Lehrstuhl für Data Science

A comparative evaluation of pruning techniques for Artificial Neural Networks

Bachelorarbeit von

Paul Häusner

Prüfer
Prof. Dr. Michael Granitzer

April 24, 2019

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Research question 2

2 Background **4**

2.1 Introduction to Artificial Neural Networks 4

2.1.1 The architecture of artificial neural networks 5

2.1.2 Gradient descent 10

2.1.3 Backpropagation algorithm 13

2.1.4 The overfitting problem 17

2.2 Perturbation theory 20

3 Methods **22**

3.1 Pruning methods 23

3.1.1 Magnitude based methods 23

3.1.2 Top-Down methods 24

3.1.3 Layer-wise methods 27

3.2 Pruning strategies 28

3.3 Implementation of sparse neural networks 30

4 Results **32**

4.1 Comparison of pruning strategies 34

4.2 Comparison of pruning methods 39

4.3 Baseline experiment 43

5 Discussion **47**

5.1 Comparison of pruning techniques 47

5.2 Pruning as regularization technique 50

6 Conclusion **52**

Bibliography **54**

Eidesstattliche Erklärung **59**

Abstract

Pruning of neural networks is one of several techniques that can be used to create sparse neural networks and to prevent neural networks from overfitting in general. In this thesis, different types of pruning techniques are compared with each other. Therefore, pruning techniques in general are divided into pruning strategies, which determine how many elements are deleted in each pruning step, and pruning methods that rank the weights based on their importance for the network. The different methods and strategies have been applied to some fully-connected networks that have previously been trained to a minimum on the MNIST dataset.
Our experiments have shown that iterative pruning is especially useful if we want to create highly compressed networks, while fixed number pruning is able to create the highest performing ones. In contrast, single pruning is not able to create as good networks as the other two approaches do. From the different magnitude based methods that have been applied, the magnitude class blinded approach clearly outperforms the other ones. The Optimal Brain Damage method, which is based on the second order derivative of the cost function, performs nearly the same as the magnitude class blinded method, but with an increasing number of pruning steps the accuracy drops and the additional needed computations don't pay off in our experiments. Furthermore, pruning in general is able to reduce the overfitting in the examined network dramatically and is even able to outperform other regularization techniques although more computational steps are required in the pruning approach compared to the other ones.

List of Figures

2.1 The model of a single neuron including the pre-activation value a with k input variables with weights attached to each of them and an additional bias term. 6
2.2 Comparison of the sigmoid function (a) and the step function (b), which are two widely used as activation functions for neurons in artificial neural networks. 7
2.3 A fully-connected neural network with five input variables, two hidden layers and two output variables is shown. The bias terms are not shown in order to maintain readability. 8
2.4 One possible way how the gradient descent algorithm finds the minimum of the function from a random starting point in $\mathbb{R}^2$ (adapted from [Hut18]) 12
2.5 Comparison of the error the network makes on the trainingset and the validation set with respect to the number of trained epochs. The best point to stop the training early is marked (adapted from [Shi+16]). . . . 18

3.1 Comparison of a fully-connected network (a) and a sparse neural network (b) that can be created from the fully connected network using pruning techniques. 22

4.1 The experimental process 33
4.2 Comparison of the accuracy after pruning is applied on four different initial models for the magnitude class uniform and the magnitude class blinded pruning methods. 34
4.3 Comparison of the effect of different pruning rates for iterative pruning and fixed number pruning on the accuracy of the model using the magnitude class blinded pruning method. 35

4.4 Comparison of fixed number pruning with iterative pruning using the magnitude class blinded pruning method and two fixed epochs of retraining after each pruning step. 36
4.5 Comparison of the single pruning strategy with iterative pruning and fixed number pruning. All evaluated models had a weight count around 10,000. For this experiment the magnitude class blinded pruning method is used. The network performance is evaluated before and after the retraining was applied to the model. 37
4.6 Comparison of the effect of the variable and two epochs fixed retraining on the accuracy of the model with different pruning methods. As a baseline the non-retrained accuracy is shown. 39
4.7 Comparison of the different used pruning methods in combination with fixed number pruning with a pruning rate of 5,000 and iterative pruning with a pruning rate of 25%. 40
4.8 Comparison of Optimal Brain Damage and magnitude class blinded pruning methods in combination with the fixed number pruning strategy using a pruning rate of 5,000. The methods are compared before and after the retraining is applied. 41
4.9 Comparison of the accuracy of magnitude class blinded (MCB), magnitude class distributed (MCD), magnitude class uniform (MCU), Optimal Brain Damage (OBD), layer-wise OBS (L-OBS), and random pruning methods in combination with the single pruning strategy before and after the retraining is applied. 42
4.10 Comparison of the model's accuracy in different settings: the original non-pruned model, the pruned model, the fine-tuned version of the pruned model and the created sparse model trained from scratch. 43
4.11 Comparison of the original and pruned model with models created using the popular regularization techniques weight decay and Dropout. The goal of all three compared techniques is to improve the generalization performance of the network. 44
4.12 Comparison of the performance of the two used architectures using iterative pruning with a pruning rate of 50% and a number of different pruning methods. The smaller model (b) has initially 79,400 weight while the bigger model (a) contains initially 266,200 weights. 45

5.1 Comparison of the results from previous research on pruning in RNNs and our own experiments on feed-forward networks. 48

List of Tables

3.1 Overview of the previously introduced pruning strategies and the pruning methods each strategy is used with. 29

4.1 The used hyper-parameters for the initially trained network selected using manual search. 32

1 Introduction

1.1 Motivation

Artificial neural networks are one of the most popular machine learning technologies these days. The tasks they can solve become more and more complex every day and with the increasing power the size of the network, meaning the number of neurons and connections between them, grows as well. A few decades ago, the number of parameters was fairly small, reaching up to a few hundred-thousand weights in the LeNet300-100 network for example [LeC+98], but nowadays deep neural networks with up to 100 million parameters are common, like the VGG-16 network for large scale image recognition [SZ15]. Therefore, it requires more and more storage capacity to save all the parameters, and the training and usage of the network becomes increasingly time-consuming.

Recent work has shown that neural networks contain many redundant weights [Den+13]. Therefore, it is possible to delete these unnecessary weights from the network in order to create a less complex model without experiencing a loss in performance. On the contrary, it is even possible that the network performs better after some redundant weights have been removed, since the network is less likely to overfit the training data [BH89]. Another advantage of removing redundant weights from the network, which is generally called pruning, is the decrease in complexity and size of the model. The decreased total size makes it easier to integrate deep neural networks with initially many parameters on embedded devices like smartphones, which tend to have a limited amount of computing power and storage [YC16].

On the other hand, the training process takes longer if it is started with a rather small network that is bearly able to learn the data's structure [Bur88], and the network is more likely to end in a local minimum with unsatisfying performance [RHW86]. Therefore, it is desirable to start the training process with a bigger network than actually needed

in order to reach satisfying network performance in a small amount of time and avoid underfitting.

In comparison to most of the other regularization techniques that also increase the network's ability to generalize, pruning can be applied when the network is already trained [Agg18]. Therefore, it is possible to generate a wide number of different models with different compression rates and different generalization ability from a single pre-trained model.

1.2 Research question

In order to reduce the complexity of the model after the training is done, a number of algorithms have been developed, which try to simplify the model of the neural network by removing the redundant weights. Each of these so-called pruning techniques tries to predict which weights can be removed without gaining a significant loss in performance and eventually calculate the change that needs to be applied to other weights in order to maintain a small overall error in the network. The algorithms vary in their computational complexity, the need for retraining, and their possible compression rate of the network for a desired accuracy [Ree93].

In this thesis, some of the developed pruning techniques are compared with each other. Therefore, in an already trained fully-connected network the predicted weights are removed and in the following, the performance of the sparse network is measured. The selection of the to be removed elements is based on evaluations of the weight's importance by the pruning method.

The goal of this thesis is to find the pruning technique that achieves the best results in terms of compression rate, stability, and the out-coming network's performance. Furthermore, the pruning methods are compared with each other based on their results and computational complexity in both theory and practice. In addition to the compared pruning methods, a number of pruning strategies for deleting the weights are used. Each pruning strategy determines how many elements are pruned in each pruning step and how many steps are executed. After each pruning step, the network performance is evaluated and retraining is executed if necessary.

Finally, pruning, in general, is compared with other techniques that try to find the best network structure and prevent the network from overfitting and are widely used in modern neural networks. All pruning methods and strategies are only evaluated in fully-connected feed-forward networks that have previously trained to a local minimum.

Since the research question of this thesis only considers the various pruning techniques, other methods preventing overfitting, like weight decay or Dropout, are not used for the initially trained model, since they affect the final structure and the parameter distribution of the network [HP89; Sri+14]. Some of the examined pruning methods are based on these parameters of the network as well and therefore, the just mentioned regularization techniques would affect the pruning procedure unpredictably.

As a baseline comparison in a highly compressed model which still performs well and was produced using one of the examined pruning techniques, the learned magnitudes of the weights are reset to a random uniform distribution, and the sparse model is trained from scratch. Afterward, the from scratch trained model is compared with the one that has been initially created by the pruning algorithm.

2 Background

2.1 Introduction to Artificial Neural Networks

The ultimate goal of Artificial Intelligence is to build an algorithm that is able to behave in an intelligent way. Although there is no exact definition of "intelligence", humans are often considered to be intelligent beings. Therefore, it is a logical step to try and copy the way humans learn and think: with their brain. The human brain is capable of solving a huge number of tasks that seem very simple for us humans but have shown to be very complex for machines like image recognition or text processing [RN09].

The human brain is an extremely complex system containing 100 billion processing units, which are refered to as neurons with up to 100 trillion connections between them [STK05]. The connections between the neurons, the so-called synapses, and the neurons themselves form a neural network together. Any brain activity involves electrical signals which are transmitted between the neurons by the synapses until they reach a designated area. By strengthening or weakening the single connections, the so-called synaptic plasticity, the brain is able to learn and memorize information. It is even possible that some synapses are removed and new ones are added [Hug58]. How exactly the brain is able to process the incoming signals, to learn from them and to remember the learned information is still not very well understood by the asociated field of neuroscience. Therefore, this scientific field is a big area of research with the goal of understanding how the human brain exactly works [Mer].

Although Artificial Intelligence is still far away from reaching human level of performance on a lot of tasks, it was accomplished to solve some difficult tasks by developing algorithms that are inspired by the human brain and in some tasks the algorithms created by human engineers even outperform the human brain [Sil+18]. These algorithms are called *artificial neural networks*, since they are inspired by the human neural networks, and nowadays they are used in a wide variety of fields like image recognition

[KSH12], medical diagnosis [Ama+13], and speech recognition [GMH13] to only name a few examples.

Artificial neural networks are especially popular in the field of *supervised machine learning*, where the machine tries to predict the correct values of samples based on previously seen similar samples where the right solution was provided to the network [RN09]. In the following section, the basic blocks of artificial neural networks will be introduced and how the artificially created “brain” is able to learn from the given examples in order to correctly predict new samples that the algorithm has never seen before will be explained.

2.1.1 The architecture of artificial neural networks

Just like in their biological role model, the human brain, the basic blocks an *artificial neural network* (ANN) is made of are called *neurons*. During the 1950s already, a special kind of artificial neuron, the so-called perceptron, was invented by Rosenblatt [Ros58]. A perceptron has a number of inputs $x_1, ..., x_k$ and a binary output, meaning the output can either be zero or one. Each input is a binary number as well, but the input x_i has a *weight* w_i attached to it, which indicates how strong and therefore how important the connection from this particular input is.

The perceptron's output is one if the sum of the weighted inputs is above some predefined threshold. The threshold of the perceptron can be modeled by another input to the perceptron, which always takes the value one, and the weight of this particular input determines the level of the threshold. This additional input, which shifts the threshold to the desired value, gets called *bias* b. Therefore, the output of the perceptron can be expressed in a single equation as follows [Nie15]:

$$output = \begin{cases} 1, & \text{if } \sum_{i=1}^{k} w_i x_i + b \geq 0 \\ 0, & \text{otherwise} \end{cases} \tag{2.1}$$

The major pitfall of the perceptron is that its output is always a linear function. Since most of the real-worlds' problems are highly non-linear, perceptrons are not able to model the real-world structure and therefore, it is not suitable as a learning algorithm for many real-world problems. A popular example is the fact that a perceptron is not able to learn the XOR function, since it can't be seperated by a linear line which is returned by the perceptron's output function [YBZ02].

In order to overcome the just described issue, a non-linear function can be used in order to determine the output of the network instead of the linear step function that was used in the perceptron. The function which is applied to the weighted sum of the inputs a, that are computed as shown in equation 2.2, gets called *activation function*.

$$a = \sum_{i=1}^{k} w_i x_i + b \tag{2.2}$$

As already mentioned, the activation function in a perceptron is the step function, as defined in equation 2.1, which is one if either the value is equal or bigger than zero and zero otherwise. A widely used non-linear activation function is the so-called sigmoid function $\sigma(\cdot)$, which is defined by the following equation [Nie15]:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.3}$$

If the previously introduced pieces are put together, the model of a single neuron with the sigmoid activation function can be built as shown in figure 2.1. The weighted inputs of the network are summed up and the bias is added. This sum, which forms the pre-activation value a, is then turned into the output y of the neuron by applying the activation function to it.

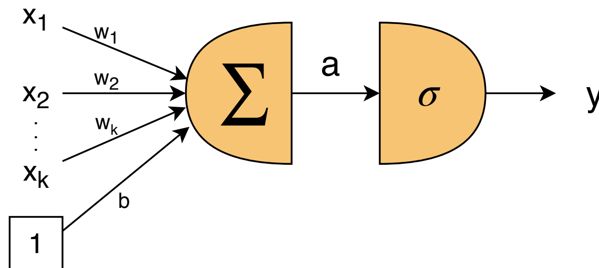
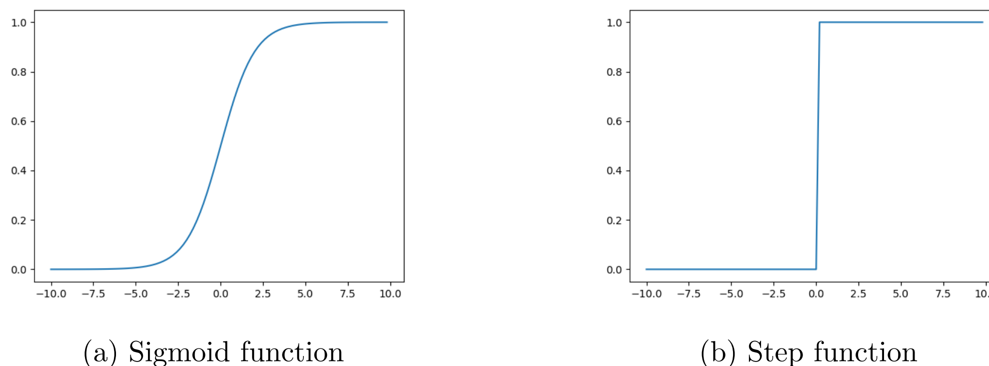


Figure 2.1: The model of a single neuron including the pre-activation value a with k input variables with weights attached to each of them and an additional bias term.

