaking results [21]. Although AlexNet introduced new concepts such as dropout, its success is mainly due to its size. With eight layers, $650,000$ neurons, and about 60 million parameters, it is roughly ten times larger than LeNet.
In the following years, new records were set every year. Until finally in 2017 SENet reached an error rate of 2.3% [18], which is barely half that of a human (error rate 5.1%) [38]. It should be noted that during the competition period, the error rate decimated from 28.3% to 2.3%. The number of parameters remained generally high, however, the most remarkable development is that the networks became deeper from 8 layers (AlexNet) to 152 (SENet).

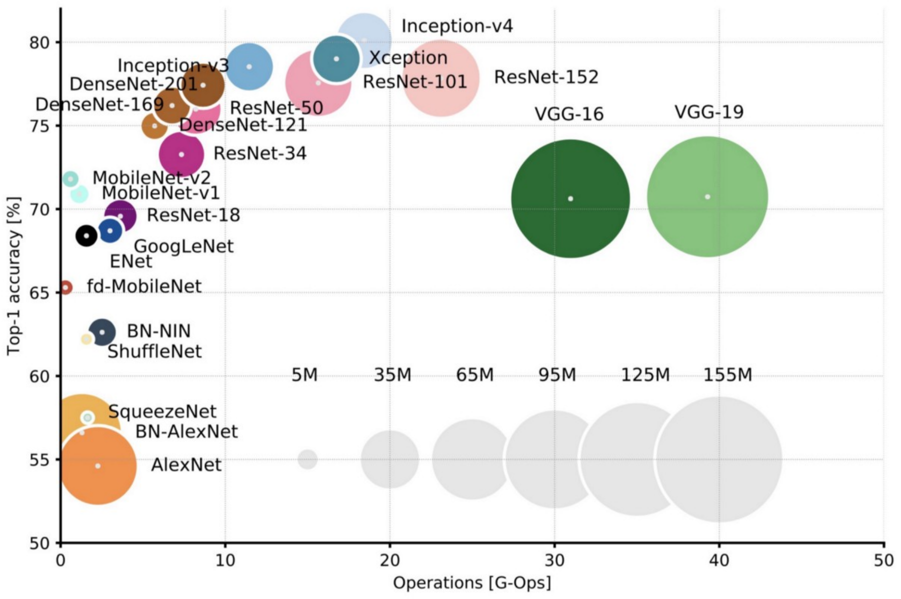

Figure 1.1: Plot of the accuracy, the number of computing operations required, and the number of parameters of common CNNs [5]. The plot illustrates the hyperbolic relationship between prediction accuracy and resource consumption.

In summary, it can be stated that deep networks in particular have become established. However, the general size of the networks has also increased. Figure 1.1 shows the accuracy, the number of computing operations required, and the number of parameters of well-known CNNs. It is obvious that the development towards higher accuracy is mainly paid for by the number of FLOPs/inference time required. The relationship between accuracy and inference time is hyperbolic, meaning a slight increase in accuracy costs a lot of FLOPs/inference time [5].

1.2 Problem Statement & Motivation

CNNs are getting deeper and bigger. This development leads to higher accuracy on the one hand, but also to increasing resource consumption on the other hand as described in section 1.1. These resources are mainly computing power for training/inference, memory, and energy consumption, which are mostly restricted.

The hyperbolic relationship between accuracy and resource consumption [5] is particularly problematic for applications such as autonomous driving [43]. Since high accuracy is mandatory, but the evaluation of the data shall also be swift. For example, a human crossing the road has to be detected immediately. Other problematic areas are Internet of Things (IoT) or Industry 4.0. These applications, in particular, are driven by mobile and embedded systems. While prediction accuracy is typically not as critical as in self-driving cars, resources are even more limited [31].

To further advance these applications, new approaches are needed to improve the prediction accuracy and resource consumption of CNNs. Improvements only by increasing the size and depth of the networks are already reaching their limits.

The goal of this thesis is to reduce the resource consumption of CNNs. In particular, the fully connected layers are investigated since most of the parameters are part of these layers. For instance, AlexNet consists of 60 million parameters, of which 58 million are within the fully connected layer [1]. The reduction in resource consumption should be achieved by approximating weight matrices as a product of sparse factors. For example, assuming a weight matrix $W \in \mathbb{R}^{m \times n}$ the advantage of this approach is that the calculation of the matrix-vector product requires only $O(k)$ instead of $O(nm)$ operations, where k is the number of non-zero elements.
It should be mentioned explicitly that a reduction of the resource consumption at similar accuracy is very close to an increase of the accuracy, since it allows e.g., embedded systems to use more accurate systems and therefore usually also larger CNNs. The higher resource consumption during training due to the extra factorization step is not critical since such models are trained only once on separate systems with higher performance.

1.3 Outline

The second chapter of this thesis explains basic requirements and concepts. The following chapter reviews the current literature on structured matrices that can significantly reduce resource consumption. After giving an overview in chapters two and three, I introduce the methodology of this thesis in chapter four In the subsequent chapter, the experiment results will be presented. These results are discussed and placed in the current scientific context in chapter 6. In the last chapter, I summarize the results of the thesis.

2 Preliminaries

2.1 Convolutional Neural Networks

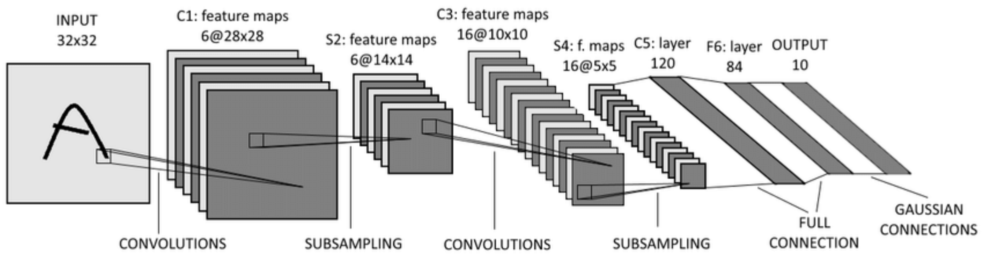

Figure 2.1: Architecture of the LeNet-5 network. Consisting of the three components convolutional, pooling and fully connected layer.

CNNs have revolutionized modern image processing. In 2012, a milestone was achieved by AlexNet. For the first time, a NN was able to classify objects with higher accuracy than the standard algorithms of that time. As already shown, the accuracy was improved in the following years by new concepts like dropout [21], inception modules [40] or shortcuts [15]. However, the basic structure has remained the same since LeNet. This basic structure is shown in figure 2.1 and can still be considered as the nucleus of CNNs. They consist of three essential components : convolutional layer, pooling layer, and fully connected layer. The foundation for this structure was laid in 1959 in experiments by David H. Hubel, and Torsten Wiesel [19]. They observed in experiments with monkeys that neurons in the visual cortex have a small local receptive field. Meaning that they only respond to a specific region of the visual field. Also, they demonstrated that some neurons respond to horizontal lines whereas others to lines with a different orientation. Furthermore, they showed that some neurons have larger receptive fields and respond to more complex patterns. These observations led to the conclusion that higher-level neurons are based on the outputs of low-level neurons. This behavior was replicated by stacked CNNs. While low-level neurons in deeper layers filter information such as edges and corners, the higher-level neurons assemble this information into objects such as houses or similar. This concept is shown schematically in figure 2.2.

Convolutional Layers

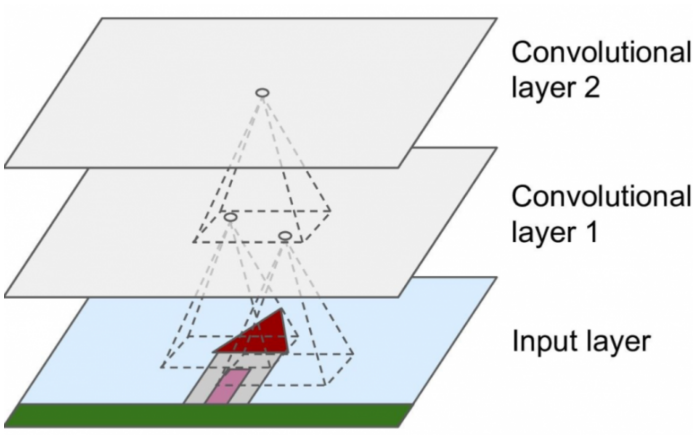

Figure 2.2: Schematic representation of feature extraction in an image.

The essential component of a CNNs is the convolutional layer. This type of layer is likely primarily responsible for the good performance of CNNs. A significant difference to a regular NN is that the input neurons of the convolutional layer are only connected to the pixels in their receptive fields.
This structure allows the network to recognize coarse structures in the first layers, which are later assembled into high-level features. This process is shown schematically in figure 2.2 [11]. During the training process, CNNs learn filters or convolutional kernels that are needed to recognize objects in the image. A feature map arises when all neurons in a layer use the same filters. They highlight the areas in the image that look most similar to the learned filter. To be able to recognize more features and objects in an image, CNNs consist of stacked feature maps in each layer.

Pooling Layers

Typically, each convolutional layer is followed by a pooling layer. The goal of this layer is to reduce the parameters. Thus, on the one hand, the required computing power is diminished. On the other hand, it improves generalization and therefore to avoiding overfitting [11]. The basic principle is very similar to the convolutional layer. Thus pooling layers are linked only with outputs in their receptive field from the previous layer. However, aggregation functions are used instead of filters to process the data further. These are typically a maximum or a mean function. The maximum aggregation leads to the exclusion of all values except the maximum value in the given receptive field.

Fully Connected Layers

The fully connected layer consists of regular artificial neurons that are connected to all neurons of the previous layer. Thus, a single neuron in layer n is connected to all outputs of the neurons in layer $n - 1$. The outputs are each weighted with a factor assigned to the specific neuron. These values plus bias are then summed and an activation function is applied to them. Typically the activation function is a ReLu, LReLU or tanh function. Due to the full interconnectedness and weighting, it should be noted that the fully connected layers consist of many parameters, which massively affect the resource consumption [1].

2.2 ImageNet

Benchmark Data

There are a few reasons why benchmark datasets are needed, especially in Machine Learning (ML). Foremost, it is difficult for a single researcher to collect and label 14 million images on his own. This would probably consume most of the resources, money and time. Another problem is the evaluation of results. It is not possible to compare models with each other and to estimate which model performs better without having standardized benchmarks. Without this ranking, it is difficult to assess which new techniques or methods are worth examining in more detail. A prime example of the progress that can be achieved through benchmark data coupled with a competition is ImageNet. Table 2.1 shows the development of the error rate at the ILSVRC in the period from 2010 to 2017. In this short period, the error rate was reduced to one tenth. Furthermore, competition showed the limits of traditional methods, which paved the way for CNNs.

Year	Winner	Error rate (%)
2010	NEC-UIUC (TM)	28.2
2011	XRCE (TM)	25.8
2012	AlexNet	16.4
2013	ZFNet	11.7
2014	VGG	7.3
2014	GoogleNet	6.7
2015	ResNet	3.6
2016	ResNeXt	3.0
2017	SENet	2.3

Table 2.1: Table of winning networks of ILSVRC with the corresponding error rate in the period from 2010 to 2017. Traditional methods are marked with (TM).

The data

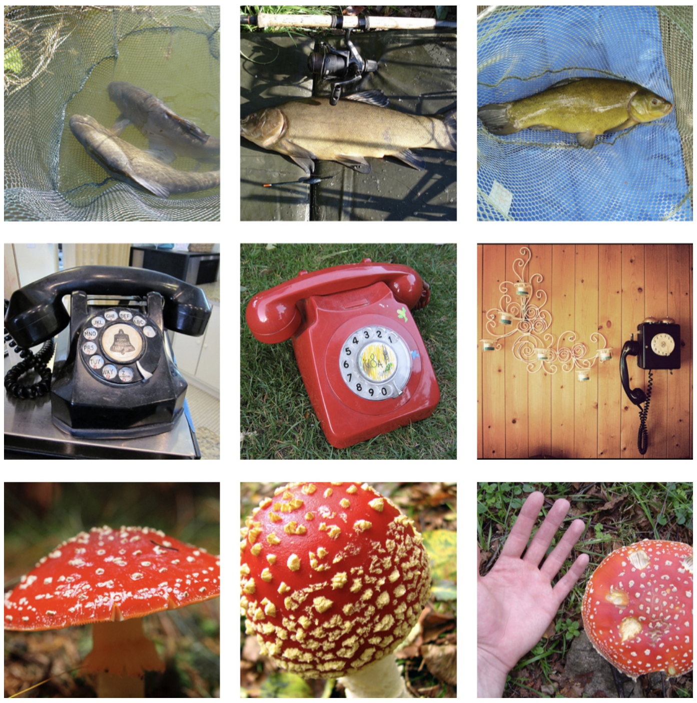

Figure 2.3: An excerpt of sample images from the imagenet dataset. The first row shows a tench, the second row a dial telephone, the third a fly agaric [38].

