Technical University of Denmark

Master Thesis in Mathematical Modelling and Computation

Evaluation of synergistic effects of tensor decomposition methods within (deep) neural network applications

Tobias Engelhardt Rasmussen, s153057

Supervisors

Andreas Baum (andba@dtu.dk)
Line Katrine Harder Clemmensen (lkhc@dtu.dk)

February 2021

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Abstract

Preface

This Master Thesis; "Evaluation of synergistic effects of tensor decomposition methods within (deep) neural network applications" is conducted in order to fulfill the requirements for achieving the Master of Science in Engineering; Mathematical Modelling and Computing. The work is carried out in the fall semester of 2020 at the Department of Applied Mathematics and Computer Science (DTU Compute) at the Technical University of Denmark in Kongens Lyngby. The project was supervised by Andreas Baum, Assistant Professor, and Line Katrine Harder Clemmensen, Associate Professor, both at DTU Compute.

Kongens Lyngby, January 2021

Tobias Engelhardt Rasmussen

Contents

List of Figures

List of Tables

List of Abbreviations

ALS	Alternating Least Squares
ANN	Artificial Neural Network
API	Application Programming Interface
BTD	Block Term Decomposition
CNN	Convolutional Neural Network
CP	CANDECOMP / PARAFAC / Canonical (Polyadic) Decomposition
FLOP	FLoating point OPeration
GPU	Graphics Processing Unit
HOOI	Higher-Order Orthogonal Iteration
HOSVD	Higher-Order Singular Value Decomposition
ML	Machine Learning
MNIST	Modified National Institute of Standards and Technology
NN	Neural Network
RGB	Red Green Blue
SVD	Singular Value Decomposition
THETIS	THree dimEnsional TennIs Shots
TNN	Tensorized Neural Network
TT	Tensor Train (decomposition)
VBMF	Variational Bayesian Matrix Factorization

Nomenclature

In this thesis, vectors will be denoted as bold lower case letters $\boldsymbol{x}$, while bold upper case letters denote matrices $\boldsymbol{X}$, with elements $\boldsymbol{X}(i,j) = x_{i,j}$. Tensors of dimension at least 3 are written in calligraphic letters, hence $\mathcal{X}^{d_1 \times d_2 \times \cdots \times d_N}$ is a tensor of order N with elements $\mathcal{X}(i_1, i_2, \ldots, i_N) = x_{i_1, i_2, \ldots, i_N}$. A list of symbols used throughout the thesis is given below.

Symbol	Description	Operation
$\langle \mathcal{A}, \mathcal{B} \rangle$	Inner product	$\langle \mathcal{A}, \mathcal{B} \rangle = \sum_{i,j,k} a_{i,j,k} \cdot b_{i,j,k}$
$\circ$	Outer product	$\boldsymbol{a} \circ \boldsymbol{b} = \boldsymbol{C}$ where $c_{i,j} = a_i \cdot b_j$
$\boldsymbol{X}^\top$	Transpose	$\boldsymbol{X}^\top(i,j) = \boldsymbol{X}(j,i)$
$\|\mathcal{A}\|_F$	Frobenius norm	$\sqrt{\langle \mathcal{A}, \mathcal{A} \rangle}$
$\boldsymbol{X}_{(n)}$	Matricizing	$\mathcal{X}^{I_1 \times I_2 \times \ldots I_N} \longrightarrow \boldsymbol{X}_{(n)}^{I_n \times I_1 \cdot I_2 \cdots I_{n-1} \cdot I_{n+1} \cdots I_N}$
$\times_n$	n-mode product	$\mathcal{X} \times_n \boldsymbol{M} = \mathcal{Z}$ where $\boldsymbol{Z}_{(n)} = \boldsymbol{M}\boldsymbol{X}_{(n)}$
$\otimes$	Kronecker product	$\boldsymbol{A} \otimes \boldsymbol{B} = \boldsymbol{C}$ where $\boldsymbol{C}\,(k + K(i-1), l + L(j-1)) = a_{ij} \cdot b_{kl}$
$\boldsymbol{X}^\dagger$	Moore-penrose inverse	$\boldsymbol{X}^\dagger = (\boldsymbol{X}^\top \boldsymbol{X})^{-1} \boldsymbol{X}^\top$
$\odot$	Khatri-Rao product	$\boldsymbol{A} \odot \boldsymbol{B} = \boldsymbol{C}$ where $\boldsymbol{C}\,(k + K(i-1), j) = a_{ij} b_{kj}$

1 Introduction

Machine Learning (ML) has been a very useful tool in many areas for a long time. One of the biggest leaps in the discovery of the computer was Alan Turing's learning machine that was used to decrypt the German cipher during World War II [25]. Alan Turing was widely considered the father of computer science and artificial intelligence[1]. Ever since his work and with computers becoming better and more accessible, people have studied different ways of making machines learn different tasks. Today ML is used for everything from recognising handwriting to diagnosing breast cancer[9].

One of the most famous and versatile ML algorithms is the neural network (NN), which builds upon the idea of the brain with neurons firing up and sending information to each other through a network. NNs are very powerful since they are easy to train (finding the optimal weights), and easy to modify to fit almost any given problem. This means that NNs achieve state-of-the-art performance in a large variety of problems. Artificial neural networks (ANNs) are simply a set of neurons arranged in layers sending information from one layer to another from the input to the output. Stacking multiple layers of neurons makes the algorithm able to extract higher level features[26], hence many layers (deep learning) are often preferred for complex problems.

A big area in ML and NN research is image classification and detection, which is respectively to classify an image into a set of predetermined classes or detect a given object in an image. By far the major method in learning images is the convolutional neural network (CNN). In a CNN a number of filters are trained and run through the image to calculate activations. These filters will then be trained to see different features in the image such as lines or curves. Stacking multiple layers of convolutions enables the algorithm to learn more abstract features such as faces, printed text or wrinkly clothes[28]. CNNs are the preferred NN architectures for image analysis, since they keep achieving state-of-the-art performance.

1.1 The problem

Even though NNs are very powerful and there is nothing to be said about their ability to learn, there is one problem. The amount of parameters to be trained and the amount of time it takes grow rapidly as the number of layers and the amount of neurons in these layers grow. Also the amount of parameters grow exponentially as the input resolution increases. For instance the MNIST data set, which consists of 28×28 pixel values per image will need $28 \cdot 28 = 784$ input neurons. If one is to have a fully connected (dense) layer from these inputs to another layer of just 100 neurons, the amount of parameters to be trained is $784 * 100 + 100 = 784,100$ (there is one bias term per neuron). Denil et al. found in 2013 that many of the weights in a deep NN are redundant[3]. They found that if only a fraction of the weights were trained and the rest predicted from these, they would get close to the same accuracy as a fully trained network. This means that a high accuracy should be obtainable from a network with fewer parameters, if the architecture and algorithm itself are chosen wisely.

Many approaches have been attempted to solve this problem. For instance Liu et al. in 2015 used

pruning to zero-out[1] more than 90 % of the learned parameters with a loss in accuracy of under 1% on a given data set. Also there have been many attempts to use tensor decomposition to reduce the number of parameters. Tensor decomposition is a way of approximating a multi-dimensional data array using a limited amount of parameters by looking at the variation in different dimensions. Numerous approaches have been generalizing this concept to change the architecture as in the work of Lebedev et al. in [13] and of Wang et al. in 2016 [24]. These methods decompose the weights of a pre-trained network and use the decompositions in the new architecture. After fine-tuning the performance should be increased both with respect to the number of parameters and the computation time, while the accuracy should remain approximately the same.

1.2 Aim of this project

This thesis investigates previous attempts and other potential approaches to evaluate the potential synergistic effects of using these methods within a deep NN. The aim is to examine the effects of using different tensor decomposition methods either before (as input) or within a potentially deep NN. The methods will be investigated and compared to baseline results in order to analyze the accuracy, running times, and number of parameters. Two data sets will be used to assess the performance differences. The first is the rather simple MNIST (Modified National Institute of Standards and Technology) data set[14], which consists of handwritten digits from 0 to 9. The other is the THETIS (Three dimEnsional TennIs Shots) data set [4], which consists of videos of individuals performing different types of tennis shots. The purpose is to first apply the methods to the simpler MNIST data set, since this is well-studied and relatively easy to work with, and then apply them to the more complicated THETIS data set in order to test the methods on a more demanding problem.

[1] Setting insigificant parameters to 0 (parameters that are already relatively close to 0)

2 Theory

In this section, the theory used to develop the methods and obtain the result in this thesis will be discussed. First, neural networks (NNs) will be covered - both the simpler artificial NNs (ANNs) and more complex networks such as the convolutional NN (CNN) in both 2 (image) and 3 (videos) dimensions. Also methods for calculating the theoretical speed of a NN will be covered. This is followed by the theory behind tensors and different methods in tensor decomposition. There will be more focus on Tucker decomposition since it is the one used in this thesis, and how to choose the appropriate rank in the case of Tucker decomposition.

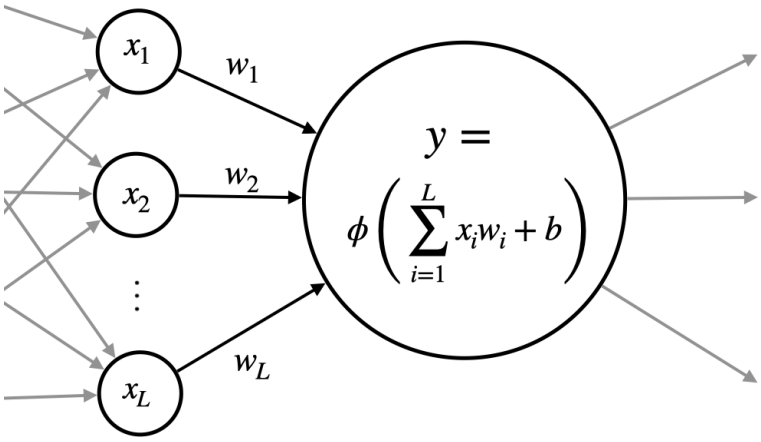


Figure 2.1: Illustration of the flow of information through a neuron. Each edge (arrow) has a weight w associated with it, and each neuron (circle) has an activation associated with it. The activations correspond to the information that is passed through the network, i.e. in this case the L xs are used as inputs into the big neuron, where y will be the resulting activation. ϕ is a non-linear transform called the activation function.

2.1 Neural Networks

The NN is a very useful tool in non-linear classification and regression. It gets its name from what gave rise to the idea of it around half a century ago - the brain. Similar to the brain, a NN consists of a network of neurons that pass on information between each other. The way this is done is illustrated in Figure 2.1. Each neuron is, with a given input, associated with a value called its activation. This activation is based on the inputs to that very neuron, and is what is passed on from that neuron. The activation is calculated as the weighted sum of the activations of the inputs to that neuron plus a bias, all non-linearly transformed using an activation function ϕ. Typical choices of ϕ are given in Figure 2.2, while the typical activation of the last layer (the output) is the softmax-function, given by:

$$\text{Softmax: } \sigma : \mathbb{R}^K \to \mathbb{R}^K \qquad \sigma(\boldsymbol{x})_i = \frac{e^{x_i}}{\sum_{k=1}^{K} e^{x_k}} \quad \text{where } \boldsymbol{x} \in \mathbb{R}^K \tag{1}$$

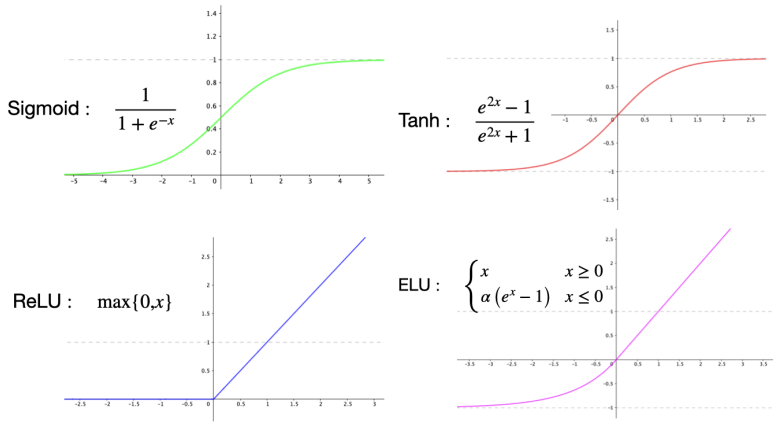


Figure 2.2: Typical activation functions used in NNs which are all non-linear transforms. The dotted grey lines represent the values of -1 and 1.

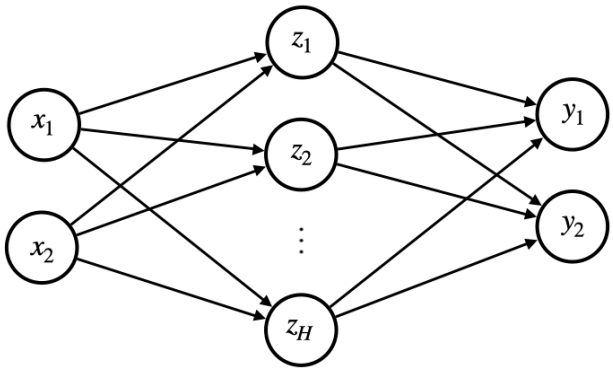


Figure 2.3: Illustration of a very simple ANN with only 2 inputs (x_1, x_2), 2 output (y_1, y_2), and a single hidden layer with H neurons $(z_1, z_2, \ldots, z_H)$.

This is due to its ability to ensure a sum of 1 of all the outputs. In the end this value is the probability of the input belonging to each of the output classes. The predicted class is the one that has the highest probability.

The size and layout of a NN is called the network architecture. The simplest type of NN architecture is the ANN. An example of a very simple ANN with only 2 input values for each observation, and binary output is given in Figure 2.3. This network has only a single hidden layer with H neurons.

2.1.1 Learning

Learning a NN corresponds to finding the weights of the network that give the highest output accuracy - the best predictions. The method used to find the optimal weights is called back-propagation. This method simply sends inputs through the network and read the outputs. The output error is

then propagated backwards through the network in order to adjust the weights accordingly. This is done repeatedly until convergence, normally dividing the training data into smaller batches and performing corrections using every batch. If the amount of data is limited, cross-validation is often used, which trains and validates on the same data. In cross-validation the data is split into a set of folds, and the network is trained on all the data except for one fold which is then used for validating the results.

To represent the error, a so-called loss function is used. Given an observation with input $\boldsymbol{x}$ and the corresponding true prediction y, we introduce the cost function given by:

$$L(\Theta|\boldsymbol{x}, y) = -\hat{p}(y) + \log\left(\sum_{k=0}^{K} e^{\hat{p}(k)}\right) \tag{2}$$

Where Θ represents all the weights of the network, K is the number of output classes, and $\hat{p}(k)$ is the predicted probability of the input belonging to class k. This function is also called the *cross-entropy* loss function and is a the typical choice in classification problems. The cross-entropy loss function can be seen as a function of the weights of the network (here Θ) because the weights are used to find the predictions $\hat{p}$. The gradient of this loss-function with respect to each of the weights in the network is used to update the weights accordingly by a gradient-descent step:

$$\Theta_{new} = \Theta - \gamma \, \nabla L(\Theta) \tag{3}$$

Where γ is the learning rate that can be used to determine the speed of the learning, i.e. bigger values implies bigger jumps, while smaller values are used for fine-tuning. This derivative can be found either analytically, by applying the chain rule backwards through the network, or automatically using the *autograd*-function in *PyTorch*. In this thesis the *autograd*-function will be used to find the derivatives, hence for the analytical derivations cf. [6] or [29].

2.1.2 Architectures

NNs are very flexible and can easily be adapted to fit every need. Not only can they be built off an arbitrary number of layers of different sizes, but also there are multiple different types of layers. Each layer finds progressively more complex features in the input data, hence more layers will be able to find more complex features. NNs can be both feed-forward (information flows forward) or recurrent (information flows backwards). The two types of layers that we will cover in this thesis are the dense layer and the convolutional layer respectively. Both have powerful properties and will be discussed further in the following.

2.1.2.1 The dense layer

The dense layer is the basic building block of NNs, and ANNs are almost always built of these. A dense layer (also called fully-connected layer) is a layer in which each of neurons have a connection with all the neurons in the previous layer. Both the middle and output layers in the network illustrated in Figure 2.3 are dense layers. In CNNs the dense layers are often used in the last part of the network to condense the information from the convolutional layers.

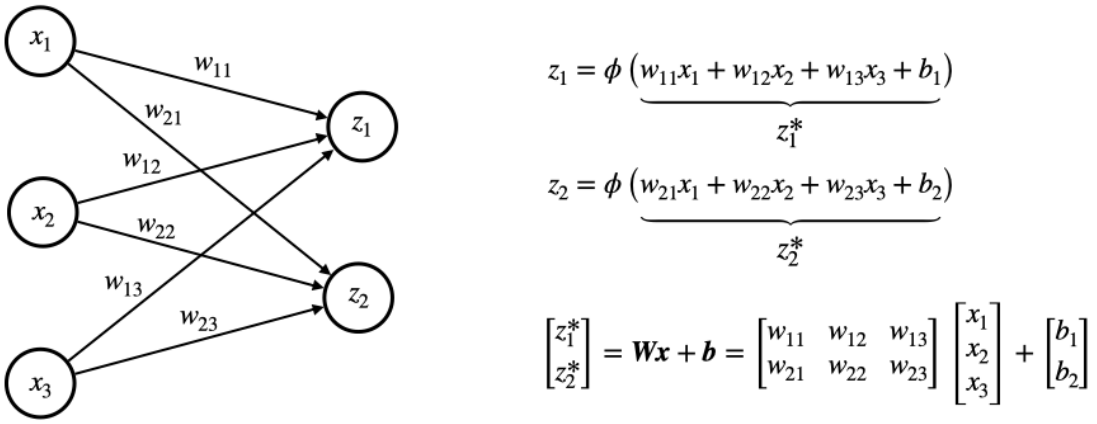


Figure 2.4: Illustration of the calculation of the dense layer. x_i and z_i are the ith activation of the two layers respectively. w_{ji} is the weight corresponding to the connection between x_i and z_j, and b_i is the bias to be added to z_i. The dense layer is calculated as the non-linear transform of the bias plus the matrix-vector product between the weight matrix $\boldsymbol{W}$ and the input vector $\boldsymbol{x}$. ϕ is the activation function or non-linear transform.

The dense layer is calculated as the non-linear transform of a matrix-vector product with the matrix consisting of the weights on the edges connecting the layer to the previous, and the vector being the input from the previous layer. This concept is illustrated in Figure 2.4.

2.1.2.2 The convolutional layer

The convolutional layer has become a very powerful tool in image classification due to its ability to find features in images using the spatial information. In a convolutional layer the weights are arranged in a number of filters, each of which can be trained to look for specific features. For instance, a low-level filter can search for lines or curves in various directions, while higher-level filters look for e.g. faces, bodies, or wrinkles on clothes [28]. The filters are placed in every position in the input image to calculate the activation at that position simply by multiplying the weights of the filter by the image and summing everything. The resulting activations will also be arranged as an image with high values corresponding to high activations.

