Establishing Effective Techniques for Increasing Deep Neural Networks Inference Speed

ALBIN SUNESSON
Establishing Effective Techniques for Increasing Deep Neural Network Inference Speed

Albin Sunesson
albinsu@kth.se

Master’s Thesis in Machine Learning (30 ECTS credits)
at the School of Computer Science and Engineering
Royal Institute of Technology year 2017
Supervisor at CSC was Arvind Kumar
Examiner was Örjan Ekeberg

Royal Institute of Technology
School of Computer Science and Communication

KTH CSC
SE-100 44 Stockholm, Sweden

URL: www.kth.se/csc
Abstract

Recent trend in deep learning research is to build ever more deep networks (i.e. increase the number of layers) to solve real world classification/optimization problems. This introduces challenges for applications with a latency dependence. The problem arises from the amount of computations that needs to be performed for each evaluation. This is addressed by reducing inference speed. In this study we analyze two different methods for speeding up the evaluation of deep neural networks.

The first method reduces the number of weights in a convolutional layer by decomposing its convolutional kernel. The second method lets samples exit a network through early exit branches when classifications are certain. Both methods were evaluated on several network architectures with consistent results.

Convolutional kernel decomposition shows 20-70% speed up with no more than 1% loss in classification accuracy in setups evaluated. Early exit branches show up to 300% speed up with no loss in classification accuracy when evaluated on CPUs.

Sammanfattning

Etablering av effektiva tekniker för att öka inferenshastigheten i djupa neurala nätverk.


Dekomposition på fältningskärnan visar 20-70% hastighetsökning med mindre än 1% försämring av klassifikationssäkerhet i evaluerade konfigurationer. Tidiga förgreningar visar upp till 300% hastighetsökning utan någon försämring av klassifikationssäkerhet när de evalueras på CPU.
CONTENTS

1 Introduction ................................................................................................................................. 6
  1.1 Reducing inference speed ........................................................................................................ 6

2 Theoretical Background .............................................................................................................. 8
  2.1 Neural Networks ..................................................................................................................... 8
  2.2 Forward Pass .......................................................................................................................... 9
  2.3 Activation functions ............................................................................................................... 9
  2.4 Fully Connected layers ......................................................................................................... 10
  2.5 Convolutional layers ............................................................................................................ 10
  2.6 Pooling Layer ....................................................................................................................... 11
  2.7 Back propagation .................................................................................................................. 12
  2.8 Supervised learning ............................................................................................................ 12
  2.9 Deep neural networks ........................................................................................................... 12

3 Related work - Speed-up techniques ....................................................................................... 14
  3.1 Convolutional layer - Tensor Decomposition ....................................................................... 14
     3.1.1 Rank selection ................................................................................................................ 15
  3.2 BranchyNet .......................................................................................................................... 15
  3.3 Fully connected layer – matrix compression ......................................................................... 16
  3.4 Lower numeric precision ..................................................................................................... 16

4 Method ....................................................................................................................................... 17
  4.1 Decomposition ...................................................................................................................... 17
     4.1.1 Scheme: .......................................................................................................................... 17
     4.1.2 Rank selection ................................................................................................................ 19
     4.1.3 Network structures – CIFAR-10 Image Recognition ..................................................... 20
     4.1.4 Network structure - Geolocation .................................................................................. 21
  4.2 Branches ............................................................................................................................... 22
     4.2.1 Network structure - LeNet ............................................................................................ 24
     4.2.2 Network structure - AlexNet ......................................................................................... 25
  4.3 Technology ............................................................................................................................. 25

5 Results ......................................................................................................................................... 27
  5.1 Decomposition ...................................................................................................................... 27
     5.1.1 K as a hyperparameter ................................................................................................. 27
5.2 Fine Tuning........................................................................................................................................... 29
  5.2.1 Trained from random initialization............................................................................................... 31
  5.2.2 Scheme........................................................................................................................................ 32
  5.2.3 Rank effect ................................................................................................................................... 37
5.3 Branches.............................................................................................................................................. 39
6 Discussion............................................................................................................................................... 40
  6.1 Decomposition................................................................................................................................. 40
  6.2 Branches........................................................................................................................................... 42
7 Conclusion................................................................................................................................................ 43
8 Ethics and Sustainability ....................................................................................................................... 44
9 Future Work............................................................................................................................................ 45
10 References............................................................................................................................................. 46
# Introduction

In the past decade, deep learning architectures have leapfrogged past other approaches to provide state-of-the-art performance on a great variety of machine learning applications. This includes image recognition (Krizhevsky, 2012); (Simonyan, 2014), speech recognition (Abdel-Hamid, 2013) (Abdel-Hamid, 2012), biomedical informatics (Di Lena, 2012), and many other fields, ameliorating results from techniques with a more heuristic entry point (Abdel-Hamid, 2012).

Though methods related to what we now think of as “deep learning” have roots in the early 90s, only recently has it been possible to apply them successfully across such varied domains. The primary reason is that required advancements in network design such as neuron activation (Maas, 2013), normalization (Srivastava, 2014), optimization (Duchi, 2011), and more, have only been developed in the past five years. An important contributor to the acceleration of research in deep learning has been the adoption of GPUs for computation, which has allowed researchers to quickly iterate and develop enormously complex networks - resNet (He, 2016) and googleNet (Szegedy, 2015) being two prominent examples.

In the race to more complex networks, computing power is never enough and a significant amount of research has been done on reducing the computational load required for training. This includes techniques such as weight pruning (Han, 2015), soft weight sharing (Ullrich, 2017), and different hashing techniques (Chen, 2015); (Han, 2015). Though the research has been successful in reducing the model size and time spent on training weights, it has not significantly reduced the inference speed of the resulting model.

When we talk about inference speed in this report it refers to the time it takes for a sample to be propagated through a network that is trained, which is to say, its weights are fixed.

Whereas the inference is computationally less expensive in comparison with training it is still considerable for a deep neural network that contains many connections. Real time evaluation is troublesome for applications that are latency dependent, the inference speed is a bottleneck. Other applications such as video streaming require high throughput which could also be impossible to fulfill if inference is too computationally taxing. It seems that techniques for reducing inference time have a lot of potential applications.

## 1.1 Reducing inference speed

This report will be investigating techniques that aim to speed up the evaluation step of deep neural networks while still retaining a sufficient accuracy. No considerations will be taken to the training time of the network. Evaluation of a neural network is done with a forward pass through the trained network with fixed weights. Inference speed up is reached by reducing computations in the forward pass of the neural network.
One speed up technique investigated is a compression scheme that reduces the network size by decomposing one of its most computational parts, the convolutional layer. The compression scheme is developed by Tai et al (Tai, 2015). We further extend this method by introducing an intermediate PCA-analysis that eliminates a hyperparameter inferred in the decomposition.

We also investigate the possibility to infer early exit branches to networks to increase the evaluation speed up.
2 THEORETICAL BACKGROUND

2.1 NEURAL NETWORKS
An artificial neural network is a computational model used in machine learning. It consists of a set of interconnected nodes (or neurons) inspired by the neurons in the human brain. These are typically arranged in layers that the input of the network is propagated through. Each connection of the network consists of a weight that the input to that connection is multiplied with and each node consist of an activation function that determines what signal should be propagated to the next neuron in the network based on its input.

