FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Deep Learning in Medical Image Analysis

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Medizinische Informatik

eingereicht von

Philipp Seeböck
Matrikelnummer 0925270

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Robert Sablatnig
Mitwirkung: Univ.Lektor Dipl.-Ing. Dr.techn. Georg Langs

Wien, 16.01.2015

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Deep Learning in Medical Image Analysis

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Medical Informatics

by

Philipp Seeböck
Registration Number 0925270

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Robert Sablatnig
Assistance: Univ.Lektor Dipl.-Ing. Dr.techn. Georg Langs

Vienna, 16.01.2015

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Philipp Seeböck
Westbahnstrasse 1, 1070 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

Though it is stated in the guidelines of the faculty that ”acknowledgements are optional”, I think to say ”Thank you!” to the people who supported me during my work in this thesis is the least that I can do. This is why I would like to use this space in order to return something back to these special people.

First of all, I would like to say a big thank you to René Donner (MedUni Vienna), who supported me in all aspects of this work. He answered all my questions almost instantly, no matter when and why they arose. He helped me deal with technical challenges, showed great patience in several situations, even explained things twice if necessary and gave me good hints which helped me improve the quality of this work.

My special thanks to Georg Langs (MedUni Vienna), who encouraged me throughout the time I worked on my thesis, made my life easier through his support and gave me the opportunity to write my thesis in the field of machine learning. Another thank goes to Robert Sablatnig (TU Vienna), who improved my scientific writing style and explained to me the tough everyday reality of an academic.

I am most indebted to the most fantastic sister and best parents in the world for their great support. They were there for me all the time, in all matters, which in turn helped me to finish this work. Especially in those periods in which I was not very well, they showed me why they are so important for me.

I am particularly grateful to my girlfriend Désirée - not only for her endless patience, but also for her positive nature that increased my motivation in multiple situations. She made me laugh and helped me find solutions which I would not have found without her.

Abstract

Machine learning is used in the medical imaging field, including computer-aided diagnosis, image segmentation, image registration, image fusion, image-guided therapy, image annotation, and image database retrieval. Deep learning methods are a set of algorithms in machine learning, which try to automatically learn multiple levels of representation and abstraction that help make sense of data. This in turn leads to the necessity of understanding and examining the characteristics of deep learning approaches, in order to be able to apply and refine the methods in a proper way.

The aim of this work is to evaluate deep learning methods in the medical domain and to understand if deep learning methods (random recursive support vector machines, stacked sparse auto-encoders, stacked denoising auto-encoders, K-means deep learning algorithm) outperform other state of the art approaches (K-nearest neighbor, support vector machines, extremely randomized trees) on two classification tasks, where the methods are evaluated on a handwritten digit (MNIST) and on a medical (PULMO) dataset. Beside an evaluation in terms of accuracy and runtime, a qualitative analysis of the learned features and practical recommendations for the evaluated methods are provided within this work. This should help improve the application and refinement of the evaluated methods in future.

Results indicate that the stacked sparse auto-encoder, the stacked denoising auto-encoder and the support vector machine achieve the highest accuracy among all evaluated approaches on both datasets. These methods are preferable, if the available computational resources allow to use them. In contrast, the random recursive support vector machines exhibit the shortest training time on both datasets, but achieve a poorer accuracy than the afore mentioned approaches. This implies that if the computational resources are limited and the runtime is an important issue, the random recursive support vector machines should be used.

Kurzfassung

Machine Learning wird im Bereich der medizinischen Bildverarbeitung eingesetzt, wobei unter anderem Computergestützte Diagnose, Segmentierung, Registrierung, bildgestützte Therapie, Annotierung und Bilddatenbankabfragen Aufgabenbereiche sind. Deep Learning Methoden sind spezielle Machine Learning Ansätze, welche versuchen automatisch mehrere Ebenen der Abstraktion (bzw. Repräsentation) zu lernen, um Daten interpretieren und verstehen zu können. Dies führt, um in der Lage zu sein diese Methoden auf die richtige Art und Weise anwenden und verfeinern zu können, zu der Notwendigkeit die Charakteristiken der Deep Learning Methoden zu untersuchen und zu verstehen.

Das Ziel dieser Arbeit ist deshalb Deep Learning Methoden in einem medizinischen Kontext zu evaluieren und herauszufinden, welche Vorteile Deep Learning Methoden (random recursive support vector machines, stacked sparse auto-encoders, stacked denoising auto-encoders, K-means deep learning algorithm) im Vergleich mit state of the art Ansätzen (K-nearest neighbor, support vector machines, extremely randomized trees) bringen. Dafür werden die Methoden auf zwei Klassifikationsaufgaben evaluiert, wobei einerseits ein Datenset mit handgeschriebenen Zahlen (MNIST) und andererseits ein medizinisches Datenset (PULMO) verwendet wird. Neben einer Evaluierung in Bezug auf Vorhersagegenauigkeit und Laufzeit, enthält diese Arbeit auch eine qualitative Analyse der gelernten Eigenschaften der Deep Learning Ansätze und praktische Empfehlungen, wobei dies helfen soll die Anwendung und Verbesserung der Methoden in Zukunft zu unterstützen.

Die Ergebnisse deuten darauf hin, dass stacked sparse auto-encoders, stacked denoising auto-encoders und support vector machines die höchste Vorhersagegenauigkeit auf beiden Datensätzen erreichen. Das bedeutet, dass diese Methoden zu bevorzugen sind, falls die vorhandenen Rechnerressourcen dies erlauben. Im Gegensatz dazu weisen die random recursive support vector machines die kürzeste Trainingsdauer auf beiden Datensätzen auf, erreichen allerdings eine niedrigere Vorhersagegenauigkeit im Vergleich der vorher genannten Methoden. Das impliziert, dass die random recursive support vector machines verwendet werden sollten, falls die Rechnerressourcen stark begrenzt sind und die Laufzeit ein wichtiger Faktor bei der Auswahl der Methode ist.

Contents

CHAPTER 1

Introduction

This chapter gives an overview of the motivation, problem statement, aim and structure of this work. First of all, the motivation to analyze deep learning methods in a medical domain is described in the first section. Subsequently, the aim of the work is explained. The chapter concludes with an outline of the general structure of this thesis.

1.1 Motivation

Machine learning is used in the medical imaging field, including computer-aided diagnosis, image segmentation, image registration, image fusion, image-guided therapy, image annotation, and image database retrieval. This means that there are multiple areas in medicine, where machine learning methods can be applied and can help improve patients' health care [77]. For instance, machine learning methods can be used for early detection of breast cancer [78]. One particular task that can be addressed with machine learning approaches is classification, where objects are classified (e.g. abnormal or normal, benign or malign) based upon input features [13, 74, 76]. The classification of lung diseases in computed tomography scans is one example for the application of machine learning methods on this task [69, 74].

Deep learning methods are a set of algorithms in machine learning, which learn multiple levels of representation and abstraction that help make sense of data [7]. Higher-level abstractions are defined from lower-level ones, so more complex functions can be learned. In particular, if a function can be compactly represented by a deep architecture, the same function could require an extremely large architecture if the depth of this architecture is made more shallow [7]. Furthermore, this brief overview demonstrates that the algorithmic literature offers a high variety of deep learning approaches. In this thesis we will perform an empirical evaluation of representative approaches, and discuss the conclusions from these findings.

1.2 Aim of the Work

The aim of this work is to evaluate deep learning methods in the medical domain and to study if deep learning methods outperform state of the art approaches. More precisely, the methods are evaluated on two classification tasks, where one task consists of classifying images of a medical dataset (computed tomography images of the lung). The other classification task involves a dataset of handwritten digits which enables the opportunity to identify the differences of the approaches between a conventional problem and a task in the medical domain. Beside an evaluation in terms of accuracy, the methods are also compared in terms of runtime. Here, not only the absolute values of the accuracy and the runtime are compared with each other, but also the influence of the hyper-parameter settings and the number of used examples on the performance are examined. Furthermore, the features which are learned by the deep learning approaches are evaluated too. For instance, it is examined if the deep learning approaches are successful in learning problem specific features automatically. Another objective of this thesis is to work out practical recommendations for the evaluated methods, where they should help improve the application and refinement of the evaluated methods in future. In summary, the following main objectives can be identified:

- Evaluation of the general performance of the methods based on two classification tasks.
- Examination of the influence of the hyper-parameters on accuracy and runtime of the approaches.
- Summary of practical recommendations for the examined methods in this work, derived from the comparative experiments.

1.3 Structure of the Work

The thesis is organized as follows:

Chapter 2: Section 2.1 provides a background of general machine learning approaches such as supervised and unsupervised learning, under- and overfitting, hyper-parameters and preprocessing. With the knowledge of this general introduction into the field of machine learning, the conventional state of the art approaches can be explained subsequently. First, the K-nearest neighbor approach is described in Section 2.2, followed by an introduction into support vector machines in Section 2.3 and concluded by an explanation of the extremely randomized trees in Section 2.4.

Chapter 3: In the beginning of Section 3.1, an introduction into the basic ideas of deep learning, respectively deep architectures, is given. Deep neural networks are described in detail in Section 3.2, random recursive support vector machines are discussed in Section 3.3 and the K-means deep learning algorithm is explained in Section 3.4

Chapter 4: This chapter describes the details of the implementation, where issues such as the used programming languages, software packages and frameworks are addressed for K-nearest neighbor approach (Section 4.1), support vector machines (Section 4.2), extremely randomized trees (Section 4.4), neural nets (Section 4.5), random recursive support vector machines (Section 4.3) and K-means deep learning algorithm (Section 4.6).

Chapter 5 and Chapter 6: Chapter 5 reports results of the evaluation on the handwritten digit dataset, while the evaluation of the medical dataset is given in Chapter 6. Beside a description of the dataset and the evaluation settings, both chapters provide an evaluation in terms of accuracy, runtime, hyper-parameters, number of used examples and special characteristics of the methods.

