DEGREE PROJECT IN THE FIELD OF TECHNOLOGY
MEDIA TECHNOLOGY
AND THE MAIN FIELD OF STUDY
COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2018

An Investigation of Low-Rank Decomposition for Increasing Inference Speed in Deep Neural Networks With Limited Training Data

VICTOR WIKÉN

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

An Investigation of Low-Rank Decomposition for Increasing Inference Speed in Deep Neural Networks With Limited Training Data

Victor Wikén

Supervisor: Stefano Markidis
Examiner: Erwin Laure

September 22, 2018

Abstract

In this study, to increase inference speed of convolutional neural networks, the optimization technique low-rank tensor decomposition has been implemented and applied to AlexNet which had been trained to classify dog breeds. Due to a small training set, transfer learning was used in order to be able to classify dog breeds. The purpose of the study is to investigate how effective low-rank tensor decomposition is when the training set is limited. The results obtained from this study, compared to a previous study, indicate that there is a strong relationship between the effects of the tensor decomposition and how much available training data exists. A significant speed up can be obtained in the different convolutional layers using tensor decomposition. However, since there is a need to retrain the network after the decomposition and due to the limited dataset there is a slight decrease in accuracy.

Sammanfattning

För att öka inferenshastigheten hos faltningssnätverk, har i denna studie optimeringstekniken *low-rank tensor decomposition* implementerats och applicerats på AlexNet, som har tränats för att klassificera hundraser. På grund av en begränsad mängd träningsdata användes *transfer learning* för uppgiften. Syftet med studien är att undersöka hur effektiv *low-rank tensor decomposition* är när träningsdatan är begränsad. Jämfört med resultaten från en tidigare studie visar resultaten från denna studie att det finns ett starkt samband mellan effekterna av *low-rank tensor decomposition* och hur mycket tillgänglig träningsdata som finns. En signifikant hastighetsökning kan uppnås i de olika faltningslagren med hjälp av *low-rank tensor decomposition*. Eftersom det finns ett behov av att träna om nätverket efter dekompositionen och på grund av den begränsade mängden data så uppnås hastighetsökningen dock på bekostnad av en viss minskning i precisionen för modellen.

Contents

Glossary

activation function A nonlinear function

back-propagation Efficient method to compute gradients

epoch One iteration through the training set

feature map The output from a convolutional operation

fine-grained classification problem Classification problem where the classes share common parts

inference The evaluation of new data after the model has been trained on the training data

kernel The weights which is used for convolution applied on the input

learning rate Positive scalar which determines the step size to update the learnable parameter θ

MLP Multilayer perceptron: A composition of functions

stride The step size used for moving the kernel over the input

tensor A multidimensional array

training Utilizing the gradients computed by back-propagation to minimize the loss

training set The set of data used to train the model

transfer learning Utilizing pretrained weights

1 Introduction

In this section, the motivation, problem domain and the research question for this study are introduced.

1.1 Motivation

Having a fast inference speed of a neural network is crucial for deploying real time applications. A significant amount of research has been made studying how to increase the accuracy of convolutional neural networks. However, research has often focused on increasing the accuracy while neglecting the size of the network as well as the inference speed [1, 2].
Recent studies have suggested redundancies exists in convolutional neural networks and this property can be used to increase the inference speed [3, 4].
In a convolutional neural network the majority of the computations consist of the convolutional computations [4]. However, the size of the network model comprise mostly of the fully connected layers [5, 4]. To decrease the size of a convolutional network it is important to both reduce the size of the fully connected layers as well as the convolutional layer [4].

1.1.1 Problem domain

The chosen problem domain for this study is dog breed classification. The problem is a fine-grained classification problem with 120 relatively similar classes where different classes share common parts [6]. The dataset is relatively small compared to other datasets, consisting of 10K images in total. Previous studies have often focused on increasing inference speed on the ImageNet dataset which training set consists of of 1.2 million images [7, 4]. The purpose of using a limit dataset, and evaluating the effects of the decomposition, is that many real world machine learning problems have a limited amount of data.

1.2 Research question

The purpose of this study is to answer the following research question:

- How is the effectiveness of the decomposition affected by having a limited training set?

In order to answer the research question, the following questions will be investigated:

- How does the decomposition affect:
    1. inference speed?
    2. accuracy of the model?

1.3 Societal interest

Increasing inference speed is important for real time machine learning applications and a common problem is having a limited training set. Therefore the results of this study could be of interest to those who are developing real time machine learning applications. Artificial intelligence is more and more becoming a part of everyday life with applications such as Siri, custom recommendations of media content and , potentially, in the near future self driving cars for commercial use.

1.4 Ethics and sustainability

Artificial intelligence is a field that shows great potential but is also a field which requires great responsibility. Artificial intelligence can help improve society when it comes to creating smarter solutions and optimizations for real world problems, such as sustainability problems. However, Artificial intelligence could also be misused with catastrophic consequences. The purpose of this study is to investigate how inference speed can be increased, while maintaining accuracy, when the training data is limited. Increasing inference speed is a step towards more powerful Artificial intelligences.
The dataset consists of dog images and therefore there was no sensitive information used in this study.

1.5 Report outline

The outline of this report is as follows:

1. The necessary theory to understand the thesis is presented in section **2 Background**.
2. A literature survey about the related work to this study is presented in section **3 Previous work**.
3. The experimental set-up is presented in section **4 Experimental Set-up** providing information about the necessities in order to set up the experiments of this study.
4. The methodology of this study is presented in section **5 Methodology for Low-rank decomposition**, explaining in detail the methodology of the study how the low-rank decomposition was applied.
5. The experiments conducted and how they were evaluated is presented in section **6 Experiments**
6. The results of the study is presented in section **7 Results**
7. The discussion and the conclusion of the study is presented in section **8 Discussion and Conclusion**

2 Background

This section describes the necessary theory in order to understand the methodology of this study.

2.1 Machine learning

Machine learning as we know it today was first coined 1959 by Samuel in Ref. [8] and he described it as follows:
"Programming computers to learn from experience should eventually eliminate the need for much of this detailed programming effort". Commonly the following definition is used:
"the field of study that gives computers the ability to learn without being explicitly programmed"

2.2 Supervised learning

Supervised learning is a method to give the computers the ability to learn by providing the learning algorithm with data with the correct target output. An example of supervised learning is a Feed forward neural network [9].

2.3 Feed forward neural network

The purpose of a feed forward network is to approximate some function f^* that, for example, maps an input $\boldsymbol{x}$ to a category y, $y = f^*(\boldsymbol{x})$. The feed forward network learns the value of a parameter $\boldsymbol{\theta}$, for the mapping function $y = f(\boldsymbol{x}; \boldsymbol{\theta})$, which best approximates the function f^*.

The term **neural** in *"Feed forward neural network"* has its origin from the fact that idea behind the networks were loosely inspired by neuroscience. The name *"network"* originates from the fact that feed forward neural networks are composed of different functions. The composition of these functions is called a multilayer perceptron (MLP) and how these functions are composed can be described with a direct acyclic graph. An example of a multilayer perceptron can be seen in Figure 1.
For three functions the following chain can be formed: $f(\boldsymbol{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x})))$.
With l functions: $f(\boldsymbol{x}) = f^{(l)}(f^{(l-1)}(..(f^{(1)}(\boldsymbol{x})..)))$, each function constitutes a layer, the first function being the first layer, the second function being the second layer and the lth function being the lth layer. The lth layers is called the output layer, while the layers between the input and the output layer are called **hidden layers**.
The name *"hidden"* comes from the fact that the training algorithm needs to decide how to use these layers to get the desired output without specifying how to do so by the training data. This is different compared to the output layer which is directly specified by each sample of the training data, which includes the correct y, of what action to take to get a result close to y. The hidden layers are not provided with a desired output from the training data and is therefore called hidden. The number of composed functions, or layers, is referred to as the **depth** of the network. If the layers are many, the network is called deep, hence, deep neural networks are deep networks with many hidden layers. Each element in a hidden layers is referred to as a **unit**. The number of units in each layers is referred to as the **width** of the network [9].

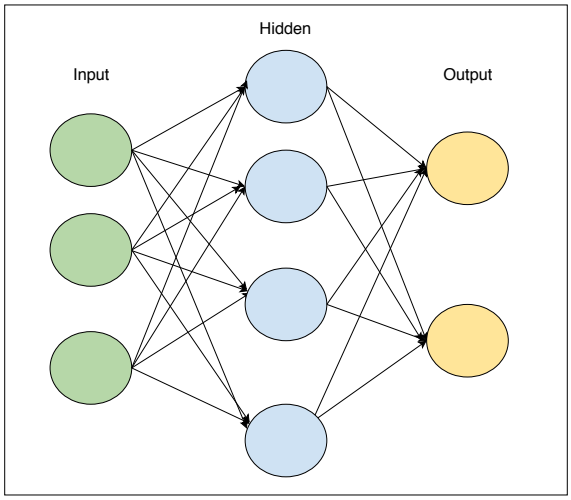


Figure 1: An example of a Multilayer perceptron (MLP) (figure adapted from [9]).

2.3.1 Activation function

In order to learn a nonlinear function a nonlinear activation function is applied after each linear function $f()$. [10]
The activation function is a nonlinear function such as:

- **Softmax**, defined as $a(\mathbf{z}) = \frac{exp(\mathbf{z})}{\sum_j exp(z_j))}$
- Rectified linear unit (**ReLU**), defined as $a(\mathbf{z}) = max(0, \mathbf{z})$

ReLU is the recommended default activation function in modern neural networks. **Softmax** is mostly used for the output layer to represent the probability distribution over n classes [9].

2.3.2 Forward propagation

Forward propagation allows information to flow forward through the network. A feed forward network is a composition of non-linear functions on linear combinations of the input, where the coefficients in the linear combinations are learnable adaptive parameters [10].
A single layer is defined as:

$$\boldsymbol{z} = a(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) \tag{1}$$

where $a()$ is the activation function, $\boldsymbol{x}$ is the initial input to the network, $\boldsymbol{W}$ is the adaptive weights and b is the adaptive bias. If the input $\boldsymbol{x} \in \Re^{n*n}$ then the weights $\boldsymbol{W}$ must be $\boldsymbol{W} \in \Re^{k*n}$, where k is the number of outputs. A network consisting of L layers is defined as:

$$\boldsymbol{z_1} = a(\boldsymbol{W_1}\boldsymbol{x} + \boldsymbol{b_1}) \tag{2}$$

$$\boldsymbol{z_l} = a(\boldsymbol{W_l}\boldsymbol{z_{l-1}} + \boldsymbol{b_l}) \quad , \, l = 2, .., L \tag{3}$$

Where the weights $\boldsymbol{W_l} \in \Re^{|z_l|*|z_{l-1}|}$ [10].

2.3.3 Loss function

The loss function $L(\hat{\mathbf{y}}, \mathbf{y})$, or cost function, is the measurement of the **loss** for the predicted output $\hat{\mathbf{y}}$ and the target output $\mathbf{y}$. Commonly, neural networks are trained using maximum likelihood with the negative log-likelihood used as the loss function [9].

2.3.4 Back-propagation

Back-propagation is a computationally inexpensive method to compute the gradients. The information obtained from the cost is passed backwards through the network in order to compute the gradient, hence, the name back-propagation. The chain rule is used to compute the gradients of functions composed of other functions with known gradients [9].

2.3.5 Gradient Descent

Gradient descent is an algorithm used for solving an optimization problem of either minimizing or maximizing some function $f(x)$ by altering x. Gradient descent works by shifting x in small steps with the opposite sign of its derivative. For neural networks it is the **Loss function** that we want to minimize. Minimizing is achieved by maximizing $-f(x)$. Minimizing the loss function, by utilizing the gradients computed by the back-propagation, is referred to as **training** the network [9].