![Connected nodes with input signals $x_i$, weights $w_i$, and an activation function $f$](image)

The most common architecture of a neural network is a feed forward network. In a feed forward network the input signal is propagated from the first layer(input) to the last layer(output) consecutively through each layer in one direction only. After a signal is propagated through the network it is evaluated according to some predefined evaluation metric and the error is sent back through the network to update the weights using back propagation. This is how the network is able to learn. Evaluating the output can be done in several ways depending on the goal of the network. After a network is trained its weights are fixed and stored for deployment. The model is then used by propagating signals through the network and retrieving the output signals but without updating the weights. When training and evaluating a neural network training data is divided into separate partitions, one that is used for training and updating weights and one that is used to validate the performance. This is done to make sure that the network has not learned very specific features that only satisfies the training examples and will be unable to generalize. (Bishop, 2006)
2.2 **FORWARD PASS**
When evaluating a feed forward neural network, samples are propagated through each layer and are updated in accordance with the weights and connections of that layer, this is referred to as a forward pass. The training phase of a network consists of both a forward pass and a backward pass where all the weights are updated whereas the evaluation of the network consists of only the forward pass. As a sample is propagated consecutively through each layer the forward pass becomes more computationally expensive for every layer that is added to the network. For a deep neural network the amount of computations needed often pose a problem during the training phase where both the forward and backward pass needs to be performed for a large number of samples. However, the forward pass during the evaluation phase will pose a problem in applications that are dependent on a very low latency (Kim, 2015) (Han, 2015). Reducing the computations needed in a forward pass will reduce the inference speed for classifying a sample and reduce problems of latency.

2.3 **ACTIVATION FUNCTIONS**
The activation function is a non-linear function that determines the output of a computational node in the network based on its input or inputs. In a neural network it is desirable to have activation functions that are continuously differentiable. This enables gradient based optimization methods on the network. In this report two different activation functions are used, the tangent hyperbolic function (tanh) and rectified linear units(ReLU) (Nair, 2010). They are given by:

\[
tanh(x) = \frac{2}{1 + e^{-2x}} - 1
\]

\[
ReLU(x) = \begin{cases} 
0 & \text{for } x < 0 \\
 x & \text{for } x \geq 0 
\end{cases}
\]

Where \(x\) is given by the summed inputs to the node.
2.4 **FULLY CONNECTED LAYERS**

Fully connected layers consist of neurons that are connected with the previous layer in an all-to-all manner. That is, all the neurons in the layer have distinct individual connections with all the neurons in the next layer.

This layer is a main building block in most deep-learning networks.

![Fully connected layers in an artificial neural network](image)

*Figure 2 - Fully connected layers in an artificial neural network*

2.5 **CONVOLUTIONAL LAYERS**

A convolutional layer consists of a set of learnable kernels (or filters) that operates on a 3D input volume. The kernels have a fixed size on its receptive field that extends through the full depth of the input volume. During the forward pass, the kernels are convolved over the width and height of the input volume to the layer and computes a dot product between each weight in the filter and the inputs and produces a 2-dimensional activation map of each filter. These are then stacked across the depth dimension to form the output volume of the convolutional layer. With this procedure, every neuron in the output volume becomes activated corresponding to the activity in a spatial area of the input. Convolutional neural networks are effective for detecting local features in the input data.

A convolutional layer exists of a set of hyperparameters:

- The number of kernels – K
- The 2-dimensional size of each kernel – D
- The stride that determines how the kernel is convolved over the input volume – S
- Zero padding added around the input volume – P

Using these hyperparameters the convolutional layer takes an input volume of size \( Z \in \mathbb{R}^{X \times Y \times C} \) and produces an output volume of size \( F \in \mathbb{R}^{X' \times Y' \times N} \) where:

\[
x' = \frac{X - D + 2P}{S} + 1
\]

\[
y' = \frac{Y - D + 2P}{S} + 1
\]

\[N = K\]
Now a slice of size $X' \times Y'$ of the output volume consists of the $n$:th convolution over the input volume with a stride $S$. (LeCun, 1995)

*Figure 3 - Illustration of the spatial properties of a convolutional filter.*

Lecun et al. (LeCun, 1995) introduced convolutional layers for recognizing images of handwritten digits. An early developed convolutional neural network by Lecun et al. was LeNet (LeCun, 1998), scoring a good result on the MNIST data set. Convolutional layers have since been used extensively in more complex network (Krizhevsky, 2012) classifying more complex image data sets and performing other tasks (Goodfellow, 2014).

2.6 **POOLING LAYER**

Convolutional layers are commonly followed by a pooling layer. Pooling layers are a form of non-linear down-sampling methods. All methods have in common that they divide the input of the layer to non-overlapping partitions. The output of each partition is then determined by the specific method. The most common one is max pooling. Max pooling outputs the maximum value of each partition. Other methods include average pooling and L2-norm pooling.
Pooling layers are used to reduce the number of parameters and computations in the network. It is also used to prevent networks from over-fitting. It can for example make networks more robust against translational variance. (Scherer, 2010)

2.7 BACK PROPAGATION
The back propagation algorithm is used to adjust the weights of a network to best fit the mapping f. The intuition behind the back propagation is to calculate the error of the network using a loss function E. Then calculate how much each weight in the network contributes to the loss \( \frac{\partial E}{\partial w} \). Depending on their contribution to the total loss the weights are then updated in accordance with an optimization algorithm. In this report the two optimization algorithms stochastic gradient descent (Bishop, 2006) and RMSProp (Tieleman, 2012) are used. Back propagation is only applied during the training phase of the network.

2.8 SUPERVISED LEARNING
In supervised learning the network is learning by looking at previous examples that are correctly annotated. It is given a set of training data that consists of example pairs with an input \( x \in X \) and a desired output \( y \in Y \). The goal is to find a neural network that gives a mapping \( f : X \rightarrow Y \) between the input and the output inferred from the data. The input data \( x \) is passed through the network and the corresponding output of the network is recorded. The output is then compared with the correct answer and the error is calculated using some defined cost function. The error is then propagated back through the network and all the weights are updated according to the error and the back-propagation algorithm. It this report all experiments conducted will be supervised learning.

In addition to supervised learning there exists other types of learning including unsupervised learning (James, 2013) and reinforcement learning (Sutton, 1998).

2.9 DEEP NEURAL NETWORKS
Neural networks that contain more than one hidden layer are referred to as deep neural networks. Deep nets are based on multiple levels of representations of data (features) that are hierarchically ordered, higher level orders are derived from lower level orders. These networks
have the ability to learn representations on multiple levels. Each representation corresponds to a different level of abstraction. This form a hierarchy of learned concepts. (Deng, 2014)
3 RELATED WORK - SPEED-UP TECHNIQUES

Whereas evaluating pre-trained deep neural networks require less computation than training it still demands a significant number of large matrix multiplications. This introduce problems of latency in a variety of applications. It is a common scenario to have access to a powerful GPU backend for training purposes but deploying models for evaluation on CPU for practical or economic reasons. There is, for example, a growing trend in bringing pre-trained deep neural networks to low power, embedded devices. Other examples that put specific demands on low latency evaluation are real time language translation and object detection in video streams.

Speeding up neural networks has been investigated using a variety of different techniques. Most of them have been oriented towards decreasing the training time of the network. Some of these are in conjunction with speeding up the inference time of the network. There are also a few methods developed solely for the purpose of speeding up the evaluation. Here is an account of some of the most common ones.

3.1 CONVOLUTIONAL LAYER - TENSOR DECOMPOSITION

A low rank decomposition of a convolutional layer makes use of the fact that most convolutional kernels, represented as a 4D-tensor, contains a significant amount of redundancy (Tai, 2015). It has been shown that it is often possible to significantly reduce the size and complexity of the convolutional layers with a negligible small information loss. Low rank matrix approximations are a well-established mathematical minimization problem where the cost function is the fit between a given matrix and an approximating matrix under the constraint that it has a reduced rank. This is successfully used in a variety of applications. Generalized as a tensor decomposition the approximation form a non-convex optimization problem and is in general difficult to compute (Tai, 2015). Learning separable 1D filters has been suggested in the context of dictionary learning (Rigamonti, 2013).

Specific for convolutional layers, and convolutional neural networks, Jaderberg et al. (Jaderberg, 2014) proposes a scheme that minimizes the L2 reconstruction error of the original kernels. Jaderberg et al. uses an iterative algorithm that finds a local approximate solution for each layer. These are then fixed and the layers above are fine-tuned based on a reconstruction criterion. The technique outlined by Jaderberg et al. has been shown to be successful in order to reduce the computations needed for inference of the network. However, the iterative nature of the solution makes the transformation of the network a time-consuming task and would greatly benefit from being simplified. The approximation scheme by Jaderberg et al. can also only be guaranteed to find a locally optimal solution to the optimization problem.