Chapter 7: In this chapter, the major results of Chapter 5 and Chapter 6 are summarized and compared with each other. Furthermore, practical recommendations are given.

Chapter 8: Finally, Chapter 8 summarizes the major findings and gives insight into future work.

CHAPTER 2

Machine Learning

This chapter covers general machine learning topics such as supervised and unsupervised learning. Because the literature of this discipline, due to decades of research activities, is extremely large, only a small proportion can be considered in this thesis [62]. Nevertheless we cover the topics, which are needed in order to understand the specific machine learning approaches which are evaluated and to better assess the findings of this thesis. First of all, for the purpose of a better understanding of the differences between the approaches, an overview of learning types is given. Subsequently the choice of hyper-parameters is discussed, due to the fact that they have to be chosen for all methods which are evaluated within this work. Since it is necessary to comprehend how the ”best” method (respectively configuration) can be found, this issue is explained in Section 2.1.3. In order to be able to interpret the results and to understand the training of models and its challenges, it is necessary to cover the issue of underfitting and overfitting, which is done in Section 2.1.4. Thereafter, common preprocessing techniques are described, since preprocessing is used as a part of the evaluation pipeline in this thesis.

Beyond the general part, the focus in this chapter is on ’classic’ machine learning approaches, which represent the current state of the art [32, 64, 75] and are relevant in connection with the practical work. Since *K-Nearest Neighbors*, *Support Vector Machines* and *extremely randomized trees* are evaluated and compared with each other in Chapter 5 and Chapter 6, it is necessary to understand the basic idea, the functionality and algorithmic details of each method. This is crucial in order to comprehend the evaluation and the findings of these approaches. The focus of Section 2.2, Section 2.3 and Section 2.4 is centered on classification, due to the fact that the methods are evaluated on two classification tasks.

2.1 Machine Learning in General

In machine learning, the typical goal is to find a mapping from input patterns to an output value [13]. For instance, we have images of objects as input data (represented by pixel intensity values) and correct labels (one for every type of object) as corresponding output values. Then

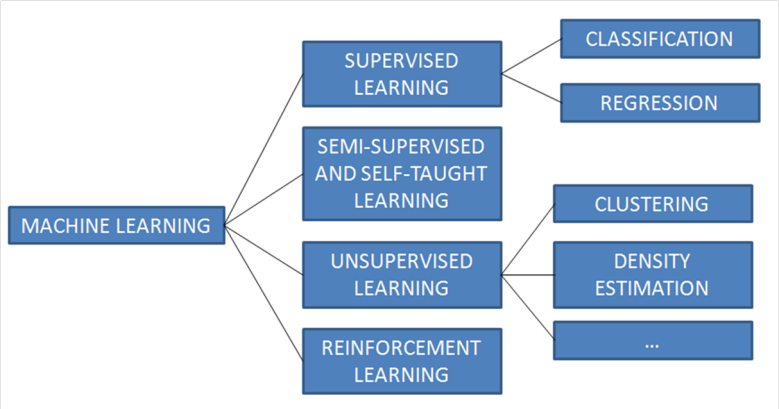


Figure 2.1: Overview of Machine Learning Settings

the aim of the algorithm is to learn this mapping (from the samples to the output value), to be able to predict the correct output of a new input sample.

In practice, this direct mapping from input to output values is often a very complex and non-linear function, which cannot be discovered by machine learning algorithms which exclusively learn a linear mapping [21]. This is the reason, why researchers model data using hand-optimized features, so that the algorithm can learn the simplified mapping from these features to the output value. Unfortunately this strategy leads to the necessity of creating new hand-optimized features for each novel task and the process of engineering features is complex and difficult [21]. In this work we also take a look at algorithms, which attempt to learn feature representations themselves from data.

Generally, the so-called *training set* is used to tune the parameters of the machine learning model [13]. This training set consists of n input vectors $\{x_1, x_2, ..., x_n\}$ and optionally n target vectors (one for each sample). During the training phase, also known as the learning phase, the parameters of the model, which determine the mapping from the input vector x_i to the output value $y(x_i)$, are adapted [13].

After the training phase, the *test set* is used to identify the quality of the obtained model [13]. The ability of the model to correctly predict the target value of new (unknown) samples is called *generalization*. Since the training set typically covers only a small amount of all possible input vectors, generalization is an important purpose in machine learning [13]. There are different techniques to address this issue, which are introduced in Section 2.1.3.

2.1.1 Algorithm Types

There are different machine learning settings [13], which are illustrated in Figure 2.1 and are explained below. The characteristics of the data set and the specific task determine the algorithm type [21], as described in the following.

Supervised Learning The *supervised learning* problem is a classic machine learning setting [13]. The training data is made up of tuples (x_i, y_i), where x_i is the input and y_i the corresponding target vector [21]. The case in which the target value is discrete, such as the digit recognition problem, where the images are mapped to a finite number of discrete categories, is called *classification* problem. If the target value is continuous, the task is called *regression* problem [13]. An example of a regression problem would be the prediction of the price of a house (continuous output value), where the input variables are the number of rooms and the living area.

Unsupervised Learning In *unsupervised learning* problems, the training data comprises examples of the input vectors without any corresponding target values [13]. There are different objectives in unsupervised learning problems, such as *clustering*, *density estimation* and *visualization* [13]. The goal of *clustering* is to discover groups of similar examples, on the basis of measured or perceived similarities between the examples [13]. To determine the distribution of the data within the input space, is the purpose in *density estimation*. For *visualization*, the data is projected down, from a high-dimensional space, to two or three dimensions [13].

Semi-Supervised and Self-Taught Learning There are two approaches, which are using unlabeled data in supervised learning tasks and are therefore halfway between supervised and unsupervised learning. Both the possibility to have algorithms learn from unlabeled data and the challenge to get (enough) labeled data for supervised learning tasks are motivating factors for these approaches [70]. The main idea is to give the algorithm large amounts of unlabeled data to learn a good feature representation of the input and then feed the labeled data to the algorithm to solve the supervised task on the basis of the learned feature representation [19].

In the *semi-supervised learning* setting, the training data set can be divided into two parts: the data samples $X_l = \{x_1, ..., x_l\}$ with corresponding labels $Y_l = \{y_1, ..., y_l\}$, and the data samples $X_u = \{x_{l+1}, ..., x_{l+u}\}$ where the labels are not known [19]. On the other hand, the *self-taught learning* setting is the more powerful setting, because it does not assume, in contrast to the semi-supervised learning setting, that the unlabeled data X_u follows the same distribution (or class labels) as the labeled data X_l [70].

To stress the difference between the two mentioned approaches, we consider an example, where the aim is to distinguish between images of cats and images of dogs. Both approaches use labeled images, whereby each example is either an image of a dog or an image of a cat. If there are unlabeled data examples that are all images of either a cat or a dog (but are not labeled), then this setting would be called *semi-supervised*. In contrast, in *self-taught* learning there is an unlabeled data set, which consists of random images (perhaps downloaded off the Internet) and is therefore drawn from a different distribution than the labeled data set.

Reinforcement Learning *Reinforcement learning* is concerned with the problem of how to interact with the environment and to choose suitable actions in a given situation, in order to maximize the reward [13]. In contrast to supervised learning, there are no examples with an optimal output given but instead, the algorithm should learn which actions to execute in a specific situation by interacting with the environment (process of trial and error) [13].

2.1.2 Hyper-Parameters

Bengio defines the problem in the following way:

> "We define a hyper-parameter for a learning algorithm A as a value to be selected prior to the actual application of A to the data, a value that is not directly selected by the learning algorithm itself." [9, p.7].

Choosing hyper-parameters is therefore formally equivalent to model-selection, i.e. choosing the most appropriate value/algorithm in the given set of values/algorithms [9]. Hyper-parameters can be continuous (e.g. learning rate) or discrete (e.g. the number of neurons in one layer) and can be seen as an outside control button [9]. To point out the difference between hyper-parameters and parameters, we consider an example where we would like to train a polynomial function to fit a specific function. The polynomial function takes the form

$$y(x, \mathbf{w}) = w_0 + w_1 x + w_2 x^2 + ... + w_M x^M = \sum_{j=0}^{M} w_j x^j \tag{2.1}$$

where M is the order of the polynomial and w_i are the polynomial coefficients ($\mathbf{w} = w_0, w_1, ..., w_M$). M has to be chosen 'by hand' and *before* the training of the polynomial is started. In contrast, the polynomial coefficients w_i are adapted within the training procedure itself. As a result M can be seen as a hyper-parameter, whereas the polynomial coefficients are parameters. For all learning algorithms which are examined in this work, hyper-parameters have to be adjusted. In some cases, such as neural networks, the number of hyper-parameters can be relatively high (ten or more), which complicates the matter [9]. In this thesis, the specific hyper-parameters of the various learning algorithms are discussed too.

2.1.3 Model Selection

Different machine learning models show differences in terms of complexity and the number of free parameters, which can be adjusted [13]. Beside the model-selection itself, the values for the parameters of the model have to be determined (as mentioned in Section 2.1.2). More precisely, the main objective in this context is to choose the values of the free parameters in such a way, that the best predictive performance on new data is achieved [29].

To evaluate the predictive performance of different values for the parameters, the training set is not a good indicator, due to the problem of overfitting (more details on this matter can be found in Section 2.1.4). Instead, if there is enough data, the dataset can be divided into three parts. The model is trained with the first part of the data and different hyper-parameter settings, whereby this fraction is called *training set* [13]. The second part of the data is not used for training and therefore suitable to compare the various hyper-parameter settings and select the one with the best evaluated predictive performance. This independent data set is called *validation set*. To finally evaluate the selected model in terms of generalization, a third part of the data, called *test set*, is necessary. This is because the validation set was used to select the final model and is therefore no longer entirely independent from the model (overfitting to the validation data can occur) [13].