Since color-images have multiple input channels (one red, one green, one blue), they are represented as order-3 tensors of size $H \times W \times S$, where H is the image height in number of pixels, W is the width of the image, and S is the number of input channels. The convolution of an input image $\mathcal{X}$ with this size into an output tensor $\mathcal{Y}$ of size $H' \times W' \times T$ is given by:

$$\mathcal{Y}(h', w', t) = \sum_{i=1}^{D_H} \sum_{j=1}^{D_W} \sum_{s=1}^{S} \mathcal{K}(i, j, s, t)\ \mathcal{X}(h_i, w_j, s) \tag{4}$$

Where:

$$h_i = (h' - 1)\,\Delta_H + i - P_H \qquad w_j = (w' - 1)\,\Delta_W + j - P_W \tag{5}$$

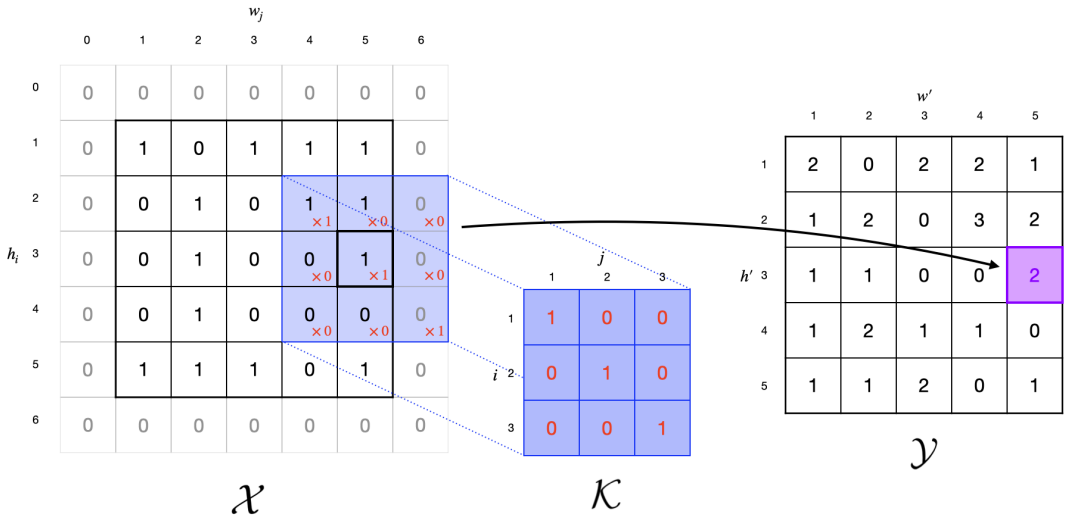


Figure 2.5: Illustration of a simple convolution of a picture with a single input channel. The calculation of the value $\mathcal{Y}(3, 5, \cdot)$ is the one that is highlighted. The arrow signifies the sum of the result of applying the filter $\mathcal{K}$ to the input image $\mathcal{X}$. The input image will always be 1-indexed, hence the padding (grey 0s) is indexed appropriately. The very simple filter is trained to look for diagonal lines, hence the output will have a high activation if a diagonal line is present in $\mathcal{X}$. If there are more input channels, the input image $\mathcal{X}$ and the filter $\mathcal{K}$ will both be 3-dimensional tensors.

In (4) $\mathcal{K}$ is the 4-dimensional kernel of size $D_H \times D_W \times S \times T$ which corresponds to the T different stacked filters of size $D_H \times D_W \times S$. Δ corresponds to the stride, i.e. how much the filters should move between each calculation (usually 1). P corresponds to the padding which is how many 0s should be put on the edges of the image, and can be used to ensure the same input and output size in terms of the spatial dimensions. (h_i, w_j) is the top left corner of the position in the input image and the three sums correspond to the application of the tth filter to that position. The application of a filter to an image with a single input channel is illustrated in Figure 2.5. The output size $H' \times W'$ is calculated by the formula:

$$H' = \frac{H - D_H + 2 \cdot P_H}{\Delta_H} + 1 \tag{6}$$

$$W' = \frac{W - D_W + 2 \cdot P_W}{\Delta_W} + 1 \tag{7}$$

The 3D convolution in videos There are multiple ways of doing convolutions for videos, that are simply a stack of images. One way is to do convolutions on every frame in a sequence and gathering the result - another way is to apply a 3-dimensional filter.[2] The 3-dimensional filter will not only look for features in the spatial dimensions, but also for temporal information, i.e.

[2]This filter will actually be 4-dimensional if there are multiple input dimensions

movement. The 3D convolution is very similar to the 2D case (4), except for another sum and higher dimensions. The 3D convolution of an input video $\mathcal{X}$ of size $F \times H \times W \times S$ (F is the frames) into an output tensor $\mathcal{Y}$ of size $F' \times H' \times W' \times T$ is given by:

$$\mathcal{Y}(f', h', w', t) = \sum_{i=1}^{D_F} \sum_{j=1}^{D_H} \sum_{l=1}^{D_W} \sum_{s=1}^{S} \mathcal{K}(i, j, l, s, t)\ \mathcal{X}(f_i, h_j, w_l, s) \tag{8}$$

Where:

$$f_i = (f' - 1)\,\Delta_F + i - P_F \qquad h_j = (h' - 1)\,\Delta_H + j - P_H \qquad w_l = (w' - 1)\,\Delta_W + l - P_W \tag{9}$$

Now the convolutional kernel $\mathcal{K}$ is 5-dimensional and have the size $D_F \times D_H \times D_W \times S \times T$, which is a stack of 4-dimensional filters. The output size F' is calculated in the same way as (6) and (7) by:

$$F' = \frac{F - D_F + 2 \cdot P_F}{\Delta_F} + 1 \tag{10}$$

2.2 Tensors and Tensor Decomposition

A tensor is a data array of an arbitrary number of dimensions, hence matrices are order-2 tensors, while vectors are order-1 tensors. Tensors are frequently used in many different fields that have to do with computer science[15]. Tensors have many great properties due to their many dimensions, but also quickly grow very large. Two central concepts used in calculations with tensors are matricization and the n-mode product, which will be discussed in subsubsection 2.2.1.

The concept of tensor decomposition or tensor factorisation concerns itself with exploiting the tendencies of the data in order to find a representation that is smaller and easier to work with. There are multiple representations or decompositions that will be discussed in detail later in this section. The Tucker method will be discussed in a bit more detail, due to its importance to the work done in the thesis. Since the python library *TensorLy* has great implementations of the different algorithms, it will be used to calculate the decompositions throughout the thesis. The algorithms for estimating the decomposition are given in Appendix A.

2.2.1 Basic tensor representation and operations

2.2.1.1 Tensor matricization

It is often useful to reshape a tensor of order > 2 into a matrix. This is for instance in order to calculate the product of the tensor and a matrix. Tensors can be matricized with respect to one of its dimensions, i.e. using the nth dimension, the matricization of a tensor $\mathcal{X}$ becomes:

$$\mathcal{X}^{I_1 \times I_2 \times \dots I_N} \xrightarrow[matricization]{} \boldsymbol{X}_{(n)}^{I_n \times I_1 \cdot I_2 \cdots I_{n-1} \cdot I_{n+1} \cdots I_N} \tag{11}$$

Which means that the first dimension of the resulting matrix is equal to the nth dimension of $\mathcal{X}$ while the second dimension will be equal to the product of the remaining dimensions. Going from a matrix to a tensor is called un-matricization or tensorizing and is given by:

$$\boldsymbol{X}_{(n)}^{I_n \times I_1 \cdot I_2 \cdots I_{n-1} \cdot I_{n+1} \cdots I_N} \xrightarrow[un-matricization]{} \mathcal{X}^{I_1 \times I_2 \times \dots I_N} \tag{12}$$

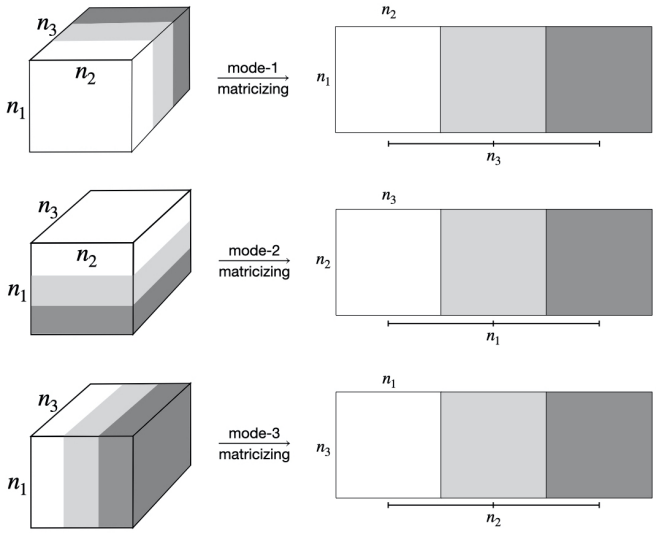


Figure 2.6: Matricization of an order-3 tensor using each of the three modes. n_i is the size of the ith dimension.

Notice that the original (or desired) shape of $\mathcal{X}$ must be know in order to tensorize a matrix. In Figure 2.6 matricization of an order-3 matrix using each of the modes is illustrated.

2.2.1.2 The n-mode product

The n-mode product is multiplying a a matrix onto a tensor resulting in a tensor. It is actually a three-step process of matricizing the tensor, multiplying with the matrix, and tensorizing the result to approximately the same shape. The n-mode product between an order N tensor $\mathcal{X}$ of size $I_1 \times I_2 \times \cdots \times I_N$ and a matrix $\boldsymbol{M}$ of size $J \times I_n$ is given by:

$$\mathcal{X}^{I_1 \times I_2 \times \cdots \times I_n \times \cdots \times I_N} \times_n \boldsymbol{M}^{J \times I_n} = \mathcal{Z}^{I_1 \times I_2 \times \cdots \times I_{n-1} \times J \times I_{n+1} \times \cdots \times I_N} \tag{13}$$

Which corresponds to tensorizing or un-matricizing the matrix-matrix product:

$$\boldsymbol{Z}_{(n)} = \boldsymbol{M}\boldsymbol{X}_{(n)} \tag{14}$$

Where the un-matricization of $\boldsymbol{Z}$ needs to have the same shape as $\mathcal{X}$ except J instead of I_n.

2.2.2 CP / Parafac

CP (Candecomp) decomposition that also goes by the names of Parafac or canonical decomposition is the most basic representation. The CP decomposition represents the tensor as the sum of R outer-vector products, where each dimension has a loading vector for each R. R is the rank of the decomposition and obviously this allows for more specific details the higher R gets. The CP decomposition of an order-3 tensor $\mathcal{X}$ of size $n_1 \times n_2 \times n_3$ is given by:

$$\mathcal{X} \approx \sum_{d=1}^{R} \boldsymbol{a}_d \circ \boldsymbol{b}_d \circ \boldsymbol{c}_d \tag{15}$$

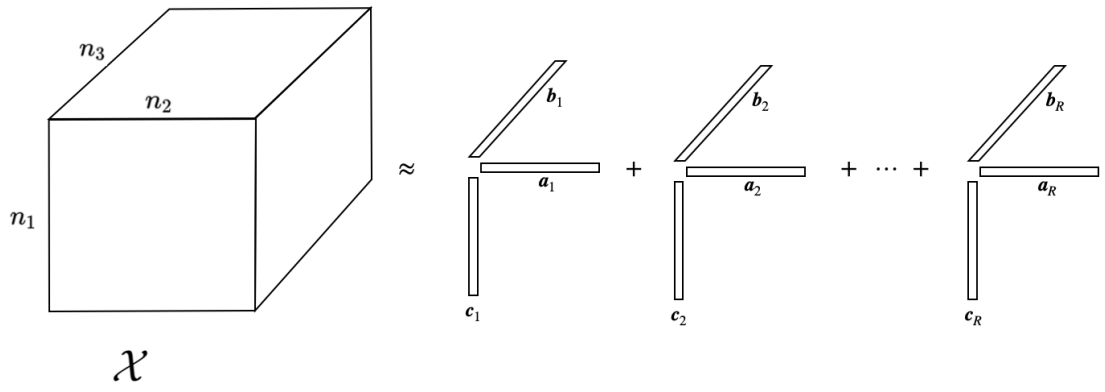


Figure 2.7: The tensor $\mathcal{X}$ is decomposed using the CP decomposition. The CP decomposition approximation is the sum of a sequence of R outer-vector products, with R being the rank of the decomposition. The vectors have length corresponding to the length of the dimension in $\mathcal{X}$, e.g. all the $\boldsymbol{a}$-vectors have length n_1.

Or element-wise:

$$\mathcal{X}(i,j,l) \approx \sum_{d=1}^{R} \boldsymbol{a}_d(i) \cdot \boldsymbol{b}_d(j) \cdot \boldsymbol{c}_d(l) \tag{16}$$

Where $\boldsymbol{a}_i$, $\boldsymbol{b}_i$, and $\boldsymbol{c}_i$ are the ith loading vectors of each dimension respectively. If each of these loading vectors are concatenated to form loading matrices, the decomposition can also be written in the following way using a diagonal or identity tensor $\mathcal{D}$ and n-mode products:

$$\mathcal{X}^{n_1 \times n_2 \times n_3} \approx \mathcal{D}^{R \times R \times R} \times_1 \boldsymbol{A}^{n_1 \times R} \times_2 \boldsymbol{B}^{n_2 \times R} \times_3 \boldsymbol{C}^{n_3 \times R} \tag{17}$$

$\mathcal{D}$ is also called the core tensor. For the loading matrices we have that:

$$\boldsymbol{A} = \begin{bmatrix} | & | & & | \\ \boldsymbol{a}_1 & \boldsymbol{a}_2 & \dots & \boldsymbol{a}_R \\ | & | & & | \end{bmatrix} \qquad \boldsymbol{B} = \begin{bmatrix} | & | & & | \\ \boldsymbol{b}_1 & \boldsymbol{b}_2 & \dots & \boldsymbol{b}_R \\ | & | & & | \end{bmatrix} \qquad \boldsymbol{C} = \begin{bmatrix} | & | & & | \\ \boldsymbol{c}_1 & \boldsymbol{c}_2 & \dots & \boldsymbol{c}_R \\ | & | & & | \end{bmatrix} \tag{18}$$

The CP decomposition is illustrated in Figure 2.7.

Due to the nature of the CP decomposition only the ith loading of each loading vector can interact with each other, making the CP decomposition rather restricted. However the restriction of this representation implies a unique optimal solution of the CP decomposition, which desirable. The method is good at resolving different additive physical profiles, e.g. if addition of multiple chemicals to a solution have different effects on a response variable, these can be resolved. CP decomposition however fails when different dimensions are correlated, due to strong cancellation effects between the various components of the representation, making it slow to converge and hard to interpret [15].

2.2.3 Tucker

Tucker decomposition is the less restricted version of the CP decomposition - or actually CP decomposition is a special case of Tucker. In Tucker decomposition loadings can interact with each

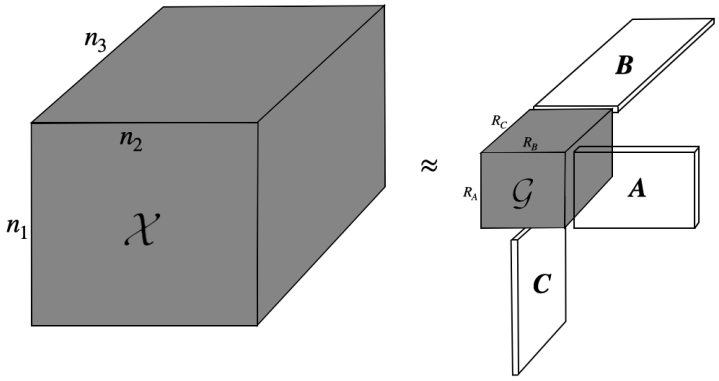


Figure 2.8: The tensor $\mathcal{X}$ is decomposed using Tucker-3 decomposition, which means that all 3 dimensions will be decomposed. The Tucker decomposition is a sequence of n-mode products between the core tensor $\mathcal{G}$ and the three loading matrices $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$. The size of $\mathcal{G}$ is equal to the ranks of the decomposition, i.e. $R_A \times R_B \times R_C$. The length of the first dimension of the loading matrices correspond to the length of the given dimension, e.g. for $\boldsymbol{A}$ it is n_1

other even crisscrossing, which gives this representation a good deal of freedom. This is achieved allowing the core-tensor $\mathcal{D}$ ($\mathcal{G}$ for Tucker) to be non-diagonal, hence we also get to choose special ranks for each dimension. Thus the Tucker decomposition is a sum of outer-vector products scaled by the core element. Based on the matrix-representation of the CP decomposition (17), the Tucker decomposition of an order-3 tensor $\mathcal{X}$ of size $n_1 \times n_2 \times n_3$ is given as:

$$\mathcal{X}^{n_1 \times n_2 \times n_3} \approx \mathcal{G}^{R_A \times R_B \times R_C} \times_1 \boldsymbol{A}^{n_1 \times R_A} \times_2 \boldsymbol{B}^{n_2 \times R_B} \times_3 \boldsymbol{C}^{n_3 \times R_C} \tag{19}$$

Where $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$ are the loading matrices of each of the dimensions and R_A, R_B, and R_C are the ranks of the decomposition in each of the dimensions. $\mathcal{G}$ is the core tensor that we in the Tucker case allow to be non-diagonal. We can also write the Tucker representation element-wise:

$$\mathcal{X}(x_1, x_2, x_3) \approx \sum_{r_A=1}^{R_A} \sum_{r_B=1}^{R_B} \sum_{r_C=1}^{R_C} \mathcal{G}(r_A, r_B, r_C) \cdot \boldsymbol{A}(x_1, r_A) \cdot \boldsymbol{B}(x_2, r_B) \cdot \boldsymbol{C}(x_3, r_C) \tag{20}$$

The concept of Tucker decomposition is shown in Figure 2.8. The different rank along each of the dimensions, makes the user able to choose how many different features the algorithm should find along each dimensions. The flexibility of the Tucker decomposition makes it useful for many applications, one of them being the data compression task. The flexibility however also comes at a cost; the representation does not yield unique optimal solutions, and the results harder to interpret than for CP decomposition.

