Exploring the effects of memory usage on CP/TT decomposed CNNs

J.A. Klip

Master of Science Thesis

TU Delft Delft University of Technology

Delft Center for Systems and Control

Exploring the effects of memory usage on CP/TT decomposed CNNs

Master of Science Thesis

For the degree of Master of Science in Embedded Systems Engineering
at Delft University of Technology

J.A. Klip

February 17, 2025

Faculty of Mechanical Engineering (ME) · Delft University of Technology

Abstract

Artificial Intelligence (AI) put an increasing amount of strain on our total energy consumption and CO_2 production. Not only is AI becoming increasingly more popular, but also AI models keep growing and thus need an increasing amount of computational resources. Recent research tries to mitigate this effect by creating more efficient hardware and by using Green AI, which is research in AI with additional focus on the computational resources a model requires. During this research one such Green AI method will be studied. This research focusses on the effects of memory usage on Canoncial Polyadic Decomposition (CPD) and Tensor Train (TT) decomposed Convolutional Neural Networks (CNN)s. These decomposed kinds of CNNs reduce the amount of parameters in the model, but may increase the amount of memory that is required to run the model. Therefore, first a theoretical analysis will be done on the memory used by these models. This analysis will then be validated by doing a real life scenario test and the effects of memory usage on inference time will be explored. Finally, regressions models will be made to see whether it is possible to predict the inference time. The results of these tests show that decomposed CNNs require more memory and the memory required in the real-life scenarios is higher than that was expected in the theoretical analysis. For the tested systems, it is shown that memory does influence the inference time negatively. Additionally, it was found that for very small kernels some initialization bias seems to be present, which makes the inference time larger,despite the CNN having less parameters and requiring less memory. Finally, it is shown that despite this inference bias, it is possible to predict whether the use of decomposed CNNs is beneficial to use compared to a regular CNN in terms of inference time.

Table of Contents

Preface and Acknowledgements

First of all this project has given me deeper insights in the inner working of AI, which I had not earlier encountered during my masters Embedded Systems. When I started my bachelor in 2017, I had never heard of AI and when I started my masters in 2021. It was not as prominent and popular as it is today just shy of 4 years later. Modern AI no already do have a big impact on the environment and I hope that this thesis will be part of making AI just a bit greener.

All code created for this thesis can be found in the public Github repository, which can be found at https://github.com/jasperklp/Thesis_CP_TT_CNN.

Finally, I would like to thank all people which helped me during my thesis. First, I would like to thank my supervisor Dr.Ir. Kim Batselier with all guidance and insights he provided during all parts of my thesis project. Additionally, I would like to thank Eva Memmel for the input she gave in a few of my master thesis meetings. Furthermore I would like to thank all of my friends and family, which have supported me during this journey. Of these people, I would especially want to thank my parents for their support, Bas Cheizoo en Suzanne Schuurman for proofreading my report and Xanne for all the support she gave me throughout my thesis.

Delft, University of Technology — J.A. Klip
February 17, 2025

Chapter 1

Introduction

Today, Artificial Intelligence (AI) is becoming more and more important. Whilst gaining in popularity quickly over the last decade [28], the models to train and run the AI grew larger. For AI, a lot of energy is consumed in large data centres and whilst running these models an increasing amount of computational power is needed. This gives the models better performance and a better experience for the user, but this adds to the cost of power consumption. Thus increasing CO_2 emissions as large data centres are required to train and to make inferences. In the context of AI making inferences is the term used for giving an AI model an input and obtaining the result. This is different from training, because when training, often, an inference is done and then the result is analysed to improve the model. The latter is not done in the case of just inference [20, 38].

To mitigate the negative effects of these larger models on time and CO_2 emissions, two research directions are explored. The first is by using more advanced hardware, which is specifically created to run AI tasks or even specific AI tasks belonging to a certain type of model [36, 38]. The other research direction is by the use Green AI. "Green AI refers to AI research that yields novel results while taking into account the computational cost, encouraging a reduction in resources spent" from [38]. The latter of these two research directions will be further investigated during in this thesis [38, 41].

1-1 Motivation

This thesis will mainly focus on a specific type of AI, which is the Convolutional Neural Networks (CNN). CNNs are often used for applications involving image recognition. The idea for this thesis originates from the thesis of Demi Breen [7] and Fredrik Hogenbosch [15]. Both theses use tensor decomposed CNNs. Decomposed CNNs are a typeset of improvements on the regular CNNs.

In a decomposed CNN, the kernel of the convolution operation is decomposed into four smaller kernels using a low-rank tensor decomposition. In short, a low rank tensor decomposition is a method to split one multidimensional array (tensor) into multiple smaller tensors, often these

smaller tensors are 2 or 3 dimensional. With this the total number of elements can be reduced by creating a network of smaller tensors. The network is chosen such that the original tensor can be recreated or approximated. A variable that often plays a role here is the rank with this parameter the reduction of parameters can be chosen. However, some conditions in choosing this rank may apply. By choosing the rank of a tensor decomposition one can influence how much elements are in this new network and by this the amount of compression can be set to the original tensor.

Due to this decomposition of kernel the convolutional operation of the original CNN convolutional layer is altered in four smaller convolution operations. Choosing a higher or a lower compression has benefits and downsides. The first benefit of a higher compression is that due to having less parameters in total less computational operations are required. Additionally, the kernels are smaller to store saving extra requirements on hardware. A downside of these decomposed kernels is that more outputs are generated which may lead to extra memory consumption. Furthermore, when the kernel is compressed to much the accuracy of the model may decrease resulting in a less useful model. Hence, the amount of compression can be seen as a trade off on the one hand it may save computational resources while on the other hand the model accuracy may decay and more memory may be required [10, 27].

Demi's thesis The thesis of Demi [7] focusses on measuring and predicting energy savings using Canoncial Polyadic Decomposition (CPD), Tensor Train (TT) and Tucker decomposed CNN, which are all tensor decomposed CNNs. She compares these decomposed CNNs with regular CNNs. After doing some tests she finds that the TT decomposed CNN often results in less energy usage. With Tucker and CPD her mileage varies, but more often than not she concluded that for the systems she took decomposing does not lead to energy savings. This seems contradictory because by decomposing the CNN the amount of parameters and hence the amount of calculation that needs to be done during inference is reduced by up to 10x. In her thesis, she indicates that increased memory is used by the decomposed CNNs, but she does not point it out as a possible source of increased energy consumption.

Frederik's thesis Fredrik's thesis [15] focuses on the Canonical Polyadic-Convolutional Neural Network (CP-CNN) to improve recognition of EEG images, which are a type of medical images. During his thesis, he noticed that only for CP-CNNs with heavy compression did his training times decreased, but that for most kinds of compression, it increased. Additionally, he mentions that for some less extreme compression his memory consumption more than doubles. Both these phenomena are not further discussed, because they do not contribute to his research goal. Despite this, it is a noticeable phenomenon, because the purpose of using these tensor decomposed CNNs is to reduce the amount of parameters in the model. This does not seem to cohere to that.

Summarizing Both these theses indicate that there are some drawbacks to using decomposed CNNs compared to regular ones. Additionally, both suggest that memory may influence both inference and training speed. Demi's thesis indicates that there can be an increase in energy usage as well. This led to the problem formulation given in the next section.

1-2 Problem formulation

Main Question From both theses, the question arises of what influence memory has on inference time and to what extent. It is known that memory may have a great influence, but is this true in this case? From here, the main question of this thesis arises.

How does memory usage influence the inference time for Canonocial Polyadic decomposed and Tensor Train decomposed CNN layers compared to non-decomposed CNN layers?

Subquestions To answer this main question three subquestions are defined to lay down a path to find an answer to the main research question. These subquestions are a breakdown of the main question.

Subquestion 1 Can we derive the theoretical memory usage of a decomposed CNN and a non-decomposed (or 'regular') CNN?

Subquestion 2 How does the theoretical analysis compare to a real-life scenario and can we see an influence of the memory usage on the inference time?

Subquestion 3 Can we predict when it is more advantageous in terms of inference time to use a decomposed CNN or a regular CNN.

1-3 Thesis outline

In Chapter 2 some background information will be given on the tensor decompositions used. The tensor decompositions are a mathematical framework used to change parts of the regular CNN. From these so-called tensor decompositions the decomposed CNN as a group derive their name. After this, a small look will be taken at the inner workings of CNNs. Later these inner workings will be combined with the Tensor decompositions, which will give the CP-CNN and the Tensor Train Convolutional Neural Network (TT-CNN). Last but not least, in Chapter 2 a small introduction will be given about the computation of linear algebraic problems.

In Chapter 3, the methods will be explained that will be used to solve the problems imposed by the main and sub-research questions. Hence, first, a theoretical analysis will be given, and then the method to derive the memory size in practice will be given. Thirdly, the method that was used to get inference time measurements was given and finally, the method for predictions was given.

In Chapter 4, the results from the experiments set in Chapter 3 will be given with the only exception of the theoretical analysis, which is completely given in Chapter 3.

Finally, in Chapter 5, first, the sub-research questions will be answered after which the main research questions will be answered. Further, there will be a discussion section in which a critical look is taken at general remarks about the research, and ultimately a recommendation for future work is given.

1-4 Implementation, software and hardware used.

In this section, the implementation software and the hardware used are discussed to clarify the context of this thesis further.

Hardware The hardware used is a laptop with an Intel Ultra 155H processor and 16 gigabytes of LPDDR5X RAM. This laptop is used as Demi used a laptop as well in her thesis and this makes it possible to get a comparison using similar hardware. It is also expected that the same bottlenecks are present in this system as in hers.

Software

PyTorch PyTorch is a very easy-to-use machine learning library, which enables the user to easily make CNN layers and other layer. For all tests its float32 format was, which is the default for CPU. This was done to get consistent results [4].

Tensorly-Torch Tensorly-Torch is a library based on both PyTorch [3] and Tensorly [23]. PyTorch is discussed above and Tensorly is a tensor decomposition library. Tensorly-Torch combines these features by providing tensor decomposed CNN layers. This library is used as it was used in both Demi's and Frederik's thesis and is therefore in part chosen for all tests to be able to make a comparison with their results [3].

scikit-learn Scikit-learn is a machine learning library. It can be used for classification, preprocessing and model selection. In this thesis, it is used to perform linear regression. This can be seen in Section 3-4 and Section 4-3 [32].

permetrics Permeterics is used to evaluate the quality of the linear regression model. It is an existing library which contains a lot of evaluation metrics such as Root Mean Squared Error (RMSE), Variance accounted for (VAF) and many others [39, 40].

Aside from this other software tools were explored, but not used to compose the final result.

Pytorch tensor decompositions This library made by Jacob Gildenblat. This software is not used for tests or any result that contributed to the output. Despite this, it gave a useful insight into possible implementations for the CP-CNN [11].

Intel Vtune Intel Vtune is a software evaluation tool. The aim of using this software was to obtain gross metrics which may be able to investigate or give metrics on how much energy was used. This software was used for the regular CNN and the CP-CNN. Unfortunately, this software did not prove to be stable for the CP-CNN. It had very few cache misses, and the CP-CNN seems to perform a lot better than the regular CNN compared with the tests without Intel Vtune. Therefore, it was theorized that the software interfered with the memory speed/CPU speed too much to get valid and useful insights [18].

1-5 List of Symbols

In this section, two lists of symbols will be given. Table 1-1 shows the symbols regarding the tensor math. Table 1-2 show the symbols regarding the CNNs.

Table 1-1: Symbols regarding tensors

a	a scalar or a vector
A	a matrix
$\underline{A}$	a tensor
a^n	A factor vector with n representing the nth factor matrix.
a_i, $A_{i,j}$, $\underline{A}_{i,j,k,...}$	A vector, matrix or tensor with i, j and k representing indices. The tensor can have any number of indices.

Table 1-2: Symbols regarding CNNs

T	Number of output channels
S	Number of input channels
d_n	nth spatial dimension of the kernel/filter
O	Output image
I	Input image
K	Kernel
K^y	Kernel of a decomposed CNN with y indicating what part of the convolution this kernel represents.

Chapter 2

Background

2-1 Tensor decompositions

2-1-1 Definition

Tensors or multidimensional arrays are a way to represent data. This is done in zero, one or multiple dimensions. A tensor is a generalized mathematical concept of a scalar (0th order tensor), vector (1st order tensor) or matrix (2nd order tensor), and of higher order versions.

Figure 2-1 gives an overview of a representation using diagrams. The circle represents an entity (i.e. one object) and a line represents an index. A line is only drawn if the index does have more than one element. Consequently, the scalar in Figure 2-1 does not have any lines, the vector has one, the matrix has two and the tensor has three. However, the tensor could have any amount of indices (lines). [9]

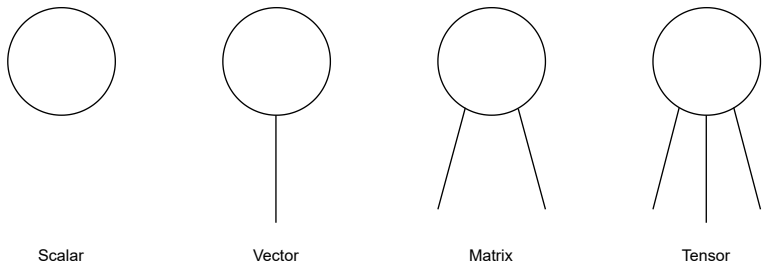


Figure 2-1: Tensor diagram overview. Each line represents an (array) index. [9]

2-1-2 Preliminaries

In this subsection, some useful definitions on types tensors and operations are given.

Tensor types

Diagonal tensor A diagonal tensor is a tensor with only non-zero entries on its superdiagonal. This means that tensor $\underline{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ is diagonal if and only if $\underline{X}_{i_1,i_2,\ldots,i_N} = 0 \ \forall i_1, i_2, \ldots i_N \notin \{i_1 = i_2 = \cdots = i_N\}$ [22].

Tensor operations

Inner product The inner product (also known as dot product) is the sum of the product entries. This can only be done for tensors of equal size. For tensors $\underline{A}, \underline{B} \in \mathbb{R}^{I_1 \times I_2 \times \ldots I_N}$, the dot product is given by $\underline{AB} = \sum_{i_1=0}^{I_1-1} \sum_{i_2=0}^{I_2-1} \cdots \sum_{i_N=0}^{I_N-1} \underline{A}_{i_1,i_2,\ldots i_N} \underline{B}_{i_1,i_2,\ldots i_N}$ [22].

Outer product The outer product of a set of three vectors $a \in \mathbb{R}^I$, $b \in \mathbb{R}^J$ and $c \in \mathbb{R}^K$ results in a 3rd-order tensor $\underline{X} \in \mathbb{R}^{I \times J \times K}$. The entries of $\underline{X}$ are determined by $x_{i,j,k} = a_i b_j c_k$. This can be generalized such that for a set of N vectors an Nth order tensor is created. Note that for the 2nd order case the operation can also be written as $\underline{X} = ab^\top$, in which $b^\top$ is the transposed version of b.
Furthermore, this definition can be extended to the the outer product of two tensors. In this case, for any tensor $\underline{A} \in \mathbb{R}^{I_1 \times I_2 \times \ldots I_N}$ and $\underline{B} \in \mathbb{R}^{J_1 \times J_2 \times \ldots J_M}$ an N+Mth tensor $\underline{X} \in \mathbb{R}^{I_1 \times I_2 \times \ldots I_N \times J_1 \times J_2 \times J_M}$ can be created by $\underline{X} = \underline{A} \circ \underline{B}$ [9, 22].

Mode-n matricization Mode-n matricization is denoted by $B = A_{<n>}$ for $A \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_n \times \cdots \times I_N}$ and $B \in \mathbb{R}^{I_n \times I_1 I_2 \ldots I_{n-1} I_{n+1} \ldots I_N}$. The n-mode matricization reorders the elements and indices of a tensor into a matrix. This reordering is necessary for some operations to be performed on the tensor. A diagram can be seen in Figure 2-2. [9, 22].

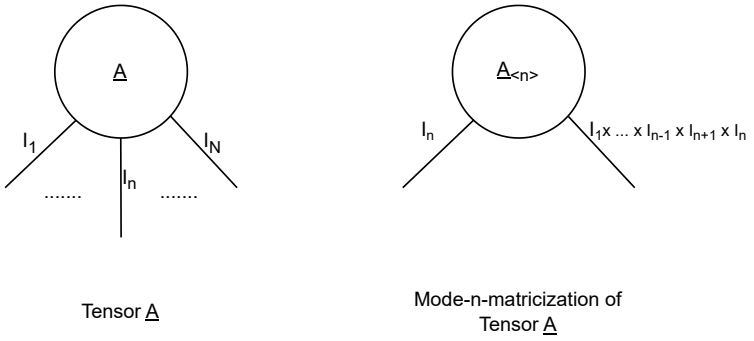


Figure 2-2: Tensor $\underline{A}$ is on the right and on the left is the mode-n-matricized version of tensor $\underline{A}$ [22]

Mode-n product The mode-n product is denoted by $\underline{C} = \underline{A} \times_n B$ for $A \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_{n-1} \times I_n \times I_{n+1} \times \cdots \times I_N}$, $B \in \mathbb{R}^{J \times I_n}$ and $C \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_{n-1} \times J \times I_{n+1} \times \cdots \times I_N}$. This is an operation between any tensor $\underline{A}$ and any matrix B. During this operation mode-n matricization is done on the nth mode of tensor $\underline{A}$. Then matrix multiplication is done on matrix B and tensor $\underline{A}_{<n>}$. The product of this multiplication is then $\underline{C}_{<n>}$, which is then reordered and reshaped to tensor $\underline{C} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_{n-1} \times J \times I_{n+1} \times \cdots \times I_N}$. This operation can also be expressed as $C_{<n>} = B\underline{A}_{<n>}$ [9, 22].