Another technique is *cross-validation*, where the dataset is divided into two parts: the training set and the test set. This technique is especially used if the amount of data is limited and as much of the available data as possible should be used for training [13]. There are several variants of cross-validation, but the main idea remains the same for all. A common type is *k-fold cross-validation*, where the training set is randomly divided into k groups [13]. $k-1$ groups are used to train a set of models, which are then evaluated on the remaining group. This process is repeated for each group, so that, in the end, every group has been used once for evaluating [13]. The performance scores of all runs are then averaged and used for model-selection. Finally the test set is used to evaluate the performance of the selected model. Beside *k-fold cross-validation*, another specific variant is the *leave-one-out-technique*, where k is chosen to be equal to the total number of samples. A drawback of cross-validation is that the number of training runs (for a single setting) is increased by the factor of k [13].

Beside the validation error, which is a proxy for the generalization error, computational aspects may also be taken into consideration in the process of selecting hyper-parameters [9]. Computing resources are limited and, for instance, have an impact on the choice of intervals of values to consider. There are several techniques for optimizing hyper-parameters, such as *coordinate descent*, *multi-resolution search*, *grid search* or *random search* [9]. In the strategy called *coordinate descent*, only one hyper-parameter is changed at a time, whereby a change is always made from the best hyper-parameter-configuration found presently. The *grid search* is simply an exhaustive search of all possible combinations of the selected hyper-parameters [9]. The main disadvantage of this technique is that the number of configurations grows exponentially with the number of hyper-parameters (e.g. 6 hyper-parameters, each allowed to take 5 different values, lead to $5^6 = 15625$ combinations). On the other hand, the grid search is fully parallelizable, which is the advantage of this technique (ability to run different combinations on different computers). The *random search* is an alternative, where all parameters are changed at the same time (in contrast to grid search) and each hyper-parameter is independently sampled from a prior distribution (e.g. uniform distribution, inside the interval of interest) [9]. Random sampling can be more efficient than a grid search, in particular if the number of hyper-parameters goes beyond two or three. On the other hand, the qualitative analysis of the results is more complicated [9]. In *multi-resolution search*, the idea is to start the search with large sized steps of the hyper-parameters. After several 'best configurations' have been found, one can start from there and explore more locally around them to optimize the configuration in detail (with smaller sized steps) [9].

2.1.4 Underfitting and Overfitting

As mentioned above, the objective in machine learning is to maximize the predictive accuracy on new data and not on training data. If the model fits perfectly to the training data, the predictive accuracy on new data must not necessarily be good [29].

For better understanding we consider an example, where we want to fit the function $\sin(2\pi x)$ with a polynomial of order M. The order M of the polynomial has to be chosen, to determine the complexity of this linear model. As shown in Figure 2.2, the higher order polynomial (M=9) shows a perfect fit to the training data, but oscillates wildly and represents the 'true' function very badly. This is called *overfitting* [29]. On the other hand, the lower order polynomials

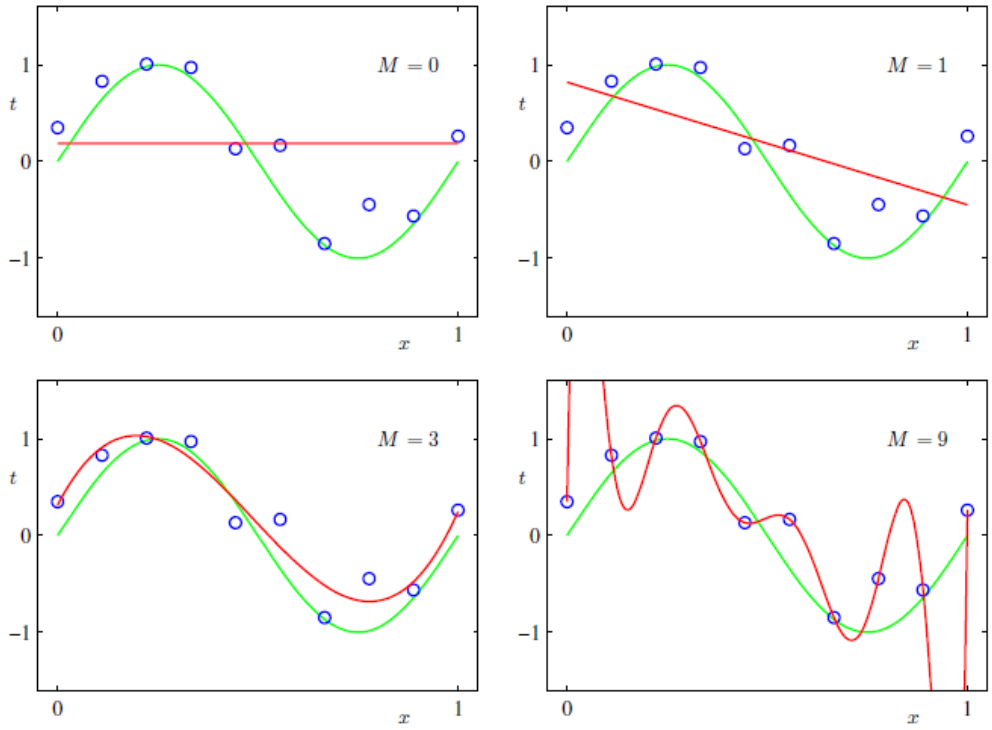


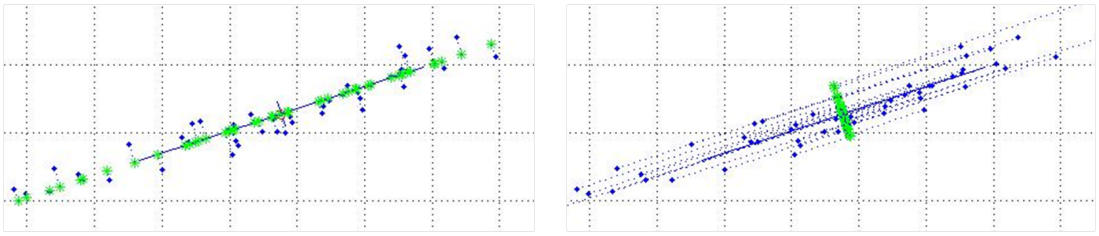
Figure 2.2: Plots of polynomials with various orders M, shown as red curves. The function to fit $\sin(2\pi x)$ is shown as a green curve. The blue circles represent the data points of the training set ($\sin(2\pi x)$ with added random noise) [13, p.7]

(M=0, M=1) are not complex enough to fit the underlying trends in the training data (as shown in Figure 2.2) and are therefore rather poor representations of the 'true' function too. This is called *underfitting* [13].

2.1.5 The Role of Preprocessing

As stated in Coates [21], preprocessing is a common step in machine learning. There are several different methods, like *PCA* and *whitening*, which can be used, depending on the type of input data. This section points out the basic ideas of preprocessing and helps to understand why preprocessing can help to improve the performance of machine learning approaches. Since preprocessing is used as a part of the evaluation pipeline in this thesis, this section is necessary in order to assess the findings in Chapter 5 and Chapter 6.

Normalization of Data A simple preprocessing step in machine learning consists of computing the mean value (of all dimensions) of an example and subtracting this mean value from every dimension of that example [21]. For images, this can be seen as *normalization of the brightness*.



(a) (b)

Figure 2.3: Illustration of PCA, considering an example, where the data is only two-dimensional. The two eigenvectors form a new basis in which the data can be represented. (a) Projection of the data onto the first eigenvector. (b) Projection of the data onto the second eigenvector.

In some tasks the overall-brightness information of an image is dispensable, wherefore this normalization is appropriate (e.g. object-recognition tasks [68]).

Another normalization step is to compute the standard deviation of an example and divide the example by this computed value [21]. In images, this has the property of *contrast normalization* (as mentioned in Coates et al. [23]).

PCA for Dimensionality Reduction *Principal component analysis* is an algorithm which can be used for dimensionality reduction [44]. This can be useful to reduce the running time of machine learning algorithms, since runtime may depend on the number of input features. The main idea of PCA is to find a more compact representation of the data, under the assumption that the input-variables are correlated with each other and therefore redundant (e.g. nearby pixels in images [47]). The aim is to describe the original high-dimensional data as well as possible, using a lower dimensional subspace [1]. Geometrically, this can be seen as a linear projection of the original data example into a (lower-dimensional) coordinate system, under the condition to retain as much information as possible (as shown in Figure 2.3). The main axes of this new coordinate system are called *principal components* [1].

To achieve this, in PCA the covariance-matrix of the given dataset is computed. The elements of the covariance matrix determine the statistical relation between the pairs of variables. The eigenvectors of this covariance-matrix form an orthonormal basis [1, 44]. The *eigenvectors* of this covariance-matrix point into the direction of highest variation of the data, form a new basis in which the data can be represented and are equal to the principal components [44]. If e_1 and e_2 are the eigenvectors and x_i is an input-example, the corresponding representation of x_i in the new (e_1, e_2)-basis is $\hat{x_i} = (e_1^T x_i, e_2^T x_i)$. The coefficients of the data with respect to this new basis are uncorrelated, which is a desired property [42]. The *eigenvalues* represent the variance of the data projected onto the corresponding eigenvectors. The eigenvalue indicates, how much of the general variance is covered by its corresponding eigenvector [1]. For example, in Figure 2.3 it can be seen that the first eigenvector has a higher variance and therefore a higher eigenvalue than the second eigenvector.