2.3.6 Stochastic Gradient Descent

Stochastic gradient descent is an extension of Gradient descent and is used by most deep learning architectures to train the network with the gradients obtained from back propagation. Since deep networks usually require a large training set it is computationally infeasible to calculate the loss over the entire training set $\mathbb{T} = \{x^{(1)}, ..., x^{(m)}\}$.

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^{m} \nabla_{\theta}(x^{(i)}, y^{(i)}, \boldsymbol{\theta}) \tag{4}$$

The computational complexity of this operation is $O(m)$ and as the size of the training set m increases the computational time scales linearly. To accommodate for this restriction, Stochastic gradient descent can be used. Stochastic gradient descent utilizes the fact that the gradient is an expected value that can be approximately estimated using a small subset of the training set. This is referred to as a **minibatch** of examples $\mathbb{B} = \{x^{(1)}, ..., x^{(m')}\}$, where $m' << m$. A new minibatch is drawn uniformly from the training set at each step of the algorithm.

$$\boldsymbol{g} = \frac{1}{m'} \nabla_{\boldsymbol{\theta}} \sum_{i=1}^{m'} \nabla_{\theta}(x^{(i)}, y^{(i)}, \boldsymbol{\theta}) \tag{5}$$

The computational complexity of this operation is $O(m')$ which is fixed as the size of the training set increases. The stochastic gradient descent algorithm updates the learnable parameter $\boldsymbol{\theta}$ as:

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \epsilon \boldsymbol{g} \tag{6}$$

where ϵ is a positive scalar which determines the step size, referred to as the learning rate. The size of the minibatch is referred to as the **batch size**. One iteration over the entire training set is referred to as an **epoch** [9].

2.3.7 Adam Gradient Descent

Adam gradient descent is another method for efficient stochastic optimization, which was introduced in Ref. [11]. Adam computes individual adaptive learning rates for each parameter and stores both an exponentially decaying average of past squared gradients and an exponentially decaying average of past gradients [11, 12]. Kingma and Ba showed empirical results demonstrating that Adam works well both in practice but also compares favorably to other stochastic optimization methods [11].

2.3.8 Regularization

Regularization is any method to alter the learning algorithm with the intent to make the learning algorithm reduce the error on the test data but not on the training data [9].

2.3.9 Batch normalization

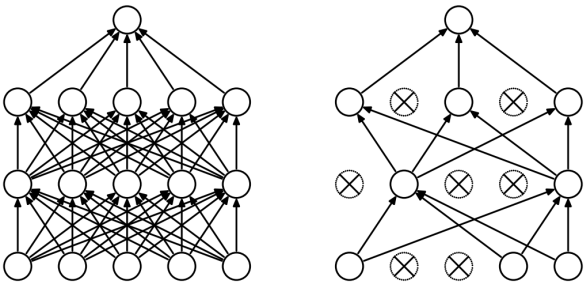
Batch normalization was introduced by Ioffe and Szegedy in Ref. [13]. Deep networks have been known to be difficult to train. As described in section **2.3.4 Back-propagation** each parameter is updated by the gradient, under the assumption that the other layers do not change.
The reasons being that in practice all the layers are updated simultaneously. When many functions are composed together, and are changed simultaneously with values computed under the assumption that the other functions were constant, unexpected results can occur [14].
This requires the training process to have a sufficiently low learning rate, which results in longer training periods, and also careful parameter initialization [13].
Batch normalization addresses these problem by using normalization as part of the network architecture by performing normalization for each training minibatch.
The benefits are not only allowing for deeper networks but batch normalization allows for higher learning rates, makes networks less dependent on good initialized values and acts as a regularizer [13].

2.3.10 Overfitting

Overfitting is a problem in machine learning when there is a large distance between the training and test accuracy, the training accuracy being significantly higher than the test accuracy [9]. Overfitting occurs when the model is too complex for the amount of training data available. Increasing the amount of training data can reduce the problem of overfitting [10].

2.3.11 Dropout

Since deep neural networks can have a large number of parameters it can be a powerful machine learning model. However, since there is a large number of parameters overfitting can be a significant problem. Dropout is a method to prevent overfitting and has shown to be effective in improving the performance of neural networks. Dropout regulates the network by preventing units from co-adapting too much, which is achieved by randomly dropping units along with their connection during training [15].



(a) Standard Neural Net (b) After applying dropout.

Figure 2: Drop out illustration, figure taken from [15].

2.3.12 Inference and prediction

Inference is the evaluation of new data after the model has been trained on the training data [10]. As described in section **2.3.1 Activation function** the **Softmax** operation is used to represent the probability distribution over n classes. The class with the highest probability is then chosen as the models prediction for the input data [16].

2.3.13 Top-1- and Top-5 accuracy

The top-1- and top-5 accuracy are measurements of classification/prediction accuracy of the model. The top-1 accuracy is the accuracy of the model classifying an input with the correct class. The top-5 accuracy is the accuracy of having the correct class in one of the 5 highest probabilities in the output of the model [9].

2.4 Convolutional neural networks

A convolutional neural network is a feed forward network that utilizes convolution in at least one of the layers.
A convolution is a linear operation defined as:

$$s(t) = (x * w)(t) = \int_{a=-\infty}^{\infty} x(a)w(t-a) \tag{7}$$

and for discrete convolution:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a) \tag{8}$$

For convolutional neural networks, x and w in equation(7) are often referred to as the **input** and as the **kernel**, respectively. The output of the operation can be referred to as a **feature map**.

Convolution is not restricted to one-dimensional inputs, for many machine learning problems the input is often a multidimensional array, referred to as a tensor [9].
The order of the tensor determines the dimensions of the array. A one order tensor is referred to as a vector, a second order tensor as a matrix and a third order tensor or higher, is referred to as a high order tensor
A N-order tensor $\boldsymbol{X}$ has the form $\Re^{I_1 * I_2 * .. * I_n}$.
Flattening a tensor is the transformation from, for example, a tensor of dimensions $8x3x2$ to a matrix $8x6$ [17].

If the input is a two-dimensional image I the kernel K is usually two-dimensional as well, and the convolution would be defined as followed:

$$S(i,j) = (I * K)(i,j) = \sum_{a=m} \sum_{a=n} I(m,n)K(i-m, j-n)) \tag{9}$$

The convolutional operation is commutative, which means that we can define the convolution as:

$$S(i,j) = (K * I)(i,j) = \sum_{a=m} \sum_{a=n} I(i-m, j-n)K(m,n)) \tag{10}$$

The benefits of the latter is that the there is less variation in the range for values m and n which can make it more straightforward to implement.

In practice many machine learning libraries implement cross-correlation but call it convolution. The reasons being that the commutative property is not really useful in practice but useful when writing proofs. It is the same as convolution but without flipping of the kernel [9]:

$$S(i,j) = (I * K)(i,j) = \sum_{a=m} \sum_{a=n} I(i+m, j+n)K(m,n)) \tag{11}$$

2.4.1 Properties of convolutional neural networks

Convolutional neural networks have three important properties:

- sparse interactions
- shared weights
- equivariant representation.

Convolutional neural networks have **sparse interactions**, meaning that each input does not interact with every output unit, compared to traditional neural networks where that is the case. Sparse interactions are achieved by using a kernel smaller than the input. Having sparse interactions reduces the memory requirements and improves the statistical efficiency of the model.

In comparison to traditional neural networks convolutional neural networks have **parameter sharing**, or tied weights. Instead of only using each element of the weight matrix once when computing, the output layer is is used for more than one function in the model. Each member of the kernel is used in all positions of the input.

Convolutional networks are **equivariant to translation** which means that as the input changes the output changes equally. If two functions are commutative they are equivariant to each other. The order in which two equivariant functions are applied to the input does not affect the result of the output [9].

2.4.2 Pooling

The pooling operation helps make the representation approximately invariant to small input changes, meaning that a small change in the input will not result in different values for most of the pooled outputs.
The pooling operations achieves this by reducing the dimensions of the feature maps of the previous layer. This is done by a statistical summary to retain the most important information. A common pooling operation is max pooling, which takes the maximum value of a group of nearby values [9].

2.4.3 Stride

The step size used for moving the kernel over the input is referred to as stride. Stride can both be applied for the convolution and the pooling operation [9].

2.4.4 Padding

With different configurations of kernel size and stride it is not always the case that it will be a perfect match for the input size [9]. An example of non-perfect match can be seen in Figure 3.
In this study there are two methods applied when the aforementioned scenario occurs.

1. Append zeros to keep an operation from down sampling the size.
2. Discard the non-matching values in the input, so that size is reduced after the operation has been applied.

In this study the first method will be referred to as **SAME** padding and the second will be referred to as **VALID** padding.

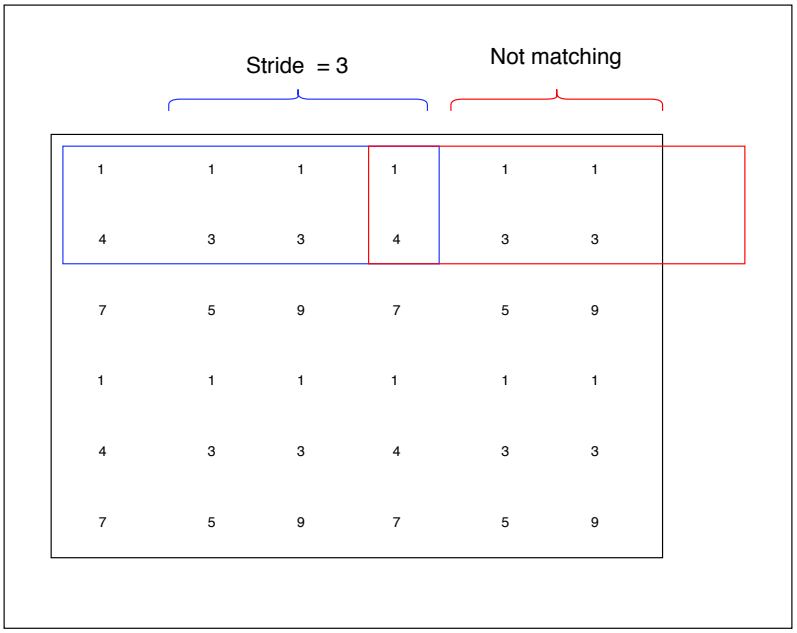


Figure 3: Example of a non-perfect match.

In Figure 3 the stride is 3 and the kernel has a width of 4 and a height of 2. Since the input only has a width of 6 the kernel can only fit once with stride 3. The last two values in the row will not match and therefore there is a non-perfect match. One of the two padding options can then be applied. Either drop the values and thereby reduce the size of the input, or append zeros to the input to keep the same size.

2.4.5 Local response normalization

Local response normalization creates competition for big activities amongst the output from the neurons computed from the different kernels [18]. Local response normalization was used in the AlexNet architecture. Krizhevsky et al. verified that the normalization improved accuracy on the CIFAR-10 dataset using a four layer convolutional neural network with 2% [19].
Local response normalization is defined as:

$$b^i_{x,y} = a^i_{x,y} / \left(1 + \alpha \sum_{j=i-N/2}^{i+N/2} (a^j_{x,y})^2\right)^\beta \tag{12}$$

Where $a^i_{x,y}$ is the activation computed by applying kernel i at position (x, y) where N is the total number of kernels in that layer [19].

