Comparison of different approaches to reduce the number of parameters in Deep Neural Networks

A Thesis
Presented to
The Faculty of the Department of Natural Sciences
Paris Lodron University of Salzburg

In Partial Fulfillment
of the Requirements for the Degree of
Master of Science

Submitted by
Tamara Maier
01656470

Supervisor: Univ.-Prof. Dr. Christian Borgelt

Course of Study: Computer Science

Salzburg, July, 2019
STATUTORY DECLARATION

I declare herewith that I have written this thesis independently and no other resources and sources other than those specified have been used. All passages which have been taken literally or contextually from the sources indicated have been referenced as such.

_________________________________________  ________________
(date)                                      (signature)
Deep neural networks contain a large number of parameters. For this reason the training and testing of these networks have a high computational cost. [8] Moreover these deep networks also require a lot of memory. [2]

As a result compression and efficiency have become more important in recent research. [1] Therefore many pruning methods have been proposed to obtain compact models with negligible accuracy loss.

In this thesis three different pruning approaches are analyzed.

First singular value decomposition is presented. Then a pruning technique called neuron importance score propagation is introduced. And finally $\varepsilon$-ResNet is analyzed.

To examine whether the techniques achieve reduction while retaining the predictive power experiments on image classification are performed.

All three pruning techniques successfully reduced the examined networks with only a small decrease in accuracy.

However combining two methods leads to even more reduction. The best pruning result is achieved by applying neuron importance score propagation after the network was already pruned using $\varepsilon$-ResNet. In the performed experiments up to 98.56% of the parameters can be reduced while obtaining almost the initial accuracy.

Therefore pruning methods are very effective and can achieve significant compression.

Model compression not only accelerates testing time, but also reduces the required memory which allows the networks to run on hardware limited devices such as smart phones, robots or cars. [1]
## Contents

Abstract .......................... ii
List of figures ........................ v
List of tables ........................ vii

1 Introduction ...................... 1

2 Deep Neural Networks .......... 3
   2.1 Multilayer Perceptrons .......... 3
   2.2 Convolutional Neural Networks ..... 7
   2.3 Residual Networks .............. 8

3 Overview of Parameter Reduction Approaches .......................... 11

4 Singular Value Decomposition .... 14
   4.1 Theory ........................ 14
   4.2 Experiments .................. 15
      4.2.1 Experimental Setup .......... 16
      4.2.2 SVD on ResNet ............. 18
      4.2.3 SVD on LeNet .............. 22
      4.2.4 SVD on a Fully Connected Neural Network .... 25
   4.3 Conclusion ................... 27

5 Neuron Importance Score Propagation .......................... 29
   5.1 Theory ........................ 29
      5.1.1 Problem Definition .......... 30
      5.1.2 Solution .................. 32
   5.2 Implementation ................ 34
      5.2.1 NISP: 1-way tensor to 1-way tensor .... 35
      5.2.2 NISP: 1-way tensor to 3-way tensor .... 36
      5.2.3 NISP: 3-way tensor to 3-way tensor .... 36
   5.3 Experiments ................... 40
      5.3.1 NISP on a Fully Connected Neural Network .... 40
      5.3.2 NISP on LeNet ............. 46
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Architecture of a multilayer perceptron</td>
<td>4</td>
</tr>
<tr>
<td>2.2</td>
<td>Example of an architecture for a convolutional network</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>Building block of ResNet</td>
<td>9</td>
</tr>
<tr>
<td>2.4</td>
<td>Difference in structure of the original ResNet and &quot;pre-activation&quot; ResNet</td>
<td>9</td>
</tr>
<tr>
<td>4.1</td>
<td>Reconstructed deep neural network (DNN) model after SVD</td>
<td>15</td>
</tr>
<tr>
<td>4.2</td>
<td>Random images of the MNIST dataset</td>
<td>16</td>
</tr>
<tr>
<td>4.3</td>
<td>Random images of the Cifar-10 dataset</td>
<td>17</td>
</tr>
<tr>
<td>4.4</td>
<td>Singular values of the weight matrix of the first fully connected layer</td>
<td>22</td>
</tr>
<tr>
<td>4.5</td>
<td>Accuracies after applying SVD for different k’s on the first fully connected layer</td>
<td>23</td>
</tr>
<tr>
<td>4.6</td>
<td>Singular values of the weight matrix of the second fully connected layer</td>
<td>24</td>
</tr>
<tr>
<td>4.7</td>
<td>Accuracies after applying SVD for different k’s on the second fully connected layer</td>
<td>24</td>
</tr>
<tr>
<td>4.8</td>
<td>Singular values of the weight matrix of the first fully connected layer</td>
<td>26</td>
</tr>
<tr>
<td>4.9</td>
<td>Accuracies after applying SVD for different k’s on the first fully connected layer</td>
<td>26</td>
</tr>
<tr>
<td>4.10</td>
<td>Singular values of the weight matrix of the second fully connected layer</td>
<td>27</td>
</tr>
<tr>
<td>4.11</td>
<td>Accuracies after applying SVD for different k’s on the second fully connected layer</td>
<td>27</td>
</tr>
<tr>
<td>5.1</td>
<td>Grayscale heatmap of the importance scores of the input layer reshaped to the size of the image</td>
<td>41</td>
</tr>
<tr>
<td>5.2</td>
<td>Comparison of NISP and randomly pruning on fully connected layer</td>
<td>45</td>
</tr>
<tr>
<td>5.3</td>
<td>Accuracies during fine-tuning after applying NISP with pruning ratio $\frac{1}{2}$ on ResNet using Cifar-10 dataset</td>
<td>48</td>
</tr>
<tr>
<td>5.4</td>
<td>Accuracies during fine-tuning after applying NISP with pruning ratio $\frac{1}{4}$ on ResNet using Cifar-10 dataset</td>
<td>49</td>
</tr>
<tr>
<td>5.5</td>
<td>Accuracies during fine-tuning after applying NISP with pruning ratio $\frac{1}{2}$ on ResNet using MNIST dataset</td>
<td>51</td>
</tr>
<tr>
<td>5.6</td>
<td>Accuracies during fine-tuning after applying NISP with pruning ratio $\frac{1}{4}$ on ResNet using MNIST dataset</td>
<td>52</td>
</tr>
<tr>
<td>6.1</td>
<td>Difference in structure of the original ResNet and &quot;pre-activation&quot; ResNet</td>
<td>55</td>
</tr>
<tr>
<td>6.2</td>
<td>Basic transformation in standard ResNet and $\varepsilon$-ResNet</td>
<td>56</td>
</tr>
</tbody>
</table>
List of Figures

8.1 accuracies after applying SVD for different k’s on the first fully connected layer of fully connected network that was already pruned using NISP . . . . . . . 69
8.2 accuracies after applying SVD for different k’s on the first fully connected layer of LeNet-style network that was already pruned using NISP . . . . . . 70
8.3 accuracies during fine-tuning of the model pruned with $\varepsilon$-ResNet and NISP using a pruning ratio of $\frac{1}{2}$ to classify MNIST . . . . . . . . . . . . . . . . . . 72
8.4 accuracies during fine-tuning of the model pruned with $\varepsilon$-ResNet and NISP using a pruning ratio of $\frac{1}{4}$ to classify MNIST . . . . . . . . . . . . . . . . . . 73
8.5 accuracies during fine-tuning of the model pruned with $\varepsilon$-ResNet and NISP using a pruning ratio of $\frac{1}{2}$ to classify Cifar-10 . . . . . . . . . . . . . . . . . . 74
8.6 accuracies during fine-tuning of the model pruned with $\varepsilon$-ResNet and NISP using a pruning ratio of $\frac{1}{4}$ to classify Cifar-10 . . . . . . . . . . . . . . . . . . 75
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>SVD with different values of k applied on fully connected layer of ResNet</td>
<td>19</td>
</tr>
<tr>
<td>5.1</td>
<td>NISP applied on fully connected network with different pruning ratios</td>
<td>42</td>
</tr>
<tr>
<td>5.2</td>
<td>Confusion matrix of unpruned network which classifies images of MNIST</td>
<td>43</td>
</tr>
<tr>
<td>5.3</td>
<td>Confusion matrix of network pruned with NISP and a pruning ratio of $\frac{1}{3}$</td>
<td>44</td>
</tr>
<tr>
<td>5.4</td>
<td>NISP applied on LeNet with different pruning ratios</td>
<td>46</td>
</tr>
<tr>
<td>5.5</td>
<td>Confusion matrix of unpruned network which classifies images of Cifar-10</td>
<td>50</td>
</tr>
<tr>
<td>5.6</td>
<td>Confusion matrix of network pruned with NISP and a pruning ratio of $\frac{1}{4}$</td>
<td>50</td>
</tr>
<tr>
<td>6.1</td>
<td>Classifying MNIST with $\varepsilon$-ResNet for different $\varepsilon$</td>
<td>59</td>
</tr>
<tr>
<td>6.2</td>
<td>Classifying Cifar-10 with $\varepsilon$-ResNet for different $\varepsilon$</td>
<td>61</td>
</tr>
<tr>
<td>6.3</td>
<td>Classifying a subset of Cifar-100 with $\varepsilon$-ResNet for different $\varepsilon$</td>
<td>62</td>
</tr>
</tbody>
</table>
1 Introduction

In recent years, the success of deep neural networks has become remarkable. In various applications such as speech, natural language and image processing deep neural networks achieve outstanding results and can often even exceed human performance. [1, 2]

One reason for this increasing progress is the improved hardware and software infrastructure of computers. [3, p.12]

With the computational ability to train deep neural networks, enormous resources are used to achieve the best performance. [4] Consequently the networks are extremely deep, which means that they have millions or even billions of parameters.

Due to the huge amount of parameters the networks are highly over-parameterized and contain a lot of redundancy.

In addition these networks are computationally and memory-wise very expensive. [2]

As a result the networks are not always applicable to real world scenarios. For example, real-time predictions are not possible as the predictions are about a factor 100 away in terms of speed from what the networks can accomplish. Furthermore running the networks on hardware-limited devices such as smart phones, robots or cars is difficult. [1]

However by removing the redundancy equally good results can be achieved with more compact models.

Therefore in current research compression and efficiency have become more and more important. Various methods have been proposed to decrease the size of these networks while retaining their predictive power.

In this thesis three different pruning methods will be analyzed in detail.

To evaluate the effectiveness of the methods, experiments on networks for image classification are performed. The goal is to find a pruning approach which reduces the number of parameters with negligible accuracy loss.
CHAPTER 1. INTRODUCTION

The rest of the thesis is organized as follows. Chapter 2 gives a short introduction to deep neural networks by explaining the concept of multilayer perceptrons, convolutional neural networks and residual networks.

After that an overview of different types of pruning techniques is given in Chapter 3.

Then three selected pruning approaches are examined. First singular value decomposition is presented in Chapter 4. In the next Chapter 5, a technique called neuron importance score propagation is evaluated. The last presented method to reduce the number of parameters is referred to as $\varepsilon$-ResNet. This technique is analyzed in Chapter 6.

After a comprehensive analysis of the three different techniques, the pruning results are compared in Chapter 7.

Next, in Chapter 8 the presented techniques are combined to examine whether one pruning approach already removes all of the redundancy or if better results can be achieved by combining two methods.

In the final Chapter 9 the examined techniques and their pruning results are summarized.
2 Deep Neural Networks

2.1 Multilayer Perceptrons

Feedforward neural networks consist of layers which contain neurons just like the structure of a general neural network. However in a feedforward neural network the data only flows from the first layer through the network to the last layer. There are no feedback connections. That means the connections between the neurons do not have cycles. [3]

Multilayer perceptrons are currently the most popular type of feedforward neural networks and are mostly used for classification problems.

Classification problems can be processed using supervised learning which is a common form of machine learning.

To solve a classification task, supervised learning produces a function that maps inputs to the wanted outputs, given a labeled training set. [5]

The following figure shows the basic structure of a multilayer perceptron.
A multilayer perceptron can have any number of layers and each layer can have any number of neurons. The neurons are connected with each other and each connection defines a weight. The weights are denoted by $v_{nm}$ and $w_{kn}$ in the image. In addition, each neuron can have a bias [7]. The neurons are also often referred to as nodes.

The first layer is the input layer, the last layer is the output layer and the layers in between are the hidden layers.

This structure is referred to as a network, because many different functions are composed together. The depth $n$ of a network corresponds to the number of layers. By adding more layers, the network gets deeper and can represent functions of increasing complexity [3].

A network with $n$ layers can be defined as a function $F^{(n)} = f^{(n)} \circ f^{(n-1)} \circ \cdots \circ f^{(1)}$.

The l-th layer $f^{(l)}$ can be written in the following form

$$f^{(l)}(x) = \sigma^{(l)}(w^{(l)}x + b^{(l)}) ,$$

$\sigma^{(l)}$ is an activation function, $w^{(l)}$ are the weights and $b^{(l)}$ the bias.[8]
CHAPTER 2. DEEP NEURAL NETWORKS

Commonly used activation functions are rectified linear unit (ReLU), logistic functions
and tangens hyperbolicus.

Using the layered structure, the network inputs of a layer can be computed by a matrix
multiplication plus a bias vector. The matrix multiplication is calculated by multiplying
the weight matrix and the output vector of the preceding layer. This is the computation
of the argument of the activation function.

In recent papers, the actual application of the activation function is then modeled as a
separate layer.

For a classifier the function \( y = f^*(x) \) maps the input \( x \) to a category \( y \). The goal of the
neural network is to approximate \( f^* \) by defining a mapping \( y = f(x; \theta) \). During training
the network learns the values of the parameters \( \theta \), which approximate the function best.

For classification tasks, the output layer usually has the same number of neurons as
there are categories to classify. The output is then a vector of scores for each category.
The desired category should have the highest score.

Training the network consists of showing the network the labeled training data and com-
puting an objective function, which measures the error or rather the distance between
the output of the network and the actual labels. This function is called the cost or loss
function. That means the cost can be considered as a function of the parameters which
are the weights and bias values. Hence the training process can be seen as an optimization
task which is to find those values of the parameters that optimize or rather minimize the
cost for the given dataset.

A fairly simple approach to solve such an optimization problem is gradient descent. Gra-
dient descent is always applicable whenever the function to optimize is differentiable.

The computed gradient vector indicates how to adjust the weights and biases so that
the loss function decreases. To make all training instances contribute the loss function is
averaged over all the training examples.

The calculation of a proper gradient descent would have to use a gradient that is computed
from all training instances, a so called batch training. However, especially if the training
dataset is large, computing such a proper gradient is costly. For this reason in practice a
method called stochastic gradient descent (SGD) is widely used.

The idea of stochastic gradient descent is to approximate the true gradient, that means
calculated from all training examples. The approximation is computed from a random sample which is usually referred to as a mini batch. This allows more updates of the parameters in the same unit of time, since not all training examples have to be propagated through the network for each training step. There is then a tradeoff between the number of training examples in the sample and the reliability of the gradient approximation. The fewer number of training examples, the more updates per unit of time are possible. However the more training examples are evaluated, the more reliable the gradient will be in the sense that it will be a better approximation of the true gradient. [3]

To compute the gradient vector of the loss function with respect to the parameters, a method called backpropagation is used. The idea of backpropagation is to apply the chain rule of differentiation repeatedly. This leads to a recursive formula, each application of which propagates the error one layer back. [9]

To sum up, the training of a neural network is using backpropagation to compute the gradient vector of the cost and adjusting the parameters to minimize the error between the outputs of the network and the actual outputs. This process is repeated for many small samples from the training data.