To obtain a dimensionality reduction from n to k dimensions, the k components with the

highest eigenvalues are retained, while the other ones are dropped [42]. This way, as much variance as possible is retained in the resulting representation. Referring to the example in Figure 2.3, if we would like to reduce the dimension from $n = 2$ to $k = 1$, the first eigenvector (alias principal component) would be retained, while the second one would be dropped. In practice the feature dimension is generally much higher and the choice of k is therefore more complex [42]. A more detailed explanation of PCA and its properties can be found in Jolliffe [44] or Abdi et al. [1].

Whitening for Improvement of Feature Learning A motivation for performing whitening on data, is that the machine learning model should learn interesting regularities in the images, instead of learning that nearby pixels are similar [47]. When considering the example of images as input data, in which nearby pixels exhibit a strong correlation, a machine learning model would learn these correlations instead of discovering more useful characteristics, so long as these correlations are not removed during preprocessing. The objective of *whitening* is therefore to obtain a representation, where the input features are less correlated with each other and have the same variance. In PCA, after the transformation to the new coordinate system, the transformed (d-dimensional) data $\hat{x}$ has a covariance-matrix of

$$\begin{pmatrix} \lambda_1 & 0 & \dots & 0 \\ 0 & \lambda_2 & \dots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & \lambda_d \end{pmatrix} \tag{2.2}$$

where the diagonal-entries $\lambda_1, \lambda_2, ..., \lambda_d$ are the eigenvalues of the corresponding eigenvectors $E = (e_1, e_2, ..., e_d)$ [42]. The other entries are zero, which means that the features are uncorrelated to each other. To fulfill the second criterion for whitened data (unit variance of input features), the principal components are divided by their standard deviations. In summary, it can be stated that whitened components s_i can be computed as following:

$$s_i = \frac{\hat{x_i}}{\sqrt{\lambda_i}} = \frac{E^T x_i}{\sqrt{\lambda_i}} \tag{2.3}$$

The covariance-matrix of the resulting data is equal to the identity matrix [42]. This whitening procedure is called *PCA-whitening*. Sometimes a small constant ϵ is added to the eigenvalues ($\frac{\hat{x_i}}{\sqrt{\lambda_i + \epsilon}}$) to avoid numerical instabilities (some of the λ_i can be close to zero) and filter out aliasing artifacts (e.g. Coates et al. [23] use a constant). It is important to note that there are also many other whitening transformations beside the PCA-whitening (e.g. ZCA-whitening which differs from PCA-whitening only by a rotation)[1] [42].

2.2 K-Nearest Neighbor Approach for Classification

The *K-nearest neighbor* approach is one of the approaches for classification, which is very intuitive and simple to understand, but works very well in practice [26]. In the classification

[1]The way of whitening the data is not unique. If the data set S is white, then any orthogonal transformation (rotation, reflection) of this data set is also white. [42].

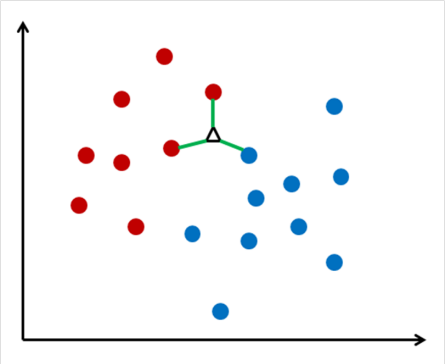


Figure 2.4: Example for a K-nearest neighbor classifier, where $K = 3$. The black triangle is the new data point, which is classified according to the majority class membership of the 3 closest training data points. In this example, the back triangle would be classified as red.

problem, as can be seen in Figure 2.4, a new (unlabeled) data point is classified according to the labels of the k nearest neighbors [13, 26, 83]. The classification of a new point can now be divided into two steps: First, the k nearest neighbors have to be determined. There are different techniques to compute the distance between two examples (e.g. Euclidean distance, Manhattan distance) and the performance of this algorithm depends on the used *distance metric* to identify the nearest neighbors [26, 83]. Secondly, the class of the new data point has to be determined, corresponding to the labels of the k nearest neighbors, which have been identified in the previous step [13, 26, 83]. There are various ways to calculate the final class from the k nearest neighbors, beside the simple majority vote [26]. For example, Cunningham et al. [26] introduce a technique, where the votes of nearer neighbors have more influence on the final classification than neighbors which are further away.

In contrast to SVMs or Random Forests, the K-nearest neighbor algorithm has no, or only a minimal, explicit training phase. Instead, the whole training set is used for the prediction of a new data sample (in contrast to the 'sparse representation' in SVM, where only the *support vectors* are used for prediction) [13, 26, 83].

Though the main focus of the K-nearest neighbor algorithm is on classification, the regression problem can be solved with this approach too. The target value is then computed of the values of the k nearest neighbors [28]. Further, this method can be used for density estimation too. For a given position x, the density $p(x)$ can be calculated as

$$p(x) = \frac{K}{VN} \tag{2.4}$$

where V is the volume of the sphere (centered at x, and the radius of the sphere is allowed to grow until it contains precisely K data points), and N is the number of observations in the training set [13].

2.2.1 K-Nearest Neighbor: Hyper-Parameters

The number of neighbors K has to be chosen. A high value of K generates fewer regions of each class which are larger, whereas a smaller value of K leads to more small regions of each class [13]. Therefore we can see that K controls the degree of smoothing and should neither be chosen to small nor to large [13, 26, 83].

2.3 Support Vector Machines for Classification

Support Vector Machines (SVMs) were first introduced in 1992 by Boser et al. [14] and became popular for solving problems in classification, regression and novelty detection [11]. An important property of SVMs is that the learning of the model parameters involves the optimization of a convex function. As a result, there are no 'false' minima and every local solution is a global optimum, unlike in neural networks [13]. Another advantage, in contrast to neural networks, is that only few parameters are needed for tuning the learning machine [13]. To discuss all features and mathematical derivations (in detail) of SVMs would go beyond the scope of this work. Therefore, the main ideas of support vector machines are introduced in this section and the key concepts of the model are explained step-by-step.

2.3.1 Maximum Margin Classifier

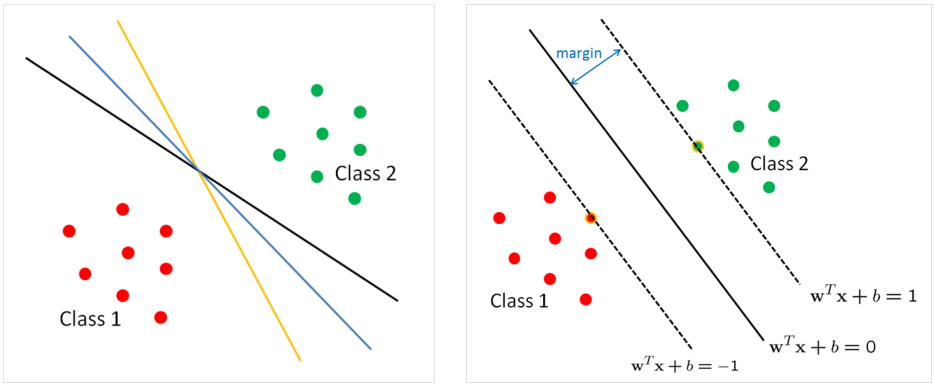
First, we consider the two-class classification problem, where we assume that the classes are linearly separable. The training data set comprises the input vectors $\{x_1, ..., x_n\}$ with the corresponding target values $\{y_1, ..., y_n\}$, where $y_i \in \{-1, 1\}$ and new data points are classified according to the sign of $sign[w^T x + b]$, where $w^T x + b = 0$ denotes the decision hyperplane (w determines the orientation of the plane, and b the offset of the plane from the origin) [11]. If we look at Figure 2.5(a), we can see that there are many possible solutions for the decision boundary. The first key concept of support vector machines is to choose the decision hyperplane with the *maximum margin*, where the *margin* is the smallest distance between the plane and any of the samples, as illustrated in Figure 2.5(b) [13]. In this example, the solution depends only on two points, which are marked in Figure 2.5(b). These two points (a subset of the training data set), which determine the location of the boundary, are called *support vectors*.

Mathematically, the maximum margin solution can be formulated as a constrained optimization problem:

$$minimize \qquad \frac{1}{2}\|w\|^2 \tag{2.5}$$

$$subject\ to \qquad y_i(w^T x_i + b) \geq 1 \qquad \forall i \tag{2.6}$$

where minimizing $\frac{1}{2}\|w\|^2$ is equal to maximizing the margin (given by $\frac{2}{\|w\|}$), and the constraint ensures that all points are classified correctly (one constraint for each point) [13]. Next, the



(a) (b)

Figure 2.5: (a) Examples for possible solutions of the two-class classification problem (linearly separable). (b) Illustration of the *maximum margin*.

Lagrange theory[2] is used to rewrite this constrained optimization problem as:

$$maximize \qquad W(\alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j y_i y_j (x_i^T x_j) \tag{2.7}$$

$$subject\ to \qquad \alpha_i \geq 0, \quad \sum_{i=1}^{n} \alpha_i y_i = 0 \tag{2.8}$$

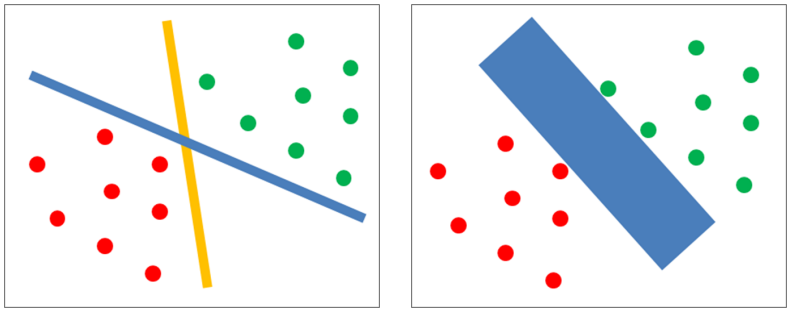
where α_i is called Lagrange multiplier (one for each constraint), and the new objective function is in terms of α_i only [11]. The 'original' problem is called *primal problem*, whereas the new formulation is known as the *dual problem*: if we know all α_i, we know w, and if we know w, we know all α_i (because $w = \sum_{i=1}^{n} \alpha_i y_i x_i$) [13]. The decision function, for a new input z, can therefore now be written as $sign[\sum_{i=1}^{n} \alpha_i y_i (x_i^T z) + b]$. Many of the α_i are zero, which means that w is a linear combination of a few data points. These data points, where α_i is not zero, are called *support vectors* and determine the decision hyperplane. It is interesting to note that the larger α_i is, the bigger the influence of the data point on the position of the hyperplane is [13].