The ImageNet project is an image database that is mainly used for research purposes. ImageNet consists of roughly 15 million images divided into about 20,000 classes [21]. The images were taken from the internet and classified using Amazon Mechanical Turk. An excerpt is shown in figure 2.3. ImageNet became known mainly for its annual image classification competition, ImageNet Large-Scale Visual Recognition Challenge, held between 2010 and 2017. For the competition, a subset of 1000 classes with at least 1000 images was selected each year [21]. Resulting in 1.2 million train, 50,000 validation and 100,000 test images. The contest had a significant impact on CNN's rapid development during this period [5]. It is one of the reasons that classification accuracy increased dramatically during this period. This allowed ImageNet to become the defacto standard benchmark for CNNs [2]. Many platforms such as Pytorch or Tensorflow offer CNNs pre-trained on ImageNet. Since the experiment is based on such pre-trained models, it is mandatory to validate the results with ImageNet data.

2.3 Singular Value Decomposition

There are many examples of large matrices with non-random entries, such as images, neural network weight matrices, and measured data. These data are often described by several dependent variables which are generally correlated and include noise [22] When decomposing the eigenvalues of such matrices, it can be observed that they have few large eigenvalues (signal) and many very small ones (noise). If this is true, the matrices can be compressed by setting small eigenvalues and the corresponding eigenvectors to zero [39]. In general, however, images or weight matrices are not squared. Consequently, a different decomposition is necessary. The solution is a generalization of the eigenvalue decomposition, the Singular Value Decomposition (SVD).

The SVD, decomposes a matrix $A \in \mathbb{R}^{mxn}$ into a product

$$A = U\Sigma V^T. \tag{2.1}$$

where U is a unitary matrix $\in \mathbb{R}^{m\times m}$, Σ a rectangular diagonal matrix $\in \mathbb{R}^{m\times n}$ with non-negative values on the diagonal and U is a unitary matrix $\in \mathbb{R}^{n\times n}$.
The basic idea to compute the SVD is the fact that for each matrix A the product A^tA or AA^t is symmetric and positive semidefinite [3]. It is possible to calculate the eigenvalue decomposition for the correlation matrix $C = A^TA$. The roots of the determined eigenvalues of C are called singular values and have similar properties to the eigenvalues. They are arranged by size on the diagonal of Σ. The eigenvectors of A^TA and AA^T are two orthogonal bases, forming the columns of U and V [39]. This is shown in equation 2.2 for the right eigenvectors, analogously this also applies for the left eigenvectors.

$$\begin{aligned} C &= X^TX \\ &= (U \cdot \Sigma \cdot V^T)^T \cdot (U \cdot \Sigma \cdot V^T) \\ &= (V \cdot \Sigma \cdot U^T)^T \cdot (U \cdot \Sigma \cdot V^T) \\ &= V \cdot \Sigma \cdot \Sigma \cdot V^T \end{aligned} \tag{2.2}$$

The SVD allows compression for arbitrary data matrices. The so called truncated SVD is one of the central methods for many engineering and scientific tasks. It allows the best possible low-rank approximation of a matrix, measured by the Euclidean or l_2 distance [14]. A typical example is the compression of the well-known Lenna image as shown in figure 2.4.

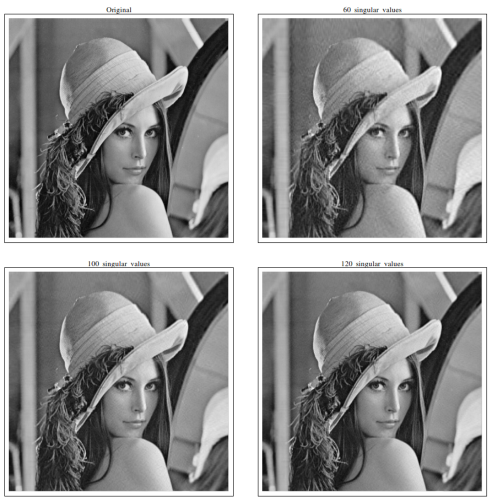

Figure 2.4: Original as well as compressions of Lenna image, with ranks 60, 100 and 120. The image demonstrates that an approximation with rank 100 is sufficient for most applications [1].

Another concept closely related to SVD is Principal Component Analysis (PCA). PCA is a linear transformation of variables, so that as few orthogonal variables as possible describe the relevant information [4]. PCA aims to simplify the data and is used for regression tasks as well as for detection of outlier in the data. It can be shown that these principal components are the eigenvectors of the covariance matrix [42]. Thus, the PCA of a given matrix $X \in \mathbb{R}^{m \times n}$ can also be viewed as the sum of rank-1 matrix approximations. Each summand consists of the individual left eigenvectors (called scores) and right eigenvectors (called loadings) times the corresponding singular value as shown in equation 2.3.

$$X = \sum_{i=1}^{m} = \sigma_i \cdot u_i \cdot v_i \tag{2.3}$$

Numerical Computation

The analytical approach for calculating SVD is usually not applicable. Over time, algorithms for numerical calculation have been developed. Since the columns of V are eigenvectors of $A^T A$, methods for calculating eigenvalues can be used. A well-known approach for calculating these is the power iteration [12]. However, due to its relatively poor convergence properties [28], this algorithm has been superseded by other approaches like random projections, but it is still the basis of several techniques.

[1] http://i.stack.imgur.com/FAC0i.png

The algorithm is used to calculate the dominant eigenvalue and the corresponding eigenvector of a given symmetric matrix $A \in \mathbb{R}^{n \times n}$, with eigenvectors $v_1, v_2, ..., v_n$ ordered according to the size of the associated eigenvalues. So that $|\lambda_1| \geqq |\lambda_1| ... \geqq |\lambda_n|$. Since the eigenvectors of A span $\mathbb{R}^n$ we can write for any vector b :

$$b = c_1 v_1 + c_2 v_2 + ... + c_n v_n \tag{2.4}$$

Following one can derive

$$\begin{aligned} Ab &= c_1 A v_1 + c_2 A v_2 + ... + c_n A v_n \\ &= c_1 \lambda_1 v_1 + c_2 \lambda_2 v_2 + ... + c_n \lambda_n v_n \\ &= \lambda_1 (c_1 v_1 + c_2 \frac{\lambda_2}{\lambda_1} v_2 + ... + c_n \frac{\lambda_n}{\lambda_1} v_n \\ A^2 b &= \lambda_1^2 (c_1 v_1 + c_2 \frac{\lambda_2}{\lambda_1}^2 v_2 + ... + c_n \frac{\lambda_n}{\lambda_1}^2 v_n \\ &\vdots \\ A^k b &= \lambda_1^n (c_1 v_1 + c_2 \frac{\lambda_2}{\lambda_1}^n v_2 + ... + c_n \frac{\lambda_n}{\lambda_1}^n v_n \end{aligned} \tag{2.5}$$

For $k \longrightarrow \infty$, the ratio $(\frac{\lambda_i}{\lambda_1})^k \longrightarrow 0$ for $i \in 2, 3, ..., n$ since λ_1 is the biggest eigenvalue by definition. Leading to the approximation :

$$A^k b \approx \lambda_1^k v_1 \tag{2.6}$$

Thus, one can numerically calculate the dominant eigenvalue and the corresponding eigenvector with the following procedure.

Algorithm 1 Calculating the dominant Eigenvalue v and corresponding Eigenvector λ using Power iteration

```
1: Given : Matrix $A$, non-zero random vector $v$
2: while not converged do
3:     $v \longleftarrow A \cdot v$
4:     $norm \longleftarrow compute\ norm(v)$
5:     $v \longleftarrow \frac{v}{norm(v)}$
6:     $\lambda \longleftarrow \frac{v^T A v}{v^T v}$
   return $\lambda, \quad v$
```

3 Literature Review

As mentioned in section 1.2, this thesis focuses on efficient matrix-vector computation for neural networks by approximately decomposing a matrix in a product of sparse factors. It should be mentioned that such a factorization is not only of significant interest for neural networks. There exist several other applications, like solving inverse problems, that can enormously benefit from such a factorization. The reason that motivates the extra factorization step is that the matrix-vector product of a dense weight-matrix $W \in \mathbb{R}^{m \times n}$ requires $O(mn)$ computations. In contrast, the ASF reduces these to $O(k)$, where k is the number of non-zero elements. In literature, there exist lots of approaches to minimize resource consumption. One method is for example to exploit structures in the matrices to reduce the required computations. Since such structured matrices are data-sparse, I want to put sparse matrices and their products in relation to these. In the following sections the structures are described on the basis of a matrix $M \in \mathbb{R}^{m \times n}$ with $m < n$.

3.1 Semiseparable Matrices

Since semiseparable matrices are a vast field of research containing many subclasses, we want to define them in the most general way. These are non-symmetric quasi-semiseparable matrices, defined as matrices wherein all subblocks of the upper or lower triangular part are of rank $r \leq 1$ [41]. There exist several representations of semiseparable matrices e.g., generator, diagonal-subdiagonal, givens-vector, or quasiseparable representation. However, since not every representation can express all subclasses of semiseparable matrices, one has to choose it according to whose application.
If all subblocks are of rank $r = 1$, it is possible to calculate the matrix-vector product using $O(n)$ operations.

3.2 Matrices of Low Displacement Rank

Matrices of low displacement rank contain the most prominent structured matrices, like Toeplitz, Hankel, Vandermond, or Cauchy matrices. Their most important properties are [36] :

- representation with a small number of parameters,

- matrix-vector multiplication with subquadratic resource consumption,
- close connection to computations with polynomials, particular their multiplication, division and interpolation,
- are associated with a linear displacement operator L, allowing to recover the original matrix easily from L and the image matrices or displacements $L(M)$.

In general, matrices of low displacement rank are defined as shown in equation 3.1. Where $L : \mathbb{R}^{m\times n} \rightarrow \mathbb{R}^{m\times n}$ is a linear operator of Sylvester type, for a fixed pair of operator matrices A and B.

$$L(M) = \nabla_{A,B}(M) = AM - MB \tag{3.1}$$

The most common operator matrices are unit f-circulant matrices Z_F or diagonal matrices $D(v)$ shown in equations 3.2 and 3.3. For each of the already mentioned structured matrices, we can associate a linear operator $L = \nabla_{A,B}(M)$ so that the rank of the displacement $L(M)$ remains small e.g. $O(1)$ or $O(min(m,n))$.

$$Z_F = \begin{pmatrix} 0 & \dots & 0 & f \\ 1 & \ddots & & 0 \\ & \ddots & \ddots & \vdots \\ & & 1 & 0 \end{pmatrix} \tag{3.2}$$

$$D(v) = \begin{pmatrix} v_1 & 0 & \dots & 0 \\ 0 & v_2 & & \vdots \\ \vdots & & \ddots & \vdots \\ 0 & \dots & \dots & v_i \end{pmatrix} \tag{3.3}$$

If the operator matrices are known, algorithms can be implemented that exploit the structures of the matrices and thus save computations. For this purpose, the matrix M is first compressed using the displacements. Afterward, the calculations are carried out on the compressed matrix. Finally, the result is decompressed. This leads to resource consumption of $O(n\ log(n))$ for Toeplitz and Hankel matrices [36].

3.3 Hierarchical Matrices

Hierarchical matrices or short $\mathcal{H}$-matrices consist of low-rank sub-matrices, as illustrated in figure 3.1. However, this does not necessarily imply that the $\mathcal{H}$-matrix itself is low-rank. By defining index sets I and J of the partitions, a block cluster tree $T(I \times J)$, the partitions P where by P^+ are the low-rank sub-blocks and P^- are full rank sub-blocks, and the rank distribution $r : P \rightarrow \mathbb{N}_+$ of an $\mathcal{H}$-matrix, then the set $H(r,P) \subset \mathbb{R}^{I\times J}$ of $\mathcal{H}$-matrices can be defined as all matrices $M \in \mathbb{R}^{I\times J}$ for which

$$rank(M|_b) \leq r(b) \qquad \text{for all b} \in \text{P} \tag{3.4}$$

holds [13]. In general, we aim to find block cluster trees such that major parts of the matrix can be approximated with low-rank matrices. If the partitions are known, it is possible to exploit the block structure, leading to subquadratic resource consumption for a matrix-vector product [13].