Rank selection for the Tucker decomposition is either choosing appropriate ranks for the problem, or to find the minimal set of ranks (R_A, R_B, R_C) that ensure that:

$$\mathcal{X} = \sum_{r_A=1}^{R_A} \sum_{r_B=1}^{R_B} \sum_{r_B=1}^{R_B} \mathcal{G}(r_A, r_B, r_C) \cdot \boldsymbol{a}_{r_A} \circ \boldsymbol{b}_{r_B} \circ \boldsymbol{c}_{r_C} \tag{21}$$

I.e. the approximation being equal to the actual tensor. Here $\boldsymbol{a}_i$, $\boldsymbol{b}_i$, and $\boldsymbol{c}_i$ are the ith column of the loading matrices respectively. Finding the optimal ranks is a challenging problem, which have

been solved in different ways. Due to the challenge of the problem, this thesis will make use of the analytical solution to variational Bayesian matrix factorization (VBMF) by inspiration from Kim et. al [10]. This approach is chosen because it is a good and easily reproducible heuristic, even though it is sub-optimal. VBMF will be discussed in the following.

2.2.3.1 Variational Bayesian Matrix Factorization

The global analytical solution to the VBMF have been derived by Nakajima et. al in 2013 [16], and can be used as a method for rank selection for the Tucker model[3]. The authors use a probabilistic model to find the optimal low-rank factorization of a matrix, automatically taking care of the noise. Given an matrix $\boldsymbol{V}$ of size $L \times M$ that is assumed to be the sum of a target matrix and an error matrix:

$$\boldsymbol{V}^{L\times M} = \boldsymbol{U}^{L\times M} + \boldsymbol{E}^{L\times M} \tag{22}$$

The goal is to find matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ such that:

$$\boldsymbol{U} = \boldsymbol{B}\boldsymbol{A}^{\top} \tag{23}$$

This is achieved by considering a probabilistic model of $\boldsymbol{V}$ given prior variances, and use this model to compute the posterior distributions of $\boldsymbol{V}$, $\boldsymbol{A}$, and $\boldsymbol{B}$. Doing this is generally a non-convex problem why the analytical solution is key. This however does only work for matrices, hence tensors of order > 2 need to be converted into matrices using matricization discussed in subsubsection 2.2.1. Matricization using the ith dimension gives the estimated rank of that same dimension of the tensor. It should be noted that this method only works when the $L < M$, since we need to have $\frac{L}{M} < 1$ to satisfy the nature of the probabilistic distribution.

2.2.4 Block-Term Decomposition

Block-Term Decomposition (BTD) is simply a sum of Tucker terms, i.e. given an order-3 tensor $\mathcal{X}$ and a rank R the BTD of $\mathcal{X}$ is given by:

$$\mathcal{X} = \sum_{r=1}^{R} \mathcal{G}_r \times_1 \boldsymbol{A}_r \times_2 \boldsymbol{B}_r \times_3 \boldsymbol{C}_r \tag{24}$$

Where we now have R sets of a core and loading matrices instead of a single one of each, which also means that the ranks of the individual Tucker terms can differ to comply with various needs.

2.2.5 Tensor-Train decomposition

Tensor-Train (TT) decomposition that also goes by the name "matrix product state decomposition", was developed by Oseledets in 2011, as a simple and non-recursive tensor decomposition representation.[18] TT-decomposition does not make use of a core tensor, but allows for all the loadings arrays to be order-3 tensors. Given an order-d tensor $\mathcal{X}$ of size $n_1 \times n_2 \times \cdots \times n_d$, we will have a sequence of ranks $r_0, r_1, \ldots, r_d$, with boundary conditions $r_0 = r_d = 1$. The TT-decomposition of $\mathcal{X}$ is given by:

$$\mathcal{X}(i_1, i_2, \ldots, i_d) \approx \mathcal{G}_1[i_1]\mathcal{G}_2[i_2]\cdots\mathcal{G}_d[i_d] \tag{25}$$

[3]Also applicable to CP-decomposition, since it is a special case of Tucker decomposition

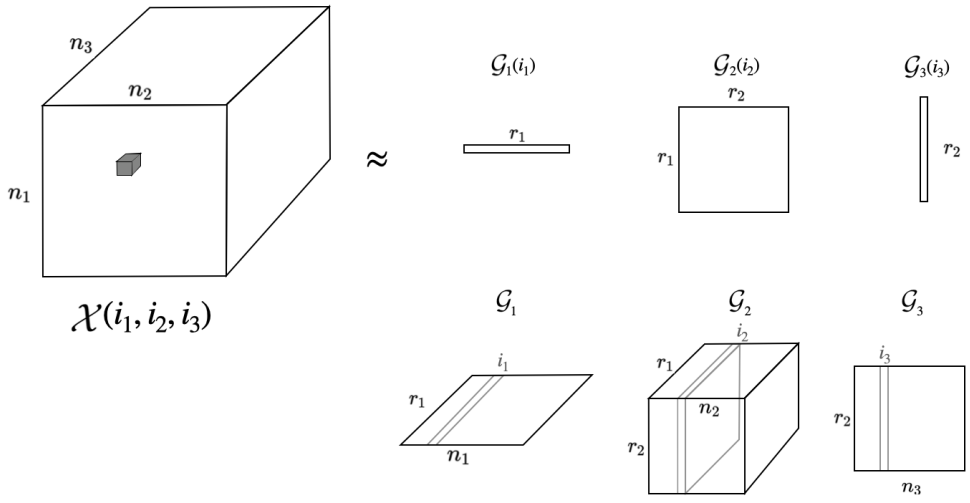


Figure 2.9: The approximation of a single value of the target tensor using the TT-decomposition. The value is the result of a vector-matrix-vector product between $\mathcal{G}_1(i_1)$, $\mathcal{G}_2(i_2)$, and $\mathcal{G}_3(i_4)$ that are slices of the loading tensors $\mathcal{G}$. The bottom row shows how the vectors and the matrix are taken out of the $\mathcal{G}$s using the given indices i_1, i_2, and i_3.

Where $\mathcal{G}_i$ is the ith loading tensor of size $r_{i-1} \times n_1 \times r_i$, and $\mathcal{G}_i[j]$ is the jth lateral slice of $\mathcal{G}$, which makes it a matrix of size $r_{i-1} \times r_i$. The calculation of a single value at position $(i_1, i_2, \ldots, i_d)$ is illustrated in Figure 2.9.

The TT-decomposition is not prone to the curse of dimensionality, it is stable, and has asymptotically the same number of parameters as CP-decomposition. Oseledets have in [18] also derived efficient versions of basic tensor operations in the TT-format.

3 Previous Work

In this section a recap of some different approaches to speed up or decrease the number of parameters will be given. In 2013 Denil et al. found NNs to be heavily over-parametrized [3]. They did this by training only a fraction of the weights and using these to predict the rest of the weights. They reported that in some cases more than 95% of the weights could be predicted, hence would not need to be learned. Ever since, numerous attempts have been proposed to optimise the evaluation time and number of parameters in NNs. One approach have been to decompose the dense layer of a NN, an other have tried speeding up the evaluation of a convolution, and some have had success in decomposing and speeding up an entire network. All the work described below follow approximately the same structure:

1. **Decompose weights of pre-trained NN** using some specific algorithm to get a smaller representation of the weights or filters
2. **Changing the evaluation algorithm** in order for it to correspond to original evaluations and in order to do back-propagation
3. **Fine-tuning** which is training the network using the new algorithm and back-propagation

Some approaches will be discussed in the following. In the end the methods will be compared in terms of performance in relation to the number of parameters and the computation time.

3.1 Decomposing the dense layer

In 2015 Nokinov et al. used TT-decomposition to decompose the weights of a dense (fully-connected) layer in an ANN[17]. This method exploits the power of the dense layer and allows it to have a big amount of neurons. The dense layer performs a linear transformation on an input vector $\boldsymbol{x}$ of dimension N:

$$\boldsymbol{y} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b} \tag{26}$$

Where $\boldsymbol{W} \in \mathbb{R}^{M \times N}$ is the weight matrix and $\boldsymbol{b} \in \mathbb{R}^M$ is the bias vector. M is the dimension of the output $\boldsymbol{y}$. Since the weights of the dense layer are given in a matrix, Nokinov et al. folds this matrix into a 3-dimensional tensor in order to decompose it using the TT algorithm. They define the TT-layer to be the same transformation but with the weights stored in the TT-format. Let $\mathcal{X}$ be the input tensor of dimension d (formed from $\boldsymbol{x}$), and let the decomposition of the weight matrix have cores $\boldsymbol{G}_k[i_k, j_k]$. In the TT-layer the transformation corresponding to (26) is given as:

$$\mathcal{Y}(i_1, i_2, \ldots, i_d) = \sum_{j_1, j_2, \ldots, j_d} \boldsymbol{G}_1[i_1, j_1]\boldsymbol{G}_2[i_2, j_2] \ldots \boldsymbol{G}_d[i_d, j_d]\mathcal{X}(j_1, j_2, \ldots j_d) + \mathcal{B}(i_1, i_2, \ldots, i_d) \tag{27}$$

Where $\mathcal{B}$ is the bias tensor. The properties of the TT-format allows for computation of back-propagation, which is then used for fine-tuning the network. The authors report a best-case compression of a dense layer of up to 200,000 times and of a whole network of up to 7.

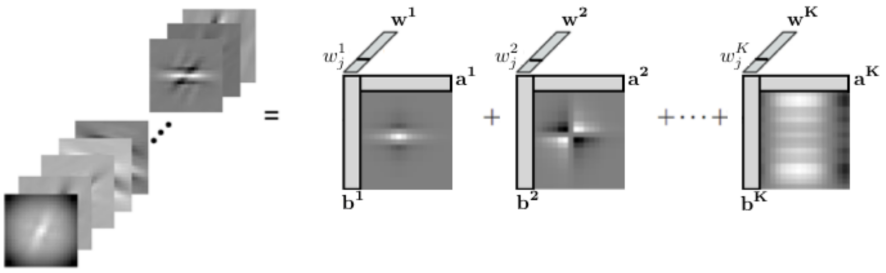


Figure 3.1: Taken from [22]. Left: A bank of two-dimensional filters is stacked together to form a 3-dimensional tensor. Right: The tensor is decomposed in the sum of K rank-one tensors. Thus, the original filters are approximated by the weighted sum of the separable filters $s_k = \boldsymbol{a}_k \circ \boldsymbol{b}_k$

3.2 Speeding up the evaluation of a convolution

In 2013 Rigamonti et al. [20] found that filters can be computed as a linear combination of smaller separable filters. A filter is called separable when it can be expressed using multiple parts of lower dimension, as for instance using the CP-decomposition as illustrated by [22] in Figure 3.1. Ever since many approaches have been considered using this principle in different ways. For instance Jaderberg et al. proposed two different schemes for using rank-1 filters to exploit the cross-channel / filter redundancy [7]. Since these schemes did not built upon any decomposition algorithm per se, the approximation of the separated filters was done by numerically minimizing either the L_2 reconstruction error or the reconstruction error of the output of the given layer (indirectly). Two other approaches uses standard decomposition algorithms to ease the evaluation time. One was proposed by Lebedev et al. in 2015 and used CP-decomposition to decompose the convolutional kernel [13], another was proposed by Wang and Cheng in 2016 and used BTD. They will both be discussed in the following.

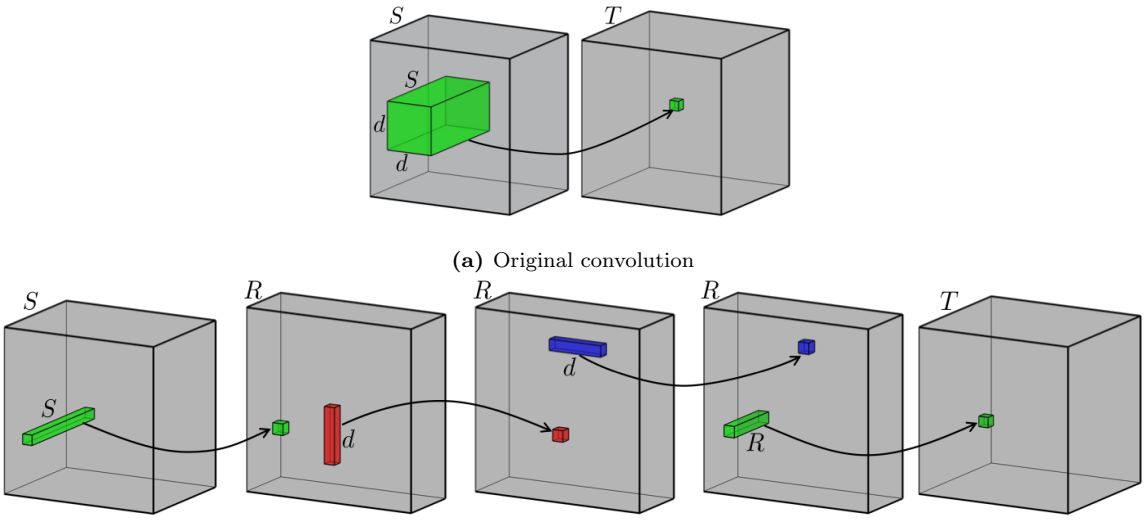
3.2.1 Convolution using CP-decomposition

Lebedev et al. proposed to decompose the 4-dimensional convolutional kernel using CP-decomposition [13]. The idea is to reduce the computation time of the evaluation of a convolutional layer. After having decomposed the kernel into a set of rank-1 tensors, they re-write the convolution into an expression of nested sums using the loadings of the decomposition. The traditional convolution of an input tensor $\mathcal{U}$ of size $W \times H \times S$ into an output tensor $\mathcal{T}$ of size $(W-d+1) \times (H-d+1) \times t$ looks like this:

$$\mathcal{V}(x,y,t) = \sum_{i=x-\delta}^{x+\delta} \sum_{j=y-\delta}^{y+\delta} \sum_{s=1}^{S} \mathcal{K}(i-x+\delta, j-y+\delta, s, t)\, \mathcal{U}(i,j,s) \tag{28}$$

Where $\mathcal{K}$ is the 4-dimensional kernel[4] of size $d \times d \times s \times t$ and $\delta = \frac{d-1}{2}$ is the half width of the filter in the spatial dimension. Using the decomposition of $\mathcal{K}$ with loading matrices $\boldsymbol{K}_w$, $\boldsymbol{K}_h$, $\boldsymbol{K}_s$, and

[4]The kernel corresponds to a stack of 3-dimensional filters



(a) Original convolution

(b) Convolution using CP-decomposed kernel

Figure 3.2: Taken from [13]. The difference between the original convolution and the sequence of convolutions using the loadings vectors of the CP-decomposition. The grey boxes corresponds to the 3-dimensional tensors within the CNN, with the frontal sides corresponding to the spatial dimension. For instance the first box in each line corresponds to the input image where S is the number of input channels. The arrows show linear mappings, or how to calculate a single value using a filter (top) or a decomposed filter (bottom).

$\boldsymbol{K}_t$, the (approximated) convolution (28) can be written as:

$$\mathcal{V}(x,y,t) = \sum_{r=1}^{R} \boldsymbol{K}_t(t,r) \left(\sum_{i=x-\delta}^{x+\delta} \boldsymbol{K}_w(i-x+\delta,r) \left(\sum_{j=y-\delta}^{y+\delta} \boldsymbol{K}_h(j-y+\delta,r) \left(\sum_{s=1}^{S} \boldsymbol{K}_s(s,r)\, \mathcal{U}(i,j,s) \right) \right) \right) \tag{29}$$

With R being the rank of the decomposition. Now each of the parenthesis can be expressed as intermediate tensors, which means that the whole convolution can be broken into a sequence of four simple convolutions. Figure 3.2 shows the difference between a traditional convolution and the sequence of convolutions using the decomposition. The expression of the convolution using the decomposed kernel allows as a sequence of simpler convolutions allows for the use of standard back-propagation. Furthermore it enables easy fine-tuning of the entire network. The authors report that even though the reconstruction error decreases with the increasing rank, a high accuracy is not needed in order to get good prediction results. They also state that their method works well for smaller architectures, while it has a harder time on more complex ones.

3.2.2 Convolution using Block Term Decomposition

Wang and Cheng proposed speeding up the evaluation of a convolution using BTD, which benefits from both sparsity and low-rank. For simplicity they combine the two spatial dimensions into one which means that the convolutional kernel becomes 3-dimensional ($S \times T \times WH$). Since they

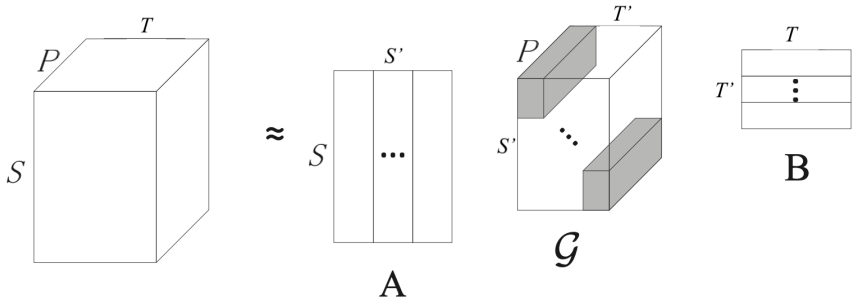


Figure 3.3: Taken from [24]. Illustrates how the R factor matrices and core tensors can be concatenated.

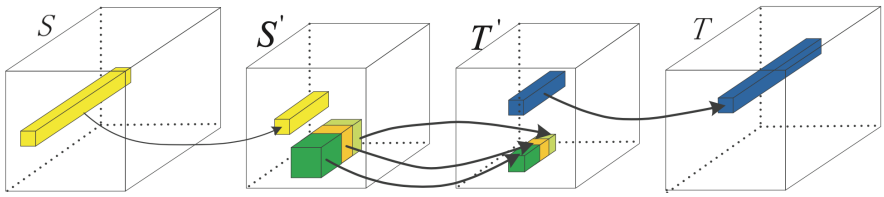


Figure 3.4: Taken from [24]. Illustrates how the convolution is done using the BTD. Each of the boxes correspond to an intermediate tensor with the frontal sides corresponding to spatial dimensions. The arrows show a linear mapping using the factors of the filter.

will not exploit the spatial dimension due to low rank, their decomposition becomes a tucker-2 decomposition of the kernel $\mathcal{K}$:

$$\mathcal{K} \approx \sum_{r=1}^{R} \mathcal{G}_r \times_1 \boldsymbol{A}_r \times_2 \boldsymbol{B}_r \tag{30}$$

Where R is the rank of the BTD, $\boldsymbol{A}_r \in \mathbb{R}^{S \times s}$ and $\boldsymbol{B}_r \in \mathbb{R}^{T \times t}$ are the factor matrices along input and output channel dimensions respectively. $\mathcal{G}_r \in \mathbb{R}^{s \times t \times P}$ is the core of the r'th subtensor. Now the authors concatenate all the R matrices into big matrices $\boldsymbol{A}$ and $\boldsymbol{B}$. And the core tensors into a big group-sparse core matrix $\mathcal{G}$. The result is visualized in Figure 3.3. The convolution can be done faster using this decomposition and the sequence of convolutions shown in Figure 3.4. The authors also propose a new algorithm for doing the decomposition itself. They call the algorithm Principle Component Iteration (PCI) which is a generalisation of higher order orthogonal iteration (HOOI)

3.3 Speeding up the entire network

All the methods discussed above only describe how to decompose a single layer in the NN; the dense layer or the convolutional layer respectively. In 2016 one method was proposed that would be able to decompose an entire network at once.