2-1-3 Canoncial Polyadic Decomposition (CPD)

Canoncial Polyadic Decomposition (CPD) (also known as CANDECOMP, PARAFAC or just CP decomposition) is a tensor decomposition, which decomposes an Nth-order tensor in a set of N factor matrices and a scaling coefficient tensor $\underline{\Lambda}$. Figure 2-3 shows the structure of the decomposition for a 4D tensor.

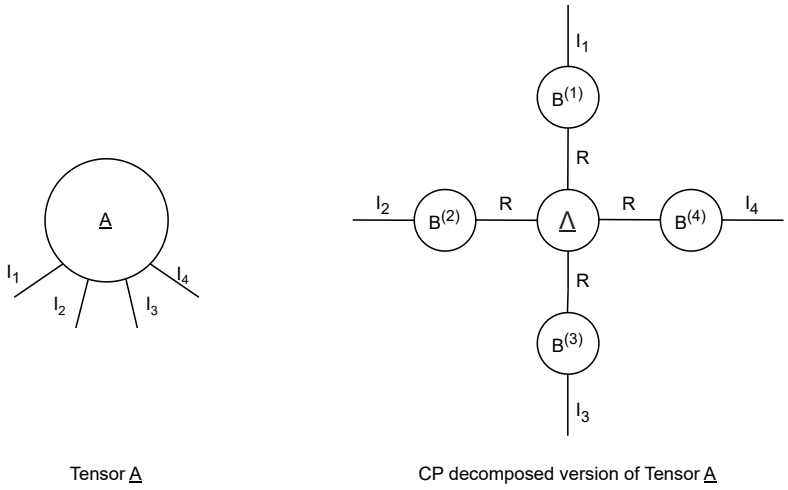


Figure 2-3: The image shows the network structure for a 4D-tensor decomposed using CPD [9]

The factor matrices are of the form $B^{(n)} \in \mathbb{R}^{I_n \times R}$. In this representation n represents the nth index of the original tensor. I_n has the length of nth index of the original tensor and R represents the rank of the decomposition. Each tensor can be decomposed using CPD and this solution is unique under some mild conditions. However, there is only one value for the rank for which this holds. Finding this optimal rank is an NP-hard problem. Some heuristics are known for lower-order tensors, but they do not provide any guarantee. This rank is optimal in the sense that when the rank is chosen lower than this optimal rank, then the CPD will give an approximation of the original tensor. When the rank is chosen higher than this optimal rank original tensor, the original tensor can be reconstructed, but the solution will no longer by unique. Additionally, extra storage is required but with no additional benefit [9, 14, 22].

Definition The CPD is written as shown in the Equation 2-1 in which R denotes the rank of the decomposition [9, 22]

$$\begin{aligned} \underline{A} &= \underline{\Lambda} \times_1 B^{(1)} \times_2 B^{(2)} \times_{...} \cdots \times_N B^{(N)} \\ &= \sum_{r=1}^{R} \lambda_r b_r^1 \circ b_r^2 \circ \ldots \circ b_r^N. \end{aligned} \tag{2-1}$$

In Equation 2-1, it can be seen that the rank forms an important part of the CPD structure. It together with the diagonal scaling tensor connects all the separate factor matrices. The

second line is another way of writing the same structure down. In this case, $\lambda_r \in \mathbb{R}^R$ is the vector of non-zero elements from the diagonal scalar tensor $\underline{\Lambda}$ and r gives the rth element. $b_r^n \in \mathbb{R}^{I_n}$ is the rth column of matrix $B^n \in \mathbb{R}^{I_n \times R}$ and b_r^n is therefore a vector. Since the scaling tensor $\underline{\Lambda}$ is diagonal, both approaches are equivalent.

To find the CPD, often an algorithm known as Alternating Least Squares (ALS), which is shown in Appendinx A-2 is used. Other algorithms do also exist such as Modified Alternating Least Squares (MALS), but ALS is most commonly used [22]. The explantation of the ALS algorithm is shown in Appendix A-2.

2-1-4 Tensor Train

The Tensor Train (TT) also known as Matrix Product State (MPS), consists of a structure of linked tensors. These tensor are called cores. Aside from the first and the last core, the cores are 3rd order tensors. The first and the last cores are 2nd order by some definitions. Others define them as 3rd order as well but in that case the first (or last) mode has length 1, which is equivalent to the 2nd order definition. The notation for the nth core is given by $G_n \in \mathbb{R}^{R_{n-1} \times I_n \times R_n}$. An overview in diagrams is shown in Figure 2-4 [9, 30].

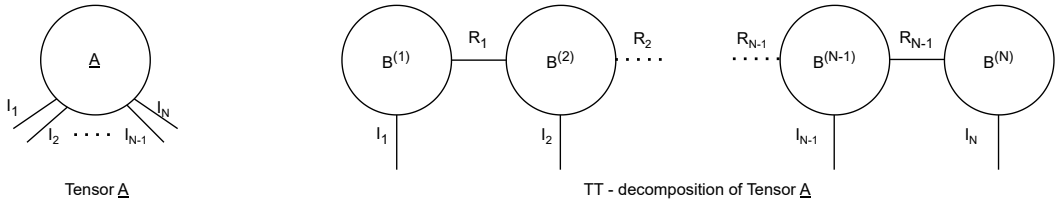


Figure 2-4: Tensor Diagram of the Tensor Train

Definition The Tensor Train decomposition is shown in Equation 2-2 [30]

$$\underline{A} = \sum_{r_0, r_1, \ldots, r_N} G_1(r_0, i_1, r_1) G_2(r_1, i_2, r_2) \ldots G_N(r_{N-1}, i_N, r_N), \quad \underline{A} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}. \tag{2-2}$$

From Figure 2-4 and Equation 2-2, it is visible that for the TT decomposition rank plays an important role as well, but the mechanism behind it is a little different than for CPD. For the TT decomposition, not all ranks have to be the same for all factor matrices. Additionally, the cores are not connected to a single scaling tensor, but instead, they are connected to each other. This gives the TT in some sense more flexibility compared to the CPD [9, 30, 34].

To decompose a tensor into the Tensor Train format, multiple techniques can be used of which the TT-SVD algorithm which is shown in Appendix A-3 is the most commonly used [9, 30, 34]. Aside from this algorithms using ALS and MALS have been proposed which can be more efficient. Also, they can update a Tensor Train in an iterative fashion which makes it useful for other purposes than decomposing. This is certainly the case when the tensor is already in TT format [16, 34, 37].

2-2 CNN

A Convolutional Neural Networks (CNN) is an Artificial Intelligence (AI) tool which are mostly used in image recognition. Here a brief explanation of the convolution operation is given. After which, some tweaking options of the convolution operation are given. These can also be used in CNNs themselves. Finally, a step is made from the convolution operation to a CNN.

2-2-1 Convolution

The convolution operation In this thesis, 2D convolution/2D CNNs will be used. However, to explain this operation more easily, 1D space is used for this explanation. This will later be expanded to 2D space. When talking about 2D discrete space, one often talks about images. For this 1D space, a sampled audio signal will be taken as an example.

The discrete time convolutional operation is shown in Equation 2-3. In this equation, $K \in \mathbb{R}^A$ denotes the kernel $I \in \mathbb{R}^{K+A-1}$ the input and $O \in \mathbb{R}^{(K)}$ the output. In this example the input and output image are both a set of discrete time samples. The kernel can be seen as a sliding window. Each output sample is constructed of a multiplication of a part of the input sample space with this sliding window. This gives that the length of the output is the length of the input plus the length of the kernel minus 1 (To see where the minus 1 comes form. Imagine sliding a kernel of length 1 in that case the input and output have the same size). In most cases the kernel is often much smaller than the input vector. The convolutional operation is given by Equation 2-3 [21]

$$O(k) = \sum_a K(a) * I(k-a). \tag{2-3}$$

The effect of the kernel in this context can be that kernel K gives an sound effect to the audio signal. The input in this context can be a guitar signal, the desired output could be a certain effect on the guitar sound. The values of the kernel will determine how the sound is altered.

This 1D-convolution function can easily be expanded to 2D space by introducing an extra spatial dimension. In this case, the spatial dimensions of K, O and I are expanded to $K \in \mathbb{R}^{A \times B}$, $O \in \mathbb{R}^{X \times Y}$ and $I \in \mathbb{R}^{(X+A-1)\times(Y+B-1)}$. This is shown in Equation 2-4 [21]

$$O(x,y) = \sum_a \sum_b K(a,b) * I(x-a, y-b). \tag{2-4}$$

2-2-2 Stride, padding and dilation

Stride, padding and dilation are tools to alter the convolution operation of Equation 2-3 and Equation 2-4. These operations can be used independently of one another and can also be used to change the CNN. Additionally, the stride, padding and dilation do not have to be the same in every spatial dimension.

Stride When using stride, one alters the steps size for which the kernel kernel slides. In the case discussed above, the step after x is $x+1$. When the stride increases, one is going to take bigger steps. For example, a stride of 2 gives $I(x-a)$, $I(x+2-a)$, etc. This is shown in Equation 2-5, where $S_x\,(S_y)$ represents the stride in the $X\,(Y)$ dimension. In the example, a stride of 2 is taken but can be any strictly positive integer. Using stride changes the shrinks the size of the output image to $O \in \mathbb{R}^{(X/S_x)\times(Y/S_y)}$, the size of the kernel and the input image remain unchanged. Note that $S_x\,(S_y)$ should be an integer devisor of $X\,(Y)$. The effects of stride will not be explored during this thesis. However, it may be mentioned in some contexts [21, 31]

$$O\,(x,y) = \sum_a \sum_b K(a,b) * I\,(S_x x - a, S_y y - b)\,. \tag{2-5}$$

Padding Padding is an extension of the input image. This can be done in multiple ways, e.g. by using zeros, the same value as the border of the image, or using a value from the other side of the image. By using padding, one can keep the dimension of the input image the same as the dimensions of the output image. By looking closely at Equation 2-3 and Equation 2-4, it can be seen that when the size of the kernel is larger than 1 (or 1×1), the output image is smaller than the input image. The amount of padding does not have to be the same in both spacial dimensions, but it is very common to do so. The dimensions for padding will be denoted as P_x and P_y for the padding in the X and Y spatial dimensions respectively. For padding $Px\,(P_y)$ the chosen padding pattern (zero, same ect.) is applied to both sides of the dimension. Therefore an $P_x\,(P_y)$ padded image $A \in \mathbb{R}^{X\times Y}$ will have padded dimensions $A \in \mathbb{R}^{(X+2P_x)\times(Y+2P_y)}$. This changes the dimensions of Equation 2-4 to $O \in \mathbb{R}^{(X+2P_x)\times(Y+2P_y)}$ and $I \in \mathbb{R}^{(X+A-1+2P_x)\times(Y+B-1+2P_y)}$. The dimensions of the kernel remain the same.

Additionally, padding can prevent loss of information at the boundary [21]. Padding will be used in this thesis to keep the input image equal to the output image.

Dilation When using dilation the kernel will be expanded. When dilation is used a number of $D_x\,(D_y)$ 0's is placed between each value of the kernel, increasing the effective area of the kernel without introducing new parameters. This is visible in Equation 2-6. In this example, the dilation is described by $D_x\,(D_y))$ for the $X\,(Y)$ spatial dimension. Dilation changes the size of the output image to $O \in \mathbb{R}^{(X-2D_x)\times(Y-2D_y)}$. In this thesis, the effects of dilation will not be investigated. However, since it is often mentioned in the context of CNNs, it is still explained as it may be mentioned in some contexts [13]

$$O\,(x,y) = \sum_a \sum_b K\,(a,b) * I\,(x - D_x a, y - D_y b)\,. \tag{2-6}$$

2-2-3 Convolutional layer

The convolution operation discussed in the former section forms the basis of a CNN-convolutional layer. However, in a CNN the kernel is often called filter. Additionally, there

are often multiple input images, output images and kernels which are stacked. Those multiple layers are called channels.

Convolution This stack of multiple input images gives the input of the convolution the dimensions $I \in \mathbb{R}^{S \times W \times H}$, where S is the number of input channels, H is the height of the input image and W is the width.

For the kernel, this gives dimensions $K \in \mathbb{R}^{T \times S \times d_1 \times d_2}$, in which T is the number of output channels, S is the number of input channels and d_1 and d_2 are the spacial dimensions of the kernel. Common sizes for the kernel in CNN layers are 1×1, 3×3, 5×5 and 7×7.

Finally, the output image has shape $O \in \mathbb{R}^{T \times H' \times W'}$ in which T represents the amount of channels in the output image, and H' and W' the height and width of the output images. The width and height of the output images differ from the dimensions of the height and width dimensions of the input channels in the same way the dimensions of the input and output size differed for the normal convolution operation presented in Section 2-2-1.

These extra dimensions for the kernel, input image and output image give the basis for the convolution operations. By doing the convolution as described in Equation 2-7, a forward pass of a layer of a CNN can be done. Equation 2-7 describes the complete mathematical operation for a convolutional layer [27]

$$O(t,x,y) = \sum_{a=0}^{d_1-1} \sum_{b=0}^{d_2-1} \sum_{s=1}^{S} K(a,b,s,t) * I(s, x-a, y-b) \quad \forall t \in T, \forall x \in H', \forall y \in W'. \quad (2\text{-}7)$$

In which $O \in \mathbb{R}^{T \times H' \times W'}$ is the output image, $K \in \mathbb{R}^{T \times S \times d_1 \times d_2}$ is the kernel, $I \in \mathbb{R}^{S \times H \times W}$ is the input image . Doing the convolutional operation this way is called fully connected, because all input images influence all output images [13, 21, 31]. An overview of the convolution is visible in Figure 2-5 on the left side.

Bias To complete the CNN layer a bias is often added to each convolutional layer. This bias has the same shape of the output image and can be calculated by $y = O + b$ where $Y, O, b \in \mathbb{R}^{T \times H' \times W'}$. While the bias is formally part of the convolutional layer, it is of lesser importance to this thesis as the operation is the same for all CNN layers [13, 21, 31]

2-2-4 Grouped CNN layers

Grouped CNNs (also called depthwise convolutions) are a less popular CNN option as they are not covered by every paper and survey. Reviews [1, 21] do not explain much about it. In the first review, they are mentioned in some fixed CNN networks. In the second work, they are not mentioned at all.

Fully connected or grouped In non-grouped convolutions (i.e. "normal" convolutions), every input influences every output. Therefore these layers are often referred to as fully connected. In grouped convolutions, the connections are only in part or even isolated, there are only

some connections between different input and output layers. Or in some cases, each input image generates a specific output image. Another way of looking at grouped convolutions is by representing them as N separate convolutions. For grouped convolutions the amount of groups must be a divisor of the input and the output channels. When N is the number of groups then the kernels are of the form $K^n \in \mathbb{R}^{T/N \times S/N \times d_1 \times d_2, \forall n \in N}$ [6, 25, 42].

These grouped layers are not the main focus of this thesis and will not be used separately. However, they are part of the Canonical Polyadic-Convolutional Neural Network (CP-CNN) network explained in the next section. In the CP-CNN, only the case where $N = T = S$ is used. Therefore only this case is shown in the shown in Figure 2-5.

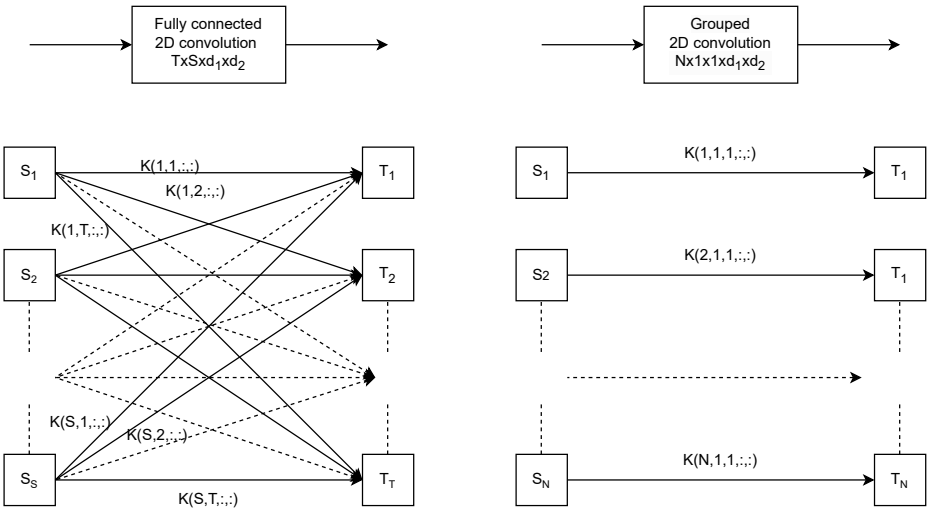


Figure 2-5: Overview of fully grouped and fully connected 2D convolution. Each square represents a channel of the input/output image $\left(S_n \in \mathbb{R}^{1 \times H \times W}\right)$ or $\left(T_n \in \mathbb{R}^{1 \times H' \times W'}\right)$. Each arrow represents a slice of the kernel realization $K \in \mathbb{R}^{1 \times 1 \times d_1 \times d_2}$ (fully connected) or $K^{slice} \in \mathbb{R}^{1 \times 1 \times 1 \times d_1 \times d_2}$ (grouped). The : index marks that all values are considered.