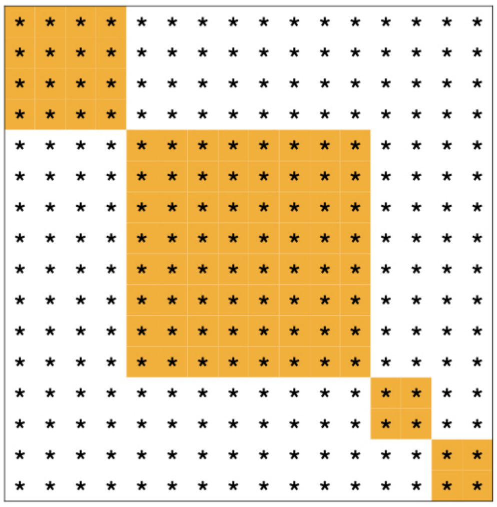

Figure 3.1: Hierarchical matrix. The color indicates the rank of the matrix, whereby the orange sub-blocks have full rank and the white blocks are low-rank.

3.4 Approximative Sparse Factorization

It is assumed that for all matrices where a fast algorithm for matrix-vector computation exists, there exists a representation as a product of sparse matrices [29]. This was already shown for some fast linear transformations like the Fast Fourier Transformation, the Hadarmad transformation or the Discrete Wavelet transformation. With this knowledge, it is reasonable to assume that sparse factorizations are closely related to structured matrices.

Sparse matrices are characterized by the fact that most of their entries are zero. An example is shown in equation 3.5.

$$M_S = \begin{pmatrix} * & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & * & 0 & 0 \\ * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & * \\ 0 & 0 & * & 0 & 0 & 0 \end{pmatrix} \tag{3.5}$$

The storage of the matrix $M_S \in \mathbb{R}^{o \times p}$ with $o = p = 6$ in dense format would store 80% of non-informative zero values. Since this consumes an unnecessarily high

amount of resources for large sparse matrices, other storage formats emerged. The most prominent are the COOrdinate (COO) and the Compressed Row Storage (CSR) format. The COO format stores the non-zero elements as triplets (row, column, value), whereas the CSR format stores the row values in a compressed form, reducing the number of row elements to be stored to $o + 1$.
Since for the matrix-vector product only calculations with non-zero elements have to be considered, the required computing power is reduced to $O(k)$ where k is the number of non-zero elements. Consequently, one can define the sparsity π of M as $\pi(M) = |M|_0 = k$. Furthermore by specifying the sparse approximative factorization of a given matrix M in κ sparse factors as $M = S_1 S_2 ... S_\kappa$ we can specify the overall sparsity of the approximation as $\sum_1^\kappa \pi(S_i)$ [34]. With the help of these definitions, the objective of sparse factorization can be stated as

$$\min_{S_1, S_2, ... S_\kappa} \sum_1^\kappa \pi(S_i) \qquad w.r.t. \qquad M \approx \prod_1^\kappa (S_i). \tag{3.6}$$

4 Sparse Approximative Factorization Methods

Fast linear transformations like the ones mentioned in section 3.4 are indispensable for many applications such as machine intelligence. They are common used for data preprocessing, feature generation or kernel approximation. The joint property of these transformations is the existence of algorithms that calculate the matrix-vector product with subquadratic resource consumption [6]. As already discussed, I assume that all structured matrices, for which such a fast algorithm exists, can be represented as a product of sparse matrices [7]. As I expect that weight matrices of neural networks are not random and thus have certain structures, I hope to find these and exploit them. In the following, I present algorithms, which approximately decompose a matrix $M \in \mathbb{R}^{m \times n}$ into a product of $\kappa \in \mathbb{N}_+$ sparse factors S_i as depicted in equation 4.1.

$$M \approx \sum_{i=1}^{\kappa} S_i \tag{4.1}$$

It should be emphasized that 4.1 is ideally the solution to the minimization problem presented in section 3.4. However, this problem is non-convex and non-smooth due to the sparsity constraints.

4.1 Butterfly factorization

The butterfly factorization can learn several fast transformations, such as the discrete Fourier transform, the discrete cosine transform, or the Hadamard transformation [27], with minimal prior knowledge and high approximation accuracy. It decomposes M into a product of $\kappa = log(n) + 3$ factors. The resulting factors S_i have a sparsity of $\pi(S_i) = O(n)$, leading to overall resource consumption of $O(n\ log(n))$.

Algorithm

This section presents the butterfly factorization for a matrix M. For simplicity and without loss of generality I assume $m = n = 4^a$ with $a \in \mathbb{N}_+$. The restriction on the shape of the matrix is not critical since we can add zero padding to other shaped matrices. I also define a set of points to access the rows X, and columns Ω, these points are described with the help of two binary trees T_X and T_Ω of height $L = log_2(n)$.

Now I can further denote at each tree level l, the ith node at level l in T_x as A_i^l for $i = 1, 2, ...2^{l-1}$ and the jth node at level $L - l$ in T_Ω as B_j^{L-l} for $j = 1, 2, ...2^{L-l} - 1$. These nodes partition the matrix in submatrices $K^l_{A_i^l, B_j^{L-l}}$ which will be referred to as $K^l_{i,j}$ [27]. The structure of the binary trees is illustrated in figure 4.1. We call M of complementary low-rank if all defined subbmatrices $K^l_{i,j}$ are numerically low-rank [27]. If M has this property, the butterfly factorization can recover the original matrix up to numerical precision. Since several fast transformations have this property, I hope to discover similar structures, at least approximately, in neural networks weight matrices. If the weight matrices do not have this property, I assume that it is approximated by the best complementary low-rank matrix measured by Frobenius norm.

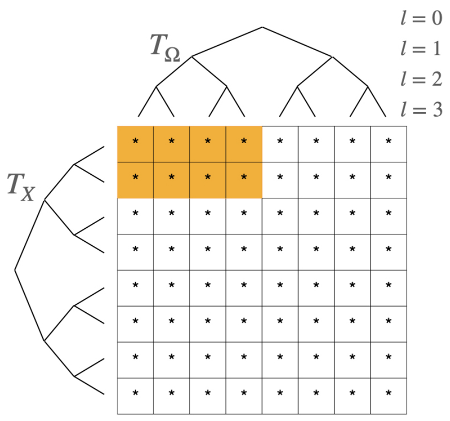

Figure 4.1: Matrix with associated binary trees T_x and T_Ω. Furthermore I highlighted the submatrix $K^2_{1,1}$.

The butterfly factorization is held out in two stages, the middle level factorization and the recursive factorization. With the aim to approximate M as a product of $L+3$ sparse factors as depicted in equation 4.2.

$$M \approx U^L G^{L-1} \cdots G^h M^h (H^h)^T \cdots (H^{L-1})^T (V^L)^T \tag{4.2}$$

The first stage, the middle level factorization decomposes each submatrix $K^h_{i,j}$, using a rank $r = 1$ low-rank approximation constructed with the truncated SVD, at level $l = h = \frac{L}{2}$ and arrange them in the initial factorization.

$$M \approx U^h M^h (V^h)^T \tag{4.3}$$

where U^h and $(V^h)^T$ are block diagonal matrices and M^h is a permutation matrix. The decomposition consequently looks as illustrated in figure 4.2.

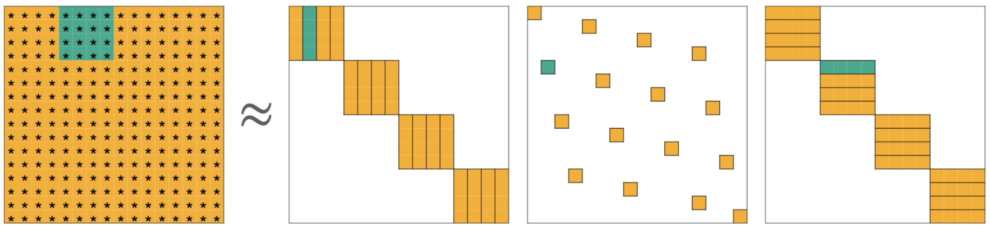

Figure 4.2: Middle level butterfly factorization. The alignment of the rank-1 SVD components is shown as an example for submatrix $K^2_{2,1}$

After the middle level decomposition the matrices U^h and V^h are recursively decomposed in $U^l \approx U^{l+1}G^l$ and $(V^l)^T \approx (H^l)^T(V^{l+1})^T$ for $l = h, h+1, ..., L-1$. These recursive factorization steps yield to equation 4.2 and are explained in more detail in the following sections.

Recursive factorization of U^h

Considering that all factorizations on level l result in a block diagonal matrix U^h for $l = h$ as shown in equation 4.4.

$$U^h = \begin{pmatrix} U_0^h & & & \\ & U_1^h & & \\ & & \ddots & \\ & & & U_{n-1}^h \end{pmatrix}, \tag{4.4}$$

with

$$U_i^h = \begin{pmatrix} U_{i,0}^h & U_{i,1}^h & \cdots & U_{i,n-1}^h \end{pmatrix}. \tag{4.5}$$

After recalling this we can split U_i^h in two subblocks by row, resulting in equation 4.6.

$$U_i^h = \left(\frac{U_i^{h,t}}{U_i^{h,b}}\right), \tag{4.6}$$

where t indicates the top and b the bottom half. If we combine the equations 4.6 and 4.5, we obtain

$$U_i^h = \left(\begin{array}{cccc} U_{i,0}^{h,t} & U_{i,1}^{h,t} & \cdots & U_{i,n-1}^{h,t} \\ \hline U_{i,0}^{h,b} & U_{i,1}^{h,b} & \cdots & U_{i,n-1}^{h,b} \end{array}\right). \tag{4.7}$$

If the assumption holds, that M is complementary low-rank, then the subblocks $\left(U_{i,2j}^{h,t} U_{i,2j+1}^{h,t}\right)$ and $\left(U_{i,2j}^{h,b} U_{i,2j+1}^{h,b}\right)$ are low-rank for $i = 0, 1, ..., m-1$ and $j = 0, 1..., \frac{m}{2} - 1$ [27]. Following the subblocks are approximated with a rank $r = 1$

approximation constructed with a truncated SVD. Next, the resulting singular vectors are embedded into the matrices U^{l+1} and G^{l}, as shown in figure 4.3. This step is performed recursively for the resulting U^{l+1} matrix until the tree level $l = L - 1$ is reached. Resulting in the factorization

$$U^{h} \approx U^{L} G^{L-1} \quad ... \quad G^{l} \tag{4.8}$$

for $l = h, h+1, ..., L-1$ with each factor having only $O(n)$ entries.

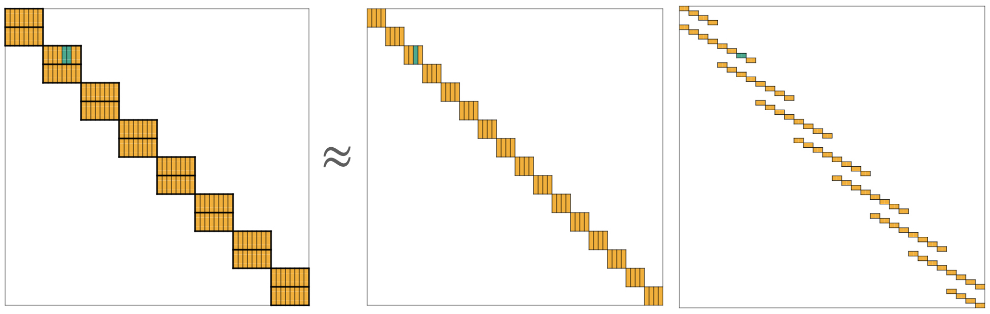

Figure 4.3: Recursive butterfly factorization of $U^2 = U^3 G^2$ for $M \in \mathbb{R}^{64\times 64}$. I marked the subblocks $U^{2,t}_{i,j}$ and $U^{2,t}_{i,j}$ and furthermore highlighted the scheme to insert the calculated singular vector of U^2 in U^3 G^2.

Recursive factorization of V^h

The decomposition of V^h is very similar to U^h. However, $(V^l)^T$ is decomposed into $(V^l)^T \approx (H^l)^T (V^{l+1})^T$. Based on the complementary low-rank property of the matrix M, we can again conclude that $K^{L-l-1}_{i,2j}$ and $K^{L-l-1}_{i,2j+1}$ are low-rank. Consequently, we can approximate these blocks with a rank $r = 1$ low-rank approximation constructed with the truncated SVD. Further, this factorization is done recursively, resulting in the sparse factorization

$$(V^h)^T \approx (H^h)^T ... (H^{L-1})^T (V^L)^T. \tag{4.9}$$

In summary, the Butterfly factorization decomposes a matrix M into a product of $L+3$ sparse factors. It follows that only $O(n\ log(n))$ arithmetic operations are necessary to calculate a matrix-vector product. If modern methods like randomized projections are used for factorization, it is possible to calculate the butterfly factorization with $O(n\ log(n))$ operations [27].