To evaluate how well the network can generalize to new data, a separate test dataset must be used. The test dataset is also labeled and applied to the network. Just like described for the training data, the loss function can be computed. The difference now is that the parameters are not adjusted anymore.

The evaluation on the test data will be considerably worse than what can be achieved on the training data. However this separate test dataset is needed to avoid overfitting. Overfitting occurs when the properties of the training dataset are memorized. That means the network performs almost perfectly on the training data, but does not generalize well to new data. Then the difference between the training error and the test error is very large.

For this reason the training dataset is used for parameter adaption and the test dataset is required for evaluating the quality of the trained network on unseen data. [3, p.110]

Multilayer perceptrons are a powerful technique for many applications, but the input is limited to a one dimensional array. Data that comes in the form of multidimensional arrays, for example color images would have to be linearized to an one dimensional array to serve as an input. Therefore a special kind of multilayer perceptron was designed to
CHAPTER 2. DEEP NEURAL NETWORKS

process this type of data better. [9]

These networks are called convolutional neural networks and will be explained in detail in the following section.

2.2 Convolutional Neural Networks

Convolutional neural networks (CNN) are a specific type of multilayer perceptrons. They were especially designed to process multidimensional data, such as images and videos. A CNN is a multilayer perceptron that uses a convolution instead of general matrix multiplication in at least one layer. [3]

CNNs have fewer parameters and fewer connections than general neural networks, as CNNs take advantage of properties like local connections and shared weights.

A CNN can have two different types of layers, convolutional layers and pooling layers.

The neurons in a convolutional layer are organized in feature maps also called channels. Each neuron within these feature maps is connected to local patches in the feature maps of the previous layer through a set of weights. This set of weights is called kernel. [9] The kernel is like a filter that is used to slide over the image. Therefore only a small portion of the image, which is referred to as the local receptive field, is processed at once. The kernel is also often just referred to as filter. The stride of the kernel indicates, how big the step should be, with which the kernel slides over the image. The local receptive field allows the neurons to access fundamental features, such as edges or corners. Local features that may occur in different parts of an image can be captured by the kernel, as the kernel uses the same set of weights for different positions in the image. For this reason CNNs are more robust to shifting, scaling and rotation. [10]

The pooling layer merges semantically similar features into one. [9] Especially max pooling maintains the detection of the feature, but blurs the location. This is primarily useful for image recognition or object detection tasks, as it is more important that a feature occurs, than knowing the exact location of it.

The following figure shows a possible architecture for a convolutional neural network.
This network is a classic CNN architecture that was introduced in [11] to recognize handwritten digits. The two convolutional layers are both followed by subsampling, which corresponds to the pooling layer. These layers are followed by a flattening convolutional layer, that just reshapes all the elements in the feature maps into an one dimensional array. This layer is needed to connect the last two fully connected layers with the convolutional layers.

Especially networks for classification tasks end with one or more fully connected layers.

The parameters of CNNs are trained just like in feedforward neural networks. The appropriate values of the kernel are learned by using the backpropagation algorithm.

CNNs can have any number of convolutional, pooling and fully connected layers. By stacking more layers, the network gets deeper and consequently has more parameters.

With a larger number of parameters the network is more difficult to optimize and the risk of overfitting increases. A popular network architecture called ResNet was presented in [12] to ease the training of deep neural networks.

The idea and structure of this network will be described in the following section.

### 2.3 Residual Networks

ResNet, short for residual network, is a very successful network architecture for deep neural networks. [12]
ResNet lets the layers learn residual functions with reference to the layer inputs, rather than learning unreferenced functions. The Experiments in [12] prove that residual mapping is easier to optimize and therefore eases the training of deep neural networks.

Figure 2.3 shows a building block of a residual mapping. The identity mapping is performed by the so called 'shortcut connections'. These connections allow the network to skip one or more layers. So instead of letting the stacked layers of the network fit an underlying mapping $\mathcal{H}(x)$, they fit a residual mapping $\mathcal{F}(x) := \mathcal{H}(x) - x$. The original mapping is cast back to $\mathcal{F}(x) + x$. In the extreme case that the layers are not needed in the network, that means the identity mapping would be optimal, it is easier to optimize a network that pushes the residual to zero instead of fitting an identity mapping by a stack of nonlinear layers [12]. The structure of a building block used in [12] is shown in figure 2.4 in the left block.

In [13] it is proven that propagating the forward and backward signals directly from one building block to another building block, leads to faster training and better generalization. For the direct propagation identity mappings are used as the skip connections. To construct identity mappings, the activation functions, most commonly ReLU, and Batch Normalization [14] are viewed as "pre-activation" of the weight layers, which is demonstrated in figure 2.4 in the right block. In contrast the original "post-activation" is shown in the same figure 2.4, but in the left block.

![Figure 2.3: building block of ResNet](image)

![Figure 2.4: Difference in structure of the original ResNet and "pre-activation" ResNet](image)
Deeper networks may lead to higher accuracies, but they are also computationally and memory-wise more expensive.

The next chapter gives an overview of several different approaches of how to reduce the number of parameters in deep neural networks.
3 Overview of Parameter Reduction Approaches

Deep Neural Networks (DNN) have become more and more popular, as they are very successful in a wide range of applications. Especially in the field of image classification and object detection, DNNs have repeatedly achieved record breaking results. [15]

However these DNNs have millions of parameters, which make them computationally and memory-wise very expensive. For this reason, DNNs can often not be used for real time predictions and have problems running on hardware limited devices like smart phones, robots and cars. [1] Therefore the interest in model compression has increased.

There already exists a number of different approaches to tackle this problem. The two main types of strategies are pruning and quantization.

Quantization decreases the size of the model by reducing the bit precision for the parameters. There are several techniques of quantization, such as binary quantization, k-means quantization, product quantization or residual quantization. Quantization aims to reduce the storage instead of speeding up testing time. But it is also possible that with quantization more parameters are needed.[1]

However in the following we will only examine pruning approaches, as these in fact reduce the number of parameters. Pruning not only achieves acceleration but also decreases parameter redundancy. Either weights, neurons or complete layers can be pruned. For each of these approaches there exists different techniques. The techniques are differentiated according to pruning after training or during the training process. But most methods assume that the network is already fully trained and then pruning techniques are applied.

One of the first approaches to prune parameters in neural networks was to remove individual weights [16, 17] or individual neurons [18, 19, 20] depending on their influence on the output.
In [21, 22] a weight pruning method is presented, which removes weights with small magnitude.

By pruning weights the sparsity of CNNs is increased. Only with specially designed software or hardware, that can handle this sparsity, the model can achieve compression and acceleration. [23]

Singular value decomposition is also a pruning method that prunes weights, but does not need special software or hardware. Singular value decomposition is applied to the weight matrix and prunes the singular values that are close to zero. However by applying singular value decomposition and pruning the small values, the weight matrix is decomposed into two matrices with smaller dimension. These two matrices are then applied back to the model. So the number of layers in the network increases, but the layers are simplified by using smaller weight matrices. [24]

Further pruning techniques using low rank approximations are examined in [25, 26]. However in contrast to singular value decomposition these methods also decompose filters of CNNs and not only weight matrices of fully connected layers. The filters are pruned based on the observation that they are of low rank and can be decomposed by matrix factorization [23].

In [2] a low rank approximation technique is presented that is applied during the training process. This method introduces a regularizer which encourages each parameter matrix in the network to have low rank in the training loss. This paper argues that the parameter matrices of pre-trained networks do not necessarily have to have low rank. So by applying low rank approximation methods after training could truncate some of the relevant information which might lead to accuracy loss.

Additional methods of weight pruning are discussed in [27]. Also different neuron pruning methods are discussed. The paper concludes that neuron pruning is the better approach.

A widely used neuron pruning technique is to prune the neurons based on their importance. In [21, 28] neurons or channels with small magnitude of weights are pruned. The importance is determined by only considering one layer at a time. The method used in [21] prunes the neurons layer-by-layer independently. In [28] and [29], which consider two consecutive layers, the neurons are also pruned layer-by-layer, but greedily. However, neurons that seem unimportant in an earlier layer might significantly effect responses of important neurons in later layers [8]. For this reason a global measurement of neuron
importance across the network is proposed in [8].

There are several more neuron pruning methods. Such as pruning neurons that are similar to each other [30] or pruning neurons randomly [31]. Another approach is to prune unnecessary neurons in a data-driven way like in [23].

In [32] the network is also pruned in a data-driven way, but instead of pruning neurons complete layers are pruned. First a scaling factor scales the outputs of a residual block in a residual network [12]. Sparsity regularizations are then added on those factors, which solve the optimization problem by a modified stochastic Accelerated Proximal Gradient method. The corresponding residual block of the factors which are forced to zero, can be removed. [32]

The idea in [4] is to also remove redundant layers in a residual block. But instead of adding additional scaling factors that have to be trained, this approach discards unnecessary layers by redesigning the network architecture to achieve strict identity mapping. This technique is another example for a pruning method which is applied during the training process.

In the following thesis three pruning techniques will be discussed in detail. I decided to examine one approach that prunes weights, one that prunes nodes and one that prunes complete layers.

First singular value decomposition will be examined. Singular value decomposition is a very simple approach which prunes weights after a network has been trained.

After that the neuron pruning method using importance scores described in [8] will be analyzed. This method measures the importance of the neurons across the entire network instead of only considering the statistics of one or two layers like in other existing methods.

The last method that will be examined is the method which prunes complete layers introduced in [4]. In comparison to the other similar method in [32], this method does not need any additional factors that have to be trained. Furthermore this technique is a continuation of the original residual network idea, which already led to excellent results [12].
4 Singular Value Decomposition

In this chapter we will look at one simple approach to reduce the number of parameters in a neural network, by applying singular value decomposition (SVD) to the weight matrices of a fully connected layer.

4.1 Theory

Singular value decomposition is a method from linear algebra and can be used to reduce the dimension of a matrix by decomposing this matrix into two matrices with smaller dimensions.

Applying the singular value decomposition on a matrix $A_{m \times n}$, we get

$$A_{m \times n} = U_{m \times n} \Sigma_{n \times n} V^T_{n \times n}.$$

$\Sigma$ is a diagonal matrix which has the singular values of $A$ on its diagonal in decreasing order. The columns of $U$ and $V$ are orthonormal and are called left-singular and right-singular vectors of $A$. [33, p. 110] [24]

If $A$ is a sparse matrix, a lot of the singular values of $A$ should be close to 0. Therefore setting those values to 0 should not have a noticeable effect on the elements of $A$. By only using the $k$ largest singular values of $A$, we can rewrite the singular value decomposition as

$$A_{m \times n} = U_{m \times k} \Sigma_{k \times k} V^T_{k \times n} = U_{m \times k} N_{k \times n},$$

with $N_{k \times n} = \Sigma_{k \times k} V^T_{k \times n}$.

Applying SVD on a weight matrix of a neural network replaces one fully connected layer with two layers. With this method the number of parameters are reduced from $m \cdot n$ to
(m + n) · k. So we only have a significant model reduction if k is much smaller than m and n. When implementing this method, k can be set to any number between 1 and m.

The following graphik shows how to apply the decomposed matrices back to the original model.

![Reconstructed deep neural network (DNN) model after SVD](image)

**Figure 4.1:** Reconstructed deep neural network (DNN) model after SVD

[24, Fig. 4]

If the reconstructed model can not reach the initial accuracy, then the model can be fine-tuned. Fine-tuning means that the training of the model is continued for a few more epochs. [24]

### 4.2 Experiments

After looking at the theory of singular value decomposition, we will now look at some practical examples.
4.2.1 Experimental Setup

In this section, I will describe the Experimental Setup for all the practical examples of this and the next chapters.

For all the following Experiments I will be using Google Colab\(^1\). Google Colab is a free to use Jupyter notebook, that allows free usage of a Tesla K80 GPU. The GPU offers a use of twelve GB of RAM for up to twelve hours in a row.[34]

Therefore the experiments are limited to the described computing power. So I will only be using the three datasets MNIST [11], Cifar-10 and subdatasets of Cifar-100 [35] for the implementation.

The **MNIST** dataset contains pictures of handwritten digits. This dataset consists of 60,000 training examples and 10,000 test examples with images of the size $28 \times 28$ [11].

In the following figure some example images of the MNIST dataset are shown.

```
9  4  3  7  8
1  7  0  6  7
2  9  6  4  1
```

**Figure 4.2:** Random images of the MNIST dataset

**Cifar-10** shows color images of ten different classes with a size of $32 \times 32$. The ten classes are airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck. Each class has 5,000 training examples and 1,000 test examples [35].

A few examples of this dataset are presented in the following picture.

---

\(^1\)https://colab.research.google.com/


**Cifar-100** is similar to Cifar-10. It also shows color images with a size of $32 \times 32$, but instead of ten classes this dataset has 100. The 100 classes are organized in 20 super classes, each containing five classes. The entire class structure is listed in the appendix of [35]. As Cifar-100 has more classes, every class only has 500 training examples and 100 test examples.

Before training all the datasets are normalized with the datasets mean and standard deviation. In addition Cifar-10 and subsets of Cifar-100 are augmented following the standard data augmentation as described in [4]. The images are padded by four pixels on each side and then randomly cropped back to the size $32 \times 32$. And these two datasets are horizontally flipped as well. Data augmentation enlargens the dataset using transformations to improve generalization and reduce overfitting. [36]

The neural networks will be implemented in PyTorch. The loss function that I will be using is cross entropy loss [3, p.72, 129] and as a gradient descent optimization algorithm I will be using Adaptive Moment Estimation (Adam) [37]. Both of these methods are already implemented in the PyTorch module.

I set the weight decay parameter to a fixed value of $6 \cdot 10^{-6}$. This parameter prevents the weights from growing to large and can improve generalization. [38, 3] The learning rate parameter of the gradient descent optimizier starts with a value of 0.01 and decreases
depending on the training loss. If the training loss does not drop for two epochs in a row, then the learning rate decreases by a factor of ten.

4.2.2 SVD on ResNet

To evaluate how well singular value decomposition is able to reduce the number of parameters in neural networks, I started by applying SVD on ResNet. This network architecture is especially used for deep neural networks and is described in Chapter 2 in Section 2.3.

Classifying images of Cifar-10

To apply SVD, the network has to be trained first. I trained Cifar-10 on ResNet with 18 layers, following the architecture described in [13]. This architecture is almost identical with the architecture displayed in [12, Table 1] with some small changes. The network starts with a convolutional layer but instead of a kernel size of $7 \times 7$ suggested in [12, Table 1], a kernel size of $3 \times 3$ is used. Also the max pooling layer following the convolutional layer in [12, Table 1], is not considered. The next four convolutional blocks are as in [12, Table 1]. Each block consists of two building blocks with the structure of the "pre-activation" ResNet in 2.4 (right block). The network ends with an average pooling and a fully connected layer for classifying the ten categories of Cifar-10. All together this network has $11,171,146$ parameters.