Furthermore, Lebedev et al. (Lebedev, 2014) proposes a technique for canonical polyadic (or tensor rank) decomposition of the kernel tensor that uses a non-linear least squares computation. It is evaluated both on a character-classification convolutional neural network (CNN) and AlexNet showing a significant speed-up with a small loss of accuracy for smaller
networks. However, they found problems of exploding gradients for deeper networks such as AlexNet and never reached any satisfying results for larger network structures.

Zhou et al. (Zhou, 2015) proposes a variety of formulations for reshaping the tensor kernel of a convolutional layer into a matrix. They then approximate the resulting matrix as a Kronecker Product of matrices that compresses the number of parameters. Different formulations for reshaping the tensor are based on different assumptions of how the data is organized. This requires knowledge or an intuition about the problem that may not always be available.

This report is investigating the approximation scheme supposed by Tai. C et al. (Tai, 2015).

3.1.1 Rank selection
Using tensor decomposition as proposed by Tai. C et al. introduces the rank of the decomposition as a new hyperparameter. This gives an extra complexity to the training of the network as one also need to find the proper rank of our approximation through a grid search or some other technique. Yong-Deok Kim et al. (Kim, 2015) introduced a compression scheme they called one-shot whole network compression that uses a Tucker-decomposition on the kernel tensor to reduce its complexity. The ranks in the Tucker decomposition are selected using a global analytic solution to a variational bayes matrix factorization (VBMF) (Nakajima, 2010) (Nakajima, 2013).

3.2 BranchyNet
Early exit branches for deep neural networks were proposed by Teerapittayanon et al. (Teerapittayanon, 2016). It utilizes the fact that most samples can be accurately classified at an early layer in a deep neural network. To account for this, the network is given branches where these samples can exit the network, thus avoiding unnecessary computations throughout the rest of the layers. To be able to decide whether a sample should be classified at an early exit branch entropy is defined over the softmax layer to define its confidence:

\[
Entropy(\bar{y}) = -\sum_{c \in C} y_c \log(y_c)
\]

where \( \bar{y} \) is a vector that contains all the computed probabilities and \( C \) is the set of all possible labels. BranchyNet was further developed by Teerapittayanon et al. to compute fast inferences on a deep neural network over a cloud with end devices (Teerapittayanon, 2017). Since the entropy value that determines whether a sample is classified or should be propagated further through the network needs to be calculated for each sample individually BranchyNet is best suited for real time applications that does not batch samples during inference.
3.3 **FULLY CONNECTED LAYER—MATRIX COMPRESSION**

Fully connected layers are a common component in deep neural networks. Compressing the size of fully connected layers has been investigated both in terms of memory consumption and inference speed (Novikov, 2015) (Cox, 2016) (Zhou, 2015) (Xue, 2013). Zhou et al. (Zhou, 2015) proposes an approximation of the weight matrix as a Kronecker product of two matrices of lower rank. Their approximation can also consist of a linear combination of several Kronecker products for a weight matrix that is not rectangular. Novikov et al. (Novikov, 2015) proposes a scheme that compresses weight matrices by treating it as a tensor and applying a Tensor Train decomposition (Oseledets, 2011) algorithm on it. In the context of automatic speech recognition Xue et al. have proposed to use singular value decomposition with a reduced rank on the weight matrix on fully connected layers (Xue, 2013).

3.4 **LOWER NUMERIC PRECISION**

Lowering arithmetic precision has been investigated to reduce the computational cost for each operation in a neural network (Gupta, 2015); (Courbariaux, 2014); (Courbariaux, 2015); (Vanhoucke, 2011). Reducing the precision with which weights are stored and operations performed also reduce memory consumption and make it possible to develop specialized hardware and utilize GPU computations more efficiently. Courbariaux et al. and Jonghong et al. (Kim, 2014) have shown that for certain tasks it is possible to train networks with binary computations constraining weights and computations to +1 or -1 (Courbariaux, 2015). Lowering the numerical precision is orthogonal to and can be combined with most other techniques for speeding up deep neural networks. However, this will not be investigated in this report.
4 Method

Neural networks are evaluated using a forward pass. Problems of latency are due to the number of computations that needs to be performed for each such forward pass. In this report two methods are tested to reduce the number of computations. The first method decreases the computations in the kernel of a convolutional layer. The second method decreases the computations by letting certain sample “escape” unnecessary layers via early branches. It should be noted that the second method specifically target the forward pass in the evaluation phase of the network.

4.1 Decomposition

The low rank regularization on the convolutional kernel was made in accordance with the compression proposed by Tai et al. (Tai, 2015). It has an analytical and data independent solution that makes it fast and effective in deployment. The compression scheme and regularization algorithm are described below.

4.1.1 Scheme:

The convolutional kernel can be described as a 4D tensor $W \in \mathbb{R}^{N \times d \times d \times C}$ where $N$ and $C$ are the number of output and input channels respectively and $d$ is the kernel size. Let the input feature map be given as $Z \in \mathbb{R}^{X \times Y \times C}$ which gives the output feature map as:

$$
    F_n(x, y) = \sum_{i=1}^{C} \sum_{x'=1}^{X} \sum_{y'=1}^{Y} Z^c(x', y')W^c_n(x - x', y - y')
$$

Now the approximation scheme used looks like:

$$
    \tilde{W}_n^c = \sum_{k=1}^{K} H_k^c (V_k^c)^T
$$

where $K$ is a hyperparameter controlling the rank of the approximation. $H \in \mathbb{R}^{N \times 1 \times d \times K}$ form a horizontal filter and $V \in \mathbb{R}^{K \times 1 \times d \times 1 \times C}$ forms a vertical filter. With this the convolution becomes:

$$
    \tilde{W}_n * Z = \sum_{c=1}^{C} \sum_{k=1}^{K} H_k^c (V_k^c)^T * Z^c = \sum_{c=1}^{C} \left( \sum_{k=1}^{K} V_k^c * Z^c \right)
$$

Here the function we want to minimize is over the Frobenius norm as:

$$
    E_1(H, V) := \sum_{n,c} \|W^c_n - \sum_{k=1}^{K} H_k^n (V_k^c)^T\|^2_F \quad (1)
$$

This is a minimization problem that has a closed form solution.
Figure 5 – The low rank regularization divides the original filters to a horizontal part and a vertical part. The parameterization is illustrated above. To the left is the original filter and to the right the low rank constrained filter.

4.1.1.1 Algorithm
Define a bijection that maps a tensor to a matrix as:

\[ \mathcal{T} : \mathbb{R}^{C \times d \times d \times N} \rightarrow \mathbb{R}^{Cd \times dN} \]

by letting the tensor element \((i_1, i_2, i_3, i_4)\) map to the matrix element \((j_1, j_2)\) as

\[

t_1 = (i_1 - 1)d + i_2
\]

\[

t_2 = (i_4 - 1)d + i_3
\]

Then define \(\mathcal{W} := \mathcal{T}[W]\), where \(W\) is the convolutional kernel. Take the singular value decomposition as \(\mathcal{W} = UDQ^{T}\). Now we set:

\[
V^c_k(j) = U_{(c-1)d+j,k}D_{k,k}^{1/2}
\]

\[
H^k_n(j) = Q_{(n-1)d+j,k}D_{k,k}^{1/2}
\]

Which is a non-unique solution to the minimization problem.