4.2 FAµST

The $FA\mu ST$ algorithm is a technique to approximative factorize a matrix into a product of sparse factors. The algorithm allows M to be approximative decomposed into any number of factors κ with any sparsity constraints regarding them. Typical sparsity constraints are for example the limitation of the number of non-zero elements per matrix or the number of non-zero elements per row or column. While this increases the algorithm's search space, it also makes it more potent than comparable approaches.

PALM

The possibility to pose a problem as a non-convex optimization problem is interesting in many cases and gives immense modeling power. However, non-convex problems are often NP-hard to solve [20]. The recently emerged algorithm Proximal Alternating Linearized Minimization (PALM) is able to handle non-convex problems as described in 4.10.

$$\min_{X_1,X_2,...,X_j} \Phi(X_1,X_2,...,X_j) = H(X_1,X_2,...,X_j) + \sum_{i=1}^{j} f_i(X_i) \tag{4.10}$$

whereby H is smooth and all f_i's are lower semi-continuos [23]. For simplicity, one can assume that all f_i's are indicator functions of the constrains τ_i. With these assumptions, the PALM algorithm can guarantee convergence up to a stationary point. This is achieved by alternately updating all blocks of variables with a projected gradient step. The algorithm is shown in 2 where P_{τ_i} denotes the projections operator on τ_n, a_n indicates the step size which depends on the Lipschitz constant of the gradient of H.

Algorithm 2 Summary of PALM algorithm.

```
1: Input : Number of iterations $I$, Number of components $j$
2: for $i = 1 : I$ do
3:     for $n = 1 : j$ do
4:         Set $X_n^{i+1} = P_{\tau_n}\left(X_n^i - \frac{1}{a_n} \nabla_{X_n} H(X_1^i, X_2^i, ..., X_n^i)\right)$
```

Algorithm

If the target as defined in 3.6 is reformulated as :

$$\min_{\lambda,S_1,S_2,...,S_\kappa} \Phi(\lambda,S_1,S_2,...,S_\kappa) = \frac{1}{2}\left\| M - \lambda \prod_{i=1}^{\kappa} S_i \right\|_F^2 + \sum_{i=1}^{\kappa} \delta_{\epsilon_i}(S_i) \tag{4.11}$$

with λ, that is used as as normalization factor avoiding the scaling ambiguities arising naturally when the constraint sets are (positively) homogeneous [23], so that each $\|S_i\|_F = 1$ and the indicator function, defining the subset of interest, $\delta_{\epsilon_i}(S_i)$ of the constraint set of ϵ_i. Typical constraint sets are explained in the following section. We can state that the PALM algorithm is applicable to the sparse factorization problem. In the following I will first introduce required concepts, these will then be assembled to the $FA\mu ST$ algorithm.

Projection Operator

The basic component of PALM is the projection on the constraints. These should be simple and easy to calculate. The most straightforward constraint ϵ is to limit the number of non-zero elements of a factor S_i to k. So that for the sparse and normalized factors $||S_i||_0 \leq k$ and $||S_i||_F = 1$ holds. The projection P_ϵ of this constraint then keeps only the k absolute largest values and sets all others to 0. After that, the matrix is normalized to satisfy the second constraint [23]. Typical desirable sparsity constraints ϵ are :

- Constraints on the total number of non-zero elements per factor,
- Number of non-zero elements per row or column,
- Impose non-negativity,
- Imposing a circulant structure,
- Triangular matrices constraints,
- Diagonal matrices constraints.

Gradient and Lipschitz modulus

We are now interested in the update rule for a specified factor S_i at time t further denoted as S_i^t. Also, we can summarize all left and right factors as :

$$L = \prod_{n=1}^{j-1} S_n^{t+1} \qquad (4.12) \qquad\qquad R = \prod_{n=j+1}^{\kappa} S_n^t \qquad (4.13)$$

Thus, we can consequently summarize H as :

$$\begin{aligned} H(S_1^{t+1}, S_2^{t+1}, ..., S_{i-1}^{t+1}, S_i^t, ..., S_\kappa^t, \lambda^i) &= H(L, S_i^t, R, \lambda) \\ &= \frac{1}{2} \left\| M - \lambda^t L S_i^t R \right\|_F^2 \end{aligned} \qquad (4.14)$$

Resulting in the subsequent gradient w.r.t S_i at iteration t [23]:

$$\nabla_{S_i} H(L, S_i^t, R, \lambda^t) = \lambda^t L^T \left(\lambda^t L S_i^t R - M\right) R^T \qquad (4.15)$$

Once all factors are updated we still need to update the normalization term λ for this we define $\hat{M} = \prod_{i=1}^{\kappa} S_i^{t+1}$ leading to the update rule w.r.t. λ as follows.

$$\nabla_{\lambda^t} H(S_1^{t+1}, S_2^{t+1}, ..., S_\kappa^{t+1}\lambda^t) = \lambda^i Tr(\hat{M}^T \hat{M}) - Tr(\hat{M}^T \hat{M}) \tag{4.16}$$

Using these concepts, we can specify the PALM for Multi-layer Sparse Approximation (PALM4MSA) algorithm that decomposes the matrix M.

Algorithm 3 PALM for Multi-layer Sparse Approximation (PALM4MSA) algorithm [23]

```
1:  Given :
2:  Operator $M$, desired number of factors $\kappa$
3:  constraint sets $\epsilon_i$ and initialization $\{S_i^0\}$ for $i \in \{1, 2, ...\kappa\}$
4:  number of iterations $N$
5:  for $t = 0 : N - 1$ do
6:      for $i = 1 : \kappa$ do
7:          $L \leftarrow \prod_{l=1}^{i-1} S_l^t$
8:          $R \leftarrow \prod_{l=i+1}^{\kappa} S_l^{t+1}$
9:          $c_i^t > (\lambda^t)^2 \|L\|_2^2 \|R\|_2^2$
10:         $S_i^{t+1} \leftarrow P_{\epsilon_i}\left(S_i^t - \frac{1}{c_i}\lambda^t L^T \left(\lambda^t L S_i^t R - M\right) R^T\right)$
11:     $\hat{M} \leftarrow \prod_{i=1}^{\kappa} S_i^{t+1}$
12:     $\lambda^{t+1} \leftarrow \frac{Tr(M^T \hat{M})}{Tr(\hat{M}^T \hat{M})}$
    return Estimated factorization $\lambda^N, S_i^N$ for $i \in \{1, 2, ..., \kappa\}$
```

4.3 EigenGame

The EigenGame algorithm recently introduced by DeepMind is a novel approach to calculate the PCA. It tries to solve the PCA from the point of view of a game. Each eigenvector to be computed is controlled by a player who tries to maximize his own utility function. From this perspective the eigenvectors form the unique strict Nash equilibrium of the proposed game [10].

Nash Equilibrium

Game theory, is a mathematical concept that allows to model conflicts and cooperation between intelligent rational decision-makers [32]. It not only considers the players own behavior but also the behavior of other players. Basically, two types of games are distinguished, the cooperative and the non-cooperative game. In non-cooperative games, the players act in pure self-interest [33]. The Nash equilibrium describes the solution of a non-cooperative game. It is a state in which each player makes the best choice for himself, taking into account the actions of other players [17].

Algorithm

PCA is often interpreted as learning a projection of a matrix X to a subset that captures maximum variance. In the following I assume X to be symmetric. I will also refer to the true eigenvectors as v_i, whereas the approximated eigenvectors will be referred to as $\hat{v}_i$. The matrix of all eigenvectors, ordered according to their eigenvalues, is further denoted by V resp. $\hat{V}$. The subset $v_{j<i}$ denotes the set of eigenvectors $\{v_j | j \in \{1, 2, ...i-1\}\}$.
The interpretation of maximizing variance is equivalent to maximizing the diagonal elements of a matrix $R(\hat{V}) = \hat{V}^T X^T X \hat{V} = \hat{V}^T M \hat{V}$,

$$\max_{\hat{V}^T\hat{V}=I} \sum_i R_{ii} = Tr(R) = Tr(\hat{V}^T M \hat{V}) = Tr(\hat{V}\hat{V}^T M) = Tr(M). \tag{4.17}$$

However, it is also possible to solve the inverse problem of minimizing all non-diagonal entries.

$$\min_{\hat{V}^T\hat{V}=I} \sum_{i \neq j} R_{ij} \tag{4.18}$$

By further examining the entries of R it can be stated, that the diagonal entries $R_{ii} = \langle \hat{v}_i, M\hat{v}_i \rangle$ are Rayleigh quotients which map each eigenvector v_i to its corresponding eigenvalue λ_i. The non-diagonal elements $R_{ij} = \left\langle \hat{v}_i, M\hat{v}_j \right\rangle$ with $i \neq j$ measure the alignment between the two approximated eigenvectors $\hat{v}_i$ and $\hat{v}_j$. By minimizing the non-diagonal elements or maximizing the diagonal elements we can determine all eigenvectors, however, if we are only interested in the top-k eigenvectors the problems arise that, V is not square anymore so $VV^T \neq I$. For the top-k eigenvectors $VV^T = P$ is a projection since we can still assume, that V is orthogonal. Furthermore, equation 4.18 places no preference on recovering large over small eigenvectors [10], it only forces the columns of $\hat{V}$ to be eigenvectors. This results in the conclusion to minimize non-diagonal and maximizing the diagonal elements. Leading to an objective function as shown in equation 4.19.

$$\max \sum_i R_{ii} - \sum_{i \neq j} R_{ij} \tag{4.19}$$

This objective function still ignores the hierarchy of eigenvectors. Thus $\hat{v}_1$ is penalized for aligning with any $\hat{v}_k$. That should not be the case because the approximation of the largest eigenvector should be free to search for the direction that captures the maximum variance independent of the other vectors [10]. This problem leads to the consideration to penalize the vectors only by the alignment to parent vectors. We define parent vectors as eigenvectors with a larger singular value i.e. $\hat{v}_{j<i}$. These considerations result in the utility function $u_i(\hat{v}_i|\hat{v}_{j<i})$ we aim to maximize.

$$\begin{aligned} u_i(\hat{v}_i|\hat{v}_{j<i}) &= \hat{v}_i^T M \hat{v}_i - \sum_{j<i} \frac{(\hat{v}_i^T M \hat{v}_j)^2}{\hat{v}_j^T M \hat{v}_j} \\ &= \|X\hat{v}_i\|^2 - \sum_{j<i} \frac{\langle X\hat{v}_i, X\hat{v}_j\rangle^2}{\langle X\hat{v}_j, X\hat{v}_j\rangle} \end{aligned} \tag{4.20}$$

To determine the maxima of the utility, a gradient ascent method is used. For this purpose, the following gradient of the utility function is required. The gradient consists of a generalized Gram-Schmidt step followed by a standard matrix product similar to power iteration and Oja's rule.

$$\begin{aligned} \nabla u_i(\hat{v}_i|\hat{v}_{j<i}) &= 2M[\hat{v}_i - \sum_{j<i} \frac{\hat{v}_i^T M \hat{v}_j}{\hat{v}_j^T M \hat{v}_j}] \\ &= 2X^T[X\hat{v}_i - \sum_{j<i} \frac{\langle X\hat{v}_i, X\hat{v}_j\rangle}{\langle X\hat{v}_j, X\hat{v}_j\rangle} X\hat{v}_j] \end{aligned} \tag{4.21}$$

To calculate the eigenvectors the following algorithm is proposed [10].

Algorithm 4 Algorithm to calculate a single eigenvector $\hat{v}_i$

```
1: Given :
2: Matrix $X_t \in \mathbb{R}^{m\times n}$, maximum error tolerance $\rho$,
3: random initial vector $\hat{v}_i^0$, step size $\alpha$
4: $\hat{v}_i \leftarrow \hat{v}_i^0$
5: $T = \left\lceil \frac{5}{4} min \left( \| \nabla_{\hat{v}_i^0} u_i \| /2, \rho_i \right)^{-2} \right\rceil$
6: for $t = 1 : T$ do
7:     $rewards \leftarrow X_t \hat{v}_i$
8:     $penalties \leftarrow \sum_{j<i} \frac{\langle X_t\hat{v}_i, X_t\hat{v}_j\rangle}{\langle X_t\hat{v}_j, X_t\hat{v}_j\rangle} X_t\hat{v}_j]$
9:     $\nabla_{\hat{v}_i} \leftarrow 2X_t^T[rewards - penalties]$
10:    $\nabla_{\hat{v}_i}^R \leftarrow \nabla_{\hat{v}_i} - \langle \nabla_{\hat{v}_i}, \hat{v}_i \rangle \hat{v}_i$
11:    $\hat{v}_i' \leftarrow \hat{v}_i + \alpha \nabla_{\hat{v}_i}^R$
12:    $\hat{v}_i \leftarrow \frac{\hat{v}_i'}{\|\hat{v}_i'\|}$
   return $\hat{v}_i$
```

Algorithm 4 not only guarantees convergence it also is possible to decentralize and parallelize the calculations. This is possible because the hierarchical restriction to calculate parent vectors first seems unnecessary since parent vectors become quasi-stationary as they approach their optimum [10].