Training the described network, I reached an accuracy of 92.08\% after only 70 epochs.

Only fully connected layers have weight matrices, so SVD can only be applied on one layer in ResNet. The weight matrix $A$ of the fully connected layer has a size of $10 \times 512$. By applying SVD, which is already implemented in the numpy module [39], on the weight matrix $A$, $\Sigma$ is calculated. The singular values of $A$ are on the diagonal of $\Sigma$ in decreasing order.
Unfortunately the singular values of $A$ are not very small. So reducing the number of parameters by only keeping the $k$—biggest singular values of $A$ will probably lead to a noticeable accuracy loss.

The following table shows the accuracy and the number of parameters of the neural network after keeping the $k$—biggest singular values of $A$. The values of the column *accuracy after fine-tune* are the accuracies after fine-tuning the network for another ten epochs.

<table>
<thead>
<tr>
<th>$k$</th>
<th>accuracy after fine-tune</th>
<th>accuracy after fine-tune</th>
<th>number of parameters in FC</th>
<th>difference of parameters</th>
<th>number of parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>91.82%</td>
<td>91.79%</td>
<td>4,708</td>
<td>422</td>
<td>11,170,724</td>
</tr>
<tr>
<td>8</td>
<td>87.50%</td>
<td>91.87%</td>
<td>4,186</td>
<td>944</td>
<td>11,170,202</td>
</tr>
<tr>
<td>7</td>
<td>86.20%</td>
<td>91.72%</td>
<td>3,664</td>
<td>1,466</td>
<td>11,169,680</td>
</tr>
<tr>
<td>6</td>
<td>80.38%</td>
<td>91.95%</td>
<td>3,142</td>
<td>1,988</td>
<td>11,169,158</td>
</tr>
<tr>
<td>5</td>
<td>80.13%</td>
<td>91.73%</td>
<td>2,620</td>
<td>2,510</td>
<td>11,168,636</td>
</tr>
<tr>
<td>4</td>
<td>72.85%</td>
<td>91.09%</td>
<td>2,098</td>
<td>3,032</td>
<td>11,168,114</td>
</tr>
<tr>
<td>3</td>
<td>63.86%</td>
<td>91.11%</td>
<td>1,576</td>
<td>3,554</td>
<td>11,167,592</td>
</tr>
<tr>
<td>2</td>
<td>53.38%</td>
<td>89.85%</td>
<td>1,054</td>
<td>4,076</td>
<td>11,167,070</td>
</tr>
<tr>
<td>1</td>
<td>20.11%</td>
<td>58.78%</td>
<td>532</td>
<td>4,598</td>
<td>11,166,548</td>
</tr>
</tbody>
</table>

Table 4.1: SVD with different values of $k$ applied on fully connected layer of ResNet.
The fully connected layer has 5,130 parameters, 5,120 parameters are used in the $10 \times 512$ weight matrix and 10 parameters are needed for the bias. The column *number of parameters in FC* holds the number of parameters in the fully connected layer after SVD was applied.

The table shows that only for $k = 9$, that means just one singular value was set to zero, the network hardly has an accuracy loss. This merely reduces the network by 422 parameters, which is only 0.004% of the parameters. However if you fine tune the model for another ten epochs after applying SVD, the network, independent of $k$, barely loses accuracy. Only for $k = 1$ the network can’t reach the initial accuracy after fine tuning for ten epochs.

So even though the weight matrix is not a sparse matrix, applying SVD still reduces the number of parameters with almost no accuracy loss after fine tuning. Therefore the number of parameters can also be reduced with SVD by changing the structure of the fully connected layer to have less parameters.

That means if SVD can reduce the number of parameters by changing the structure of the layer, then the values of the weight matrix are irrelevant, so it would make more sense to apply SVD before training the network. Strictly speaking SVD is not actually applied, rather the structure that would result from applying SVD is used and the parameters of this structure are randomly initialized before training.

I used the same network architecture described above and applied SVD with $k = 3$ before training the network. I chose $k = 3$, because this parameter led to a considerable parameter reduction, but still achieved almost the initial accuracy in the example above. This network reached an accuracy of about 90% after training for 70 epochs. So applying SVD before training leads to a network with less parameters, but in this case can not quite reach the same accuracy, as the network without SVD, in the same amount of epochs. Consequently applying SVD in this example is kind of a trade-off between the number of parameters and the number of epochs. The lower accuracy could have also resulted from a less favorable initialization, as the initialization of the parameters is random. So to be able to make a general statement, the training should be repeated several times, but due to the limited computing power this is not carried out.

However SVD in this example only reduces parameters by about 0.03%, which is negligible in such a big network. Therefore it makes more sense to look at cases where the weight matrix is in fact a sparse matrix or the network has more fully connected layers.
CHAPTER 4. SINGULAR VALUE DECOMPOSITION

Classifying images of MNIST

In this subsection, I will train ResNet on the MNIST dataset and then apply SVD on the weight matrix of the fully connected layer. The MNIST dataset is easier to classify than the Cifar-10 dataset and achieves almost perfect accuracy after only training for a few epochs. As the classification problem is simpler, the weight matrix of the fully connected layer may be a sparse matrix and therefore more singular values would be close to zero.

I trained the MNIST dataset on the 18-layer ResNet described previously. After only 20 epochs the network achieves an accuracy of 99.39%. This network has 11,169,994 parameters, which is slightly less than the ResNet to classify Cifar-10 images. This is due to the fact that the MNIST dataset only has grayscale images, so the input of the first convolutional layer is only one channel. In contrast the Cifar-10 dataset contains color images which consist of three channels. Therefore the input of the first convolutional layer has three channels which leads to slightly more parameters.

The singular values of the weight matrix are shown below.


Unfortunately these singular values are also not very small. Applying SVD on this matrix would probably lead to similar results like in the previous example. And reducing the number of parameters without accuracy loss would only be possible with fine-tuning.

These examples show that it is not very effective to apply SVD on ResNet to reduce the number of parameters in the network. ResNet only has one fully connected layer, which is the layer before classification, and the weight matrix of this layer does not contain singular values close to zero. Furthermore the size of the weight matrix in ResNet is \(10 \times 512\), so at the most 9 singular values could be set to zero. This would lead to a significant reduction in the fully connected layer, as the 5,130 parameters could be reduced by 4,598 parameters, which is a reduction of almost 90%. Nonetheless this reduction only contributes to 0.04% reduction in the entire network. So even if the weight matrix would be a sparse matrix and a lot of the singular values could be set to zero, the number of parameters would hardly be reduced.

Reducing the other parameters would require adapted techniques, for example, decomposition of the kernels, because SVD is only easily applicable on fully connected layers.
In the next section, I will see how well SVD can reduce parameters in a network with more fully connected layers.

4.2.3 SVD on LeNet

LeNet is a small standard CNN architecture that was introduced in [11] to recognize handwritten digits. This architecture is described in Chapter 2, the introductory chapter to CNNs. The exact network structure is shown in figure 2.2 in that chapter. In this section, I will be using an architecture in LeNet-style. So the network will have a slightly different structure. This network, just like LeNet, consists of two convolutional layers and two fully connected layers. Each convolutional layer is followed by a pooling layer and a rectified linear unit function. The first fully connected layer is followed by a rectified linear unit function as well. Only the two fully connected layers have different sizes than in LeNet. In the original LeNet architecture the first fully connected layer connects 120 neurons with 84 neurons, which are then connected with the 10 output neurons. In this network the first fully connected layer connects 320 neurons with 50 neurons. And those 50 neurons are connected with the 10 neurons for classification. All the parameters add up to 21,840.

Training this network on the MNIST dataset for only 10 epochs, leads to an almost perfect accuracy of about 98.8%. The weight matrix of the first fully connected layer has a size of $50 \times 320$. The singular values of this weight matrix are shown in the following graph.

![Figure 4.4: singular values of the weight matrix of the first fully connected layer](image_url)
The singular values of this weight matrix are significantly smaller than the singular values in the ResNet network. About 15 of the values are very small, so we would expect that setting these values to zero should not affect the accuracy. However if even more singular values are set to zero, the accuracy does not decrease. The next graph shows the accuracies after keeping the $k$ largest singular values. These are the accuracies right after applying SVD without fine-tuning the network.

![Graph showing accuracies after applying SVD for different $k$'s on the first fully connected layer.]

**Figure 4.5:** accuracies [in %] after applying SVD for different $k$'s on the first fully connected layer

Until about $k = 15$, that means 35 singular values were set to zero, the model hardly loses accuracy. Even though values up to almost 5 are set to zero.

For $k = 15$ the network still has an accuracy of about 98.3%, but is reduced by 10,450 parameters. So by applying SVD, this model can be reduced by 47.85% with almost no accuracy loss.

Next we will look at the singular values of the second fully connected layer, but independently of the first fully connected layer. So the accuracies are calculated on the network, without applying SVD on the first fully connected layer.

The size of the weight matrix of the second fully connected layer is $10 \times 50$. The singular values of this matrix are plotted in the following left graph.
CHAPTER 4. SINGULAR VALUE DECOMPOSITION

The right plot of the graph above shows the accuracies after applying SVD for different k’s. Even though the largest singular value is about 2.3, only setting the two smallest singular values to zero does not decrease the accuracy. In comparison, in the first fully connected layer values up to almost 5 could be set to zero without accuracy loss. Therefore it is not possible to find a general threshold for the singular values that indicates how many values can be set to zero without accuracy loss. Instead for every layer the best k has to be found, which is also time consuming.

In this second fully connected layer setting the two smallest values to zero may not decrease the accuracy, but the number of parameters is also not reduced. For $k = 9$ the network has even more parameters than the original one. Since applying SVD on a layer replaces one layer with two layers. In this case the layer with the weight matrix of size $10 \times 50$ is replaced by two layers with one matrix of size $9 \times 50$ and another matrix of size $10 \times 9$. The weight matrix of size $10 \times 50$ has 500 parameters and after applying SVD the two matrices together have 540 parameters.

Consequently to reduce the number of parameters in this model it would only make sense to apply SVD on the first fully connected layer. Doing so the model can be reduced by almost 50%.

The reason why so many parameters could be reduced is probably because the MNIST dataset is so simple and the model therefore contains a lot of redundancy.

As the MNIST dataset is so simple, it is also possible so train this dataset on a fully
connected neural network. In the next section, we will look at how well SVD can reduce parameters in a network only containing fully connected layers.

4.2.4 SVD on a Fully Connected Neural Network

The MNIST dataset is relatively easy and just contains grayscale values, so it is possible to train this dataset on a network with only fully connected layers.

The structure of the fully connected layer is displayed in the following graphic.

The network has one input layer, one hidden layer and one output layer. The input layer has 784 nodes, one for every pixel in the $28 \times 28$ image. The hidden layer has 30 nodes and the output layer has 10 nodes for classifying the digits. The rectified linear unit function is the activation function of the hidden layer. The activation function of the output layer is simply the identity function.

The described network has a total of 23,860 parameters and reached an accuracy of 96.47% after training for 20 epochs. Even though this network only has two fully connected layers, it still has more parameters than LeNet, which consists of two convolutional layers and two fully connected layers. The reason for this is that convolutional layers take advantage of weight sharing and therefore do not have very many parameters. Also the convolutional layers at the beginning of the network produce less output values than there are inputs.
to this network. So the large input layer with 784 nodes is not needed.

This network also can not quite reach the same accuracy as LeNet, because convolutional networks are better at processing images.

However in the following we will still look at how well SVD can reduce the number of parameters. The size of the first weight matrix is $30 \times 784$. The singular values of this matrix are plotted in the following left graph and the accuracies after applying SVD for different k’s is shown in the following right graph.

![Figure 4.8: singular values of the weight matrix of the first fully connected layer](image)

![Figure 4.9: accuracies [in %] after applying SVD for different k’s on the first fully connected layer](image)

Even though the singular values are not very small, this model barely loses accuracy up to $k = 17$. For $k = 17$ the network still has an accuracy of about 95% and the parameters are reduced by 9,682.

Applying SVD on the second fully connected layer, independently of the first fully connected layer, is probably not necessary. Since the number of parameters are not reduced if only one or two singular values are set to zero.

This is confirmed looking at the following graphic. The left plot shows the singular values of the weight matrix and the right plot shows the accuracies after applying SVD for different k’s.
For $k = 8$ the accuracy already decreases to 89%.

Consequently the results are similar to the results for applying SVD on LeNet. Only applying SVD on the first fully connected layer reduces the size of the network. This fully connected network can be reduced by 40.58% without accuracy loss. In comparison LeNet can be reduced by 47.85%. The reason for this could be that this network is even simpler than LeNet and therefore probably does not contain as much redundancy. As a result less parameters can be pruned.

4.3 Conclusion

Singular value decomposition is a simple approach to reduce the number of parameters in a network that has been trained. The weights are pruned by applying SVD on the weight matrix of a fully connected layer. The weight matrix is decomposed into two matrices with smaller dimensions. The number of parameters changes from $m \cdot n$ to $(m + n) \cdot k$. However $(m + n) \cdot k$ does not necessarily have to be smaller than $m \cdot n$, which was also demonstrated in the example above in Section 4.2.3. Consequently SVD only achieves a considerable model reduction if $k$ is significantly smaller than $m$ and $n$.

Therefore this method is not very effective on most CNNs. Common CNN architectures, such as ResNet, only have one fully connected layer, which is the layer before classification. As demonstrated in the previous experiments reducing the number of parameters in this
layer with SVD is not possible without accuracy loss.

The reason for this could be that the last fully connected layer contains too much information, as that layer is the direct input of the classification task. In the next chapter another pruning method is analysed, where the last fully connected layer plays a key role, because of its importance.

However by applying SVD on a network with more fully connected layers, a significant model reduction can be achieved. Almost 50% of the parameters in LeNet can be pruned with negligible accuracy loss. All of these parameters were pruned in the first fully connected layer. Just as in ResNet almost no parameters in the last fully connected layer could be reduced without decreasing accuracy.

Applying SVD to a network with only two fully connected layers behaves similar to LeNet. Only reducing the number of parameters in the first fully connected layer led to a considerable reduction. Even though singular values in the second fully connected layer might be smaller than singular values in the first fully connected layer, setting them to zero decreases the accuracy. As a result finding a general threshold for the singular values is difficult.

All in all SVD is especially effective on networks with more fully connected layers. However CNNs can process image data better than fully connected networks. For this reason most networks for image classification tasks are CNNs. Therefore a method that can also prune parameters in convolutional layers probably leads to a higher model reduction.

In the next chapter, we will look at a pruning technique that prunes nodes in fully connected layers and channels in convolutional layers.
5 Neuron Importance Score Propagation

In this chapter we will look at a method called neuron importance score propagation (NISP). This approach is presented in the paper [8], which this complete chapter is based on. The described method reduces the number of parameters in neural networks by pruning neurons in every layer with the least importance. To retain the accuracy the network is then fine-tuned.

Like in the previous chapter, we will first look at the theory of this method and then some practical examples.