This constrained optimization problem is a convex *quadratic programming* task. There are robust algorithms for solving such quadratic programming problems [11]. Though both formulations give the same result, in practice the dual formulation is preferable [11], because it has quite simple constraints and allows the model to be reformulated using kernels, as discussed later [11].

2.3.2 Theoretical Foundation

The support vector machines are properly motivated by statistical learning theory [13]. Broadly speaking, statistical learning proves that bounds for the generalization error can be obtained. The

[2]The key concepts of Lagrange multipliers can be reviewed in [13, Appendix E].



(a) (b)

Figure 2.6: (a) Many possible 'narrow' margin planes. (b) Few possible 'broad' margin planes.

bounds are a function of the training data misclassification error and the complexity, respectively the capacity of the model [11].

Maximizing the margin of separation of a linear function reduces the complexity or capacity of this function [11]. This is a desirable property, because it minimizes the bounds for the generalization error. In other words, maximizing the margin leads to better generalization with a high probability [11]. This is motivated by the consideration that a model with high capacity is much more likely to overfit the training data, which leads to a poor generalization performance. Geometrically, this deliberation is illustrated in Figure 2.6. A plane with a 'narrow' margin can take many possible positions and still separate the data perfectly. On the contrary, a plane with a 'broad' margin has limited flexibility to separate the data. We can see intuitively that a 'broad' margin plane is less complex than a 'narrow' one and that maximizing the margin regulates the complexity of the model. Furthermore, the size of the margin does not directly depend on the dimensionality of the data [11]. That is why good performance can be expected even for high-dimensional data. For further details, literature on this topic exists, e.g. [79].

2.3.3 Soft Margins

So far, the case in which the two classes are linearly separable (perfectly), has been considered. Now *soft margins* are introduced, a technique to handle a two-class classification problem, which can not be separated perfectly [13]. This constellation is shown in Figure 2.7.

The idea is to reduce the influence of individual data points and to allow some of the training points to be misclassified (lying on the 'wrong side' of the hyperplane) [11]. To do this, *slack variables* are introduced, $\xi_i \geq 0$ where $\xi_i \in \{\xi_1, ..., \xi_n\}$, with one slack variable for each data point [13]. The primal problem therefore becomes:

$$minimize \qquad \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n}\xi_i \tag{2.9}$$

$$subject\ to \qquad y_i(w^T x_i + b) \geq 1 - \xi_i, \quad \xi_i \geq 0 \qquad \forall i \tag{2.10}$$

where $C > 0$ controls the trade-off between maximizing the margin and minimizing the error, and the constraint is now less stringent ('error' allowed, but penalized) [13]. $\xi_i = 0$, if the data

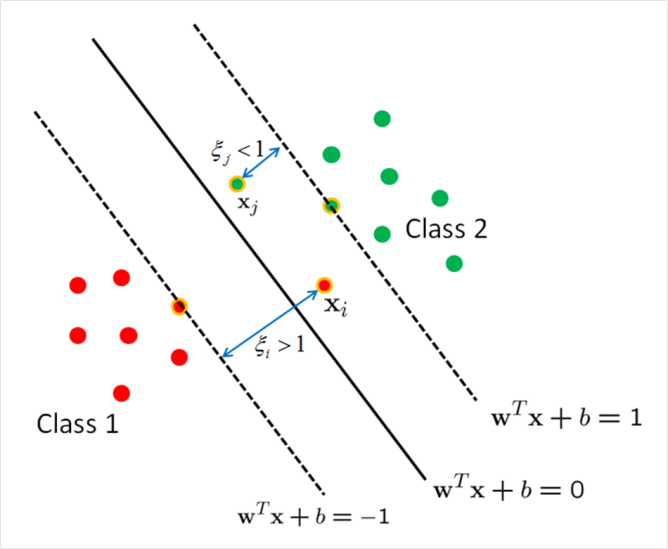


Figure 2.7: Soft Margin Hyperplane, where the support vectors are highlighted in orange.

point lies on or inside the correct margin, and $0 < \xi_i \leq 1$ if the point lies inside the margin, but on the correct side of the decision hyperplane. If the point lies on the wrong side of the decision boundary and is misclassified, $\xi_i > 1$ [13]. The penalty for misclassification increases linearly with ξ and $\sum_{i=1}^{n} \xi_i$ is an upper bound to the number of misclassified points. Here, too, the primal problem can be formulated as dual representation:

$$maximize \qquad W(\alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j y_i y_j (x_i^T x_j) \tag{2.11}$$

$$subject\ to \qquad C \geq \alpha_i \geq 0, \quad \sum_{i=1}^{n} \alpha_i y_i = 0 \tag{2.12}$$

where the new optimization problem is very similar to the one in the linear separable case, except that there is an upper bound C on α_i now [11]. Once again, this is a convex quadratic programming task. Finally it is important to note that the optimal value of C has to be found experimentally (e.g. using a validation set).

2.3.4 Kernel Method

By now, the case where a linear boundary is reasonable, has been considered. But if we consider the example in Figure 2.8, it can be seen that no linear boundary could separate the data of the input space usefully [11]. The idea is to transform the data points from the original *input space* into a (high-dimensional) *feature space*[3], where the data can be properly divided by a linear function (as shown in Figure 2.8). This linear function in the feature space corresponds to a non-linear function in the original input space [13].

[3]The feature space is the space of $\theta(x_i)$ after transformation [17].

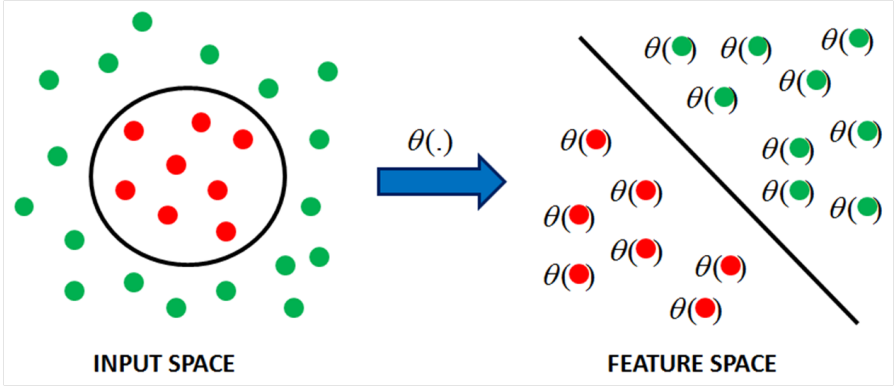


Figure 2.8: Mapping from the input to the feature space. The linear hyperplane in the feature space corresponds to a non-linear hyperplane in the input space. In practice the feature space exhibits a higher dimension than the input space [17].

The transformation is defined as $\theta(x) : R^n \rightarrow R^{n'}$, where $n \ll n'$ [11]. The (dual) optimization problem can then be written as:

$$maximize \qquad W(\alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j y_i y_j \theta(x_i)^T \theta(x_j) \tag{2.13}$$

$$subject\ to \qquad C \geq \alpha_i \geq 0, \quad \sum_{i=1}^{n} \alpha_i y_i = 0 \tag{2.14}$$

Here, the problem is that it can be very costly to compute $\theta(x_i)$. Because the data points in the constrained optimization problem appear only as inner product, *kernels* can be used to avoid this costly computation [13]. The *kernel function*[4] is given by the relation

$$K(x_i, x_j) = \theta(x_i)^T \theta(x_j) \tag{2.15}$$

and can be seen as a similarity measure between the arguments [11]. With the kernel, the inner product between the mapped points can be evaluated without explicitly knowing the underlying function of the mapping. Substituting the kernel into the dual SVM yields:

$$maximize \qquad W(\alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j y_i y_j K(x_i x_j) \tag{2.16}$$

$$subject\ to \qquad C \geq \alpha_i \geq 0, \quad \sum_{i=1}^{n} \alpha_i y_i = 0 \tag{2.17}$$

for the training of the SVM [13]. This is again a convex quadratic programming task. For the prediction of new samples, the 'kernel-trick' can be used again to avoid computing $\theta(x)$

[4] It is important to note that not all kernel functions are allowed. The kernel-function must satisfy the so-called *Mercer Criterion*. Further details can be found in Burges [17].

explicitly. New points z are now classified according to

$$sign[\sum_{i=1}^{N_s} \alpha_i y_i \theta(s_i)^T \theta(z) + b] = sign[\sum_{i=1}^{N_s} \alpha_i y_i K(s_i, z) + b] \tag{2.18}$$

where s_i are the support vectors and N_s the number of support vectors [13]. There are several popular kernels, but for practical purposes, we mention just two of them:

$$Polynomial\ of\ Degree\ 'p' \quad K(a,b) = (a \cdot b + 1)^p \tag{2.19}$$

$$Radial\ Basis\ Function (RBF) \quad K(a,b) = e^{-\|a-b\|^2/2\sigma^2} \tag{2.20}$$

For the polynomial-kernel, p needs to be determined, and for the RBF-Kernel σ needs to be selected [11]. The RBF-Kernel can also be written with parameter γ instead of σ:

$$K(a,b) = e^{-\gamma\|a-b\|^2} \tag{2.21}$$

This is mentioned, because this variant is used in the experiments of this work [18, 41]. By changing kernels we can get different non-linear classifiers, without a need to change the underlying algorithm. In this way, all the benefits of the original linear SVM are retained. On the other hand, the choice of the kernel function restricts the type of transformation which can be applied to the data and can therefore be difficult [11].