At first sight, the sigmoid function that was introduced in equation 2.3 looks quite different compared to the previously used step function, which had been used in the perceptron. As shown in figure 2.2 the behavior of the two introduced functions is actually pretty similar. The only actual difference is that instead of jumping from zero to one at a certain point, a smooth transition between the two values is created [Nie15]. Beside the two just introduced activation functions, a wide number of other activation

functions exists. The concrete choice of the activation function depends on the given problem and the network's desired output and is an important design decision for the neural network [Agg18].



(a) Sigmoid function

(b) Step function

Figure 2.2: Comparison of the sigmoid function (a) and the step function (b), which are two widely used as activation functions for neurons in artificial neural networks.

Since the sigmoid function produces a non-linear output, the network is now able to learn more complex structures in the data. Although later work has shown that the XOR problem and other tasks that are not linear separable can be solved by using multilayer networks of perceptrons [YBZ02], neurons with continuous activation functions like the sigmoid function are preferred nowadays. The reason for this is that it is a lot easier to train a neuron with non-linear activation functions than training a perceptron. The learning of a neuron is done by increasing or decreasing the weights of a neuron until the networks output reaches the desired values, similar to learning in the human brain where the synapses are strengthened or weakened based on the electrical signals the synapses have transmitted. In a neuron with a non-linear activation function, a small change in a weight also leads to a small change in the output [Nie15].

In contrast to the behavior of a neuron, in a perceptron with the step function as its activation function a small adjustment of a weight either leads to a huge change in the output, meaning the output switches from zero to one or vice versa, or no change at all. Therefore, it is a lot easier to find the correct weights and bias that produce the desired output in a neuron, since it is possible to adjust these values in small steps until the desired output is reached. This is especially useful if there are too many inputs to determine the values of the weights that produce the desired output directly.

In order to solve a difficult problem with many inputs, like image recognition, where every single pixel of the image is an own input to each neuron, many neurons have to work together by forming a network of neurons. Inspired by the nature, the neurons are organized in layers. In the *input layer* the raw input signals are fed. The only task for the input layer neurons is to feed this received signal to the next layer. The last layer of the network is called the *output layer*, since its signals are returned to the user. Because the user of the network doesn't see what happens between input and output layer, these layers get called *hidden layers*. In figure 2.3 a small multilayer network with two hidden layers, each containing three neurons is shown [Nie15; Zel00].

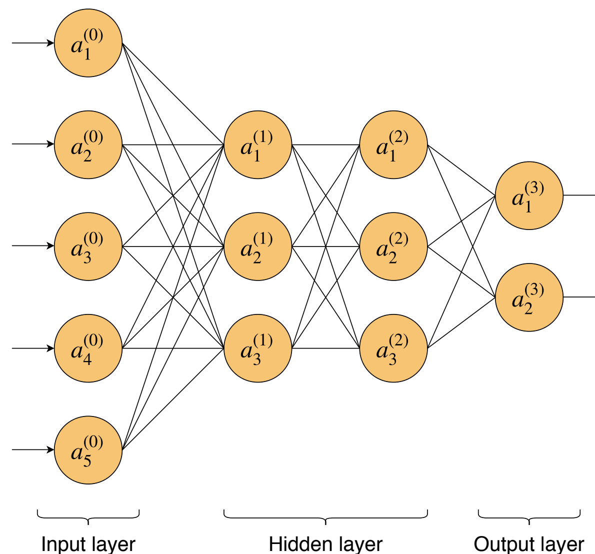


Figure 2.3: A fully-connected neural network with five input variables, two hidden layers and two output variables is shown. The bias terms are not shown in order to maintain readability.

Although the *universal approximation theorem* states that a network with one hidden layer and a finite amount of neurons can model nearly any function, architectures with many layers are common nowadays. The reason for this is that by adding more layers the total number of parameters might decrease and it gets easier to train the network compared to a network with a single hidden layer and a huge number of neurons in that layer [GBC16]. Therefore, deep neural networks with over 100 layers are not uncommon nowadays [He+16].

Choosing the correct number of hidden layers and neurons in them is still one of the most difficult tasks in the design of a neural network and therefore, often has to be done

manually or by determining a good parameter with extensive computing power using grid search [Nie15]. However, there are also new approaches that try to speed up the hyper-parameter search like the random search algorithm [BB12].

In order to solve a wide variety of problems with artificial neural networks, a number of architecture styles, which are used for different types of tasks, have been developed. In *feed-forward* networks, neurons from the layer l only receive input from nodes of the the previous layer $l-1$ or more general, if shortcuts are allowed, only inputs from nodes from layers with a lower index are fed into the node. In particular, there are no cyclic structures in the network, because this would make it way more difficult to train the neural network and therefore, the network is modeled as a direct, acyclic graph [Zel00].

In *fully-connected* layers, each node from the layer l has an incoming connection from each node from the previous layer $l-1$, which form the input of the node. The weight that is connecting node k from layer $l-1$ to node m in layer l is called $w_{mk}^{(l)}$. The last layer, which is the output layer of a network, will be referred to as layer L. It is easy to see that all the weights that connect layer $l-1$ to layer l are forming a matrix, which is called *weight matrix*, where each row describes the incoming weights for a single node from layer l and each column describes the outgoing weights from a single node from layer $l-1$. Furthermore, $k^{(l)}$ denotes the number of output weights in a layer l and therefore, the weight matrix from layer l has the size $k^{(l)} \times k^{(l-1)}$. The vector, which contains the pre-activation values from the neurons of a layer l, can be expressed as the matrix product of the outputs from layer $l-1$, called $y^{(l-1)}$, and the weight matrix which is denoted by $w^{(l)}$. Therefore, this vector can be calculated efficiently using the matrix product as shown in equation 2.4 [Zel00].

$$a^{(l)} = w^{(l)} \cdot y^{(l-1)} \tag{2.4}$$

One pitfall of feed-forward and especially fully-connected networks for sequential data is that it is not possible to maintain information between single inputs in the network and therefore it is not possible to recognize dependencies between the inputs. An attempt to overcome this issue is to use *recurrent neural networks* (RNN) which make use of an internal state in the network in order to solve problems that require high dependency in the data like text processing or speech recognition. Along with the data that is passed into the network, the internal state of the previous data piece is added to the network using a feedback-loop. This enables the network to “remember” the previously seen

data. However, this additional input leads to a bootstrapping problem, since for the first input no previously state is available [Agg18; GBC16].

Convolutional neural networks (CNN) are another popular form of feed-forward networks which contain convolutional layers that are sparse connected layers following a strict pattern. The pattern has been adapted from animal brains, where a similar pattern was discovered in the visual cortex, which is responsible for image recognition [LB98]. Therefore, convolutional networks are used especially in the field of image recognition, since the sparse structure corresponds to the pixel position in the image, but the technique has also been transferred to other types of problems. Convolutional networks often also contain other types of layers like fully-connected layers. Their usage has led to a huge performance improvement in neural networks [Agg18].

2.1.2 Gradient descent

In order to determine how good the network is currently performing in solving the desired problem, a measurement of the current error of the network, which gets called *cost function* $E(\cdot)$, is needed. The cost function compares the actual output of the network with the desired one that is available in the dataset for supervised machine learning problems. Therefore, the cost function shows how big the error is the network currently is making on a set of provided samples. A popular measurement of the error is the *Mean-Square-Error* (MSE) which is defined for a network with a single output as follows [Nie15]:

$$E = \frac{1}{2n} \sum_{i=1}^{n} (y - \hat{y})^2 \tag{2.5}$$

Here, y is the actual output of the network, $\hat{y}$ is the desired output given by the dataset, and n is the number of training examples on which the network's performance is measured. The $1/2$ factor is added to the error because this leads to a less complex derivation, which will be shown in section 2.1.3 during backpropagation, and the constant factor doesn't affect the optimization term [Zel00]. The cost function from equation 2.5 can be expanded for multiple output nodes $y_1, ..., y_k$ by just summing up the error each single node makes as shown in the following equation:

$$E = \frac{1}{2n} \sum_{i=1}^{n} \sum_{j=1}^{k^{(L)}} (y_j - \hat{y}_j)^2 \tag{2.6}$$

As one can see, the cost function from equation 2.5 is just a special case of the general cost function from equation 2.6. The final goal of training the neural network is to find a global minimum of this cost function based on all of the network's inputs and desired outputs that are available in the dataset with respect to the variables, which, in this case, are all the adjustable weights and biases of the network. Since the number of weights and biases can be extremely high, e.g. over 100 million in the VGG-16 network, it is computationally not possible to determine the minimum of the function with classical calculus [SZ15].

Therefore, other techniques to minimize the cost function have to be used. One of the most used training algorithms for neural networks is the iterative optimization algorithm *gradient descent*. The key idea of gradient descent is adjusting the weights and biases of the network in small steps. These steps are in proportion to the negative of the error function's gradient and are executed until a minimum of the cost function is reached [Zel00; GBC16].

The main difficulty to overcome is how to calculate the partial derivatives for all of the weights and biases of the network in a traceable fashion. This problem can be solved using the backpropagation algorithm, which is introduced in section 2.1.3. The gradient descent algorithm uses the formula from equation 2.7 in order to update the value of the weight w_{ij}, once the associated partial derivative of the cost function has been computed. Analogous to the update of the weights, the biases are updated as well. In order to update all variables of the network, the complete gradient of the cost function has to be computed.

$$w_{ij} = w_{ij} - (\eta \cdot \frac{\partial E}{\partial w_{ij}}) \tag{2.7}$$

The factor η describes the learning rate of the algorithm. The learning rate is needed, since the algorithm is not able to find a minimum otherwise. The gradient descent algorithm makes use of the fact that the change in the function caused by a parameter perturbation can be modeled using the derivative in a small surrounding of the current value. Therefore, the learning rate needs to be quite small in order to keep the approximation close to reality [GBC16]. Typical values for the learning rate are between 10^{-2} and 10^{-5}. The choice of the learning rate is crucial. If the learning rate is too small, the optimization might get stuck in a local minimum. On the other hand, if the learning rate is too big, the global minimum can not be reached, since the algorithm is overshooting [Nie15].

Although gradient descent doesn't provide a guarantee that the global minimum will be reached, experience has shown that the algorithm often finds a satisfying minimum [GBC16]. In order to increase the probability that a good minimum is found by the algorithm, the network can be trained with different initial weight and bias values multiple times, so the final networks can be compared with each other in terms of performance and the best one can be selected [Pat97].

Figure 2.4 shows how the gradient descent algorithm can find a minimum by taking small steps in the opposite direction of the gradient. An often used analogy for gradient descent is imagining a ball that is placed at a spot on the function and it then rolls downhill until it reaches a local minimum [Nie15].

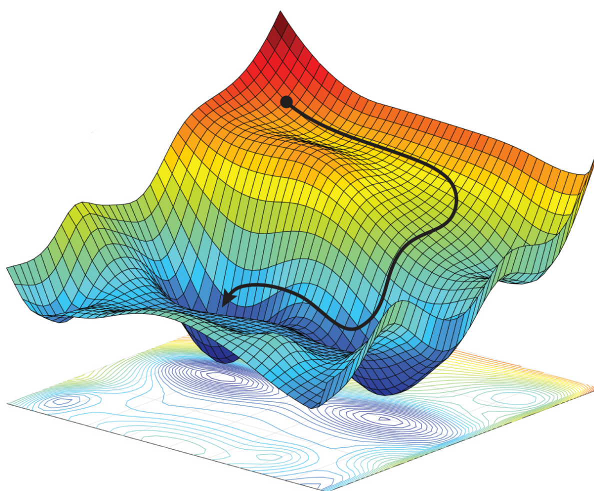


Figure 2.4: One possible way how the gradient descent algorithm finds the minimum of the function from a random starting point in $\mathbb{R}^2$ (adapted from [Hut18])

.

A popular variation of the gradient descent algorithm is the *stochastic gradient descent* algorithm in which, instead of computing the gradient as the mean value of all the training samples, the gradient is computed in isolation for every training sample and then the gradient step is executed with this particular gradient vector. This corresponds to computing the error from equation 2.6 for the parameter $n = 1$ [Agg18].

Taking only one training example per gradient step is often not enough and it takes a long time until a minimum is reached, since the algorithm often changes the parameters in a wrong way for most of the samples. Therefore, a mixture of the both previously introduced variations, which is called *mini batch gradient descent*, is often used. In

this variation of the algorithm, a number of samples from the training dataset, which is referred to as batch, is used. Typical batches include between 25 and 250 samples depending on the examined dataset [Agg18]. If the batch size is chosen sufficiently and the dataset is uniquely distributed, the gradient computed of the batch is a good approximation of the real gradient for the whole dataset [Nie15].

Since in every step only a small change in the parameters is made, the data has to be fed through the network multiple times. Each iteration of the complete dataset is called *epoch*. In the originaly introduced gradient descent algorithm, each single gradient step corresponds to a single full epoch [Agg18].

2.1.3 Backpropagation algorithm

In the last section, the gradient descent algorithm, which makes use of the partial derivatives with respect to each weight and bias of the network and is able to minimize the error of the artificial neural network, has already been introduced. The problem, however, is how to calculate the whole gradient vector, which consists of all the partial derivatives, especially the parts of the gradient that are not directly connected to the output of the network but in the hidden layers. To compute these elements of the gradient vector, the *backpropagation* algorithm has been invented [RHW86].