2-2-5 Other types of layer

A complete and effective CNN also consists of other types of layers besides the convolutional layers. These topics will be briefly touched on, but not further discussed, because the decomposed part CNNs only affects the convolutional layer. The convolutional part is the most computationally intense [21]. Therefore, these other layers are not in the interest of this thesis.

Pooling layers In a CNN there are also other types of layers such as pooling layers, which reduce the size of the image by adding parts of pixels together. This shrinks the input image drastically and thus reduces computational complexity [13, 21, 31].

Activation layers Furthermore, there are activation layers, which are non-linear functions that also change the values of the image. A commonly used activation function is the ReLu

function, which essentially uses function $O_{i,j,k} = max\left(I_{i,j,k}, 0\right)$ for $O, I \in \mathbb{R}^{I \times H' \times W'}$ in which O is the output of the function, I is the input of the function. The showed result is for a 2D-CNN as the operation acts on all channels I and for the whole spatial dimension [13].

2-3 Decomposed CNNs

This section discusses the decomposed CNN layers. These layers use a tensor decomposed kernel. Because of this, they can be used interchangeably because one can go from the regular form to the decomposed form and vice versa. There are two types of decomposed CNN layers. The CP-CNN which will be discussed first and is based on a CNN convolutional layer which is decomposed using the CPD tensor decomposition. The other decomposition that will be discussed is the Tensor Train Convolutional Neural Network (TT-CNN). This decomposition is based on the Tensor Train decomposition method discussed in Section 2-1.

2-3-1 Canonical Polyadic-Convolutional Neural Network

In the first part of this section, the derivation of the CP-CNN will be given the way it is implemented in Tensorly-Torch [3]. After this, some considerations regarding this choice will be given.

Derivation of CP-CNN model The first step in deriving a CP-CNN model is decomposing the kernel using the CPD tensor decomposition. This is shown in Equation 2-8 [27]

$$K\left(t, s, d_1, d_2\right) = \sum_{R} K^t\left(t, r\right) K^s\left(s, r\right) K^H\left(d_1, r\right) K^W\left(d_2, r\right). \tag{2-8}$$

In this equation, $K \in \mathbb{R}^{T \times S \times d_1 \times d_2}$ is the original kernel. $K^t \in \mathbb{R}^{T \times R}$, $K^S \in \mathbb{R}S \times R$, $K^H \in \mathbb{R}d_1 \times R$ and $K^W \in \mathbb{R}^{d_2 \times R}$ are the resulting factor matrices. This can then be inserted Equation 2-7, which results in Equation 2-9

$$\begin{aligned} O(t, x, y) = \sum_{R} K^t\left(t, r\right) \Bigg(\sum_{a=0}^{d_1 - 1} K^x\left(a, r\right) * \\ \left(\sum_{b=0}^{d_2 - 1} K^y\left(b, r\right) \left(\sum_{s=1}^{S} K^s\left(s, r\right) I(s, x-a, y-b) \right) \right) \Bigg) \quad \forall t \in T, \forall x \in X, \forall y \in Y \end{aligned} \tag{2-9}$$

where $O \in \mathbb{R}^{T \times H' \times W'}$ and $I \in \mathbb{R}^{S \times (X + d_1 - 1) \times (Y + d_2 - 1)}$ is the same as in Equation 2-7. Equation 2-9 shows that the convolution can be split into four smaller convolutions, starting with the far right sum and working towards the left. Each convolution will then give its own output image, which will then be the input image for the next convolution. It should be noted that all kernels have a dependency in R over which is summed in the final convolution. Therefore, the middle two kernels become a grouped convolution as those operations are isolated by this dependency in R.

Schematically this gives the convolution shown in Figure 2-6.

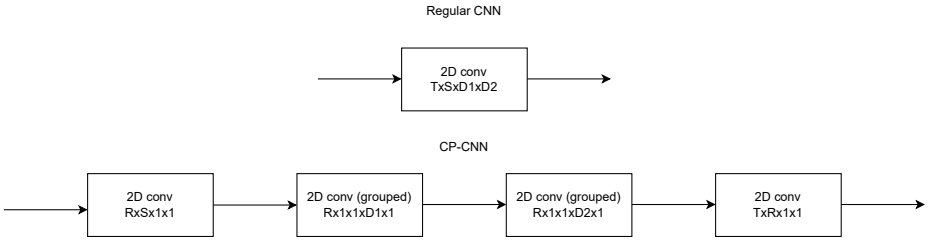


Figure 2-6: Schematic overview of regular CNN and CP-CNN. In which for each convolution the kernel is denoted as output channels x input channels x kernel dimension 1 x kernel dimension 2 ($T \times S \times d_1 \times d_2$)

Similar decomposition options Additionally, [25] mentions that for the first and last layer in the CP-CNN, tensor contraction leads to the same result as doing the 1x1 convolution. Despite the author being a developer of the Tensorly-Torch project [3], still, the convolutional function of PyTorch is used in the implementation. Secondly, the original paper describing the CP-CNN [27] omits the scalar diagonal tensor $\underline{\Lambda} \in \mathbb{R}^{R \times R \times R \times R}$. However, the Tensorly-Torch [3, 25] toolbox, which will be used later in this thesis, does by default use this diagonal tensor $\underline{\Lambda}$. For the tests, in this paper this $\underline{\Lambda}$ is omitted just as in the original paper [27] as the addition of $\underline{\Lambda}$ does not seem to influence the effectiveness of this decomposition, but does provide extra computational overhead.

Another option is presented in [5]. They propose to keep the filters of the kernel intact and only decompose the input and output channels of the original kernel. This provides one less step in the convolutional layer of the CNN and the filters' spatial dimensions are often small, which may pose a potential benefit.

2-3-2 Tensor Train Convolutional Neural Network

The TT-CNN uses the same concept as the CP-CNN in the sense that it also decomposes the kernel into four smaller convolutions. However, this time the convolutions are slightly different since the TT-CNN does not have the grouped convolutions. Additionally, for the TT-CNN the ranks are not all equal just as in the TT. In this section, first a mathematical derivation is given, which will be followed by some considerations specific to the TT-CNN.

Derivation The implementation Tensorly-Torch [3] for the TT-CNN is the one given by [10]. In this paper, the kernel is first permuted. Usually the kernel is represented by $K \in \mathbb{R}^{T \times S \times d_1 \times d_2}$ for a 2D-convolution. [10] permutes this to $K \in \mathbb{R}S \times d_1 \times d_2 \times T$. This is done because otherwise there exists a large inflexibility in ranks, as due to the nature of the tensor train the rank between j and d_1 cannot be larger than $d_1 * d_2$. Then the kernel is decomposed using the TT decomposition introduced in Section 2-1-4. After this, the shape of the kernel is as follows:

$$K(s, d_1, d_2, t) = \sum_{r1=0}^{R_1-1} \sum_{r_2=0}^{R_2-1} \sum_{r3=0}^{R3-1} K^S(s, r_1) K^H(r_1, d_1, R_2) K^W(r_2, d_2, r_3) K^T(r_3, t) \quad (2\text{-}10)$$

with $K \in \mathbb{R}^{S \times d_1 \times d_2 \times T}$, $K^S \in \mathbb{R}^{S \times R_1}$, $K^H \in \mathbb{R}^{R_1 \times d_1 \times R_2}$, $K^W \in \mathbb{R}^{R_2 \times d_2 \times R3}$ and $K^T \in \mathbb{R}^{R_3 \times T}$.

The kernel can be inserted in Equation 2-7. The results can be written by changing the order of summation and this results in Equation 2-11. To increase readability, every new convolution has been put on a new line

$$
\begin{aligned}
O(x,y,t) = \sum_{r_3=0}^{R_3} & K^t(r_3,t) * \\
& \left(\sum_{y-\delta}^{y+\delta} K^y(r_2,d_2,r_3) * \right. \\
& \left(\sum_{x-\delta}^{x+\delta} K^x(r_1,d_1,r_2) * \right. \\
& \left. \left(\sum_{s=1}^{S} K^S(s,r_1)\, I(i,j,s) \right) \right) \Bigg) \quad \forall t \in T, \forall x \in X, \forall y \in Y.
\end{aligned}
\tag{2-11}
$$

In Equation 2-11 one can see that the convolution can be split into multiple smaller convolutions. An overview of the resulting CNN can also be seen in Figure 2-7.

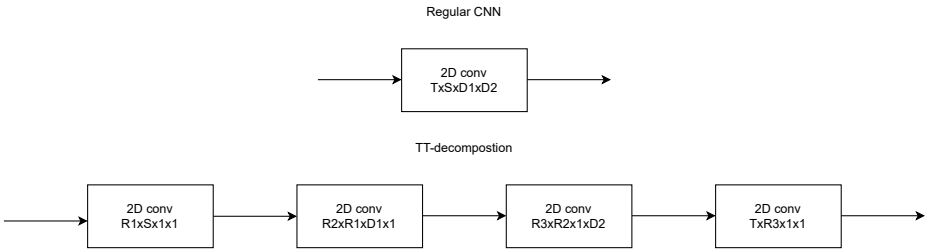


Figure 2-7: A schematic overview of the TT-CNN compared with the regular CNN. In which for each convolution the kernel is denoted as output channels x input channels x kernel dimension 1 x kernel dimension 2 ($T \times S \times d_1 \times d_2$).

Other implementation [29] proposed an alternative method. This method relies on creating a matrix-matrix multiplication of the CNN problem. Then it uses the TT-matrix format to create a tensor of a specific shape from both the kernel and the input tensor. Additionally, they provide a method to train the model with this kernel structure. A disadvantage of this method is that the input tensor needs to be reshaped in a specific order for each layer. Aside from this they also mention a performance advantage.

2-3-3 Conclusive

Both the CP-CNN and the TT-CNN differ from the regular CNN. Both have their kernels decomposed and use four separate convolutions. Hence, they split one convolution into 4 smaller convolutions. With this, they provide a compression of the kernel. However, this

comes at the cost of extra memory as the new kernels have extra in-between results, which will be further discussed in the theoretical analysis in Section 3-1. An overview of the three different kinds of kernels is shown in Figure 2-8.

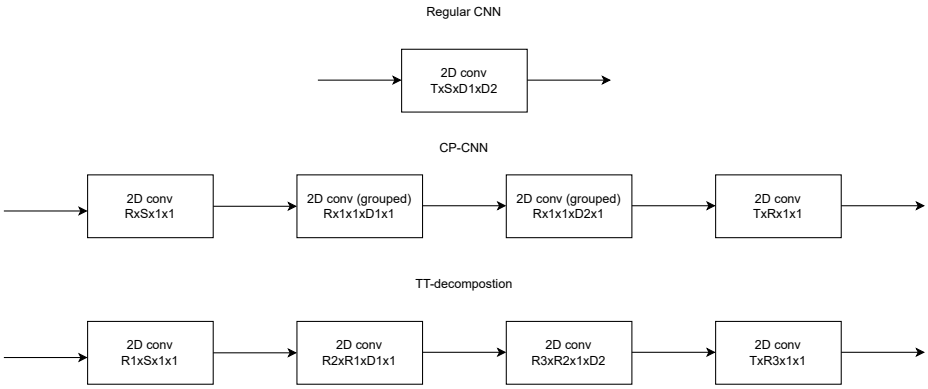


Figure 2-8: Overview of the three different CNN types. In this overview, each kernel is denoted as output channels x input channels x kernel dimension 1 x kernel dimension 2 ($T \times S \times d_1 \times d_2$).

2-4 Linear algebra in computing

In this section, a short introduction is given in linear algebra operations on the computer. This is broken down into two parts. First, a short insight is given in matrix operations and mainly, which things should be considered. Secondly, an introduction is given how one can do large linear algebraic operations optimally on every device.

2-4-1 Matrix operations

Performing linear algebraic operations on a computer may not be as easy as one thinks at first thought. Several things have to be considered to perform a linear algebraic operation fast and efficiently.

Memory Layout For doing matrix and tensor operations efficiently, not only the exact memory layout needs to be considered but also the memory hierarchy. In a computer, storage is often divided into several layers, of which the hard disk and the RAM are the most well-known. However, in the CPU there is also a tiny bit of memory called the cache. In modern multicore computers, even the cache consists of multiple layers. In general, the closer the memory is to the CPU, the faster and smaller the memory, see Figure 2-9. One can often store only tiny parts of a matrix or tensor in the cache. Therefore, it is most optimal to use that bit of data once it is there and to avoid situations in which you have to unnecessarily load data from a lower level multiple times, as this will impact performance. To this end, matrix operations are sometimes divided into smaller problems to use locality. Hence, optimizing the algorithms for this purpose may improve performance [12].

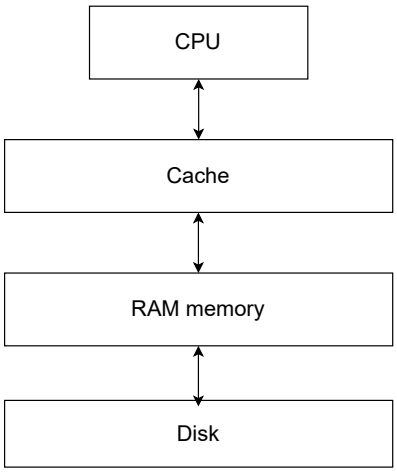


Figure 2-9: The memory hierarchy of a computer.

SIMD instructions Aside from optimizing an algorithm for optimal memory usage, an algorithm may sometimes also be optimized for the kind of operations CPUs do, as some CPUs allow for vectorized operations. These are called Singe Instruction Multiple Data (SIMD) instructions and perform the same instruction on multiple data elements at the same time. This does require the memory already to be to load the operation efficiently [12, 26].

Multi-core processors In multi-core processors, some extra factors need to be considered. For instance, if one wants to use all cores, there arises extra overhead from making sure the load over all cores is balanced, ensuring that data is not accessed simultaneously and that if a core has updated a part all other cores will get the updated piece of memory. This will lead to new problems and challenges when making linear algebraic algorithms [12].

Summarizing To do linear algebraic operations on a computer optimally, one needs to take use of algorithmic and data structural properties into account. There is not a single best solution for doing all linear algebraic operations. The performance depends heavily on the hardware used. Therefore, to get the most performance out of every computer and to have portable code, two packages called Basic Linear Algebra Subprograms (BLAS) and LAPACK were developed [12].

2-4-2 BLAS and LAPACK

While originating from the end of last century. BLAS, LAPACK and other similar packages still find their way into modern computing. Often these packages, whether written in FORTRAN or C, are still used as a backbone for many computationally intense scientific programming problems. However, they are often called by high-level languages with which the user interacts. Today complete packages exist consisting of implementations of these subprograms. Some are closed-sourced solutions available from different vendors (e.g. Intel (Intel

Math Kernel Library (IntelMKL)) [19], AMD (AMD Optimizing CPU Libraries (AOCL)) [2]), but also open-source implementations are available (e.g. OpenBlas). For Intel processors, the IntelMKL solution is often used as it is deemed most efficient. However, this is not necessarily the case for every operation [43]. Even today, new implementations are being developed to further improve performance for new computer architectures. In new implementations, fault tolerance is taken more into consideration as well [8, 43].

Chapter 3

Methodology

This chapter will discuss the methods used to experiment. It will be split into three sections. The first section discusses how the actual memory usage is acquired based on the theoretically required amount of memory. The second section discusses how the time data is collected. The final chapter will discuss how a model can be made that estimates the duration of inference needed for each type of layer.

3-1 Determining the theoretical amount of memory used

In this section, an analysis is made of the amount of memory and the number of Multiply-Accumulate (MAC) operations required for each type of CNN. First, the regular CNN is discussed, after which the two decomposed CNN types are discussed. In this analysis, the term memory elements is used whereas in computers one often speaks about the number of bits and/or bytes. To convert the memory elements to bits or bytes one has to multiply the number of memory elements by the number of bytes one value takes. A common value for the amount of bytes per element is four. This also corresponds to the amount of bytes for the float32 format, introduced in Section 1-4. This is the amount of bytes per floating point value with single precision.

3-1-1 Regular CNN

This subsection discusses the amount of memory and number of MAC operations required for a regular Convolutional Neural Networks (CNN). Both depend on the number of input channels (S), the number of output channels (T) and the size of the kernel d_n. Additionally, the size of the output images depends on the stride, padding and dilation, which are explained in Section 2-2.

To calculate the amount of memory needed we break the problem down into three parts: The input image, the kernel and the output image, which are given by Equation 3-1a, 3-1b and 3-1c

$$M_{regular\_input} = S * W * H \tag{3-1a}$$
$$M_{regular\_kernel} = S * T * d_1 * d_2 \tag{3-1b}$$
$$M_{regular\_output} = T * W' * H' \tag{3-1c}$$

in which $M_{regular\_input}$ gives the number of memory elements for the input image, $M_{regular\_kernel}$ gives the number of memory elements for the kernel and $M_{regular\_output}$ gives the number of memory elements for the output image.