Related Work

PCA is a well known tool, one approach to calculate it, is to compute the SVD with randomized projections. Nevertheless, there are other approaches like Hebb's rule [16] which is mainly used in neuroscience and allows to determine the top-k eigenvectors v of a matrix M, with a given learning rate η using the update function $v \leftarrow v+\eta Mv$. Similarly, Oja's rule updates the vectors according to $v \leftarrow v+\eta \left(I - vv^T\right)$ [35]. For $\eta \rightarrow \infty$ Oja's rule becomes the power iteration [10]. Whereas Oja's rule and power iteration are able to determine the top-k eigenvectors in sequence, however, these methods enforce the orthogonality by removing the learned subspace from the matrix. This deflation step prevents effective parallel execution, which is not the case for EigenGame. In the sequential calculation of eigenvectors, Heb's rule is very similar to EigenGame, but the update term is not a gradient of a function [10].

4.4 Sparse EigenGame

The EigenGame algorithm presented in the last section has the task to find the (top-k) principal components of a matrix. For this reason, I want to introduce modified algorithm, the Sparse EigenGame, which decomposes the matrix based on EigenGame into sparse factors. To achieve this, I combined EigenGame with the PALM algorithm presented in section 4.2. The new aspect of this method is, that instead of a regularization term, a projection of the gradient according to the sparsity constraints is used to solve the problem.

Sparse Coding

Sparse coding aims to find a representation of a matrix or vector as a weighted linear combination of basis vectors called atoms. These atoms form the dictionary $D \in \mathbb{R}^{n \times K}$ and do not have to be orthogonal. Typically, dictionaries are over-complete, meaning $K > m$ [26]. This allows modeling the problem with higher dimensionality. Sparse coding aims to determine the sparsest representation R by solving the minimization problem

$$\min_{D,R} \|M - DR\|_2^2 + \beta \|R\|_0 . \tag{4.22}$$

Several algorithms try to solve problem 4.22, e.g., Lasso-regression, Orthogonal Matching Pursuit (OMP), or Least Angle Regression (LARS). We decided to use LARS due to its numerical characteristics and the stable implementation in scikit-learn [37]. LARS is an improved version of forward stepwise regression, which for a given set of predictors x always selects the one with the highest absolute correlation x_{j_1} to the target y and performs linear regression. This procedure leaves a residuum orthogonal to x_{j_1}. After projecting the other predictors orthogonal to x_{j_1} we repeat the selecting

process [8]. However, This greedy approach is often too aggressive and may eliminate valuable information.
LARS tries to improve this aggressive greedy behavior. Similar to the stepwise forward regression, the regressor x_{j_1} with the highest correlation to the target y is selected first. After that we take the largest step possible in the direction of x_{j_1} until another regressor x_{j_2} has as much correlation to the residuum as x_{j_1}. At this stage, LARS proceeds in the direction equiangular between x_{j_1} and x_{j_2} till a third regressor has the same correlation to the residuum. This is continued until a certain termination condition is reached.

Algorithm

The algorithm consists of two steps. In the first step, I aim to calculate the "sparse eigenvectors". This is done by the sparse EigenGame. We have to keep in mind, that we want to maximize the utility of a given vector $\hat{v}_i$ according to :

$$\begin{aligned} u_i(\hat{v}_i|\hat{v}_{j<i}) &= \hat{v}_i^T M \hat{v}_i - \sum_{j<i} \frac{(\hat{v}_i^T M \hat{v}_j)^2}{\hat{v}_j^T M \hat{v}_j} \\ &= \|X\hat{v}_i\|^2 - \sum_{j<i} \frac{\langle X\hat{v}_i, X\hat{v}_j\rangle^2}{\langle X\hat{v}_j, X\hat{v}_j\rangle} \end{aligned} \tag{4.23}$$

similarly to EigenGame. However to include sparsity I used a projection on the sparsity constraints ϵ_i as described in section 4.2. Leading to a updated version of calculating the "eigenvectors" as presented in algorithm 5.

Algorithm 5 Algorithm to calculate a single sparse eigenvector $\hat{v}_i$

```
1: Given :
2: Matrix $X_t \in \mathbb{R}^{m\times n}$, maximum error tolerance $\rho$,
3: random initial vector $\hat{v}_i^0$, step size $\alpha$
4: $\hat{v}_i \leftarrow \hat{v}_i^0$
5: $T = \left\lceil \frac{5}{4} min\left(\| \nabla_{\hat{v}_i^0} u_i \| /2, \rho_i\right)^{-2} \right\rceil$
6: for $t = 1 : T$ do
7:     $rewards \leftarrow X_t \hat{v}_i$
8:     $penalties \leftarrow \sum_{j<i} \frac{\langle X_t\hat{v}_i, X_t\hat{v}_j\rangle}{\langle X_t\hat{v}_j, X_t\hat{v}_j\rangle} X_t \hat{v}_j]$
9:     $\nabla_{\hat{v}_i} \leftarrow 2X_t^T[rewards - penalties]$
10:    $\nabla_{\hat{v}_i}^R \leftarrow \nabla_{\hat{v}_i} - \langle \nabla_{\hat{v}_i}, \hat{v}_i\rangle \hat{v}_i$
11:    $\hat{v}_i' \leftarrow P_{\epsilon_i}\left(\hat{v}_i + \alpha \nabla_{\hat{v}_i}^R\right)$
12:    $\hat{v}_i \leftarrow \frac{\hat{v}_i'}{\|\hat{v}_i'\|}$
   return $\hat{v}_i$
```

In the second step I used the calculated vectors $\hat{V}$ as a dictionary to learn a sparse code C so that :

$$M \approx C\hat{V} \tag{4.24}$$

To determine C I used the already presented Least Angle Regression (LARS) algorithm.

5 Experiments

To compare the different algorithms with respect to their applicability for the approximation of weight matrices of neural networks, I approximate the weight matrix G of the last (fully connected) layer of a pre-trained googlenet network. The network is provided by Pytorch[1]. The approximation is held out with different Relative Complexitys (RCs), if possible, which is defined as

$$RC = \frac{\sum_{i=0}^{\kappa} \pi\left(S_i\right)}{\|G\|_0}. \tag{5.1}$$

Where κ is the number of factors S_i and $\pi(S)$ is the corresponding sparsity. In the first section I will briefly discuss some peculiarities of G, in the second part I will present the results. The evaluation of the approximations is based on the criteria :

- RC
- The approximation error $\left\|G - \prod_{i=1}^{\kappa} S_i\right\|_F$
- Prediction accuracy of the network with an approximated last layer on the imageNet validation data from 2012.

The used methods to approximate G are :

- Butterfly factorization
- $FA\mu ST$
- Sparse EigenGame

The results of these approaches are compared with a truncated SVD, and a dummy method keeping only the absolute top k values of the matrix.

5.1 Target Matrix G

G has the shape $(1000, 1024)$ and consequently $1, 024, 000$ parameters. The maximum entry in the matrix is 0.35, the smallest -0.24. It is notable, that not a single entry in the matrix is exactly 0. The weights are approximately normally distributed

[1] https://pytorch.org/

with a mean of -9.55^{-10} and a variance of 0.003. A histogram of the weights is shown in figure 5.1 this also reveals the slight positive skewness of 0.54.

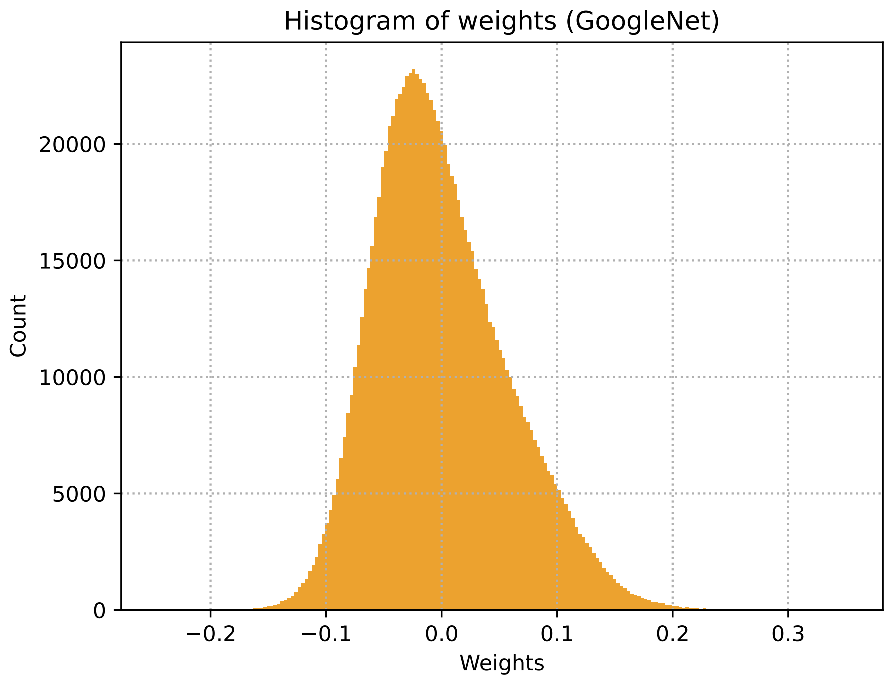

Figure 5.1: Histogram of the weights of G. A normal distribution with mean -9.55^{-10} and variance of 0.003 is evident. Furthermore, the slight positive skewness of 0.54 is recognizable.

I would like to emphasize that each class i can be assigned to the i-th neuron in the last layer. Consequently, we can assign the weights of the respective neuron g_i to class i. When examining g_i, the fluctuation of the L_2-norm of the vectors is conspicuous, ranging from 1.55 for tabby's to 2.24 for pencil sharpener. The norm of the individual vectors g_i is presented in figure 5.2. This deviation is astonishing, since one would have expected the norm to be relatively constant. I assume the deviations are since the number of images per class in the training set is not identical. The number ranges from 732 to 1300 images per class [38].

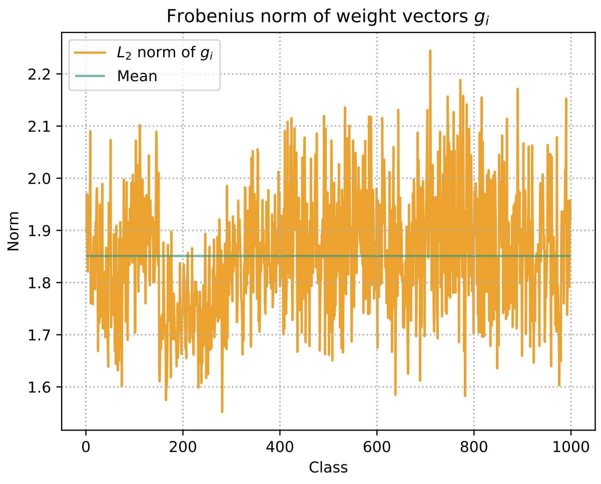

Figure 5.2: Plot of the L_2-norm of g_i. The highest value belongs to pencil sharpeners, the lowest to tabby's.

G was further examined for hierarchical structures. However, no such structures were found.

5.2 Reference algorithms

5.2.1 Dummy Method

The dummy method of setting the smallest values of G to zero performed poorly. If about 10% of the values are zero, the network does not perform better than a random predictor. I think this is due to two reasons. The first is that the ratios of the weights in the matrix get disturbed. Since the result of the last layer still passes through a softmax function, it can be assumed that the ratio of the weights to each other is more critical than the approximation accuracy.To demonstrate that the ratio of the weights is important, I multiplied the weight matrix with various factors up to 50. The resulting approximation error was with 2873.14 relatively high. However, the prediction accuracy dropped only slightly from 69.78 to $67, 37\%$.
The assumed second reason is, that the heuristic is too simple. Just because a value in the weight matrix is small does not necessarily imply that it carries less or no information.