4.1.1.2 Proof
Look at the minimization problem:

\[
E_2(\bar{W}) := \|\bar{W} - W\|_F^2
\]

Where Rank\((\bar{W})\) ≤ \(K\). Set \((H^*, V^*)\) as a solution to (1). Then construct a solution to (3) as:

\[
\bar{W} = \sum_{k=1}^{K} \begin{bmatrix} V^1_k^T \\ V^2_k^T \\ \vdots \\ V^C_k^T \end{bmatrix} \begin{bmatrix} H^1_k & H^2_k & \cdots & H^N_k \end{bmatrix}
\]
Then we know (because the Frobenius norm is separable):

\[ E_1(H^*,V^*) = E_2(\tilde{W}) \]

We know that \( \text{Rank}(\tilde{W}) \leq K \), so furthermore:

\[ E_2(W^*) \leq E_1(H^*,V^*) = E_2(\tilde{W}) \]  

(2)

Here \( W^* \) is any solution to 2. Then we construct a solution \((\hat{H},\hat{V})\) to (1) with (2), which gives:

\[ E_1(H^*,V^*) \leq E_1(\hat{H},\hat{V}) \]

With (4) this gives:

\[ E_1(\hat{H},\hat{V}) = E_2(W^*) = E_1(H^*,V^*) \]

Which proves that \((\hat{H},\hat{V})\) is a solution to (1).

4.1.1.3 Computational reduction

The computational cost of a regular convolution layer is \( O(d^2NCXY) \). Whereas the computational cost for the vertical and horizontal filter is \( O(dKCY) \) and \( O(dNKXY) \) respectively. This gives a total computational cost of \( O(dK(N + C)XY) \). Now an acceleration is achieved if we choose \( K < \frac{dNC}{N + C} \). This is a theoretical limit for what speed up can be achieved and will be hard to reach in practice due to computational overhead and other restraining factors.

4.1.2 Rank selection

Following the algorithm above infers a new hyperparameter to the convolutional network that needs to be determined, the rank \( K \) of the decomposition. Selecting the rank \( K \) will be a consideration between two different properties of the network, speed and accuracy. Lowering the value will reduce the number of computation and speed up the network. Increasing the value will increase the accuracy of the reconstruction of the original network but decrease speed and computational gain. In this report rank-selection will be done and analyzed in two different ways. Both experimentally regarding \( K \) as a hyperparameter in the training phase and using a more heuristic approach that models the loss of information from doing a low rank assumption.

The information loss is modeled using a PCA analysis on the outputs from a pre-trained convolutional layer. First look at a trained convolutional network and sample the output from a specific layer. Then the covariance matrix from the output is calculated. After that the eigenvalues of the covariance matrix is calculated. Now it is possible to calculate the fraction of the total energy that each eigenvalue is contributing with. Sorted by order in a vector \([eig_1 \ eig_2 \ ... \ eig_N]\) it is possible to see the amount that the sum of the largest \( n \) eigenvalues are contributing with. This is used as an indication of which rank is needed to get a sufficient approximation of the original filters.

Calculating the PCA accumulative energy in this way was done as an intermediate step before decomposing a pretrained network.
All the networks were tested using three different training processes.

1. Train the original network -> infer the low rank regularization on specific layers -> fine tune the network parameters. Here K is treated as a hyperparameter and different ranks are evaluated.
2. Train the original network -> choose decomposition rank by looking at the PCA accumulative energy -> infer the low rank regularization on specific layers -> fine tune the network parameters.
3. Initialize the network with the low rank structure and train the whole network from start. Here K is treated as a hyperparameter and different ranks are evaluated.

4.1.3 Network structures – CIFAR-10 Image Recognition

The proposed compression scheme was tested for image recognition on the CIFAR-10 data set (Krizhevsky, 2009) on two different network structures, referred to as Network1 and Network2. These are slightly modified versions of the NiN network (Lin, 2013) and the AlexNet network (Krizhevsky, 2012).

The first network to be analyzed consisted of 6 convolution layers organized like this:

<table>
<thead>
<tr>
<th>Layer</th>
<th>Kernel size</th>
<th>Number of filters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convolution 1</td>
<td>5x5</td>
<td>192</td>
</tr>
<tr>
<td>Convolution 2</td>
<td>1x1</td>
<td>96</td>
</tr>
<tr>
<td>Convolution 3</td>
<td>5x5</td>
<td>192</td>
</tr>
<tr>
<td>Convolution 4</td>
<td>1x1</td>
<td>192</td>
</tr>
<tr>
<td>Convolution 5</td>
<td>3x3</td>
<td>192</td>
</tr>
<tr>
<td>Convolution 6</td>
<td>1x1</td>
<td>192</td>
</tr>
</tbody>
</table>

Each layer is using rectifier linear units as activation functions. Convolution 1, 3 and 5 are followed by a batch normalization. Convolution 6 is followed by an average pooling layer of 7x7. The output is a softmax layer for classification. The network was trained using a stochastic gradient descent as optimizer. The learning rate was originally set to 0.05 with a momentum of 0.9 and a weight decay of 0.0001.

The second network looked like this:
### Network Structure

<table>
<thead>
<tr>
<th>Layer</th>
<th>Kernel size</th>
<th>Number of filters</th>
<th>Hidden units</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convolution 1</td>
<td>11x11</td>
<td>96</td>
<td>-</td>
</tr>
<tr>
<td>Convolution 2</td>
<td>5x5</td>
<td>256</td>
<td>-</td>
</tr>
<tr>
<td>Convolution 3</td>
<td>5x5</td>
<td>256</td>
<td>-</td>
</tr>
<tr>
<td>Convolution 4</td>
<td>3x3</td>
<td>384</td>
<td>-</td>
</tr>
<tr>
<td>Convolution 5</td>
<td>3x3</td>
<td>256</td>
<td>-</td>
</tr>
<tr>
<td>Fully connected 1</td>
<td>-</td>
<td>-</td>
<td>4086</td>
</tr>
<tr>
<td>Fully connected 2</td>
<td>-</td>
<td>-</td>
<td>1024</td>
</tr>
</tbody>
</table>

Each layer is using rectifier linear units as activation functions. Convolution 1 and 3 are followed by a local response normalization and convolution 2 is followed by a batch normalization. Convolution 1, 3 and 5 are followed by a maximum pooling layer of 3x3. Both fully connected layers have a dropout of 0.5. The output is a softmax layer for classification. The network was trained using a stochastic gradient descent as optimizer. The learning rate was originally set to 0.1 with a momentum of 0.9 and a weight decay of 0.0001.

#### 4.1.4 Network structure - Geolocation

The third network evaluated was developed by Alejandro Vera to infer geolocation using signaling data from radio devices combined with other relevant environment data. It consists of two different parts. The first part is a convolutional network which makes use of images of the surrounding environment. The second part is a feed forward fully connected network which takes the convolutional part as input, and augments it with interesting radio telemetry. The network is trained against a GPS-signal in proximity to the radio device, and is treated as a correct position with no noise.

![Network structure of positioning network.](image)

The structure of each part looks like this:
### Convolutional part

<table>
<thead>
<tr>
<th>Layer</th>
<th>Kernel size</th>
<th>Number of filters</th>
<th>Hidden units</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convolution 1</td>
<td>7x7</td>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td>Convolution 2</td>
<td>5x5</td>
<td>50</td>
<td>-</td>
</tr>
<tr>
<td>Fully connected 1</td>
<td>-</td>
<td>-</td>
<td>350</td>
</tr>
<tr>
<td>Fully connected 2</td>
<td>-</td>
<td>-</td>
<td>20</td>
</tr>
</tbody>
</table>

Each layer is using a hyperbolic tangent as activation function. Convolution 1 and 2 are followed by a maximum pooling layer of 2x2.

### Fully connected part

<table>
<thead>
<tr>
<th>Layer</th>
<th>Hidden units</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fully connected 1</td>
<td>512</td>
</tr>
<tr>
<td>Fully connected 2</td>
<td>512</td>
</tr>
<tr>
<td>Fully connected 3</td>
<td>128</td>
</tr>
<tr>
<td>Fully connected 4</td>
<td>32</td>
</tr>
</tbody>
</table>

Each layer is using rectifier linear units as activation functions. The output is a two-dimensional linear regression.

The network is trained using the RMSProp optimizer (Tieleman, 2012). The learning rate was originally set to 0.1 with a momentum of 0.9, a weight decay of 0.0001 and grad clip was set to zero.