2.3.5 ϵ-Support Vector Regression

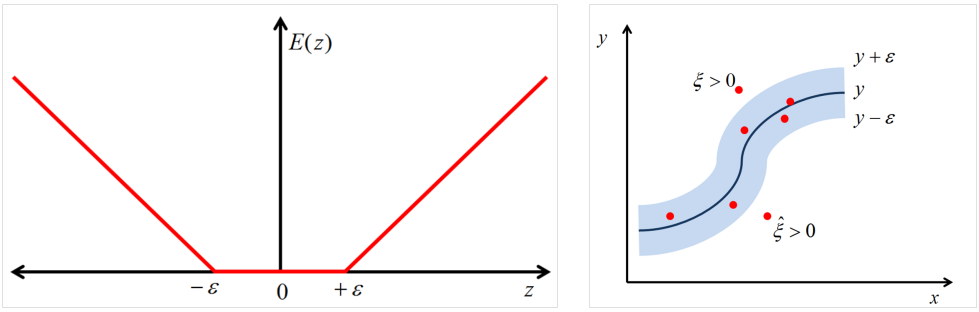
Beside the classification problem, support vector machines can be used for regression tasks too. Therefore, the *ϵ-insensitive error function* (as shown in Figure 2.9(a)) is used to penalize data points, if the absolute difference between the prediction $f(x)$ and the target is bigger than ϵ [13]. It is assumed that there is noise, so constraints like yi - w*xi - b <= epsilon and w*xi + b + yi <= epsilon are defined, to allow a deviation ϵ from the expected function [11]. Geometrically, this can be visualized as a regression 'tube' with size 2ϵ around the hypothesis-function $f(x)$ as illustrated in Figure 2.9(b).

As before, we introduce *slack variables* $\xi_i \geq 0$ and $\hat{\xi}_i \geq 0$ for the two types of training error. If a point lies inside the tube $\xi_i = \hat{\xi}_i = 0$, samples above the tube have $\xi_i > 0,\ \hat{\xi}_i = 0$, and points below the tube have $\xi_i = 0,\ \hat{\xi}_i > 0$ [13]. For a linear ϵ-insensitive loss function the task is therefore:

$$minimize \quad \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n}(\xi_i + \hat{\xi}_i) \tag{2.22}$$

$$subject\ to \quad \begin{cases} (w \cdot x_i - b - y_i) + \xi_i \geq \epsilon \\ (w \cdot x_i - b - y_i) - \hat{\xi}_i \leq -\epsilon \\ \xi_i, \hat{\xi}_i \geq 0 \end{cases} \tag{2.23}$$

where the term $\frac{1}{2}\|w\|^2$ serves to control the complexity of the regression model (*regularization term*), and C controls the trade-off between the complexity of the model and the amount of



(a) (b)

Figure 2.9: (a) The ϵ-insensitive error function (in red). The error increases linearly with the distance outside the insensitive region [13]. (b) Illustration of SVM regression. Points outside of the tube are errors.

errors, which are tolerated (a higher C leads to a more complex model) [11, 20]. At the same time, ϵ controls the width of the tube, where the broader the tube, the fewer support vectors are selected. On the other hand, a bigger ϵ leads to a 'simpler' function [20]. With the same strategy as in the sections above, the linear regression model introduced can be expanded to a non-linear model (computing the Lagrangian dual formulation and using kernels). Again, this can be solved as a quadratic programming problem [13].

2.3.6 SVM: Hyper-Parameters

If we consider the support vector machine for classification, with the exception of C, different hyper-parameters appear, depending on the type of kernel-function selected.

- The parameter $C > 0$ controls the trade-off between maximizing the margin and minimizing the error (number of misclassified examples). A large C leads preferably to a more complicated (respectively narrower) margin, so overfitting can occur for a C, which is too large [13, 17].

- For the Radial Basis Function (RBF) Kernel, the hyper-parameter γ has to be chosen. Since γ controls the RBF-width, a smaller γ leads to a 'smoother' and therefore to a simpler margin (respectively solution) [17]. Because a higher γ leads more easily to a more complicated margin, overfitting can appear for a too high γ [17].

- For the Polynomial Kernel, the degree of the polynomial has to be determined [17].

- If a linear Kernel $K(a, b) = a^T \cdot b$ is used, no kernel-parameters need to be selected [11] [17].

The choice of the kernel function is important, because it determines the type of transformation which can be applied to the data [11].

2.3.7 Multi-Class SVM

Till now, we have discussed SVM-approaches, which attempt to solve the two-class classification problem. There are several methods for Support Vector Machines to deal with multi-class problems, which can be divided into the 'single machine' and the 'divide and conquer' approaches [60]. The former attempts to build a multi-class SVM by solving a single optimization problem, while the latter divides the problem into several binary subproblems and constructs a standard SVM for each one [60]. In respect to the practical part of this thesis, only the two most popular decomposing approaches are discussed in this section.

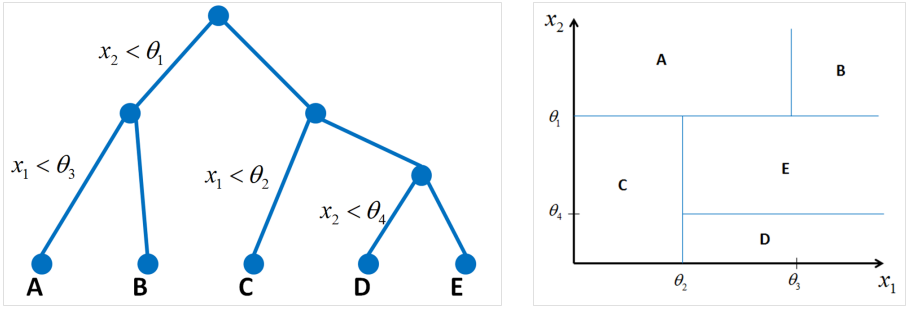
- 'One Against All': Here, one SVM for every class is constructed [2]. Each classifier is trained to distinguish the samples of one class from the samples of the rest (all remaining classes). Typically the classification of an unknown sample is done, by choosing the class with the highest probability [60].
- 'One Against One': One SVM is built for each pair of classes, wherefore $C(C-1)/2$ classifiers are needed for a problem with C classes [2]. Typically, the class with the highest score is chosen, where each classifier votes for one class [60]. With respect to training time, it may be advantageous to use the one-vs-one approach. Though it is necessary to train more classifiers than with the one-vs-all method, the training time decreases, because the training data set for each classifier is much smaller and the training time decreases more than linearly with the number of training samples [2, 60].

2.4 Decision Trees and Extremely Randomized Trees for Classification

Tree based models are widely used as they are quite simple and can be used for classification or regression problems [13, 61]. The conventional *decision tree* is a model, which uses a set of binary decision rules to compute a target value. At each node in the decision tree a specific attribute of the input sample is tested. If we consider the classification problem, the decision tree classifies new data samples by checking an attribute at each node at each step, and choosing the subsequent branch according to the concrete value of the attribute [13, 61]. This is done until a leaf node is reached, where each leaf node corresponds to a specific class. This can be seen as partitioning the input space into cuboid regions (each region corresponds to a certain class), where the edges are aligned with the axes, and each node corresponds to a certain partitioning of the input space (as shown in Figure 2.10(b)) [13].

If we consider the example in Figure 2.10(b), the first step divides the whole input space into two regions, where θ_1 is a parameter of the model. This corresponds to the *root node* in Figure 2.10(a). For a new sample x, the region it falls into is determined by starting at the root node of the tree and following a specific path down to the *leaf node*, according to the decision criteria at each node. For the regression problem, the leaf nodes contain the predictive values (e.g. the average of the target-values of those data points that fall in a specific region) [13].

To learn such a model from training data, the structure of the tree and the mode of construction have to be determined [13]. In particular, the attribute (one 'attribute' corresponds to one



(a) (b)

Figure 2.10: Example of a conventional Decision Tree. (a) The binary classification tree. (b) The corresponding partitioning of the (2-dimensional) input space.

input-dimension) and the corresponding threshold variable (alias cut-point) have to be chosen for each node, a decision to stop (make a leaf node) or to add nodes again has to be made, and at last the leaf nodes have to be assigned to a specific class (or value) [13]. There are various algorithms and techniques of addressing these issues, e.g. CART [66], ID3 or C4.5 [61]. Basically, the trees are constructed greedy-wise top-down, starting with a single root node. The nodes are added one after another, where at each step the attribute and the corresponding threshold value are chosen. If some stopping criterion is fulfilled (e.g. a fixed number of nodes), the algorithm stops [13].

2.4.1 Tree-Based Ensemble Methods

Beside the above mentioned *decision trees*, the so-called *ensemble model* approach exists [15]. In this technique, many classifiers are generated and their results are aggregated [15, 55]. An ensemble model, using decision trees, is also called *forest*. There are many different tree-based ensemble methods, where one of the most popular approaches are *random forests* [15, 55]. One popular method of ensemble learning is called boosting, where successive trees add an extra weight to samples, which have been incorrectly predicted by earlier trees. This way the final predictions are obtained by a weighted vote of all trees [55]. A second well-known method is called bagging, where every decision tree is independently constructed using a random subset of the training data, which means that for every decision tree a different subset is used for construction. The final predictions are obtained by using a simple majority vote [15, 55]. Random forests use this bagging-technique, with an additional randomization step. In standard decision trees, the cut-point of a new node is chosen by taking the best cut-point among all attributes. In random forests, on the other hand, at each node a random subset is sampled of all attributes and the cut-point is chosen by taking the best cut-point among this subset of attributes.