5.2.2 Truncated SVD

The truncated SVD is not an algorithm for sparse factorization. However, we need to recall that the truncated SVD with rank r results in the best low-rank r approximation of a matrix, measured by the Frobenius norm. The obtained factors of this factorization have the known form

$$M \approx U \Sigma V^T, \tag{5.2}$$

where $M \in \mathbb{R}^{n \times m}, U \in \mathbb{R}^{n \times r}$, $\Sigma \in \mathbb{R}^{r \times r}$, and $V^T \in \mathbb{R}^{r \times m}$. Thus, we can avoid calculations by decreasing the rank of the approximation. Thereby the resource consumption is reduced from $O(nm)$ to $O(r(m+n))$. I use this relation to diminish the number of calculations to a fraction of the conventional number of calculations. I assume it is a suitable reference algorithm. The approximation error as well as the corresponding prediction accuracy are presented in plot 5.3.

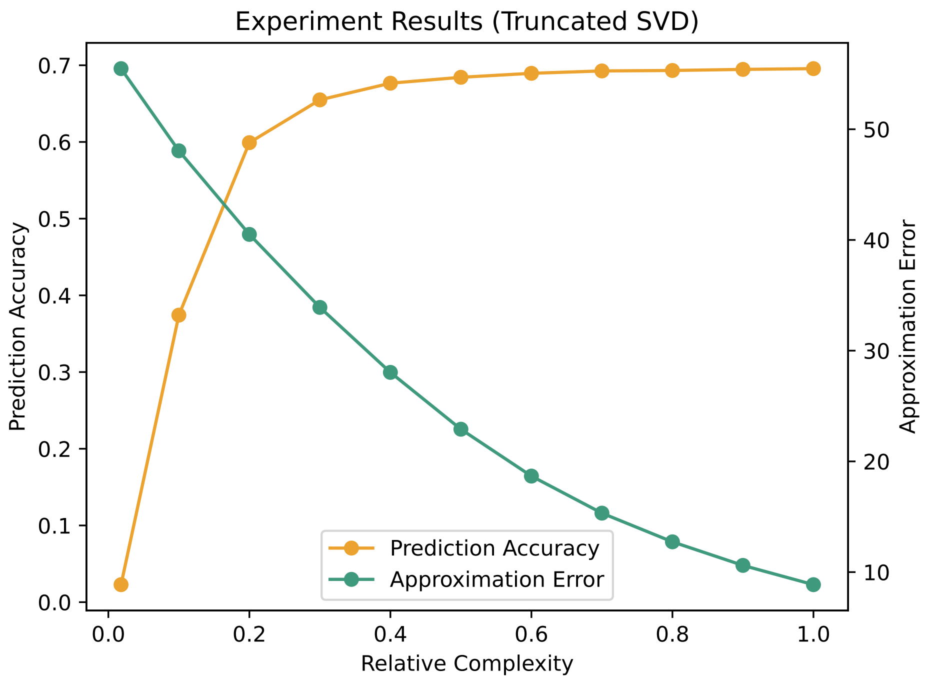

Figure 5.3: Prediction accuracy as well as the approximation error plotted over RC. A higher RC causes a lower approximation error, resulting in better predictions.

As expected, the approximation error decreases with higher RC. The lower approximation error is presumably responsible for the increase in the prediction accuracy of the network. The most interesting operating points for real-world problems are between a RC of 0.3 and 0.5 in this range, many computations can be saved. However, the prediction accuracy is not significantly worse than in the original network. To reach the point where the prediction accuracy drops by only one percent, the truncated SVD requires 70% of the parameters compared to the original matrix.

5.3 Sparse EigenGame

The learning of the factors of the sparse EigenGame consists of two steps. In the first step, the sparse dictionary D_S is learned. In the second, D_S is used to learn a sparse code C_S so that $G \approx C_S D_S$. The hyperparameters of the sparse EigenGame are :

- Number of components or atoms of the dictionary
- Sparsity of the atoms
- Regularization factor γ of the dictionary learning algorithm.

A grid-search algorithm was used to obtain good parameters for this problem. The search resulted in the choice of the hyperparameters $\gamma = 0.0001$ and the sparsity of the atoms of 0.4. The number of components was used as the setting value of the RC. I compared the results to a random dictionary with a equal setting to evaluate if the "sparse eigenvectors" are a suitable choice as a dictionary. The results are shown in plot 5.4.

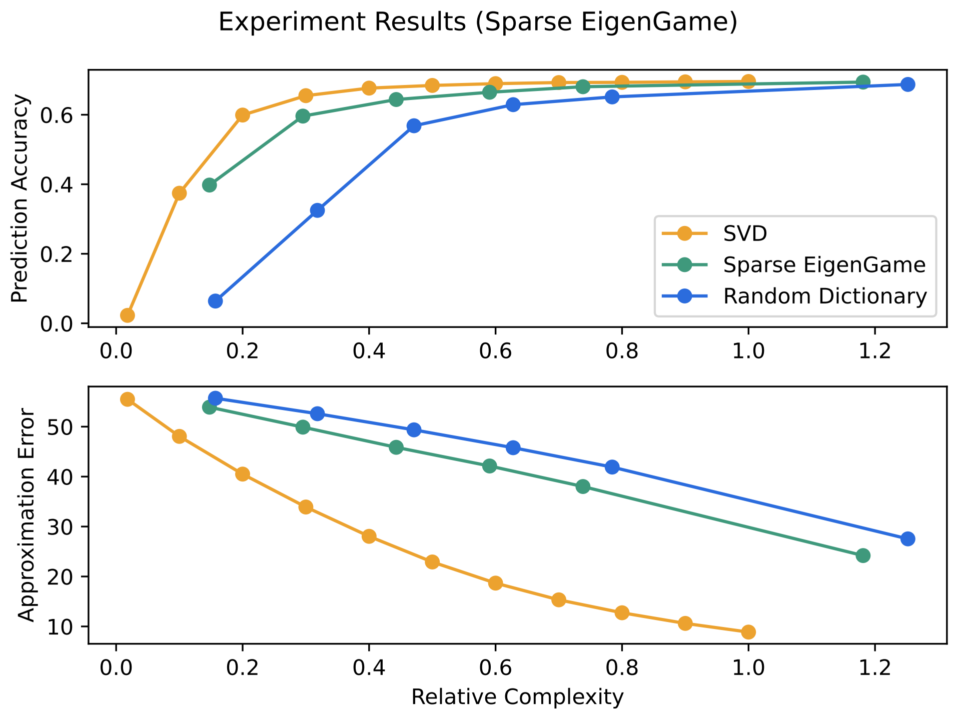

Figure 5.4: Approximation results of the sparse EigenGame. For comparison, the results of the truncated SVD as well as the approximation with a random dictionary are presented. The results of the sparse EigenGame were worse than those of the truncated SVD but better than ones of the random dictionary.

Plot 5.4 shows that the sparse EigenGame performs relatively well, however not as well as the truncated SVD. It is noticeable that the approximation error of the EigenGame decreases linearly. However, that of the SVD decreases quadratically. Furthermore, it turns out that the "sparse eigenvectors" are a good choice as a dictionary, however definitely not the best. Surprisingly, the approximation error is only slightly better compared to the random dictionary.

5.4 Butterfly Factorization

Only square matrices can be factorized using the investigated Butterfly Factorization algorithm. Since G is not square, the matrix was padded with zeros. As there are several possibilities for the padding, I tested to place G in the four corners without noticeable differences. For this reason, I decided to put G in the upper left corner and fill the rest with zeros.
The standard butterfly factorization obtained a matrix approximation error of 57.77. By definition of the algorithm, the rank r of the approximated submatrices is 1 [27]. This leads to an RC of roughly 0.02, corresponding to about 21000 non-zero elements. Even if the computational cost saving is immense, the algorithm performs with a prediction accuracy of 0.70%, only slightly better than a random predictor (0.01%). The truncated SVD, with similar RC, performed, with a prediction accuracy of 2.22%, a bit better then the Butterfly factorization. However, both factorizations perform insufficiently for real-world applications. It can be seen that too few parameters are not sufficient to approximate G well enough. This results in the poor prediction accuracy of the networks. Since the recursive factorization with higher rank turned out to be too complex, I tried to perform the middle level factorization with different RC. The results in comparison to the truncated SVD are shown in plot 5.5.

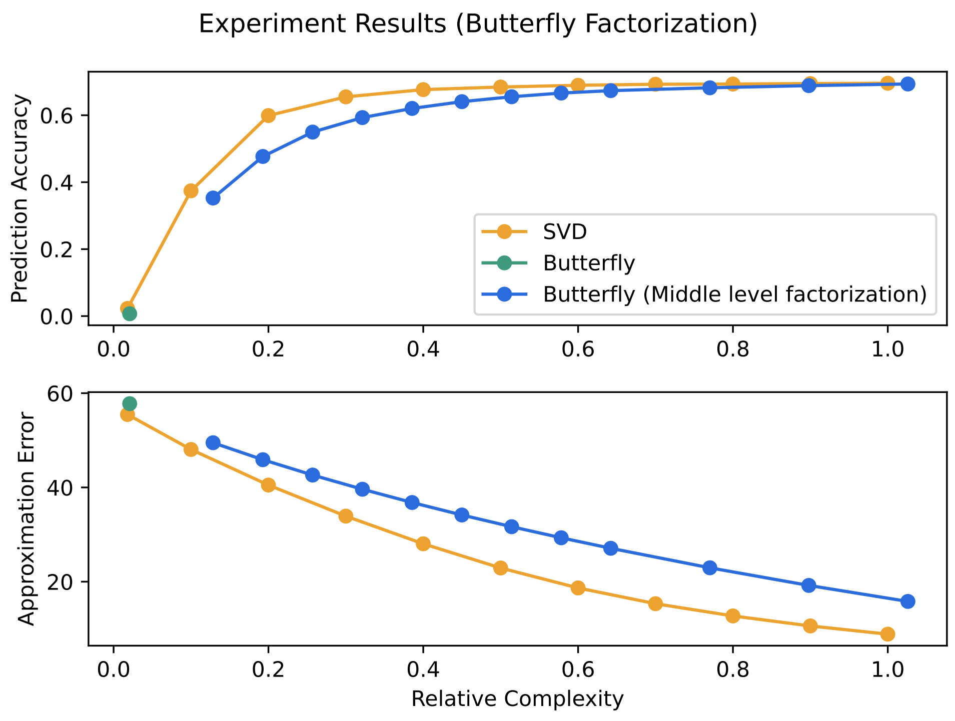

Figure 5.5: The results of the Butterfly factorization were worse than those of the truncated SVD but similar in behavior.

The behavior of the middle-level butterfly factorization is very similar to the truncated SVD. However, the performance is consistently worse. It is also noticeable that the approximation error decreases nearly linear and more slowly compared to SVD.

5.5 FAμST

The search space of the $FA\mu ST$ algorithm is determined by its constraints. For the approximation of G, I chose the most straightforward constraint $\pi(S_i) = k$. This limits the number of non-zero elements per factor to k, which is equal for all factors. In order to be able to adjust the RC of the approximation, the number of parameters k was determined as follows

$$k = \left\lceil \|G\|_0 \frac{RC}{\kappa} \right\rceil. \tag{5.3}$$

Furthermore, the approximation was tested with three and with five factors. The three-factor variant was chosen for better comparison with the SVD approach, which decomposes a matrix into three factors. The five-factor variant was chosen to investigate whether a "deep" approximation is more powerful. The results are shown in plot 5.6.

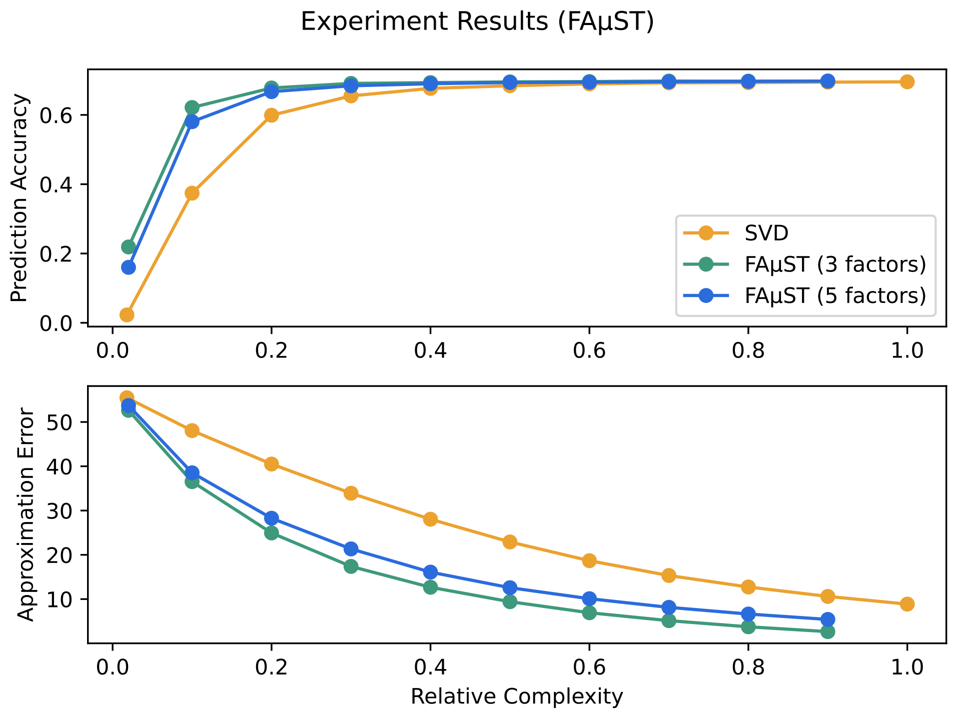

Figure 5.6: The results of the $FA\mu ST$ algorithm were constant better than those of the truncated SVD for the three and five-factor variants. Surprisingly, the five-factor variant performed worse than the three-factor variant.