2.4.6 Grouped convolution

Grouped convolution separates the filters into filter groups. Each filter group only operates on a subset of the input feature map. Some methods focus on approximating the convolution filters in the spatial domain to reduce the complexity, such as low-rank tensor decomposition. However, grouped convolution focuses on reducing the complexity of the convolutional filters in the channel domain. This reduces the computational complexity and the number of model parameters, without affecting the dimensions of the input and output feature maps [14]. Grouped convolution was introduced in Ref. [19] in order to be able to distribute the training on several GPUs [19, 14].

2.5 Network architectures

A convolutional network can be constructed in a number of ways. This includes varying the number of layers, the type of layers and adding pooling operations [9]. A common network scheme is to repeatedly stack a few convolution layers followed by a pooling operation and use the feed forward layers as the final layers of the network [16]. Examples network architectures include [16, 20, 4]:

- AlexNet
- VGG
- R3DCNN
- GoogleNet

2.6 AlexNet

AlexNet is a famous architecture, first introduced in "Imagenet classification with deep convolutional neural networks" by Krizhevsky et al. which at the time achieved significantly better result on the ImageNet test set compared to the previous state of the art [19].
The AlexNet architecture consists of 8 layers, 5 convolutional layers and 3 fully connected layers. Additionally, AlexNet has 3 pooling operations. The first is located in between the two first convolutional layers, the second after the second convolutional layer and the third is located between the last convolutional layer and the first fully connected layer. Additionally, local response normalization is added after the first and second convolutional layer [16]. Convolutional layers 2, 4 and 5 utilizes grouped convolutions with the number of groups being 2 for each respective layer. The AlexNet Architecture can be seen in Figure 4

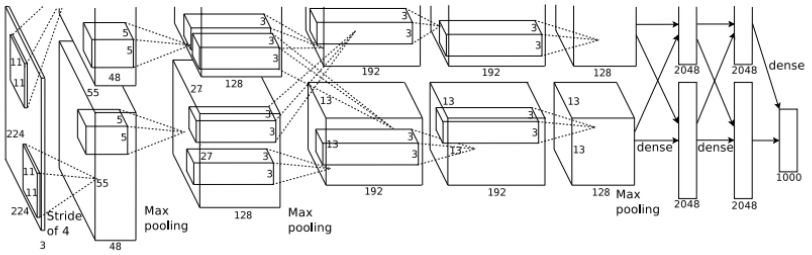


Figure 4: AlexNet network architecture, figure taken from [19].

A more detailed description can be seen in the Table 1, the table is adapted from [16].

Table 1: Original AlexNet architecture (The cells in the table containing ″-″ represents that parameter is not applicably to that layer).

AlexNet								
Layer	**Type**	**Maps**	**Size**	**Groups**	**Kernel size**	**Stride**	**Padding**	**Activation**
Input	Image(RGB)	3	224x224	-	-	-	-	-
Conv 1	Convolution	96	55x55	1	11x11	4x4	SAME	ReLU
lrn 1	Local response normalization	96	55x55	-	-	-	-	-
pool 1	Max pool	96	27x27	-	3x3	2x2	VALID	-
Conv 2	Convolution	256	27x27	2	5x5	1x1	SAME	ReLU
lrn 2	Local response normalization	256	27x27	-	-	-	-	-
pool 2	Max pool	256	13x13	-	3x3	2x2	VALID	-
Conv 3	Convolution	384	13x13	1	3x3	1x1	SAME	ReLU
Conv 4	Convolution	384	13x13	2	3x3	1x1	SAME	ReLU
Conv 5	Convolution	256	13x13	2	3x3	1x1	SAME	ReLU
pool 3	Max pool	256	6x6	-	3x3	2x2	SAME	-
Fully	Fully connected layer	-	4096	-	-	-	-	ReLU
Fully	Fully connected layer	-	4096	-	-	-		ReLU
Fully	Fully connected layer	-	number of classes	-	-	-	-	Softmax

2.7 Transfer learning

Transfer learning is a method to utilize pretrained weights for a different problem domain. Transfer learning utilises the property of the early layers learning more generic properties and the later layers learning more specific properties of the problem domain. Transfer learning is an essential aspect of this study and is commonly used when there is a limited amount of data for the specific problem or when you have limited time for training. There are two methods how to approach the training of the network on the new problem domain [21]:

1. Retrain all the layers of the network
2. Let the layers with transferred parameters remain constant during training, leaving them so called **frozen**.

The first method can be used if the dataset for the new problem domain is large or the number of parameters in the model is small while the second methods is best suited when the dataset for the new problem domain is small and the number of parameters in the model is large. The second method prevents overfitting of the model [21]. A simplified illustration of transfer learning can be seen in Figure 5.

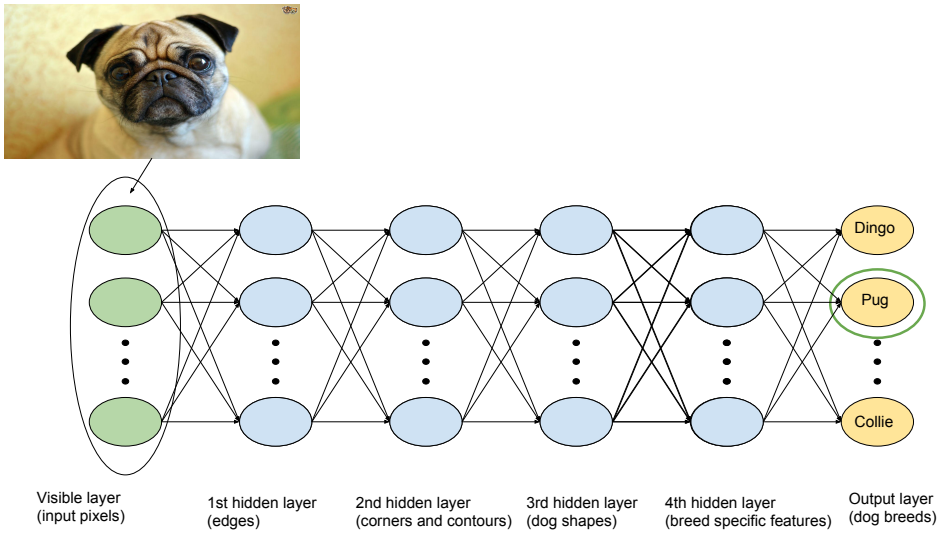


Figure 5: Simplified illustration of transfer learning (figure adapted from [9]).

2.8 Data augmentation

Data augmentation is a method to increase the size of the training set, make it more generalized and reduce overfitting. This is essential when there is a small number of training samples [22, 19]. worth noting is while the training set becomes larger there is still a high interdependency between the images [19].

Data augmentation operations applied to images can include [23] :

- Horizontal and vertical flipping
- Light and contrast manipulation
- Color manipulation
- Cropping
- Rotation

2.9 Rank decomposition

Rank decompositon is a factorization of a matrix.
For a matrix $M \in \Re^{mxn}$, a r-rank decompostion would be a product of two matrices $A \in \Re^{mxr}$ and $B \in \Re^{rxn}$ according to $M = AB$. [24]

2.10 Low-rank tensor decomposition

In this study low-rank tensor decomposition is investigated as a optimization technique to increase inference speed when the training data is limited. In "Convolutional neural networks with low rank regularization"[4] Tai et al. propos a method for computing low-rank tensor decomposition for decomposing tensors. The idea of using tensor decompositions is based on that there could exist a significant amount of redundancies in a tensor. This property can be used in convolutional neural networks in order to increase the inference speed of the network and reduce the size of the network by removing these redundancies in the convolutional kernels. Reducing the size of the network is crucial for deploying neural networks in mobile devices, due to memory limitations as well as bandwidth and latency constraints.
The proposed method uses Singular Value Decomposition in order to efficiently construct a low

rank decomposition of a tensor. Singular Value Decomposition (SVD) is a method for decomposing matrices into singular vectors and singular values. Similar to eigendecomposition, factoring matrices can help in analyzing certain properties of the matrix [9].
While using tensor decomposition results in a deeper network, utilizing batch normalization reduces the training difficulty of having a deeper network. Tai et al. managed to train a 30 level deep neural network. The proposed method was tested on AlexNet as well as other network architectures, such as, VGG and GoogleNet and achieved a speed-up of at most 1.82x, 2.05x and 1.2x respectively.

3 Previous work

In this section, a literature survey of related previous work is presented.

There have been several studies using different methods to minimize the network size [4, 3, 20, 2]. The methods include:

- tensor decompositions
- compression
- alternative network architectures

Tensor decompositions have been shown to substantially reduce the amount of computations for convolution [4].
There have been several studies on compressing the fully connected layers as well, using different methods [20, 3, 25].
Methods include:

- different types of pruning methods
- inference of compressed networks
- randomly grouped connection weights.

3.1 Structured sparsity

In "Learning Structured Sparsity in Deep Neural Networks" [26] Wen et al. propose a Structured Sparsity learning (SSL) method to regularize the filters, channels, filter shapes, and layer depth, i.e the structures of a convolutional neural network. SSL can be used to efficiently accelerate the evaluation of the network but also to increase accuracy by operating as a regulator. This is achieved by learning a compact structure based on that of a bigger neural network which reduces the computational cost. Additionally, compared to other approaches that result in a non structured random connectivity in the deep neural network, structured sparsity does not result in irregular memory accesses. The proposed method achieves a 5.1x and 3.1x speed-up on a CPU and GPU, respectively, for AlexNet with improved accuracy. The improved accuracy is obtained by the SSL operating as regularization.

The structured sparsity learning is performed by using a structured sparsity regularization applied on each layer, in addition to applying a non-structured regularization for each weight. The methods used for structured and non-structured sparsity regularization were group Lasso and l2 norm, respectively. The generic optimization target was formulated as:

$$E(W) = E_D(W) + \lambda R(W) * \lambda_g \sum_{l=1}^{L} R_g(W^{(l)}) \tag{13}$$

Where, W is the set of all the weights in the network, $E_D(W)$ is the loss on the data, $R(W)$ is the non-structured regularization applied on each weight, and $R_g(W^{(l)})$ is the structured regularization applied on each layer.

The proposed method utilizes group Lasso regularization to learn a compressed structure of the deep convolutional neural network during training. The generic optimization is then specified to:

penalizing unimportant filters and channels, learning arbitrary shapes of filters and regularizing layer depth. For specific details see the original paper [26].

3.2 Deep compression

In "Deep compression: compressing deep neural networks with pruning, trained quantization and huffman coding" [27] Han et al. propose a compression scheme with the purpose to be able to deploy neural networks on embedded and mobile devices. The proposed scheme compressed the storage requirements for AlexNet by a factor of 35, from 240MB to 6.9MB and notable without loss of accuracy. This is important for mobile devices where the bandwidth and applications size are restricted. Furthermore, minimizing the energy consumption is also of importance for mobile applications. The energy consumption and storage size of the network have made it difficult to deploy large networks as mobile applications. Reducing the size of the network would result in reduced bandwidth usage for fetching weights, and would also require less computations. This would reduce the power consumption since the aforementioned tasks are energy consuming. The compression scheme presented in the paper is a 3 stage pipeline.
The stages are, in order:

- Pruning
- Quantization
- Huffman encoding

The pruning is conducted in an unstructured approach by removing connections with weights below a certain threshold. The pruning effectively reduces the network size. However, reducing the size of the network does not necessarily result in increased inference speed since current hardware exploits regularities in computations [20].
The quantization reduces the number of bits required for representing each weight. The sharing of weights is performed at each layer by applying k-means cluster. For each cluster a reduce operation is then performed; The weights grouped together in the same cluster will end up with the same weight. The weight sharing is performed after the network has been trained. Thus, the shared weights are approximated weights of the network. The centroid initialization for the cluster algorithm has an impact on the quality of the clustering. Interesting results in Ref. [27] suggested that the pruning and the quantizations were orthogonal, i.e. pruning did not affect the result of the quantization and vice versa. The result suggests that the a different pruning method could be applied in combination with the quantization. The encoding of the network is done by Huffman encoding which utilizes that some values occur more often than others. This property allows for the more frequently used values to be represented as fewer bits [28]. The compression did not result loss of accuracy.