The backpropagation algorithm consists of two phases: the forward phase and the backward phase. In the forward phase, a number of samples from the dataset are fed into the network and the corresponding error is calculated with the error function as seen in the previous section. In the backward phase, the gradient of the error function with respect to the weights and biases is calculated. The single components of the gradient vector are the partial derivatives of the cost function with respect to each single weight and bias. The key idea of backpropagation is to firstly calculate the partial derivatives for the weights that are connected with an output node, since these can be computed straight forward, and thereupon calculate an error for each node from the hidden layer that can be used for the derivative calculation for the next layer's weights and biases. So the actual error of the network gets propergated back through the network and in the end, the complete gradient can be computed [Agg18].

For simplicity, it is assumed that the forward phase only contains one single training sample, like it is the case for stochastic gradient descent. If there were multiple examples,

the final gradient could be calculated as the sum of the gradients for each single error. The first step is, as just pointed out, calculating the partial derivative of the cost function for the weights that are connected to the output. This means all the partial derivatives for the weights and biases from the output layer L have to be calculated. The partial derivatives for the weights can be calculated using the chain rule which is leading to the following formula [Nie15]:

$$\frac{\partial E}{\partial w_{ij}^{(L)}} = \frac{\partial E}{\partial y_i^{(L)}} \cdot \frac{\partial y_i^{(L)}}{\partial a_i^{(L)}} \cdot \frac{\partial a_i^{(L)}}{\partial w_{ij}^{(L)}} \tag{2.8}$$

Now the single parts of the product can be calculated in isolation and later reassembled into the complete partial derivation. The first part of the derivation can be expanded as follows by just inserting the definition of the cost function. As previously pointed, out the $1/2$ term that is added to the cost function vanishes during the derivation, as one can see in the following equation [Nie15; Zel00].

$$\frac{\partial E}{\partial y_i^{(L)}} = \frac{\partial}{\partial y_i^{(L)}} \cdot \frac{1}{2}(y_i^{(L)} - \hat{y}_i)^2 = y_i^{(L)} - \hat{y}_i \tag{2.9}$$

Since each weight in the output layer only has influence on a single output node its connection to all other output nodes is irrelevant because they disappear during the derivation. Therefore, the equation 2.9 remains the same even if there is more than one output node in the network. The second part of the equation can be simply computed as follows:

$$\frac{\partial y_i^{(L)}}{\partial a_i^{(L)}} = \frac{\partial}{\partial a_i^{(L)}} \sigma(a_i^{(L)}) = \sigma'(a_i^{(L)}) \tag{2.10}$$

The derivation of the activation function can be solved easily as we will see later. The last part of the term can also easily be solved by inserting the formula for the calculation of the pre-activation value a_i:

$$\frac{\partial a_i^{(L)}}{\partial w_{ij}^{(L)}} = \frac{\partial}{\partial w_{ij}^{(L)}} \cdot \sum_{n=1}^{k} w_{in}^{(L)} y_l^{(L-1)} + b_i^{(L)} = y_j^{(L-1)} \tag{2.11}$$

If the results from the three equations from above are put together, the partial derivative of the error function with respect to each of the single weights from the output layer of the network is received and the result of equation 2.8 can be seen in the following equation:

$$\frac{\partial E}{\partial w_{ij}^{(L)}} = (y_i^{(L)} - \hat{y}_i) \cdot \sigma'(a_i^{(L)}) \cdot y_j^{(L-1)} \tag{2.12}$$

Similar to the calculation of the weight's partial derivative, the partial derivative for the bias can be computed. The only difference is in the last part of the product from equation 2.11 which changes as follows:

$$\frac{\partial a_i^{(L)}}{\partial b_i^{(L)}} = \frac{\partial}{\partial b_i^{(L)}} \sum_{l=1}^{k} w_{il}^{(L)} y_l^{(L-1)} + b_i^{(L)} = 1 \tag{2.13}$$

Therefore, the last term vanishes from the product and the formula for the calculation of each partial derivative of the cost function with respect to the bias b_i is given by:

$$\frac{\partial E}{\partial b_i^{(L)}} = (y_i^{(L)} - \hat{y}_i) \cdot \sigma'(a_i^{(L)}) \tag{2.14}$$

In the two final equations, the calculation of the sigmoid function's derivative is simply given by the following formula [Agg18]:

$$\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x)) \tag{2.15}$$

The previously discussed calculation of partial derivatives is with respect only to the weights and biases that are connected directly to the output of the network. The problem is that the partial derivative with respect to $y_i^{(l)}$ can't be computed directly, since $y_i^{(l)}$ is not a direct part of the calculation of the error in the case for a layer $l \neq L$. In order to calculate the gradient of the network's error, the partial derivatives with respect to these elements of the network have to be computed. The key idea to determine the missing parts of the gradient vector is to calculate the partial derivative from each element in the hidden layer $l \in \{1, ..., L-1\}$ based on the partial derivatives from the previous layers $\{l+1, ..., L\}$. Therefore, we want to assign an error to each node from the hidden layer, similar to the error that was calculated at the output layer of the network. The error of the node should represent the influence of the examined node's output on the final error of the network. This can be achieved by summing up all the weighted errors from the next layer in the network [Nie15; Ras17].

The proportional error of each node is mathematically calculated as the partial derivative of each node's pre-activation value $a_i^{(l)}$ which can be obtained by applying the multivariable chain rule to the cost function. This leads to the following equation for the partial

derivative of the cost function with respect to the pre-activation value a of node i from layer l [Agg18]:

$$\frac{\partial E}{\partial a_i^{(l)}} = \sum_{j=1}^{k^{(l)}} \frac{\partial E}{\partial a_j^{(l+1)}} \frac{\partial a_j^{(l+1)}}{\partial a_i^{(l)}} \tag{2.16}$$

$$= \sum_{j=1}^{k^{(l)}} \frac{\partial E}{\partial a_j^{(l+1)}} \cdot w_{ji}^{(l+1)} \cdot \sigma'(a_i^{(l)}) \tag{2.17}$$

$$= \sigma'(a_i^{(l)}) \cdot \sum_{j=1}^{k^{(l)}} \frac{\partial E}{\partial a_j^{(l+1)}} \cdot w_{ji}^{(l+1)} \tag{2.18}$$

The partial derivative can be divided into a part that can be solved locally and partial derivatives from layers which are closer to the output than the current node. For the unsolved partial derivatives, there are two options: either the layer $l + 1$ is an output layer and therefore, the derivation can be solved as explained with equation 2.9 and 2.10, or equation 2.18 has to be used recursively again in order to solve the term. For either of the two options, the corresponding values have already been computed in the backpropagation; just as the word “back” suggests, the calculations of the partial derivatives are started at the output layer L and the values are then propagated through the network until the input layer is reached [Agg18].

For the calculation of the partial derivatives with respect to the bias terms, the calculation is analogous as it has already been seen in the previous case for nodes connected to the output. By putting together the so far achieved results, this leads to the following formula for calculating the partial derivatives of the cost function with respect to the weights in the hidden layer:

$$\frac{\partial E}{\partial w_{ij}^{(l)}} = \frac{\partial E}{\partial a_i^{(l)}} \cdot \frac{\partial E}{\partial w_{ij}^{(l)}} \tag{2.19}$$

$$= y_j^{(l-1)} \cdot \sigma'(a_i^{(l)}) \cdot \sum_{n=1}^{k^{(l)}} \frac{\partial E}{\partial a_n^{(l+1)}} \cdot w_{ni}^{(l+1)} \tag{2.20}$$

The equation 2.19 is given by the chain rule and the second equation 2.20 is obtained by applying equation 2.18 and 2.11 that already have been computed earlier.

The backpropagation algorithm works, since the computing graph doesn't have cycles and therefore, it is computationally affordable to determine the partial derivatives in a

recursive way, since all needed values in the recursion step have already been computed and only need to be summed up. Therefore, it is sufficient to calculate the partial derivative for each element exactly once [Agg18; GBC16].

By putting together all the just introduced pieces, it is now possible to compute all partial derivatives with respect to the weights and biases, thus the gradient of the error function is obtained. Therefore, it is now possible to adjust the parameters in proportion to this just calculated gradient of the network using the gradient descent algorithm that has been introduced in section 2.1.2 in order to minimize the cost function.

This calculation of the gradient and subsequent adjustment of the parameters has to be iterated a number of times until the network's error is sufficiently small. But since the goal is to make the network able to correctly predict new samples it has never seen before, training should be stopped at a certain point. As it will be seen in the next section, it is more efficient to stop rather early than training too long in order to receive a network with good generalization performance.

2.1.4 The overfitting problem

The goal of an artificial neural network is to learn the structure in the training data and after the training is completed to correctly predict new samples the network has never seen before. In order to test how well a network is able to *generalize*, meaning how good the network's performance is on previously unseen data, the dataset is usually split into a training part which is used to train the network using backpropagation or another learning algorithm and a smaller test part which is used to evaluate the network's performance on this unseen portion of the dataset. Therefore, the goal of training a neural network is not only to decrease the error the network makes on the training set but also to keep the distance between training error and test error as small as possible [GBC16].

If the network has a too high capacity or is trained for a too long time, the training error is very small but a high difference between training and test error emerges. This so-called *overfitting* problem is one of the most important problems that needs to be solved in the whole field of machine learning [Agg18]. The overfitting problem especially occurs if the network has more parameters than needed in order to learn the structure

in the data, since that's when it is most likely that the free parameters learn some of the training data specific structure from the dataset [GBC16].

It is often very difficult to choose the right number of parameters that avoid overfitting as well as *underfitting*, where the network is not able to learn the structure of the data. Therefore, if underfitting occurs, the overall training error does not reach satisfying performance. The most common cause for underfitting is a too small capacity of the network which can be caused by a too small number of parameters for example [GBC16].

Since it is very hard to implement counter mechanisms for underfitting, besides of increasing the network capacity by adding more parameters to the network, often a fairly high number of parameters is chosen and the overfitting problem is tackled by other techniques which are easier to use [Haw04]. Techniques that try to avoid overfitting and increase the network's ability to generalize are called *regularization techniques* [GBC16].

The previously introduced effect of overfitting can be observed during the training process. Although the error on the training dataset keeps decreasing with more training epochs, the test error quickly increases after a certain point, when the network starts to overfit the data. This behavior can can be seen in figure 2.5 and the best point to stop in order to obtain a network with high generalization performance is if the training error stops decreasing, since at this marked point the network has the best generalization ability. The technique of stopping at this particular point is called *early stopping*. Because of the simplicity and understandability, early stopping is one of the most used regularization techniques [Agg18].

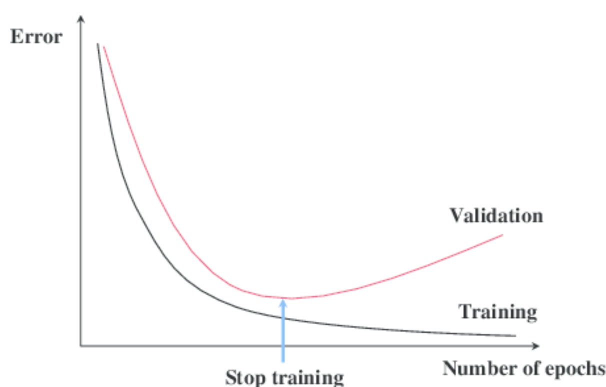


Figure 2.5: Comparison of the error the network makes on the trainingset and the validation set with respect to the number of trained epochs. The best point to stop the training early is marked (adapted from [Shi+16]).

The point to stop can either be determined by the current error on the test set or by introducing a third portion of the dataset, the so-called validation set. Similar to the test set the validation dataset is not used to train the network but to determine the point to stop the training. Since this portion of the dataset is not used for training, the network won't overfit this data, since it hasn't been seen by the network so far. The error on the validation set is used to determine when to stop training and therefore, to find a good approximation of the early stopping point as shown in figure 2.5 [Pre98]. If the network has fewer parameters, the chance of overfitting decreases, since the parameters can't hold any more information, but as already mentioned, it is very hard to find the optimal number of parameters and training takes longer if the network is barley able to learn the data [Agg18].

In order to overcome the overfitting problem, a various number of techniques have been developed. In $\mathcal{L}_1$ and $\mathcal{L}_2$ regularization, penalties are added to the cost funcition for high magnitude weights. This doesn't only reduce the weight's sizes but is also an effective and widely used mechanism to prevent overfitting. Examples for these techniques are the widely used weight decay [HP89] or the penalty suggested by Chauvin [Cha89]. The pitfall is that these techniques often don't lead to sparse networks but to models with many small magnitude weights and therefore, don't reduce the network size and complexity [Ree93].

Another type of regularization is the addition of more layers with fewer number of neurons in each layer, although the total number of nodes might rise the number of weights decreases and therefore, the network is less likely to overfit the training data [Agg18]. It is also possible to let one changeable parameter control a number of weights which is known as weight sharing [NH92]. This obviously reduces the risk of overfitting, since the number of changable parameters decreases. Dropout and Dropconnect suggest to drop some weights or neurons randomly during the training but add them back to the network later again. These two technique increase the accuracy of networks significantly and are widely used [Wan+13; Sri+14].

Another not that widely used but promising method to prevent overfitting is creating a sparse networks with few weights either before the training with genetic algorithms for example [MTH89], during the training with techniques like $\mathcal{L}_0$ regularization [LWK18], or after the training using one of several developed pruning techniques which will be introduced in detail in section 3.1.

2.2 Perturbation theory

As known from calculus, every function can be expressed with a Taylor series as an infinite sum and furthermore, it is possible to expand this series to take small changes in the input into account. Therefore, it is possible to approximate the change in a function $f(\cdot)$ if the input in the function gets perturbed by a value Δx. The Taylor series of the perturbed function is given by the following equation [Hol13]:

$$f(x + \Delta x) = f(x) + \Delta x f'(x) + \frac{1}{2}\Delta x^2 f''(x) + \mathcal{O}(||\Delta x||^3) \tag{2.21}$$

This equation can also be transferred to a function with n input variables and a perturbation vector with correspondingly many perturbations. The Taylor expansion for this multivariable function therefore changes as follows [Hol13]:

$$f(x_1 + \Delta x_1, ...x_n + \Delta x_n) = \tag{2.22}$$

$$f(x_1, ..., x_n) + \sum_{i=1}^{n} \frac{\partial f}{\partial x_i} \cdot \Delta x_i + \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \Delta x_i \Delta x_j \frac{\partial^2 f}{\partial x_i x_j} + \mathcal{O}(||\Delta x||^3) \tag{2.23}$$

The equation above requires the computation of the whole Hessian matrix H, which contains all second derivatives of a function as you can see in the third term of the sum. Since the Hessian Matrix requires $\mathcal{O}(n^2)$ memory and time to be computed, it quickly becomes unfeasible to compute the complete Hessian matrix. To overcome this issue, a number of algorithms have been developed to approximate the Hessian, or more advanced assumptions about the approximation are made.