Plot 5.6 shows that $FA\mu ST$ can approximate G very well, even with few parameters. It is the only algorithm examined that achieved better results than the truncated SVD. The prediction accuracy is already with 20% of the parameters close to the original (1% loss of accuracy). This approximative factorization makes it possible to save a massive amount of resources without significantly affecting the prediction accuracy.

6 Discussion

6.1 Summary and Interpretation

The high resource consumption of modern image processing algorithms is enormous for both training and inference. In the previous chapter, I have shown that resource consumption can be reduced to some extent. The algorithm with the best performance was $FA\mu ST$. It was the only algorithm that performed better than the truncated SVD. However, it was surprising that the five-factor version performed slightly worse than the three-factor version. I assume that this is mainly caused by the fact that the number of parameters per factor in the five-factor version is heavily reduced compared to the three-factor version. Since the factors are trained hierarchically, it is conceivable that the number of parameters per factor is not sufficient to obtain solid results.

The good approximation of $FA\mu ST$ is quite interesting for other applications, like the approximative solving of inverse problems. The approximation error measured by the Frobenius norm, for example was already with a RC of 0.3 as good as that of the Butterfly factorization with a RC of 1, similar applies to the sparse EigenGame.

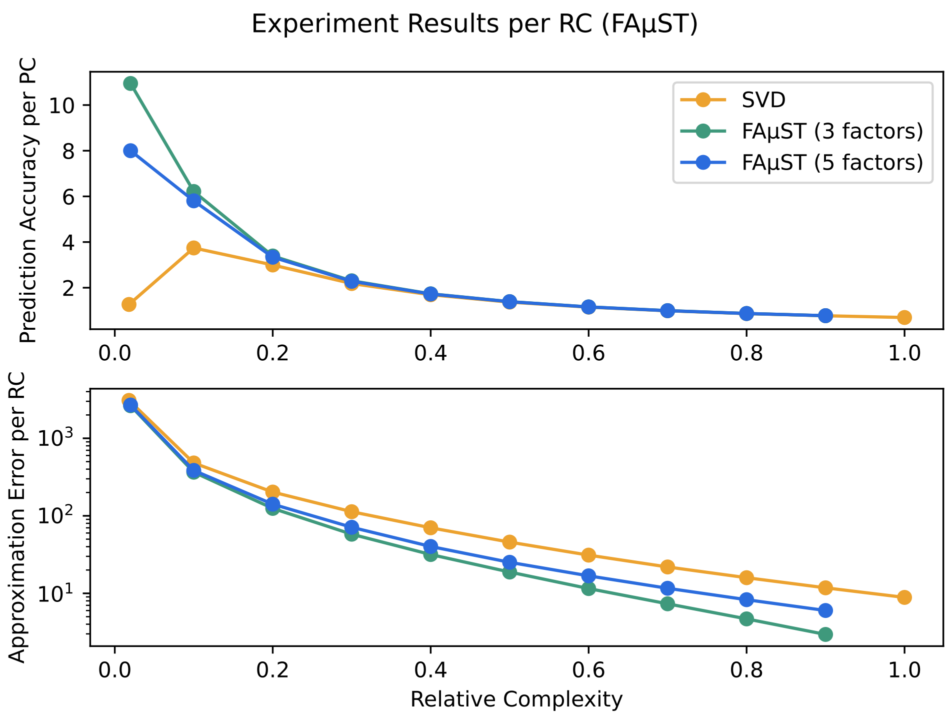

Figure 6.1: Approximation results normalized to the RC of $FA\mu ST$. For comparison, the results of the truncated SVD are presented.

Plot 6.1 shows the relative prediction accuracy as well as the relative approximation error normalized to the RC to depict the relative information content per parameter of the $FA\mu ST$ algorithm and the truncated SVD. For the truncated SVD, the approximation error per RC was expected to decrease steadily as the information content of the singular vectors decreases monotonically. This was confirmed in the experiments. The relative prediction accuracy increases for smaller values up to the maximum at a RC of 0.1, for smaller values of RC the relative prediction accuracy collapses. I assume that this is because the number of parameters is no longer sufficient to approximate G well enough, hence the predictions are poor.

The performance of the $FA\mu ST$ algorithm was similar to the truncated SVD for the relative approximation error. The relative prediction accuracy, on the other hand, was still high even for very small values of RC. It can be suspected that this is since the search space of the $FA\mu ST$ algorithm is less restricted than that of the truncated SVD. $FA\mu ST$ only requires that the global sparsity of the factors does not exceed k. The SVD must also satisfy the condition that the singular vectors are orthogonal.

The sparse EigenGame tries to learn "sparse eigenvectors," which are then used as a dictionary for a sparse coding algorithm. The "sparse eigenvectors" were determined using a modified variant of the recently presented EigenGame algorithm. This approach to solve PCA as a game was honored with the outstanding paper award of the International Conference on Learning Representations. However, it should be noted that EigenGame is not really a game since there is no interaction between the players. Information flows only in one direction (to the eigenvector with the smallest eigenvalue). Consequently, the eigenvector with the largest eigenvalue does not interact with any other eigenvector. So I assume that EigenGame will not contribute to a better understanding of k-player games as mentioned in the paper [10]. With this knowledge, the question arises whether it is worth formulating the PCA as a game theory problem.

I assume that the combination of power-iteration and Oja's rule with generalized Gram-Schmidt orthogonalization, coupled with the new perspective on the problem to be solved, is worthwhile. It is an interesting contribution to better parallelization and decentralization, which is a fundamental requirement for the calculation of the PCA of large data sets.

For the sparse EigenGame, the modification was to combine EigenGame with the PALM algorithm. The calculated eigenvectors were projected onto the sparsity constraints in each update step. This sparsity constraint limited the number of non-zero elements per vector. I expected to obtain a well-suited and sparse dictionary used for a sparse coding algorithm to approximate G. I created a similar sparse random dictionary for comparison. In plot 5.4, the results of the two approximations are shown. The predictions of the sparse EigenGame were significantly better than the predictions of the random dictionary, although the approximation error was only slightly better. This fact is quite astonishing. It can be assumed that this is due to the better approximation of the inner structures of the matrix by the "sparse eigenvectors."

However, the results were worse than the truncated SVD with similar RC. This can be explained by the fact that the results of EigenGame and the truncated SVD are identical, and the projections of the PALM algorithm disturbed the results.

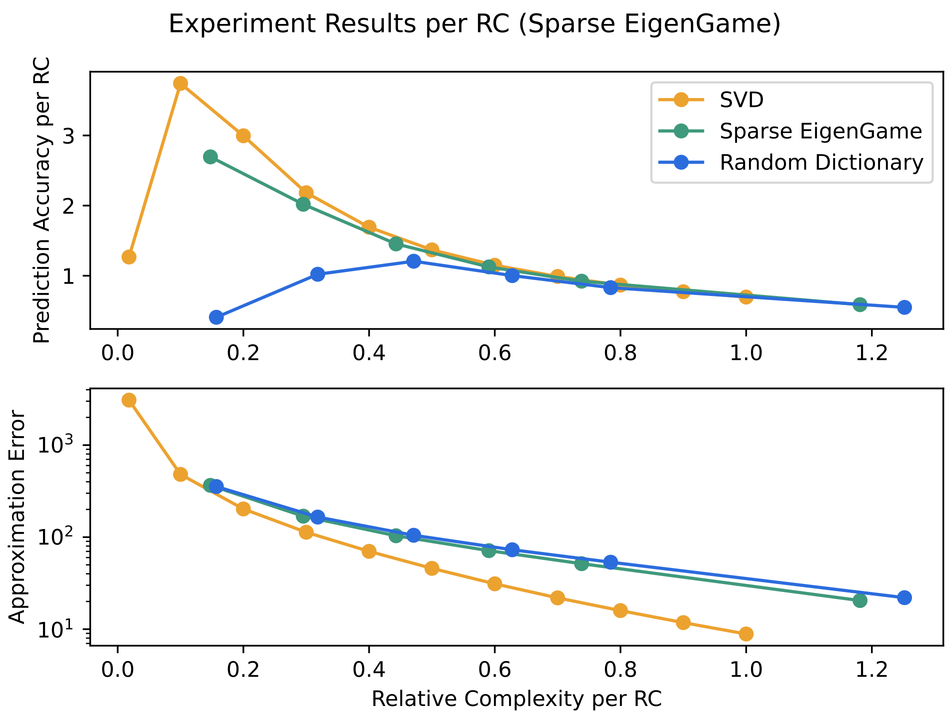

Figure 6.2: Approximation results normalized to the RC of the sparse EigenGame. For comparison, the results of the truncated SVD are presented.

Plot 6.2 shows that the sparse EigenGame achieves similar, but not as good, results as the truncated SVD per RC. The approach to creating a dictionary out of sparse eigenvectors can thus be considered as successful. Compared to the random dictionary, the results are much better. The random dictionary is as expected not able to approximate G well, especially with few parameters.

The results of the butterfly factorization were slightly worse than those of the truncated SVD. However, the algorithm's performance was still quite good (plot 5.5). I suspect that the lower performance is mainly because G suposedly has not hierarchical structure and is not complementary low-rank. Consequently, the low-rank approximations of the subblocks are worse. One can suppose that this is the case for most weight matrices. For other matrices, such as the Fourier matrix, which have the required complementary low-rank property, I obtained better approximation results with the butterfly factorization then with the truncated SVD.

However, other approaches exist to obtain sparse weight matrices in neural networks. In our case, I first trained a neural network and then approximated the last layer of the network as a product of sparse factors. Another approach is to specify certain structures in the weight matrices and then train the network. These structures are usually hand-crafted and combine fast transformations like the Hadamard or fast Fourier transformation, the selection of suitable transformations is generally costly [27]. The butterfly structure can solve this problem, as it can learn several fast transformations. The structure is depicted in figure 6.3 for a 64×64 matrix.

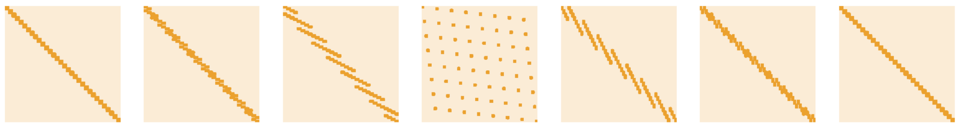

Figure 6.3: Decomposition of a 64 x 64 random matrix into its butterfly factors non-zero elements are dark orange zero elements light orange.

The time-consuming selection process is thus eliminated [6]. Experiments have already shown that the butterfly structure is well suited to replace fully connected layers, higher prediction accuracies were obtained for weight matrices with butterfly structure than for fully connected layers [6]. These results are astonishing since the network is able to learn the butterfly structure even without specification.
However, the same applies to convolutional layers in CNNs, which specify structures that can also be learned by a fully connected network [30]. Nevertheless, CNNs are better suited for image processing and achieve more accurate predictions. This is achieved by exploiting the local spatial coherence of images in the convolutional layer. By considering this fact it is conceivable, that the reason for the good performance of butterfly layers is similar to these of CNNs.