5.1 Theory

There already exists various techniques that prune neurons in neural networks, but most of these techniques assume one layer can only affect the next layer. For this reason existing methods only consider the statistics of one or two layers and prune the neurons in each layer independently or greedily [21, 28, 29].

However layers that seem unimportant in earlier layers can significantly affect the responses of important neurons in later layers. Therefore NISP measures the neuron importance of the entire network. NISP starts by measuring the importance of the neurons in the "final response layer" (FRL), which is the second-to-last layer, that is, the layer before the final classification layer. The importance scores of FRL are then propagated to all the other neurons in the network. Neurons with least importance are removed according to the pruning ratio. The pruning ratio per layer is a hyper-parameter and therefore to be specified by an user.

From a theoretical perspective, the approach in this paper defines the network pruning as a binary integer optimization problem. The goal is to minimize the weighted $\ell^1$ distance between the original final response and the one produced by the pruned network. The weighted $\ell^1$ distance is proportional to the importance scores.
The reason for minimizing the reconstruction error of the importance scores in the FRL is, because FRL is the second-to-last layer before classification and therefore the direct inputs of the classification task.

The importance of the neurons in the FRL are measured with a feature ranking technique, by treating the neurons as features. The ranking technique used here is called Infinite Feature Selection (Inf-FS) [40]. Inf-FS is a filter algorithm, that ranks the features in an unsupervised manner and then selects the best $m$ features by a simple cross-validation strategy. This method uses convergence properties of the power series of matrices and evaluates the relevance and redundancy of a feature while considering all the possible subsets of features. [40] Using this feature ranking technique, the classification layer is irrelevant for computing the feature score. Therefore this method can be used for both classification and numeric prediction.

This feature ranking algorithm is applied on the FRL and with a backward pass through the network all importance scores are computed recursively.

The exact solution of the importance scores of the neurons is hard to compute, given the complexity of nonlinearity. Therefore the approximation of these scores is based on Lipschitz continuity of a neural network. This approximation will be explained in the following section.

### 5.1.1 Problem Definition

The goal of NISP is to minimize the weighted $\ell^1$ distance between the original final response and the one produced by the network after pruning neurons of a specific layer.

In the following bold symbols are used to represent vectors and matrices.

A neural network with depth $n$ can be formulated as a nested function $F(n) = f^{(n)} \circ f^{(n-1)} \circ \cdots \circ f^{(1)}$. In which the $l$-th layer $f^{(l)}$ represents the following

$$f^{(l)}(x) = \sigma^{(l)}(w^{(l)}x + b^{(l)}).$$

$\sigma^{(l)}$ is an activation function and $w^{(l)}$ and $b^{(l)}$ are weight and bias vectors.

The skip connections in ResNet can be rewritten to this depiction by padding weights and merging layers.
The importance scores of the $l$-th layer are represented by the vector $s_l$. For every neuron the importance score is defined as a non-negative value. The neuron prune indicator $s_l^*$ of the $l$-th layer is a binary vector and computed based on the neuron importance scores $s_l$. $s_{l,i}^* = 1$ if and only if $s_{l,i}$ is one of the $N_l$ largest values in $s_l$. $N_l$ is the number of neurons that are kept after pruning the $l$-th layer. If the value of $s_{l,i}$ is not among the top $N_l$ values, then $s_{l,i}^* = 0$.

Given a dataset with $M$ samples, in which each is represented using $x_0^{(m)}$, the response of the $l$-th layer (which is the input to the $(l+1)$-th layer), for the $m$-th sample is represented by the vector $x_l^{(m)}$. For the $m$-th sample the final output of the network is $x_n^{(m)}$ and its corresponding importance score is $s_n$. A sub-network of $F^{(n)}$ from the $i$-th layer to the $j$-th layer can be defined as

$$G^{(i,j)} = f^{(j)} \circ f^{(j-1)} \circ \cdots \circ f^{(i)}.$$  \hspace{1cm} (5.1)

The aim is to compute the neuron prune indicator $s_l^*$ of the $l$-th layer so that the weighted $\ell^1$ distance between the original final response and the one produced by the network after pruning neurons of the $l$-th layer is minimized. This goal is achieved by defining an optimization objective w.r.t. the $l$-th layer neuron prune indicator

$$\arg \min_{s_l^*} \sum_{m=1}^{M} F(s_l^* \mid x_l^{(m)}, s_n; G^{(l+1,n)}),$$  \hspace{1cm} (5.2)

which is accumulated over all $M$ samples.

For a single sample the objective function is defined as

$$F(s_l^* \mid x, s_n; F) = \langle s_n, \| F(x) - F(s_l^* \odot x) \| \rangle.$$  \hspace{1cm} (5.3)

$\langle \cdot, \cdot \rangle$ is the dot product, $\odot$ is the element-wise product and $\| \cdot \|$ is the element-wise absolute value.

The solution of 5.2 suggests which neurons to prune in the $l$-th layer. But obtaining an efficient analytical solution by directly optimizing Equation 5.2 is very difficult. Therefore the solution will be approximated using the Lipschitz continuity of a neural network.
5.1.2 Solution

Reformulating the network pruning problem as a binary integer programming objective leads to Equation 5.2. Solving this Equation, that means finding the optimal neuron prune indicator, is difficult. In [8] an upper bound on this objective is derived and it is shown that a sub-optimal solution can be obtained by minimizing the upper bound. As described previously the $k$-th layer is defined as $f^{(k)}(x) = \sigma^{(k)}(w^{(k)}x + b^{(k)})$. Since most of the commonly used activation functions in neural networks such as Identity, ReLU, sigmoid, tanh, PReLU, etc. are Lipschitz continuous, it is also assumed that the activation function $\sigma^{(k)}$ is Lipschitz continuous. Under this assumption there exists for any $x, y$ a constant $C^{(k)}_{\sigma}$ such that $| \sigma^{(k)}(x) - \sigma^{(k)}(y) | \leq C^{(k)}_{\sigma} | x - y |$. This then leads to the following equation

$$| f^{(k)}(x) - f^{(k)}(y) | \leq C^{(k)}_{\sigma} | w^{(k)} | \cdot | x - y |, \tag{5.4}$$

where $| \cdot |$ is the element-wise absolute value.

Equation 5.1 can be rewritten as $G^{(i,j)} = f^{(j)} \circ G^{(i,j-1)}$. Therefore there also exists a constant $C^{(j)}_{\sigma}$, for any $x, y$, such that

$$| G^{(i,j)}(x) - G^{(i,j)}(y) | \leq C^{(j)}_{\sigma} | w^{(j)} | \cdot | G^{(i,j-1)}(x) - G^{(i,j-1)}(y) |. \tag{5.5}$$

Applying Equation 5.4 and Equation 5.5 repeatedly leads to, $\forall i \leq j \leq n$,

$$| G^{(i,n)}(x) - G^{(i,n)}(y) | \leq C^{(i,n)}_{\Sigma} | W^{(i,n)} | \cdot | x - y |, \tag{5.6}$$

where $W^{(i,j)} = | w^{(j)} | \cdot | w^{(j-1)} | \cdot \cdots \cdot | w^{(i)} |$ and $C^{(i,j)}_{\Sigma} = \prod_{k=1}^{j} C^{(k)}_{\sigma}$. Substituting $x = x^{(m)}_l$, $y = s^*_l \odot x^{(m)}_l$, $I = l + 1$ into Equation 5.6, results in

$$| G^{(l+1,n)}(x^{(m)}_l) - G^{(l+1,n)}(s^*_l \odot x^{(m)}_l) | \leq C^{(l+1,n)}_{\Sigma} | W^{(l+1,n)} | \cdot | x^{(m)}_l - s^*_l \odot x^{(m)}_l |. \tag{5.7}$$

Since $s_n$ is a non-negative vector,

$$\mathcal{F}(s^*_l \mid x^{(m)}_l, s_n; G^{(l+1,n)}) = \langle s_n, | G^{(l+1,n)}(x^{(m)}_l) - G^{(l+1,n)}(s^*_l \odot x^{(m)}_l) | \rangle \tag{5.8}$$

$$\leq \langle s_n, C^{(l+1,n)}_{\Sigma} W^{(l+1,n)} | x^{(m)}_l - s^*_l \odot x^{(m)}_l | \rangle \tag{5.9}$$

$$= C^{(l+1,n)}_{\Sigma} (W^{(l+1,n)})^r s_n, (1 - s^*_l) \odot | x^{(m)}_l |. \tag{5.10}$$
By defining \( r_l = W^{(l+1),n}T \) and accumulating over all the samples of the dataset, the equation above leads to

\[
\sum_{m=1}^{M} F(s_l^* | x_l^{(m)}, s_n; G^{(l+1),n}) \leq C_{\Sigma}^{(l+1),n} \sum_{m=1}^{M} \langle r_l, (1 - s_l^*) \odot | x_l^{(m)} | \rangle \]  

(5.11)

\[
\leq C_{\Sigma}^{(l+1),n} \sum_{i=1}^{r} \sum_{m=1}^{M} r_{l,i}(1 - s_{l,i}^*) | x_{l,i}^{(m)} | \]  

(5.12)

\[
= C_{\Sigma}^{(l+1),n} \sum_{i=1}^{r} r_{l,i}(1 - s_{l,i}^*) \sum_{m=1}^{M} | x_{l,i}^{(m)} | . \]  

(5.13)

Since \( | x_{l,i}^{(m)} | \) is bounded, there exists a constant \( C_x \) such that \( \sum_{m=1}^{M} | x_{l,i}^{(m)} | \leq C_x, \forall i \).

This results in the following upper-bound

\[
\sum_{m=1}^{M} F(s_l^* | x_l^{(m)}, s_n; F^{(l+1)}) \leq C \sum_{i=1}^{r} r_{l,i}(1 - s_{l,i}^*), \]  

(5.14)

where \( C = C_{\Sigma}^{(l+1),n}C_x \) is a constant factor. This Equation 5.14 is an upper-bound for the objective in Equation 5.2. Minimizing this upper-bound is equivalent to the following equation

\[
\text{arg min}_{s_l^*} \sum_{i=1}^{r} r_{l,i}(1 - s_{l,i}^*) \Leftrightarrow \text{arg max}_{s_l^*} \sum_{i=1}^{r} s_{l,i}^* r_{l,i}. \]  

(5.15)

As Equation 5.15 is an upper-bound, the optimal solution is only sub-optimal with respect to the original objective in Equation 5.2, but still captures the importance of the neurons.

If \( N_x \) neurons are kept after pruning the \( l \)-th layer, then the solution of Equation 5.15 is that \( s_{l,i}^* = 1 \) if and only if \( r_{l,i} \) is among the \( N_x \) largest values in \( r_l \). By applying this conclusion to every layer in the network, the neuron importance of a network can be defined as follows.

**Definition 5.1: Neuron importance score**

Given a neural network \( F^{(n)} \) containing \( n \) layers and the importance score \( s^{(n)} \) of the last layer response, the importance score of the \( k \)-th layer response can be computed as

\[
s_k = |w^{(k+1)}|^T |w^{(k+2)}|^T \cdots |w^{(n)}|^T s_n, \]  

(5.16)

where \( w^{(i)} \) is the weight matrix of the \( i \)-th layer.
[8, Definition 1]

This definition indicates that the importance score of every layer is correlated with the importance of the final response.

However the neuron importance can also be computed recursively or rather propagated along the network which leads to the following proposition.

**Proposition 5.2** (Neuron importance score propagation): *The importance score of the $k^{th}$ layer response can be propagated from the importance score of the $(k+1)^{th}$ layer by*

$$s_k = |w^{(k+1)}| s_{k+1},$$  \hspace{1cm} (5.17)

*where $w^{(k+1)}$ is the weight matrix of the $(k+1)^{th}$ layer.*

[8, Proposition 2]

This equation can be rewritten to the following equivalent form without using vector notations

$$s_{k,j} = \sum_i |w_{i,j}^{(k+1)}| s_{k+1,i}.$$  \hspace{1cm} (5.18)

$s_{k,j}$ is the importance score of the $j$-th neuron in the $k$-th layer.

Using this equation the importance score of a lower layer can be directly propagated from the adjacent layer above it.

The implementation of the NISP algorithm is based on this Proposition 5.2 and will be explained in the next section.

### 5.2 Implementation

The first step of implementing NISP is to apply the feature ranking technique on the final response layer to obtain the importance scores of these neurons. Then the equation in Proposition 5.2 is used repeatedly to compute the importance scores of all neurons in the network with a single backward pass.

The pruning ratio of each layer is a hyper-parameter and therefore given before implementation. Based on the computed neuron importance scores the prune indicator is calculated.
Neurons that have a prune indicator value of 0 are removed and these neurons are not propagated to any further low-level layers.

Since fully connected layers consist of neurons, each individual neuron is pruned. In comparison, for convolution layers the importance score of a channel is computed by summing up the importance scores of all the neurons within a channel. Then a whole channel of neurons is pruned, that is, all neurons of the channel are removed.

In principle there are three different classes in which NISP can be divided into.

In [8] these classes are defined as follows. The first class describes the propagation of the importance scores from a 1-way tensor to a 1-way tensor, which corresponds with the propagation between fully connected layers. A 1-way tensor denotes an one dimensional tensor. The next class refers to the propagation between a fully connected layer and a convolutional or pooling layer. This is the propagation from a 1-way tensor to a 3-way tensor. The last class represents the propagation between two 3-way tensors, for example between two convolutional layers.

The propagation of the bias is ignored to simplify NISP. This is possible as the Lipschitz continuity does not change if a constant is added to the argument, which is what a bias value effectively does.

5.2.1 NISP: 1-way tensor to 1-way tensor

For the propagation of a fully connected layer with $M$ input neurons and $N$ output neurons, the $N$-by-1 importance score vector $S$ of the output neurons is given,

$$S_{FC_{out}} = [S_{FC_{out1}}, S_{FC_{out2}}, \ldots, S_{FC_{outN}}]^T \in \mathbb{R}^N.$$ 

The weight matrix of the fully connected layer is denoted as $W_{FC} \in \mathbb{R}^{M \times N}$.

The importance score of the input neurons is then calculated with the following equation.

$$S_{FC_{in}} = |W_{FC}| \cdot S_{FC_{out}},$$

where $|\cdot|$ is the element-wise absolute value.
5.2.2 NISP: 1-way tensor to 3-way tensor

The given fully connected layer has a 3-way tensor as an input, which can be the response of a convolutional layer or a pooling layer. The input size of this layer is $X \times X \times C$, where $X$ represents the spatial size and $C$ the number of input channels. The output of this fully connected layer consists of $N$ output neurons. $W_{FC} \in \mathbb{R}^{(X^2 \times C) \times N}$ denotes the weights of the fully connected layer. The flattened importance score vector $S_{in} \in \mathbb{R}^{(X^2 \times C) \times 1}$ of the input tensor is calculated just like above:

$$S_{in} = |W_{FC}| \cdot S_{FC\text{out}}.$$ 

5.2.3 NISP: 3-way tensor to 3-way tensor

Convolutional Layer

The given parameters of a convolutional layer with a 3-way input tensor $\text{conv}_{in} \in \mathbb{R}^{X \times X \times N}$ and output tensor $\text{conv}_{out} \in \mathbb{R}^{Y \times Y \times F}$ are the filter size $k$, stride $s$ and number of padded pixels $p$. The output responses of a convolutional layer are calculated during the forward propagation. The calculation consists of multiple inner products between a kernel $f \in \mathbb{R}^{k \times k \times N}$ and multiple corresponding receptive cubes. The spatial convolutional kernel is $k_{fn}$ for a fixed input channel $n$ and a fixed output channel $f$. The following equation defines the calculation of the response of the output channel $f$ at position $i$ for the same position in the $n$-th input channel.

$$R_f(i) = \sum_n k_{fn} \cdot \text{in}(i),$$

where $\text{in}(i)$ is the corresponding 2-D receptive field. Similar to this calculation the importance scores are propagated. $S_{out} \in \mathbb{R}^{Y \times Y \times F}$ is the given importance cube of the output response, which is propagated to the input as follows

$$S_n(i) = \sum_f k_{fn} \cdot S_{out}(i), \quad (5.19)$$

where $S_n(i)$ is the importance of position $i$ in the $n$-th input channel and $S_{out}(i)$ is the corresponding 2-D matrix. This matrix contains the output positions whose responses come from the value of that input position during forward propagation.