Often the main interest is the actual change of the output function that is already minimized and, since the function is currently at a local minimum, the first part of the sum is equal to zero. Since many error functions are quadratic, the last term can be dropped, too, which leads to the perturbation error as follows:

$$\delta f(\Delta x) = f(x + \Delta x) - f(x) = \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \Delta x_i \Delta x_j \frac{\partial^2 f}{\partial x_i x_j} \tag{2.24}$$

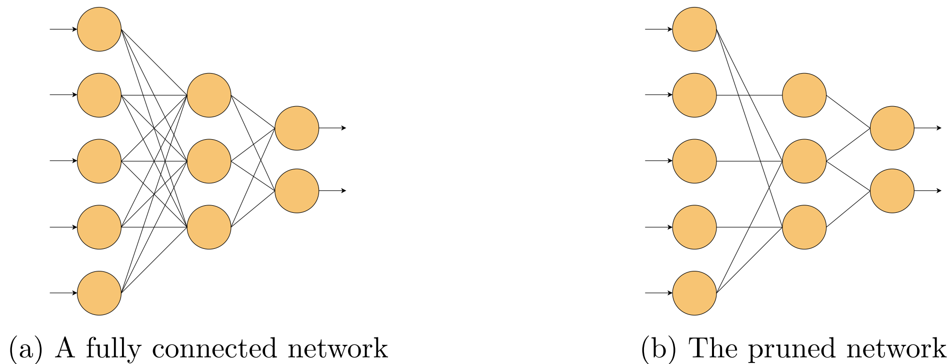
$$= \Delta x^T H_f \Delta x \tag{2.25}$$

A big disadvantage of the Hessian Matrix is that it requires a lot of memory and computing power in order to determine it. Therefore, a number of heuristic methods to calculate

the matrix in reasonable time have been developed. A popular algorithm to approximate the Hessian matrix is the BFGS algorithm. This algorithm is a Quasi-Newton method which approximates the Hessian in an iterative way [Fle00].

3 Methods

Research in neuroscience has shown that when we sleep connections in the human brain are weakend or even removed which is probably part of the learning process and contributes to the ability to generalize [Viv+17]. Therefore, the goal of pruning algorithms is to remove neurons from the artificial neural network in order to improve the learning process in these artificial "brains" as well and furthermore, research in the field of machine learning has shown that neural networks contain many redundant weights that might be removed [Den+13]. The initially trained network is fully-connected as shown in figure 3.1a. Then, some weights are selected and removed from the already trained network leading to a sparse neural network as shown in figure 3.1b.



(a) A fully connected network

(b) The pruned network

Figure 3.1: Comparison of a fully-connected network (a) and a sparse neural network (b) that can be created from the fully connected network using pruning techniques.

The research question in this thesis is how to select the correct weights to remove from the network and how many weights can be removed in order to keep a good performing network. Therefore, different techniques for selecting the correct weights and stopping the pruning at the right time have been developed and will be compared in this thesis. The *pruning methods* rank the weights from the network and assign a so-called *saliency* to each weight. After the ranking is done, some of the weights with the lowest saliency

are deleted. The ranking of the weights' saliencies requires $\mathcal{O}(n \cdot log(n))$ steps for a network with n parameters. The number of the deleted weights is based on the selected *pruning strategy* and the deletion can be done in $\mathcal{O}(n)$ steps. After each *pruning step*, which consists of the weight ranking and the deletion of one to several weights, retraining might be necessary. It is also possible that no retraining is needed, since the remaining weights can be adjusted during the pruning step. However, this additional effort often makes the saliency calculation more expensive.

3.1 Pruning methods

The simplest way to rank the weights and therefore choose which of them should be deleted from the network is to just randomly pick them. Obviously, this doesn't lead to good network performance, since the chance that important weights are deleted is relatively high. Furthermore, each pruning iteration will create a completely different network. Although retraining can make up some of the lost information, the random method often leads to an unsatisfying final model, since it can't be made sure that all output nodes keep having a connection to the remaining network, and the overall network structure gets damaged. Although this method often doesn't results in a good performing network, the calculation of the saliency can be done relatively quick by assigning a random number to each parameter. This requires only $\mathcal{O}(n)$ steps if a random number generator which creates random numbers in $\mathcal{O}(1)$ is provided. However, due to its simplicity, random pruning is a good baseline comparison for the other examined methods.

In order to obtain better networks by pruning, it is desirable to think of other methods that try to avoid deleting important weights from the network. In the following, different pruning methods that are based on different criteria will be explored. The common goal of the pruning techniques is to determine the weights which deletion only causes a small change or no change at all in the error of the network.

3.1.1 Magnitude based methods

Another simple approach is deleting the weights with the smallest absolute magnitudes. The main idea is that a small magnitude also results in a small change in the output of

the network, when the weight is deleted. Unfortunately, research has shown that small magnitude weights are often responsible for keeping the overall error low in the network [Bar97]. Therefore, after each pruning step, expensive retraining is required in order to decrease the error to a sufficient level again.

Beside the retraining issue, magnitude based pruning methods are very popular, since the computation of their weight saliency is very easy and can be done in $\mathcal{O}(1)$ for the network, since the saliencies have already been calculated by the backpropagation algorithm. Furthermore, the development of an magnitude based algorithm is very simple and doesn't require much additional effort. There are different approaches which elements will be deleted by magnitude based pruning. The simplest one suggests to remove the smallest elements from the network regardless of their class, i.e. their layer. This method is called *magnitude class blinded* pruning [Mar+18]. Another option is to delete the same amount of elements from each class, which is generally referred to as *magnitude class uniform* pruning method [SLM16].

A slightly more complicated technique is to delete elements from each class in proportion to the its standard deviation. The key idea behind the latest, so-called *magnitude class distributed* method is that in classes with a high standard deviation respectively more elements get deleted from the network than in ones where the weights are more similar [Han+15; SLM16]. The problem with this method is how to determine a good global value, which is called t, in order to reach the desired compression rate in each pruning step [Mar+18]. The disadvantage of class distributed pruning is that additional computational effort as well as programming is required in order to calculate the standard deviation and the necessary global value. Therefore, this magnitude class-distribution method requires an additional $\mathcal{O}(n)$ steps.

3.1.2 Top-Down methods

Since the performance of the network is measured by the accuracy on the test or validation set, it makes sense to develop methods that determine the weight's saliency based on the weight's direct influence on the error of the function. The most promising approach would be to try out all possible sparse representations of the network and take the best performing one. Unfortunately, there are 2^n sparse networks for an initial network with n weights. Therefore, the seek for the best network is $\mathcal{NP}$-hard and for an already small amount of weights it gets computationally completely unfeasible. For

modern neural networks, with several hundred thousand up to million weights, testing out all the possible solutions quickly becomes impossible.

Another approach that has already been used but gets unfeasible very quickly for relatively large networks is to delete each weight in isolation and then look which removed weight resulted in an acceptable change in the error function. Each pruning step requires $\mathcal{O}(n \cdot m)$ steps, where n is the number of parameters and m the time it takes to evaluate the network performance, but in each step only one weight can be deleted. Therefore, if a fixed percentage of elements should be pruned, $\mathcal{O}(n)$ pruning steps are required and therefore in total $\mathcal{O}(n^2 \cdot m)$ steps need to be executed. It is easy to see that this so-called *Oracle pruning* quickly gets unfeasible for modern neural networks with many parameters [Mol+16].

In order to calculate the saliency of the weights in a traceable and meaningful way, various techniques to approximate the change of the error have been developed. The main idea behind these approaches is to use a Taylor approximation to estimate the change in the cost function. The cost function of the network can be approximated as seen in section 2.2 and since the network is trained to a local minimum and the cost function is quadratic, the following equation is obtained for the change of the cost function by applying equation 2.24.

$$\delta E(\Delta x) = \frac{1}{2} \cdot \Delta x^T H_E \Delta x \tag{3.1}$$

Optimal Brain Damage (OBD), which has been introduced by LeCun et al., furthermore assumes that the Hessian matrix is diagonal [LDS90]. Therefore, the approximation of the change in the cost function from the previous shown equation further simplifies to the following:

$$\delta E(\Delta x) = \frac{1}{2} \sum_i [H_E]_{ii} \cdot \Delta x_i^2 \tag{3.2}$$

In equation 3.2, the term $[H_E]_{ii}$ stands for the diagonal entry of weight i in the Hessian matrix. The saliency for every parameter can therefore be modeled by the perturbation in which only the parameter itself is changed and the magnitude of the change is the negative of the current value. This results in the following formula for the saliency calculation for each parameter k:

$$s_k = \frac{[H_E]_{kk} \cdot x_k^2}{2} \tag{3.3}$$

For all other parameters, the perturbation is set to zero and therefore, their term vanishes from the sum. Consequently, only n saliencies have to be computed and the calculation of these can be done with the suggested algorithm by LeCun et al., which works similar to the already introduced backpropagation algorithm. Since for practical problems the Hessian Matrix has shown to be actually highly non-diagonal, a retraining is required after each of the pruning steps is executed.

Optimal Brain Surgeon (OBS) follows a similar idea as the previously introduced OBD method, but it doesn't assume the diagonality of the Hessian matrix and was introduced by Hassibi et al. [HSW93]. Therefore, it is more complex to compute the saliencies and for larger networks it is even unfeasible to use. The advantage of OBS is that the needed adjustments for the other weights of the network are calculated during the saliency calculation in this method as well and therefore it is not necessary to execute a retraining after the pruning is finished but rather the weights are adjusted based on the already computed inverse of the Hessian matrix. The pruning can be expressed as an optimization problem with a Lagrange condition.

$$L = \frac{1}{2} \cdot \Delta x^T H_E \Delta x + \lambda(e_q^T \cdot \Delta x + w_q) \tag{3.4}$$

In the upper equation, e_q^T corresponds to the unit vector in the weight space where the q-th value is one and all other values are zero. The optimization problem $min_q L$ can be solved using the Lagrange multiplier which leads to the following equations for the saliency calculation and the necessary changes, which are applied to the remaining weights:

$$\Delta x_q = \frac{-w_q}{[H_E^{-1}]_{qq}} \cdot H_E^{-1} \cdot e_q \tag{3.5}$$

$$L_q = \frac{1}{2} \frac{w_q^2}{[H_E^{-1}]_{qq}} \tag{3.6}$$

After the calculation, the lowest approximated change in the error rate L_q is selected to be pruned and the corresponding weight change Δx_q updates all the remaining weights in the network. The saliency calculation is nearly the same as in OBD, but instead of just setting the single weight that should be deleted to zero afterwards, the necessary change for every other weight is also calculated based on the inverse of the Hessian. Therefore, no retraining is executed.

The problem is the calculation of the inverse of the Hessian matrix which has to be re-computed in every single pruning step. This particularly is the biggest pitfall of OBS in

comparison to OBD, where it is possible, due to the assumption that the Hessian matrix is diagonal, to prune multiple elements in each pruning step without recomputing the saliencies that are based on the Hessian matrix. Although an own iterative approximation algorithm for the inverse calculation of the Hessian matrix has been developed by Hassibi et al., for modern neural networks with many parameters Optimal Brain Surgeon gets completely unaffordable in terms of computational time.

3.1.3 Layer-wise methods

Artificial neural networks are tending to get bigger and bigger, meaning the number of neurons and layers increases. Therefore, it quickly gets computationally unfeasible to calculate all the saliencies of the weights based on the output of the network as seen in the previous section. Instead of computing the saliency based on the output of the network, methods have been developed in order to calculate the saliency based on the output of the individual layers what reduces the complexity of the calculations dramatically.

Layer-wise Optimal Brain Surgeon (L-OBS) tries to transfer the idea of the previously introduced OBS algorithm to a layer-wise fashion [DCP17]. Therefore, the Hessian matrix is not calculated based on the final error of the network but instead calculated for each layer individually based on a newly introduced layer-wise error. The layer-wise error indicates how much the output has changed compared to the initially trained model. By keeping the layer-wise error that is caused by pruning as small as possible, the change in the final error also stays fairly small.

The calculations for the saliency and necessary weight updates are nearly identical to the ones already presented in the previous section for OBS. The only difference is that in L-OBS not the whole Hessian matrix is used but instead the layer-wise computation of the Hessian. The number of values that have to be computed and the size of the Hessian matrix therefore decreases dramatically, especially compared to the previously discussed OBS method. Thus, it is possible to use this method in deep neural networks that have a lot of layers like VGG-16, which contains up to 50 layers [SZ15].

The first gradient of the network's parameters can be calculated based on the layer-wise error. For fully-connected layers, which are surveyed in this thesis, the calculation of the Hessian matrix is approximated as matrix product from the first order gradient. Therefore, only the first order gradient has to be computed which is easily computable

with the previously introduced backpropagation algorithm. Furthermore, layer-wise OBS allows to prune multiple weights in each step. With these assumptions, every pruning step is only as expensive as backpropagation. An additional effort that has to be made in order to calculate the saliencies of the weights is that the original inputs have to be saved for each layer. Also the final inverse of the Hessian matrix for each layer has to be stored. However, since the Hessian matrix is only computed in a layer-wise fashion, the required storage is very limited [DCP17].

There are also other layer-wise pruning methods like Net-Trim, but since they make further assumptions like the linearity of the activation function, these will not be considered in this thesis [ANR16].

3.2 Pruning strategies

The number of elements that are pruned from the network in each step is a critical design decision of the pruning algorithm. If only a few elements are deleted, the network might have a better performance, but the model isn't compressed sufficiently. If too many weights are deleted, the opposite effect causes problems, meaning the performance might drop too much even though the network is highly compressed. Furthermore, since many of the previously introduced pruning methods need retraining after each pruning step is done, an additional rule for when and how long retraining is executed is needed.

The easiest and fastest way of pruning is to first delete a rather big number of elements from the network and in advance retrain the network, if necessary, for either a fixed number of retraining epochs, or make retraining depending on the validation error. This pruning strategy is called *single pruning* strategy, since it consists of a single pruning step followed by necessary retraining.