Kim et al. proposed using Tucker-decomposition to decompose, not only the convolutional layers, but also the dense layers [10]. They would use Tucker-2 decomposition from the second convolution

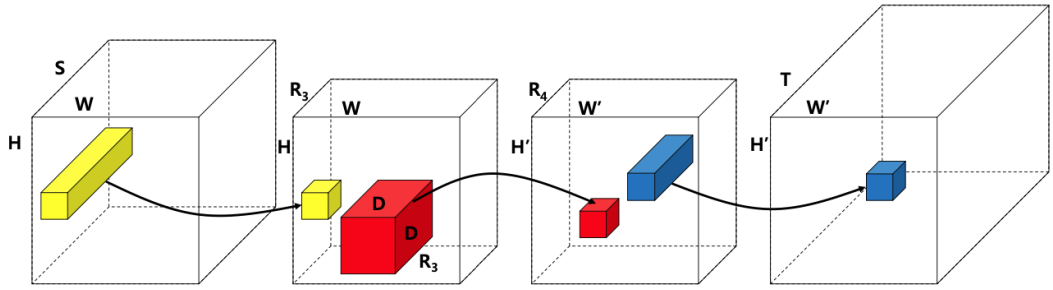


Figure 3.5: Taken from [10]. The boxes correspond to the 3-dimensional tensors in the network with the frontal sides being the spatial dimensions. Each of the arrows show a linear mappings. The two middle tensors are the intermediate tensors with R_3 and R_4 being the ranks of the decomposition.

to the first dense layer, and Tucker-1 decomposition[5] for the remaining. They did not decompose the spatial dimensions, since these already are quite small. They also proposed using the global analytical solutions for variational Bayesian matrix factorization (VBMF) developed by Nakajima et al. in 2013 [16] to find the optimal ranks for the decompositions. In a similar derivation to the one in subsubsection 3.2.1, Kim et al. found that the Tucker-2 decomposed convolutional layer can be described as a sequences of simpler convolutions. This sequence is shown in Figure 3.5. They use this sequence to do back-propagation on the entire network at once. The authors report promising results, but states that the optimality of the rank selection needs to be investigated further. They also claim that the 1×1 convolution has great potential even though it seems to be cache-inefficient, that it and will grow in popularity.[6]

3.4 Overall comparison of methods

It seems that the biggest compression happens in the dense layers, since these are very heavy in number of parameters, on the other hand the convolution takes longer since there are more evaluations per parameter. In trying to speed up the CNN, the method using BTD proposed in [24] seems to get the best results. On the VGG-16 network architecture[21] they achieve an acceleration of 6.6×, while the methods of Jaderberg et al. [7] and Kim et al. [10] achieve 3.8× and 3.3× respectively. The authors of the BTD method also discuss and test decomposing the first dense layer since this can be treated as a convolution with large filters. The method using CP-decomposition [13] worked well, but did not seem to be scaleable.

[5] Equivalent to the SVD

[6] The yellow and blue convolutions seen in Figure 3.5. It is already extensively used in for instance GoogLeNet [23]

4 Data

In this thesis two different data sets will be used; the well-studied MNIST data set [14] of handwritten digits, and the relatively new THETIS data set [4] of videos of tennis shots. The idea is to first apply the methods discussed later to the simpler MNIST data set, to make sure that it is working properly, followed by an application to the more complicated THETIS data set. Each of the data set will be described in the following.

4.1 MNIST - Handwritten digits

The MNIST (Modified National Institute of Standards and Technology) data set [14] consists of 70,000 pictures[7] of handwritten digits from 0 through 9. Each picture is 28×28 pixels and black/white which means that there is only one input channel. Each of the $28 \cdot 28 = 784$ pixel values are intensities ranging from 0 (black) to 255 (white). The data set is formed by remixing samples from the earlier NIST data set, which consisted of both digits and characters. The MNIST samples would be picked out, standardized and normalized so they would all be the same size and have the same gray-values [27]. The first 175 samples from the traning set of the MNIST database is shown in Figure 4.1.

Figure 4.1: The first 175 samples of the MNIST data set of handwritten digits

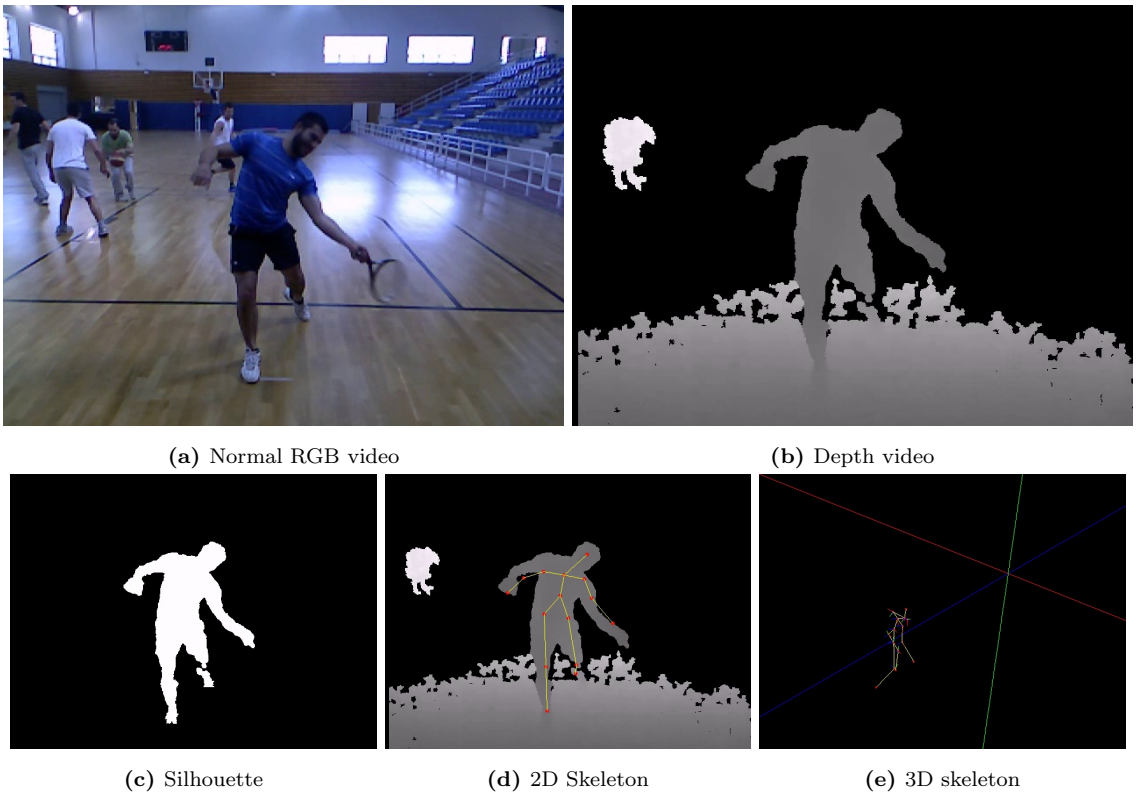
4.2 THETIS - Tennis shot videos

The THETIS (Three dimEnsional TennIs Shots) data set consists of videos of 1980 tennis shots performed by both beginners and more experienced individuals [4]. Each of the 12 types of shots types have been performed multiple times by each of the 55 individuals (31 beginners and 24 experts). Each shot consists of an RGB video, a depth video, a silhouette video, and both 2D and 3D skeleton videos. The videos have a resolution of 480×640, while they vary in length (approximately 3-7 s). Since the skeleton extraction is not always successful, only 1217 of the shots have this feature. This means that the total number of videos in the data set is 8,374 resulting in a total playtime of 7 hours and 15 minutes. An overview of the 12 types of shots is given in Table 1, while an example of each of the five features is given in Figure 4.2.

[7] The 70,000 is split into 60,000 training pictures and 10,000 testing pictures

Table 1: Overview of the 12 different types of tennis shots present in the THETIS data set.

Forehand	**Backhand**	**Service**	**Other**
Flat	With 2 hands	Flat	Smash
Slice	Slice	Slice	
Volley	Volley	Kick	
Open stands	Backhand		



(a) Normal RGB video **(b)** Depth video

(c) Silhouette **(d)** 2D Skeleton **(e)** 3D skeleton

Figure 4.2: Example of the different videos that are given for each shot in the data set. Each of the pictures show the same frame in each of the videos of the same shot. Note that the 3D skeleton looks different since this is shown from a different angle. The 3D skeleton joint positions are also given numerically for each frame in the whole dataset.

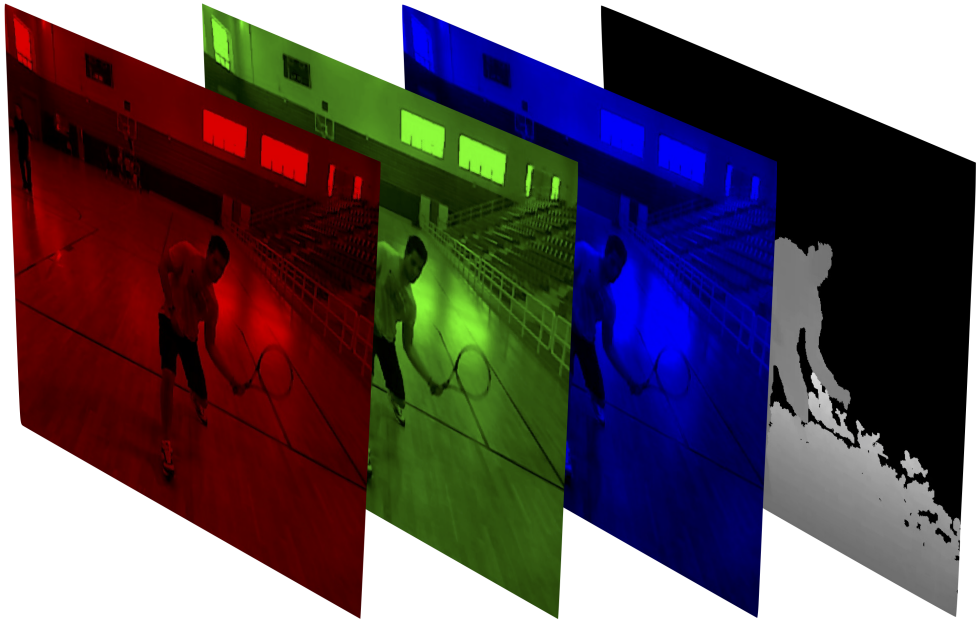


Figure 4.3: Each frame of the videos consists of 4 input channels; one red, one green, one blue, and one depth channel.

Even though this data set is quite simple in relation to the real world, it is still complicated, especially in relation to the MNIST data set. The videos feature different individuals - both men and women, right handed and left handed, professionals and beginners. Even though the same type of shot is being performed by two individuals, they can look very different from each other. Also some of the videos are recorded in a sports arena with people doing different activities in the background as for instance yoga or basketball, while the others are recorded in a changing room with no noise at all.

4.2.1 Pre-processing for THETIS

In order to properly train on the THETIS data set, some preprocessing is needed. It is desirable to have the videos scaled down both in space and time, and to use more than just the RGB video from the data set. The depth video, which shows the depth information of the video on a gray-scale, only has one channel, and was easy to concatenate onto the RGB videos. This make the data set now have 4 input channels, which means more information to train on. The 4 channels of a single frame in the data set is illustrated in Figure 4.3.

One problems is that all the videos input into the network needs to have the same size, hence also the same length (number of frames). How this is typically ensured is by splitting the video into smaller pieces of the same length, and then running the algorithm on all the pieces and letting the majority of the smaller videos determine the class.[8] While this is appropriate when trying to classify a general activity like running, swimming etc., it does not seem appropriate for this problem. This is due to the tennis shots being performed once per video, making the whole sequence a part

of the shot. The problem was solved by manually inspecting the videos and assigning the time for the approximate midway through the shot. This assigned time is then used to extract the same number of frames on either side of the midway, making all videos have the same number of frames. This also has a standardizing effect, due to the middle of the shot being at the approximate same time in all videos. The number of frames is experimentally set to 14, making all videos 28 frames.

The last thing that was done to the data, was to decrease the resolution by a factor of 4 in each spatial dimension, i.e. from a resolution of 480×640 to a resolution of 120×160. This results in the number of pixels decreasing by a factor of 16 from 307,200 to 19,200, while still having a relatively high resolution that should be fine to train on.

All of the above results in the THETIS observations having the size

$$(S \times F \times H \times W) = 4 \times 28 \times 120 \times 160 \tag{31}$$

Where S is the input channels, F is the number of frames, H is the height in pixels, and W is the width in pixels.

5 Methodology

Overall the methods examined in this thesis can be divided into two parts; namely those trying to decompose the input into a NN, and those decomposing the weights of a pre-trained NN with subsequent fine-tuning. In the following, the methods used within each of these parts will be discussed in detail. The Python library *PyTorch*[19] will be used to implement and train all NN architectures, while all decomposition estimation will be executed using the Python library *TensorLy*[12]. PyTorch provides a very popular application programming interface (API) for machine learning and tensor computing, which is easily accelerated using the graphics processing unit (GPU). TensorLy provides an API for tensor methods including the different decomposition methods described in subsection 2.2. TensorLy is also compatible with different back-ends[8], making it easy to combine with PyTorch. The algorithms used for estimating the decompositions in *TensorLy* are given in Appendix A.

After both the above mentioned methods have been discussed in detail, an overview of the architectures used in this thesis will be given, along with how to calculate the number of FLOPs used in different layers in a NN.

5.1 Method 1 - Decomposing the Input

In reality, tampering with data that is going to be learned is data preprocessing, hence decomposing the input is just a form of this. Data preprocessing is a vital part of almost all ML, because structured data of high quality is simply much easier to learn and to work with. Put in another way: "Garbage in, garbage out". In addition knowledge of the structure and the variance of the data often ease the algorithm development process. The decomposition algorithms seems appropriate for this task due to their ability to exploit the structure of the data.

The Tucker decomposition is believed to be suitable for data compression tasks [15], which is why it seems natural to apply this method to the data before feeding it in to and training the NN. The purpose is to let the decomposition do some of the learning beforehand by exploiting the variance in the data. The Tucker-decomposition is flexible since it allows for decomposition of only a subset of the dimensions[9], and because it allows for specification of ranks for each dimension individually. Since the ranks correspond to the degree of flexibility, it is easy to choose how flexible the decomposition should be along each dimension. The rank would increase with the level of desired detail. This means that if the goal of the algorithm is to differentiate a small set of features, the rank would also be kept small along the relevant dimension.

Since the observations in each the given data sets can be stacked to form a 4 (MNIST) or 5 (THETIS) dimensional tensor, it is fairly straight forward to apply the Tucker decomposition. The algorithm discussed below will only cover the MNIST data set, since the expansion to videos is trivial (one more dimension). For N MNIST observations, the stacked tensor $\mathcal{X}$ of size $(N \times 1 \times 28 \times 28)$

[8] The back-end is the Python library that is responsible for low-level calculation and specification of the tensors. Typically *NumPy*[5] or *PyTorch*[19]

[9] This is also called a *partial* Tucker decomposition

will be reshaped to $(N \times 28 \times 28)$ due to the single input channel. The decomposition of the now 3rd order tensor becomes:

$$\mathcal{X}^{N\times 28\times 28} \approx \mathcal{G}^{L\times J\times K} \times_1 \boldsymbol{A}^{N\times L} \times_2 \boldsymbol{B}^{28\times J} \times_3 \boldsymbol{C}^{28\times K} \tag{32}$$

Where L, J and K corresponds to the ranks in each of the dimensions. Here $\mathcal{G}$ is the core of the decomposition, $\boldsymbol{A}$ corresponds to the loading matrix along the between-sample dimension, while $\boldsymbol{B}$ and $\boldsymbol{C}$ correspond to the loading matrices in each of the spatial dimensions. Since the goal is to learn the differences between the different classes (digits) in the data set, only the between-sample dimension will be decomposed. The so-called Tucker-1 decomposition of the first dimension is given by:

$$\mathcal{X}^{N\times 28\times 28} \approx \mathcal{G}^{L\times 28\times 28} \times_1 \boldsymbol{A}^{N\times L} \times_2 \boldsymbol{I}^{28\times 28} \times_3 \boldsymbol{I}^{28\times 28} \tag{33}$$

$$\approx \mathcal{G}^{L\times 28\times 28} \times_1 \boldsymbol{A}^{N\times L} \tag{34}$$

This is equivalent to SVD using the unfolded representation with respect to the first dimension:

$$\boldsymbol{X}_{(1)} = \boldsymbol{A}\boldsymbol{G}_{(1)} \tag{35}$$

Now the loading matrix $\boldsymbol{A}$ holds information about what differentiates the different samples using only L features. These features will be used as the input into a very simple ANN, potentially lowering the number of input nodes dramatically if L is kept small. The algorithm is illustrated in Figure 5.1.

In this project experiments will be made with different architectures for the simple ANN (i.e. number of hidden neurons in the hidden layer(s) H), along with different values of the rank L. The results will be provided and discussed in terms of the trade-off between computational speed and accuracy.

5.1.1 Estimating the loadings for new data

Since the testing data cannot be decomposed along with the training data, the loading matrix for the testing data will be estimated assuming the same core as for the training data. This way the decomposition is also a part of the training. Based on (35) the loading matrix for the test set can be estimated by multiplying with the pseudo-inverse of the matricized core:

$$\boldsymbol{A}_{test} \approx \boldsymbol{X}_{test(1)}\boldsymbol{G}_{(1)}^{\dagger} \tag{36}$$

The two matrices $\boldsymbol{A}$ and $\boldsymbol{A}_{test}$ will be given as the input to the ANN as the training data and the validation/testing data respectively.