Finally, the number of MAC operations required is given by the number of pixels in the kernel times the number of output pixels. This is because, for each output pixel, a kernel size of MAC operations has to be taken. This is given by Equation 3-2

$$O_{regular} = S * T * d_1 * d_2 * W' * H' \tag{3-2}$$

where $O_{regular}$ gives the amount of MAC operations for the regular CNN.

3-1-2 CP-CNN

For Canonical Polyadic-Convolutional Neural Network (CP-CNN) there are more steps to cover as one regular CNN layer decomposes into four CNN layers when using CP-CNN. Additionally, the CP-CNN implementation used by Tensorly uses two grouped CNN layers, which are explained in Section 2-3. In this subsection, first, the rank will be determined by filling out the compression ratio C. The compression ratio c is the number of elements of the new kernel divided by the amount of elements of the original kernel. After this, the total number of memory elements and MAC operations are determined for each of the four layers, which were further elaborated on in Section 2-3-1.

Rank The Tensorly-Torch implementation uses the Tensorly [23] method of finding the rank R. There are two options to give the rank. Either the user itself gives the rank or one can use the following method to achieve a certain compression (c). To use the latter method the user gives the desired compression (e.g.$c = 0.1$ or 10x compression). Equation 3-3 can be used to calculate the compression. When rewritten to Equation 3-4, the rank can be deduced. In this equation, I_n denotes the indices of the original kernel and N denotes the number of indices. In the case of the CP-CNN, the latter is always four.

There are two options to obtain the rank. First the user itself gives the rank. The second method is that the user gives a certain compression c, which is the kernel of the CP-CNN dided by the number of kernel parameters of the original CNN. Using this method 10x compression gives $c = 0.1$. To achieve this desired compression the method of Tensorly [23] is used. Equation 3-3 shows the formula used to calculate the compression from an existing decomposed CP-CNN kernel. Equation 3-4 can be used to determine the rank R of the CP-CNN. However, since the rank R of a CP-CNN must be an integer the R is rounded to the nearest integer. In case this results in a rank R of 0. This is truncated to 1 as R should be non-zero, since there are no parameters when R is zero

$$c = \frac{R * (S + T + d_1 + d_2)}{S * T * d_1 * d_2} \tag{3-3}$$

$$R = \frac{c * S * T * d_1 * d_2}{(S + T + d_1 + d_2)}. \tag{3-4}$$

Once the rank has been determined, the number of memory elements and MAC operations can be determined.

Layer 1 The first layer is a CNN layer without grouping. The number of input channels is equal to the number of input channels for the regular CNN. The amount of output channels is equal to the rank. Finally, the kernel is a 1x1 kernel. This gives the number of memory operations described in Equation 3-5 and the number of MAC operations given in Equation 3-6.

$$M_{CP\_L_1\_input} = S * H * W \tag{3-5a}$$
$$M_{CP\_L_1\_kernel} = S * R * 1 * 1 \tag{3-5b}$$
$$M_{CP\_L_1\_output} = R * H * W \tag{3-5c}$$

where $M_{CP\_L_1\_input}$ gives the amount of memory elements for the input image, $M_{CP\_L_1\_kernel}$ gives the number of memory elements for the kernel and $M_{CP\_L_1\_output}$ gives the number of memory elements for the output image.

$$MAC_{CP\_L_1} = S * R * 1 * 1 * H * W \tag{3-6}$$

where $MAC_{CP\_L_1}$ gives the amount of MAC operations required for this layer.

Layer 2 The second layer is a grouped CNN layer, in the Tensorly-Torch implementation. Further on this CNN is standard. The CNN has d input channels, R output channels, and a kernel of size d_1 by 1. As already mentioned this CNN is grouped and uses R groups. This means that this layer becomes a normal 2D convolution if the rank is 1. The size of the input, kernel and output image is given in Equation 3-7 and the amount of operations is given in Equation 3-8.

$$M_{CP\_L_2\_input} = R * H * W \tag{3-7a}$$
$$M_{CP\_L_2\_kernel} = 1 * R * d_1 * 1 \tag{3-7b}$$
$$M_{CP\_L_2\_output} = R * H' * W \tag{3-7c}$$

where $M_{CP\_L_2\_input}$ is the number of memory elements needed for the input image, $M_{CP\_L_2\_kernel}$ is the number of memory elements required for the kernel and $M_{CP\_L_2\_output}$ is the number of memory elements needed for the output image.

$$O_{CP\_L_2} = 1 * R * d * 1 * H * W' \tag{3-8}$$

where $O_{CP\_L_2}$ is the amount of MAC operations needed to do a forward pass with this layer.

Layer 3 The third layer is almost the same as the second layer. Only in this layer, the kernel operation will be done horizontally instead of vertically. Also for this CNN the number of input channels is 1, the number of output channels is R and the kernel is 1 by d_2. This gives the number of memory elements by Equation 3-9 and the number of operations by Equation 3-10.

$$M_{CP\_L_3\_input} = R * H' * W \tag{3-9a}$$
$$M_{CP\_L_3\_kernel} = 1 * R * 1 * d_2 \tag{3-9b}$$
$$M_{CP\_L_3\_output} = R * H' * W' \tag{3-9c}$$

where $M_{CP\_L_3\_input}$ gives the amount of memory elements the input images needs, $M_{CP\_L_3\_kernel}$ gives the the number of memory elements the kernel needs and $M_{CP\_L_3\_output}$ gives the amount of memory elements the output image needs.

$$O_{CP\_L_3} = 1 * R * d * 1 * W' * H \tag{3-10}$$

where $O_{CP\_L_3}$ gives the number of MAC operations that are required to do a forward pass with this layer.

Layer 4 The final layer is again a regular layer. This layer convolutes the output of layer three to the same $T * H' * W'$ images as you would get from the regular CNN. This is just like the first CP-CNN layer a convolution with a 1x1 kernel. Additionally, it has R input channels and T output channels. This gives the number of memory elements and MAC operations given in Equation 3-11 and Equation 3-12 respectively.

$$M_{CP\_L_4\_input} = R * H' * W' \tag{3-11a}$$
$$M_{CP\_L_4\_kernel} = R * T * 1 * 1 \tag{3-11b}$$
$$M_{CP\_L_4\_output} = T * H' * W' \tag{3-11c}$$

were $M_{CP\_L_4\_input}$ gives the number of memory elements the input images needs, $M_{CP\_L_4\_kernel}$ gives the the number of memory elements the kernel needs and $M_{CP\_L_4\_output}$ gives the amount of memory elements the output image needs for this layer.

$$O_{CP\_L_4} = R * T * 1 * 1 * H' * W' \tag{3-12}$$

where $O_{CP\_L_4}$ gives the number of MAC operations that are required to do a forward pass with this layer.

3-1-3 TT-CNN

The Tensor Train Convolutional Neural Network has quite some similarities with the CP-CNN, but just like the normal tensor train decomposition it does have different ranks for the different indices and instead of the two grouped layers in the middle, it has normal 2D-convolutions. This was further elaborated on in Section 2-3-2.

Rank Just as the CP-CNN, the ranks for the Tensor Train Convolutional Neural Network (TT-CNN) can be given in two different ways, either by the user giving all the ranks or by the user giving a compression ratio which will be approximated. Just as with the CP-CNN, the compression ratio c is defined as the number of elements in the TT-CNN kernel divided by the number of elements in the original kernel. To approximate the rank for the TT-CNN, Tensorly-Torch uses the method from Tensorly [23, 24], which is explained below.

First, all ranks are decomposed into a general part which is the same for all ranks and an individual part. For instance, R_1 becomes $r_1 * R$. Then, the individual parts are defined as follows: $r_n = (I_n + I_{n+1})/2 \quad \forall n \in \{1, 2, \ldots, N-1\}$. This gives Equation 3-13 for the total amount of TT parameters. Finally, the kernel of the TT-CNN $K \in \mathbb{R}^{S \times d_1 \times d_2 \times T}$ (see Section 2-3-2) can be filled in for I_1, I_2, I_3 and I_4. Then $R_n \quad \forall n \in \{1, 2, \ldots, N-1\}$ can be calculated such that the total amount of parameters is compression ratio (c) times the original amount of parameters. This can be done by solving Equation 3-14 for R and retrieving the original ranks. Note that when the original ranks are retrieved each rank R_n will be rounded to the nearest integer. In case, one of the ranks is calculated as 0. This will be changed to a 1 as the ranks for the Tensor Train (TT) decomposition should be strictly positive.

$$I_1R_1 + R_1I_2R_2 + R_2I_3R_3 + R_3I_4 = I_1(r_1R) + (r_1R)I_2(r_2 * R) + (r_2R)I_3(r_3R3) + (r_R)I_4 \quad (3\text{-}13)$$

$$(r_1 d_1 r_2 + r_2 d_2 r_3)\, R^2 + (S r_1 + r_3 T) R = c * (S T d_1 d_2) \quad (3\text{-}14)$$

Layer 1 The first layer is a normal 2D CNN layer. The first layer has the same number of input channels as the original CNN (S), the number of output channels is equal to the first rank (R_1) and the kernel has size 1x1. This gives the amount of memory given in Equation 3-15 and the number of MAC operations as given in Equation 3-16.

$$M_{TT\_L_1\_input} = S * H * W \quad (3\text{-}15a)$$
$$M_{TT\_L_1\_kernel} = S * R_1 * 1 * 1 \quad (3\text{-}15b)$$
$$M_{TT\_L_1\_output} = R_1 * H * W \quad (3\text{-}15c)$$

where $M_{TT\_L_1\_input}$ gives the number of memory elements the input images needs, $M_{TT\_L_1\_kernel}$ gives the the number of memory elements the kernel needs and $M_{TT\_L_1\_output}$ gives the amount of memory elements the output image needs for this layer.

$$O_{TT\_L_1} = S * R_1 * 1 * 1 * H * W \quad (3\text{-}16)$$

where $O_{TT\_L_1}$ gives the number of MAC operations that are required to do a forward pass with this layer.

Layer 2 The second layer is also a normal 2D CNN layer. This differs from the CP-CNN where this is a grouped layer. The number of input channels is equal to rank 1 (R_1), the number of output channels is equal to rank 2 (R_2) and the kernel has shape $d_1 * 1$. This will do the vertical part of the kernel. The amount of memory elements and the MAC operations are given by Equation 3-17 and Equation 3-18 respectively.

$$M_{TT\_L_2\_input} = R_1 * H * W \tag{3-17a}$$
$$M_{TT\_L_2\_kernel} = R_1 * R_2 * d_1 * 1 \tag{3-17b}$$
$$M_{TT\_L_2\_output} = R_1 * H' * W \tag{3-17c}$$

where $M_{TT\_L_2\_input}$ gives the number of memory elements the input images needs, $M_{TT\_L_2\_kernel}$ gives the the number of memory elements the kernel needs and $M_{TT\_L_2\_output}$ gives the amount of memory elements the output image needs for this layer.

$$O_{TT\_L_2} = R_1 * R_2 * d_1 * 1 * H' * W \tag{3-18}$$

where $O_{TT\_L_2}$ gives the number of MAC operations that are required to do a forward pass with this layer.

Layer 3 Layer three is quite similar to layer 2, but just as with the CP-CNN now the horizontal action will be done instead of the vertical one. For this layer the number of input channels is R_2, the number of output channels is (R_3) and the kernel is $1 * d_2$. This gives the number of memory elements and MAC operations as given in Equation 3-19 and Equation 3-20 respectively.

$$M_{TT\_L_3\_input} = R_2 * H' * W \tag{3-19a}$$
$$M_{TT\_L_3\_kernel} = R_2 * R_3 * 1 * d_2 \tag{3-19b}$$
$$M_{TT\_L_3\_output} = R_3 * H' * W' \tag{3-19c}$$

where $M_{TT\_L_3\_input}$ gives the number of memory elements the input images needs, $M_{TT\_L_3\_kernel}$ gives the the number of memory elements the kernel needs and $M_{TT\_L_3\_output}$ gives the amount of memory elements the output image needs for this layer.

$$O_{TT\_L_3} = R_2 * R_3 * 1 * d_2 * H' * W' \tag{3-20}$$

where $O_{TT\_L_3}$ gives the number of MAC operations that are required to do a forward pass with this layer.

Layer 4 Finally, in the last layer, the rank in the last output image will be transformed to the number of output channels of the original system, just like the last layer in the CP-CNN did. For this layer the amount of input channels is R_3, the amount of output channels is T and the kernel has size 1x1. This gives the number of memory elements and MAC operations as given in Equation 3-21 and Equation 3-22 respectively.

$$M_{TT\_L_4\_input} = R_3 * H' * W' \tag{3-21a}$$
$$M_{TT\_L_4\_input} = R_3 * T * 1 * 1 \tag{3-21b}$$
$$M_{TT\_L_4\_input} = T * H' * W' \tag{3-21c}$$

where $M_{TT\_L_4\_input}$ gives the number of memory elements the input images needs, $M_{TT\_L_4\_kernel}$ gives the the number of memory elements the kernel needs and $M_{TT\_L_4\_output}$ gives the amount of memory elements the output image needs for this layer.

$$O_{TT\_L_4} = R_2 * R_3 * 1 * d_2 * H' * W' \tag{3-22}$$

where $O_{TT\_L_4}$ gives the number of MAC operations that are required to do a forward pass with this layer.

3-1-4 Implications for these different CNNs

In the last three subsections the amount of memory and MAC operations is given for each layer of the regular CNN, CP-CNN and the TT-CNN. It should be highlighted that to calculate the total amount of memory required, the output image of one layer is the input image of the layer after it and does not need to be taken twice. An overview of the amount of memory elements and MAC operations is given in Table 3-1.

Table 3-1: Table showing the number of memory elements from different origins for different CNN's

Memory elements of	Reglular CNN	CP-CNN	TT-CNN
Input image	$S * H * W$	$S * H * W$	$S * H * W$
Output image	$T * H' * W'$	$T * H' * W'$	$T * H' * W'$
In between images	-	$R * (HW + H'W + H'W')$	$(R_1HW + R_2H'W + R_3H'W')$
Kernel	$S * T * d_1 * d_2$	$R * (S + T + d_1 + d_2)$	$R_1 * S + R_2(R_1d_1 + R_3d_3) + R_4T$
MAC operations	Reglular CNN	CP-CNN	TT-CNN
Total for all layers	$S * T * d_1 * d_2$	$R * (S + T + d_1 + d_2)$	$R_1 * S + R_2(R_1d_1 + R_3d_3) + R_4T$

From Table 3-1, it can be seen that if for the CP-CNN and TT-CNN, when R (or R_1, R_2 and R_3) are in the order of magnitude of S and T the memory for the total images increase by a factor of upto 2.5x, which increases the amount of memory required, despite the number of

parameters decreases. Additionally, Table 3-1 shows the total number of MAC operations is equal to the total number of memory elements needed to store the kernel.

To analyse whether this could be problematic. The number of memory elements is worked out for four different systems.

All systems will have a kernel of size 3×3, a padding of 1. Further, there are no tweaks with stride and dilation, so they are also set to 1. This configuration with the kernel, stride, padding and dilation gives an output image equal to the input image. The four systems will differ for number of input channels, output channels and image size. An overview is given in Table 3-2. In this table, all four systems have a compression c of 0.1. This means that for each system the number of MAC operations for the CP-CNN and TT-CNN is reduced by 10 times, since the amount of MAC operations is equal to the size of the kernel as shown in Table 3-1.

Table 3-2: Four configurations for the theoretical analysis

	Input channels	Output channel	Image size	Compression ratio c
Configuration 1	16	16	16x16	0.1
Configuration 2	256	256	16x16	0.1
Configuration 3	16	16	256x256	0.1
Configuration 4	256	256	256x256	0.1

Configuration 1 For the first configuration, the CP-CNN and TT-CNN were calculated using Equation 3-4 and Equation 3-14 respectively. This gave a $R = 6$ for the CP-CNN and $R_1 = 5, R_2 = 2$ and $R_3 = 5$ for the TT-CNN. The resulting amount of memory elements are shown in Table 3-3. This table shows that the amount of memory elements required is the lowest for the CNN and that CP-CNN and TT-CNN require 1.2x and 1.1x more memory elements respectively.

Table 3-3: Result for theoretical analysis of configuration 1

	Regular CNN	CP-CNN	TT-CNN
Input image + Output image	8,192	8,192	8,192
Kernel size	2304	228	220
In between images	-	4,608	3,072
Total	10,496	13,028	11,484

Configuration 2 The second configuration has a large kernel size than the first option furthermore the input and output image size are the same. For the same compression ratio of $c = 0.1$, this does give a larger rank for both the CP-CNN and the TT-CNN. The ranks are $R = 114$ for the CP-CNN and $R_1 = 112, R_2 = 3$ and $R_3 = 112$. The amount of memory elements can be seen in Table 3-4 . This Table shows that the regular CNN uses the most memory elements and that the CP-CNN and TT-CNN require 0.4x and 0.3x that amount of memory elements respectively.

Table 3-4: Result for theoretical analysis of configuration 2

	Regular CNN	CP-CNN	TT-CNN
Input image + Output image	131,072	131,072	131,072
Kernel size	589,824	59,052	59,360
In between images	-	87,552	58,112
Total	720,896	277,676	248,544

Configuration 3 The third is a bit the other way around. It does have the same kernel size as configuration 1 but for this measurement the image size has been increased from 16×16 to 256×256. For the same compression ratio of $c = 0.1$, gives the same rank as configuration 1 for both decomposed CNNs. The amount of memory elements can be seen in Table 3-5. Table 3-5 show that the CNN requires the least number of memory elements and that the CP-CNN and TT-CNN require 1.6x and 1.4x that amount of memory.