Another approach, which is referred to as *iterative pruning*, suggests that a number of weights is pruned from the network in many consecutive pruning steps [Mol+16]. Between each of the steps, retraining can be executed. In the mot basic form of iterative pruning, a fixed percentage of weights gets deleted in each pruning step. Plus, there is also the option to increase or decrease the pruning percentage between the pruning steps. The advantage of iterative pruning compared to single pruning strategy is that it might be easier to recover from the deletion of important weights because retraining

is executed earlier. The main disadvantage of this strategy is that a higher number of steps need to be executed in order to obtain a given compression rate and therefore, the pruning procedure takes more time.

A special case of increasing iterative pruning is the so-called *fixed number pruning*. In this strategy, a fixed total number of weights is deleted in every pruning step. Since the total number of weights of the pruned network decreases in every pruning step, the relative number of pruned elements increases in every step. The disadvantage of the strategy is that the minimum number of elements is bounded by $\#w \; mod \; \#p$, where $\#w$ is the initial number of weights in the network and $\#p$ is the used pruning rate, meaning the number of weights that are deleted in every pruning step. Therefore, if the pruning rate is rather big, it is not possible to achieve high compression rates. On the other hand, if the pruning rate is very small, it generally is possible to reach high compression rates, but a huge number of pruning steps is required.

The last approach that is considered in this thesis is the so-called *bucket pruning*. In this approach a number of weights is pruned until the total saliency sum of pruned weights reaches a predefined value. This is especially considerable if the saliency is a real measurement for the caused change in the error function for pruning methods like in OBD [LDS90]. Bucket pruning can be executed in either the previously introduced iterative way or in a single approach.

Pruning strategy	Pruning method
Iterative pruning	random, magnitude, OBD
Fixed number pruning	random, magnitude, OBD
Single pruning	random, magnitude, OBD, L-OBS
Bucket pruning	OBD, L-OBS

Table 3.1: Overview of the previously introduced pruning strategies and the pruning methods each strategy is used with.

In all the previously introduced pruning strategies that are listed in table 3.1, either a fixed number or a variable number of retraining epochs can be used after each pruning step if retraining is necessary. The number of used retraining epochs and the stopping criterion have a big influence on the final model as well.

A special case of pruning is executed in Optimal Brain Surgeon. This pruning method, which has previously introduced in section 3.1, recalculates the saliency for every single pruned weight in addition to the required change for all of the other weights [HSW93]. Therefore, this method does not need an additional strategy from this section, since it is already fixed in the method.

In order to obtain the best possible model from the pruning strategy a fine-tuning is often executed when the pruning and the normal retraining between the single pruning steps is finished. In this step, the model is retrained with smaller learning rates again which increases the chances for the cost function to reach a minimum. Furthermore, the fine-tuning can make up the information that is not regained during the normal retraining procedure within the pruning steps.

3.3 Implementation of sparse neural networks

The implementation of the pruning methods and strategies is done in the open source machine learning framework pytorch[1]. Since pytorch has no support for sparse neural networks and pruning yet our own solution had to be implemented on top of the existing framework.

A new type of linear layer, which contains in addition to the actual weights, a binary mask matrix m, whose entries indicate if the corresponding weight is still active or is set to zero by the pruning methods has been developed [SLM16]. This is important, since weights can also be regularly trained to have a magnitude of zero, but in contrast to pruned weights, it must be possible to change the magnitude during further training or retraining after the pruning for these weights. In every forward step, the mask matrix gets multiplied with the weight matrix using the Hadamard product. The forward path is therefore expressed as shown in equation 3.7, where $m^{(l)}$ describes the pruning mask from layer l.

$$y^{(l)} = \sigma((w^{(l)} \odot m^{(l)}) \cdot y^{(l-1)}) \tag{3.7}$$

Therefore, during the backward phase of the backpropagation algorithm, the gradient of the pruned weight will be zero as well, since the function that is described by the weight is now a constant, which vanishes in the derivation. On the other hand, if the value is one,

[1]pytorch.org

the function described by the additional computation is the identity, whose derivative is one. Using the chain rule, this derivative is multiplied with the other parts that have been introduced in section 2.1.3 and therefore the computation of the derivation remains correct. Therefore, the deleted weights don't change during the retraining process, while the still active weights are correctly updated.

The pitfall of this method is that in this implementation, the storage of the network is not optimized and, since there is hardware support for matrix multiplications, the actually deleted weights are multiplied as well. Therefore, neither the storage nor the number of floating point operations the network requires is decreased by this implementation.

4 Results

Initially, a small number of networks is trained until they reach a local minimum on the MNIST dataset[1]. Two different initial network architectures are used. The first architecture has one hidden layer with 100 hidden nodes in it which leads to an overall number of 79,000 weights. The second network consists of two hidden layers with 300 nodes and 100 nodes in the layers and therefore the number of parameter increases to about three hundred-thousand. The training process doesn't contain any further enhancements to prevent overfitting like weight decay or Dropout since these can interfere with the pruning results. Therefore, the initial performance of the trained network is about 97 percent. The used hyper-parameters are shown in table 4.1 and have been selected using manual search prior to the actual experiments.

Parameter	Used value
Network architecture	784-100-10 and 784-300-100-10
Loss function	Cross-Entropy-Loss with Softmax function
Activation function	Rectified linear unit (ReLU)
Epochs	Up to 200
Optimizer	Batch Gradient Descent
Batch Size	64
Learning Rate	0.01 - 0.0001

Table 4.1: The used hyper-parameters for the initially trained network selected using manual search.

In our experiments, in each execution a pre-trained model is loaded and then pruned with one of the pruning methods in combination with one of the pruning strategies that have been introduced in the previous chapter. After the pruning is done, a retraining phase

[1]http://yann.lecun.com/exdb/mnist/

is started if necessary and when the retraining is finished, the network's performance is measured. The duration of the retraining is based on the validation dataset, or a fixed number of epochs is used. As previously discussed, the type of applied retraining is part of the pruning strategy. The whole experiment process can be seen in figure 4.1.

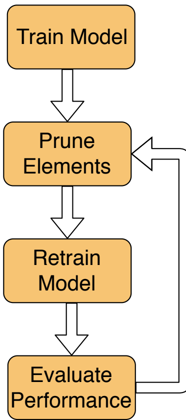
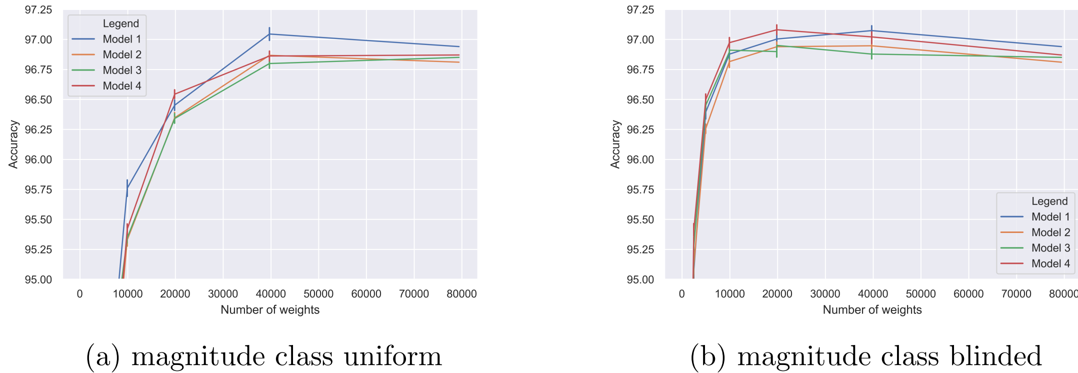


Figure 4.1: The experimental process

For the iterative pruning strategies, the process is repeated until the network is too small to prune any more weights or the accuracy has dropped significantly. As already mentioned, for the fixed number pruning, it becomes impossible to prune any more weights if the weight count is less than the number of elements that are pruned in each step. All the measured accuracies from the experiment process are saved in addition to the current number of non-pruned weights in the network. Subsequent to the experiments, the different pruning methods and pruning strategies that are examined can be compared with each other. Since the experiments rely on non-deterministic parts, e.g. for stochastic gradient descent, each experiment is repeated a number of times in order to achieve meaningful results. Based on the measured accuracies, it is now possible to determine which method produces the highest performing network and which one creates the model with the highest compression rate given a desired output accuracy. The architecture of good performing networks with a high compression rate is saved in order to later compare pruning techniques with sparse models that are trained from scratch. Furthermore, pruning in general can be compared with other regularization techniques in order to evaluate how suitable pruning techniques in general are to prevent overfitting.

For neural networks, the found local minimum in the cost function determines how good the network is performing. There are many different local minima that can be reached by the learning algorithm and lead to similar network performance as pointed out in section 2.1.2 [GBC16]. In the following section, the influence of the initially found local minimum on the pruning results will be examined. Therefore, four independent models have been trained to a local minimum and pruning was applied to each of them. The resulting accuracies with the corresponding number of weights of the pruned models can be seen in figure 4.2. The models are pruned using the magnitude class uniform method (4.2a) and the magnitude class blinded method (4.2b) in combination with the iterative pruning strategy, where 50 percent of the remaining weights are deleted from

the network in each pruning step. After each pruning step is executed, two fixed epochs of retraining are applied to the network. In order to have meaningful results, the pruning process is repeated 25 times from scratch for each of the initially trained models.



(a) magnitude class uniform

(b) magnitude class blinded

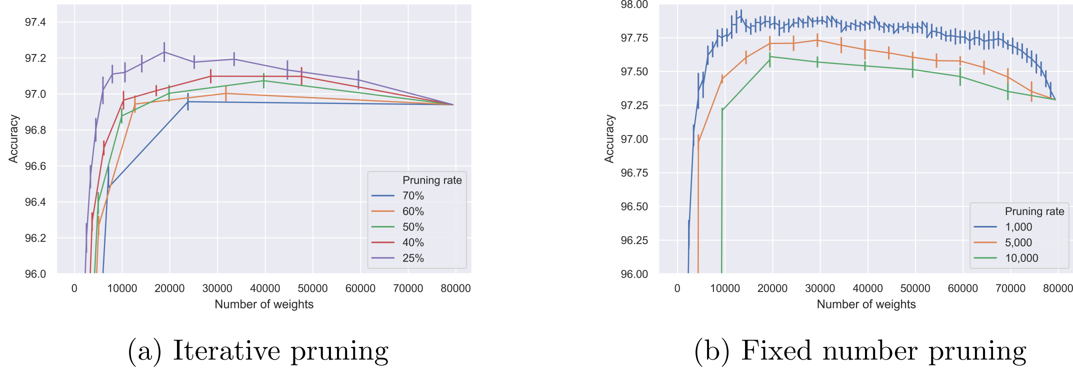
Figure 4.2: Comparison of the accuracy after pruning is applied on four different initial models for the magnitude class uniform and the magnitude class blinded pruning methods.

As you can see, the accuracy of the model doesn't decrease in the first pruning step and even for the following steps the accuracy doesn't decrease significantly using the magnitude class blinded method for all of the examined models. Instead, some of the models even have a higher accuracy once some of the weights are deleted from the network. This shows that pruning indeed is capable of preventing overfitting. Although the absolute accuracy of the initial models on the test set differ significantly from each other, the general trend is the same for all of the examined models and the accuracy drops in the same pruning step for the different models. Therefore, in the following, the experiments will be executed on a single initial model which allows to execute more repetitions in order to make the outcome of the experiments more significant. The difference in the two pruning methods that have been examined here as examples will be analyzed in detail in section 4.2.

4.1 Comparison of pruning strategies

First of all, we want to compare the different pruning strategies that have been introduced in section 3.2 with each other and examine how to choose the parameters for each strategy. Furthermore, the influence of the retraining process will be evaluated.

The number of elements that are deleted from the network in each step, the so-called pruning rate is a crucial decision for all pruning strategies. If the number is too small, it might take a long time until the network is highly compressed. On the other hand, if the pruning rate is fairly high, it might be more difficult to create high performing networks, since the network structure is damaged too much in the individual pruning steps.



(a) Iterative pruning

(b) Fixed number pruning

Figure 4.3: Comparison of the effect of different pruning rates for iterative pruning and fixed number pruning on the accuracy of the model using the magnitude class blinded pruning method.

In figure 4.3a, a range of pruning percentages for iterative pruning are compared with each other. Each pruning rate is tested 25 times in combination with the selected pruning method. Once again, the general trend of the accuracy is very similar for all percentages, while the actual performance differs between them. As expected, a smaller pruning rate also leads to a better performing network, since less information is deleted from the network and retraining is able to make up the lost information. For small pruning rates, the network's performance even keeps increasing until the network reaches a size of one-quarter of the original network.

Similar to the just examined pruning percentages for iterative pruning, the same variable needs to be tested on fixed number pruning. In figure 4.3b, the comparison of the pruning rates for the fixed number pruning strategy is shown. Similar to the iterative pruning, the smaller the pruning rate, the better the pruned network performs. Although the two evaluations from figure 4.3 are based on different initial models and therefore the absolute accuracy cannot be compared, it can be seen that both strategies behave in a similar way.

The main theoretical advantage of the iterative pruning strategy is that it combines the advantages of single pruning and fixed number pruning: It is possible to reach a high compression rate within a fairly small number of pruning steps but in each step only a considerably small number of weights are deleted from the network. Therefore, retraining is applied regularly and the network is able to regain the lost information just like in the fixed number pruning.

In figure 4.4, iterative and fixed number pruning are compared with each other. For the shown comparison iterative pruning with a pruning percentage of 25% and single pruning with a pruning rate of 5,000 is used. These numbers have been selected, since they allow us to generate a highly compressed model relatively quickly and the number of required pruning steps until 500 weights are reached is similar to each other: iterative pruning requiring 17 pruning steps and fixed number pruning requiring 15 steps using the introduced pruning rates. For each of the pruning strategies, the experiment is repeated 25 times in order to not only analyze the general performance but also the stability of the strategies.

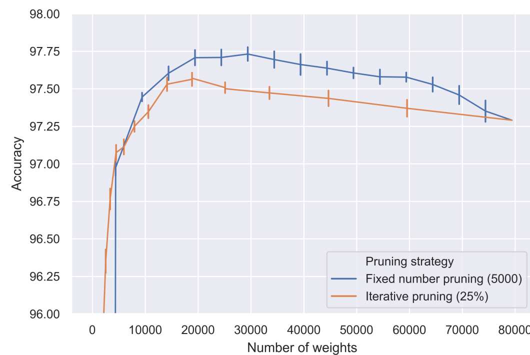


Figure 4.4: Comparison of fixed number pruning with iterative pruning using the magnitude class blinded pruning method and two fixed epochs of retraining after each pruning step.