5.2 Method 2 - Compressing a Pre-Trained Network

Using this approach is relatively easy to justify which is properly the reason why all the methods discussed in section 3 use this approach. Pre-training a network makes it easier to ensure a network that works well, also because the training is often performed on powerful computers which allows for more complex networks. After the training is concluded the network is only decomposed to

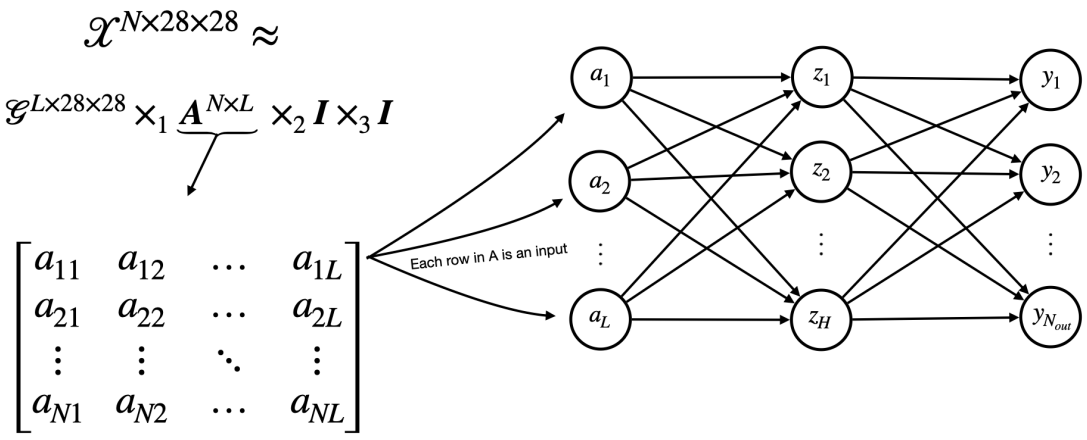


Figure 5.1: Illustration of the algorithm that decomposes the input into the NN. The stacked data tensor will be decomposed along the between-sample dimension resulting in a loading matrix $\boldsymbol{A}$ which holds L values for each observation, which will then be used as input into a simple ANN

optimise the evaluation time in order to make it useful on platforms with less computation power. As discussed in section 3 there are multiple approaches for this, but especially the methods using Tucker-decomposition and BTD seamed to be performing well and relatively straight-forward to implement and fine-tune. The method used in this project builds on the work done by Kim et al. in [10], which is briefly discussed in subsection 3.3. Similar to the work done by Kim et al., Tucker-1 or Tucker-2 will be used for specific layers through the network. Kim et al. proposed method for compressing image convolutions, why the methods will be generalized to videos in the following. The advantage of using Tucker-decompositions is that they can be applied to each of the layers in the network, compressing everything at once before fine-tuning the whole network. In the following the methods of decomposing and changing the architecture of the layers of the network will be discussed. Due to the dissimilarities of the linear vs. the convolutional layer, each of these decompositions will be implemented individually for each type of layer. In the end, the full algorithm for compressing a whole network will be given.

5.2.1 The linear / dense layer

As explained in subsection 2.1 the dense layer with N_{in} input neurons and N_{out} output neurons in simply a matrix-vector product:

$$\boldsymbol{y}^{N_{out}} = \boldsymbol{W}^{N_{out}\times N_{in}}\boldsymbol{x}^{N_i n} + \boldsymbol{b}^{N_{out}} \tag{37}$$

Table 2: The number of multiplications needed to do the matrix-vector product in the linear layer (37) with the weight matrix decomposed using Tucker-1 and Tucker-2. The matrix product is assumed to be calculated from right to left.

Method	**# Multiplications**	**Faster when:**
Uncrompressed	$N_{out}N_{in}$	-
Tucker-1	$R(N_{in} + N_{out})$	$R < \frac{N_{out}N_{in}}{N_{out}+N_{in}}$
Tucker-2	$N_{in}R_B + R_BR_A + R_AN_{out}$	$(N_{in} + R_A)(N_{out} + R_B) < 2N_{in}N_{out}$

Where $\boldsymbol{x}$ is the input, $\boldsymbol{y}$ is the input, $\boldsymbol{b}$ is the bias vector, and $\boldsymbol{W}$ is the weight matrix. The weight matrix is potentially big resulting in many parameters and high calculation time. The idea of the decomposed linear layer is to decompose the weight tensor into a set of smaller matrices that can be multiplied sequentially resulting in higher efficiency. The Tucker-1 and Tucker-2 decompositions of the weight matrix are given as:

$$\texttt{Tucker-1}: \quad \boldsymbol{W} \approx \boldsymbol{G} \times_1 \boldsymbol{A} \times_2 \boldsymbol{I} = \boldsymbol{AG} \qquad \text{OR} \qquad \boldsymbol{W} \approx \boldsymbol{G} \times_1 \boldsymbol{I} \times_2 \boldsymbol{B} = \boldsymbol{GB}^\top \tag{38}$$

$$\texttt{Tucker-2}: \quad \boldsymbol{W} \approx \boldsymbol{G} \times_1 \boldsymbol{A} \times_2 \boldsymbol{B} = \boldsymbol{AGB}^\top \tag{39}$$

Here $\boldsymbol{I}$ is the identity matrix and the core tensor $\boldsymbol{G}$ is a matrix due to the weight matrix $\boldsymbol{W}$ only having 2 dimensions. The type of Tucker-1 decomposition is chosen based on which dimension is to be decomposed.[10] Inserting the decomposition of $\boldsymbol{W}$ in the linear transform (37) yields:

$$\boldsymbol{y}^{N_{out}} \approx \boldsymbol{A}^{N_{out}\times R_A}\boldsymbol{G}^{R_A\times N_{in}}\boldsymbol{x}^{N_{in}} + \boldsymbol{b}^{N_{out}} \tag{40}$$

$\texttt{Tucker-1}:$ OR

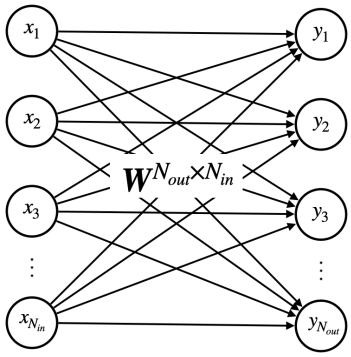
$$\boldsymbol{y}^{N_{out}} \approx \boldsymbol{G}^{N_{out}\times R_B}\boldsymbol{B}^{\top\ R_B\times N_{in}}\boldsymbol{x}^{N_{in}} + \boldsymbol{b}^{N_{out}} \tag{41}$$

$$\texttt{Tucker-2}: \quad \boldsymbol{y}^{N_{out}} \approx \boldsymbol{A}^{N_{out}\times R_A}\boldsymbol{G}^{R_A\times N_{in}}\boldsymbol{B}^{\top R_B\times N_{in}}\boldsymbol{x}^{N_{in}} + \boldsymbol{b}^{N_{out}} \tag{42}$$

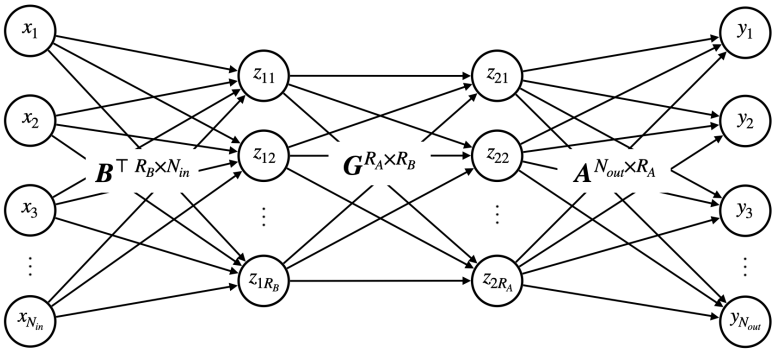
Now looking at the number of multiplications needed to do each of the products from right to left gives the values in Table 2. For instance using the Tucker-1 compression uses $N_{in}R + RN_{out} = R(N_{out} + N_{in})$ multiplications to do the matrix-product. The table also shows the restrictions on the ranks to have less multiplications than the uncompressed version, hence to be more efficient.

What also makes this decomposition smart is that the sequence of matrix multiplications can be achieved by a sequence of 2 or 3 linear layers, where the loadings from the decomposition can be directly used as weights. This makes it straight-forward to adapt the layer into a decomposed version and to fine-tune. The concept is illustrated in Figure 5.2. The biases from the original layer will be added only to the last layer in the sequence of the decomposed version and they will not be modified in any way.

[10] Dense layers often go from many inputs to a fewer outputs why the input dimension might be more desirable to compress.



(a) Original linear layer



(b) Sequence of linear layers resulting after decomposing $\boldsymbol{W}$

Figure 5.2: Illustration of the difference between the original linear layer and the decomposed sequence of linear layers resulting after applying a Tucker-2 decomposition. The matrices $\boldsymbol{A}, \boldsymbol{B}$ and $\boldsymbol{G}$ are the resulting loading matrices and core respectively after decomposing the original weight matrix $\boldsymbol{W}$. The biases from the original layer will be added only to the last layer in the decomposed sequence.

5.2.2 The convolutional layer

As explained in subsection 2.1 a convolution corresponds to calculating the result of applying a number of filters to every part of the input picture or video in space (and time). Applying the Tucker-decomposition to a convolutional layer implies turning it into a sequence of 3 (Tucker-2) or 2 (Tucker-1) smaller, less computational convolutions. The following method will only be discussed for videos, since the calculations are almost identical for lower dimensions. Kim et al. proposed a method for compressing an image filter in [10]. In the following their method will be generalized to videos based on their derivations.

5.2.2.1 Tucker-2 decomposition of the convolutional kernel

The convolution of an input tensor $\mathcal{X}$ of size[11] $F \times H \times W \times S$ into an output tensor $\mathcal{Y}$ of size $F' \times H' \times W' \times T$ is given by:

$$\mathcal{Y}(f', h', w', t) = \sum_{i=1}^{D_F} \sum_{j=1}^{D_H} \sum_{l=1}^{D_W} \sum_{s=1}^{S} \mathcal{K}(i, j, l, s, t)\ \mathcal{X}(f_i, h_j, w_l, s) \tag{43}$$

Where:

$$f_i = (f' - 1)\,\Delta_F + i - P_F \qquad h_j = (h' - 1)\,\Delta_H + j - P_H \qquad w_l = (w' - 1)\,\Delta_W + l - P_W \tag{44}$$

Where the Δs are the strides in each of the dimensions, the Ps are the corresponding padding, and (f', h', w', t) is the position in the output tensor $\mathcal{Y}$. $\mathcal{K}$ is the 5-dimensional convolutional kernel, i.e. the stack of 4-dimensional filters, and the Ds are the filter sizes in each of the spatial and temporal dimensions. Now the Tucker-2 decomposition of $\mathcal{K}$ in the input and output channel dimensions is expressed by:

$$\mathcal{K}(i, j, l, s, t) = \sum_{r_4=1}^{R_4} \sum_{r_5=1}^{R_5} \mathcal{C}(i, j, l, r_4, r_5)\ \boldsymbol{U}^{(4)}(s, r_4)\ \boldsymbol{U}^{(5)}(t, r_5) \tag{45}$$

Where $\mathcal{C}$ is the core of the decomposition of size $D_F \times D_H \times D_W \times S \times T$ and the $\boldsymbol{U}$s are the loading matrices along the input and output dimensions respectively. Now substituting this expression into (43) gives:

$$\mathcal{Y}(f', h', w', t) = \sum_{i=1}^{D_F} \sum_{j=1}^{D_H} \sum_{l=1}^{D_W} \sum_{s=1}^{S} \sum_{r_4=1}^{R_4} \sum_{r_5=1}^{R_5} \mathcal{C}(i, j, l, r_4, r_5)\ \boldsymbol{U}^{(4)}(s, r_4)\ \boldsymbol{U}^{(5)}(t, r_5)\ \mathcal{X}(f_i, h_j, w_l, s) \tag{46}$$

Rearranging, since not all factors depend on s, yields:

$$\mathcal{Y}(f', h', w', t) = \sum_{i=1}^{D_F} \sum_{j=1}^{D_H} \sum_{l=1}^{D_W} \sum_{r_4=1}^{R_4} \sum_{r_5=1}^{R_5} \mathcal{C}(i, j, l, r_4, r_5)\ \boldsymbol{U}^{(5)}(t, r_5) \underbrace{\sum_{s=1}^{S} \boldsymbol{U}^{(4)}(s, r_4)\ \mathcal{X}(f_i, h_j, w_l, s)}_{\mathcal{Q}} \tag{47}$$

Summing out s yields an intermediate tensor $\mathcal{Q}$ of size $F \times H \times W \times R_4$. Doing the same again:

$$\mathcal{Y}(f', h', w', t) = \sum_{i=1}^{D_F} \sum_{j=1}^{D_H} \sum_{l=1}^{D_W} \sum_{r_4=1}^{R_4} \sum_{r_5=1}^{R_5} \mathcal{C}(i, j, l, r_4, r_5)\ \boldsymbol{U}^{(5)}(t, r_5)\ \mathcal{Q}(f_i, h_j, w_l, r_4) \tag{48}$$

[11] Here and in the following F, H, W are the number of frames, the height and the width of the video respectively. S is the number of input channels.

Rearranging:

$$\mathcal{Y}(f', h', w', t) = \sum_{r_5=1}^{R_5} \boldsymbol{U}^{(5)}(t, r_5) \underbrace{\sum_{i=1}^{D_F} \sum_{j=1}^{D_H} \sum_{l=1}^{D_W} \sum_{r_4=1}^{R_4} \mathcal{C}(i, j, l, r_4, r_5) \mathcal{Q}(f_i, h_j, w_l, r_4)}_{\mathcal{Q}'} \tag{49}$$

Summing out everything but the output channels yields a new intermediate tensor $\mathcal{Q}'$ of size $F' \times H' \times W' \times R_5$, and the convolution can now be expressed in terms of $\mathcal{Q}'$:

$$\mathcal{Y}(f', h', w', t) = \sum_{r_5=1}^{R_5} \boldsymbol{U}^{(5)}(t, r_5)\ \mathcal{Q}'(f', h', w', r_5) \tag{50}$$

But now we have that the three intermediate tensors given by:

$$\mathcal{Q}(f, h, w, r_4) = \sum_{s=1}^{S} \boldsymbol{U}^{(4)}(s, r_4)\ \mathcal{X}(f, h, w, s) \tag{51}$$

$$\mathcal{Q}'(f', h', w', r_5) = \sum_{i=1}^{D_F} \sum_{j=1}^{D_H} \sum_{l=1}^{D_W} \sum_{r_4=1}^{R_4} \mathcal{C}(i, j, l, r_4, r_5) \mathcal{Q}(f_i, h_j, w_l, r_4) \tag{52}$$

$$\mathcal{Y}(f', h', w', t) = \sum_{r_5=1}^{R_5} \boldsymbol{U}^{(5)}(t, r_5)\ \mathcal{Q}'(f', h', w', r_5) \tag{53}$$

correspond to the following sequence of simpler convolutions:

- Eq. (51) - A 1×1 convolution with S input channels and R_4 output channels (the rank of the decomposition)
- Eq. (52) - A $D_F \times D_H \times D_W$ convolution with R_4 input channels and R_5 output channels
- Eq. (53) - A 1×1 convolution with R_5 input channels and T output channels

So what this sequence is actually doing is first using the 1×1 convolution to bring down the number of input channels to the 3D convolution, in order to perform the 3D convolution on a smaller set of input and output channels. In the end it uses the 1×1 convolution to bring the now lower number of output channels back up to what it should be. This all results in a speed-up if the ranks are chosen to be small. Due to the challenge of illustrating a 4D tensor, the concept is illustrated in the 2D version (a picture) in Figure 3.5. Like for the linear layer, the loadings of the decomposition of the kernel $\mathcal{K}$ can be used directly as weights in the adapted architecture, and also the original biases will only be added to the last layer, while the other won't have biases.

5.2.2.2 Tucker-1 decomposition of the convolutional kernel

In some situations it might not be beneficial to decompose both the input and the output channel dimensions, e.g. for an input image with only 3 input channels. The approach when doing the Tucker-1 decomposition is the same as for the Tucker-2 decomposition, however only one (relevant) rank is considered. This means that the resulting sequence have two smaller convolutions, and that there will be only two intermediate tensors corresponding to (51) and (52) if the input channel

Table 3: Number of multiplications used for a forward push for a 3D convolution using different algorithms and how the ranks should be chosen so the compressions are faster. Λ and Γ are defined in (55).

Method	**# Multiplications**	**Faster when**
Original	$S \cdot T \cdot \Gamma \cdot \Lambda$	-
Tucker-1 (in)	$R \cdot \Gamma \left(S + T \cdot \Lambda\right)$	$R < \frac{S \cdot T \cdot \Lambda}{S + T \cdot \Lambda}$
Tucker-1 (out)	$R \cdot \Gamma \left(S \cdot \Lambda + T\right)$	$R < \frac{S \cdot T \cdot \Lambda}{S \cdot \Lambda + T}$
Tucker-2	$\Gamma \left(S \cdot R_1 + R_1 \cdot R_2 \cdot \Lambda + R_2 \cdot T\right)$	$S \cdot R_1 + R_1 \cdot R_2 \cdot \Lambda + R_2 \cdot T < S \cdot T \cdot \Lambda$

dimension is to be decomposed, or (52) and (53) for the output channel dimension. The derivations will be given for each of the dimensions in Appendix B.

In order to choose the ranks so that the computation is faster, the number of multiplications needed for a forward push is now considered. We assume padding, i.e. same input and output sizes, since the difference between these is small - especially for small kernel sizes. The original convolution with S input channels, T output channels, a convolutional kernel with size $D_F \times D_H \times D_W$, and input/output size of $F \times H \times W$ uses:

$$S \cdot T \cdot F \cdot H \cdot W \cdot D_F \cdot D_H \cdot D_W \tag{54}$$

multiplications which quickly becomes a big number. We define:

$$\Gamma = F \cdot H \cdot W \qquad \Lambda = D_F \cdot D_H \cdot D_W \tag{55}$$

To be used in the following such that the original convolution uses $S \cdot T \cdot \Gamma \cdot \Lambda$. The Tucker-1 decomposition of the convolution with rank R uses:

$$S \cdot R \cdot \Gamma \cdot 1^3 + R \cdot T \cdot \Gamma \cdot \Lambda = R \cdot \Gamma \left(S + T \cdot \Lambda\right) \tag{56}$$

multiplications if the decomposition is carried out on the input channels, and

$$S \cdot R \cdot \Gamma \cdot \Lambda + R \cdot T \cdot \Gamma \cdot 1^3 = R \cdot \Gamma \cdot \left(S \cdot \Lambda + T\right) \tag{57}$$

multiplications on the output channels. For the convolution using the Tucker-2 decomposition with ranks R_1 and R_2, the forward push use:

$$S \cdot R_1 \cdot \Gamma \cdot 1^3 + R_1 \cdot R_2 \cdot \Gamma \cdot \Lambda + R_2 \cdot T \cdot \Gamma \cdot 1^3 = \Gamma \left(S \cdot R_1 + R_1 \cdot R_2 \cdot \Lambda + R_2 \cdot T\right) \tag{58}$$

multiplications. Table 3 shows how the ranks should be chosen in order for the compressions to have less multiplications than the original convolution, hence be faster.

5.2.3 Rank selection

There will be no all-round method for rank selection in this report. Where it is possible VBMF will be used to do the decomposition automatically as explained in paragraph 2.2.3.1. In the cases where VBMF fails the ranks will be set experimentally be the same as the rank in using the other dimension for which it works. This is mostly for the linear layers, where the automatic method fails when trying to find the rank in the input channel dimension. The implementation of the VBMF is taken from [2].