In "Efficient inference engine on compressed deep neural network" [29] Han et al. proposes an efficient inference engine that operates on a deep compressed network. The compression scheme in combination with the inference engine achieves 189x and 13x speed-up using a CPU and GPU, respectively, when operating on the same deep neural network without compression. This is not applicable for the scope of this thesis since it requires the specific hardware accelerator.

3.3 Pruning

An alternative pruning approach is iterative pruning as proposed in "Pruning convolutional neural networks for resource efficient inference" [20] The purpose of pruning is to remove the least important parameters while minimizing the difference in accuracy compared to the unpruned network. The method of which the least important parameters are determined. The authors describe the optimal method of pruning, i.e. an exact empirical evaluation of each parameter by removing each non-zero parameter $w \in W'$ and record the difference in cost, where W' is a subset of parameters of W. The authors denote the optimal pruning method as the oracle criterion. Additionally, due to the oracle criterion being infeasible in practice, in terms of computational cost, they also describe different heuristics as well as propose their own heuristic based on the Taylor series expansion. The authors minimize the cost difference between the pruned and the

unpruned network for the output h_i and parameter i by approximating the change in the loss function. The approximations is done by the first order Taylor expansion excluding the first order reminder due to the computational cost. The method proposed was applied to AlexNet, VGG and R3DCNN using different hardware. The speed-up obtained with fine tuning was at minimum $1.9x$ and at most $2.4x$ for AlexNet, $1.7x$ to $2.5x$ for VGG and $5.2x$ for R3DCNN.

3.4 Hashing Trick

In "Compressing Neural Networks with the Hashing Trick "[25] Chen et al. compresses the network by applying a hashing trick to randomly group connection weights into hash buckets by utilizing a low cost-hash function. The weights in the same hash bucket share the same parameter values. The proposed method performs random weight sharing without requiring extra storage, which is needed in the naive approach. The training is performed on the shared parameter values. The authors suggests that the method could be used in combination with other compression methods.

The following is a simple example of applying a hash function to group weights into hash buckets:
Let w be a unhashed vector: $[1.1, 1.2, 1.1, 1.1, 1.2, 1.3]$
and let $h()$ be a low-cost hash function.
The hash function is applied on on each element in w which results in
w': $[h(1,1), h(1,2), h(1,3)]$.

3.5 ShuffleNet

In "ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices" [2] Zhang et al. proposes ShuffleNet, a small network architecture designed to specifically fit on a mobile device with limited resources in terms of computing power. ShuffleNet uses grouped convolutions and a channel shuffle technique to keep the flow between channel groups. The purpose of the channel shuffle technique is to maintain representation that otherwise is lost using only grouped convolutions. ShuffleNet manages to get a $13x$ speed-up compared to AlexNet.

The shuffle technique is performed by the following 3 steps, in order:

1. Reshaping the output from a convolutional layer with g groups and gxn output channels into (g, n)
2. Performing matrix transposing on (g, n)
3. Flattening the output to be used as the input for the next layer

Example:
Given the following tensor A with dimensions WHN where W is the width, H is the height and N is the output channels. Then define g to be the number of groups and n to be $\frac{N}{g}$ output channels per group. The shuffle technique would perform the following operations:

1. Reshape A into $WHgn$
2. Transpose on (g, n) which results in $WHng$
3. Flatten the output into WHN

The groups have now been successfully shuffled.

4 Experimental Set-up

In this section, the technical aspects required to conduct the experiments are presented.

4.1 Hardware

The hardware used in this project was an instance of NC6, of the N-series, virtual machine on the Azure cloud. The N-series is ideal for compute intensive workloads such as deep learning [30]. The NC6 virtual machine has the following specifics:

- CPU: Intel Xeon E5-2690v3
- GPU: NVIDIA TESLA K80 (access to half a card)
- Available GPU memory: 11GiB

4.2 Dataset

The dataset used for training and evaluation was obtained from Kaggle. The dataset was part of the ImageNet dataset and can be found at [31]. The dataset consisted of 10222 images containing 120 dog breeds. The dataset was divided into training, validation and test sets.

4.2.1 Training

6133 images were used for fine-tuning the network. Adam was used for efficient stochastic optimization since it has shown to demonstrate it to work well in practice [11].

4.2.2 Validation

The valdiation set contained 2044 images.

4.2.3 Test

The test set contained 2044 images. Worth noting is that while the images were not part of the fine-tuning process, they could have been part of the training for the pretrained weights. ImageNet consists of more than 14 million images [19]. The training set consists of 1.2 million images [7]. The pretrained weights were trained on the ILSVRC 2012 training set. The number of Images used to calculate the pretrained weights was 600-times the number of test images, which could negate the effect of the network being trained on these images. The reason for using the test images was due to a lack of a test set with images guaranteed not to have been seen during the calculation of the pretrained weights. Additionally, the focus of this study is not on the state of the art performance in regards to accuracy, but instead on how the decomposition can be applied to increase inference speed while **maintaining** the accuracy when the training set is limited and is not by itself enough to train the model from scratch. However, the ideal scenario would have been to have a test set guaranteed to **not** have been part of the dataset used for calculating the pretrained weights.

4.3 Tensorflow

The implementation was done in Python using the TensorFlow library. The workflow in TensorFlow consists of two phases [32]:

1. Definition of the neural network as a symbolic dataflow graph. During the development of the project feed_dict mechanism was used in order to feed the network with input. This is achieved by defining placeholders for the inputs and the labels during the definition phase of the network.
2. Execute the optimized version of the neural network, which was done on a single GPU.

4.3.1 feed_dict mechanism

During the development of the project the feed_dict mechanism was used in order to feed the network with data. According to the documentation this is not recommended to use for other than on trivial problems and using feed_dict with large inputs often results in suboptimal performance. In general it does not provide a scalable solution [33]. However, as also stated in the documentation, if the model is run on a single GPU the performance difference compared to using the alternative may be negligible. The reason for using the feed_dict mechanism was to simplify the process of training and evaluating the model in the same session.

4.3.2 TFRecords

The images used for training and evaluating AlexNet had a width of 227 and a height of 227 in all 3 channels, an image is therefore a tensor of dimension [227 x 227 x 3]. With every value in the image represented as a 32bit float, the total amount of bytes required to store all images of the training-, validation- and testing set are:

- **Training set:** $227 * 227 * 3 * 6133 * 32bits \approx 3.8GB$.
- **Validation set:** $227 * 227 * 3 * 2044 * 32bits \approx 1.3GB$.
- **Testing set:** $227 * 227 * 3 * 2044 * 32bits \approx 1.3GB$.
- **Total amount of memory required:** $\approx 6.4GB$

It was not possible to load every image directly into memory. The available memory was $11GiB$, as stated in section **4.1 Hardware**, and loading all the images required $\approx 6.4GB$ of memory. This leaves $\approx 4GB$ of available memory left. Additionally, the network itself also needs to be loaded into memory, in Ref. [19], AlexNet required $\approx 3GB$ when using a batch size of 128 images [19]. This would in theory make it possible to load every image into memory, since there would still be $1GB$ of available memory left. However, this was not the case.
It was not possible to load every image into memory, the reason for this could be due to the memory required to load tensorflow, but also poor memory management in tensorflow. Therefore only a set number of images could be stored in memory at any given time.
To achieve this, TFRecords were used. TFRecords enables storing images as a binary file. By using TFrecords consecutive batches of images could be loaded into memory. A batch of 100 images would require the following number of bytes:

- **Batch of 100 images:** $227 * 227 * 3 * 100 * 32bits \approx 62MB$

$62MB$ being considerably less than $6.4GB$.
A TFRecord was created for both training, validation and testing.
The batches were fed into the network by using the feed_dict mechanism, which allows the network to be fed with different datasets [34]. The feed_dict mechanism, makes it trivial to evaluate the network during training.

4.4 Data augmentation

The augmentations used in this study were:

- flipping horizontally
- flipping vertically
- lightning adjustments by changing contrast
- rotation with a random angle

For each image the resulting number of images after the data augmentations have been applied is 5. One being the original unchanged image and the others being augmented. Each augmented image was assigned with the label of the original one. An example of data augmentations used in this study can be seen in Figure 6.

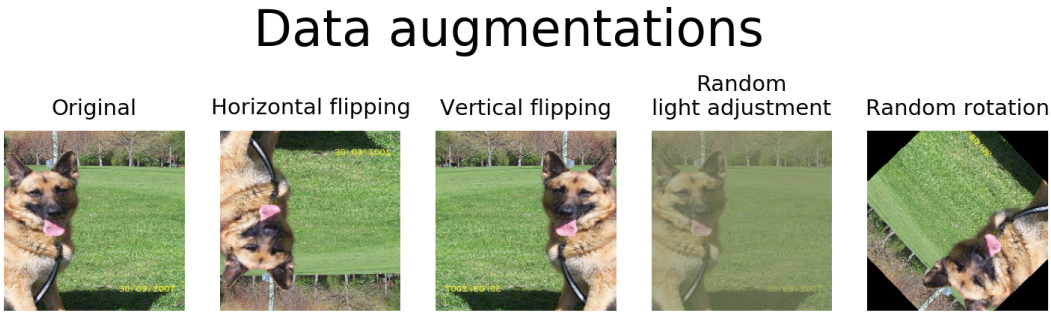


Figure 6: Used data augmentations

5 Methodology for Low-rank decomposition

In this section, the methodology for how the low-rank decomposition was applied in this study is presented.

5.1 Motivation

Low-rank decomposition was chosen as the method to increase the inference speed because it had been previously shown to increase the inference speed on AlexNet by 1.82%, with no significant drop in accuracy [4]. The method can be applied when a trained network is already available, which could be the case for many real world scenarios. Additionally, low-rank decomposition is also fairly trivial to implement which makes it a good starting point, where future work could include other methods, described in section **3. Previous work**.

5.2 Overview of how to decompose the network

The following sequential methodology was used to create the final decomposed network:

1. The original network structure was defined
2. The pretrained weights were used to train the original network on the problem domain
3. The chosen convolutional layers using the low-rank decomposition, with specified Ks, were decomposed and the weights, including the weights of the non decomposed layers, were saved
4. The decomposed network structure was created by substituting the chosen convolutional layers with the decomposed layers
5. The decomposed network was trained on the training set.