As you can see, the fixed number pruning strategy clearly outperforms the iterative pruning strategy in nearly every single pruning step based on the overall performance of the outcoming network. Especially the generalization ability of fixed number pruning is a lot higher. The first time iterative pruning is capable of competing with fixed number pruning is when the accuracy has already dropped significantly. However, iterative pruning is capable of creating much smaller models. While the smallest model that

fixed number pruning can create still contains 4,900 weights, due to the theoretical limitations of the algorithm, iterative pruning is able to create models with only around 2,000 weights and a performance loss of only 1% compared to the original model. In terms of stability, both algorithms behave nearly the same and also, the variance doesn't increase for later pruning steps.

For more advanced pruning methods, the calculation of the saliency might take several hours and for big networks it is even possible that it takes days until the calculations are finished. Therefore, it is desirable to reduce the number of required saliency calculations to a minimum. The most extreme option that has been introduced is the single pruning strategy, where the saliencies of the weights only have to be calculated exactly once. In figure 4.5, the single pruning strategy is compared with iterative pruning and fixed number pruning. Single pruning is executed with a pruning rate of 85%. This leads to a model with around 12,000 weights. For fixed number pruning, a pruning rate of 5,000 weights is used and the displayed accuracy is measured after 14 pruning steps which leads to a model that contains 9,400 remaining weights. For the iterative pruning strategy, a rate of 25% is used and the accuracy is measured after 7 steps when the model has reached a weight count of around 10,500. Each pruning strategy is tested 25 times. For fixed number pruning and iterative pruning, two fixed epochs of retraining are applied like it has been done in the previous experiments. However, for single pruning, the loss that is caused by pruning is a lot higher and therefore more retraining is required. Thus, variable retraining with a maximum of 20 epochs is applied for the single strategy.

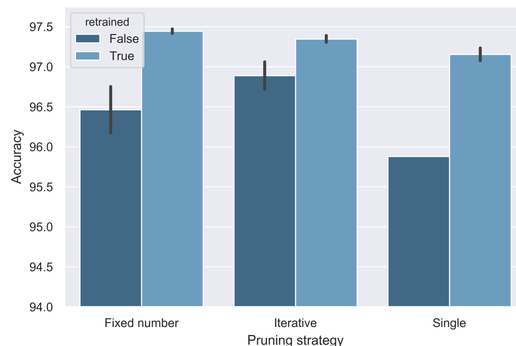


Figure 4.5: Comparison of the single pruning strategy with iterative pruning and fixed number pruning. All evaluated models had a weight count around 10,000. For this experiment the magnitude class blinded pruning method is used. The network performance is evaluated before and after the retraining was applied to the model.

As previously seen, the fixed number pruning strategy outperforms iterative pruning after retraining is applied. However, the initial performance of the network after pruning is better for the iterative pruning strategy, but in comparison to this strategy, fixed number pruning is capable of regaining more of the lost information during the retraining process. As expected, the two strategies outperform single pruning, since in each pruning step they only have a limited loss and with the occasional retraining, the network is able to regain the lost information better. In the single strategy, the non-retrained model obviously always has the same initial accuracy, since the pruning is executed in a deterministic way. Nevertheless, the final accuracy of the single pruning strategy has the highest variation. Due to the large amount of retraining, the used gradient descent algorithm is not always able to find a satisfying minimum of the cost function. In contrast to this, in iterative and fixed number pruning the initially pruned models vary quite a lot, but the final accuracy is much more stable, since a fixed number of retraining epochs is always applied.

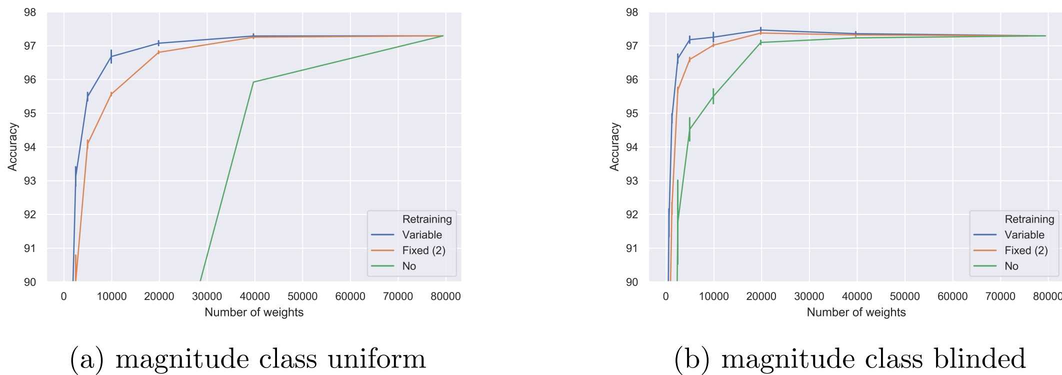
However, the difference between the examined pruning strategies is relatively small and the single strategy is a good choice for methods where it is computationally hard to calculate the saliencies or it requires a lot more programming effort to calculate the saliency for sparse models, which is the case for methods like layer-wise OBS. However, the variable retraining after the pruning that is executed leads to a lower stability of the pruning procedure, but a single pruned sparse model might get retrained various times in order find a satisfying minimum in the loss function similar to the repeated training of the initial model.

The remaining bucket pruning strategy behaves very similarly to the already examined single strategy. The only actual difference is that instead of testing out various pruning rates which indicate the number of elements that are pruned, in bucket pruning the pruning rate indicates the expected minimum loss in accuracy. However, it only makes sense to use this strategy with a very limited number of pruning methods.

Previous experiments have shown that two epochs of retraining already restore a high fraction of the lost information during pruning. Furthermore, if the pruning rate is small enough, another retraining phase will follow in a short amount of time. In the following paragraph we will examine the influence of more retraining on the network. Therefore, instead of using two fixed epochs of retraining, variable retraining is applied which allows the network to recover the lost information better. The retraining will be stopped based

on the error on the validation set. If the accuracy on the validation set stops increasing, retraining is stopped, otherwise it is continued up to a maximum of 10 epochs.

The two pruning procedures that are shown in figure 4.6 use iterative pruning with a pruning percentage of 50% in each pruning step. The experiment is repeated 25 times due to the non-deterministic behavior of the algorithm.



(a) magnitude class uniform

(b) magnitude class blinded

Figure 4.6: Comparison of the effect of the variable and two epochs fixed retraining on the accuracy of the model with different pruning methods. As a baseline the non-retrained accuracy is shown.

As expected, the accuracy is a bit higher if variable retraining is applied. With variable retraining, the network is able to react better to the lost information and prevent overfitting as well as underfitting during the retraining. However, the general progression of the accuracy of the retrained network remains the same for both fixed and variable retraining and compared to the non-retrained version of the network, both of the approaches make up most of the due to pruning lost information. The advantage of fixed retraining is the limited and expectable time consumption in comparison to the variable retraining process. Furthermore, it is possible to add a fine-tuning to the end of the process that is displayed in figure 4.1 which might be able to reduce the gap between the two examined retraining approaches.

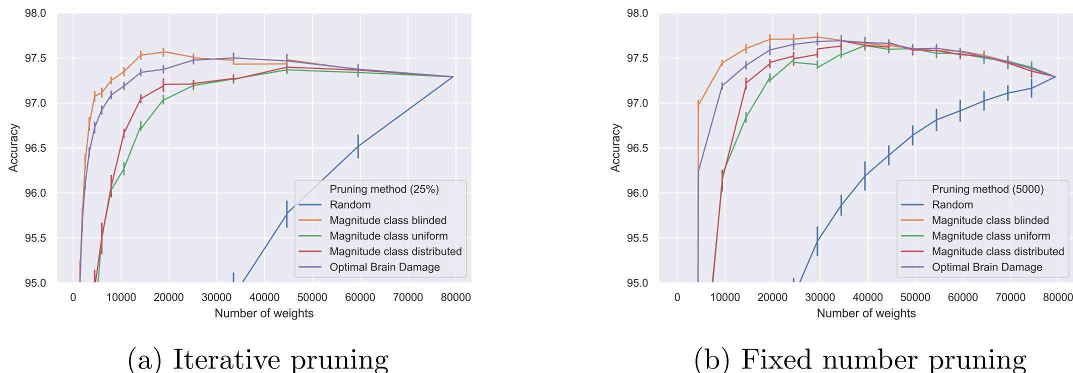
4.2 Comparison of pruning methods

As seen in the previous chapter, the different pruning strategies pretty much behave as we would expect them to. Therefore, the choice of the pruning strategy highly depends

on the available time and computing power as well as the desired outcome. In this chapter we want to compare the different used pruning methods in order to determine which one is able to produce the best performing and most compressed networks.

The network's performance is measured after retraining as well as before the retraining takes place. Therefore, we can compare the performance of the pruning methods for both: before and after the retraining took place. As previous seen, two fixed retraining epochs are able to restore most of the information that has been lost during pruning. Furthermore, fixed retraining is less time-consuming and thus the pruning procedure gets sped up. Therefore, in the following, we will use the fixed retraining approach. As a baseline for all of the method comparisons random pruning is used. If the accuracy of the final model should be improved, a fine tuning step can be applied after the pruning is done. In this step, the model is retrained with a decreasing learning rate again.

First of all, we want we compare the different pruning methods using the fixed number pruning strategy and the iterative pruning strategy. Therefore, the initial model is pruned using the different methods and the whole pruning process is repeated 25 times. As we have seen in the previous section, the smaller the pruning rate, the higher the performance of the created network. However, a compromise between runtime of the pruning procedure and the outcoming performance needs to be made. Therefore, in the following, we will use pruning rates between 1,000 and 10,000 for fixed number pruning and between 25% and 50% for iterative pruning, since these create good performing models in an acceptable number of pruning steps.

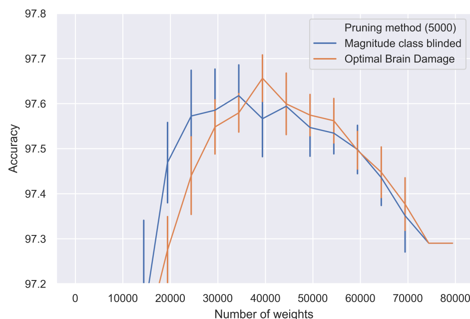


(a) Iterative pruning (b) Fixed number pruning

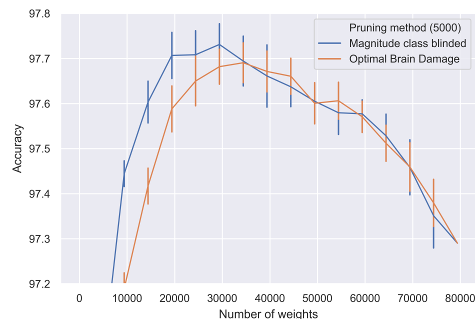
Figure 4.7: Comparison of the different used pruning methods in combination with fixed number pruning with a pruning rate of 5,000 and iterative pruning with a pruning rate of 25%.

In figure 4.7, we can see the differences between the five pruning methods that are used for fixed number pruning: magnitude class blinded, magnitude class distributed, magnitude class uniform, Optimal Brain Damage, and as a baseline comparison random pruning. In figure 4.7a, the different pruning methods are tested using the iterative pruning strategy and in 4.7b the fixed number pruning strategy is used. As expected, the random pruning method comes of as the worst of the examined methods for both of the examined strategies. Interestingly, magnitude class blinded pruning outperforms the other two magnitude based pruning methods clearly and is at least as good as Optimal Brain Damage, which requires a way higher effort to calculate its saliencies. For higher compression rates, it even outperforms Optimal Brain Damage clearly. Furthermore it is noteworthy that the two methods start to move apart when the network reaches about 30,000 weights. In order to further analyze the two just mentioned best performing pruning methods, they are compared directly with each other in figure 4.8.

In figure 4.8b, you can see that during the first few pruning steps the two methods perform nearly the same, but especially in the first pruning step OBD performs slightly better. After a few pruning steps, OBD isn't able to improve the accuracy of the network any more, in contrast to the magnitude method. In contrast to the nearly same performance of the two examined pruning methods after the retraining, the untrained performance of the model can be compared in figure 4.8a. In this case, Optimal Brain Damage clearly outperforms the magnitude based approach. Thus, although the initial performance of OBD after the pruning is better, the method is not able to regain as much information from the retraining procedure as the magnitude based approach is.



(a) Before retraining is applied



(b) After retraining is applied

Figure 4.8: Comparison of Optimal Brain Damage and magnitude class blinded pruning methods in combination with the fixed number pruning strategy using a pruning rate of 5,000. The methods are compared before and after the retraining is applied.

Layer-wise OBS was only tested using single pruning, since the calculation of the inverse of the Hessian matrix is based on heuristic algorithms that only work for full matrices. Therefore, the reference implementation of the algorithm[2] could be used [DCP17]. In figure 4.9, L-OBS is compared with the other examined pruning methods. For this experiment, a pruning rate of 80% is used. The network's performance is measured before the retraining took place as well as after the retraining is executed. The number of retraining epochs is based on the error on the validation set, thus early stopping is used, but the total number of epochs is limited to 25.

You can see that the magnitude class blinded method and Optimal Brain Damage only suffer a very small loss due to pruning that can even be further reduced by the applying retraining. In comparison, the magnitude class uniform, magnitude class distributed, and L-OBS suffer a way bigger change in the error due to pruning. Interestingly the two magnitude based approaches are able to recover from the loss pretty well and regain an accuracy of around 95%. In contrast, L-OBS is not able to recover from the pruning procedure and fairly reaches an accuracy of 93% after retraining is applied, although the performance before the retraining is applied is higher than in the two magnitude based approaches. Although the initial performance of the network that is pruned using L-OBS is nearly three times as high as the one created using random pruning, the post-retrained randomly created model outperforms the other one that uses a way more advanced technique.