Table 3-5: Result for theoretical analysis of configuration 3

	Regular CNN	CP-CNN	TT-CNN
Input image + Output image	2,097,152	2,097,152	2,097,152
Kernel size	2,304	228	220
In between images	-	1,179,648	786,432
Total	2,099,456	3,277,028	2,883,804

Configuration 4 The final combination is a mix of configuration 2 and 3, because it has the larger kernel size from configuration 2 and the larger image size from configuration 3. Since the kernel size and compression ratio are the same as in configuration 2, the ranks are the same as in configuration 2 for both the CP-CNN and the TT-CNN. The amount of memory elements can be seen in Table 3-6. Table 3-6 shows that the regular CNN uses the least number of memory elements and that the CP-CNN and TT-CNN use 1.6x and 1.4 the amount of memory elements respectively.

Table 3-6: Result for theoretical analysis of configuration 4

	Regular CNN	CP-CNN	TT-CNN
Input image + Output image	33,554,432	33,554,432	33,554,432
Kernel size	589,824	59,052	59,360
In between images	-	22,413,312	14,876,672
Total	34,144,256	56,026,796	48,490,464

Summarizing It can be seen that the CP-CNN and TT-CNN both needed more memory for the configurations where the image size compared to the kernel is as large or larger. Only for the case where the kernel size is substantially larger than the input and output image, the regular CNN consumed more memory. This gives a trade off because in the cases where the regular CNN consumes more memory it should perform worse as it also does 10x more MAC operations. For the other cases, no conclusion can be based on this analysis, because despite

the regular CNN having a lot more operations, memory handling also takes time can cause delays within a computing system. Therefore, an real life experiment has to be setup. This will be done in the following two sections.

3-2 Determine the actual amount of memory used.

To make a fair comparison between the decomposed and non-decomposed CNNs, first, the exact amount of memory has to be determined. Therefore, a model will be created which determines the amount of memory used for every possible layer layout. This gives further insight into a possible gap between theory and practice, which gives further insight into the problem. Additionally, later an experiment will be done on the prediction of the inference time as well. For that prediction model, it is also necessary to determine the memory consumption correctly as well. The model which will be made in this model will only be tested for a stride and dilation of 1 and for padding such that the input and output image have the same dimensions.

For this experiment, the default PyTorch settings are taken regarding models. This means that PyTorch may opt to choose the IntelMKL backend since that is available on the test PC. Additionally, the MKL may also have been enabled in the CPU case of Demi's thesis [7] as there is no mention in code of the MKL turning off and she was running her CPU experiment on a laptop with an Intel CPU as well.

3-2-1 Tools used to determine the actual model and make an estimation of the realistically used amount of memory

To make a model some tools are used. First, an overview is made of these tools, and after this, a summary is given on how these tools could lead to a model predicting the allocated memory of the different CNN models.

PyTorch sourcecode First, PyTorch source code was analysed to get some hints. Not much information was found about the native setting of PyTorch but a file was found which determined whether Intel MKL was used [4, aten/src/ATen/native/Convolution.cpp]. This was a first start of the model and raised the question of how much memory is required in these models.

Using the standard output of the built-in profiler Since the documentation of PyTorch gives no information about actual memory allocations, using this was not an option. Despite this, PyTorch does provide tools which can be used to measure memory. The first and most obvious seems to be the built-in profiler. However, this tool did provide the desired information, because it gives table where each row is a function in the PyTorch environment. Then for each row the self-memory and the memory is shown. In this, the self-memory is the difference between the allocated and deallocated memory during that function. The memory that is given, is the difference between all allocated and deallocated during the function call and all child functions.

There are three big caveats with using this technique that could lead to incomplete results:

1. If one were to do all calculations in one function without passing on anything to child functions. No memory would be detected as the sum of allocated and deallocated memory is zero. Maybe the only measured result is the output image when that is returned. Hence, in this way allocated memory could remain undetected.

2. If one were to use a child function or program which is not from PyTorch itself. Memory could be allocated and deallocated outside the scope of the profiler and thus remain undetected. (This occurs when using an external linear algebra package such as intel MKL)

3. If memory is allocated in a grandchild function, the memory could be measured multiple times. This results in too much memory being detected. (Because the child function reports a positive sum and the function itself reports positive memory.)

All three options seemed to occur in the PyTorch system. Therefore, this method is not considered a valid option and the data it outputs is considered meaningless to our scope.

Using the chrome trace of the profiler The built-in PyTorch profiler also has a chrome trace function. This function is intended so that the user can see an overview of all memory and function calls using Google Chrome. The function exports a JSON file which exports all memory data and the function calls with the corresponding time. This function seems to report all the memory allocations including an address and a moment when the memory is allocated. Additionally, it did seem to include all memory that is allocated by software outside the PyTorch scope i.e. IntelMKL. An example of a user_annotation and a memory_event is shown in Listing 3.1. The first array is the user annotation which is annotated by the cat and has the given name input image. "ts" shows the time the user_annotation was made. Finally, "dur" shows the duration. In this way the different parts (e.g. inference layer 1, inference layer 2, creating the input image) of the convolution are separated.

The second part shows an array of a memory event, which can be seen by the name "[memory]". For this event it also shows the starting time, based on this time the function in which it is called can be determined as the function stack with start times and duration is also available (not shown in this snippet, but is similar to the user_annotation). Additionally, by the amount of bytes it can be determined whether it is an allocation or deallocation (deallocations show an negative number of bytes) and how much memory is allocation. Using this file, an accurate measurement for the total allocated memory can be realised.


```
{
    "ph": "X", "cat": "user_annotation", "name": "Input_image", "pid"
       : 14280, "tid": 29644,
    "ts": 1842655666342.350, "dur": 3916.000,
    "args": {
      "External id": 1,"Record function id": 0, "Ev Idx": 0
    }
}
{
    "ph": "i", "cat": "cpu_instant_event", "s": "t", "name": "[memory
       ]",
    "pid": 14280, "tid": 29644,
```

```
        "ts": 1842655693015.550,
        "args": {
        "Device Type": 0, "Ev Idx": 179, "Bytes": 972800, "Device Id":
            -1, "Addr": 3261479256320, "Total Allocated": 5031872, "Total
            Reserved": 0
    }
}
```

Listing 3.1: Listing of a user label and memory event

The Intel MKLDNN documentation It was noted that there were two kinds of calculation events: native and MKL. Therefore, also the intel MKLDNN documentation was consulted as it may provide information on how it performs linear algebraic calculations and thus the memory that is allocated in doing so [17]

Intel MKL verbosity Aside from the intel MKL documentation, it could also be set to verbose. With this function, some information could be obtained with the documentation. It shows clearly which indices gets truncated to the Singe Instruction Multiple Data (SIMD) width, which is 8 in most cases. This mainly shows when memory is reordered and how the indicees are truncated.

3-2-2 Creating the model

Creating the model will be done in 3 steps. This first step is to determine which backend is used. This will be done by using the source code of PyTorch [4]. A deterministic way of deciding which backend is used is found in the code. Secondly, for each type of kernel and each backend (grouped or not grouped), it will be determined, how much memory is required.

Determining the use of native PyTorch or Intel MKL Determining the actual memory consumption was done by applying the following code, which is taken from here [4].

```
(input.device().is_cpu() &&
 input.scalar_type() == kFloat && // only on CPU Float Tensors
 // For 1x1 filters, MKLDNN is faster than THNN when multi-threaded
    ,
 // but THNN is faster when single-threaded.
 (is_strided() || is_dilated() || at::symint::size<T>(input, 0) >=
    16 ||
  at::symint::size<T>(weight, -1) != 1 || at::symint::size<T>(
     weight, -2) != 1 || at::get_num_threads() > 1) &&
 (groups > 1
  || (at::symint::size<T>(weight, -1) > 3 && at::symint::size<T>(
     weight, -2) > 3)
  || at::symint::size<T>(input, 0) > 1
  || at::symint::size<T>(input, 0)*at::symint::size<T>(input, 1)*at
     ::symint::size<T>(input, 2)*at::symint::size<T>(input, 3) >
     20480) // for some case, native is faster
```

```
);
```

Listing 3.2: PyTorch decision tree on whether MKLDNN is suitable

If this function output to true, then IntelMKL is used. Since there is no other eligible backend available for the test PC. Therefore in all other cases, the native PyTorch backend is used. Preceding this piece of code, there are some additional conditions which state that:

- Intel MKL should be enabled
- Intel MKL is not used, when the convolution is transposed and the output padding is big.
- Intel MKL should be used for the bf16 and float16 type, if the calculations are done on a Central Processing Unit (CPU)
- The input is an Intel_mkldnn tensor type.

For this thesis, these constraints do not seem to be relevant because Intel MKL is always enabled, the float32 numeric type is used and no transposed convolutions are used (The latter is also not further discussed, but it is the reversed version of the normal convolution). Finally, all inputs are given as a PyTorch tensor and not stored as an Intel_MKLDNN tensor.

The rest of the decision tree (which is for this thesis the relevant part) is summarized in Figure 3-1. This tree is also implemented in the code to estimate the expected amount of memory.

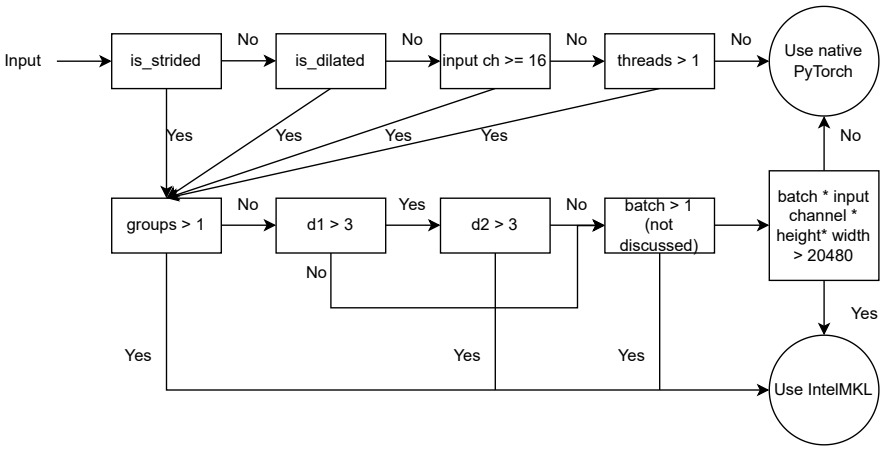


Figure 3-1: Decision tree of choosing PyTorch native or IntelMKL as mathematical backend

Determining the memory used by native PyTorch kernels. To get the amount of memory for the PyTorch native kernels. The chrome trace was used with the theoretical amount of memory required for the size of the input image, the output image and the kernel. There it was found that for the native PyTorch backend only the size of the output image was required for the amount of memory the output image takes. This is only valid for non-grouped kernels because those are the only ones that were tested as grouped kernels are always handled by the IntelMKL as can be seen by the decision tree given in Figure 3-1.

Determining the memory used by Intel MKL kernels For Intel MKL, there are two different types grouped and non-grouped. Both grouped and non-grouped kernels use SIMD instructions. As described in Section 2-4, these operations act on a data vector instead of a single element. For many home desktops or laptops having the AVX2 instruction set extention, this is 8 operations. Some newer home desktop CPUs and High performance computing (HPC) CPUs, which have the AVX512 instruction set extension, also have widths of 16. To make use of these instructions the memory has to be reordered physically. Therefore IntelMKL contributes to a lot more memory operations. Despite this, it is more time-efficient to use these operations, since the gains of doing 8 operations concurrently outweigh the loss in time to do the memory operations [26].

For MKL kernels and native PyTorch kernels the model is shown in Figure 3-2.

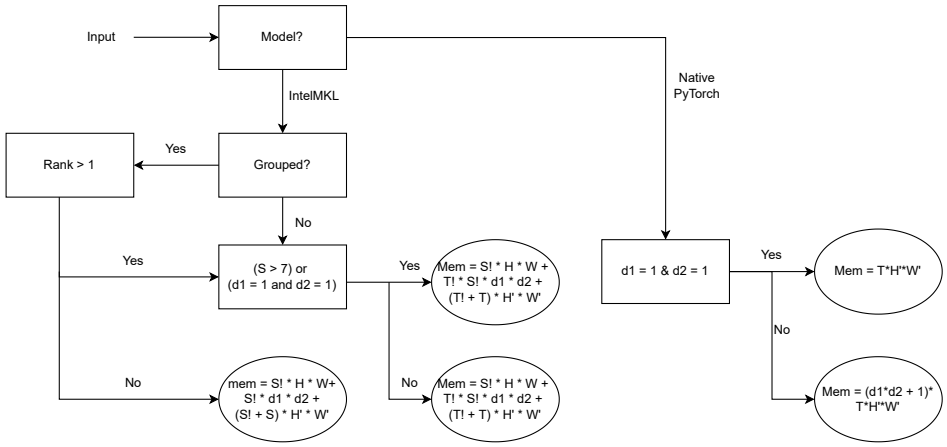


Figure 3-2: Overview of memory consumption per layer. Note that this image goes once for the regular layer and once for each layer of CP-CNN and TT-CNN, see Figure 2-8. A question mark means that the value should be rounded upwards to to SIMD size (often 8 or 16)

Conclusive In this subsection a model has been created to estimate the actual amount of memory a regular CNN, CP-CNN and TT-CNN consume. In the next section, a verification experiment will be setup to verifiy this model.

3-2-3 Verifying the memory model

To validate the model which gives the expected total amount of memory. Two different tests are set up. Which are given below. The first test aims to see how the system behaves under varying size systems. The goal of the second test is to see whether the system behaves properly under different kernel sizes. Both factors influence the amount of memory and may influence the behaviour of the actual amount of memory.

Different system sizes For the first test different system sizes were taken. The test consisted of four different input and output channel combinations and four different image sizes, which are shown in Table 3-7. This test was mainly executed to see the different influences for

the native PyTorch and IntelMKL models. The outcome of this test will be the ratio of the measured total amount of allocated memory and the expected total amount of allocated memory.

Table 3-7: Overview of different measurements for different system sizes. The input channels will be set equal to the output channels

Input channels = output channels	[4,16,128,512]	Stride	[1,1]
Image sizes	[[4,4], [16,16], [128,128], [512,512]]	Padding	[1,1]
Kernel size	[3,3]	Dilation	[1,1]

Different kernel sizes The test of kernel sizes was conducted to see whether different kernel sizes would influence the model. This is to ensure that any given kernel size would be modelled correctly. This test is conducted for four different kernel sizes ranging from 1 to 7. The padding is each time set such that the input image has the same dimensions as the output image i.e. the padding is (kernel size - 1)/2. Finally, for the input image size, number of input channels and number of output channels four different sets were made. These are summarized in Table 3-8.

Table 3-8: Overview of different measurements for different kernel sizes and padding. The input channels will be set equal to the output channels

Input channels = output channels	[4,16,128,512]	Stride	[1,1]
Image sizes	[128,128]	Padding	[(kernel size-1)/2, (kernel size-1)/2]
Kernel size	[[1,1], [3,3], [5,5], [7,7]]	Dilation	[1,1]

3-3 Collect time measurements

To link time and memory consumption, a test to measure inference time and memory consumption was also carried out. In the first part of this section, it will be explained how the measurement was done exactly. In the second part, it will be explained what exact measurements are performed and why.

3-3-1 Method to do inference time measurements

To measure the inference time, a measurement runner was created which did a couple of things.

Preliminaries This whole system was just one inference of one of the three models. It could be run for multiple epochs which essentially means that this whole procedure is run multiple times.

The first thing that is done is setting the no_grad function in PyTorch. This is done because for the scope of this thesis, the focus lies solely on inference time and taking gradients for backtracking is thus not deemed necessary.

For this procedure, the MKLDNN package could be set to verbose. This offers insightful information on the sizes and the copies that were made, but this functionality was off during inference time measurements to make sure that it does not distort the measurements.

Contrarily, the PyTorch built-in profiler was active, because the PyTorch profiler was used to measure all memory allocations and deallocations. This is done with the same chrome trace method described in Section 3-2. For these tests, it was such that it only records memory (setting which has to be set to true) besides the function stack. Other functionalities such as record shapes or modules are turned off to distort the measurements as little as possible. This is because all measurements which are not necessary but still performed also take time for the CPU and do take time.

Finally, for each epoch, before a measurement is made, an input image with a flag for the profiler is created such that the size of the input image can be traced back. Additionally, a deep copy of the model is made i.e. PyTorch is forced to make a copy of the model. This is done to also track the size of the model. The model was created beforehand (as it is outside the scope of an inference time test) but by just making a copy it could also be recorded by the PyTorch trace profiler and thus made visible in analysis afterwards and because it is made before the start of time measurement it should not affect the inference time.

Measuring inference time Finally, the inference time is measured. This is done by starting a counter before performing one inference of the chosen model and measuring the time after one measurement is taken. This is done by inputting the input image into the model. After this is finished the time is again registered and the start time is subtracted from the end time. This results in the wall time it took to perform the inference. Additionally, the input, model and output are explicitly deallocated. This is done so that it can be assured all memory allocated is also deallocated and thus all memory is accounted for.