The importance scores are propagated proportionally to the weights. The propagation
algorithm is described in the following pseudocode.

**Algorithm 1 NISP: convolutional layer**

1: **Input**: weights of the conv layer $W \in \mathbb{R}^{k \times k \times N \times F}$, flattened importance of the $f$-th output channel $S_{f_{out}}^{f} \in \mathbb{R}^{1 \times (Y \times Y)}$
2: for $n$ in $1 \ldots N$
3:   for $f$ in $1 \ldots F$
4:     $k_{fn} \leftarrow |W[;:;n,f]|$
5:     Construct $\text{BP}_{conv}^{fn}$ as 5.20 and 5.21
6:     $S_{in}^{f} \leftarrow S_{out}^{f} \cdot \text{BP}_{conv}^{fn}$
7:     $S_{in} \leftarrow [S_{in}^{1}, S_{in}^{2}, \ldots, S_{in}^{N}]$
8: end

The propagation matrices used in this pseudocode are defined in the following two equations.

$$\text{BP}_{conv}^{fn} = \begin{pmatrix} b_{1}^{fn} & \ldots & b_{j}^{fn} & \ldots & b_{k}^{fn} \\ b_{i}^{fn} & \ldots & b_{j}^{fn} & \ldots & b_{k}^{fn} \\ & \vdots & & & \\ b_{i}^{fn} & \ldots & b_{j}^{fn} & \ldots & b_{k}^{fn} \end{pmatrix}, \quad (5.20)$$

where $b_{c}^{i}$ is defined as the following building block with size $Y \times X$.

$$b_{i}^{fn} = \begin{pmatrix} k_{fn}[i, 1] & \ldots & k_{fn}[i, k] \\ k_{fn}[i, 1] & \ldots & k_{fn}[i, k] \\ & \vdots & \\ k_{fn}[i, 1] & \ldots & k_{fn}[i, k] \end{pmatrix}, \quad (5.21)$$

Equation 5.19 implies that the propagation of importance scores of convolutional layers can be decomposed into propagation between 2-D matrices, by fixing the input channel $n$ and output channel $f$. The input size is then $X \times X$ and the output size is $Y \times Y$. If the layer has padding $p$, the input size is $(X + 2p) \times (X + 2p)$ and after the importance scores are calculated the scores of size $X \times X$ are cropped from the center.

The flattened importance score vector $S_{out}^{f} \in \mathbb{R}^{1 \times (Y \times Y)}$ is already given. To propagate

---

1This pseudocode is the same pseudocode as in [8, Algorithm 1] with two small changes. $W \in \mathbb{R}^{X \times X \times N \times F}$ was changed to $W \in \mathbb{R}^{k \times k \times N \times F}$ and $S_{out}^{f} \in \mathbb{R}^{1 \times (X \times X)}$ was changed to $S_{out}^{f} \in \mathbb{R}^{1 \times (Y \times Y)}$. These seem to be minor mistakes in the pseudocode given in [8].
the importances from $S_{out}^{f}$ to the importances of the input layer $S_{in}^{f} \in \mathbb{R}^{1 \times (X \times X)}$, the propagation matrix $BP_{conv}^{fn} \in \mathbb{R}^{(Y \times Y) \times (X \times X)}$ is used.

If $BP_{conv}^{fn}(i,j) \neq 0$ then that means the $i$-th position in the output layer comes from a convolution operation with the $j$-position in the input layer. And the importance is propagated between those two positions. The matrix $b_{i}^{fn} \in \mathbb{R}^{Y \times X}$ is used to represent the mapping between a row in the output layer to the corresponding row in the input layer. Each row of $b_{i}^{fn}$ contains $k$ non-zeros, because each position in the output layer is obtained from a region with width $k$ of the input layer. These non-zeros are the $i$-th row of the convolutional kernel $k_{fn}$. An offset with stride $s$ is at the beginning of the weights in each row.

Altogether the propagation matrix $BP_{conv}^{fn}$ is a block matrix, in which each submatrix has size $Y \times X$ either containing $b_{i}^{fn}$ or a zero matrix. Each row of $BP_{conv}^{fn}$ has $b_{1}^{fn}$ to $b_{k}^{fn}$, as the height of a convolutional kernel is $k$. The beginning of the $b$s in each row of $BP_{conv}^{fn}$ has an offset of stride $s$.

To illustrate how this algorithm works, a simple example is given in [8], Figure 7.

**Pooling Layer**

The importance score propagation of a pooling layer is similar to the just described propagation of a convolutional layer.

A pooling layer with input tensor size $X \times X \times F$ and output size $Y \times Y \times F$ has a filter with size $k$ and stride $s$.

As pooling is a spatial operation that does not cross channels, the importance of each channel can be propagated independently. The flattened importance score vector of channel $f$ $S_{out}^{f} \in \mathbb{R}^{1 \times (Y \times Y)}$ of the output tensor is given. With this importance vector the flattened importance score vector of the input is calculated as:

$$S_{in}^{f} = S_{out}^{f} \cdot BP_{pooling},$$
where $\mathbf{BP}_{\text{pooling}}$ is the $(Y \times Y) \times (X \times X)$ back-propagation matrix defined as:

$$\mathbf{BP}_{\text{pooling}} = \begin{pmatrix} b_p & \cdots & b_p \\ b_p & \cdots & b_p \\ \vdots \\ b_p & \cdots & b_p \end{pmatrix},$$

where $b_p$ is defined as the following building block with size $Y \times X$.

$$b_p = \begin{pmatrix} 1 & \cdots & 1 \\ 1 & \cdots & 1 \\ \vdots \\ 1 & \cdots & 1 \end{pmatrix}.$$

The propagation matrix $\mathbf{BP}_{\text{pooling}} \in \mathbb{R}^{(Y \times Y) \times (X \times X)}$ is used to map the given importance scores $\mathbf{S}_{\text{out}}^{f} \in \mathbb{R}^{1 \times (Y \times Y)}$ to the importance scores of the input layer $\mathbf{S}_{\text{in}}^{f} \in \mathbb{R}^{1 \times (X \times X)}$.

$\mathbf{BP}_{\text{pooling}}(i, j) \neq 0$ implies that the $i$-th position in the output layer comes from a pooling operation and involves the $j$-th position in the input layer. The importance scores are propagated between these two positions. The matrix $b_p \in \mathbb{R}^{(Y \times X)}$ is used to represent the mapping between a row in the output layer and the corresponding row in the input layer. Each row of $b_p$ contains $k$ 1’s, because each element in the output layer is pooled from a region with width $k$ of the input layer. The offset from the preceding block of ones and thus each offset from the start of the matrix is an integer multiple of the stride $s$.

Altogether the propagation matrix $\mathbf{BP}_{\text{pooling}}$ is a block matrix, in which each submatrix of size $Y \times X$ is either $b_p$ or a zero matrix. Each row in $\mathbf{BP}_{\text{pooling}}$ has $k$ $b_p$’s, as the height of the pooling filter is $k$. The beginning of the $k$ $b_p$’s have an offset of an integer multiple of the stride $s$. The ones in $b_p$ will be normalized by the number of positions covered by a pooling filter and all the other elements are zero.

Just as for the propagation of the convolutional layer a simple example is given to illustrate how this algorithm works. The example is shown in [8] in Figure 8.
5.3 Experiments

In the following experiments different networks will be pruned using the neuron importance score propagation technique. This technique will be applied to the same network structures that were also used in the experiments in the previous chapter. First NISP will be implemented on the simplest network architecture which is the fully connected neural network.

5.3.1 NISP on a Fully Connected Neural Network

In this section the NISP method is applied to a fully connected neural network. The same fully connected network that was also used in the previous chapter in Section 4.2.4. This network has two fully connected layers. The first fully connected layer connects the input layer containing 784 neurons with the hidden layer, which has 30 neurons. The activation function of this layer is the rectified linear unit function. The second fully connected layer connects the 30 neurons of the hidden layer with 10 neurons, which classify the ten handwritten digits of the MNIST dataset. All the parameters in this network add up to 23,860.

MNIST was trained on this network for 20 epochs and reached an accuracy of 96.47%.

The input layer has 784 neurons, one for every pixel in the 28 × 28 image. Therefore no neurons can be removed in this layer as the inputs are necessarily there. However the importance scores can still be calculated to see which input pixels are most relevant for classification.

The importance scores of the input layer are calculated by first applying a feature ranking algorithm on the "final response layer" (FRL). The FRL in this case is the hidden layer containing 30 neurons. Then these importance scores are propagated to the input layer by multiplying them with the corresponding weight matrix as described in Section 5.2.1.

In the following image the importance scores of the input layer are visualized with a grayscale heatmap on the 28×28 grid. In this heatmap lighter grayscale values correspond to higher importance scores.
CHAPTER 5. NEURON IMPORTANCE SCORE PROPAGATION

Figure 5.1: grayscale heatmap of the importance scores of the input layer reshaped to the size of the image

This heatmap shows that the pixels in the center of the input are most important for classifying images of the MNIST dataset, as these pixels have the lightest grayscale values.

The pixels on the edges of the heatmap are black which means these importance scores have the smallest values.

The values of these importance scores become clear when looking at images of the MNIST dataset which are displayed in figure 4.2. The actual digit to be classified is in the center of the image and therefore these pixels are most relevant for classification.

Neurons with least importance can not be removed in the input layer, but that does not mean they have to be used. Setting the unimportant neurons to zero would simply mean that those neurons have no connections to the neurons in the hidden layer and thus are ignored.

However in the following experiments only neurons that can actually be removed are pruned.

The neurons in the output layer also can not be pruned as the 10 neurons are needed to classify the handwritten digits. This means only neurons in the hidden layer can be removed. The importance scores of each of the 30 neurons are calculated by applying a
feature ranking algorithm on this layer. The neurons are then pruned according to this ranking.

The following table displays the accuracy and number of parameters for different pruning ratios. The pruning ratio here defines the ratio of kept neurons. For example a pruning ratio of $\frac{3}{4}$ means, $\frac{3}{4}$ of the neurons are kept and $\frac{1}{4}$ are removed.

<table>
<thead>
<tr>
<th>pruning ratio</th>
<th>number of kept neurons</th>
<th>accuracy after fine-tune</th>
<th>difference of parameters</th>
<th>number of parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\frac{3}{4}$</td>
<td>22</td>
<td>96.09%</td>
<td>6,360</td>
<td>17,500</td>
</tr>
<tr>
<td>$\frac{1}{2}$</td>
<td>15</td>
<td>95.49%</td>
<td>11,925</td>
<td>11,935</td>
</tr>
<tr>
<td>$\frac{1}{3}$</td>
<td>10</td>
<td>93.62%</td>
<td>15,900</td>
<td>7,960</td>
</tr>
<tr>
<td>$\frac{1}{4}$</td>
<td>7</td>
<td>92.34%</td>
<td>18,285</td>
<td>5,575</td>
</tr>
<tr>
<td>$\frac{1}{8}$</td>
<td>3</td>
<td>80.68%</td>
<td>21,465</td>
<td>2,395</td>
</tr>
<tr>
<td>$\frac{1}{16}$</td>
<td>1</td>
<td>36.44%</td>
<td>23,055</td>
<td>805</td>
</tr>
</tbody>
</table>

Table 5.1: NISP applied on fully connected network with different pruning ratios

The column *number of kept neurons* are the number of neurons that are not pruned in the hidden layer. To calculate the number of kept neurons, I multiplied the pruning ratio with the 30 neurons in the hidden layer and then rounded down to the next integer number.

The accuracies in the column *accuracy after fine-tune* correspond to the accuracies of the pruned models after fine-tuning for 20 epochs.

With a pruning ratio of $\frac{3}{4}$ the network is reduced by 6,360 parameters, but still almost reaches the initial accuracy of 96.47%.

By pruning almost 50% of the parameters the network only has 11,935 parameters, but still achieves an accuracy of 95.49%, which is only 1 percentage point worse than the accuracy of the original model.

Using a pruning ratio of $\frac{1}{3}$ the hidden layer then has 10 neurons which coincides with the number of output neurons. This network still has an accuracy of 93.62%.

Even with a pruning ratio of $\frac{1}{8}$, which reduces the model by almost 90%, the digits can still be classified with an accuracy of 80.68%.

Just in the extreme case, in which the network only has 1 neuron in the hidden layer, the network only achieves an accuracy of 36.44%. However the default rate for predicting a
fixed digit is 10%, so 36.44% is not that bad, given that the classification has to be made along a single numeric dimension, namely the output of the one neuron left in the hidden layer.

To examine whether there are specific digits that are particularly affected by the reduction of neurons or if all digits are roughly equally affected, two confusion matrices are compared.

The first matrix shows the confusion matrix of the unpruned network and then the confusion matrix of the network pruned using a pruning ratio of $\frac{1}{3}$ is presented.

<table>
<thead>
<tr>
<th>Predicted Classes</th>
<th>True Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0  1  2  3  4  5  6  7  8  9</td>
</tr>
<tr>
<td>0</td>
<td>968 0 1 0 0 2 0 4 4 1</td>
</tr>
<tr>
<td>1</td>
<td>0 1,116 2 7 0 1 3 2 4 0</td>
</tr>
<tr>
<td>2</td>
<td>4 1 993 7 6 0 2 9 9 1</td>
</tr>
<tr>
<td>3</td>
<td>1 0 8 967 1 9 0 6 6 12</td>
</tr>
<tr>
<td>4</td>
<td>2 0 3 0 941 0 10 2 4 20</td>
</tr>
<tr>
<td>5</td>
<td>2 1 0 14 0 856 7 1 4 7</td>
</tr>
<tr>
<td>6</td>
<td>9 3 2 0 8 10 921 0 5 0</td>
</tr>
<tr>
<td>7</td>
<td>2 2 12 8 1 3 0 988 2 10</td>
</tr>
<tr>
<td>8</td>
<td>6 1 4 6 4 13 3 6 926 5</td>
</tr>
<tr>
<td>9</td>
<td>4 5 0 8 8 4 0 6 3 971</td>
</tr>
</tbody>
</table>

Table 5.2: Confusion matrix of unpruned network which classifies images of MNIST
These two confusion matrices show that especially the digits 2, 5 and 9 are affected by the reduction.