5.2.4 One-shot compression a an entire CNN using Tucker

Using the different algorithms for compressing a single layer in the CNN and appropriate ranks, the algorithm for compressing a whole CNN is given in Algorithm 1. The algorithm is basically training an appropriate network, decomposing all the layers given the methods above, and then fine-tuning the new decomposed network. This one-shot compression is applied when referring to method 2.

Algorithm 1 One-Shot Tucker Compression of a CNN

```
net ← define an appropriate CNN
train net using the given training data
net_dcmp ← make a copy of net
for each convolutional layer; layer in net_dcmp do
    K_layer ← take out weight tensor
    R_4, R_5 ← choose appropriate rank(s)
    G, U^(4), U^(5) ← Decompose K_layer using relevant algorithm        ▷ Or just one U
    layer_dcmp_1 ← define new 1 × 1 convolution                         ▷ If applicable
    layer_dcmp_2 ← define new 3D convolution
    layer_dcmp_3 ← define new 1 × 1 convolution                         ▷ If applicable
    K_layer_dcmp_1 ← U^(4)                                              ▷ If applicable
    K_layer_dcmp_2 ← G
    K_layer_dcmp_3 ← U^(5)                                              ▷ If applicable
    b_layer_dcmp_3 ← b_layer add the bias to the last layer             ▷ Or in the line above
    layer ← sequence(layer_dcmp_1, layer_dcmp_2, layer_dcmp_3)
end for
for each linear layer; layer in net_dcmp do
    W_layer ← take out weight matrix of size N_out × N_in
    R_A, R_B ← choose appropriate rank(s)
    G, A, B ← decompose W_layer using relevant algorithm               ▷ Or just A or B
    layer_dcmp_1 ← define new R_B × N_in linear layer                   ▷ If applicable
    layer_dcmp_2 ← define new R_A × R_B linear layer                    ▷ Or R_B × N_in or N_out × R_A
    layer_dcmp_3 ← define new N_out × R_A linear layer                  ▷ If applicable
    W_layer_dcmp_1 ← B                                                  ▷ If applicable
    W_layer_dcmp_2 ← G
    W_layer_dcmp_3 ← A                                                  ▷ If applicable
    b_layer_dcmp_3 ← b_layer add the bias to the last layer             ▷ Or in the line above
    layer ← sequence(layer_dcmp_1, layer_dcmp_2, layer_dcmp_3)
end for
train net_dcmp using the given training data                            ▷ fine-tuning
```

5.3 Architectures Used in This Thesis

In order to assess the performance of the different methods used in this thesis, baselines need to be established. To do this original or uncompressed architectures need to be chosen, based on what

have previously proved to be appropriate in the different cases, and what intuitively makes sense. The architectures for the two different parts mentioned earlier in this section will be discussed in the following.

5.3.1 Method 1

For the first method that decomposes the input different values of the rank R will be tried, hence it is only the simple ANN architecture that will need to be determined. For both data sets only one hidden layer will be provided in order to assess the amount of assistance the decomposition of the input provides to the ANN. For MNIST the number of hidden units will be experimentally set to 20, while for THETIS this number will be 100. This means that the whole ANN will consist of R input neurons, a hidden layer with 20 (MNIST) or 100 (THETIS) hidden neurons, and an output layer with 10 (MNIST) or 2 (THETIS) neurons. Figure 5.1 illustrates this architecture perfectly with L being the different ranks and $H = 20$ (MNIST) or $H = 100$ (THETIS).

5.3.2 Method 2

For the second method that decomposes a pre-trained network, a 2D and 3D CNN is applied to each of the data set respectively. For MNIST the so-called *LeNet-5* will be applied because it previously have shown good results for MNIST[14] and because it consists of a rather simple sequence of convolutions, pooling, and dense layers. The 3D version of the same architecture will be applied to THETIS, which is simply an extension of the convolutions (and pooling) to 3 dimensions (video) instead of 2. This gives the network the ability to learn not only the spatial features of the videos, but also the movement which is important in this case. Since the video problem is not to classify a general activity such as running or swimming, the whole movement needs to be considered in order for the network to stand a chance. The sizes of the filters have all been chosen using trial-and-error, and have not been optimised in any way, which means they should be considered experimental. An illustration of the two networks with all size information is given in Figure 5.3.

The choice of which Tucker method to use for each layer is inspired by Kim et al.[10] that would use Tucker-2 for all the convolutional layers (apart from the first) and the first dense layer, and Tucker-1 for the rest. An overview of which methods are used in the two architectures is given in Table 4.

Table 4: Which Tucker method (1 or 2) is applied the layers in each of the two architectures given in Figure 5.3 for method 2. The last layer is not compressed in any of the two, due to the small size

Tucker-method	Conv 1	Conv 2	Lin 1	Lin 2	Lin 3
MNIST	1	2	2	1	-
THETIS	2	2	2	1	-

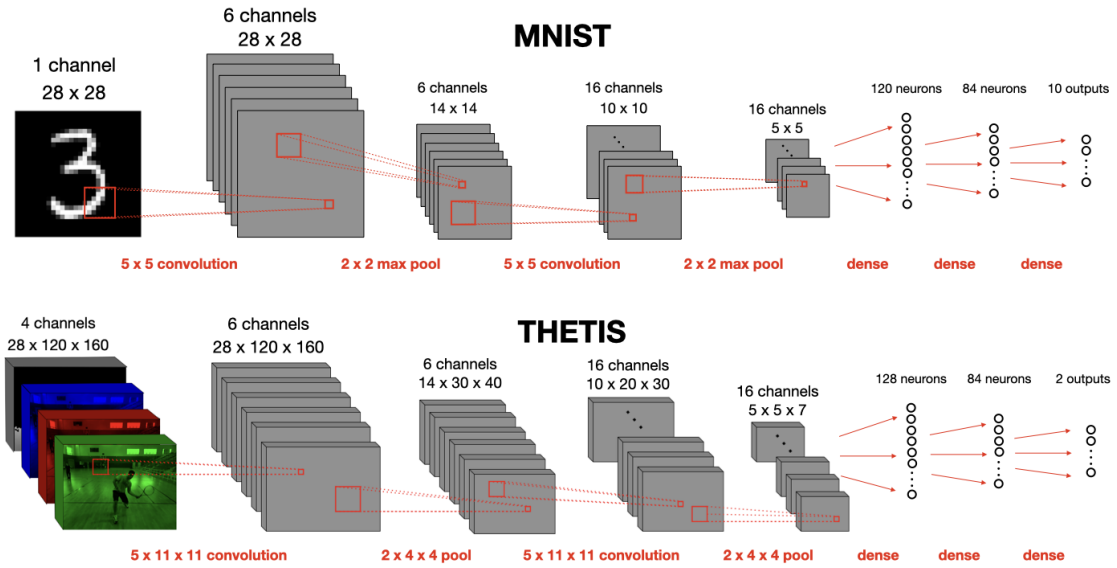


Figure 5.3: The architectures used for each of the data sets in method 2. The size of the intermediate matrices or tensors are given above them along with the number of channels. The red operations are named underneath them along with the size of the operation

5.4 Computational complexity

In order to assess the theoretical performance of the different networks, we will make use of the number of floating point operations (FLOPs) that it takes for the network to do one forward push. We assume that the non-linearity (activation function) and the max-pooling computations for free, and only consider the FLOPs used in the dense and convolutional layers respectively.

5.4.1 Dense layer

For a dense layer with N_{in} input neurons and N_{out} output neurons, the forward push is calculated as a matrix-vector product. This means that it has $N_{in}N_{out}$ multiplications and $(N_{in}-1)N_{out}$ additions making it a total of:

$$FLOPs_{lin} = N_{in}N_{out} + (N_{in}-1)N_{out} = N_{out}(2 \cdot N_{in} - 1) \tag{59}$$

FLOPs to compute a forward push through the dense layer.

5.4.2 Convolutional layer

The number of FLOPs needed for a forward push in a convolutional layer is not as simple to compute due to the many hyper-parameters. The FLOPs will be based on the number of outputs, since this is direct result of the hyper-parameters. The following will only be computed for a 3D video convolutional kernel, since it generalises to an image kernel. Consider a convolutional layer with input channels S, output channels T, convolutional kernel (D_F, D_H, D_W), and input size

(F, H, W). For every one of the $F'H'W'T$ output values, the filter is applied once which takes $D_F D_H D_W S$ multiplications and the same minus 1 additions. This makes the convolutional forward push use a total of:

$$FLOPs_{conv} = T \cdot F' \cdot H' \cdot W' \cdot (2 \cdot S \cdot D_F \cdot D_H \cdot D_W - 1) \tag{60}$$

FLOPs to calculate. Here the output size of each dimension is calculated using the formula:

$$out_{dim}(d, D, \Delta, P) = \frac{d - D + 2 \cdot P}{\Delta} + 1 \tag{61}$$

Where d is the input size, D is the kernel size, Δ is the stride, and P is the padding - all of the given dimension.

5.4.3 Approximation of loadings matrix

In method 1, the loading matrix $\boldsymbol{A}$ needs to be estimated for the testing observation $\mathcal{X}^{I_1 \times I_2 \times \cdots \times I_n}$, which is also a part of the time it takes to run the algorithm. The $\boldsymbol{A}$ is approximated by:

$$\boldsymbol{A} \approx \boldsymbol{X}_{(1)} \boldsymbol{G}_{(1)}^{\dagger} \tag{62}$$

Where we assume that $\boldsymbol{G}_{(1)}^{\dagger}$ have already been estimated and that the unfolding of $\mathcal{X}$ is for free, which means that the approximation is a matrix-matrix product. $\boldsymbol{X}_{(1)}$ will have shape $I_1 \times \delta$ where $\delta = \prod_{k=2}^{n} I_k$, and I_1 is the number of input observations. $\boldsymbol{G}_{(1)}^{\dagger}$ will have shape $\delta \times R$ where R is the rank of the decomposition. The total number of FLOPs it takes to approximate $\boldsymbol{A}$ for I_1 observations is given by:

$$R \cdot I_1(2 \cdot \delta - 1), \qquad \text{where} \quad \delta = \prod_{k=2}^{n} I_k \tag{63}$$

Where I_k is the size of the kth of the n dimensions of $\mathcal{X}$.

6 Results

Like in the methods section, this section will also be in two parts. This is due to the two compression methods, that are to be tested, are so different in nature and cannot be directly compared. To compare the methods, the a set of metrics will be applied to the original and compressed models. The metrics are:

- **Accuracy** - especially the accuracy drop is an important metric in comparing the compressed model to the full model
- **Number of Parameters** - how much space the model takes to store
- **Theoretical speed** - based on the number of FLOPs, calculated using the formulas given in subsection 5.4.
- **Actual speed** - how long it takes to compute a forward push. Computed many times reporting the mean and standard deviation of the time it takes. Due to potential differences in performance improvements, the methods will both be timed running on a CPU and GPU.

Initially the results of method 1 will be given including observations from a small experiment, following this the results of applying method 2 will given.

6.1 Method 1

In order to get a better understanding of what happens using this method, an experiment with only two classes for each data set and a rank of 2 will be carried out. Afterwards the results will be listed using different ranks for the input in order to assess performance differences.

6.1.1 Experiment with low rank

In the following an experiment is carried out to check whether the decomposition itself can carry much of the training. Since we have only 2 classes for the THETIS data set, and for the sake of the experiment 2 digits from the MNIST data set will be used. Since there are 2 classes in each data set, an initial rank of 2 will be experimentally chosen in the between sample dimension. The results for both the data set are described in the following.

6.1.1.1 Using MNIST 3s and 4s

Only the MNIST observations depicting either a 3 or a 4 will be considered in this experiment. These digits are chosen due to their distinct shapes, hence ease of classification. Since the purpose is to make the algorithm find the differences between the two different digits, the spatial dimensions will not be decomposed. Using the experimental rank of 2, the decomposition of N stacked 3s and 4s becomes:

$$\mathcal{X}^{N\times 28\times 28} \approx \mathcal{G}^{2\times 28\times 28} \times_1 \boldsymbol{A}^{N\times 2} \qquad \Leftrightarrow \qquad \boldsymbol{X}_{(1)} \approx \boldsymbol{A}^{N\times 2}\, \boldsymbol{G}_{(1)}^{2\times 28\cdot 28} \tag{64}$$

Where $\boldsymbol{A}$ is the loading matrix in the dimension corresponding to different pictures. With the rank equal to 2, $\boldsymbol{A}$ holds 2 values per picture that should ideally be separating the two digits that is to

(a) Original MNIST training examples including only 3s and 4s.

(b) Approximated versions of the training examples on the left, using the decomposition.

Figure 6.1: MNIST training examples before and after decomposing with only rank 2 in the input dimension. Notice how the decomposed 3s and 4s look more standardized. It seems that every picture is a part standardized 3 and a part standardized 4. Notice how digits that look relatively odd results in less certain approximation.

be trained. Figure 6.1 shows how the decomposition of the first 100 training examples turns out. It seems the decomposition is able how to find the "general 3" and the "general 4", and then simply use the loadings of $\boldsymbol{A}$ to specify how much of each should be used to represent every observation. Figure 6.2a shows a scatter-plot of the values of $\boldsymbol{A}$ colored to show the loadings corresponding to 3s and 4s respectively. There is some overlap between the two clusters, but it seems that they can be fairly distinguished. Using the mean of $\boldsymbol{A}$ of the the 3s, 4s and overall respectively as loadings in the decomposition yields the approximated pictures shown in Figure 6.2b-d.

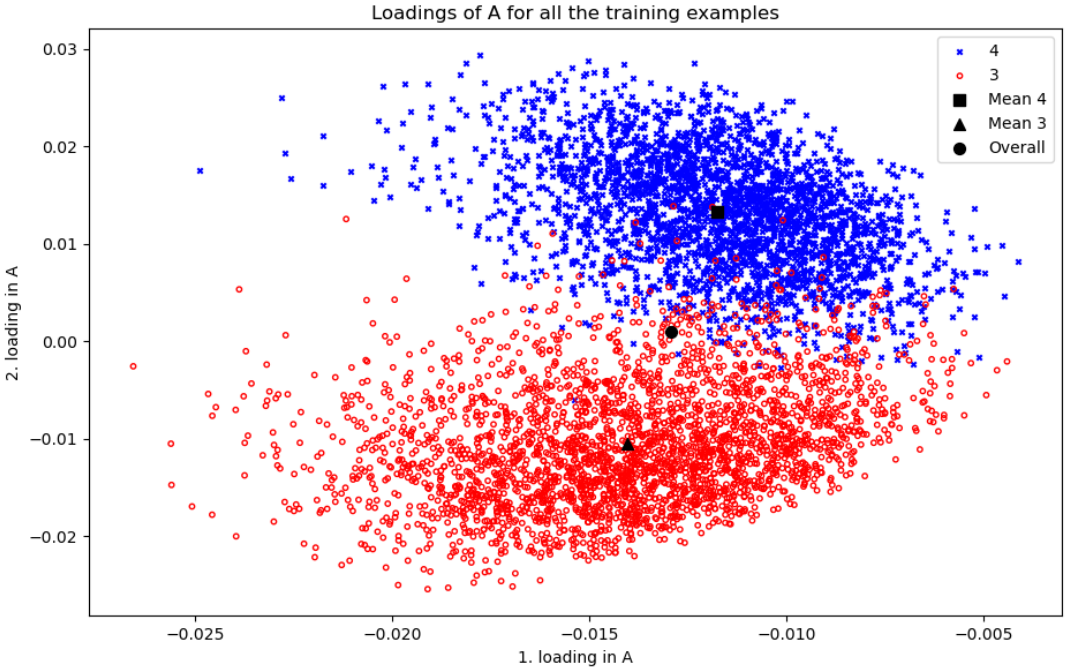
This all comes down to the loadings of $\boldsymbol{A}$ holding a great deal of information about which pictures are 3s and which are 4s. This information could be used as input into the NN instead of the actual pictures potentially reducing the number of parameters dramatically (from 784 to 2 input neurons in this example).

6.1.1.2 Using THETIS forehand flat and backhand

Following the same procedure as for the MNIST 3s and 4s we set the rank of the decomposition to 2. Stacking N videos and decomposing with rank 2 in the between sample dimension yields:

$$\mathcal{X}^{N\times 4\times 28\times 120\times 160} \approx \mathcal{G}^{2\times 4\times 28\times 120\times 160} \times_1 \boldsymbol{A}^{N\times 2} \tag{65}$$

Now the loading matrix $\boldsymbol{A}$ holds 2 values for each observation that should ideally be separating the two types of shots. Making the scatter plot of the loadings from $\boldsymbol{A}$ colored with the type this is however not the case. The plot is shown in Figure 6.3, where it is clear that there are two clusters,



(a) Scatter plot of the 2 loadings of the loading matrix $\boldsymbol{A}$ for the MNIST 3s and 4s respectively using rank 2 in the input dimension. The mean of the 2 clusters and overall is also marked with black.

(b) The mean of 4s loadings

(c) The mean of 3s loadings

(d) The mean of all loadings

Figure 6.2: Scatter plot of the 2 loadings of $\boldsymbol{A}$ for the MNIST 3s and 4s using rank 2 including mean in each of the clusters and overall. Using these means as loadings in the approximation gives the approximated "general" 3, 4 and overall given in **(b)-(d)**. Notice how the overall mean gives a mixture of a 3 and a 4.

but the clusters do not separate the type of shot, but the location of the video. The mean of each of the clusters have been used to approximate the videos given in Figure 6.3(b) and (c)

6.1.2 Results for MNIST

The results for MNIST using the method that decomposes the input are given in Table 5. Here the different models have been trained and timed, reporting the mean and the standard deviation of the time of 1,000 forward pushes through the network. Since the data is small and the models are all very fast, every forward push is conducted using 10 observations, which means that a total of 10,000 forward pushes have been timed. In Table 5 the brackets holds the time of each of the two parts of the algorithm, i.e. estimating the input loadings, and pushing through the ANN.

It seems...

6.1.3 Results for THETIS

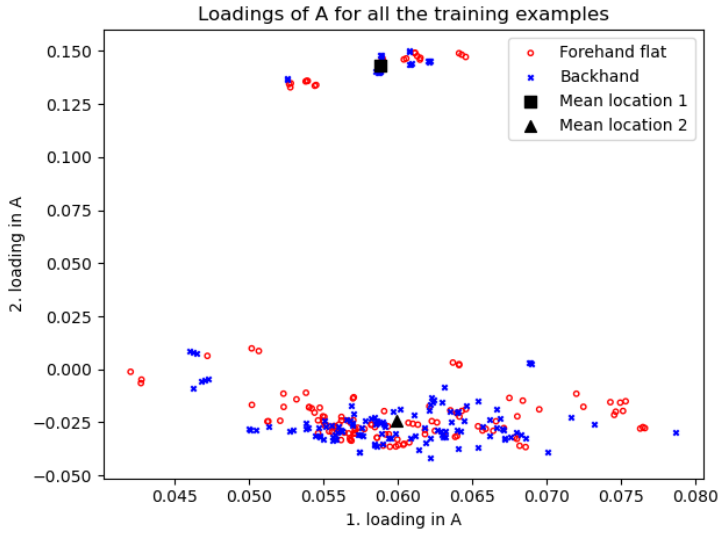
The results for THETIS are given in Table 6. The time is given as the mean and standard deviation of the time based on 1000 forward pushes through the different models, assuming the core have already been matricized and inverted in (62). The brackets holds the time broken down into the two parts - estimating the loading matrix and pushing through the ANN.