5.3 Method for low-rank decomposition

5.3.1 Algorithm

The method used for decomposition is low-rank regularization which was introduced in Ref. [4]. The decomposition is applied to the convolutional kernels of the original trained network. After decomposing a kernel, a depth wise and a height wise filter kernel is returned. Then a new network is defined using the two new kernels.

s : size of depth and height for input
$x \in \mathfrak{R}^{s*s*C}$: input
w : Kernel width
d : Kernel depth
C : Number of channels
N : Number of filters
$W \in \mathfrak{R}^{w*d*C*N}$: Original kernel
k : Parameter for low-rank decomposition
$Oconv \in \mathfrak{R}^{w*d*C*N}$: Original convolutional layer
$Dconv \in \mathfrak{R}^{w*d*C*N}$: Decomposed convolutional layer
$f(W,k)$: Function for decomposition
$\hat{V} \in \mathfrak{R}^{w*1*C*K}$: Decomposed width kernel
$\hat{H} \in \mathfrak{R}^{1*d*K*N}$: Decomposed height kernel

$$\hat{V}, \hat{H} = f(W,k) \tag{14}$$

$$W \in \mathfrak{R}^{w*d*C*N} \tag{15}$$

$$\hat{V} \in \mathfrak{R}^{w*1*C*K} \tag{16}$$

$$\hat{H} \in \mathfrak{R}^{1*d*K*N} \tag{17}$$

$$Oconv = convolution(x, W) \Rightarrow \tag{18}$$

$$Dconv = convolution(convolution(x, \hat{V}), \hat{H}) \tag{19}$$

Equation (14) is the decomposition of kernel W with rank parameter k, which results in a vertical kernel $\hat{V}$ and a horizontal kernel $\hat{H}$.
Equation (15) - (17) shows the dimensions of W, $\hat{V}$, $\hat{H}$, respectively. The original convolution with kernel W on input x is shown in equation (18). Equation (19) is the resulting convolution using the two decomposed kernels instead, where the input x is first convolved with kernel $\hat{V}$ and then the result of that is convolved with $\hat{H}$.

Figure 7 illustrates the benefit of utilizing the decomposition. Instead of having one convolutional operation, using a large kernel on a large input, two smaller kernels are used. The vertical kernel is operating on the same input as the original kernel, and the horizontal kernel is operating on a smaller input, $K < N$. The resulting output has the same size for the convolution, using the original kernel, and the two consecutive convolutions, using the decomposed kernels.

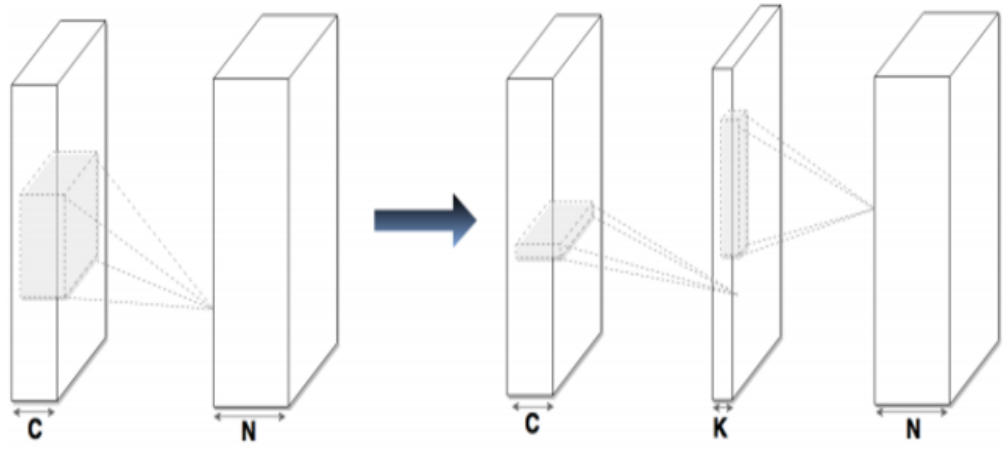


Figure 7: Illustration for low-rank decomposition where the left is the original convolutional layer and the right is the low-rank constraint convolutional layer with rank-K. Figure taken from [4].

Computing the decomposed kernels
The function for kernel decomposition creates a bijection from $W \in \Re^{w*d*C*N} \rightarrow Wb \in \Re^{wC*dN}$ where the tensor element (i_1, i_2, i_3, i_4) maps to (j_1, j_2), according to equation (20).

$$j_1 = (i_1 - 1)d + i_2, \quad j_2 = (i_4 - 1)d + i_3 \tag{20}$$

Matrices U, D and Q is then calculated by applying Singular Value Decomposition on Wb, according to equation (21).

$$UDQ^T = SVD(Wb) \tag{21}$$

$\hat{V}$ and $\hat{H}$ can then be calculated, according to equations (22) and (23).

$$\hat{V}_k^c(j) = U_{(c-1)d+j,k} \sqrt{D_{k,k}} \tag{22}$$

$$\hat{H}_n^k(j) = Q_{(n-1)d+j,k} \sqrt{D_{k,k}} \tag{23}$$

5.4 Implementation
In this section, the code for the low-rank decomposition, used in this study, is presented.

```
89
90 def low_rank_regularization(tensor_to_be_decomposed, K, sess):
91     """
92     Calculates the low rank decomposition with rank K of a tensor of shape [d1,d2,C,N] ->
93     V[d1,1,C,K]
94     H[1,d2,K,N]
95
96     Input:
97         tensor_to_be_decomposed: np.array to be decomposed,
98         K: (integer) low rank K to be used in decomposition
99         sess: current session
100
101    Output: V[d1,1,C,K], H[1,d2,K,N]
102
103    """
104
105    shape_of_tensor = tensor_to_be_decomposed.shape
106    print(shape_of_tensor)
107    #== get dimensions ===
108    d1, d2, C, N = get_dimensions_of_4d_shape(shape_of_tensor)
109    # ========================
110    W_data = tensor_to_be_decomposed
111    V = np.zeros((d1,1,C,K))
112    H = np.zeros((1,d2,K,N))
113
114    #==== calculate bijection from W[i1,i2,i3,i4]- > W_bijecttion[j1,j2] ===
115    W_bijection = calc_bijection(W_data, C, d1, d2, N)
116    print(W_bijection.shape)
117
118    #=== calculate SVD decomposition of bijection of W
119    # tf.svd(W) returns d,u,q such that W  =  u*diag(d)*transpose(conj(q))
120
121    D, U, Q = tf.svd(W_bijection)
122
123    # get as numpy arrays
124    D, U, Q = convert_D_U_Q_from_tensor_to_np_array(D, U, Q, sess)
125    V, H = calculate_V_H(V, H, K, shape_of_tensor, D, U, Q, sess)
126
127    return V, H
128
```

Figure 8: Complete algorithm to compute low-rank decomposition.

In Figure 8 the code for calculating the low-rank decomposition is presented. First, a bijection is calculated using the calc_bijection() function, which can be seen in Figure 9. Secondly, D, U, Q are calculated using the TensorFlow implementation of Singular Value Decomposition by calling the function tf.svd(). Third, the $\hat{V}$ and $\hat{H}$ are calculated, using the calculate_V_H() function, which can be seen in Figure 10.

```
35
36 def calc_bijection(W_data, C, d1, d2, N):
37     """
38     calculates the bijection from W[i1,i2,i3,i4]-> W_bijecttion[j1,j2]
39     Input: W numpy array, num channels C, filter height d1, filter withd d2, number of outputs N.
40     """
41
42     W_bijection = np.zeros((C*d1, N*d2))
43     for i1 in range(C):
44         for i2 in range(d1):
45             j1 = i1 * d1 + i2
46             for i3 in range(d2):
47                 for i4 in range(N):
48                     j2 = i4 * d2 + i3
49                     W_bijection[j1,j2] = W_data[i2,i3,i1,i4]
50     return W_bijection
51
```

Figure 9: code to calculate the bijection.

In Figure 9, the code for calculating the bijection, according to equation (20) in section **5.3.1 Algorithm**, is presented.

```
54
55 def calculate_V_H(V, H, K, shape, D, U, Q, sess):
56     """
57     Calculate V, H
58     Input:
59         V: np.array of dim [d1,1,C,K] to be filled
60         H: np.array of dim [1,d2,K,N] to be filled
61         K: low rank param K
62         shape: shape of [d1,d2,C,N]
63         D: np.array
64         U: np.array
65         Q: np.array
66     Output:
67         V: np.array V with calculated values
68         H: np.array H with calculated values
69     """
70
71     d1,d2,C,N = get_dimensions_of_4d_shape(shape)
72     for k in range(K):
73         D_part = sess.run(tf.sqrt(D[k]))
74         for c in range(C):
75             for j in range(d1):
76                 U_part = U[(c*d1)+j, k]
77                 V_j = U_part*D_part
78                 V[j,0,c,k] = V_j
79
80         for n in range(N):
81             for j in range(d2):
82                 Q_part = Q[(n * d2) + j, k]
83                 H_j = Q_part * D_part
84                 H[0,j,k,n] = H_j
85     return V, H
86
87
```

Figure 10: Code to calculate $\hat{V}$ and $\hat{H}$.

In Figure 10, the code for calculating $\hat{V}$ and $\hat{H}$, according to equation (22) and (23) in section **5.3.1 Algorithm**, is presented.

5.5 Original network architecture

The network architecture chosen for the original network was AlexNet with added batch normalization applied after the layers with unfrozen weights. Additionally, the input size of the image used in this project was [227 x 227], instead of [224 x 224], which was used in the original architecture [19].
In addition to using larger input images, **VALID** padding was used in the first convolutional layer. The reason for using input size [227 x 227] was to be able to use the pretrained weights which had been trained on images with dimensions [227 x 227]. VALID padding was used to down sample the size of the input to get the same dimensions for the layers as the pretrained weights.

The details of the network used as a baseline in this project can be seen in Table 2.

Table 2: Modified AlexNet architecture, used as baseline (The cells in the table containing "-" represents that parameter is not applicably to that layer).

AlexNet								
Layer	**Type**	**Maps**	**Size**	**Groups**	**Kernel size**	**Stride**	**Padding**	**Activation**
Input	Image(RGB)	3	227 x 227	-	-	-	-	-
Conv 1	Convolution	96	55 x 55	1	11x11	4x4	VALID	ReLU
lrn 1	Local response normalization	96	55 x 55	-	-	-	-	-
pool 1	Max pool	96	27 x 27	-	3x3	2x2	VALID	-
Conv 2	Convolution	256	27 x 27	2	5x5	1x1	SAME	ReLU
lrn 2	Local response normalization	256	27 x 27	-	-	-	-	
pool 2	Max pool	256	13 x 13	-	3x3	2x2	VALID	-
Conv 3	Convolution	384	13 x 13	1	3x3	1x1	SAME	ReLU
Conv 4	Convolution	384	13 x 13	2	3x3	1x1	SAME	ReLU
Conv 5	Convolution	256	13 x 13	2	3x3	1x1	SAME	ReLU
pool 3	Max pool	256	6 x 6	-	3x3	2x2	VALID	-
Fully	Fully connected layer	-	4096	-	-	-	-	ReLU
dropout	Dropout operation	-	-	-	-	-	-	-
Fully	Fully connected layer	-	4096	-	-	-	-	ReLU
dropout	Dropout operation	-	-	-	-	-	-	-
Fully	Fully connected layer	-	120	-	-	-	-	Softmax

5.5.1 Training

Since the training set was limited, **Transfer learning** was used. The precalculated weights can be found at [35]. As described in **2.7 Transfer Learning**, earlier layers learns more general features while later layers learns more domain specific features. The network was initialized with frozen pretrained weights on the ImageNet dataset for all layers except the two last feed forward layers. The last two feed forward layers were initialized with a truncated normal distribution and were the target for training. How many layers to freeze and retrain can be fine-tuned but initial tests suggested that the described setup was suitable.

Batch normalization To the best of my knowledge there does not exist a general consensus if to apply batch normalization before or after the activation function. In Ref. [36] Mishkin and Matas evaluated both methods. The authors concluded that the result varied but in most cases applying batch normalization after the activation function performed better.
Local response normalization The same parameters as in Ref. [19, 16] for the local response normalization were used.

Dropout During training of the original network, the dropout used in this study was 0.3 after the first and second fully connected layer.

5.5.2 Validation and testing

For the validation and test set a batch from the respecitve TFRecord was retrieved and no augmentations were applied.