After inference Finally, after each inference, the chrome trace is exported by the profiler just as described in Section 3-2-1. This data is saved. Once all epochs are done for a certain set of hyperparameters, all Chrome traces are then further processed to store only the parts that may be interesting for further analysis.

3-3-2 Inference time measurements

To see whether a relation between inference time and memory exists, it was chosen to do many measurements with a smaller epoch. A benefit of this method is that it eases the analysis on whether a measurement may be an outlier, because when multiple measurements with similar hyperparameters also have a high inference time one can take a closer look at a particular set. A disadvantage of this method is that regardless of the amount of post-processing, the chances of having an outlier increase because fewer measurements are done. Despite this disadvantage, it is still chosen to take a large hyperparameter set as one could always do

extra tests in case and it might provide more insight into the relation between memory and inference time.

An overview of the hyperparameters can be seen in Table 3-9.

Table 3-9: Overview of hyperparameters for the inference time measurement

In channels	[4, 8, 16, 32, 64, 96, 128, 192, 256]	Kernel size [Hight, Width]	[[3,3]]
Outchannels	[4, 8, 16, 32, 64, 96, 128, 192, 256]	Padding	[1]
Images [Hight, Width]	[[4,4], [8.8], [16,16], [32,32], [64,64], [96,96], [128,128], [192,192], [256,256]	Stride	[1]
Compression ratios (only for TT/CP CNN)	[0.01, 0.05, 0.1, 0.25, 0.5, 1.0]	Dilation	[1]

Table 3-9 shows a lot of different sets. For sets with multiple possibilities, all possible combinations of these sets are run. Additionally, it can be seen that 256 is chosen as the maximum for the number of input channels, output channels and image size. This was because, for the CP-CNN decompositions, the largest model would get close to 3 GB of RAM. It was chosen not to do tests consuming more memory, because if to much memory would be allocated on the PC it would start paging, i.e. expanding the total amount of RAM by taking hard disk space. Since this is not the situation that was desired to be measured, it was decided that this was the limit.

Furthermore, the kernel and padding were taken as a constant. This was done because a kernel size of 3x3 with padding seemed to be a popular kernel size. Additionally, the thesis of Demi [7] and the expected RAM vs measured RAM model from Section 3-2 showed that the amount of RAM for larger kernels increased very fast. Similar to setting the maximum number of input channels, output channels and image size to 256, it was chosen not to increase the kernel size because of reaching the maximum RAM of the laptop, penalizing large models unfairly.

Additionally, no different values were chosen for stride and dilation. This is done because these are more advanced CNN techniques and are not necessary to for every CNN. Additionally, they bring extra complexity making harder it to analysis the results fairly. In the discussion, Section 5-2, the possible implications will be hypothesised.

Finally, the compression ratio where chosen such that a broad spectrum of options could be analysed. It should be understood that for some very small systems, some of these compression ratios generate the same experiment. This is because in small systems (input channels max 16, output channels max 16) the kernel can only be compressed until the (for CP-CNN) or all (for TT-CNN) rank(s) are 1 because when the rank is zero there is nothing left of the kernel. Furthermore, because the ranks are rounded to whole integers (one cannot take a rank with decimals) two different compression ratios are approximated best by the same rank. For larger systems, this is less of a problem.

3-4 Model the amount of time required for each layer

In this section, it will be explained, which models are chosen for to make a regression model. Additionally, it is explained how the models were created. In this section for each regression model, two fits will be done. The first fit will be for the dataset containing the data points from the regular CNN dataset. The other will be made from both the CP-CNN and TT-CNN this was done because from earlier tests it seemed that they could be united into one regression model. This option was thus explored as this would make a more general model.

3-4-1 Data preprocessing

The data from the former section (Section 3-3) will also be used in this section because it seems to be useful for this test as well. However, it will be preprocessed before the model is trained.

Filtering compression 0.5 **and** 1.0 For the CP-CNN and TT-CNN models the higher compressions of 0.5 and 1.0 are filtered out because they do not seem to provide usefulness in a real scenario as they use considerably more memory than the regular CNN model and use (almost) the same amount of operations. This could distort the result of interest.

Taking the median The first step will be to take the median of the 10 epochs from the former set. This is done mostly because it filters for outliers and it should take a sample half of the noise distribution of the sampling. This is brought up because when measuring inference time one cannot assume zero mean Gaussian noise or even Gaussian noise. This is because speed-ups do not exist, but a slowdown due to another process in the PC is still possible. Therefore, by taking the mean of the 10 samples one would include a possible bias of a single sample which experienced a heavy slowdown, while there is no process to mitigate these effects. This could degenerate the quality of the model.

A 80/20 training-validation split Another reason to include one inference time value per hyperparameter set is that when making a training-validation split a set of hyperparameters does not occur in both sets. The training-validation split is made so that the quality of the model can be assessed on values it is not trained on, improving the validation quality.

3-4-2 Regression models that will be fitted

Four different regression models will be fitted for the regular CNN dataset and the CP/TT dataset. The four different models are described here below.

Linear: Memory only The first model is a linear regression model and would only fit the memory. This was done to check whether the system would be memory-bound as when it is memory-bound it would scale with the inference duration time. For this model, the expected memory is determined in Section 3-2. The predictor function is given by Equation 3-23

$$\text{inf. time} = a * \text{mem.} + b \tag{3-23}$$

Linear: Operations only The second model is similar to the former model. In this model, only the theoretical number of MAC operations are taken. These are determined in Section 3-1. For this, the theoretical number of MAC operations is taken as the sole input. The predictor function is given in Equation 3-24.

$$\text{inf. time} = a * \text{MAC} + b \tag{3-24}$$

Linear: Memory and MAC operations After the first test was done, further improvements of the model would be given by a linear model consisting of both the expected amount of memory and the theoretical number of MAC operations. The regression model that is fitted for the predictor function is given in Equation 3-25.

$$\text{inf. time} = a * \text{MAC} + b * \text{mem.} + c \tag{3-25}$$

Quadratic: Memory and MAC operations Finally, a quadratic model was given. This is an extension to a linear polynomial containing both MAC operations and expected memory. This polynomial also adds a second degree and a cross term consisting of the number of MAC operations times the amount of expected memory. The predictor function can be seen in Equation 3-26.

$$\text{inf. time} = a * \text{MAC} + b * \text{mem.} + c * \text{MAC} * \text{mem.} + d * \text{MAC}^2 + e * \text{mem.}^2 + f \tag{3-26}$$

3-4-3 Validation criteria

To quantify the quality of the models, the quality of the models will be determined by three prediction criteria:

Root Mean Squared Error (RMSE) The RMSE is the root of the sum of all errors squared. This metric has the same unit as the value that is predicted. The lower the RMSE, the lower the errors. Hence, a lower RMSE indicates a better model.

Variance accounted for (VAF) The VAF indicates what part of the variance of the predicted model is present in the original model. A 100% score indicates that the prediction is perfect. The worst obtainable score is $-\inf$ indicating that the results are random [35, 39, 40].

R squared In this thesis the term R squared is used for the Coefficient of Determination (COD). This metric indicates something about the predicated variance compared to the total variance [35].

Chapter 4

Results

This chapter will discuss the results of the last three sections of the former chapter. First, the model for the amount of actual memory necessary to run the Convolutional Neural Networks (CNN)s will be discussed. In the second section, the inference time will be measured and, in the last section, the results of the predictive model using linear regression will be discussed.

4-1 Experiment 1: Verifying the memory allocation model

In this section, the model which approximates the expected amount of memory in a real scenario will be tested.

4-1-1 Verifying the actual memory usage

To verify that the memory usage could be estimated. The tests were conducted as described in Section 3-2 to verify whether the amount of memory that would happen in practice could later be verified.

Different system sizes For the first test, many parameters were altered simultaneously. For this test the input channels were equal to the amount of output channels for this parameter there was a set of four values. Additionally, for the image size four different values were taken. The results of this test are shown in Figure 4-1.

Figure 4-1 shows that for all these points the expected amount of memory is within 5%. This means that most of the memory can be explained by the model. It shows some error for the system with image size 4×4. This is probably because, for these kinds of measurements, all numeric values are quite small (in the order of bytes to kilobytes), which means that it is susceptible to small overheads. For the larger models, it shows very little, this is because the dots overlap in 1, which means that the expected value is as large as the measured memory value.

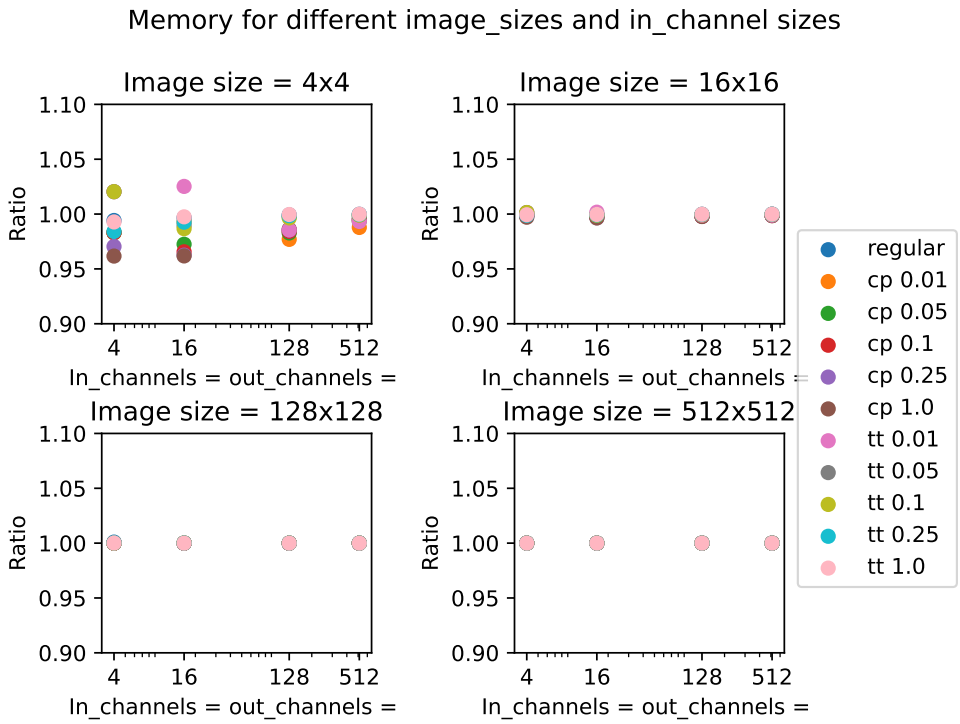


Figure 4-1: Plot showing the ratio of the expected memory size divided by the measured memory size for different systems.
In these systems the number of input channels is equal to the number of output channels
In the legend, regular/tt/cp define CNN-type used. The number defines the compression ratio c used.

Different kernel sizes Secondly the model is shown for different kernel sizes. The padding is adjusted such that the input image of the model has the same shape as the output image. For this model the image size was fixed to 128×128. The number of input and output channels was kept equal but varied. The results for this test can be seen in Figure 4-2.

Figure 4-2 shows that there is not too much deviation for this model. Only when the image kernel is small and even then it remains within the 5% range. Therefore it is assumed that the model is also correct for different kernel sizes and all memory can be predicted with a certain accuracy.

Conclusive It can be seen that the model is correct up to 5%. Most errors occur when the images are small (the 4x4 case). This indicates that not all memory is exactly captured, but a very good indication of the actual memory usage. Therefore, it is assumed that the model captures most if not all of the most imported sources for memory usage.

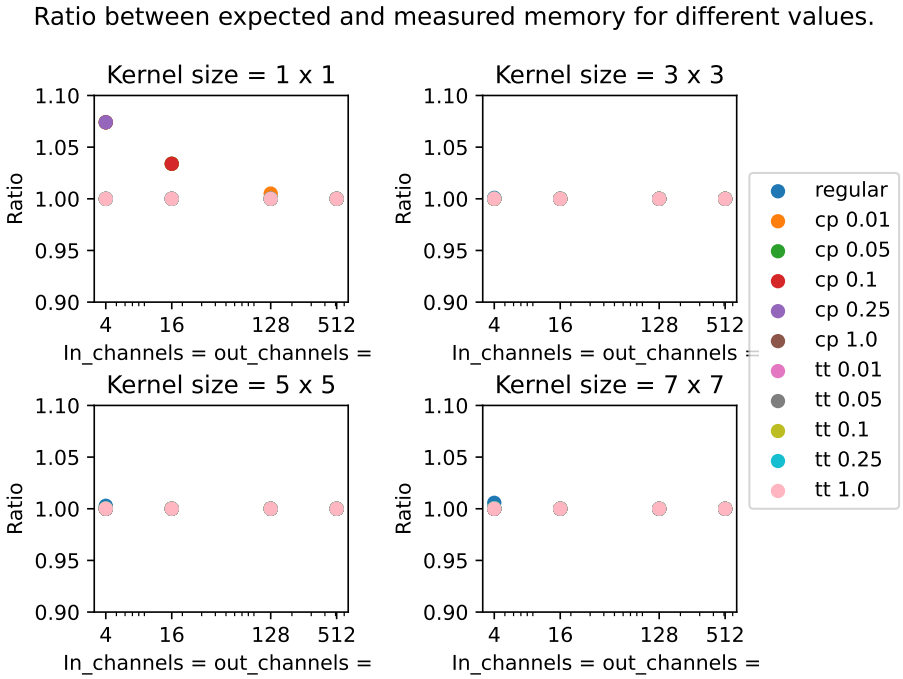


Figure 4-2: Plot showing the ratio of the expected memory size divided by the measured memory size for different systems.
In these systems the number of input channels is equal to the number of output channels
In the legend, regular/tt/cp define CNN-type used. The number defines the compression ratio c used.

4-1-2 Comparing actual memory usage to theoretical memory usage

In this final section a comparison is made between the theoretical analysis and the measurements. This is done in order to see how well the actual amount of memory can be predicted using the theoretical analysis. Figure 4-3 (on next page) shows the ratio of the expected amount of memory divided by the theoretical amount of memory. The figure shows that the point are consistently above 1, which means that actual memory usage is consistently higher than the amount which was calculated theoretically using the method of Section 3-1.

Conclusive The theoretical amount of memory calculated using the method of Section 3-1 is too optimistic as the real memory consumption is consistently the memory theorised.

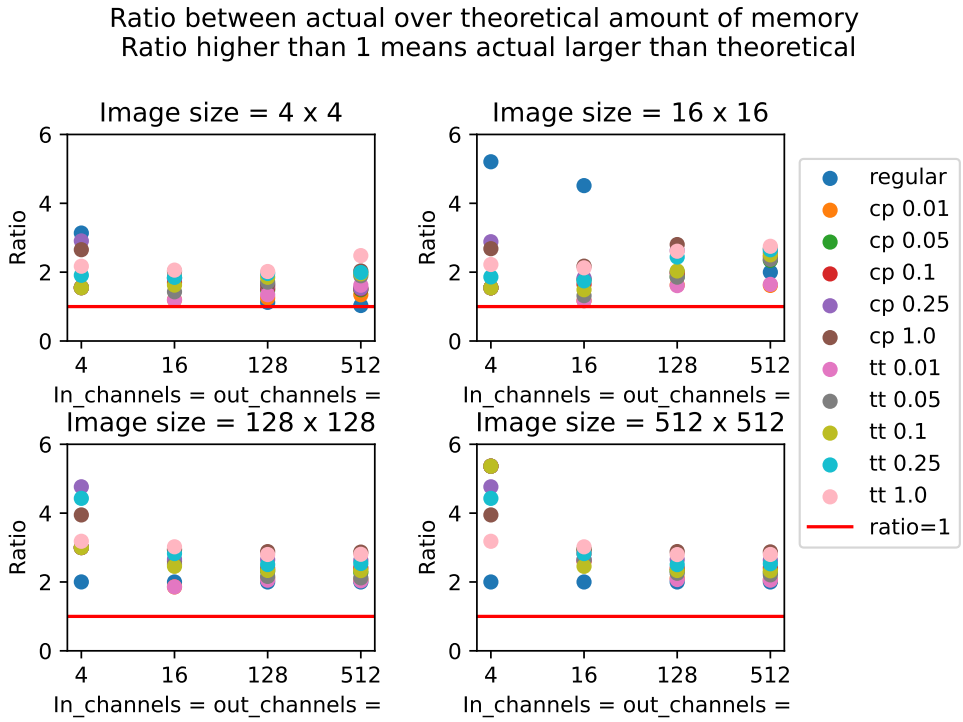


Figure 4-3: Ratio between actual and theoretical memory usage
In the legend, regular/tt/cp define CNN-type used. The number defines the compression ratio c used.

4-2 Experiment 2: Measure the inference time

In this chapter, the inference time will be measured. To do this the test described in Section 3-3 will be done. In these tests, many measurements were taken with different numbers of input channels, output channels, and image sizes.

4-2-1 Comparing memory directly to inference time

To get an insight into the data the plot of Figure 4-4 (which is shown on page 46) was created. In this plot four subplots are visible. All four plots show the results for the regular CNN and the results of the Tensor Train Convolutional Neural Network (TT-CNN) and Canonical Polyadic-Convolutional Neural Network (CP-CNN) with a compression of c in $\{0.01, 0.05, 0.1, 0.25\}$. Despite c in $\{0.5, 1.0\}$ being also measured, they are left out because they performed badly concerning inference time, which was expected. They both do not seem useful in a practical because they do not compress (much), using a lot more memory. Therefore, taking more inference time compared to the regular CNN.