### 4.2 Branches

Network structures with early exit branches have a single entry point but is given several exit points arranged as consecutive branches. If at an early branch in the network the classification of a sample has a high confidence the sample exits at that branch and is not propagated any further. Thereby it reduces computations needed for the classification. If the confidence of a correct classification is not good enough it is propagated further in the network. To determine whether a classification is certain, look at the entropy of the output, defined as:

\[
Entropy(\vec{y}) = \sum_{c \in C} y_c \log(y_c)
\]

where \(\vec{y}\) is a vector that contains all the computed probabilities and \(C\) is the set of all possible labels. Values for the entropies that samples should be exited at are given by threshold values \(T^*_n\).
The networks with exit branches are trained with both the main part of the network and the side branches at the same time. For image classification, which is investigated in this report, outputs of the networks are usually softmax layers. So, the objective loss function can be written as:

\[ L_{\text{network}}(\hat{y}, y) = \sum_{n=1}^{N} w_n L(\hat{y}_n, y) \]

where \( N \) is the number of outputs of the network and \( w \) is the weight provided to each output. In this report all branch weights are set to 1. \( L \) is given as:

\[ L(\hat{y}, y) = -\frac{1}{|C|} \sum_{c \in C} y_c \log \hat{y}_c \]

where \( \hat{y} \) is the output of the softmax layer.

Now the training is composed of the two steps, a forward pass and a backward pass. The forward pass is first propagated through the whole network with both the main part and the side branches. The output of all branches is recorded and evaluated. The cost function defined above is calculated and the error is propagated through all branches and the main network with a backward pass.

Evaluation of the network is now done using the entropy measurement defined above and in accordance with the following algorithm:

**Inference:**

```
for n = 1 to N do
    e = entropy(\hat{y}_n)
    if e < T_n then
        return \hat{y}
    end
end
```

Here \( T_n \) is the threshold value associated with each branch.

A sample is propagated to the first exit branch in the network, there the entropy value is calculated and compared with a threshold. If the calculated entropy is lower than a certain threshold the output is returned and no further calculations are done. If the entropy is higher than the threshold the sample is propagated further to the consecutive branch and so forth. If the sample reaches the main exit point it is classified in accordance with the usual procedure. Calculating the entropy for an exit branch adds an extra computational cost. However, if the proportion of samples that is classified at an early branch is large the computations will be reduced.
Early exit branches were tested on two different network architectures, AlexNet (Krizhevsky, 2012) and LeNet (LeCun, 1995).

4.2.1 Network structure - LeNet
LeNet consists of three convolutional layers and two fully connected layers. After the first convolutional layer a branch with one convolutional layer and one fully connected layer is added.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Kernel size</th>
<th>Number of filters</th>
<th>Hidden units</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convolutional Layer</td>
<td>5x5</td>
<td>5</td>
<td>-</td>
</tr>
<tr>
<td>Convolutional Layer</td>
<td>5x5</td>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td>Convolutional Layer</td>
<td>5x5</td>
<td>20</td>
<td>-</td>
</tr>
<tr>
<td>Fully Connected Layer</td>
<td>-</td>
<td>-</td>
<td>500</td>
</tr>
<tr>
<td>Fully Connected Layer</td>
<td>-</td>
<td>-</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Layer</th>
<th>Kernel size</th>
<th>Number of filters</th>
<th>Hidden units</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convolutional Layer</td>
<td>2x2</td>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td>Fully Connected Layer</td>
<td>-</td>
<td>-</td>
<td>10</td>
</tr>
</tbody>
</table>
All activation functions are sigmoids and each convolutional layer is followed by a 2x2 maximum pooling layer. The network was trained using stochastic gradient descent and a learning rate of 0.1.

LeNet with branches was trained on the MNIST data set (LeCun, 1998).

### 4.2.2 Network structure - AlexNet

AlexNet consists of 5 convolutional layers and three fully connected layers. Here two branches are inferred. Both branches consist of one convolutional layer and one fully connected layer and are placed after the second and the third convolutional layers.

#### Main Network

<table>
<thead>
<tr>
<th>Layer</th>
<th>Kernel size</th>
<th>Number of filters</th>
<th>Hidden units</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convolutional Layer 5x5</td>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Convolutional Layer 5x5</td>
<td>64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Convolutional Layer 3x3</td>
<td>96</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Convolutional Layer 3x3</td>
<td>96</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Convolutional Layer 3x3</td>
<td>64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fully Connected Layer</td>
<td>-</td>
<td></td>
<td>256</td>
</tr>
<tr>
<td>Fully Connected Layer</td>
<td>-</td>
<td></td>
<td>128</td>
</tr>
<tr>
<td>Fully Connected Layer</td>
<td>-</td>
<td></td>
<td>10</td>
</tr>
</tbody>
</table>

#### Branches

<table>
<thead>
<tr>
<th>Layer</th>
<th>Kernel size</th>
<th>Number of filters</th>
<th>Hidden units</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convolutional Layer 3x3</td>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fully Connected Layer</td>
<td>-</td>
<td></td>
<td>10</td>
</tr>
</tbody>
</table>

All activation functions were rectified linear units and some convolutions are followed by a 3x3 pooling layer. The network was trained using stochastic gradient descent and a learning rate of 0.1. AlexNet with branches was trained and evaluated on the CIFAR-10 data set.

### 4.3 Technology

All experiments were conducted using the multi-language deep learning framework MXNet (Chen, 2015). MXNet is embedded in the host language and builds computational graphs using declarative symbolic expressions. To communicate with the host language it provides imperative tensor expressions using NDArrays. Furthermore, MXNet is lightweight (can be embedded on devices) and its ability to combine symbolic and imperative computations makes it flexible and
efficient. Benchmarked against other popular frameworks such as Torch7, Caffe, and TensorFlow it shows competitive results (Chen, 2015).

Julia (Bezanson, 2012) was used as host language for the MXNet framework. Julia is a high-level, dynamic, programming language mainly used for numerical computing. It is easy to deploy and use. It has an expressive type system that is especially suitable for scientific computing which requires high performance on numeric calculations.

The training and inference were run on a workstation with four Nvidia GTX 1080 GPUs. MXNet worked well with setup by using CUDA for GPU accelerated tensor operations, CuDNN which provides optimized GPU kernels for common ML operations such as convolutions, and NCCL for operations on multiple GPUs connected by a PCI bus.
5 RESULTS

5.1 DECOMPOSITION

5.1.1 K as a hyperparameter

Network1 was trained on 50000 samples for 45 epochs with a batch size of 500 samples. Evaluated on a test set with 10000 samples it reached an original classification accuracy of 81.5%. The original weights were then stored and the same network was used for every decomposition.

After each decomposition the resulting network was fine-tuned for 10 epochs and then evaluated on the same test set.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Rank</th>
<th>Speedup</th>
<th>Accuracy (Δ)</th>
<th>Weight reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layer1</td>
<td>( K_1 = 5 )</td>
<td>x1.5</td>
<td>-0.13 %</td>
<td>x2.95 x32</td>
</tr>
<tr>
<td>Layer3</td>
<td>( K_2 = 10 )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Layer1</td>
<td>( K_1 = 5 )</td>
<td>x1.5</td>
<td>-0.77 %</td>
<td>x2.95 x21.3</td>
</tr>
<tr>
<td>Layer3</td>
<td>( K_2 = 15 )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Layer1</td>
<td>( K_1 = 10 )</td>
<td>x1.4</td>
<td>+1.35 %</td>
<td>x1.48 x16</td>
</tr>
<tr>
<td>Layer3</td>
<td>( K_2 = 20 )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Layer1</td>
<td>( K_1 = 10 )</td>
<td>x1.4</td>
<td>+0.37 %</td>
<td>x1.48 x10.7</td>
</tr>
<tr>
<td>Layer3</td>
<td>( K_2 = 30 )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1 - Speedup results Network1

The accuracy and speed up are reported as differences with respect to the original network. The net speed up was increased with lower ranks. The best classification accuracy was an increase of +1.35% with rank \( K_1 = 10 \) and \( K_2 = 20 \).