2.4.2 Extremely Randomized Trees

So far, simple decision trees and ensemble methods have been discussed. The approach, which is described in this section, is called *Extremly Randomized Trees* and is similar to the random

Input: a subset S
Output: a split s_* or nothing
if $|S| < n_{min}$ **then**
  Select K attributes $\{a_1, ..., a_K\}$ randomly;
  for $i \leftarrow 1$ ***to*** K **do**
    Select a random cut-point s_i for a_i;
    Compute a score for s_i;
  end
  Return the split s_* with the highest score;
else
  Stop Splitting;
end

Algorithm 2.1: Pseudo-code illustrating how to split a node in the Extra-Trees algorithm described in Section 2.4.2.

forest method [36]. In the following, the main differences and ideas of the algorithm are pointed out.

Extra-Trees algorithm

The *Extra-Trees algorithm* was introduced in 2006 by Geurts et al. [36] and is at least as good as the *random forest* approach, as mentioned in [36, 43]. The main differences in comparison with other tree-based ensemble methods are that the whole training set is used to build a tree and the cut-point of a node is chosen completely at random [36].

The trees are constructed as mentioned in the previous section: greedy-wise top-down [36]. In every step, the first decision has to be whether to continue with the splitting procedure, or to build a leaf node. If the number of training samples $|S_i|$, corresponding to a certain node $node_i$, is above the parameter n_{min}, splitting occurs (as shown in Pseudo-Algorithm 2.1) [36]. The steps shown in Algorithm 2.1 are performed recursively for the resulting sub-nodes (until the conditions for splitting are not fulfilled anymore for any node), where S is split into the subsets S_l and S_r, corresponding to the split s_* [36].

Altogether M trees are constructed, whereby the whole training-set is used. For the final predictions, the predictions of the trees are aggregated by majority vote in classification tasks and by arithmetic average in regression tasks [36]. Further details of the algorithm and its properties can be found in Geurts et al. [36].

Extremely Randomized Trees: Hyper-Parameters

M, K and n_{min} are the hyper-parameters, which have to be determined for this method. A more detailed explanation is given in Geurts et al. [36].

- The parameter M specifies the number of trees. As stated in Breiman [15] and Geurts et al. [36], a higher value of M leads to a greater prediction-accuracy. The choice of

M is therefore determined by computational requirements and considerations concerning accuracy.

- At each node, which is split, K attributes are selected randomly. The smaller K is, the higher is the randomization of the trees [36]. For example if $K = 1$, the splits are chosen totally at random and independently from the output variable (because the score is irrelevant).

- n_{min} denotes the minimum number of samples, which are required for splitting a node [36]. A higher value leads to smaller trees, since the splitting is stopped more quickly. Smaller trees in turn can lead to a reduced accuracy and at the same time to a reduced risk of overfitting.

2.5 Summary

In this chapter, some general machine learning issues, like preprocessing, model selection and overfitting, have been introduced. Beyond the general part, three state of the art approaches have been described. The K-nearest neighbor approach exhibits no explicit training phase and classifies a new (unlabeled) data point according to the labels of the k nearest neighbors in the training set. In contrast, the training stage of an SVM involves the optimization of a convex function, where the general objective is to find a hyperplane which separates the data optimally. The extremely randomized tree approach is a tree-based ensemble method that trains multiple classifiers and aggregates their outputs to get the final predictions.

CHAPTER 3

Deep Learning

In this chapter, a general introduction to deep learning is given. The main ideas of deep architectures are discussed, motivations for using them are outlined and it is described why deep learning approaches can be advantageous. This is necessary in order to understand the main ideas of deep learning approaches, which in turn helps to handle and interpret the results presented in Chapter 5 and Chapter 6.

Beside the focus on this general part, this chapter also covers the deep learning approaches which are evaluated in this thesis. In Section 3.2, *Deep Neural Networks* [7, 39] are discussed, where the content of each subsection builds upon the previous subsection and helps to understand the subsequent one. First of all, the basic concepts of a single neuron are described in Section 3.2.1. Subsequently, this knowledge is used to explain how neurons are connected together in order to form a whole neural net in Section 3.2.2. The question, how to train the parameters of a network, is covered in Section 3.2.3. Since the parameters of the network are trained in respect to a specific target, three different target functions are discussed in Section 3.2.4. Due to the fact that regularization can be used to improve the training (respectively the performance of a neural network), the techniques which are used in this thesis are explained in Section 3.2.5. These sections are necessary in order to understand neural network approaches in general and are therefore also essential for the comprehension of deep neural networks and their results in Chapter 5 and Chapter 6.

Since auto-encoders are used for the unsupervised training step of deep neural networks in this thesis, they are introduced in Section 3.2.6. Subsequently, techniques that force auto-encoders to learn useful features, are explained in Section 3.2.7. In Section 3.2.8, problems and limitations of the conventional training procedure are outlined. This serves as motivation for the subsequent section, where it is described how auto-encoders can be used for a pre-training of the network in order to handle these problems, respectively to overcome them. Lastly, the hyper-parameters of the deep neural network are discussed in Section 3.2.10.

Since the *Random Recursive Support Vector Machine* [82] is also a deep learning approach which is evaluated in this thesis, the basic idea, structure, training and motivation of this method

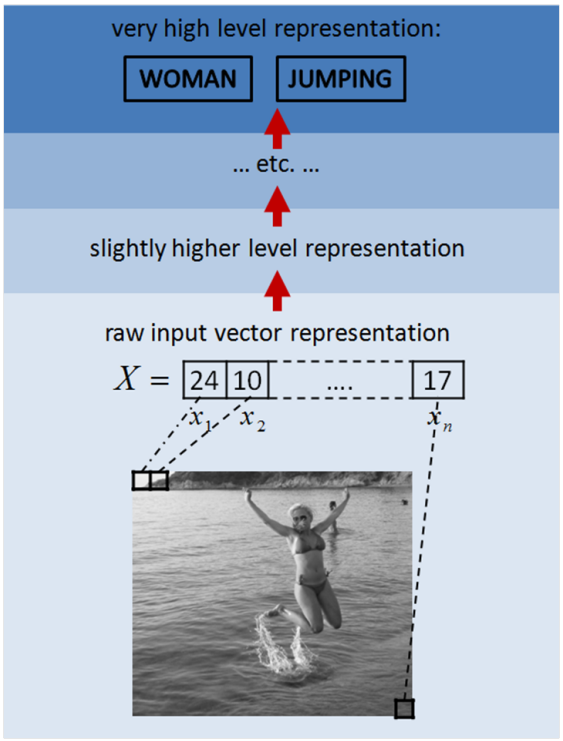


Figure 3.1: Example illustrating the concept of deep learning architectures. The idea is to transform the raw input image into gradually higher levels of representation (e.g. from edges to object-parts up to the category WOMAN), where higher-level representations are more abstract functions of the input than lower-level representations [7]. In this figure, a darker shade denotes a higher-level representation and a brighter shape denotes a lower-level representation.

is discussed in Section 3.3. Finally, the *K-means deep learning algorithm* [22] is introduced in Section 3.4, due to the fact that this method is evaluated in Chapter 5 and Chapter 6, too.

3.1 Deep Architectures

There are levels of abstraction and representation in deep architectures, which allow to model high-dimensional, non-linear data such as images, audio and text [7]. Here, abstractions can be understood as categories (like 'WOMAN' in Figure 3.1) or as features (a mapping of input data), where features can be discrete (e.g. input pixel is 'bright' or 'dark', depending on the intensity value) or continuous (e.g. the size of an object in the input image) [7]. An example for a typical structure of a deep architecture is shown in Figure 3.1.

The abstraction 'WOMAN' can be seen as a high-level abstraction built upon lower-level abstractions (e.g. object-parts), which in turn are built upon even lower level abstractions (e.g. detected edges) and so on. Lower-level abstractions are more directly tied to particular input dimensions, whereas the connection of higher-level abstractions to the input is more remote

(through other intermediate-level abstractions) [7].

The main focus of deep learning is to automatically discover abstractions from low-level features to high-level representations. Ideally this learning procedure is done with as little human effort as possible and without hand-engineering of features [7]. There are several motivations and advantages of deep architectures:

- Bengio [7] claims, that insufficient depth can hurt: When a function can be compactly represented by a deep architecture, the same function could require an extremely large architecture if the depth of this architecture is made more shallow.

- The brain has a deep architecture too: For example, the visual cortex shows levels of abstraction with its sequence of processing stages (from simple edge-detection in lower-levels to more complex visual shapes in higher-levels) [27]. This deep architecture is a very efficient representation, since only 1-4% of the neurons are active simultaneously at a given time [54]. This efficient representation is called *sparsity*.

- *Multi-task learning* deals with sharing (learned) representations across tasks. This sharing can be beneficial to boosting the performance, in particular where the model is suffering from a low number of labeled training data. Deep architectures naturally provide such a re-use of components, due to the multi-level structure [7, 27]. Low-level features, such as edge-detectors, and intermediate-features, like object parts, can be useful for a large group of visual tasks (and not only for the task 'detecting WOMAN').

3.2 Deep Neural Networks for Classification

Artificial neural networks are inspired by biology, more precisely by the neural net of the human brain (McCulloch and Pitts [59]). In order to be able to model non-linear data, first of all, general concepts of neural networks, like neurons, weights and activation functions are explained in this section. After that, more specific variants of neural networks and methods of how to design and train them are described. Since there are many different attempts to design and train neural networks [7, 13], to address all of them would go beyond the scope of this work. Therefore this section only contains specially selected topics, which are most relevant in respect to implementation and evaluation.