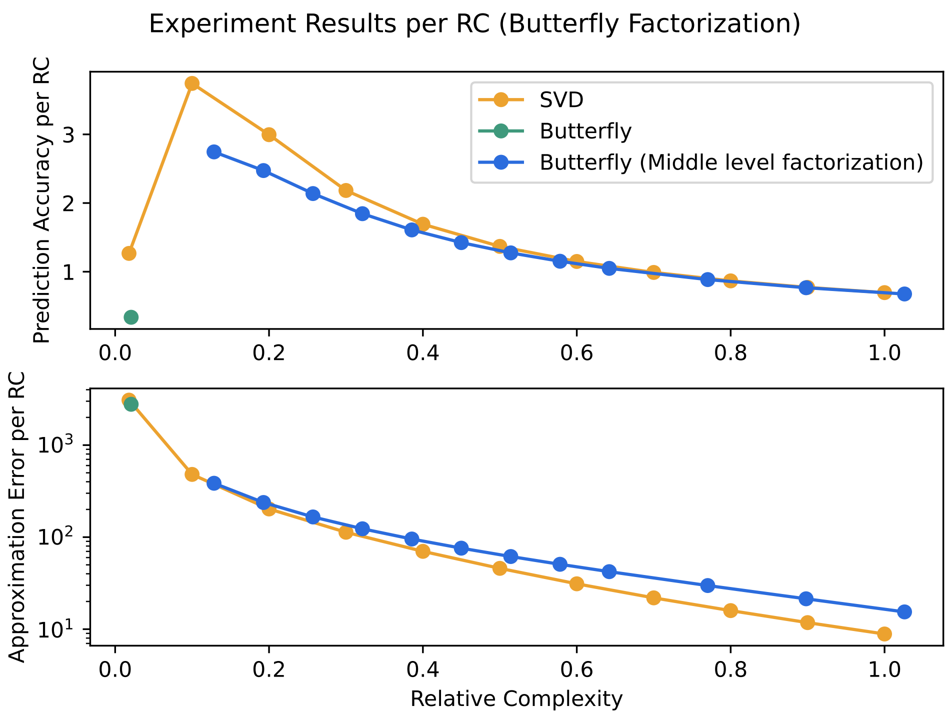

Figure 6.4: Approximation results normalized to the RC of the butterfly factorization. For comparison, the results of the truncated SVD are presented

As expected, the relative prediction accuracy as well as the relative approximation error per RC behaves very similarly to SVD, as depicted in figure 6.4. I suspect that the reasons for this are identical.

6.2 Limitations and Recommendations

In this thesis, the last layer of the googlenet network was approximated with sparse factors to reduce the resource consumption of the inference. For simplicity, the thesis was limited to the approximation of one layer. Approximating all layers would go beyond the scope of this work. However, it should be noted that all layers must be considered if one aims to reduce resource consumption significantly. Furthermore, methods that approximate G with other structures and methods that assume specific structures during the training of the networks were not examined. It is difficult to assess whether the results obtained for googlenet can be applied to other networks. This was also not investigated in this thesis.

In conclusion, the $FA\mu ST$ algorithm is best suited to decompose the weight matrix

G into sparse factors after training. It is recommended to test this also for other pre-trained neural networks and also for more layers. The butterfly algorithm did not perform better than the truncated SVD. Still, it has been shown in other experiments, as mentioned previously, that the butterfly structure can perform better than a fully connected layer. This property makes the butterfly structure probably the most interesting algorithm of this work. It is advisable to investigate this in further work. However, the algorithm is not suggested for sparse approximation after training. The sparse EigenGame also performed worse than the comparative algorithm, but it could be optimized considerably. For example, it would be conceivable to continuously reduce the vectors sparsity to obtain better approximations.

7 Conclusion

This thesis aimed to reduce resource consumption for the inference of neural networks. This should be achieved by sparse approximative factorization of the weight matrices. In the context of this work only the last layer of the network was approximated, however it is to be noted that all layers must be considered in order to reduce the resource consumption significantly. Also, only the fully connected layers were analyzed in order not to go beyond the scope of this work. For this purpose, three algorithms were investigated, namely, Butterfly factorization, sparse EigenGame, and $FA\mu ST$. To test the applicability of the approximate sparse factorization for real-world problems, I approximated the last layer of the googlenet network. Then, the neural network with approximated last layer was validated with the ImageNet validation data from 2012 to test whether the approximations achieved valid results. Based on the performed experiments, it can be stated that algorithms exist which are well suited for such a factorization. The $FA\mu ST$ algorithm was able to reduce the parameters of the last fully connected layer to fifth and still achieved results that were only 1% worse than the original layer. The other algorithms did not perform as well. However, it was made clear that different approaches exist to obtain sparse layers. In our case, neural networks were trained first, then the last layer of the network was approximated by a product of sparse matrices. Other methods specify certain structures in the matrices and then train the networks. Other experiments showed that the butterfly structure is better suited for the second approach [27]. The sparse EigenGame is not the best choice for either the first or second approaches.
In conclusion, the goal of reducing the resource consumption of CNNs, through the approximate sparse factorization, has been achieved. Based on these results, it is conceivable to approximate all layers in neural networks by a product of sparse matrices. This can reduce the resource consumption of neural networks considerably, possibly allowing an application in time-critical systems or low-resource devices.

Acronyms

$FA\mu ST$	Flexible Approximate MUlti-layer Sparse Transform. 3, 26, 28, 35, 41–44, 48, 51, 56
AI	Artificial Intelligence. 7
ASF	Approximative Sparse Factorization. 3, 19
CNN	Convolutional Neural Network. 3, 7, 8, 10–12, 47, 51
CV	Computer Vision. 7
FLOP	Floating-point Operation. 3, 9
ILSVRC	ImageNet Large Scale Visual Recognition Challenge. 8, 13
IoT	Internet of Things. 10
LARS	Least Angle Regression. 32, 33
ML	Machine Learning. 13
NN	Neural Network. 7, 8, 11, 12
OMP	Orthogonal Matching Pursuit. 32
PALM	Proximal Alternating Linearized Minimization. 27, 28, 45
PCA	Principal Component Analysis. 16, 29, 30, 45
RC	Relative Complexity. 35, 38–41, 43–46, 48, 56
SVD	Singular Value Decomposition. 15, 16, 24, 26, 37–46, 48, 49, 56

List of Figures

Bibliography

[1] S. S. Basha, S. R. Dubey, V. Pulabaigari, and S. Mukherjee. "Impact of fully connected layers on performance of convolutional neural networks for image classification". In: *Neurocomputing* 378 (2020), pp. 112–119.

[2] L. Beyer, O. J. Hénaff, A. Kolesnikov, X. Zhai, and A. van den Oord. "Are we done with ImageNet?" In: *CoRR* abs/2006.07159 (2020).

[3] S. Bosch. *Lineare Algebra*. Vol. 4. Springer, 2006.

[4] R. Bro and A. K. Smilde. "Principal component analysis". In: *Analytical methods* 6(9) (2014), pp. 2812–2831.

[5] A. Canziani, A. Paszke, and E. Culurciello. "An Analysis of Deep Neural Network Models for Practical Applications". In: *CoRR* abs/1605.07678 (2016).

[6] T. Dao, A. Gu, M. Eichhorn, A. Rudra, and C. Ré. "Learning fast algorithms for linear transforms using butterfly factorizations". In: *International conference on machine learning*. PMLR. 2019, pp. 1517–1527.

[7] C. De Sa, A. Cu, R. Puttagunta, C. Ré, and A. Rudra. "A two-pronged progress in structured dense matrix vector multiplication". In: *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM. 2018, pp. 1060–1079.

[8] B. Efron, T. Hastie, I. Johnstone, and R. Tibshirani. "Least angle regression". In: *The Annals of statistics* 32(2) (2004), pp. 407–499.

[9] K. Fukushima and S. Miyake. "Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition". In: *Competition and cooperation in neural nets*. Springer, 1982, pp. 267–285. ISBN: 978-3-642-46466-9.

[10] I. Gemp, B. McWilliams, C. Vernade, and T. Graepel. "EigenGame: PCA as a Nash Equilibrium". In: *International Conference on Learning Representations*. 2020.

[11] A. Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, 2019. ISBN: 978-1-4920-3264-9.

[12] G. H. Golub and H. A. Van der Vorst. "Eigenvalue computation in the 20th century". In: *Journal of Computational and Applied Mathematics* 123(1-2) (2000), pp. 35–65.

[13] W. Hackbusch. *Hierarchical matrices: algorithms and analysis*. Vol. 49. Springer, 2015. ISBN: 978-3-662-47323-8.

[14] C. O. B. Hage. "Robust Structured and Unstructured Low-Rank Approximation on the Grassmannian". Ph.D. thesis. Technische Universität München, 2017.

[15] K. He, X. Zhang, S. Ren, and J. Sun. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[16] D. O. Hebb. *The organization of behavior: A neuropsychological theory*. Psychology Press, 2005.

[17] C. A. Holt and A. E. Roth. "The Nash equilibrium: A perspective". In: *Proceedings of the National Academy of Sciences* 101(12) (2004), pp. 3999–4002.

[18] J. Hu, L. Shen, and G. Sun. "Squeeze-and-excitation networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 7132–7141.

[19] D. H. Hubel and T. N. Wiesel. "Receptive fields and functional architecture of monkey striate cortex". In: *The Journal of physiology* 195(1) (1968), pp. 215–243.

[20] P. Jain, P. Kar, et al. "Non-convex Optimization for Machine Learning". In: *Foundations and Trends® in Machine Learning* 10(3-4) (2017), pp. 142–363.

[21] A. Krizhevsky, I. Sutskever, and G. E. Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems* 25 (2012), pp. 1097–1105.

[22] T. Kurita. "Principal Component Analysis (PCA)". In: *Computer Vision: A Reference Guide*. Ed. by K. Ikeuchi. Cham: Springer International Publishing, 2021, pp. 1013–1016. ISBN: 978-3-030-63416-2.

[23] L. Le Magoarou and R. Gribonval. "Flexible multilayer sparse approximations of matrices and applications". In: *IEEE Journal of Selected Topics in Signal Processing* 10(4) (2016), pp. 688–700.

[24] Y. LeCun, B. Boser, J. Denker, D. Henderson, R. Howard, W. Hubbard, and L. Jackel. "Handwritten digit recognition with a back-propagation network". In: *Advances in neural information processing systems* 2 (1989).

[25] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86(11) (1998), pp. 2278–2324.

[26] H. Lee, A. Battle, R. Raina, and A. Y. Ng. "Efficient sparse coding algorithms". In: *Advances in neural information processing systems*. 2007, pp. 801–808.

[27] Y. Li, H. Yang, E. R. Martin, K. L. Ho, and L. Ying. "Butterfly factorization". In: *Multiscale Modeling & Simulation* 13(2) (2015), pp. 714–732.

[28] F. Lin and W. W. Cohen. “Power iteration clustering”. In: *ICML*. 2010, pp. 655–662.

[29] Y. Liu, X. Xing, H. Guo, E. Michielssen, P. Ghysels, and X. S. Li. “Butterfly factorization via randomized matrix-vector multiplications”. In: *SIAM Journal on Scientific Computing* 43(2) (2021), A883–A907.

[30] W. Ma and J. Lu. “An equivalence of fully connected layer and convolutional layer”. In: *arXiv preprint arXiv:1712.01252* (2017).

[31] M. Motamedi, D. Fong, and S. Ghiasi. “Fast and energy-efficient cnn inference on iot devices”. In: *arXiv preprint arXiv:1611.07151* (2016).

[32] R. B. Myerson. *Game theory*. Harvard university press, 2013.

[33] J. Nash. “Non-cooperative games”. In: *Annals of mathematics* (1951), pp. 286–295.

[34] B. Neyshabur and R. Panigrahy. “Sparse matrix factorization”. In: *arXiv preprint arXiv:1311.3315* (2013).

[35] E. Oja. “Simplified neuron model as a principal component analyzer”. In: *Journal of mathematical biology* 15(3) (1982), pp. 267–273.

[36] V. Pan. *Structured matrices and polynomials: unified superfast algorithms*. Springer Science & Business Media, 2001.

[37] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. “Scikit-learn: Machine learning in Python”. In: *Journal of machine learning research* 12(Oct) (2011), pp. 2825–2830.

[38] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. “ImageNet Large Scale Visual Recognition Challenge”. In: *International Journal of Computer Vision (IJCV)* 115(3) (2015), pp. 211–252.

[39] G. Strang, G. Strang, G. Strang, and G. Strang. *Introduction to linear algebra*. Vol. 3. Wellesley-Cambridge Press Wellesley, MA, 1993. ISBN: 978-1-73314-665-4.

[40] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. “Going deeper with convolutions”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 1–9.

[41] R. Vandebril, M. Van Barel, and N. Mastronardi. *Matrix computations and semiseparable matrices: linear systems*. Vol. 1. JHU Press, 2007.

[42] M. E. Wall, A. Rechtsteiner, and L. M. Rocha. “Singular value decomposition and principal component analysis”. In: *A practical approach to microarray data analysis*. Springer, 2003, pp. 91–109.

[43] M. Yang, S. Wang, J. Bakita, T. Vu, F. D. Smith, J. H. Anderson, and J.-M. Frahm. “Re-thinking CNN frameworks for time-sensitive autonomous-driving applications: Addressing an industrial challenge”. In: *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2019, pp. 305–317.