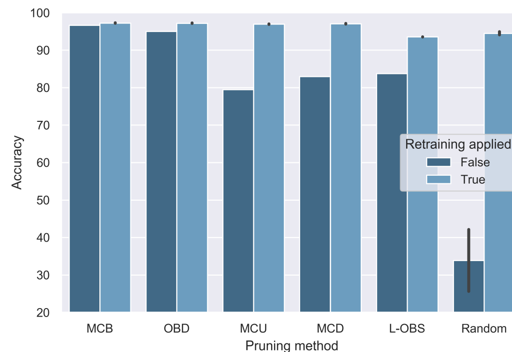


Figure 4.9: Comparison of the accuracy of magnitude class blinded (MCB), magnitude class distributed (MCD), magnitude class uniform (MCU), Optimal Brain Damage (OBD), layer-wise OBS (L-OBS), and random pruning methods in combination with the single pruning strategy before and after the retraining is applied.

[2]https://github.com/csyhhu/L-OBS

4.3 Baseline experiment

As already seen, pruning techniques are able to generate models that are highly compressed but still have a very high accuracy on the test dataset. In this section, we will have a look at how pruning techniques are able to compete with other regularization techniques and other kind of techniques to generate sparse models.

The accuracies of the models that have been generated with the pruning techniques will be compared with sparse models that are trained from scratch. The best performing model that has been created by a pruning algorithm was generated using the fixed number pruning strategy with a pruning rate of 1,000 in combination with the magnitude class blinded pruning method. This pruned model has 13,400 weights which corresponds to 16.8% of the original weight count. The accuracy of the original model before the pruning is 97.29% and the pruned version has an accuracy of 98.01% before fine-tuning is applied, which corresponds to a raise in the accuracy of 0.72%.

In figure 4.10, the accuracy of the previously introduced model that is created using pruning is compared with the original model and other created models. For each of the tested models, the network was trained 10 times and the best result is shown in the figure. In order to further improve the generalization performance, early stopping with a decreasing learning rate is used.

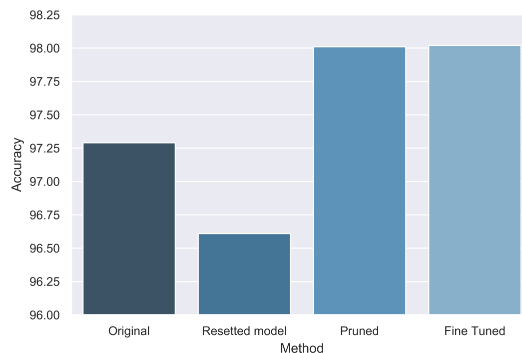


Figure 4.10: Comparison of the model's accuracy in different settings: the original non-pruned model, the pruned model, the fine-tuned version of the pruned model and the created sparse model trained from scratch.

As you can see, pruning is indeed capable of reducing the overfitting that was contained in the original model. Furthermore, the pruned model only contains 13,400 weights

which corresponds to a compression rate of nearly 85% compared to the original model. However, if we train a sparse model with the same network structure as the created one, the accuracy of this model is significantly lower than both: the original model and the one created by pruning. This shows that pruning is capable of creating highly compromised models that still have a very high accuracy. It is not possible to create these models directly.

By applying a fine-tuning step, the accuracy of the model does not really change and the best achieved performance is 98.02%. This shows that although a small fixed number of pruning steps, in our case only two epochs, is used, the pruning procedure is still able to find a very good minimum in the cost function. Thus the network doesn't require further improvement.

In figure 4.11, the pruned model is compared with other popular regularization techniques. The models are trained using early stopping in combination with a decreasing learning rate over time in order to also achieve a good generalization performance. Each of the techniques is executed 10 times and the best performing model is shown in the graph.

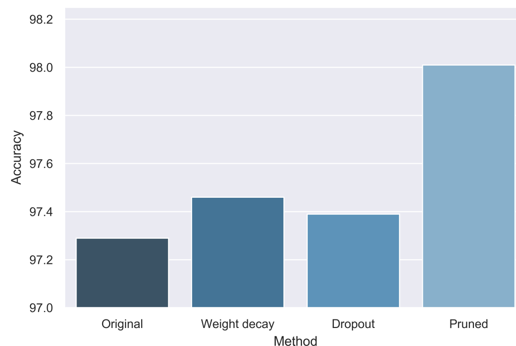
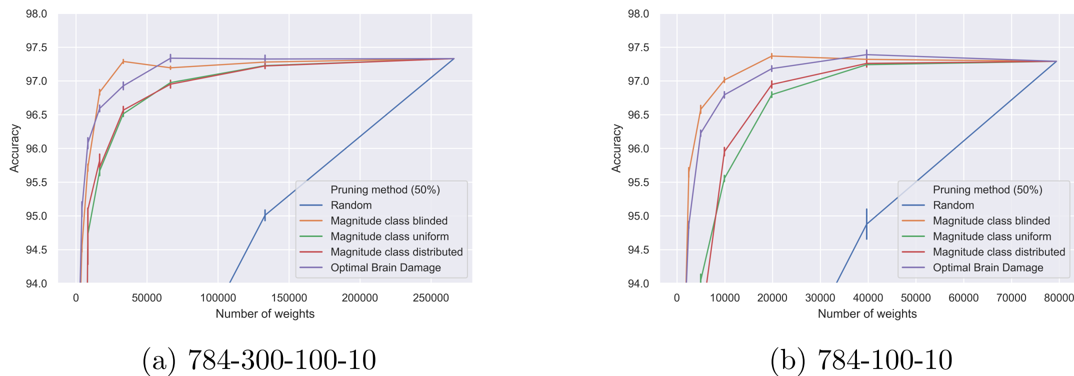


Figure 4.11: Comparison of the original and pruned model with models created using the popular regularization techniques weight decay and Dropout. The goal of all three compared techniques is to improve the generalization performance of the network.

For the $\mathcal{L}_2$ regularization technique weight decay, a λ value of 0.0008 is used and a Dropout layer is added to the hidden one with a drop probability of 0.5. The values have been selected using a combination of grid search and manual search. The same initial architecture and the same hyper-parameters for training as described previously

in table 4.1 are used. As you can see, the two examined regularization techniques are both able to create a network that has a higher accuracy than the originally created one, but both examined methods don't reach the level of performance that was reached by pruning. This shows us that pruning has a very high regularization ability, while also creating a highly compressed model.

All the previous experiments have been executed on a network with initially 79,400 weights. In order achieve meaningful results, various versions of iterative pruning are also executed on a network that has a different initial architecture. The architectures of the models have already been described in the introduction to this chapter and are summed up in table 4.1. The new model contains two hidden layers with 300 and 100 neurons in each of them. Therefore, the initial number of weights increases to nearly 300,000. With more weights in the initial network, the redundancy of the information might also be higher; therefore, it is possible to achieve higher compression rates. Thus, the comparison of the architectures need to be based on the actual number of weights in the created networks. In figure 4.12, the accuracies of the two introduced architectures are compared with each other. Each model is pruned ten times with each of the pruning methods and after each pruning step two epochs of retraining are applied.



(a) 784-300-100-10

(b) 784-100-10

Figure 4.12: Comparison of the performance of the two used architectures using iterative pruning with a pruning rate of 50% and a number of different pruning methods. The smaller model (b) has initially 79,400 weight while the bigger model (a) contains initially 266,200 weights.

In figure 4.12b, the performance of the smaller model that was also explored in the last sections is shown. In comparison, the performance of the bigger model with two hidden layers is shown in figure 4.12a. Each model is pruned using the different shown pruning

methods in combination with the iterative pruning strategy with a pruning rate of 50%. As you can see, the general behavior of the pruning methods stay the same for both of the examined architectures.

Furthermore, since the bigger model contains more total weights, there also are more redundant ones in the network. Therefore, it is possible to delete more total weights without a significant loss in performance. In both cases, the performance drops between the second and fourth pruning step, but since in iterative pruning the total amount of pruned elements is depending on the total weight count, in the bigger model a much higher compression rate can be achieved. Unsurprisingly, random pruning is not able to create a good performing model because during pruning a lot of important weights are deleted. In the bigger model, Optimal Brain Damage has a very clear advantage during the first few steps but then falls short compared to the magnitude class blinded method. The performance of the other two magnitude based methods nearly stays the same, but while in the smaller model the magnitude class distributed method has a slight advantage in the new model they nearly perform the same way. Compared to the previously seen experiments, the stability of the pruning methods seems higher, however this is only caused by the lower number of repetitions.

5 Discussion

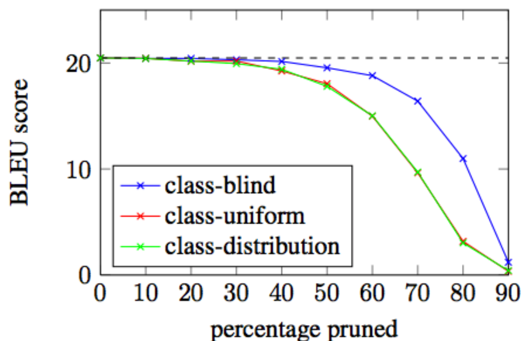
In the previous chapter the outcome of the executed experiments has been presented. In the following sections the results from the previous chapter will be discussed. First of all the different compared pruning methods and pruning strategies which are summed up as pruning techniques will be discussed and in the following pruning as a regularization technique will be evaluated.

5.1 Comparison of pruning techniques

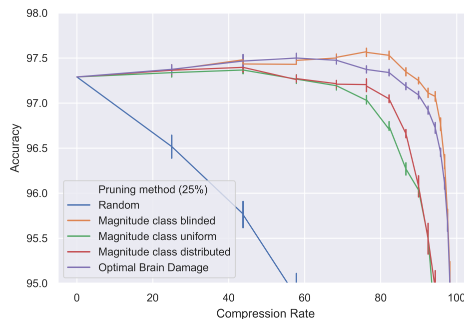
As seen in the previous chapter all the pruning methods that have been tested clearly outperform the baseline random pruning method. Therefore, all the presented pruning methods indeed find weights that can be deleted without gaining a significant loss in performance.

Previous research has already compared the different magnitude based approaches that are also compared in this thesis. In figure 5.1 our achieved results are compared with the results published by See et al. [SLM16]. In figure 5.1b the results from the previous chapter are shown. The graph shows the accuracy for the model that was pruned using the iterative pruning strategy with a pruning rate of 25%. Instead of using the total number of weights on the x-axis the compression rate is used which is just the current number of weights divided by the original number of weights in order to make the results comparable. As you can see in both of the experiment series the magnitude class blinded pruning method outperforms the other ones. However, in the results published by See et al. magnitude class uniform and magnitude class distributed perform nearly the same way, while in our experiments magnitude class distributed performs a little bit better than the other approach. The reason for this might be that we used a different initial model and more repetitions of the experiment. Furthermore, the saliency calculation of

magnitude class distributed is more advanced and therefore it seems to better select the deletable weights. Moreover, in the results achieved by See et al. that are displayed in figure 5.1a the single pruning strategy is used. As we have seen in the previous chapter the pruning can be improved significantly by applying iterative pruning with e.g. a fixed number or a fixed percentage of elements deleted in each pruning step. This allows to create much higher performing networks since retraining is applied more regularly. Therefore, our network is able to increase the overall performance of the network and thus reduce the overfitting in the network, the network pruned by See et al. is not. Another reason for the different performance might be the different initial architecture since in our experiments fully-connected networks are surveyed in contrast to the RNNs that are surveyed by See et al. Due to the different architectures of the networks and the more efficient pruning strategy in our experiments we are able to create higher compressed networks as well as reduce overfitting of the network. However, the general behavior of the surveyed pruning methods is the same in both experiments [SLM16].



(a) Results from previous research [SLM16].



(b) Our own results

Figure 5.1: Comparison of the results from previous research on pruning in RNNs and our own experiments on feed-forward networks.

Optimal Brain Damage is able to compete with the magnitude class blinded method in the first few pruning steps, but when the network contains 30,000 weights the accuracy of the model pruned with OBD drops significantly compared to the one which is pruned by the magnitude class blinded method. A reason for this might be that due to the limited retraining the network has not yet reached a local minimum and therefore the approximations that are made during the calculation of the OBD saliency are wrong and

thus wrong weights are deleted. Nevertheless, magnitude based pruning performs very well and the advantage over OBD even increases when you have a look at the runtime of each of the pruning steps. While each pruning step for magnitude based pruning requires around 60 seconds each step for OBD requires nearly 1200 seconds which is 20 times more including the two epochs of retraining. However, the saliency calculation can be sped up significantly by implementing OBD in the core of the framework instead of implementing it on top of it as we did [Mol+16]. Furthermore, there is also the possibility to not calculate the diagonal of the Hessian matrix directly but to use approximation techniques [MSS12]. The disadvantage of calculating the Hessian using approximations is that the accuracy of the saliency diminishes. Therefore, it is very likely that this approximated pruning method performs worse than the ones we examined. Therefore, a trade-off between runtime and performance of the algorithm is needed.

Another reason for the better performance of OBD and magnitude class blinded pruning compared to the other approaches is that in these first two there is only a single weight ranking from which the weights are deleted while in other approaches for each of the single layers a own ranking is created and then from each ranking weights are deleted more or less independently. However, it is highly unlikely that every layer contains the same amount of redundancies. Therefore, it actually might be necessary to test out a pruning rate for each of the single layers in order to examine how many weights can be deleted from this layer. Some previous research already used varying pruning rates for different layers which might explain the performance improvement they achieved compared to our results, for example in the L-OBS method [DCP17], whose implementation[1] contained pruning rates between 30% and 93% in one single model with two hidden layers. However, the used pruning rates are manually selected what requires additional manual effort. Therefore, further research is required for this case: it is not sufficient to know that the network itself has redundancies but each layer's redundancy needs to be measured in order to create the best possible model.

Deleting the weights that cause the smallest change in the error function might not always be necessary since retraining is required in nearly all cases. Therefore, if some weights are deleted the network might have the possibility to reach another minimum in the error function that provides a better accuracy than the previously reached one. This is also highly depending on the initially reached minimum in the cost function. This can be seen in the previous chapter especially in figure 4.9, where the randomly

[1] https://github.com/csyhhu/L-OBS

pruned network has the highest loss initially caused by pruning, but after the retraining the method outperforms some of the other methods.

Summarizing the previously seen results from the experiments the magnitude class blinded method is the method that is able to create the highest performing networks in our survey. Optimal Brain Damage is also able to create very good performing networks which especially compete with the other ones in the first few pruning steps. However, if we take the additional computational effort that is required in OBD into account as well as the additional programming effort the magnitude class blinded approach clearly wins.