5.5.3 Decomposition of learned weights

Low-rank regularization is applied to the final weights with a specified K for each of the chosen layers to be decomposed.

5.6 Decomposed network architecture

The decomposed network is created by consecutive convolutions with the decomposed weights V_k and H_n. Activation and batch normalization is applied after convolving with H_n.
A fully decomposed network architecture can be seen in Table 3. The K1, K2, K3, K4 and K5, in Table 3, represents the value of K for the low-rank tensor decomposition for convolutional layer 1, 2, 3, 4, 5, respectively.

Table 3: Architecture of a fully decomposed AlexNet The cells in the table containing "-" represents that parameter is not applicably to that layer).

AlexNet								
Layer	**Type**	**Maps**	**Size**	**Groups**	**Kernel size**	**Stride**	**Padding**	**Activation**
Input	Image(RGB)	3	227 x 227	-	-	-	-	-
Conv V_k	Convolution	K1	55 x 227	1	11x1	4x1	VALID	ReLU
Conv H_n	Convolution	96	55 x 55	1	1x11	1x4	VALID	ReLU
lrn	Local response normalization	96	55 x 55	-	-	-	-	-
pool 1	Max pool	96	27 x 27	-	3x3	2x2	VALID	-
Conv 2 V_k	Convolution	K2	27 x 27	2	5x1	1x1	SAME	ReLU
Conv 2 H_n	Convolution	256	27 x 27	1	1x5	1x1	SAME	ReLU
lrn	Local response normalization	256	27 x 27	-	-	-	-	-
pool 2	Max pool	256	13 x 13	-	3x3	2x2	VALID	
Conv 3 V_k	Convolution	K3	13 x 13	1	3x1	1x1	SAME	ReLU
Conv 3 H_n	Convolution	384	13 x 13	1	1x3	1x1	SAME	ReLU
Conv 4 V_k	Convolution	K4	13 x 13	2	3x1	1x1	SAME	ReLU
Conv 4 H_n	Convolution	384	13 x 13	1	1x3	1x1	SAME	ReLU
Conv 5 V_k	Convolution	K5	13 x 13	2	3x1	1x1	SAME	ReLU
Conv 5 H_n	Convolution	256	13 x 13	1	1x3	1x1	SAME	ReLU
pool 3	Convolution	256	6 x 6	-	3x3	2x2	VALID	-
Fully	Fully connected layer	-	4096	-	-	-	-	ReLU
Fully	Fully connected layer	-	4096	-	-	-	-	ReLU
Fully	Fully connected layer	-	120	-	-	-	-	Softmax

5.6.1 Training

The layers trained for the decomposed networks were only the decomposed layers. Batch normalization was applied after each trained layer. The amount of retraining for each network configuration varied and required at most 200 **epochs** to achieve the results presented in this study.

6 Experiments

In this section, the conducted experiments in this study are presented, as well as how they were done.

6.1 Different values for K

The experiments were conducted with different configurations of Ks as can be seen in Table 4. A configuration used in a previous study Ref. [4], conducted by Tai et al., was used as a starting point.

Based on the result of the configuration used in the previous study, other configurations were devised. The different configurations were found by manually experimenting with different values for K for one layer at a time and monitoring the effects of the parameter for that particular layer, both in terms of accuracy of the whole model as well as the time for that layer. After a parameter was found to reduce the computational time of that layer and not impact the accuracy more than 5%, for the top-1 accuracy, it was used in combination with other decomposed kernels.

Table 4: Values for K for decomposed AlexNet (Left out parameter means that the layer remain undecomposed)

Configurations for K					
	Conv 1	**Conv 2**	**Conv 3**	**Conv 4**	**Conv 5**
	K 1	**K 2**	**K 3**	**K 4**	**K 5**
Previous study	8	40	60	100	200
Experiment 1	8	-	-	-	-
Experiment 2	33	-	-	-	-
Experiment 3	8	-	60	100	-
Experiment 4	-	-	60	-	-
Experiment 5	-	-	-	100	-
Experiment 6	-	-	100	110	-
Experiment 7	-	-	60	100	-
Experiment 8	-	-	256	384	-

6.2 Profiling

The profiling was done using **timeline**, a function for creating a time and memory trace in Json format, which can be visualized using chrome://tracing. Using timeline enables fine grained profiling for the operations of interest.

The results of the experiments were averaged over 3 runs since the time of each operation varied slightly from run to run, with the maximum value of the standard deviation, in any layer, for all runs, for all experiments being $0.343ms$.

7 Result

In this section, the results of the study are presented.

7.1 Inference time

As a first step the average computation time for the operations in the original network were profiled, using a batch size of 100 images. The reason for profiling the original network was to get a base line computational time for the different operations, as well as to get information regarding which operations was the most computationally expensive.

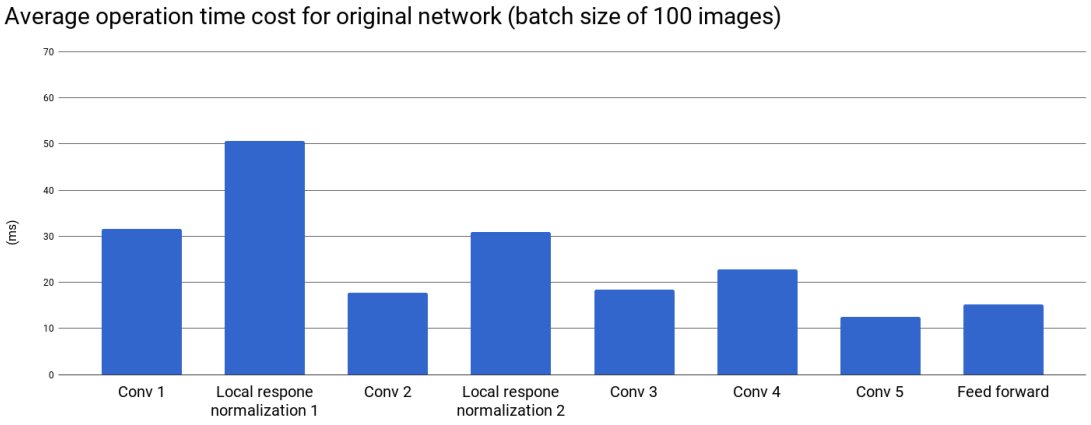


Figure 11: Operational costs original network. No standard deviation is shown in the figure since it was at most, for all experiments, 0.343ms $\approx$ 1% of the computation time.

On the x-axis in Figure 11 the different operations are presented, the y-axis is the computational time in milliseconds. Figure 11 shows that the local response normalization is a time consuming operation, the first and the second local response normalization being the first and third most expensive operation, respectively. Additionally, it is appherent that the convolutional layers take up more time than the feed forward layers.

7.2 Impacts of various values for Ks

7.2.1 Time

As a second step the average time of the original network and the network configuration used in Ref. [4] was compared.
In Figure 12 the average computational time for the different convolutional layers in the original network are represented as blue bars and the configuration used in Ref. [4] represented as red bars. The x-axis represents the different convolutional layers and the y-axis shows the computational time in milliseconds. The K1, K2, K3, K4 and K5 represents the value of K for the low-rank tensor decomposition for convolutional layer 1, 2, 3, 4, 5, respectively.

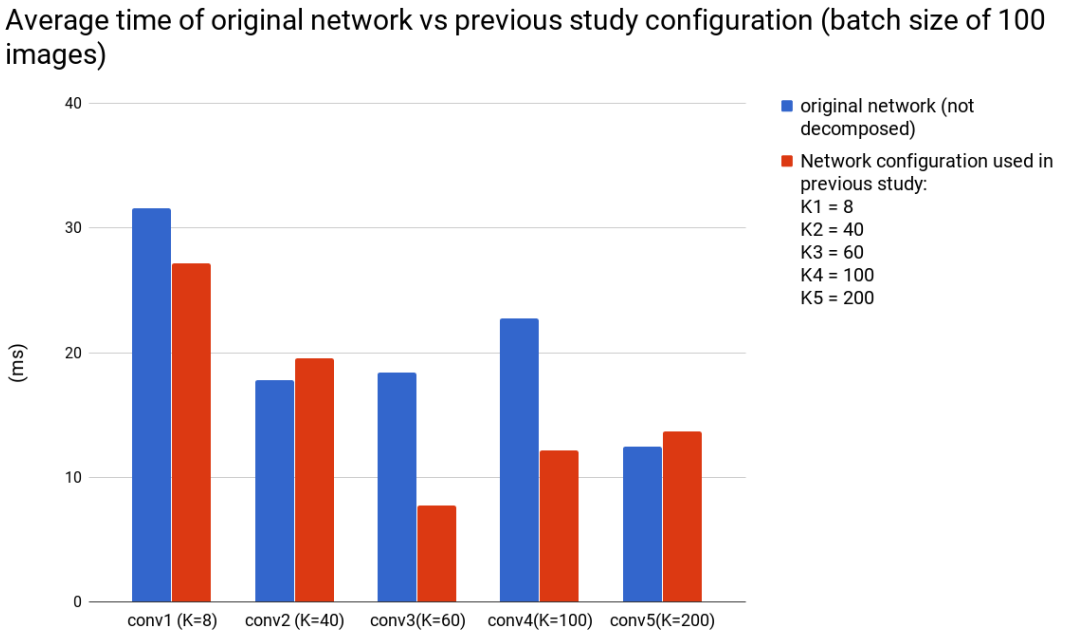


Figure 12: Original network compared to the decomposed network, using the values for K used in Ref. [4] with a batch size of 100 images. No standard deviation is shown in the figure since it was at most, for all experiments, 0.343ms $\approx$ 1% of the computation time.

In Figure 12 the time for the convolutional layers of the original network and the decomposed network, using the same values for K used in Ref. [4], are compared. As can be seen in the figure the computational time for convolutional layer 2 and 5 is actually greater for the decomposed network.

As a third step the same configurations as in Figure 12 was profiled, using a batch size of 2044 images instead of 100 images. The result of which can be seen in in Figure 13. This was done in order to see the effects of using a larger batch size. 2044 was the number of images in the entire test set.

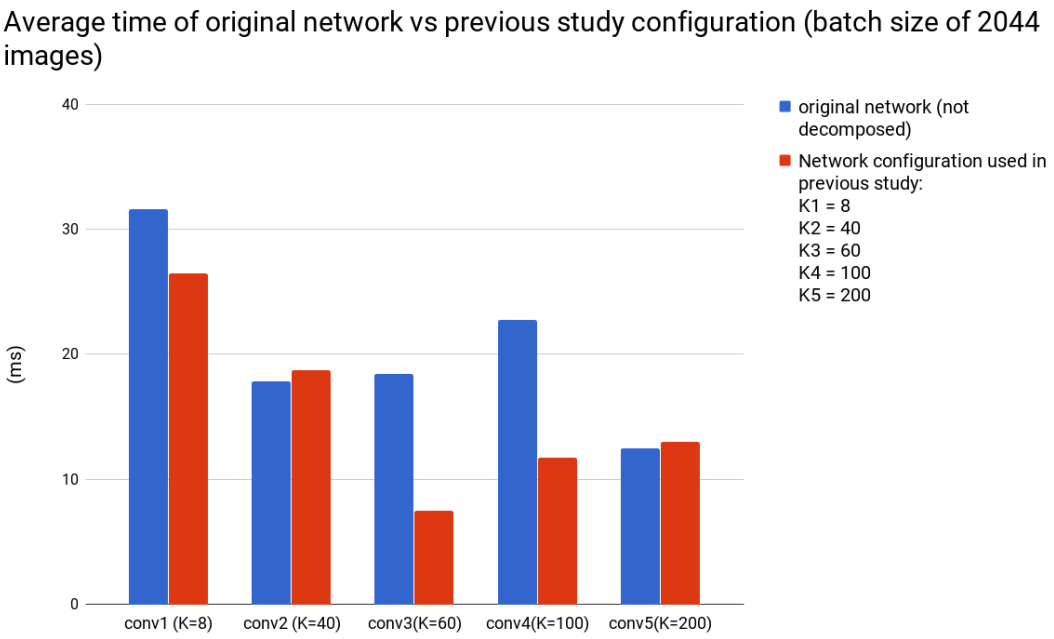


Figure 13: Original network vs decomposed network, using the values for K used in Ref. [4] with a batch size of 2044 images, the same size as the entire training set. No standard deviation is shown in the figure since it was at most, for all experiments, 0.343ms $\approx$ 1% of the computation time.