The digits 0, 1, 4 and 6 can be classified almost just as well with the pruned network as with the unpruned network.

As the MNIST dataset is so simple, the network probably contains a lot of redundancy which would explain why so many parameters can be pruned without a significant accuracy loss.

For this reason random pruning will be compared with the NISP method to examine whether any random neurons can be pruned or if the importance scores are in fact decisive.

In the following 50% of the neurons in the hidden layer are randomly pruned.

The figure shows the accuracies during fine-tuning of the randomly pruned network compared to the network in which 50% of the neurons were pruned using the NISP method.
The blue graphs show the accuracies of the NISP method during fine-tuning. Each orange graph represents the accuracies of a randomly pruned network during fine-tuning.

There is a significant improvement in accuracy of the NISP method after epoch 12, because the learning rate was reduced. The learning rate, as already mentioned in Section 4.2.1, is reduced when the training loss does not drop for two epochs in a row.

In two of the four cases that are displayed in the graphics, random pruning is almost just as successful as the NISP method. In the top right and bottom right corner the accuracy of the randomly pruned network can not reach the same accuracy as when applying NISP.

As the neurons are randomly pruned, in some cases the most important ones could be removed. The more layers a network has, the more likely it is that important neurons are pruned.

Furthermore fine-tuning, which means further training, allows other neurons to take over and acquire the functionality that a pruned neuron had. Hence a faster adaption in the fine-tuning phase is also an advantage of the NISP method.

Therefore only pruning neurons with least importance is reasonable.

In the next section NISP will be applied to a network with LeNet structure to examine
how well the method is able to prune channels of convolutional layers.

5.3.2 NISP on LeNet

In this section a network with LeNet structure will be pruned using the NISP technique. The network is the same network that was also used in the previous chapter in Section 4.2.3. This network has two convolutional layers and two fully connected layers. The convolutional layers are each followed by a max pooling layer and a rectified linear unit function. The first fully connected layer connects 320 neurons with 50 neurons and the activation function is also a rectified linear unit function. The second fully connected layer connects the 50 neurons with 10 neurons for classification. Altogether the network has 21,840 parameters.

Training this network for 10 epochs on the MNIST dataset leads to an accuracy of 98.76%.

First the feature ranking algorithm is applied on the FRL. The FRL in this network is the layer containing 50 neurons. In this layer the first neurons are pruned according to the pruning ratio. The measured importance scores are then propagated further to the lower layers up to the first convolutional layer. In each layer neurons or channels with least importance scores are pruned depending on the pruning ratio. For every layer I used the same pruning ratio.

The following table shows the results of the NISP method for different pruning ratios.

<table>
<thead>
<tr>
<th>pruning ratio</th>
<th>accuracy after fine-tune</th>
<th>difference of parameters</th>
<th>number of parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\frac{3}{4}$</td>
<td>99.01%</td>
<td>9,721</td>
<td>12,119</td>
</tr>
<tr>
<td>$\frac{1}{2}$</td>
<td>98.53%</td>
<td>16,165</td>
<td>5,675</td>
</tr>
<tr>
<td>$\frac{1}{4}$</td>
<td>97.84%</td>
<td>20,431</td>
<td>1,409</td>
</tr>
<tr>
<td>$\frac{1}{8}$</td>
<td>92.96%</td>
<td>21,494</td>
<td>346</td>
</tr>
</tbody>
</table>

Table 5.4: NISP applied on LeNet with different pruning ratios

For each pruning ratio the NISP approach was applied to the described network and then fine-tuned for 20 epochs.

With a pruning ratio of $\frac{3}{4}$ or even $\frac{1}{2}$ the networks accuracy is hardly reduced. Also with a pruning ratio of $\frac{1}{4}$ the accuracy loss is negligible. Even though the number of parameters of this network is reduced by about 93.5%.
By applying NISP with a pruning ratio of $\frac{1}{8}$ the resulting network only has 346 parameters, but can still classify the MNIST dataset with 92.96% accuracy.

With this pruning ratio of $\frac{1}{8}$ the output of the first convolutional layer is reduced from 10 channels to 1 channel. The second convolutional layer of this reduced network takes that 1 channel and produces 2 output channels instead of the original 20 channels. The input neurons of the first fully connected layer drop from 320 to 32. These 32 neurons are then connected to 6 neurons instead of 50 neurons. The 10 neurons of the output of the second fully connected layer stay the same as these are needed for classifying the 10 handwritten digits.

This reduced network with 346 parameters only contains 1.58% of the parameters of the original network.

Even though this LeNet network is already fairly small compared to most standard networks, a lot of parameters can still be pruned with negligible accuracy loss.

Next the NISP method will we applied on ResNet. ResNet is a much larger and more complex network than the two previous networks. So the following section will examine how well the NISP approach can prune parameters in this more complex model.

5.3.3 NISP on ResNet

To apply the NISP method on ResNet the same 18-layer ResNet architecture as in Section 4.2.2 will be used.

This network is a lot larger than the fully connected neural network and LeNet network. Therefore applying the NISP approach and fine-tuning the pruned model is computational more expensive. For this reason the results for multiple different pruning ratios are not compared.

The previous two networks could easily be pruned with a pruning ratio of $\frac{1}{2}$ without accuracy loss. LeNet could even be pruned using a pruning ratio of $\frac{1}{4}$.

Consequently I limited the comparison to a pruning ratio of $\frac{1}{2}$ and $\frac{1}{4}$.

Just as in the previous experiment the same pruning ratio is used throughout the entire network.

ResNet has a lot more parameters, so it will be interesting to see whether the same ratio...
as in the other two networks can be pruned.

Classifying images of Cifar-10

The ResNet network which classifies images of Cifar-10 has 11,171,146 parameters and reaches an accuracy of 92.08% after training for 70 epochs.

With a pruning ratio of $\frac{1}{2}$ the resulting network has 2,796,202 parameters, which is about 25% of the original network.

This pruned network achieves an accuracy of 91.01% after fine-tuning for 50 epochs. So by using only 25% of the original network the pruned network reaches almost the initial accuracy.

The following plot shows the accuracies of this reduced network during fine-tuning.

![Figure 5.3: accuracies [in %] during fine-tuning after applying NISP with pruning ratio $\frac{1}{2}$ on ResNet using Cifar-10 dataset](image)

This plot confirms the necessity of fine-tuning. After the first epoch the pruned network still only has an accuracy of about 73%. Continuing fine-tuning up until epoch 25 the accuracy slowly increases to about 91% and then stays constant for the rest of the fine-tuning process.

A pruning ratio of $\frac{1}{2}$ only led to a minimal accuracy loss. So next a pruning ratio of $\frac{1}{4}$ will be used to examine whether the network can be reduced even more.
Applying NISP on the 18-layer ResNet with a pruning ratio of $\frac{1}{4}$ leads to a reduction of about 93.7%. The resulting network only has 700,762 parameters.

Fine-tuning this pruned network for 50 epochs also reaches an accuracy of 89.98% which is not much less than the initial accuracy of 92.08%.

The next graphic shows the accuracies during the fine-tuning process.

![Figure 5.4: accuracies [in %] during fine-tuning after applying NISP with pruning ratio $\frac{1}{4}$ on ResNet using Cifar-10 dataset](image)

As more of the network was pruned more fine-tuning epochs are needed. With a pruning ratio of $\frac{1}{2}$ the network could reach an accuracy of about 91% after 25 epochs. In this case at least 38 epochs are required to achieve an accuracy of almost 90%.

To see whether the degradation is roughly uniform across all classes two confusion matrices are presented.

The confusion matrix of the unpruned network is compared with the confusion matrix of the network pruned with a pruning ratio of $\frac{1}{4}$. 

49
### Table 5.5: Confusion matrix of unpruned network which classifies images of Cifar-10

<table>
<thead>
<tr>
<th>Predicted Classes</th>
<th>True Classes</th>
<th>airplane</th>
<th>automobile</th>
<th>bird</th>
<th>cat</th>
<th>deer</th>
<th>dog</th>
<th>frog</th>
<th>horse</th>
<th>ship</th>
<th>truck</th>
</tr>
</thead>
<tbody>
<tr>
<td>airplane</td>
<td>930</td>
<td>2</td>
<td>21</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>28</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>automobile</td>
<td>6</td>
<td>954</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>7</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>bird</td>
<td>22</td>
<td>0</td>
<td>891</td>
<td>24</td>
<td>24</td>
<td>16</td>
<td>16</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>cat</td>
<td>7</td>
<td>2</td>
<td>28</td>
<td>840</td>
<td>15</td>
<td>78</td>
<td>15</td>
<td>7</td>
<td>5</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>deer</td>
<td>4</td>
<td>0</td>
<td>16</td>
<td>10</td>
<td>945</td>
<td>11</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>dog</td>
<td>5</td>
<td>0</td>
<td>12</td>
<td>87</td>
<td>17</td>
<td>861</td>
<td>4</td>
<td>11</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>frog</td>
<td>3</td>
<td>2</td>
<td>20</td>
<td>21</td>
<td>10</td>
<td>4</td>
<td>937</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>horse</td>
<td>6</td>
<td>0</td>
<td>9</td>
<td>10</td>
<td>22</td>
<td>23</td>
<td>1</td>
<td>928</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>ship</td>
<td>18</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>962</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>truck</td>
<td>9</td>
<td>14</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>10</td>
<td>960</td>
<td></td>
</tr>
</tbody>
</table>

The classes 'bird', 'cat', 'deer', 'ship' and 'truck' are affected most by the reduction. However some classes can even be classified better using the pruned network. The categories 'automobile', 'dog' and 'horse' achieve higher classification accuracies using the pruned network.
Classifying images of Cifar-10 is a more difficult classification task than classifying images of MNIST. Therefore pruning the network trained on MNIST with a pruning ratio of $\frac{1}{2}$ and $\frac{1}{4}$ should also be possible with negligible accuracy loss.

**Classifying images of MNIST**

The 18-layer ResNet used to classify images of the MNIST dataset has 11,169,994 parameters. Training this network for 20 epochs achieves an accuracy of 99.39%.

By applying the NISP method with a pruning ratio of $\frac{1}{2}$ on this network the model can be reduced by 8,374,368 parameters which is about 75% of the model.

After fine-tuning this network for 20 epochs an accuracy of 99.47% is reached which is also the accuracy of the original model.

The accuracies during fine-tuning are shown in the following image.

![Figure 5.5: accuracies [in %] during fine-tuning after applying NISP with pruning ratio $\frac{1}{2}$ on ResNet using MNIST dataset](image)

Already after the first epoch the pruned model reaches an accuracy of almost 99%. In comparison pruning the network which classifies images of the Cifar-10 dataset, needed at least 25 epochs to reach an accuracy close to the initial accuracy.

Also with a pruning ratio of $\frac{1}{4}$ this network can achieve an accuracy of 99.46% after fine-tuning for 20 epochs. The pruned network has 700,474 parameters.

The following plot, just like the previous plot, shows the accuracies during fine-tuning.
The pruned network which only uses 93.7% of the parameters of the original network also reaches an accuracy of over 98% after the first epoch.

However these results are not surprising as in the previous Section 5.3.2 images of MNIST could be classified with almost 99% accuracy with a network only containing 5,675 parameters.

5.4 Conclusion

Neuron importance score propagation is a pruning technique which prunes neurons or complete channels after the network has been trained.

The importance scores are first calculated for the "final response layer" which is the second-to-last layer before classification and therefore the direct inputs of the classification task. These importance scores are computed by applying a feature ranking algorithm. Then based on these computed importance scores the importance scores of the entire network are measured. Starting with the FRL the scores are propagated to all the earlier layers in the network. In each layer the neurons or channels with least importance are pruned according to the pre-defined pruning ratio.

The pruned network is then fine-tuned to regain its accuracy.

The performed experiments proved the effectiveness of this approach.
The fully connected network used to classify MNIST images could be pruned by about 50% with negligible accuracy loss.

The network with LeNet-style architecture which also classifies images of the MNIST dataset could even be reduced by 93.5% with only small accuracy loss.

Even the larger and more complex model for classifying images of Cifar-10 could be significantly reduced. Using only 6.3% of the parameters the network still classified the images with an accuracy of almost 90%. In comparison the original network has 10,470,384 parameters more and also only achieves a slightly better accuracy of 92.08%.

However the problem with this approach is choosing the pre-defined pruning ratio as it is hard to know in advance how many parameters can be pruned.

In the supplementary material of [8] a technique for selecting the pruning ratio is analyzed. The presented method is called PCA accumulated energy analysis.

Furthermore [8] also examines whether different pruning ratios should be used for convolutional and fully connected layers.

Up until now, we have looked at an approach which prunes weights and an approach which prunes neurons. In the next chapter, we will look at a pruning method which prunes complete layers. In comparison to the previous two approaches which were pruned after training, this next approach is applied during the training process.
In this chapter a pruning technique named $\varepsilon$-ResNet will be analyzed. This technique was presented in the paper [4] on which this chapter is also based on.

$\varepsilon$-ResNet is a pruning method that prunes complete layers during the training process. This approach is called $\varepsilon$-ResNet as the method is a continuation of the original residual idea. By adding some additional rectified linear units in the original ResNet, redundant layers, which produce responses smaller than a threshold $\varepsilon$, are removed.

As in the previous two chapters, we will first look into the theory and then apply this technique in some practical experiments.

### 6.1 Theory

$\varepsilon$-ResNet is a pruning technique built on the original residual idea, which is characterized in Chapter 2 in Section 2.3.

Residual networks contain 'shortcut connections' which let the stacked layers of the network fit a residual mapping $F(x) := H(x) - x$ instead of the underlying mapping $H(x)$. These shortcut connections perform identity mappings which allow the network to skip one or more layers.

The following image shows the building block of such a residual mapping.
A residual block consists of two batch normalization layers, two ReLU functions and two convolutional layers. The left block displays the original structure of a building block used in [12]. In [13] ReLU and Batch Normalization are viewed as "pre-activation" of the weight layers which is demonstrated in the right block of the image. Residual blocks consisting of pre-activations of the weight layers have a pure addition on the left path. With this pure addition the forward and backward signals can be directly propagated from one building block to another building block. This direct propagation leads to faster training and better generalization.

The entire residual network is constructed by stacking these residual blocks.

As already mentioned in Section 2.3 the advantage of the shortcut connections is faster optimization. In the extreme case that the layers of a building block are not needed in the network, that means the identity mapping would be optimal, it is easier to optimize a network that pushes the residual to zero instead of fitting an identity mapping by a stack of nonlinear layers. [12]

However in experiments the residuals hardly ever go to perfect zeros. Therefore [4] proposes a technique which promotes identity mapping in a strict sense.

By adding a few additional rectified linear unit functions, residual blocks that produce responses below a threshold $\varepsilon$ are pushed to zero.