Interestingly the accuracy of the decomposed models only match that of the original network, when the computational complexity and the computation time are exceeded. This fact casts doubt on the functionality of this method. When running on the GPU the computation time is exceeded at an even lower rank than the CPU, which could mean that the GPU is more penalized by the split, i.e. the algorithm. One thing that is also interesting is that the estimation of the loading matrix accounts for the majority of the computation time by far. This is due to the estimation of the loading matrix getting the full input which means a lot of parameters.

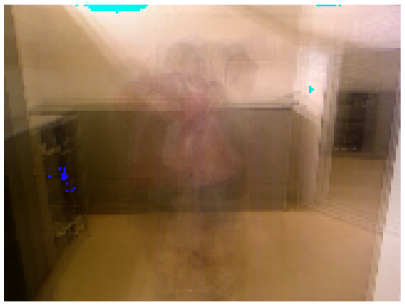
6.2 Method 2

6.2.1 Results for MNIST

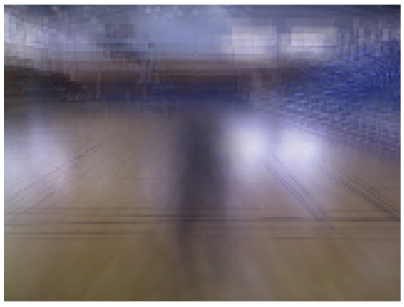
6.2.2 Results for THETIS



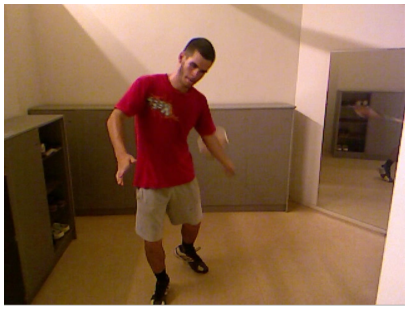
(a) Scatter plot of the loadings of $\boldsymbol{A}$ for the THETIS forehands and backhands. There seems to be two clusters, but they are not divided according to the shot type, but rather the background information. The mean of the two clusters are given, which are used to approximate the videos in **(b)** and **(c)**.



(b) Approximation using the loadings of the mean of the little cluster in **(a)**. This corresponds to one of the two locations.



(c) Approximation using the loadings of the mean of the big cluster in **(a)**. This corresponds to one of the two locations.



(d) An example of the location seen approximated in **(b)**

(e) An example of the location seen approximated in **(c)**

Figure 6.3: The loadings of $\boldsymbol{A}$ for the THETIS data illustrated by a scatter plot in **(a)**, and used to make the two approximation **(b)** and **(c)** by estimating the mean of each cluster seen in the scatter plot.

Table 5: The results of running the input decomposition method on the MNIST data set using different ranks for the decomposition. The time is reported as the mean and standard deviation of 1000 samples where 10 observations are evaluated using the model. Notice that even though the rank of

Rank	**Parameters (K)**	**FLOPs (K)**	**Time CPU (ms)**	**Time GPU (ms)**	**Accuracy (%)**
2	1.83	3.58	1.481 ± 0.025 $(= 0.904 + 0.647)$	0.149 ± 0.012 $(= 0.014 + 0.118)$	**33.61**
ratio	0.11	0.11	0.871	1.155	0.36
5	4.25	8.41	2.445 ± 0.039 $(= 1.885 + 0.551)$	0.144 ± 0.011 $(= 0.013 + 0.115)$	**52.20**
ratio	0.25	0.26	1.438	1.116	0.55
7	5.86	11.62	3.114 ± 0.047 $(= 2.525 + 0.565)$	0.142 ± 0.010 $(= 0.013 + 0.113)$	**61.71**
ratio	0.35	0.37	1.832	1.101	0.65
10	8.27	16.44	1.660 ± 0.033 $(= 0.968 + 0.681)$	0.141 ± 0.011 $(= 0.013 + 0.113)$	**67.55**
ratio	0.50	0.52	0.976	1.093	0.72
15	12.30	24.48	1.906 ± 0.035 $(= 1.372 + 0.527)$	0.141 ± 0.011 $(= 0.013 + 0.113)$	**72.14**
ratio	0.74	0.77	1.121	1.093	0.76
20	16.32	32.51	1.965 ± 0.031 $(= 1.354 + 0.598)$	0.142 ± 0.011 $(= 0.014 + 0.113)$	**79.41**
ratio	0.98	1.02	1.156	1.101	0.84
30	24.37	48.58	2.397 ± 0.035 $(= 1.785 + 0.599)$	0.142 ± 0.011 $(= 0.015 + 0.115)$	**74.66**
ratio	1.46	1.53	1.410	1.101	0.79
40	32.42	64.65	2.298 ± 0.039 $(= 1.756 + 0.528)$	0.144 ± 0.011 $(= 0.024 + 0.116)$	**74.89**
ratio	1.94	2.04	1.352	1.116	0.79
50	40.47	80.72	3.005 ± 0.043 $(= 2.242 + 0.752)$	0.146 ± 0.013 $(= 0.025 + 0.114)$	**81.56**
ratio	2.43	2.54	1.768	1.132	0.86
70	56.57	112.86	3.661 ± 0.047 $(= 3.042 + 0.581)$	0.144 ± 0.010 $(= 0.022 + 0.115)$	**75.13**
ratio	3.39	3.56	2.154	1.116	0.80
100	80.72	161.07	4.796 ± 0.067 $(= 3.897 + 0.823)$	0.144 ± 0.011 $(= 0.027 + 0.114)$	**83.98**
ratio	4.84	5.08	2.821	1.116	0.89
150	120.97	241.42	6.664 ± 0.086 $(= 5.655 + 0.849)$	0.145 ± 0.012 $(= 0.034 + 0.115)$	**83.65**
ratio	7.25	7.61	3.920	1.124	0.89
Original	16.68	31.73	1.700 ± 0.052	0.129 ± 0.013	**94.44**

Table 6: Results for running the input decomposition algorithm using different ranks for the decomposition. The time is given as the mean and standard deviation of 1000 forward pushed for each model. The numbers in brackets correspond to the break-down of the time of the estimation of the input loadings and the forward push through the network respectively. Notice how the accuracy of the original network is not reached before exceeding the computational complexity and time.

Rank	**Parameters (M)**	**FLOPs (M)**	**Time CPU (ms)**	**Time GPU (ms)**	**Accuracy (%)**
30	64.52	129.03	30.534 ± 0.325 $(= 30.322 + 0.081)$	0.340 ± 0.073 $(= 0.246 + 0.209)$	**70**
ratio	0.30	0.30	0.33	0.33	0.76
40	86.02	172.04	38.481 ± 0.371 $(= 38.085 + 0.087)$	0.496 ± 0.102 $(= 0.357 + 0.243)$	**70**
ratio	0.40	0.40	0.42	0.47	0.76
50	107.53	215.05	47.790 ± 0.300 $(= 47.474 + 0.093)$	0.685 ± 0.155 $(= 0.491 + 0.285)$	**82**
ratio	0.50	0.50	0.52	0.66	0.89
60	129.03	258.06	58.151 ± 0.373 $(= 57.811 + 0.093)$	0.729 ± 0.149 $(= 0.521 + 0.293)$	**78**
ratio	0.59	0.60	0.63	0.70	0.85
70	150.54	301.07	72.011 ± 0.450 $(= 71.641 + 0.092)$	0.940 ± 0.186 $(= 0.674 + 0.340)$	**74**
ratio	0.69	0.70	0.76	0.90	0.80
80	172.04	344.08	75.648 ± 0.398 $(= 75.409 + 0.093)$	0.793 ± 0.159 $(= 0.568 + 0.308)$	**76**
ratio	0.79	0.80	0.82	0.76	0.83
90	193.55	387.09	90.569 ± 0.476 $(= 90.355 + 0.093)$	1.194 ± 0.231 $(= 0.853 + 0.394)$	**82**
ratio	0.89	0.90	0.98	1.14	0.89
100	215.05	430.10	99.905 ± 0.591 $(= 99.507 + 0.093)$	1.346 ± 0.257 $(= 0.960 + 0.427)$	**82**
ratio	0.99	1.00	1.08	1.29	0.89
110	236.56	473.11	106.205 ± 0.685 $(= 106.114 + 0.092)$	1.496 ± 0.284 $(= 1.067 + 0.460)$	**80**
ratio	1.09	1.10	1.15	1.43	0.87
120	258.06	516.12	113.461 ± 0.986 $(= 113.081 + 0.093)$	1.394 ± 0.269 $(= 0.995 + 0.438)$	**88**
ratio	1.19	1.20	1.23	1.33	0.96
150	322.58	645.15	138.881 ± 0.910 $(= 138.528 + 0.094)$	1.809 ± 0.339 $(= 1.288 + 0.528)$	**94**
ratio	1.49	1.50	1.51	1.73	1.02
200	430.10	860.20	174.968 ± 1.870 $(= 174.853 + 0.094)$	2.362 ± 0.429 $(= 1.682 + 0.648)$	**94**
ratio	1.98	2.00	1.90	2.26	1.02
Original	217.19	430.08	95.243 ± 1.642	1.045 ± 0.255	**92**

Table 7: Caption

MNIST Layer		S/R_{in}	T/R_{out}	Weights	FLOPs (K)
	Orig	1	6	156	231.28
Conv 1	Comp		1	37	43.9 (= 38.42 + 5.49)
	Impr			×4.22	×5.27
	Orig	6	16	2416	478.5
Conv 2	Comp	4	6	736	171.6 (= 34.5 + 119.4 + 17.7)
	Impr			×3.28	×2.79
	Orig	400	120	48120	96
Lin 1	Comp	18	18	9804	19.33 (= 14.38 + 0.63 + 4.32)
	Impr			×4.91	×4.97
	Orig	120	84	10164	20.16
Lin 2	Comp	-	9	1920	3.66 (= 2.15 + 1.51)
	Impr			×5.29	×5.51
	Orig	84	10	850	1.68
Lin 3	Comp	-	-	850	1.68
	Impr			×1	×1.00
	Orig			61706	827.62
Total	Comp			13347	240.18
	Impr			×4.62	×3.45

Table 8: Caption

MNIST Layer		Time CPU	Time GPU
Conv 1	Orig	7.085 ± 1.347	0.143 ± 0.003
	Comp	12.274 ± 0.207 $(= 3.422 + 8.852)$	0.223 ± 0.003 $(= 0.087 + 0.136)$
	Impr	$\times 0.577$	$\times 0.64$
Conv 2	Orig	2.279 ± 0.074	0.156 ± 0.004
	Comp	4.182 ± 0.104 $(= 1.401 + 0.876 + 1.906)$	0.292 ± 0.004 $(= 0.074 + 0.089 + 0.128)$
	Impr	$\times 0.545$	$\times 0.54$
Lin 1	Orig	0.31 ± 0.067	0.096 ± 0.002
	Comp	0.373 ± 0.012 $(= 0.203 + 0.057 + 0.113)$	0.235 ± 0.005 $(= 0.093 + 0.063 + 0.079)$
	Impr	$\times 0.831$	$\times 0.41$
Lin 2	Orig	0.11 ± 0.014	0.078 ± 0.002
	Comp	0.132 ± 0.006 $(= 0.057 + 0.075)$	0.135 ± 0.004 $(= 0.061 + 0.074)$
	Impr	$\times 0.833$	$\times 0.58$
Lin 3	Orig	0.112 ± 0.006	0.08 ± 0.003
	Comp	0.105 ± 0.004	0.079 ± 0.001
	Impr	$\times 1.067$	$\times 1.01$
Total	Orig	9.896 ± 1.380	0.552 ± 0.008
	Comp	17.065 ± 0.365	0.964 ± 0.007
	Impr	$\times 0.58$	$\times 0.57$

Table 9: Caption

THETIS	**Layer**	S/R_{in}	T/R_{out}	Weights (K)	FLOPs
	Orig	1	6	14.52	15609.2M
Conv 1	Comp	2	2	2.45	2618.6M (= 7.5 + 2600.9 + 10.2)
	Impr			×5.94	×5.96
	Orig	6	16	58.10	696.9M
Conv 2	Comp	2	3	3.74	44.4M (= 0.4 + 43.5 + 0.5)
	Impr			×15.54	×15.7
	Orig	2800	128	358.53	716.8K
Lin 1	Comp	1	1	3.06	5856 (= 5599 + 1 + 256)
	Impr			×117.28	×122.4
	Orig	128	84	10.84	21.5K
Lin 2	Comp	-	1	0.3	423 (= 255 + 168)
	Impr			×36.12	×50.84
	Orig	84	10	0.17	336
Lin 3	Comp	-	-	0.17	336
	Impr			×1.00	×1.00
	Orig			442.15	16306.8M
Total	Comp			13.35	2663.1M
	Impr			×33.13	×6.12

Table 10: Caption

THETIS Layer		Time CPU (ms)	Time GPU (ms)
Conv 1	Orig	3518.114 ± 21.41	2.282 ± 0.064
	Comp	2643.672 ± 120.653	0.289 ± 0.005
		$(= 412.22 + 1697.69 + 533.77)$	$(= 0.09 + 0.07 + 0.13)$
	Impr	$\times 1.33$	$\times 7.90$
Conv 2	Orig	75.644 ± 0.439	0.135 ± 0.003
	Comp	38.156 ± 0.544	56.615 ± 0.176
		$(= 9.26 + 22.89 + 6.01)$	$(= 0.07 + 0.07 + 56.47)$
	Impr	$\times 1.98$	$\times 0.002$
Lin 1	Orig	0.57 ± 0.102	0.105 ± 0.003
	Comp	0.275 ± 0.018	1.366 ± 0.029
		$(= 0.16 + 0.04 + 0.07)$	$(= 0.56 + 0.73 + 0.08)$
	Impr	$\times 2.07$	$\times 0.08$
Lin 2	Orig	0.11 ± 0.011	109.633 ± 0.254
	Comp	0.089 ± 0.007	0.131 ± 0.002
		$(= 0.04 + 0.05)$	$(= 0.06 + 0.07)$
	Impr	$\times 1.24$	$\times 839.61$
Lin 3	Orig	0.081 ± 0.009	0.845 ± 0.005
	Comp	0.079 ± 0.068	0.078 ± 0.002
	Impr	$\times 1.03$	$\times 10.9$
Total	Orig	3594.520 ± 21.607	113.000 ± 0.263
	Comp	2682.271 ± 13.760	58.478 ± 0.176
	Impr	$\times 1.34$	$\times 1.93$

7 Discussion

From the results given in the previous section, it is clear that the speed of a NN method is not simply a function of the number of FLOPs that is used to produce the output, but that much more goes on that impacts the performance.

- Penalty for more layers that impacts the performance
- When to use which method
  - The decomposition of the input allows for much lower input dimensions and seems appropriate for simple classification tasks where some structure is already given. It could also be appropriate for instance for background removal or dynamic information extraction. When the complexity rises (for instance with MNIST digits that look alike) then this algorithm seems to fall short.
  - The decomposition of the pre-trained network works well for a network that is already working and is in some sense like pruning the network... Find structure and remove the unnecessary.
  - Pre-training a network allows for much more complex networks that might be easier to train. In this way the challenging part is reserved for the powerful machine while the actual application is less power and time consuming.
- Decomposing only input-output channels
  - Smart because the spatial (or temporal) dimensions of the kernel are often small (3 or 5), and because multiple filters might be finding almost the same information or something that is redundant across multiple filters.
- Power of the algorithm
  - Tucker - decomposition can rather easily be used to decompose both linear and convolutional layers.

7.1 Future work

- Rank selection
  - Evaluating the performance of VBMF and standardising so it always work. (Has problems for linear layers in one dimension)
  - Looking into other methods of doing rank selection - maybe a simpler heuristic
  - Evaluate the importance of getting good weights. Others have shown that the approximation of the weights using the decomposition need not to be perfect. (fine-tuning)
- 1×1 convolution
  - Look into what causes the 1x1 convolution to be cache-inefficient.

  - Investigate how the 1 x 1 convolution could be performed more cache efficiently (e.g. rotating everything so it would be the same just computed from an other angle)

- Other methods for decomposition
  - Try using decomposition as part of the learning / evaluation sequence using the pytorch autograd.
  - Look into using / combining other decomposition methods
    * TT - decomposition (only done for linear layers)
    * BTD - showed very promising results for the convolutional layer

References

[1] Anthony F Beavers. "Alan Turing : Mathematical Mechanist". In: *Alan Turing His Work Impact* (2013), pp. 481–485.

[2] Cas van den Bogaard. *VBMF python implementation*. 2017. URL: https://github.com/CasvandenBogaard/VBMF.

[3] Misha Denil et al. "Predicting parameters in deep learning". In: *Adv. Neural Inf. Process. Syst.* (2013), pp. 1–9. ISSN: 10495258. arXiv: 1306.0543.

[4] Sofia Gourgari et al. "THETIS: Three dimensional tennis shots a human action dataset". In: *IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit. Work.* (2013), pp. 676–681. ISSN: 21607508. DOI: 10.1109/CVPRW.2013.102.

[5] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (2020), pp. 357–362. ISSN: 14764687. DOI: 10.1038/s41586-020-2649-2. arXiv: 2006.10256. URL: http://dx.doi.org/10.1038/s41586-020-2649-2.

[6] Trevor Hastie, Jerome H. Friedman, and Robert Tibshirani. "Neural Networks". In: *Elem. Stat. Learn. Data Mining, Inference, Predict.* Second. New York: Springer US, 2001, pp. 389–416. ISBN: 978-0-387-84857-0. DOI: 10.1007/b94608.

[7] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. "Speeding up convolutional neural networks with low rank expansions". In: *BMVC 2014 - Proc. Br. Mach. Vis. Conf. 2014* (2014). DOI: 10.5244/c.28.88. arXiv: 1405.3866.

[8] Andrej Karpathy et al. "Large-scale video classification with convolutional neural networks". In: *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.* (2014), pp. 1725–1732. ISSN: 10636919. DOI: 10.1109/CVPR.2014.223.

[9] Naresh Khuriwal and Nidhi Mishra. "Breast Cancer Diagnosis Using Deep Learning Algorithm". In: *Proc. - IEEE 2018 Int. Conf. Adv. Comput. Commun. Control Networking, ICACCCN 2018* (2018), pp. 98–103. DOI: 10.1109/ICACCCN.2018.8748777.

[10] Yong Deok Kim et al. "Compression of deep convolutional neural networks for fast and low power mobile applications". In: *4th Int. Conf. Learn. Represent. ICLR 2016 - Conf. Track Proc.* (2016), pp. 1–16. arXiv: 1511.06530.