Network2 was trained on 50000 samples for 50 epochs with a batch size of 500 samples. Evaluated on a test set with 10000 samples it reached an original accuracy of 80.6%. After each decomposition the resulting network was fine-tuned for 10 epochs and then evaluated on the same test set.
Table 2 - Speedup results Network2

No combination of ranks resulted in an improvement of the results after the decomposition of Network2. The smallest decrease of accuracy came from the network with the highest ranks and the smallest increase of speed. It had an accuracy decrease of only -0.16%. For two of the tested ranks the accuracy dropped with more than 6%. This is a significant drop that indicates that it was hard for the network to regain the information lost in the decomposition.

Network3 was trained on 3252 samples for 140 epochs with a batch size of 30 samples. It was evaluated on a test set that consisted of 1383 samples. Its original error margins were a median of 84.6 meters and a mean of 163.8 meters. After the decomposition it was fine-tuned for 10 epochs and evaluated on the same test set.

Table 3 - Speedup results Network3

The result from K1 = 5 and K2 = 10 is especially noteworthy, highly improving the results after the decomposition for both the median and mean value of the positioning error.
5.2 Fine Tuning

After the decomposition of the original network all network weights were fine-tuned for 10 epochs. This was done before evaluating the results that are shown in the tables above. Below graphs for three of the decompositions are shown. These were very similar independent of the choice of the hyperparameters $K_i$.

![Network 1 fine-tuning](image)

*Figure 8 - Network 1 fine-tuned after a decomposition using $K_1 = 10$ and $K_2 = 20*

The original network was trained for 45 epochs with an accuracy of 81.5 %. As can be seen in the figure, it was significantly reduced after the two convolutional networks were decomposed. However, after only one epoch of fine-tuning the parameters it has restored almost all of its accuracy. Furthermore, after 10 epochs of fine-tuning it reaches an accuracy of 82.9 %.
Figure 9 - Network2 fine-tuned after decomposition with $K_1 = 10$, $K_2 = 20$ and $K_3 = 2$

Network2 fine-tuned after decomposition with $K_1 = 10$, $K_2 = 20$ and $K_3 = 20$. Trained for 45 epochs it reaches an accuracy of 80.6 %. After the decomposition the accuracy drops down to 12.5 %, barely better than random guesses. This is significantly improved in the first epoch of fine-tuning and the accuracy is almost restored after 5 epochs. After 10 epochs of fine-tuning it get 80.7 % of accuracy.
Network3 fine-tuned after decomposition with $K_1 = 3$ and $K_2 = 10$. Notice that the y-axis shows the mean and median positioning error. The network was originally trained for 140 epochs and reached a median error of 84.6 meter and a mean error of 163.8 meter. After the decomposition this was increased to a median error of 87.6 meter and a mean error of 168.9 meter. Fine-tuning for three epochs lowered the error values to 75.3 meters and 159.6 meters respectively.

5.2.1 Trained from random initialization

The networks were initialized with the structure of the decomposed networks with a random distribution over the weights. This was done to compare the difference between pre-training the network and decomposing it using the algorithm described above with a small fine-tuning and training the network from start.

**Network1**

<table>
<thead>
<tr>
<th>Layer</th>
<th>Rank</th>
<th>Speedup</th>
<th>Accuracy</th>
<th>Weight reduction</th>
</tr>
</thead>
</table>
| Layer1 | $K_1 = 5$  
Layer3 | $K_2 = 10$ | x1.5 | 76.2% | x2.95  
x32 |
| Layer1 | $K_1 = 5$  
Layer3 | $K_2 = 15$ | x1.5 | 80% | x2.95  
x21.3 |
| Layer1 | $K_1 = 10$  
Layer3 | $K_2 = 10$ | x1.4 | 78.6% | x1.48  
x16 |
| Layer1 | $K_1 = 10$  
Layer3 | $K_2 = 30$ | x1.4 | 79.1% | x1.48  
x10.7 |

*Table 4 - Speedup results Network1, trained from random initialization*
Network1 learned with the decomposed architecture and all tested ranks reached a score above 75%. None of them reached an accuracy that were equivalent with the procedure of decomposing a pre-trained network.

**Network2**

<table>
<thead>
<tr>
<th>Layer</th>
<th>Rank</th>
<th>Speedup</th>
<th>Accuracy</th>
<th>Weight reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layer1</td>
<td>$K_1 = 5$</td>
<td>x1.9</td>
<td>~10%</td>
<td>x6.4 x34.9 x64</td>
</tr>
<tr>
<td>Layer2</td>
<td>$K_2 = 10$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Layer3</td>
<td>$K_3 = 10$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Layer1</td>
<td>$K_1 = 10$</td>
<td>x1.7</td>
<td>~10%</td>
<td>x3.2 x23.3 x42.7</td>
</tr>
<tr>
<td>Layer2</td>
<td>$K_2 = 15$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Layer3</td>
<td>$K_3 = 15$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Layer1</td>
<td>$K_1 = 10$</td>
<td>x1.7</td>
<td>~10%</td>
<td>x3.2 x17.5 x32</td>
</tr>
<tr>
<td>Layer2</td>
<td>$K_2 = 20$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Layer3</td>
<td>$K_3 = 20$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Layer1</td>
<td>$K_1 = 20$</td>
<td>x1.5</td>
<td>~10%</td>
<td>x1.6 x11.6 x21.3</td>
</tr>
<tr>
<td>Layer2</td>
<td>$K_2 = 30$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Layer3</td>
<td>$K_3 = 30$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Table 5 - Speedup results Network2, trained from random initialization*

When initialized with the low rank architectures none of the decomposed versions of Network2 managed to learn anything. After 50 epochs of training their accuracy were all steady around 10%, which for CIFAR-10 is the same result as random guesses.

**5.2.2 Scheme**

The accumulative energy is calculated on the response from 1000 randomly sampled training images of the trained networks. The results are averaged over the samples. It should be noticed that they showed a very high consistency; few samples deviated from the averaged results.
The figure above shows the accumulative energy for the first and the third convolutional layers in Network1. Here \( d \) is the number of largest eigenvalues included in the summation. For the first convolutional layer the accumulative energy was above 90% with the first 12 eigenvalues. For the third convolutional layer the accumulative energy of the first 47 eigenvalues was above 90%. Both layers consisted of 192 layers but this indicates that the convolutional layer 1 contains a larger amount of redundancy that can be reduced without losing accuracy.
Figure above show the accumulative energy for the first three convolutional layers in Network2. The accumulative energy raises above 90 % with the first 20, 16 and 10 eigenvalues for layer 1, 2 and 3 consecutively. Note that the first convolutional layer only has 96 filters.

![Graph showing accumulative energy for the first three layers in Network2.](image)

**Figure 13 – PCA accumulated energy for the response of the convolutional layers, d is the number of eigenvalues.**

Figure above show the accumulative energy for the first two convolutional layers in the positioning network. The accumulative energy is above 90 % for the first 3 eigenvalues for layer 1 and for the first 4 eigenvalues for layer 2. Note that the first convolutional layer only has 10 filters.

**Network1**

Evaluating the accumulative energy of the first and third convolutional layer of Network1 it was found that it is above 90% for the 12 and 47 first eigenvalues. The network was then fine-tuned with the following results:

<table>
<thead>
<tr>
<th>Layer</th>
<th>Rank</th>
<th>Speedup</th>
<th>Accuracy (Δ)</th>
<th>Weight reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layer 1</td>
<td>$K_1 = 12$</td>
<td>x1.2</td>
<td>+1.14</td>
<td>x1.12</td>
</tr>
<tr>
<td>Layer 3</td>
<td>$K_3 = 47$</td>
<td></td>
<td></td>
<td>x6.8</td>
</tr>
</tbody>
</table>

The result is not as good as the best values of $K_i$ when it was treated as a hyperparameter. However, it is still improving both the accuracy and the net speed. To get an intuition about how well the filters are preserved the normalized activation for the same sample of the first 100 filters before and after the network has been decomposed is printed. From the figures below it can be seen that the outputs are similar and that the structure and spatial properties are preserved.
Layer 1 convolutional filters

Figure 14 - Normalized activation output from the first 100 filters in the first convolutional layer. To the left the original layer and to the right the decomposed layer with rank 12.