The following figure demonstrates how the strict identity mapping is achieved.
Figure 6.2: (a) shows the standard ResNet mapping given by $H(x) = F(x) + x$, which is described in 4.2.2. (b) shows the $\varepsilon$-ResNet mapping given by $H(x) = S(F(x)) + x$. If all of the residual responses in $F(x)$ are less than a threshold $\varepsilon$, then $S(F(x)) = 0$. If at least one of the responses is not smaller, then the same mapping $S(F(x)) = F(x)$ as in the standard network is done.

Figure (a) shows the initial residual mapping $H(x) := F(x) + x$. In the right graphic (b) the residual mapping of $\varepsilon$-ResNet is illustrated. A function $S$ is included which modifies the mapping to $H(x) = S(F(x)) + x$.

If all of the responses in the original block $F(x)$ are smaller than a threshold $\varepsilon$, then $S(F(x)) = 0$. However if at least one response is larger than $\varepsilon$, then the original mapping $S(F(x)) = F(x)$ of the standard ResNet is applied.

These strict identity mappings are produced during the training process. After the network has been trained all residual blocks whose responses are less than $\varepsilon$ can be removed. That means this pruning method prunes complete residual blocks consisting of two batch normalization layers, two ReLU functions and two convolutional layers at once. For predictions the reduced network is applied.

The extra rectified linear unit functions do not need any additional parameters. Therefore the number of parameters in $\varepsilon$-ResNet and ResNet are the same for identical network structures.

The next figure explains exactly how the included function $S$ is able to promote these
strict identity mappings.

**Figure 6.3:** (a) shows the network structure of one of the residual blocks in the standard pre-activation network [13] described in 4.2.2. (b) shows the network structure of one of the residual blocks in \( \varepsilon \)-ResNet. A sparsity-promoting function \( S() \) is added to discard the residual if all the individual responses are less than \( \varepsilon \). (c) shows the network structure for the sparsity-promoting function \( S() \) using four ReLU’s and one multiplicative gate. The pair \((i, j)\) denote the weights \(i\) and bias term \(j\) for the associated ReLU function. \( L \) refers to a very large positive constant and \( \varepsilon \) denotes the constant which is used for discarding layers.

[4, Fig. 4]

The figures (a) and (b) in principle show the same residual mappings as already shown in Figure 6.2, but include the pre-activation building block.

Figure (c) illustrates the proposed network structure of this so called **sparsity-promoting function** \( S(\mathcal{F}(x)) \).

\( \varepsilon \)-ResNet uses the residual mapping

\[
\mathcal{H}(x) = S(\mathcal{F}(x)) + x,
\]
instead of the original residual mapping $\mathcal{H}(x) = \mathcal{F}(x) + x$.

Assuming $\mathcal{F}(x)$ is a vector of length $n$ and each element of the vector is denoted by $\mathcal{F}(x)_i$ for $i \in \{1, \ldots, n\}$, then the sparsity-promoting function $S(\mathcal{F}(x))$ is defined as follows.

$$S(\mathcal{F}(x)) = \begin{cases} 
0 & \text{if } |\mathcal{F}(x)_i| < \varepsilon, \forall i \in \{1, \ldots, n\} \\
\mathcal{F}(x) & \text{otherwise.}
\end{cases}$$

The described structure of four ReLUs approximates the function $S(\mathcal{F}(x))$.

To examine the behavior of the sparsity-promoting function two different cases will be viewed.

First, the case in which all the response elements are smaller than the threshold $\varepsilon$ is considered. That means $|\mathcal{F}(x)_i| < \varepsilon$ is fulfilled. As all the responses are below $\varepsilon$ the output from the summation (+) will be zero. This zero input to the third ReLU produces an output of 1 which then leads to an output of zero from the forth ReLU. Therefore $T(\mathcal{F}(x)) = 0$ and the following equation is obtained.

$$S(\mathcal{F}(x)) = T(\mathcal{F}(x)) \times \mathcal{F}(x) = 0$$

The other case describes the scenario where at least one response is larger than $\varepsilon$. This case occurs when $\mathcal{F}(x) > \varepsilon$ or $\mathcal{F}(x) < -\varepsilon$. If one of these conditions holds, the output of the first summation (+) will be a non-zero positive value. A non-zero positive input to the third ReLU causes an output of zero. Consequently the output of both the forth ReLU and $T(\mathcal{F}(x))$ will be 1. This affects the equation as follows.

$$S(\mathcal{F}(x)) = T(\mathcal{F}(x)) \times \mathcal{F}(x) = \mathcal{F}(x)$$

By adding these four additional rectified linear unit functions this method is able to prune complete layers during the training process.

To examine how well this technique can remove redundant layers experiments using dif-
ferent datasets are performed.

6.2 Experiments

This pruning technique is only applicable to networks that have the previously described residual structure.

Therefore all the experiments will be using the 18-layer pre-activation ResNet architecture characterized in 4.2.2.

This network starts with a convolutional layer which is then followed by eight building blocks with the structure of the pre-activation ResNet in 6.1 (right block). The network ends with an average pooling and a fully connected layer.

As the experiments are limited to this defined network architecture, three different datasets will be used to evaluate this pruning method.

First the network which classifies images of the MNIST dataset will be pruned. Then the pruning technique will be applied to the network for the Cifar-10 dataset. And at last a subset of the Cifar-100 dataset will be used.

6.2.1 Classifying images of MNIST

The network used to classify images of the MNIST dataset has 11,169,994 parameters and achieves an accuracy of 99.39% after training for 20 epochs.

The following table summarizes the results of this pruning method for various $\varepsilon$. Each network is trained for 20 epochs.

<table>
<thead>
<tr>
<th>$\varepsilon$</th>
<th>accuracy</th>
<th>difference of parameters</th>
<th>number of parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>99.39%</td>
<td>0</td>
<td>11,169,994</td>
</tr>
<tr>
<td>1</td>
<td>98.82%</td>
<td>4,794,624</td>
<td>6,375,370</td>
</tr>
<tr>
<td>1.5</td>
<td>99.01%</td>
<td>8,260,608</td>
<td>2,909,386</td>
</tr>
<tr>
<td>2</td>
<td>99.25%</td>
<td>10,621,952</td>
<td>548,042</td>
</tr>
<tr>
<td>2.5</td>
<td>47.5%</td>
<td>10,991,360</td>
<td>178,634</td>
</tr>
</tbody>
</table>

Table 6.1: Classifying MNIST with $\varepsilon$-ResNet for different $\varepsilon$
\\( \varepsilon = 0 \) corresponds to the initial pre-activation ResNet.

Up to \( \varepsilon = 2 \) the networks could reach the same accuracy as the initial pre-activation ResNet.

A threshold \( \varepsilon = 2 \) means that all layers with responses less than 2 can be removed. This network could be pruned by 95.1\% without accuracy loss. With this threshold only the first three of the eight building blocks are required. So the 18-layer network could be reduced to an 8-layer network without losing accuracy.

Using \( \varepsilon = 2.5 \) all of the eight building blocks are removed, but this network also only achieves an accuracy of 47.5\% after training for 20 epochs.

The experiments with the other pruning methods in the previous chapters already demonstrated that classifying images of the MNIST dataset is a simple classification task which can be accomplished using small networks. Therefore larger networks such as ResNet contain a lot of redundancy which can easily be pruned without decreasing accuracy.

In the next example the 18-layer ResNet used to classify images of Cifar-10 will be pruned. The images of the Cifar-10 dataset are more complex than the MNIST dataset. Therefore the network probably has less redundancy that can be removed.

### 6.2.2 Classifying images of Cifar-10

To classify images of Cifar-10 the 18-layer ResNet requires 11,171,146 parameters. This network reaches an accuracy of 92.08\% while training for 70 epochs.

For this network the same thresholds \( \varepsilon \), as in the previous experiments, are applied.

The results are presented in the table below. For each \( \varepsilon \) the network is trained for 70 epochs.
### Table 6.2: Classifying Cifar-10 with ε-ResNet for different ε

<table>
<thead>
<tr>
<th>ε</th>
<th>accuracy</th>
<th>number of parameters</th>
<th>difference of parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>92.08%</td>
<td>11,171,146</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>91.56%</td>
<td>2,910,538</td>
<td>8,260,608</td>
</tr>
<tr>
<td>1.5</td>
<td>91.26%</td>
<td>6,450,506</td>
<td>4,720,640</td>
</tr>
<tr>
<td>2</td>
<td>89.55%</td>
<td>2,541,130</td>
<td>8,630,016</td>
</tr>
<tr>
<td>2.5</td>
<td>51.31%</td>
<td>179,786</td>
<td>10,991,360</td>
</tr>
</tbody>
</table>

Even though the classification task of Cifar-10 is more difficult than of MNIST, the network could still be pruned with negligible accuracy loss using a threshold ε up to a value of 2.

With ε = 2 only the forth, fifth and sixth building block are required to still reach an accuracy of 89.55%. This pruned network merely uses 22.7% of the initial parameters.

Using ε = 1 achieves almost the same reduction. This network can be pruned by 73.9% and can still classify the images with an accuracy of 91.26%. In this case the last two of the eight building blocks are removed.

Interestingly the network pruned using ε = 1.5, has more parameters than the network with the threshold ε = 1. With ε = 1 all building blocks with responses less than 1 are pruned and correspondingly with ε = 1.5 all building blocks with responses less than 1.5 are pruned. Therefore one would expect that ε = 1.5 leads to a higher reduction.

However the outputs which are produced by the residual blocks depend on various stochastic elements, like the initial weights and the order and mini-batch arrangement of the training examples. This can lead to a residual block being switched off with a smaller value for ε that is maintained with a larger value.

Consequently this described occurrence, that ε = 1.5 achieves less reduction than ε = 1, is possible.

As also in the network which classifies images of MNIST, all the responses of the building blocks are less than 2.5. So if all of these building blocks are removed the resulting network only consists of one convolutional layer, one average pooling layer and the fully connected layer. As a result this extremely pruned network can only achieve an accuracy of 51.31% after training for 70 epochs. However compared to the baseline accuracy of 10%, this
result is still not that bad.

In the final experiment this pruning technique is examined using this 18-layer network that was trained to classify a subset of Cifar-100.

6.2.3 Classifying images of a subset of Cifar-100

The Cifar-100 dataset consists of 100 different classes. Training a network to classify images of this dataset is quite time consuming. For this reason only a subset containing 10 classes is used.

As already mentioned in Section 4.2.1 the classes of Cifar-100 are divided into 20 superclasses, each containing 5 classes. The 5 classes belonging to a superclass are similar. Therefore the subset was chosen by randomly selecting 10 classes each belonging to a different superclass.

Following this procedure the randomly selected subset contains these classes: baby, bear, squirrel, poppy, plate, keyboard, wardrobe, trout, caterpillar and streetcar.

To train this subset the exact network with the already trained parameters which classifies images of Cifar-10 is used. The parameters that were trained for 70 epochs are frozen and only the parameters of the fully connected layer are updated during the fine-tuning process of another 30 epochs. After that the parameters of the entire network are fine-tuned for an additional 30 epochs. Altogether the network was then trained for 130 epochs.

This exact training process is applied for the same values of $\varepsilon$ as also in the previous two sections.

A summary of the results is shown in the following table.

<table>
<thead>
<tr>
<th>$\varepsilon$</th>
<th>accuracy</th>
<th>difference of parameters</th>
<th>number of parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>88.1%</td>
<td>0</td>
<td>11,171,146</td>
</tr>
<tr>
<td>1</td>
<td>87.6%</td>
<td>8,260,608</td>
<td>2,910,538</td>
</tr>
<tr>
<td>1.5</td>
<td>87.3%</td>
<td>4,720,640</td>
<td>6,450,506</td>
</tr>
<tr>
<td>2</td>
<td>85.4%</td>
<td>8,630,016</td>
<td>2,541,130</td>
</tr>
<tr>
<td>2.5</td>
<td>58.6%</td>
<td>10,991,360</td>
<td>179,786</td>
</tr>
</tbody>
</table>

Table 6.3: Classifying a subset of Cifar-100 with $\varepsilon$-ResNet for different $\varepsilon$
For $\varepsilon = 0$, which corresponds to the original ResNet, the network reaches an accuracy of 88.1%.

The networks with the different $\varepsilon$ achieve similar pruning results as in the experiments with the Cifar-10 dataset. This is reasonable, because these networks started out with the same parameter values.

Therefore the network with $\varepsilon = 1$ is also able to prune more parameters than the network with $\varepsilon = 1.5$.

Like also in the previous experiments, $\varepsilon = 1$ and $\varepsilon = 2$ achieve the highest reduction with only minimal accuracy loss.

However in this case the trade-off between reduction and accuracy is best using $\varepsilon = 1$. This network prunes 73.9% of the original model and still only has negligible accuracy loss. To obtain this reduction also the last two of the eight building blocks were discarded.

### 6.3 Conclusion

The pruning technique examined in this chapter is based on the original residual network idea.

Adding a few additional rectified linear unit functions, all layers with responses below the threshold $\varepsilon$ are set to zero. These layers are redundant and can be discarded without any loss in performance. Hence this method is called $\varepsilon$-ResNet.

This is a pruning approach that is applied during the training process and therefore fine-tuning is not required. Moreover no further parameters are needed as only rectified linear unit functions are added.

Using this approach the 18-layer ResNet could be significantly reduced.

The ResNet which classifies images of MNIST could be pruned the most. Applying this technique the network could be reduced from 11,169,994 parameters to merely 548,042 parameters while still reaching the initial accuracy.

This method also achieved successful pruning results on the 18-layer network used to classify Cifar-10 images. With a threshold of $\varepsilon = 2$ the 11,171,146 parameters could be pruned by 77.3%. With 2,541,130 parameters the resulting network can still classify the images with an accuracy of 89.55%.
To train a network on a subset of Cifar-100 the already trained parameters of the network trained on Cifar-10 were used. Therefore the pruning results are similar.

This network could also be pruned with a threshold of $\varepsilon = 2$. Only using 2,541,130 parameters, the network reached an accuracy of 85.4% which is a little bit less than the initial accuracy of 88.1%.

If a threshold of $\varepsilon = 1$ is used, the network only has a few more parameters than when using $\varepsilon = 2$, but achieves a slightly better accuracy of 87.6%.

Altogether $\varepsilon$-ResNet significantly reduced all three networks with minimal accuracy loss. However the value of $\varepsilon$ must be determined in advance. Generally the higher $\varepsilon$ is, the more layers can be pruned, but if too many layers are removed the accuracy decreases. In the performed experiments $\varepsilon = 2$ seems to be a favorable threshold for all three datasets.

Although for a network with a more complex classification task, a smaller threshold $\varepsilon$ could be reasonable. As the network then probably contains less redundancy that can be pruned.

For the network that classifies images of a subset of Cifar-100 $\varepsilon = 1$ led to a slightly smaller reduction, but the accuracy was higher than with a threshold $\varepsilon = 2$. In this case $\varepsilon = 1$ would probably be the better threshold.

After a detailed analysis of three different pruning approaches, the next chapter examines whether one approach is more successful than another.
7 Comparison of the Three Analyzed Approaches

In the previous chapters three different approaches to reduce the number of parameters in deep neural networks were analyzed. In this chapter we will examine whether one of the approaches performs better than another.

This is difficult to evaluate, as the methods are not directly comparable.