Comparing Figures 12 and 13 it can be seen that having a larger batch size results in the computational time for the decomposed convolutional layers 2 and 5 being closer to the original computational time for respective layer. However, convolution layer 2 and 5 still performs worse for the decomposed network. The effect of increasing the batch size from 100 to 2044 is still negligible.

In Figure 14 the average computational time for the convolutional layers for the different configurations of Ks, described in section **6.1 Different values for K**, as well as for the original network is presented. The different colors in the figure represent different network configurations. The x-axis represents the different convolutional layers and the y-axis represents computational time in milliseconds.

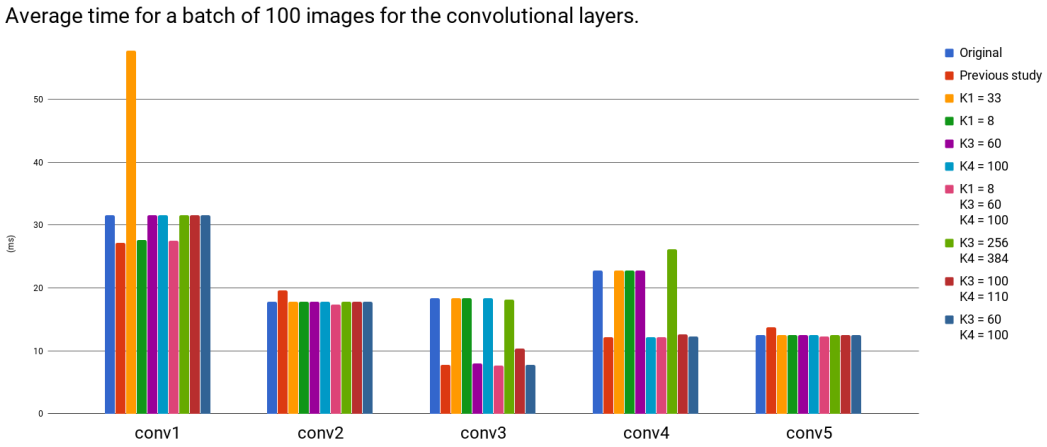


Figure 14: Results for different configurations of parameters K for different layers. No standard deviation is shown in the figure since it was at most, for all experiments, 0.343ms $\approx$ 1% of the computation time.

In Figure 14 it can be clearly seen that the decomposition has an effect on the time for the convolutional operations. However, as can bee seen in the first convolutional layer, having $K1 = 33$ has a negative impact on the time of the operation. Having $K1 = 8$ has a positive impact, resulting in the time being reduced with $> 10\%$ for a batch of 100 images.
Figures 12, 13 and 14 clearly show that the decomposition does not perform well on convolutional layers 2 and 5. Therefore the other configurations, used in the experiments, does not use decomposition for convolutional layers 2 and 5.

7.2.2 Accuracy

The accuracy for the different network configurations can be seen in Figure 15. The green representing the **top-1 accuracy** (the accuracy for a correctly classified dog breed). The purple is representing the **top-5 accuracy** (the accuracy of the correct dog breed being one of the 5 highest probabilities for the output of the network). The x-axis represents the different network configurations with different decomposed layers with different values for K. K1, K3 and K4, presents the values used for K for the decomposition of convolutional layers 1,3 and 4, respectively. The Previous study constitutes of K1 = 8, K2 = 40, K3 = 60, K4 = 100 and K5 = 200, as can be seen in Table 4 in section **6.1 Different values for K**. The y-axis represents the accuracy in percentage.

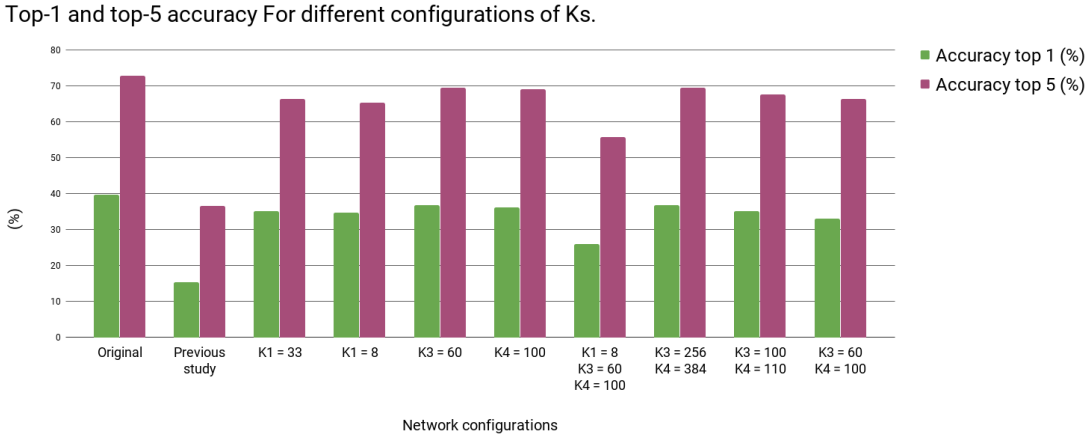


Figure 15: Accuracy for different configurations. No standard deviation is shown since the accuracy remains unchanged for different runs

As can be seen in Figure 15 the accuracy is negatively affected by the decomposition. Using the configuration used in the previous study [4] the accuracy is reduced significantly. Other configurations of K perform better and can achieve comparable accuracy to the original network. However, the decomposed networks do not manage to achieve equal or higher accuracy than the original network.

7.2.3 Time vs accuracy

In Figure 16 the time and accuracy are compared for the all the configurations, as well as the original network.
The x-axis represents the different network configurations, the left y-axis the average sum of the computational time for the different convolutional operations in milliseconds and the right y-axis represents the accuracy of the models in %. The green and purple bars in the graph represents the top-1 accuracy and the top-5 accuracy, respectively, and the blue line is the computational time in milliseconds.

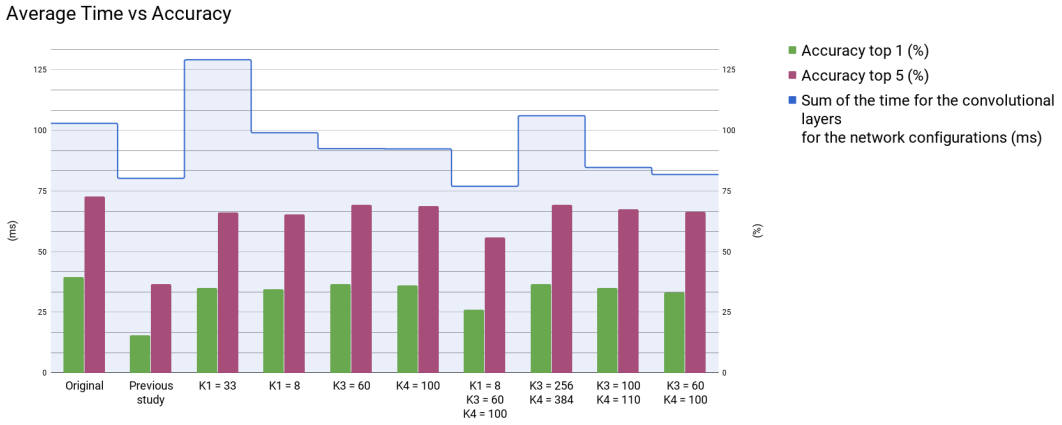


Figure 16: Time against accuracy, accuracy being presented in percentage and time in milliseconds. No standard deviation is shown in the figure since it was at most, for all experiments, 0.343ms $\approx$ 1% of the computation time.

As can be seen in Figure 16 the sum of the time to compute the convolution layers can be significantly reduced compared to the original network while still achieving comparable performance in terms of accuracy. Having multiple decomposed layers can reduce the time even further, however the accuracy is also reduced. The network configurations of interest, that achieves a speed-up with the least drop in accuracy, are:

- K3 = 60
- K4 = 100
- K3 = 100, K4 = 110
- K3 = 60, K4 = 100

7.3 Best network configurations

The four best network configurations in regards to the least drop in accuracy, compared to the original network, and the speed-up gained for convolutional layers are presented in Figure 17. The green and purple bars represents the difference in the top-1 accuracy and top-5 accuracy compared to the original network, respectively. The yellow line represents the total speed-up for all the convolutional layers, for the different configurations obtained compared to the original network. The x-axis represents the different configurations, and same as before, the different Ks represents the value for the low-rank decomposition used for that convolutional layers. K3 and K4 signifying the values used for the decomposition for convolutional layer 3 and 4, respectively. Left out values for Ks signifies that no decomposition was made for those convolutional layers. The left y-axis represents the drop in accuracy and the right y-axis represents the speed-up in percentage for the decomposed networks compared to the original network.

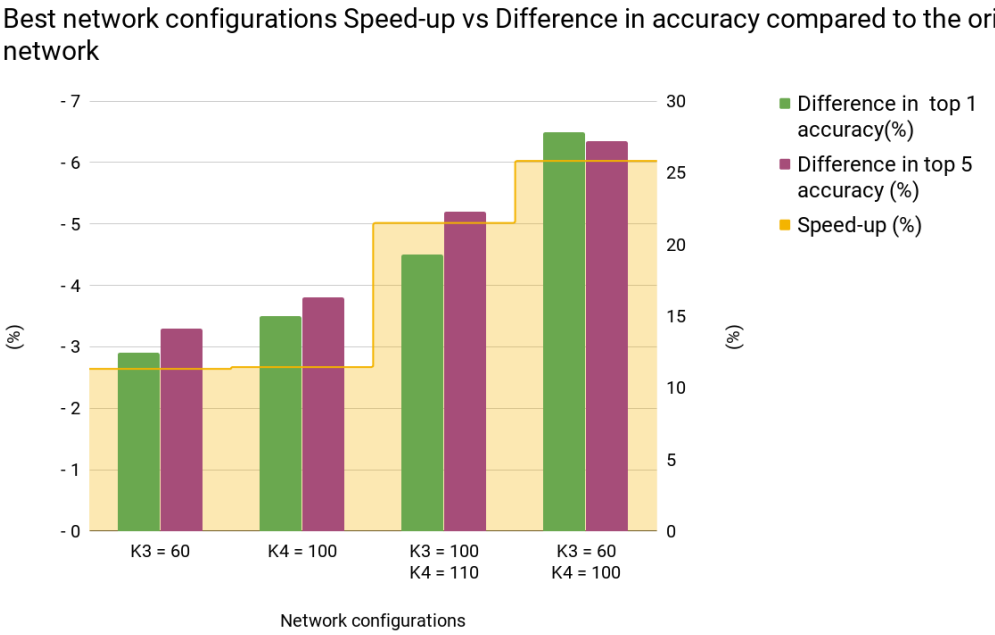


Figure 17: Speed-up compared to the difference in accuracy for the four best network configurations compared to the original network. No standard deviation is shown in the figure since it was at most, for all experiments, 0.343ms $\approx$ 1% of the computation time.