CPU or memory boundedness The results can be seen in Figure 4-4 and from this plot, some results can be interpreted. First Figure 4-4a and Figure 4-4c are discussed. In these plots the memory consumption is plot against time. It can be seen that all results not

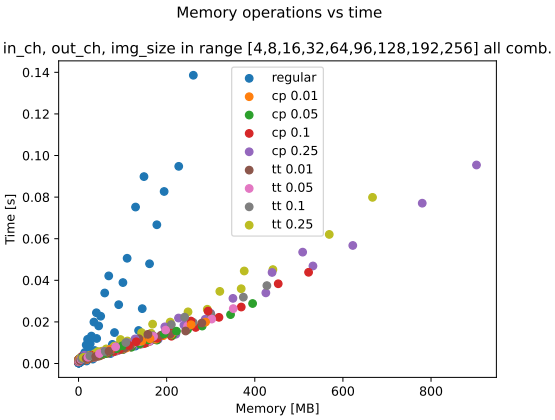
coloured blue (which are the results for the CP-CNN and TT-CNN)seem to have a somewhat linear relationship with memory and time. This relationship is more clear from 0 MB up to approximately 200 MB, but for higher memory values it seems that a linear relation seem to be followed. In Figure 4-4a and Figure 4-4c, it seems that the blue dots (which is the regular CNN) do follow a linear trend but this trend is a lot steeper and do seem to show more variance.

Figure 4-4b and 4-4d show the amount of Multiply-Accumulate (MAC) operations over time. In these two plots reverse effect can be seen although it is not as pronounced. In Figure 4-4b and 4-4d the blue dots do seem to follow a rough linear pattern. In this plot the non-blue dots, which represents the decomposed systems do seem to have a steeper increase. Also this relation seem to be somewhat linear, but only for the systems having more than 2e9 MAC operations.

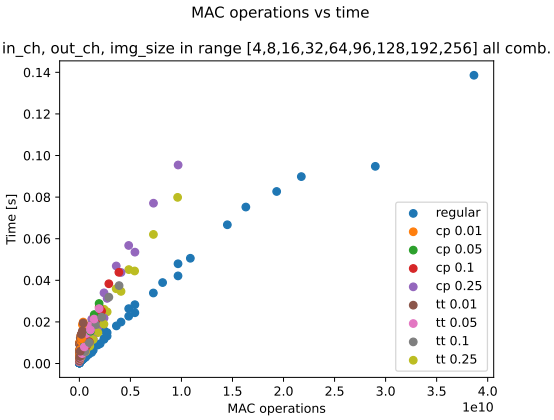
Combining the last two paragraphs indicates that certainly for small systems. The regular CNN is bound by the amount of MAC operations and the CP-CNN and TT-CNN by their memory usage.

Extremely small systems Additionally if one looks at the far left lower side of Figure 4-4c and/or Figure 4-4d, one can see a high spread in time is present and that the points are above the blue line. This indicates that the CP-CNN and TT-CNN do have a small bias compared to the regular CNN. This could be explained by the fact that more layers require extra initialization time which can not be explained by either the amount of memory used or the number of MAC operations.

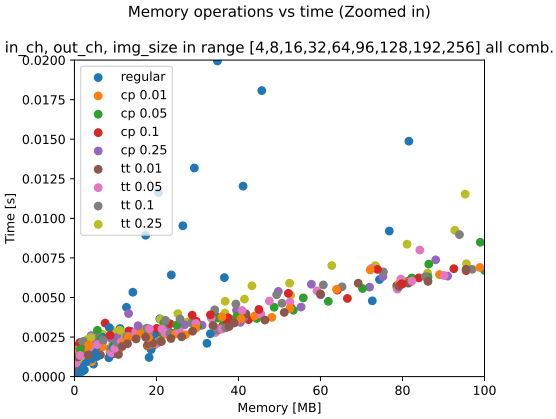
Conclusive Where it can be seen that regular CNNs are affected by the amount of memory allocated. There does not seem to be a clear relation between memory consumption and inference time and therefore they do not seem to be memory bound. For CP-CNN and TT-CNNs this is different, because the amount of time it does take to do an inference for some data does seem to increase with memory. In the next section closer look is taken at this using a linear regression model.



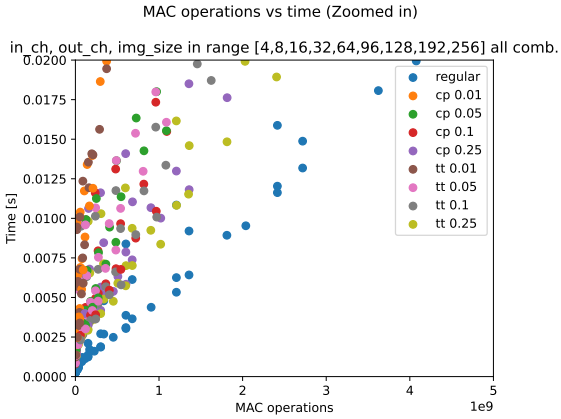
(a) Memory vs time plot for different systems



(b) MAC vs time plot for different systems



(c) Zoomed-in Memory vs time plot for different systems



(d) Zoomed-in MAC vs time plot for different systems

Figure 4-4: Results of 2nd experiment for regular CNN and, CP-CNN and TT-CNN c in $\{0.01, 0.05, 0.1, 0.25\}$
Note that the lower two figures are zoomed in versions of the upper to figures.
In the legend, regular/tt/cp define CNN-type used. The number defines the compression ratio c used.

4-2-2 Comparing various systems for memory and time

Additionally, as a second part of this experiment, the data was filtered and plotted differently compared to the plot in the previous section. For this plot only the compression ratio 0.1 and 0.01 are shown for the CP-CNN and TT-CNN. To give a better insight on highly compressed kernels and to see whether a biased in initialization time seems to exist for the CP-CNN and TT-CNN. Additionally, this plot makes subplots of different configurations for which the number of input and output channels and, the image size increases. In each subplot, two ratio's are shown. First the ratio between the inference time of the regular and decomposed CNN this is done such that the inference time of all five systems is divided by the inference time of the regular CNN. The other ratio shown in the subplot is the amount of memory for each system, this is done in the same way as the time. This is done to get a normalized overview of the difference in time and memory between the three decomposed CNN for different hyperparameter choices.

This is shown in Figure 4-5 and two conclusions are drawn from this figure:

Smaller systems In systems with a small kernel (left column and top row of Figure 4-5, it can be seen that the total memory consumption for the CP-CNN and TT-CNN is lower than or approximately the same as the regular CNN. Despite this, the inference time is larger relative to the regular CNN. This indicates that memory is not the reason that the inference time is longer, but there is some other source which gives a higher inference time.

Larger systems For the systems which have more than four input and output channels, four or a larger image size than 4x4. Two gradual trends can be seen. The first trend is that the relative amount of memory consumption starts to increase. Secondly, the inference time gradually decreases with a larger image size and/or kernel. From this, two things are remarked. The first remark is that there is some negative bias in terms of inference time towards these larger systems regardless of the amount of memory used. Secondly, it is remarked that memory is not necessarily devastating for the performance, at least not for this system. The MAC operations savings do provide better performance.

Summarizing There does not seem to be a relation between the relative amount of memory used and the relative performance of the CP-CNN and TT-CNN this indicates that there are also other factors at play. These other factors seem to have the most influence on systems with either a small image size or a small kernel size.

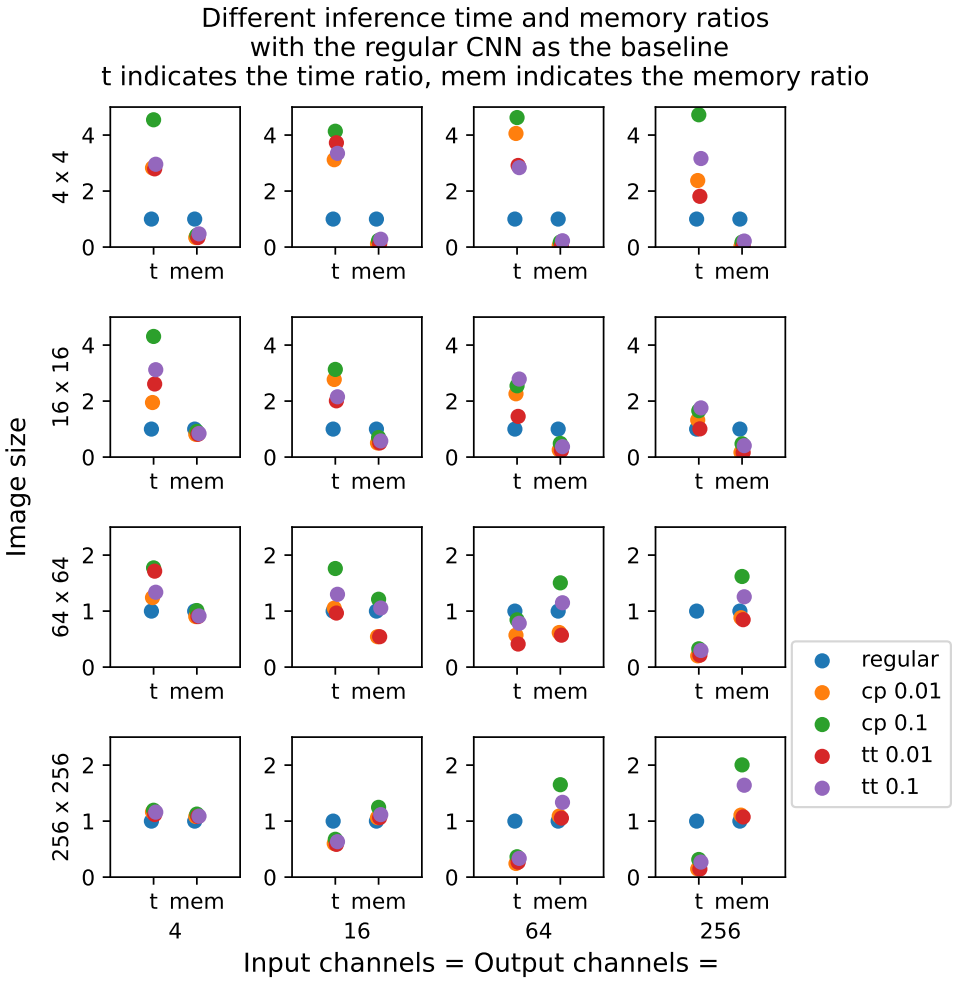


Figure 4-5: Ratio for inference time and memory usage between regular CNN and some CNN types
In the legend, regular/tt/cp define CNN-type used. The number defines the compression ratio c used.

4-3 Experiment 3: Verify the time model

Finally, regression models were made for one the regular CNN and one for the CP-CNN and TT-CNN combined. For the latter results from both the CP-CNN and the TT-CNN were put into one model. This was first tried as it would more generalize the model which would be seen as an advantage. The quality of the models can be seen in Table 4-1. First the results will be discussed for the linear models consisting of only the amount of memory and the number of MAC operations. After this the results of the models which consists of both the amount of memory and MAC operations will be discussed.

The MAC operations and memory only models For the MAC operations and memory only models. Two things draw attention. First of all, for the TT/CP models the amount of memory used can quite well predict the inference time with a Variance accounted for (VAF)

of 93.32 in the validation run. On the other hand, it can also be seen that the regular CNN shows a quite good correlation with the number of operations that have to be done, which can be seen by the VAF of 96.69 in the validation run. From this it is concluded that the Tensor Train (TT)/Canoncial Polyadic Decomposition (CPD) decomposed CNNs are memory bound and the regular CNN appears to be compute bound.

The models with both memory and MAC operations For the other two models it can be seen that adding extra parameters to the regression model gives better results, because the VAF increases compared to the models of the former paragraph as the minimum VAF for these higher order is 97.64 compared to the maximum VAF of 96.69 for the valiation set using just one linear parameter. Therefore it is concluded if information from the amount of memory and the amount of MAC operations is included.

Conclusive This test indicates that the inference time can be well predicted for these types of CNN problems with good accuracy. This makes it a useful method to first acquire data of the device on which the inference is run and see whether it is useful to use TT-CNNs and/or CP-CNNs instead of regular CNNs only.

Table 4-1: Results of polynomial regression fitting

Model	Training			Validation		
	RMSE	VAF	R squared	RMSE	VAF	R squared
TT/CP memory degree 1	$1.16 * 10^{-3}$	94.77	0.947	$1.34 * 10^{-3}$	93.32	0.933
Regular memory degree 1	$6.64 * 10^{-3}$	75.25	0.753	$6.64 * 10^{-3}$	75.31	0.752
TT/CP MAC degree 1	$1.84 * 10^{-3}$	86.85	0.868	$2.09 * 10^{-3}$	83.73	0.837
Regular MAC degree 1	$2.25 * 10^{-3}$	97.28	0.972	$2.44 * 10^{-3}$	96.69	0.996
TT/CP memory + MAC degree 1	$5.31 * 10^{-4}$	98.91	0.981	$5.49 * 10^{-4}$	98.88	0.989
Regular memory + MAC degree 1	$1.82 * 10^{-3}$	98.21	0.982	$2.05 * 10^{-3}$	97.64	0.976
TT/CP memory + MAC degree 2	$4.77 * 10^{-4}$	99.11	0.991	$5.05 * 10^{-4}$	99.05	0.991
Regular memory + MAC degree 2	$1.51 * 10^{-3}$	98.78	0.988	$1.85 * 10^{3}$	98.05	0.980

Chapter 5

Conclusion

In this chapter, first, the subresearch questions will be answered after which the main research question will follow. In the second section, a discussion will be given about the achieved results after which future research directions will be laid out.

5-1 Conclusion

Subquestion 1 The first subquestion states: Can we derive the theoretical memory usage of a decomposed Convolutional Neural Networks (CNN) and a non-decomposed (or 'regular') CNN?

Looking at the results of Section 3-1 it can be seen that a theoretical derivation can be made. It can also be seen that there are certain situations in which the amount of memory for a Canonical Polyadic-Convolutional Neural Network (CP-CNN) or Tensor Train Convolutional Neural Network (TT-CNN) can be higher. In the test case is was up to 1.6 times.

Subquestion 2 The second subquestion states: How does the theoretical analysis compare to a real-life scenario and can we see an influence of memory usage on the inference time?

In the results of Section 4-1 it can be seen that the amount of memory in a real-life scenario can be up to 5.5 times more than what was theorised in Section 3-1. Hence, the theoretical analysis gives a bit of a rosy view of the memory used. Furthermore, Section 4-2 shows that memory usage greatly influences the amount of inference time. However, this may not always lead to a lower inference time. This analysis shows that memory is not the main bottleneck in terms of inference time for the tested system. Despite this, it does show that the CP-CNN and TT-CNN do use more memory for larger CNNs.

Subquestion 3 The last subquestion states: Can we predict when it is more advantageous in terms of inference time to use a decomposed CNN or a regular CNN?

The results shown in Section 4-3 show that the amount of inference can be predicted very accurately for the given system. A Variance accounted for (VAF) of ranging from 93.32 up to 96.69 could be achieved on the verification data by using one parameters (either Multiply-Accumulate (MAC) or memory usage). Using both memory usage and the number of MAC operations, the prediction VAF could be improved to 99.05 for the decomposed CNN and 98.05 for the regular CNN. These prediction shows a very close correlation to the actual measured time. Therefore, it is concluded that a predicting whether a CP-CNN or TT-CNN is more advantageous in terms of inference time compared to a regular CNN can be made.

Main research question Finally, the main research question states: How does memory usage influence the inference time for Canonocial Polyadic decomposed or Tensor Train decomposed CNN layers compared to non-decomposed CNN layers?

During this thesis, it is shown that during inference on a CP-CNN or TT-CNN model, more memory may be required then when using a regular CNN. However, this does not necessarily lead to a longer inference time. For small systems, the results suggest that some other force acts on the inference time. For larger systems the CP-CNN and TT-CNN do seem to reduce inference time.

5-2 Discussion

In this section, some aspects of this research are placed in a larger perspective.

Accuracy of CP-CNN and TT-CNN models While the influence on the tensor decompositions on inference times are measured for certain hyper parameters. The accuracy of the decomposed kernels is not tested. Due to the compression provided by the CP-CNN and TT-CNN future research may investigate the effects of compression on accuracy. While [10, 25, 27], do give insights on the effect of accuracy using the CP-CNN and TT-CNN for their model, no rules of tumb do seem to exists.

What could be the reason that for smaller systems a larger inference time is seen? It seems that for smaller systems some bias effect is at play. When one zooms really well into Figure 4-4d. It can be seen that for most no MAC operations, still, a lot of time is needed for the decomposed CNNs. Therefore, it seems that some time is required to initialize and that this is more significant for these smaller problems. It is theorized that maybe the fact that the CP-CNN and TT-CNN consist of multiple layers could lead to longer initialization times. This initialization time could originate from for instance the action of allocating memory, thread creation and al sort of CPU process creation tasks. In future research, it may be interesting to test how much time is spend initializing the models. As it seems that for the smaller kernel in this thesis i.e. 4x4x3x3 and 8x8x3x3 initialization time seems to account for a great portion of the inference time.

What could the influence on GPU systems be? Graphics Processing Unit (GPU)s do seem to provide a lot more computational power and have separate memory for these calculations.