Singular value decomposition which was introduced in Chapter 4 is only applicable to fully connected layers. However most commonly used networks for image classification tasks are convolutional neural networks, as they can process image data better than fully connected networks. Therefore this technique is not very successful on network structures such as ResNet. The described 18-layer ResNet only has one fully connected layer which merely accounts for about 0.046% of the parameters.

Nonetheless on network architectures with more fully connected layers SVD is able to reduce the number of parameters significantly.

Applying SVD on the presented fully connected neural network the model could be reduced by 40.57% with negligible accuracy loss.

Moreover the network with LeNet-style architecture could be reduced by 47.85% using SVD. Even though this neural network has two convolutional layer and only two fully connected layers.

The reduced model still reached an accuracy of 98.3% which is only slightly less than the initial accuracy of 98.76%.

The technique presented in Chapter 5 called neuron importance score propagation not only prunes neurons in fully connected layers, but also entire channels in convolutional and pooling layers. This approach achieves better results than SVD, however this technique
requires fine-tuning. Therefore the training process is longer which usually results in higher accuracies.

The NISP method can prune the fully connected neural network by 49.98%, but only obtains an accuracy of 95.49% after fine-tuning for another 20 epochs. 95.49% is almost the initial accuracy of the original model.

Furthermore the LeNet-style network can even be reduced by 93.5% using NISP. The pruned network has only 1,409 parameters and reaches an accuracy of 97.84% after fine-tuning for 20 epochs.

Since this pruning method is also able to prune convolutional layers, NISP can considerably reduce the number of parameters in the 18-layer ResNet.

The 18-layer ResNet used to classify images of the Cifar-10 dataset can be reduced from 11,171,146 parameters to 700,762 parameters with only minimal accuracy loss. This model still achieves an accuracy of 89.98% which is only about 2 percentage points less than the original accuracy. However this result is only possible if the pruned network is fine-tuned for 50 epochs.

In comparison, the 18-layer ResNet can only be reduced by 8,630,016 parameters using the approach which was analyzed in Chapter 6. This approach is called ε-ResNet and prunes complete layers, but the structure of ResNet is required for this pruning method. ε-ResNet automatically removes redundant layers during the training process by merely adding a few additional rectified linear unit functions to the original ResNet structure.

Therefore this method accomplishes the reduction of 8,630,016 parameters by only training for the initial 70 epochs. The pruned model still reaches an accuracy of 89.55%.

ε-ResNet may not be able to prune as many parameters as NISP, but does not require fine-tuning and so the training process is shorter.

NISP becomes cumbersome for larger networks, as the importance scores have to be calculated for every layer. Moreover larger networks also have a more time consuming fine-tuning process.

To sum up, all three pruning techniques can successfully reduce the number of parameters in neural networks while maintaining almost the initial accuracy. However SVD only performs well on networks that mainly consist of fully connected layers and ε-ResNet can just be applied to networks with ResNet structure.
NISP achieves significant reduction on any type of network architecture.

Among the three analyzed approaches, the NISP method is able to reduce the described networks best, but this method also has the longest training process.

In the next chapter these presented pruning approaches will be combined to analyze whether one approach already removes all of the redundancy in the network or if the combination of two techniques achieves even better pruning results.
8 Combined Approaches

In the previous chapters three different techniques to reduce the number of parameters in neural networks were analyzed in detail. The experiments to each method confirmed that the networks contain a lot of redundancy and can be pruned with negligible accuracy loss.

This chapter examines whether one of the methods already removes all of the redundancy or if better pruning results can be achieved by combining two techniques.

In the following experiments the approach using singular value decomposition will be combined with the neuron importance score propagation technique. Further the neuron importance score propagation approach will also be combined with $\varepsilon$-ResNet.

The combination of $\varepsilon$-ResNet and SVD will not be performed, as $\varepsilon$-ResNet is only applicable to networks with a residual structure. The experiments in Chapter 4 already showed that SVD is not able to prune networks with such an architecture.

8.1 Neuron Importance Score Propagation and Singular Value Decomposition

SVD is only able to prune parameters of fully connected layers. Therefore SVD does not prune every possible parameter. Especially networks with convolutional layers probably still contain redundancy that can be removed if a further technique is applied.

Consequently applying NISP after the network was pruned using SVD will certainly lead to a further reduction.

For this reason the approaches will be applied in reverse order. That means the following networks will first be pruned with the NISP technique and then SVD will be applied to examine whether even more of the network can be reduced.
To start with NISP and SVD will be applied to a fully connected neural network. Then a network with LeNet-style will be pruned using both techniques.

### 8.1.1 NISP and SVD on a Fully Connected Neural Network

In Section 5.3.1 the fully connected network consisting of two fully connected layers was successfully pruned using NISP.

The original network is described in Section 4.2.4. The first fully connected layer connects the 784 input neurons with the 30 neurons in the hidden layer. These 30 neurons are then connected with the 10 output neurons for classifying images of the MNIST dataset.

Training this network for 20 epochs on the MNIST dataset leads to an accuracy of 96.47%. With a pruning ratio of $\frac{1}{2}$ the 30 neurons in the hidden layer are pruned to 15 neurons. In this way the 23,860 parameters are reduced to 11,935 parameters. This pruned model still reaches an accuracy of 95.49%.

In the following SVD is only applied to the first fully connected layer, as the experiments in 5.3.1 showed that just this layer can be pruned with negligible accuracy loss.

The weight matrix of the first fully connected layer has a size of $15 \times 784$. So at most 14 singular values can be set to zero.

The following plot shows the accuracies of the pruned model after SVD is applied for various k’s. k denotes the number of the k largest singular values that are not set to zero.

![Figure 8.1](image)

**Figure 8.1:** accuracies [in %] after applying SVD for different k’s on the first fully connected layer of fully connected network that was already pruned using NISP
Even for $k = 14$ the accuracy drops from 95.49% to about 90%. This implies that NISP already removed a lot of the redundancy in this network. If there still is redundancy, then it is not of the form that can be removed by SVD. Therefore combining these two approaches does not achieve better results.

8.1.2 NISP and SVD on LeNet

The network with LeNet-style structure consists of two convolutional layers and two fully connected layers. The detailed architecture is described in Chapter 4 in Section 4.2.3.

After training this network for 10 epochs, images of the MNIST dataset can be classified with an accuracy of 98.76%. In Section 5.3.2 NISP was applied to this network with different pruning ratios. With a pruning ratio of $\frac{1}{4}$ the best reduction with minimal accuracy loss was achieved.

Using this pruning ratio the 21,840 parameters can be reduced to merely 1,409 parameters while still reaching an accuracy of 97.84%.

To evaluate whether even more parameters can be pruned, SVD is applied to the first fully connected layer.

The first fully connected layer of the pruned model connects 80 neurons with 12 neurons. So at most 11 singular values can be set to zero.

SVD is applied for different $k$'s. The accuracies of the pruned models are displayed in the following plot.

![Figure 8.2: accuracies [in %] after applying SVD for different k's on the first fully connected layer of LeNet-style network that was already pruned using NISP](image-url)
CHAPTER 8. COMBINED APPROACHES

Only for $k = 11$ the network can obtain its accuracy. However by decomposing the $12 \times 80$ weight matrix into two matrices with sizes $11 \times 80$ and $12 \times 11$ the number of parameters increases by 52. Just for $k < 11$ the number of parameters are reduced, but already for $k = 10$ the accuracy decreases significantly. Therefore also in this network SVD can not prune any further parameters.

Applying SVD to the second fully connected layer is not necessary, as the experiments in Section 4.2.3 already confirmed that no further reduction can be accomplished without losing accuracy.

These examples show that NISP already prunes all the redundancy in the examined networks. By combining NISP with SVD no further reduction can be achieved.

Next we will analyze whether the combination of the approaches NISP and $\varepsilon$-ResNet reaches better results as one of the approaches alone.

### 8.2 Neuron Importance Score Propagation and $\varepsilon$-ResNet

$\varepsilon$-ResNet is a pruning technique which is applied during the training process. For this reason the networks will first be pruned using $\varepsilon$-ResNet and then NISP will be applied to examine whether more parameters can be removed.

#### 8.2.1 Classifying images of MNIST

The 18-layer ResNet characterized in Section 4.2.2 has 11,169,994 and reaches an accuracy of 99.39% after training MNIST for 20 epochs.

With $\varepsilon = 2$ the building blocks four, five, six, seven and eight are identified as redundant and can be removed. This pruned network only has 548,042 parameters and achieves an accuracy of 99.25%.

Next NISP is applied with a pruning ratio of $\frac{1}{2}$.

After the second pruning method is applied the network merely has 138,858 parameters, but still can classify images of MNIST with an accuracy of 99.32%.

However this reduced network was fine-tuned for another 20 epochs.
The following image presents the accuracies during the fine-tuning process of the pruned network.

**Figure 8.3:** accuracies [in %] during fine-tuning of the model pruned with \(\varepsilon\)-ResNet and NISP using a pruning ratio of \(\frac{1}{2}\) to classify MNIST

The plot shows that not so many epochs were necessary to retain the initial accuracy. Already after the fifth epoch an accuracy of about 99.25% was achieved.

Therefore we can conclude that \(\varepsilon\)-ResNet does not remove all of the redundancy in this network. NISP with a pruning ratio of \(\frac{1}{2}\) could easily prune more parameters with no accuracy loss.

Probably even more parameters can be reduced. For this reason NISP with a pruning ratio of \(\frac{1}{4}\) will be applied to the network pruned with \(\varepsilon = 2\).

With only 35,642 parameters this network still reaches an accuracy of 99.25% after fine-tuning for 20 epochs.

The next image shows the accuracies of the network during fine-tuning.
CHAPTER 8. COMBINED APPROACHES

Figure 8.4: accuracies [in %] during fine-tuning of the model pruned with $\varepsilon$-ResNet and NISP using a pruning ratio of $\frac{1}{4}$ to classify MNIST

With a pruning ratio of $\frac{1}{4}$ more fine-tuning epochs are required compared to the network with a pruning ratio of $\frac{1}{2}$. After epoch 16 the accuracy stays stable at a value of about 99.25%.

Nonetheless even with a pruning ratio of $\frac{1}{4}$ NISP can reduce the already pruned network with no accuracy loss.

$\varepsilon$-ResNet can reduce the number of parameters significantly, but clearly this method does not remove all of the redundancy.

Next the 18-layer ResNet which classifies images of Cifar-10 will be pruned using both $\varepsilon$-ResNet and NISP.

8.2.2 Classifying images of Cifar-10

The 18-layer ResNet which classifies images of Cifar-10 can also be successfully pruned with $\varepsilon$-ResNet.

With $\varepsilon = 2$ the building blocks one, two, three, seven and eight can be removed with small accuracy loss. The resulting network has 2,541,130 parameters and achieves an accuracy of 89.55% after a training process of 70 epochs.

In comparison the original network has 11,171,146 parameters and classifies the images with an accuracy of 92.08%.
As in the previous experiment, first a pruning ratio of $\frac{1}{2}$ is applied to the already reduced network.

After the network is reduced with the second pruning technique the model only has 637,738 parameters. Fine-tuning this network for 50 epochs leads to an accuracy of 90.15%.

The following plot shows the accuracies during fine-tuning.

![Figure 8.5: accuracies [in %] during fine-tuning of the model pruned with $\varepsilon$-ResNet and NISP using a pruning ratio of $\frac{1}{2}$ to classify Cifar-10](image)

After about 15 epochs the network reaches an accuracy of approximately 90%. This is about the same accuracy that is achieved after the first pruning technique was applied. Therefore also in this case $\varepsilon$-ResNet did not remove all of the redundancy.

To examine if even more parameters can be reduced, NISP with a pruning ratio of $\frac{1}{4}$ is applied to the network pruned with $\varepsilon = 2$.

This network is reduced to 160,666 parameters and can reach the initial accuracy as well. After fine-tuning for 50 epochs the model classifies the images with 88.89% accuracy.

The accuracies during this fine-tuning process are displayed in the next image.
CHAPTER 8. COMBINED APPROACHES

Figure 8.6: accuracies [in %] during fine-tuning of the model pruned with ε-ResNet and NISP using a pruning ratio of $\frac{1}{4}$ to classify Cifar-10

This plot shows that at least 25 fine-tuning epochs are necessary to retain the initial accuracy.

The combination of ε-ResNet and NISP with a pruning ratio of $\frac{1}{4}$ lead to a significant reduction with only small accuracy loss.

The 11,171,146 parameters are reduced to only 160,666 parameters. This is a reduction of 98.56% while still obtaining an accuracy of 88.98%.

In comparison by only applying NISP with a pruning ratio of $\frac{1}{4}$ the network is reduced to 700,762 parameters. The accuracy of this network is 89.98% which is only slightly better than the accuracy of the network pruned using both techniques even though the number of epochs is the same.

Both of these networks were first trained for 70 epochs and then fine-tuned for another 50 epochs.

Consequently by combining ε-ResNet and NISP better pruning results are achieved than when only one of these approaches is applied.
9 Conclusion

Deep neural networks have become very successful in various applications such as image classification.

However these deep neural networks are highly over-parameterized and contain a lot of redundancy. Consequently similar results could be achieved with more compact networks. [2, 1]

Moreover the compact networks are computationally and memory-wise more efficient. Therefore many different methods have been proposed to prune existing network architectures. [2]

After a short overview of various pruning approaches, three of them were analyzed in detail.

Singular value decomposition is a pruning technique which prunes weights. The number of parameters is reduced by decomposing the weight matrix of fully connected layers to two matrices with smaller dimensions. [24]

Next a pruning method called neuron importance score propagation was analyzed. Neurons or complete channels are removed depending on their importance. To compute the importance scores first the importance scores of the FRL are calculated by applying a feature ranking algorithm and then the scores are propagated to all the lower layers in the network. [8]

Lastly $\varepsilon$-ResNet was presented. This pruning technique builds on the original ResNet idea. By adding a few rectified linear unit functions complete layers that have responses less than $\varepsilon$ are automatically discarded. [4]

To evaluate these pruning techniques experiments were performed on three different network architectures for image classification.

All three approaches could successfully reduce the number of parameters with negligible
accuracy loss.

Among these methods NISP could achieve the best pruning results. However this technique requires fine-tuning which means that the training process is longer.

Moreover NISP is the only method which is applicable to all examined network architectures. Applying SVD is merely successful on networks containing mainly fully connected layers. And ε-ResNet can only prune layers of networks with a residual structure.

After analyzing each pruning technique, the methods were combined to examine whether a single technique already removes all of the redundancy or if the combination of two approaches achieves even better pruning results.

Applying SVD after the network was already pruned with NISP did not achieve any further reduction. However combining ε-ResNet with NISP led to outstanding pruning results.

The 18-layer network which classifies images of the Cifar-10 dataset could be reduced by 98.56% with only a small decrease in accuracy.

This significant compression accelerates testing time extremely. Shorter testing times are a considerable advantage for real time predictions.

Moreover this immense compression also reduces the required memory which allows networks to run on hardware limited devices such as smart phones, robots or cars. [1]

All in all the experiments in this thesis showed that deep neural networks can be pruned significantly to more compact models with negligible accuracy loss which brings many advantages.
Bibliography


[22] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks. In Proceedings of the 28th International