[11] Tamara G Kolda and Brett W Bader. "Tensor Decompositions and Applications". In: *SIAM Rev.* 51.November (2009), pp. 455–500. URL: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0%7B%5C#%7Donline.

[12] Jean Kossaifi et al. "Tensorly: Tensor Learning in Python". In: *J. Mach. Learn. Res.* 20 (2019), pp. 1–6.

[13] Vadim Lebedev et al. "Speeding-up convolutional neural networks using fine-tuned CP-decomposition". In: *3rd Int. Conf. Learn. Represent. ICLR 2015 - Conf. Track Proc.* (2015), pp. 1–11. arXiv: 1412.6553.

[14] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. *The MNIST database of handwritten digits*. 1998.

[15] Morten Mørup. "Applications of tensor (multiway array) factorizations and decompositions in data mining". In: *Wiley Interdiscip. Rev. Data Min. Knowl. Discov.* 1.1 (2011), pp. 24–40. ISSN: 19424787. DOI: 10.1002/widm.1.

[16] Shinichi Nakajima et al. "Global analytic solution of fully-observed variational bayesian matrix factorization". In: *J. Mach. Learn. Res.* 14.1 (2013), pp. 1–37. ISSN: 15324435.

[17] Alexander Novikov et al. "Tensorizing neural networks". In: *Adv. Neural Inf. Process. Syst.* 2015-Janua (2015), pp. 442–450. ISSN: 10495258. arXiv: 1509.06569.

[18] Ivan V Oseledets. "Tensor-Train Decomposition". In: *SIAM J. Sci. Comput.* 33 (2011), pp. 2295–2317.

[19] Adam Paszke et al. "PyTorch : An Imperative Style , High-Performance Deep Learning Library". In: NeurIPS (2019). arXiv: arXiv:1912.01703v1.

[20] Roberto Rigamonti et al. "Learning separable filters". In: *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.* (2013), pp. 2754–2761. ISSN: 10636919. DOI: 10.1109/CVPR.2013.355.

[21] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *3rd Int. Conf. Learn. Represent. ICLR 2015 - Conf. Track Proc.* (2015), pp. 1–14. arXiv: arXiv:1409.1556v6.

[22] Amos Sironi et al. "Learning separable filters". In: *IEEE Trans. Pattern Anal. Mach. Intell.* 37.1 (2015), pp. 94–106. ISSN: 01628828. DOI: 10.1109/TPAMI.2014.2343229.

[23] Christian Szegedy et al. "Going deeper with convolutions". In: *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.* 07-12-June (2015), pp. 1–9. ISSN: 10636919. DOI: 10.1109/CVPR.2015.7298594. arXiv: 1409.4842.

[24] Peisong Wang and Jian Cheng. "Accelerating convolutional neural networks for mobile applications". In: *MM 2016 - Proc. 2016 ACM Multimed. Conf.* (2016), pp. 541–545. DOI: 10.1145/2964284.2967280.

[25] Wikipedia. *Alan Turing - Wikipedia*. 2020. URL: https://en.wikipedia.org/wiki/Alan%7B%5C_%7DTuring (visited on 09/16/2020).

[26] Wikipedia. *Deep Learning*. URL: https://en.wikipedia.org/wiki/Deep%7B%5C_%7Dlearning (visited on 11/11/2020).

[27] Wikipedia. *MNIST database*. 2020. URL: https://en.wikipedia.org/wiki/MNIST%7B%5C_%7Ddatabase%7B%5C#%7Dcite%7B%5C_%7Dnote-5 (visited on 10/27/2020).

[28] Jason Yosinski et al. "Understanding Neural Networks Through Deep Visualization". In: (2015). arXiv: 1506.06579. URL: http://arxiv.org/abs/1506.06579.

[29] Zhifei Zhang. "Derivation of Backpropagation in Convolutional Neural Network (CNN)". In: (2016), pp. 1–7.

A Decomposition Estimation

In the following the methods for estimating the decompositions will be reviewed. The decomposition methods that will be covered are Tucker, BTD, CP / PARAFAC, TT-decomposition. The algorithms will be described assuming a 3rd order tensor, but generalizes to both lower and higher orders. It is worth mentioning that none of the algorithms have closed form solutions, hence will need to be fit in order to minimize the error. Additionally the methods are not guaranteed to converge to the global optimum.

A.1 Tucker Decomposition

Traditionally, Alternating Least Squares (ALS) have been used to estimate the Tucker decomposition[15, p. 27], however the algorithm that is used today in *TensorLy*[12] is called Higher-Order Orthogonal Iteration (HOOI). The ALS algorithm simply updates each of the loading matrices using the estimates of the others, and keep doing this until convergence. The updates uses the Moore-Penrose pseudo-inverse in the following way:

$$\boldsymbol{A} \leftarrow \boldsymbol{X}_{(1)} \left(\boldsymbol{G}_{(1)}(\boldsymbol{C} \otimes \boldsymbol{B})^\top\right)^\dagger \tag{66}$$

$$\boldsymbol{B} \leftarrow \boldsymbol{X}_{(2)} \left(\boldsymbol{G}_{(2)}(\boldsymbol{C} \otimes \boldsymbol{A})^\top\right)^\dagger \tag{67}$$

$$\boldsymbol{C} \leftarrow \boldsymbol{X}_{(3)} \left(\boldsymbol{G}_{(3)}(\boldsymbol{B} \otimes \boldsymbol{A})^\top\right)^\dagger \tag{68}$$

$$\mathcal{G} \leftarrow \mathcal{X} \times_1 \boldsymbol{A}^\dagger \times_2 \boldsymbol{B}^\dagger \times_3 \boldsymbol{C}^\dagger \tag{69}$$

Imposing orthogonality simplifies this such that the estimation of the core can be omitted. This is achieved by using SVD to estimate the loading matrices at each step, i.e. as the first number of left singular vectors corresponding to the rank of the given dimension. In the end the HOOI algorithm comes down to iteratively solving the right hand sides of:

$$\boldsymbol{A}\boldsymbol{S}_1\boldsymbol{V}_1^\top = \boldsymbol{X}_{(1)} (\boldsymbol{C} \otimes \boldsymbol{B}) \tag{70}$$

$$\boldsymbol{B}\boldsymbol{S}_2\boldsymbol{V}_2^\top = \boldsymbol{X}_{(2)} (\boldsymbol{C} \otimes \boldsymbol{A}) \tag{71}$$

$$\boldsymbol{C}\boldsymbol{S}_3\boldsymbol{V}_3^\top = \boldsymbol{X}_{(3)} (\boldsymbol{B} \otimes \boldsymbol{A}) \tag{72}$$

Upon convergence the core can be estimated by the pseudo-inverse in the same way as for the ALS algorithm:

$$\mathcal{G} \leftarrow \mathcal{X} \times_1 \boldsymbol{A}^\dagger \times_2 \boldsymbol{B}^\dagger \times_3 \boldsymbol{C}^\dagger \tag{73}$$

The loading matrices can be initialized either by guessing randomly or by Higher-Order SVD (HOSVD) which is simply given by the SVD of the matricized arrays:

$$\boldsymbol{A}\boldsymbol{S}_1\boldsymbol{V}_1^\top = \boldsymbol{X}_{(1)} \tag{74}$$

$$\boldsymbol{B}\boldsymbol{S}_2\boldsymbol{V}_2^\top = \boldsymbol{X}_{(2)} \tag{75}$$

$$\boldsymbol{C}\boldsymbol{S}_3\boldsymbol{V}_3^\top = \boldsymbol{X}_{(3)} \tag{76}$$

A.2 Block-Term Decomposition (BTD)

Since the BTD is simply a sum of Tucker decompositions, the algorithm for estimating it is a generalization of HOOI which is used for Tucker. Wang et al. proposes this algorithm in [24] and claims it superior to other methods. The algorithm, which they call Principle Component Iteration (PCI), is breaking the input tensor into a number of principle components, which are trained iteratively by calculating the residual tensor using the other components until convergence. The algorithm is given in Algorithm 2.

Algorithm 2 Principle Component Iteration

procedure PCI($\mathcal{X}, R$)
  initialize $\mathcal{X}_r \leftarrow 0$ for $r = 1, \ldots, R$
  repeat
    for $r = 1$ to R **do**
      $\mathcal{X}_{res} \leftarrow \mathcal{X} - \texttt{sum}\left(\mathcal{X}_1, \ldots, \mathcal{X}_{r-1}, \mathcal{X}_{r+1}, \ldots, \mathcal{X}_R\right)$ ▷ Residual tensor
      $\mathcal{X}_r \leftarrow \texttt{HOOI}\left(\mathcal{X}_{res}\right)$ ▷ Updating r'th component
    end for
    $\mathcal{X}_{res} \leftarrow \mathcal{X} - \texttt{sum}\left(\mathcal{X}_1, \ldots, \mathcal{X}_R\right)$
  until $\mathcal{X}_{res}$ converges or maximum iterations are reached
  return $\mathcal{X}_1, \ldots, \mathcal{X}_R$
end procedure

A.3 PARAFAC / CP decomposition

The CP decomposition is estimated using ALS in *TensorLy*, since this method have proven superior to other method[15, p. 29]. Due to the CP decomposition being a special case of the Tucker model with a super-diagonal or identity core, we have:

$$\mathcal{X} \approx \sum_{r=1}^{R} \boldsymbol{a}_r \circ \boldsymbol{b}_r \circ \boldsymbol{c}_r \tag{77}$$

Where R is the rank of the decomposition. This can also be expressed using the matricized $\mathcal{X}$ and the Khatri-Rao product:

$$\boldsymbol{X}_{(1)} \approx \boldsymbol{A}\left(\boldsymbol{C} \odot \boldsymbol{B}\right)^{\top} \tag{78}$$

$$\boldsymbol{X}_{(2)} \approx \boldsymbol{B}\left(\boldsymbol{C} \odot \boldsymbol{A}\right)^{\top} \tag{79}$$

$$\boldsymbol{X}_{(3)} \approx \boldsymbol{C}\left(\boldsymbol{B} \odot \boldsymbol{A}\right)^{\top} \tag{80}$$

From these the objective, which is minimizing the error $\|\mathcal{X} - \hat{\mathcal{X}}\|_F$, can be achieved by iteratively updating:

$$\boldsymbol{A} \leftarrow \boldsymbol{X}_{(1)} (\boldsymbol{C} \odot \boldsymbol{B}) \left(\boldsymbol{C}^\top \boldsymbol{C} \bullet \boldsymbol{B}^\top \boldsymbol{B} \right)^{-1} \tag{81}$$

$$\boldsymbol{B} \leftarrow \boldsymbol{X}_{(2)} (\boldsymbol{C} \odot \boldsymbol{A}) \left(\boldsymbol{C}^\top \boldsymbol{C} \bullet \boldsymbol{A}^\top \boldsymbol{A} \right)^{-1} \tag{82}$$

$$\boldsymbol{C} \leftarrow \boldsymbol{X}_{(3)} (\boldsymbol{B} \odot \boldsymbol{A}) \left(\boldsymbol{B}^\top \boldsymbol{B} \bullet \boldsymbol{A}^\top \boldsymbol{A} \right)^{-1} \tag{83}$$

until convergence[11, p. 28]. The matrices $\boldsymbol{A}, \boldsymbol{B}$ and $\boldsymbol{C}$ will all have R columns which correspond to the vector loadings for each rank. Like for Tucker Decomposition these can be initialized either randomly or by HOSVD.

A.4 Tensor-Train (TT) decomposition

Also goes by the name of Matrix-Product-State decomposition, and was developed by Oseledets in [18]. In the same work he describes what he calls the TT-format of a tensor, how to estimate it and how to do calculations in the TT-format. The non-recursive algorithm developed by Oseledets is given in Algorithm 3. In addition to the d-dimensional tensor, the algorithm is given a prescribed accuracy ϵ and ensures that the approximated tensor $\hat{\mathcal{X}}$ satisfies:

$$\|\mathcal{X} - \hat{\mathcal{X}}\|_F \leq \epsilon \|\mathcal{X}\|_F \tag{84}$$

Hence the algorithm is not given ranks beforehand, but finds these automatically. The algorithm cam easily be adapted to take in ranks and just calculate the SVD of the unfolding (line 8) using the given ranks.

Algorithm 3 TT- Singular Value Decomposition

1: **procedure** TT-SVD($\mathcal{X}, \epsilon$)
2: initialize:
3: $\delta \leftarrow \frac{\epsilon}{\sqrt{d-1}} \|\mathcal{X}\|_F$ ▷ Truncation parameter
4: $\mathcal{Z} \leftarrow \mathcal{X}, \quad r_0 \leftarrow 1$ ▷ Temporary tensor
5: **for** $k = 1$ to $d - 1$ **do**
6: $\boldsymbol{Z} \leftarrow \texttt{reshape}\left(\mathcal{Z}, \left[r_{k-1} n_k, \frac{\prod_{i=1}^{d} n_i}{r_{k-1} n_k}\right]\right)$
7: $r_k = \texttt{rank}_\delta(\boldsymbol{Z})$ ▷ δ-rank of the unfolded tensor ensuring accuracy
8: $\boldsymbol{U}, \boldsymbol{S}, \boldsymbol{V} \leftarrow \texttt{tSVD}(\mathcal{Z}, r_k)$ ▷ δ-truncated SVD (first r_k vectors)
9: $\mathcal{G}_k \leftarrow \texttt{reshape}\left(\boldsymbol{U}, [r_{k-1}, n_k, r_k]\right)$ ▷ Computing kth core
10: $\boldsymbol{Z} \leftarrow \boldsymbol{S}\boldsymbol{V}^\top$ ▷ Updating temporary tensor
11: **end for**
12: $\mathcal{G}_d \leftarrow \boldsymbol{Z}$ **return** $\mathcal{G}_1, \ldots, \mathcal{G}_d$
13: **end procedure**

B Derivations of Tucker-1 Decomposition of the Convolutional Kernel

The convolution of an input tensor $\mathcal{X}$ of size $F \times H \times W \times S$ into an output tensor $\mathcal{Y}$ of size $F' \times H' \times W' \times T$ is given by:

$$\mathcal{Y}(f', h', w', t) = \sum_{i=1}^{D_F} \sum_{j=1}^{D_H} \sum_{l=1}^{D_W} \sum_{s=1}^{S} \mathcal{K}(i, j, l, s, t)\ \mathcal{X}(f_i, h_j, w_l, s) \tag{85}$$

Where:

$$f_i = (f' - 1)\,\Delta_F + i - P_F \qquad h_j = (h' - 1)\,\Delta_H + j - P_H \qquad w_l = (w' - 1)\,\Delta_W + l - P_W \tag{86}$$

Where the Δs are the strides in each of the dimensions, the Ps are the corresponding padding, and (f', h', w', t) is the position in the output tensor $\mathcal{Y}$. $\mathcal{K}$ is the 5-dimensional convolutional kernel, i.e. the stack of 4-dimensional filters, and the Ds are the filter sizes in each of the spatial and temporal dimensions.

B.1 Decomposing the Input Channel Dimension

Now the Tucker-1 decomposition of $\mathcal{K}$ with respect to the input channel dimension is given by:

$$\mathcal{K}(i, j, l, s, t) = \sum_{r_4=1}^{R_4} \mathcal{C}\,(i, j, l, r_4, t)\,\boldsymbol{U}^{(4)}\,(s, r_4) \tag{87}$$

Inserting this into (85) and performing rearrangements, yields:

$$\mathcal{Y}(f', h', w', t) = \sum_{i=1}^{D_F} \sum_{j=1}^{D_H} \sum_{l=1}^{D_W} \sum_{r_4=1}^{R_4} \mathcal{C}\,(i, j, l, r_4, t) \underbrace{\sum_{s=1}^{S} \boldsymbol{U}^{(4)}\,(s, r_4)\ \ \mathcal{X}(f_i, h_j, w_l, s)}_{\mathcal{Q}} \tag{88}$$

Summing out S yields an intermediate tensor $\mathcal{Q}$ of size $F \times H \times W \times R_4$. The remaining is simply a 3D convolution on R_4 input channels and T output channels:

$$\mathcal{Y}(f', h', w', t) = \sum_{i=1}^{D_F} \sum_{j=1}^{D_H} \sum_{l=1}^{D_W} \sum_{r_4=1}^{R_4} \mathcal{C}\,(i, j, l, r_4, t)\ \ \mathcal{Q}(f_i, h_j, w_l, r_4) \tag{89}$$

While $\mathcal{Q}$ is a $1 \times 1 \times 1$ convolution on S input and R_4 outout channels:

$$\mathcal{Q}(f, h, w, r_4) = \sum_{s=1}^{S} \boldsymbol{U}^{(4)}\,(s, r_4)\,\mathcal{X}\,(f, h, w, s) \tag{90}$$

Which means that the convolution (85) can be expressed as a $1 \times 1 \times 1$ convolution (90) followed by a 3D convolution (89).

B.2 Decomposing the Output Channel Dimension

Now the Tucker-1 decomposition of $\mathcal{K}$ with respect to the output dimension is given by:

$$\mathcal{K}(i,j,l,s,t) = \sum_{r_5=1}^{R_5} \mathcal{C}\left(i,j,l,s,r_5\right) \boldsymbol{U}^{(5)}\left(t,r_5\right) \tag{91}$$

Inserting this into (85) and performing rearrangements, yields:

$$\mathcal{Y}(f',h',w',t) = \sum_{r_5=1}^{R_5} \boldsymbol{U}^{(5)}\left(t,r_5\right) \underbrace{\sum_{i=1}^{D_F}\sum_{j=1}^{D_H}\sum_{l=1}^{D_W}\sum_{s=1}^{S} \mathcal{C}\left(i,j,l,s,r_5\right)\ \mathcal{X}(f_i,h_j,w_l,s)}_{\mathcal{Q}'} \tag{92}$$

Now summing out everything except r_5 yields an intermediate tensor $\mathcal{Q}'$ of size $F' \times H' \times W' \times R_5$. The remaining corresponds to a $1 \times 1 \times 1$ convolution on R_5 input and T output dimensions:

$$\mathcal{Y}(f',h',w',t) = \sum_{r_5=1}^{R_5} \boldsymbol{U}^{(5)}\left(t,r_5\right) \mathcal{Q}'(f',h',w',r_5) \tag{93}$$

While the intermediate tensor $\mathcal{Q}'$ corresponds to a 3D convolution on S input and R_5 output channels:

$$\mathcal{Q}'(f,h,w,r_5) = \sum_{i=1}^{D_F}\sum_{j=1}^{D_H}\sum_{l=1}^{D_W}\sum_{s=1}^{S} \mathcal{C}\left(i,j,l,s,r_5\right)\ \mathcal{X}(f_i,h_j,w_l,s) \tag{94}$$

Which means that the convolution (85) can be expressed by a 3D convolution (94) and a $1 \times 1 \times 1$ convolution (93).