Based on this thesis it is not easy to predict what would happen, but GPUs are often more parallelized. Hence, it could be that the bias seen in this Central Processing Unit (CPU) case is more prominent for GPUs. Additionally, it could be that the problem for each layer must be large enough to benefit optimally from the GPUs parallel capacities. Giving the decomposed CNN a disadvantage by having four sequential layers instead of 1. Therefore, for GPU it would be interesting to see what happens, but it is expected that the system must be even larger compared to the CPU to benefit from the MAC operation saving.

How does this relate to energy? It would be very interesting to see if Artificial Intelligence (AI) can really be made greener in some cases. Since memory, compute time and energy are strongly related for some lower-level languages [33], it would be interesting to see if the inference time found in this research relates to energy consumption. Demi's [7] thesis does of course, give an insight into this, but the hyperparameters she chose (mainly small image size) did also not perform well in this thesis as well. Maybe that decomposed CNNs become more performant relative to the CNN in larger scale systems as the negative effect of sequential operations vanish.

Could disabling the IntelMKL reduce inference time? The IntelMKL seem to provide a lot of memory overhead for this system. It may be interesting to see if this leads to worse performance. It seems that it is most efficient to convert the input at once to the MKLDNN format and keep it in this format. This may be more optimal than constantly switching format which Tensorly-Torch does now. If this change in format does reduce memory reordering it may provide a benefit. If the same amount of reordering operations will remain necessary for the IntelMKL case then it would be interesting to see if it provides any benefit at all.

Using other number representations than float32 A smaller number representation (e.g. NvideaTensorfloat32, bfloat16 or int8) should benefit the decomposed CNNs. For tested system, it seemed that the models are primarily memory bound, decreaseing the size of the elements could reduce the memory bound effects. The regular CNN should also benefit but relatively less as it also seems to be compute-bound.

What would happen to other types of CNNs? For this thesis a bound has been set on 2D-CNNs, and the stride and dilation were fixed. First, if the dimension for the kernel is changed, the amount of in between images changes. Hence, one would use relatively more memory for the CP-CNN case, as one gets an extra image with the output size times R). Just as with the 2D-case, this effect would be less strong for the TT-CNN as the in-between image would again be taken with a very low-rank term. This means that both would consume more memory, but this would be more significant for the CP-CNN. The opposite is true for the 1D case. Secondly, both stride and dilation would cause a negative effect as for both stride and dilation about the same memory needs to be read, but fewer operations would be done. This benefits the regular CNN more as it is more CPUbound. Thus both a higher stride and dilation will not affect the decomposed CNNs much but it does for the regular CNN. Finally, it would be interesting to see happen when the spatial dimensions (d_1 and d_2) are not decomposed. This will reduce the amount of layers by 1 and eliminate one of the in between images and possibly reducing the inference time. Certainly for the CP-CNN this

will save a lot off memory as it will reduce one of the output images. For the TT-CNN the image multiplied by R_2 will be eliminated, posing less of a gain because this image is often very small as R_2 is often very small. For the CP-CNN this method is implemented by [5]. However, in this paper it is compared to another decomposed type of CNN not discussed in this paper, making it hard to compare.

5-3 Further research

The results for this system are only tested for inference time and on a CPU. It would be good to use this insight to see what the effects would be on energy consumption certainly for larger CNNs as there seems to be no work available on that. Secondly, it may be insightful full what the effect will be on GPUs since GPUs are often used for training and do inference on CNNs. Another useful thing may be to investigate whether initialization time poses a thread to decomposed CNNs in terms of energy consumption and inference time. Finally, the effects of different numerical representations such as float16, bfloat16 and int8 could be investigated as this could ease the memory burden the decomposed CNN pose.

Appendix A

CPD and TT algorithm

This Appendix provides an algorithm to obtain both the Canoncial Polyadic Decomposition (CPD) and Tensor Train (TT) tensor decompositions from a single kernel.

A-1 Preliminaries

Some operations are required to obtain get the decompositions, which are not mention in the rest of the thesis. They are mentioned here.

Tensor operations

Khatri-Rao product For matrices, the Khatri-Rao product, also known as the column-wise Kronecker product is denoted by $C = A \odot B$ for $A \in \mathbb{R}^{I \times K}$, $B \in \mathbb{R}^{J \times K}$ and $C \in \mathbb{R}^{IJ \times K}$. The result C is defined in Equation A-1 [9, 22]. A general tensor definition is given in [9].

$$C = A \odot B = \begin{bmatrix} a_1 \otimes b_1 & a_2 \otimes b_2 & a_3 \otimes b_3 & \dots a_K \otimes b_K \end{bmatrix} \tag{A-1}$$

A-2 CPD

The Alternating Least Squares (ALS) algorithm used to find the CPD is shown in Algorithm 1 on the next page. This is an iterative process which fixes, for each factor matrix for each iteration, one factor matrix. Then it calculates the value the factor matrix should have. In each iteration, after all factor matrices have been updated, all factor matrices are normalized and the norms are stored in $\underline{\Lambda}$. The operations stops when the algorithm has converged or a set number of iterations has been used.

Algorithm 1 Canoncial Polyadic Decomposition [9, 22]

Input:	$\underline{\mathrm{A}}$	The input tensor with N indices
Output:	$B^{(n)}$	N factor matrices
	Λ	scalar coefficients

1: $\forall B^n$ **initialize** $B^n \in \mathbb{R}^{I_n \times R}$
2: **repeat**
3: **for** i = 1:N **do**
4: $B^{(n)} = A_n \left(B^{(N)} \odot \ldots \odot B^{(n-1)} \odot B^{(n-1)} \odot \ldots B^{(1)} \right)^{\dagger}$
5: **normalize** $B^{(n)}$, store norms in $\underline{\Lambda}$
6: **end for**
7: **until** (iteration == iteration limit || converged)

A-3 TT

To find a Tensor Train decompositions, two algorithm blocks are used. The first algorithm is truncated Singular Value Decomposition (SVD), which is an algorithm that does an SVD and then eliminates a part of the matrices to create a smaller matrices with some error. A benefit of this method is that the amount of error which is created can be determined and hence given as an input. The amount of error is the squared of the magnitude of the singular values which will be removed (together with there accompanying rows and columns in the left and right singular matrices) divided by the total magnitude of all singular values.

To obtain a TT decomposition, the algorithm is executed $N-1$ times where N is the number of indices. Each iteration, the truncated SVD algorithm will be used. From this SVD, the left singular matrix U is reshaped to form a core and the singular values times the right singular matrix $S * V^{\top}$, goes further to the next iteration. After the final iteration the remainder is reshaped into the last core.

The algorithm is also shown in the algorithm blocks below and on the next page.

Algorithm 2 *truncated_svd* [9, 30]

Input:	A	The matrix on which the SVD and truncation is done
	δ	The maximum truncation error
Output:	U	The first n left orthogonal vectors
	S	A matrix containing the n singular values
	V	The first n right orthogonal vectors

1: [U,S,V] = SVD(A)
2: i = length(S); $e = 0$
3: **while** $\sqrt{(e + S_i)} < \delta$ **do**
4: $e = e + S_i$
5: $i = i - 1$
6: **end while**
7: U = reshape(:, 1:i); S = reshape(1:i,1:i); V = reshape(1:i,:)

Algorithm 3 TT-SVD [9, 30]

Input: $\underline{A}$ The input tensor with N indices
δ The maximum approximation error
Output: $\underline{G_n}$ The N cores of the tensor train decomposition

$\delta = \frac{\epsilon}{\sqrt{N-1}}||\underline{A}||_F$
$r_0 = 1$
for i = 1:N-1 **do**
  A =reshape$\Big(A, \Big[r_{k-1}, n_k, \frac{numel(C)}{r_{k-1}n_k}\Big]\Big)$
  $[U, S, V]$ = truncated_svd(C, δ) (see Algorithm 2)
  G_i = reshape$(U, [r_{k-1}, n_k, r_k])$
  $A = SV^\top$
end for
$G_N = A$

Bibliography

[1] Laith Alzubaidi, Jinglan Zhang, Amjad J Humaidi, Ayad Al-Dujaili, Ye Duan, Omran Al-Shamma, José Santamaría, Mohammed A Fadhel, Muthana Al-Amidie, and Laith Farhan. Review of deep learning: concepts, cnn architectures, challenges, applications, future directions. *Journal of big Data*, 8:1–74, 2021.

[2] AMD. Amd optimizing cpu libraries (aocl). https://www.amd.com/en/developer/aocl.html. (Accessed 19.03.2024).

[3] Anima Anandkumar, Wonmin Byeon, Jean Kossaifi, and Saurav Muralidharan. Tensorly-torch. https://github.com/tensorly/torch, 2024. Version 0.5.0.

[4] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, CK Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Michael Suo, Phil Tillet, Eikan Wang, Xiaodong Wang, William Wen, Shunting Zhang, Xu Zhao, Keren Zhou, Richard Zou, Ajit Mathews, Gregory Chanan, Peng Wu, and Soumith Chintala. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. ACM, April 2024.

[5] Marcella Astrid and Seung-Ik Lee. Cp-decomposition with tensor power method for convolutional neural networks compression. In *2017 IEEE International Conference on Big Data and Smart Computing (BigComp)*, pages 115–118. IEEE, 2017.

[6] Eli Bendersky. Depthwise separable convolutions for machine learning. https://eli.thegreenplace.net/2018/depthwise-separable-convolutions-for-machine-learning/, 2018. (Online, Accessed 14.02.2025).

[7] Demi Breen. Towards sustainable CNNs: Tensor decompositions for green ai solutions. Master's thesis, Delft University of Technology, 2024.

[8] U.M. Cakmak and M. Cuhadaroglu. *Mastering Numerical Computing with NumPy: Master scientific computing and perform complex operations with ease*. Packt Publishing, 2018.

[9] Andrzej Cichocki, Namgil Lee, Ivan V Oseledets, A-H Phan, Qibin Zhao, and D Mandic. Low-rank tensor networks for dimensionality reduction and large-scale optimization problems: Perspectives and challenges part 1. *arXiv preprint arXiv:1609.00893*, 2016.

[10] Mateusz Gabor and Rafał Zdunek. Convolutional neural network compression via tensor-train decomposition on permuted weight tensor with automatic rank determination. In *International Conference on Computational Science*, pages 654–667. Springer, 2022.

[11] Jacob Gildenblat. pytorch-tensor-decompositions. https://github.com/jacobgil/pytorch-tensor-decompositions, 2021. commit = aca536ffacbd04be39d5a527dc2dafb9ac6a69df.

[12] Gene H Golub and Charles F Van Loan. *Matrix computations*. Fourth edition. John Hopkins University press, Balitmore, 2013.

[13] Jiuxiang Gu, Zhenhua Wang, Jason Kuen, Lianyang Ma, Amir Shahroudy, Bing Shuai, Ting Liu, Xingxing Wang, Gang Wang, Jianfei Cai, et al. Recent advances in convolutional neural networks. *Pattern recognition*, 77:354–377, 2018.

[14] Johan Håstad. Tensor rank is np-complete. In *Automata, Languages and Programming: 16th International Colloquium Stresa, Italy, July 11–15, 1989 Proceedings 16*, pages 451–460. Springer, 1989.

[15] Frederik Hogenbosch. Canonical polyadic decomposition for eeg analysis. Master's thesis, Delft University of Technology, 2024.

[16] Sebastian Holtz, Th Rohwedder, and R Schneider. The alternating linear scheme for tensor optimisation in the tt format. *Preprint*, 71, 2011.

[17] Intel Inc. Intel(r) math kernel library for deep neural networks. https://oneapi-src.github.io/oneDNN/v1.0/dev_guide_convolution.html, 2019. Accessed: 2025-01-19.

[18] Intel Inc. Intel vtune (tm) profiler. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html#gs.i6xhgk, 2024. Accessed: 2025-02-04.

[19] Intel. Developer reference for intel® oneapi math kernel library - c. https://www.intel.com/content/www/us/en/docs/onemkl/developer-reference-c/2024-0/overview.html. (Accessed 19.03.2024).

[20] Rohan Jha, Rishabh Jha, and Mazhar Islam. Forecasting us data center co2 emissions using ai models: emissions reduction strategies and policy recommendations. *Frontiers in Sustainability*, 5:1507030, 2025.

[21] Nikhil Ketkar and Jojo Moolayil. *Convolutional Neural Networks*, pages 197–242. Apress, Berkeley, CA, 2021.

[22] Tamara G Kolda and Brett W Bader. Tensor decompositions and applications. *SIAM review*, 51(3):455–500, 2009.

[23] Jean Kossaifi, Yannis Panagakis, Anima Anandkumar, and Maja Pantic. Tensorly: Tensor learning in python. *Journal of Machine Learning Research*, 20(26):1–6, 2019.

[24] Jean Kossaifi, Yannis Panagakis, Anima Anandkumar, Maja Pantic, et al. Tensorly v0.9.0. https://github.com/tensorly/tensorly, Nov 2024.

[25] Jean Kossaifi, Antoine Toisoul, Adrian Bulat, Yannis Panagakis, Timothy M Hospedales, and Maja Pantic. Factorized higher-order cnns with an application to spatio-temporal emotion estimation. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 6060–6069, 2020.

[26] Daniel Kusswurm. *X86-64 Core Architecture*, pages 1–23. Apress, Berkeley, CA, 2023.

[27] Vadim Lebedev and Victor Lempitsky. Speeding-up convolutional neural networks: A survey. *Bulletin of the Polish Academy of Sciences. Technical Sciences*, 66(6):799–811, 2018.

[28] Shana Lynch. The state of ai in 9 charts. https://hai.stanford.edu/news/state-ai-9-charts, 2022. Accessed: 2025-02-17.

[29] Alexander Novikov, Dmitrii Podoprikhin, Anton Osokin, and Dmitry P Vetrov. Tensorizing neural networks. *Advances in neural information processing systems*, 28, 2015.

[30] Ivan V Oseledets. Tensor-train decomposition. *SIAM Journal on Scientific Computing*, 33(5):2295–2317, 2011.

[31] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. arxiv 2015. *arXiv preprint arXiv:1511.08458*, 2015.

[32] Fabian Pedregosa, Gael Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[33] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Ranking programming languages by energy efficiency. *Science of Computer Programming*, 205:102609, 2021.

[34] Anh-Huy Phan, Andrzej Cichocki, André Uschmajew, Petr Tichavsky, George Luta, and Danilo Mandic. Tensor networks for latent variable analysis. part i: Algorithms for tensor train decomposition. *arXiv preprint arXiv:1609.09230*, 2016.

[35] Vagelis Plevris, German Solorzano, Nikolaos P Bakas, and Mohamed El Amine Ben Seghier. Investigation of performance metrics in regression analysis and machine learning-based prediction models. In *8th European Congress on Computational Methods in Applied Sciences and Engineering (ECCOMAS Congress 2022)*. European Community on Computational Methods in Applied Sciences, 2022.

[36] Kartik Prabhu, Albert Gural, Zainab F Khan, Robert M Radway, Massimo Giordano, Kalhan Koul, Rohan Doshi, John W Kustin, Timothy Liu, Gregorio B Lopes, et al. Chimera: A 0.92-tops, 2.2-tops/w edge ai accelerator with 2-mbyte on-chip foundry resistive ram for efficient training and inference. *IEEE Journal of Solid-State Circuits*, 57(4):1013–1026, 2022.

[37] Thorsten Rohwedder and André Uschmajew. On local convergence of alternating schemes for optimization of convex problems in the tensor train format. *SIAM Journal on Numerical Analysis*, 51(2):1134–1162, 2013.

[38] Roy Schwartz, Jesse Dodge, Noah A Smith, and Oren Etzioni. Green ai. *Communications of the ACM*, 63(12):54–63, 2020.

[39] Nguyen Van Thieu. PerMetrics: A Framework of Performance Metrics for Machine Learning Models. *Journal of Open Source Software*, 9(95):6143, March 2024.

[40] Nguyen Van Thieu and Seyedali Mirjalili. Mealpy: An open-source library for latest meta-heuristic algorithms in python. *Journal of Systems Architecture*, 2023.

[41] Roberto Verdecchia, June Sallou, and Luís Cruz. A systematic review of green ai. *WIREs Data Mining and Knowledge Discovery*, 13(4):e1507, 2023.

[42] Saining Xie, Ross Girshick, Piotr Dollar, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[43] Yujia Zhai, Elisabeth Giem, Quan Fan, Kai Zhao, Jinyang Liu, and Zizhong Chen. Ft-blas: a high performance blas implementation with online fault tolerance. In *Proceedings of the ACM International Conference on Supercomputing*, pages 127–138, 2021.

Glossary

List of Acronyms

SIMD	Singe Instruction Multiple Data
CNN	Convolutional Neural Networks
SVD	Singular Value Decomposition
CPD	Canoncial Polyadic Decomposition
ALS	Alternating Least Squares
MALS	Modified Alternating Least Squares
TT	Tensor Train
MPS	Matrix Product State
CP-CNN	Canonical Polyadic-Convolutional Neural Network
TT-CNN	Tensor Train Convolutional Neural Network
MAC	Multiply-Accumulate
VAF	Variance accounted for
RMSE	Root Mean Squared Error
COD	Coefficient of Determination
AI	Artificial Intelligence
CPU	Central Processing Unit
GPU	Graphics Processing Unit
HPC	High performance computing
BLAS	Basic Linear Algebra Subprograms
IntelMKL	Intel Math Kernel Library
AOCL	AMD Optimizing CPU Libraries