The iterative pruning strategies clearly outperform the single pruning strategy. For the iterative pruning strategies always a trade-off between runtime and performance of the outcoming network has to be made when the pruning rate is chosen. Furthermore, depending on the goal of the pruning the actual strategy can be chosen. Fixed number pruning is able to create the highest performing networks and outperformed the standard iterative pruning. However, if very high compression rates are aimed for iterative pruning comes out ahead of fixed number pruning. However, it is also possible to change the pruning rates in a more advanced way than we did in order to even further improve the pruning process.

5.2 Pruning as regularization technique

As we have seen in chapter 4.3 the pruning procedure is indeed capable of reducing the overfitting in the network and therefore is a good regularization technique. However, the pruning comes at a very high price. While weight decay and also other $\mathcal{L}_1$ and $\mathcal{L}_2$ regularization techniques as well as Dropout can be applied during the initial training the pruning procedure needs to be applied after a training has been finished.

Therefore, the additional effort that is required by pruning needs to be added to the required initial training. In order to create the model that has been examined in the previous chapter 66 pruning steps were necessary and each pruning step not only consisted of a ranking of all the weights but also of two epochs of retraining. Thus, in total 132 epochs of retraining have been applied to the network during the retraining. Additionally the initial network had to be trained using 20 epochs. In contrast weight decay

and Dropout both required less than 25 epochs in order to improve the performance of the network. However, during the pruning process many different models are created and thus the computational effort pays off.

6 Conclusion

In this thesis we have seen that pruning techniques in general are capable of creating highly compressed networks which still have a very high accuracy. Pruning techniques are not only able to create highly compressed networks they also prevent overfitting by deleting the redundant weights that exist in the network as previous theoretical research has shown [Den+13]. Compared to other popular regularization methods pruning performs pretty well but the required additional computations are much higher than for other techniques. However, pruning was not tested in combination with other regularization techniques.

For the examined pruning strategies there always has to be made a trade-off between a high pruning rate which leads to a better result model and the runtime of the pruning technique which decreases for higher pruning rates. However, the iterative pruning techniques perform a lot better than the single pruning strategy as you would expect. The examined pruning methods have shown they are able to find the weights which only have a small influence on the error, but especially the more advanced methods like OBD are computationally very expensive and do not lead to a huge advantage in the result model compared to primitive pruning methods like the magnitude class blinded method. Furthermore, some of the advanced methods are even too computationally expensive to test out their performance like it is the case for OBS and require a huge amount of additional effort to implement them. By deleting actually important weights it might also be possible that the network is able to find a better minimum than the previous one during retraining which is required in nearly all cases.

In the future the calculation and usage of sparse networks need to be improved and existing methods need to be integrated. This will lead to the desired reduction in storage and required FLOPs during the calculations. Examined algorithms like OBD need to be implemented in a more efficient way which will lead to a huge speed-up and makes them more usable for real-world problems. Although it has shown in both theory and

practice that neural networks contain many redundant weights the redundancy is most likely not distributed uniformly over the network. Therefore, it is necessary to evaluate how much redundancy is in each layer of the network and therefrom how many weights can be deleted from each of them.

Furthermore, pruning techniques need to be evaluated in addition to other examined regularization techniques like weight decay and Dropout in order to improve the overall accuracy of the generated networks. The different examined pruning techniques have been tested on a very small network compared to the current state of the art networks for more advanced datasets like ImageNet[1]. The pruning techniques need to be evaluated on the bigger and more advanced networks in the future as well. Pure feed-forward networks are used very rarely in modern networks and therefore pruning also has to be evaluated on other network types like CNNs and RNNs where other techniques like ThiNet already exist and existing methods can be expanded [LWL17].

[1]http://www.image-net.org/

Bibliography

[Agg18] C. C. Aggarwal. *Neural Networks and Deep Learning - A textbook*. Springer, 2018 (cit. on pp. 2, 7, 10, 12, 13, 15–19).

[ANR16] A. Aghasi, N. Nguyen, and J. Romberg. "Net-Trim: A Layer-wise Convex Pruning of Deep Neural Networks". In: *Computing Research Repository* (Nov. 2016) (cit. on p. 28).

[Ama+13] F. Amato et al. "Artificial neural networks in medical diagnosis". In: *Journal of Applied Biomedicine* 11 (Dec. 2013), pp. 47–58 (cit. on p. 5).

[Bar97] P. L. Bartlett. *For Valid Generalization, the Size of the Weights is More Important Than the Size of the Network*. 1997 (cit. on p. 24).

[BH89] E. Baum and D. Haussler. "What Size Net Gives Valid Generalization?" In: *Neural Computation* 1.1 (1989), pp. 151–160 (cit. on p. 1).

[BB12] J. Bergstra and Y. Bengio. "Random Search for Hyper-parameter Optimization". In: *Journal of Machine Learning Research* 13.1 (Feb. 2012), pp. 281–305 (cit. on p. 9).

[Bur88] D.J. Burr. "Experiments on neural net recognition of spoken and written text". In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 36.7 (July 1988), pp. 1162–1168 (cit. on p. 1).

[Cha89] Y. Chauvin. "A Back-propagation Algorithm with Optimal Use of Hidden Units". In: *Advances in Neural Information Processing Systems 1*. Ed. by David S. Touretzky. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1989, pp. 519–526 (cit. on p. 19).

[Den+13] M. Denil et al. "Predicting Parameters in Deep Learning". In: *Computing Research Repository* (June 2013) (cit. on pp. 1, 22, 52).

[DCP17] X. Dong, S. Chen, and S. J. Pan. "Learning to Prune Deep Neural Networks via Layer-wise Optimal Brain Surgeon". In: *Computing Research Repository* (2017) (cit. on pp. 27, 28, 42, 49).

[Fle00] R. Fletcher. *Practical Methods of Optimization*. John Wiley & Sons, Ltd, May 2000 (cit. on p. 21).

[GBC16] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016 (cit. on pp. 8, 10–12, 17, 18, 33).

[GMH13] A. Graves, A. Mohamed, and G. E. Hinton. "Speech Recognition with Deep Recurrent Neural Networks". In: *Computing Research Repository* (Mar. 2013) (cit. on p. 5).

[Han+15] S. Han et al. "Learning both Weights and Connections for Efficient Neural Networks". In: *Neural Information Processing Systems* (2015) (cit. on p. 24).

[HP89] S. J. Hanson and L. Y. Pratt. "Comparing Biases for Minimal Network Construction with Back-Propagation". In: *Advances in Neural Information Processing Systems 1*. Ed. by D. S. Touretzky. Morgan-Kaufmann, 1989, pp. 177–185 (cit. on pp. 3, 19).

[HSW93] B. Hassibi, D. G. Stork, and G. J. Wolff. "Optimal Brain Surgeon and general network pruning". In: *IEEE International Conference on Neural Networks*. Vol. 1. IEEE. Mar. 1993, pp. 293–299 (cit. on pp. 26, 30).

[Haw04] D. M. Hawkins. "The Problem of Overfitting". In: *Journal of Chemical Information and Computer Sciences* 44.1 (Jan. 2004), pp. 1–12 (cit. on p. 18).

[He+16] K. He et al. "Deep Residual Learning for Image Recognition". In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016 (cit. on p. 8).

[Hol13] M. H. Holmes. *Introduction to Perturbation Methods*. Springer New York, 2013 (cit. on p. 20).

[Hug58] J. R. Hughes. "Post-Tetanic Potentiation". In: *Physiological Reviews* 38.1 (Jan. 1958), pp. 91–113 (cit. on p. 4).

[Hut18] M. Hutson. "Has artificial intelligence become alchemy?" In: *Science* 360.6388 (May 2018), pp. 478–478 (cit. on p. 12).

[KSH12] A. Krizhevsky, I. Sutskever, and G. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 1097–1105 (cit. on p. 5).

[LB98] Y. LeCun and Y. Bengio. "Convolutional Networks for Images, Speech, and Time Series". In: *The Handbook of Brain Theory and Neural Networks*. Ed. by Michael A. Arbib. Cambridge, MA, USA: MIT Press, 1998, pp. 255–258 (cit. on p. 10).

[LDS90] Y. LeCun, J. S. Denker, and S. A. Solla. "Optimal Brain Damage". In: *Advances in Neural Information Processing Systems 2*. Ed. by D. S. Touretzky. Morgan-Kaufmann, 1990, pp. 598–605 (cit. on pp. 25, 29).

[LeC+98] Y. LeCun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE*. 1998, pp. 2278–2324 (cit. on p. 1).

[LWK18] C. Louizos, M. Welling, and D. P. Kingma. "Learning Sparse Neural Networks through L_0 Regularization". In: *International Conference on Learning Representations*. 2018 (cit. on p. 19).

[LWL17] J. Luo, J. Wu, and W. Lin. "ThiNet: A Filter Level Pruning Method for Deep Neural Network Compression". In: *The IEEE International Conference on Computer Vision (ICCV)*. Oct. 2017 (cit. on p. 53).

[Mar+18] A. Marchisio et al. "PruNet: Class-Blind Pruning Method For Deep Neural Networks". In: *International Joint Conference on Neural Networks (IJCNN)*. July 2018, pp. 1–8 (cit. on p. 24).

[MSS12] J. Martens, I. Sutskever, and K. Swersky. "Estimating the Hessian by Backpropagating Curvature". In: *International Conference on Machine Learning*. Vol. 2. June 2012 (cit. on p. 49).

[Mer] Merriam-Webster Online. *Merriam-Webster Online Dictionary*. Accessed: 2019-03-09. URL: `http://www.merriam-webster.com` (cit. on p. 4).

[MTH89] G. F. Miller, P. M. Todd, and S. U. Hegde. "Designing Neural Networks Using Genetic Algorithms". In: *Proceedings of the Third International Conference on Genetic Algorithms*. George Mason University, USA: Morgan Kaufmann Publishers Inc., 1989, pp. 379–384 (cit. on p. 19).

[Mol+16] P. Molchanov et al. “Pruning Convolutional Neural Networks for Resource Efficient Inference”. In: *Computing Research Repository* (Nov. 2016) (cit. on pp. 25, 28, 49).

[Nie15] M. A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015 (cit. on pp. 5–15).

[NH92] S. Nowlan and G. Hinton. “Simplifying Neural Networks by Soft Weight-Sharing”. In: *Neural Computation* 4.4 (July 1992), pp. 473–493 (cit. on p. 19).

[Pat97] D. W. Patterson. *Künstliche neuronale Netze. Das Lehrbuch*. Prentice Hall, 1997 (cit. on p. 12).

[Pre98] L. Prechelt. “Neural Networks: Tricks of the Trade”. In: *Neural Networks: Tricks of the Trade*. Ed. by Genevieve B. Orr and Klaus-Robert Müller. Springer Berlin Heidelberg, 1998. Chap. Early Stopping - But When?, pp. 55–69 (cit. on p. 19).

[Ras17] T. Rashid. *Neuronale Netze selbst programmieren*. O’Reily, 2017 (cit. on p. 15).

[Ree93] R. Reed. “Pruning algorithms-a survey”. In: *IEEE Transactions on Neural Networks* 4.5 (Sept. 1993), pp. 740–747 (cit. on pp. 2, 19).

[Ros58] F. Rosenblatt. “The perceptron: A probabilistic model for information storage and organization in the brain.” In: *Psychological Review* 65.6 (1958), pp. 386–408 (cit. on p. 5).

[RHW86] D. Rumelhart, G. Hinton, and R. Williams. “Learning representations by back-propagating errors”. In: *Nature* 323.6088 (Oct. 1986), pp. 533–536 (cit. on pp. 1, 13).

[RN09] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Third. Upper Saddle River, NJ, USA: Pearson, 2009 (cit. on pp. 4, 5).

[SLM16] A. See, M. Luong, and C. D. Manning. “Compression of Neural Machine Translation Models via Pruning”. In: *Conference on Computational Natural Language Learning* (2016) (cit. on pp. 24, 30, 47, 48).

[Shi+16] S. Shin et al. “A New Robust Design Method Using Neural Network”. In: *Journal of Nanoelectronics and Optoelectronics* 11 (Feb. 2016), pp. 68–78 (cit. on p. 18).

[Sil+18] D. Silver et al. “A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play”. In: *Science* 362.6419 (Dec. 2018), pp. 1140–1144 (cit. on p. 4).

[SZ15] K. Simonyan and A. Zisserman. “Very Deep Convolutional Networks for Large-Scale Image Recognition”. In: *International Conference on Learning Representations*. 2015 (cit. on pp. 1, 11, 27).

[STK05] O. Sporns, G. Tononi, and R. Kötter. “The Human Connectome: A Structural Description of the Human Brain”. In: *PLoS Computational Biology* 1.4 (2005), p. 42 (cit. on p. 4).

[Sri+14] N. Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research* 15 (June 2014), pp. 1929–1958 (cit. on pp. 3, 19).

[Viv+17] L. de Vivo et al. “Ultrastructural evidence for synaptic scaling across the wake/sleep cycle”. In: *Science* 355.6324 (Feb. 2017), pp. 507–510 (cit. on p. 22).

[Wan+13] L. Wan et al. “Regularization of Neural Networks using DropConnect”. In: *Proceedings of the 30th International Conference on Machine Learning*. Ed. by Sanjoy Dasgupta and David McAllester. Vol. 28. Proceedings of Machine Learning Research 3. Atlanta, Georgia, USA: PMLR, June 2013, pp. 1058–1066 (cit. on p. 19).

[YBZ02] Z. Yanling, D. Bimin, and W. Zhanrong. “Analysis and study of perceptron to solve XOR problem”. In: *The 2nd International Workshop on Autonomous Decentralized System, 2002.* Nov. 2002, pp. 168–173 (cit. on pp. 5, 7).

[YC16] Y. Guoand A. Yao and Y. Chen. “Dynamic Network Surgery for Efficient DNNs”. In: *Advances in Neural Information Processing Systems 29*. Ed. by D. D. Lee et al. Curran Associates, Inc., 2016, pp. 1379–1387 (cit. on p. 1).

[Zel00] A. Zell. *Simulation Neuronaler Netze*. R. Oldenburg Verlag München, 2000 (cit. on pp. 8–11, 14).

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie, dass ich die Bachelorarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Passau, April 24, 2019

Paul Häusner