Layer 3 convolutional filters

Figure 15 - Normalized activation output from the first 100 filters in the third convolutional layer. To the left the original layer and to the right the decomposed layer with rank 47.
Evaluating and fine-tuning Network2 with the scheme aforementioned gave the following results.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Rank</th>
<th>Speedup</th>
<th>Accuracy (Δ)</th>
<th>Weight reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layer 1</td>
<td>$K_1 = 20$</td>
<td>x1.7</td>
<td>-1.05%</td>
<td>x1.6</td>
</tr>
<tr>
<td>Layer 2</td>
<td>$K_2 = 16$</td>
<td></td>
<td></td>
<td>x21.8</td>
</tr>
<tr>
<td>Layer 3</td>
<td>$K_3 = 10$</td>
<td></td>
<td></td>
<td>x64</td>
</tr>
</tbody>
</table>

It decreased the accuracy of the network with -1.05 % but gave a net speed up of x1.7. This was a high reduction of weights without dropping down to an accuracy under $-\Delta 6\%$ which was the case for other configurations of Network2.

I print the activation of the 96 filters in the first convolutional layer and 100 of the filters in the second convolutional layer.

**Layer 1 convolutional filters**

![Layer 1 convolutional filters](image)

*Figure 16 - Normalized activation output from the 96 filters in the first convolutional layer. To the left the original layer and to the right the decomposed layer with rank 20.*
Layer 3 convolutional filters

Figure 17 - Normalized activation output from the first 100 filters in the second convolutional layer. To the left the original layer and to the right the decomposed layer with rank 16.

Network3

Evaluating and fine-tuning Network3 with the scheme aforementioned gave the following results.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Rank</th>
<th>Speedup</th>
<th>Accuracy mean (Δ)</th>
<th>Accuracy median (Δ)</th>
<th>Weight reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layer 1</td>
<td>$K_1 = 3$</td>
<td>x1.2</td>
<td>-0.1 m</td>
<td>+5.5 m</td>
<td>x2.12</td>
</tr>
<tr>
<td>Layer 2</td>
<td>$K_3 = 4$</td>
<td></td>
<td></td>
<td></td>
<td>x10.4</td>
</tr>
</tbody>
</table>

This gave an improved accuracy of the median and did not lose any significant accuracy of the mean value. The second convolutional layer had a significantly lower rank than was evaluated with $K_i$ as a hyperparameter. That the accuracy did not decrease with the low rank indicates that the second convolutional layer contained redundancy that could successfully be regularized.

5.2.3 Rank effect

To illustrate the effect of the decomposition a specific filter is highlighted in the second convolutional layer of Network3. This shows the normalized activations for the same sample for three different ranks. Rank 10 is a close approximation to the original filter. Rank 4 is still a good approximation and similar to the original filter. $K=1$, the lowest rank possible, is losing a lot of information and is a poor approximation of the original filter even though it is still possible to see that it is derived from the original filters.
Figure 18 - Activations of a specific filter in the second convolutional layer. The same sample is propagated through the layer with four different ranks on the decomposition. To the left is the original filter followed with rank $K=10$, $K=4$ and $K=1$ consecutively.

The correlation between accuracy and the PCA accumulative energy for the two decomposed convolutional layers of Network1 is plotted. Here one layer is kept intact with the original structure while the other was decomposed. Both layers show a decline in accuracy in relation to the accumulative energy below ~95%.

Figure 19 – Classification accuracy plotted for the decomposition with PCA accumulative energy for two convolutional layers.

Each point in the figure is evaluated empirically with a d number of filters that correlates to the PCA energy. For 100 % PCA energy there is no decomposition. It should be noted that the correlation between decomposing layers simultaneously is not regarded in the figure. Decomposing the first convolutional layer show a smaller accuracy loss compared to decomposing the second convolutional layer.
5.3 BRANCHES

LeNet was trained and evaluated on the MNIST data set with a training set of 60 000 samples and an evaluation set of 10 000 samples. It was trained for 30 epochs before evaluation. AlexNet was trained on the CIFAR-10 data set with 50 000 training samples and 10 000 evaluation samples. It was trained for 45 epochs.

CPU

<table>
<thead>
<tr>
<th>Network</th>
<th>Accuracy</th>
<th>Speedup</th>
<th>Thresholds</th>
<th>Branch exits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lenet</td>
<td>99.1 %</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Lenet - branched</td>
<td>99.1 %</td>
<td>x3.4</td>
<td>0.025</td>
<td>92 %, 8 %</td>
</tr>
<tr>
<td>Alexnet</td>
<td>79.5 %</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Alexnet - branched</td>
<td>79.2 %</td>
<td>x1.3</td>
<td>0.0001, 0.05</td>
<td>63 %, 20 %, 17 %</td>
</tr>
</tbody>
</table>

Table 6 - Result branches, evaluated on CPU

GPU

<table>
<thead>
<tr>
<th>Network</th>
<th>Accuracy</th>
<th>Speedup</th>
<th>Thresholds</th>
<th>Branch exits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lenet</td>
<td>99.0 %</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Lenet - branched</td>
<td>99.2 %</td>
<td>x0.9</td>
<td>0.025</td>
<td>93 %, 7 %</td>
</tr>
<tr>
<td>Alexnet</td>
<td>79.3 %</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Alexnet - branched</td>
<td>79.4 %</td>
<td>x0.7</td>
<td>0.0001, 0.05</td>
<td>62 %, 22 %, 16 %</td>
</tr>
</tbody>
</table>

Table 7 - Result branches, evaluated on GPU

Both LeNet and AlexNet got a classification accuracy that was tantamount with and without branches. In LeNet a vast majority of all the samples exited at the early branch. It is shown that only a small proportion of the samples need to be propagated through the whole network in order to be correctly classified. For AlexNet most of the samples exited through the two branches. It was around twice as many samples that were propagated through the whole network structure for AlexNet than for LeNet. It should be noted that AlexNet was trained on a more complicated data set. The accuracy and proportions between exited samples did not differ significantly between CPU- and GPU-computations, which was expected. However, the speed increase differed between the two computational methods.

Using CPU computations both networks improved the inference speed with added branches. LeNet had a bigger increase of inference speed. A bigger proportion of samples exited early in LeNet, which led to a larger fraction of the total computations cut.

Using GPU computations the networks did not decrease the evaluation time. This was because the structure of the network introduced a logical operation that needed to be performed in the CPU-memory. Copying values back and forth between GPU and CPU memory eliminated the speed-up gain from the branches.
6 Discussion

6.1 Decomposition

The tensor rank decomposition of Cheng et al. was evaluated on image recognition for two different convolutional neural networks. This infers a new hyperparameter, the rank $K$, into the training of the network. The value of $K$ determines two different properties of the decomposed network. If $K$ is low, the compression and the speed up is significant. This is done at the expense of increasing the reconstruction error in the approximation. Therefore, the accuracy will be lowered for a well-trained network. From table 2 it can be seen that this is the case for all ranks evaluated on Network2. However, for Network1 and the positioning network finding a good set of values of the rank leads to improved results. This indicates that the decomposition can be used to regularize networks that contains a high amount of redundancy and suffers from being over fit to training data. From table 1 it can be seen that a decomposition with $K_1 = 10$ and $K_2 = 20$ increases the classification accuracy with 1.35%. This gives a new perspective that needs to be taken into account when setting the value of $K$. There is a trade-off between the amount of compression and speed increase and how big the reconstruction error is. However, for some cases even if the reconstruction error is increased it seem to improve the performance of the network as a consequence of regularization on the over fitted network. When choosing a rank this is another consideration that should be taken into account. A rank that gives a good trade-off between speed and accuracy might be improved by lowering the rank further, increasing both the compression and the accuracy. This is shown both for image classification in Network1 but also in the positioning network that augments data from several sources, where analysis of images is one fraction of the network.