In Figure 17 it can be seen that using the network configuration with K3= 60 and K4=100 the greatest gain in speed-up can be achieved, however it also has the largest drop in accuracy for both the top-1 and top-5 accuracy. The network configuration using only K3 = 60 has the least drop in accuracy, but also the least gain in speed-up.

Lastly, the speed-up for the convolutional layers of the four best network configurations are compared to the drop in accuracy relative to accuracy achieved with the original network. The purpose of which is to try to determine the best compromise between maintaining the accuracy while increasing the speed-up. The results of which is shown in Figure 18. The green and the purple bars represents the ratio for the speed-up and the relative drop in top-1 and top-5 accuracy, respectively. The x-axis represents the network configurations and the y-axis represents the ratio. A high ratio represents a high speed-up and a small relative drop in accuracy.

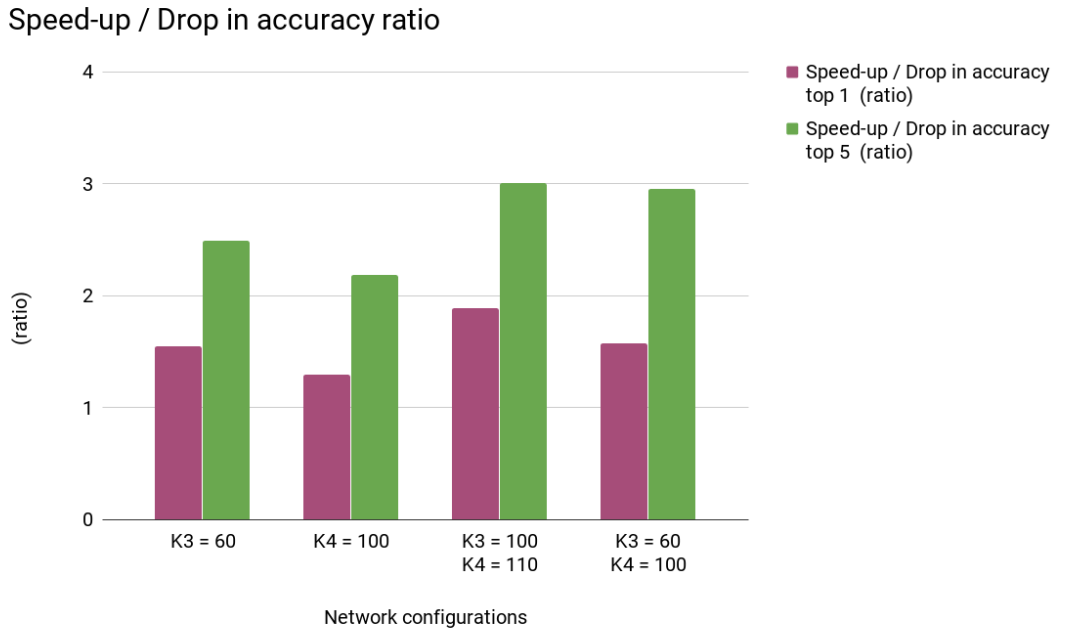


Figure 18: Speed-up against the accuracy drop relative to the accuracy achieved with the original network. The results are represented in percentage. No standard deviation is shown in the figure since it was at most, for all experiments, 0.343ms $\approx$ 1% of the computation time.

In Figure 18 it can be seen that using network configuration with K3 = 100 and K4 = 110 has the highest ratio between speed-up gain and top-1 and top-5 accuracy loss.

8 Discussion and conclusion

In this section, a summary of what has been done to answer the research question is held, as well as a discussion regarding the implementation, future work and conclusion of the study.

8.1 Summary

This study had the following research question:

- How is the effectiveness of the decomposition affected by having a limited training set?

In order to answer the research question, the following questions were investigated:

- How does the decomposition affect:
    1. inference speed?
    2. accuracy of the model?

Previous results shown in Ref. [4] suggest that decomposition can be used to greatly increase the inference speed with no significant drop in accuracy. The results obtained in this study suggests that the decomposition is affected by the limited training set. The decomposed network could not achieve the same accuracy as the original network using the same Ks that were used in the previous study. The training set was not enough for the original network to learn representative features by itself. Instead, transfer learning was required with all convolution weights being frozen and only the two last fully connected layers being trained in order to achieve good results. After the decomposition, additional training was required. Since the training set was not enough for the original network to learn representative features by itself, the limited training set could have affected the possibility to restore the original accuracy with the Ks proposed in the original paper.

As stated in **1.1 Motivation**, it is important to reduce the size of the feed forward layers in order to reduce the amount of parameters, however, it can be seen in Figure 11 that the feed forward layers are not the bottleneck in regards to increasing the inference speed. Interestingly, the decomposition for convolutional layers 2 and 5 does not perform well, and affects the computational time negatively for various Ks, as can be seen in Figures 12 and 13.
The reason for negative impact of the decomposition for convolutional layers 2 and 5 could be that the overhead of the decomposition is too large compared to the actual computational time for the respective undecomposed layers. As can be seen in Figure 11, both convolutional layer 2 and 5 are the two least time-consuming convolutional layers. However, convolutional layer 3 has a similar computational time as convolutional layer 2, even though the number of maps is significantly larger for convolutional layer 3 compared to convolutional layer 2 and 5, as can be seen in Table 1. Therefore the positive effect of the decomposition could be greater for convolutional layer 3.
By examining the results of Figures 12 and 13 it can be seen that the computational time for decomposed convolutional layers 2 and 5 are getting closer to the original computational time for the respective layer. This suggests that an improvement could be achieved using an even larger batch size. Other values of K2 and K5 could also potentially improve the results, however, due to the bad performance of these two convolutional layers the other layers were targeted for decomposition. Using K1= 33 had a negative impact on the performance in terms of time, the reason for this could be that the overhead for utilizing two convolutional operations instead of one ,when K1= 33, becomes too great. The output from the first convolutional operation becoming to large and, in combination with the overhead, resulting in a worse performance.

The results clearly show that there is a compromise required between the inference speed and the accuracy. The results of the four best networks, as can be seen in Figure 17, show that the drop in top-1 and top-5 accuracy is between $\approx 3\%$ and $\approx 7\%$, while the increase in inference speed is between $\approx 11\%$ and $\approx 25\%$.

The best ratio between drop in accuracy and increase in inference speed can be seen in Figure 18, and is achieved by the following configuration:

- K3 = 100, K4 = 110

However, depending on what to prioritize, other network configurations could be better suited. If for instance the speed of the network is important while the accuracy is not, then:

- K3 = 60, K4 = 100

could be a suitable option. If maintaining the accuracy achieved by the original model is important, then:

- K3 = 60

could be the best option.

Additionally, the decompositions is not affected by a decomposition of another layer in regards to the time for the operation to finish, as can be seen in Figure 14. Having multiple decompositions does affect the accuracy of the network, as can be seen in Figure 15.

The local response normalization operation is also a significant bottleneck, which is unaffected by the decompositions, as can be seen in Figure 11.

8.1.1 Answer to the research question

Using low-rank tensor decomposition, an increase in inference speed can be achieved. The low-rank decomposition has a negative impact on the accuracy. Due to the limited training set the accuracy of the original model can not be achieved by the decomposed network. However, by carefully selecting which layers to decompose and which K to apply, comparable results can be obtained in regards to accuracy, while also increasing the inference speed. Therefore the effectiveness of the decomposition is affected by the limited training. The limited training set constraints which layers are decomposable and what values for K can be used in order to achieve comparable results in regards to accuracy and achieve an increase in inference speed.

8.2 Future work

Future work would include comparing results with the previous study Ref. [4] using the same dataset to strengthen the results of this study. Additionally, implementing and evaluating other methods of increasing inference speed, described in section **3 Previous work**. The accuracy of each network configuration could potentially also be improved by fine-tuning the training process.

8.3 Conclusion

The results obtained from this study compared to a previous study indicate that there is a strong relationship between the effects of tensor decomposition and the amount of training data available. A significant speed-up can be obtained in the different convolutional layers using tensor decomposition. However, since there is a need to retrain the network after the decomposition and due to the limited dataset there is a slight drop in accuracy.

9 References

[1] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. An analysis of deep neural network models for practical applications. *arXiv preprint arXiv:1605.07678*, 2016.

[2] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. *arXiv preprint arXiv:1707.01083*, 2017.

[3] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143, 2015.

[4] Cheng Tai, Tong Xiao, Yi Zhang, Xiaogang Wang, et al. Convolutional neural networks with low-rank regularization. *arXiv preprint arXiv:1511.06067*, 2015.

[5] Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. Compression of deep convolutional neural networks for fast and low power mobile applications. *arXiv preprint arXiv:1511.06530*, 2015.

[6] Jiongxin Liu, Angjoo Kanazawa, David Jacobs, and Peter Belhumeur. Dog breed classification using part localization. In *European Conference on Computer Vision*, pages 172–185. Springer, 2012.

[7] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[8] Arthur L Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 3(3):210–229, 1959.

[9] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[10] Chris Bishop, Christopher M Bishop, et al. *Neural networks for pattern recognition*. Oxford university press, 1995.

[11] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[12] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[13] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[14] Yani Ioannou, Duncan Robertson, Roberto Cipolla, Antonio Criminisi, et al. Deep roots: Improving cnn efficiency with hierarchical filter groups. 2017.

[15] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[16] Aurélien Géron. *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems*. " O'Reilly Media, Inc.", 2017.

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

[18] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[19] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[20] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient transfer learning. *arXiv preprint arXiv:1611.06440*, 2016.

[21] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In *Advances in neural information processing systems*, pages 3320–3328, 2014.

[22] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[23] Ren Wu, Shengen Yan, Yi Shan, Qingqing Dang, and Gang Sun. Deep image: Scaling up image recognition. *arXiv preprint arXiv:1501.02876*, 7(8), 2015.

[24] Tara N Sainath, Brian Kingsbury, Vikas Sindhwani, Ebru Arisoy, and Bhuvana Ramabhadran. Low-rank matrix factorization for deep neural network training with high-dimensional output targets. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 6655–6659. IEEE, 2013.

[25] Wenlin Chen, James Wilson, Stephen Tyree, Kilian Weinberger, and Yixin Chen. Compressing neural networks with the hashing trick. In *International Conference on Machine Learning*, pages 2285–2294, 2015.

[26] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. In *Advances in Neural Information Processing Systems*, pages 2074–2082, 2016.

[27] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[28] Mamta Sharma. Compression using huffman coding. *IJCSNS International Journal of Computer Science and Network Security*, 10(5):133–141, 2010.

[29] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: efficient inference engine on compressed deep neural network. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 243–254. IEEE, 2016.

[30] Azure N-Series preview availability . `https://azure.microsoft.com/en-us/blog/azure-n-series-preview-availability/`. Accessed: 2018-09-05.

[31] Kaggle dog breed identification. `https://www.kaggle.com/c/dog-breed-identification`. Accessed: 2018-03-01.

[32] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[33] TensorFlow performance guide. `https://www.tensorflow.org/performance/performance_guide`, . Accessed: 2018-05-23.

[34] TensorFlow programmers guide. `https://www.tensorflow.org/programmers_guide/datasets`, . Accessed: 2018-06-04.

[35] AlexNet pretrained weights. `http://www.cs.toronto.edu/~guerzhoy/tf_alexnet/`. Accessed: 2018-03-23.

[36] Dmytro Mishkin and Jiri Matas. All you need is a good init. *arXiv preprint arXiv:1511.06422*, 2015.