Of the two image classification networks Network1 was more successfully decomposed than Network2. Where Network2 managed to almost retain its accuracy with a significant speed up and weight reduction Network1 also improved its classification accuracy. This could be because Network1 contained a higher amount of redundancy that over fitted the network, making the decomposition work as a regularizer. It could also be because Network2 was decomposed on three different layers simultaneously as compared to two for Network1. In figure 20 it can be seen that the decomposition of the layers in Network1 individually effects the accuracy of the network in a slightly different way. This show that all layers may contribute with different amounts of redundancy to the network. How this affects the network and questions concerning how the layers are effected by simultaneous decompositions have not been investigated in this report. That is something that calls for further research.

After the decomposition all networks had to be fine-tuned to regain a high classification accuracy. This is due to that the weights jump out of the local minima but still remain very close to it, which makes it easy to find. For Network1 and Network2 only one epoch was sufficient to restore most of the accuracy and after ten epochs they had both reached or improved the original accuracy. The positioning network showed a smaller accuracy loss after the
decomposition and required fewer epochs to regain its original accuracy. This is most probably due to the fact that the convolutional layers do not constitute an equally important part in that network.

We tested a straightforward easy to use approach to avoid the tedious and time-consuming intermediate step of finding a good rank of the decomposition. Instead of treating K as a hyperparameter we analyzed the PCA accumulative energy of the output of a convolutional layer. Choosing the number of filters that gave an accumulative energy above 90% shows very promising results. It gave a significant compression on all tested networks. For Network1 and the positioning network it increased the classification accuracy and for Network2 it only gave a small decrease in accuracy. This is in line with the results we found when optimizing for K as a hyperparameter. The results were also surprisingly consistent and reliable. The accumulative energy did not change significantly between different samples. Neither did it change significantly if a network was retrained from scratch with the same architecture. The threshold value of 90% showed to be a good compromise between compression and accuracy but has not been extensively analyzed. Finding an optimal threshold given your preferences and identifying at which level the accuracy is hard to restore by fine-tuning the parameters is open for future research. It should also be noticed that we calculate the PCA accumulative energy of all the layers of a network separately. That is, we do not take into account how the energy of a layer is affected by earlier layers in the network being decomposed. This is something that maybe could have improved the results further.

Looking at the filters printed for the decomposed convolutional layers in figure 14-17 it is clear that they keep their structures after they are decomposed. This is an indication that they still manage to capture most of the information after they are decomposed. Even such properties as directionality are included. Looking at figure 18 it can be seen that the properties of a decomposed filter is better preserved with a higher rank, which is what we would expect. It can also be seen that decreasing the rank gives smooth decrease in reconstruction accuracy, which is a good property that lets you find a suitable trade-off between accuracy and speed-up for the convolutional layer.

Training the network with the decomposed structures from random initialization was shown to be less successful than the decomposition scheme. For Network2 it did not manage to learn anything when trained from start for anyone of the evaluated configurations. Network1 learned and scored a high accuracy when trained from start but did not reach an equally high accuracy as the pre-trained, decomposed and fine-tuned variants. A probable explanation to this is that it is hard for the networks to find local minima’s in the energy loss function when it is trained from start. The decomposition significantly lowers the accuracy of the network but it still keeps the training weights close to a local minimum. Therefore, only a small amount of fine-tuning is needed in order for the network to reach it and get a high accuracy score. However, if the decomposed networks need to find the minimum trained from random initialization it gets harder for them to find and the network will therefore suffer from inability to learn.
6.2 Branches
Using branches for early exits in the network structure works without losing classification accuracy for both networks evaluated in this report. In table 6-7 it can be seen that a significant number of samples exits at an earlier stage in the network. This reduces the amount of computation that is needed for the network to classify a set of images. One should note that there is an extra computational step that is required for calculating the entropy at each output. The branches also add a computational cost if the samples are not exited early. Hence, the networks ability to increase inference speed is dependent on a sufficient number of samples exiting through an early branch. This is a consideration that is very important to take in to account if one decides to implement the technique.

Evaluated on CPUs we saw a significant speed up in the evaluation for both networks. LeNet had a greater speed reduction than AlexNet. This is in accordance with the aforementioned discussion. LeNet had a much higher proportion of samples exiting at an early branch (note that it was trained on MNIST, a simpler data set than CIFAR-10) and was therefore adding less extra computations. It is not certain that a simpler data set will lead to more early exits, that is determined by the entropy and not the accuracy. However, a simpler data set is most likely to lower the entropy as well since that is a measurement of the certainty in the classification. This was what we saw in this case.

In this report we found it hard to improve the inference speed by adding branches to a network running on GPUs. This was due to the added computational overhead from checking the entropy value of each branch. To check whether the entropy value of a certain branch is above the threshold value it needs to be copied to CPU-memory as GPUs don't have the instructions required for making those sorts of comparisons. One solution would be to batch the samples and not copy them between memories individually. This would decrease the time spent copying each sample. Also, there is usually a speed increase to gain from mini-batching samples into the network even in networks that are being used for real-time processing. However, given that each sample needs to be evaluated individually at the branches, batching samples does not solve the overall problem.

It should be noted that although a branched network may increase inference speed for specific cases it does not reduce memory consumption. On the contrary, the added branches require additional memory like any other part of the network.
It was established that both speed and the number of weights could be reduced with no or a small loss of accuracy using the low rank regularization proposed by Tai et al. on convolutional kernels.

We extend the results to a neural network which infers geo-spatial positioning from radio signals and satellite images. This showed a promising result for applications where data is augmented from several sources.

Furthermore, we show that PCA-analysis can be successfully used as an intermediate step for choosing the rank of the decomposition. This is an original contribution to the original algorithm, and it allows the programmer to select the hyperparameter heuristically.

We show that adding branches to the network improves inference speed on some data without losing accuracy. We also determined the infeasibility of applying the technique to GPUs which are typically used to run the types of networks that would benefit most from this technique.
It has become common to discuss the topic of ethics in relation to the fields of AI (artificial intelligence) and machine learning. The grandiose questions of a "general AI" (with human-like abilities of abstract reasoning) and a potential "singularity" (when an AI-system has the ability to improve itself) have been covered both in books, movies and newspapers, as well as within the academic sciences (Moor, 2006). Less visionary scenarios need to be reflected on by scientists and engineers on a more regular basis. How does our technical improvements effect the environment and our chances to reach a sustainable society? How do we handle eventual social transformations that may arise from our work? A common example of this is the risk of a very high level of unemployment when a variety of jobs are being automated in a high pace. This work does not have any direct connection to these questions in the sense that it does not concern any new applications of machine learning techniques. However, one may argue that it has an indirect effect due to its ability to improve a variety of applications.

Regarding sustainability there is an efficiency to gain in applying the investigated methods. This will reduce the environmental effect that our applied algorithms have since less energy is needed. As regards eventual social and societal changes, as a consequence of the rapid development within the field of machine learning, there is a strong case to be made for the probability that the positive societal impact of machine learning will outweigh the negative ones.
9 Future Work

The workstations used for this thesis were extremely powerful and it might not be obvious how they would benefit directly from these optimization techniques. Nevertheless, though the throughput of the workstations in terms of 32bit FLOPs was impressive, a clear bottleneck was the 8GB of RAM on each card. Not only did it restrict the method of training by forcing smaller batch sizes, but certain networks were too big to fit on the GPU and too complex to reasonably train on CPU. An idea which was raised for future research was to see how decomposition could be used to fit bigger and more powerful networks onto a GPU. A simple test would be to take a pretrained version of a large network such as ResNet-200, decompose it until it fits onto the GPU, and then retrain it with a new dataset.
10 References


Teerapittayanon, S., McDanel, B., & Kung, H. T. (2017). Distributed Deep Neural Networks over the Cloud, the Edge and End Devices. *International Conference on Distributed Computing Systems*.