3.2.1 Basic Concepts

Neural networks are made up of single *neurons*, which are connected with each other and therefore form the neural network [46]. A neuron is the basic building block of a neural network and can be seen as a simple model itself. The basic structure of a single neuron is shown in Figure 3.2. Considering the example in Figure 3.2, the neuron takes multiple inputs x_i and computes the output as follows:

$$output = f(\sum_{i=1}^{3} w_i x_i + b) \tag{3.1}$$

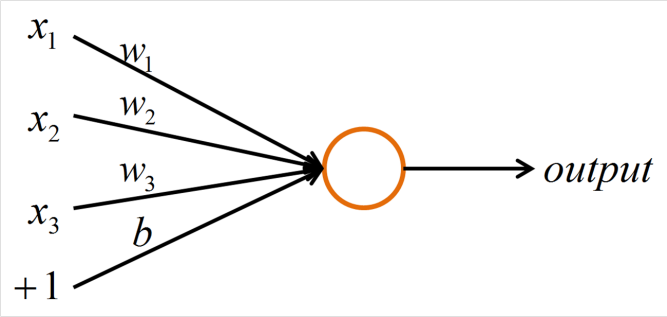
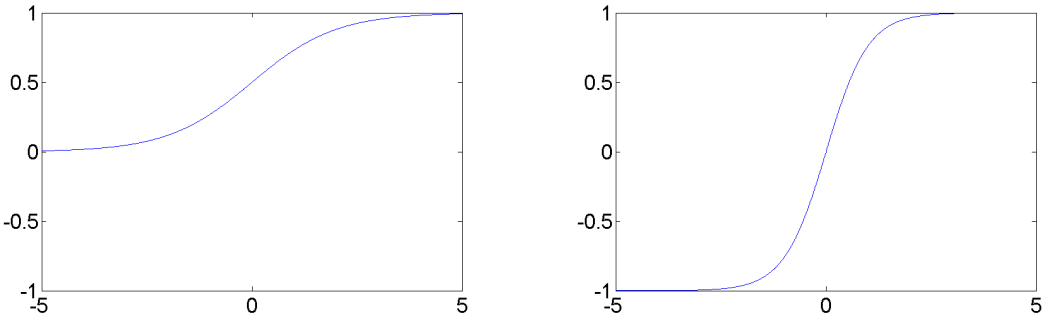


Figure 3.2: Illustration of a single neuron with four input values x_1, x_2, x_3, b.



(a) (b)

Figure 3.3: Plots of nonlinear activation functions. (a) 'Logistic sigmoid' function. (b) Hyperbolic tangent ('tanh') function.

where the parameters w_i are defined as *weights*, the parameter b as *bias* and $f(\cdot)$ is a nonlinear *activation function* [13]. The output of a neuron is also called *activation*.

In this computation process, every input value x_i is weighted and therefore multiplied by an individual weight w_i. The weighted input values and the bias b are summed up, where the *bias* parameter allows to add an offset to the data. The result of this linear combination is transformed using a nonlinear activation function $f(\cdot)$ to get the final output of the neuron [46]. In order to be able to find nonlinear mappings, this introduction of non-linearity is necessary [7]. Common choices in machine learning for the nonlinear activation function are the logistic sigmoid function:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{3.2}$$

or the hyperbolic tangent function:

$$f(x) = tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{3.3}$$

which are plotted in Figure 3.3 [13].

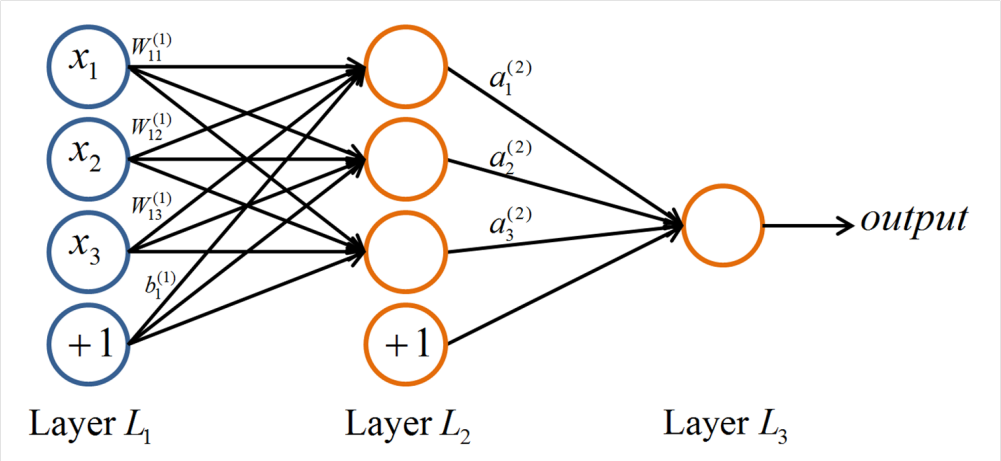


Figure 3.4: Example for a feed-forward neural network, also known as multilayer perceptron. The inputs are marked with blue circles, whereas the neurons are illustrated with orange circles. While the weight parameters are represented as links between the nodes, the bias parameters are denoted as links coming from additional nodes labeled with "+1". $a_i^{(l)}$ denotes the activation (meaning output value) of unit i in layer l. The parameters which are used in Equation 3.4 are displayed at the corresponding links.

3.2.2 Multilayer Perceptron

A neural network model consists of neurons, which are connected together [46]. The output of one neuron can be the input of another neuron, as can be seen in the simple example for a neural network model shown in Figure 3.4. This model is an example for a small *feed-forward* neural network also known as *multilayer perceptron*, in which the neurons are grouped into layers, each layer is fully connected to the next one (each neuron of a layer is connected with each neuron of the subsequent layer) and the connections do not form any closed directed cycles [13]. The inputs are also illustrated as circles (but in blue) and the circles labeled with "+1" correspond to the bias unit mentioned above. The bias units have no inputs, since they always output the value $+1$ [13]. Every link between the bias unit and the nodes has a bias parameter b_i, where $b_i * +1$ is used to compute the output of the corresponding neuron as described in Equation 3.1. While the leftmost layer is called *input layer*, the rightmost layer is called *output layer*. The layer in the middle is denoted as *hidden layer*, because its values are not observed in the training set [13,46]. The neurons (illustrated with orange circles) are also known as *units*, due to the fact that every neuron can be seen as a single computational unit.

To clarify how the output of a neural network is computed, the example model depicted in Figure 3.4 is examined in more detail. Here, the i-th layer is labeled as L_i and the number of layers in the example network is defined as $n_L = 3$ [1]. Thus, the first (input) layer is defined as

[1] As stated in [13], counting the number of layers is ambiguous. Hence the network in Figure 3.4 may be described as a 'three-layer network' (all layers are counted, including the input-layer), as a 'two-layer network' (because only the number of layers of adaptive weights is considered) or as a 'single-hidden-layer' network (only the

L_1, the second (hidden) layer as L_2 and the third (output) layer as L_3. The parameters of the model are $(W, b) = (W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)})$, where $b_i^{(l)}$ is the bias corresponding to unit i in layer $l+1$ and $W_{ij}^{(l)}$ denotes the weight parameter corresponding to the connection between the unit j in layer l and the unit i in layer $l+1$. Therefore, $W^{(1)} \in \mathbb{R}^{3\times3}$, $W^{(2)} \in \mathbb{R}^{1\times3}$, $b^{(1)} \in \mathbb{R}^3$ and $b^{(2)} \in \mathbb{R}^1$ in this example [13, 46].

Initially, activations of the hidden units are computed as:

$$a_1^{(2)} = f(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + b_1^{(1)}) \tag{3.4}$$

$$a_2^{(2)} = f(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + b_2^{(1)}) \tag{3.5}$$

$$a_3^{(2)} = f(W_{31}^{(1)}x_1 + W_{32}^{(1)}x_2 + W_{33}^{(1)}x_3 + b_3^{(1)}) \tag{3.6}$$

where a_i^l denotes the activation of unit i in layer l and $f(\cdot)$ is a non-linear activation function, which is introduced in Section 3.2.1 [13, 46]. The activations of the hidden units are then fed to the next layer (the output layer), which takes these activations as input. The output of this last layer is then computed as:

$$output = a_1^{(3)} = \sigma(W_{11}^{(2)}a_1^{(2)} + W_{12}^{(2)}a_2^{(2)} + W_{13}^{(2)}a_3^{(2)} + b_1^{(2)}). \tag{3.7}$$

where the specific notation σ emphasizes that the activation function for the output layer can differ from the activation function in the hidden units. The choice of σ is determined by the problem setting [13], and is discussed in detail in Section 3.2.4.

Summarizing, if we define x as the vector of inputs, $a^{(l)}$ as the vector of the unit-activations in layer l and extend $f(\cdot)$ to be applied element-wise (e.g. $f([x_1, x_2]) = [f(x_1), f(x_2)]$), the overall network function can be written as:

$$output = a^{(3)} = \sigma(W^{(2)}a^{(2)} + b^{(2)}) = \sigma(W^{(2)}f(W^{(1)}x + b^{(1)}) + b^{(2)}). \tag{3.8}$$

The process of evaluating this function is called *forward propagation*, due to the fact that the inputs are 'forwarded' through the network [46]. The example neural network presented in this section can easily be generalized by considering additional layers, varying the number of units in a layer or the structure of connectivity between the neurons [13]. For example, a deep architecture can be achieved by adding multiple hidden-layers to the examined example. It is important to note that the non-linear activation function is necessary in a deep network, because the composition of consecutive linear transformations is itself only a linear transformation, as stated in [13]. Furthermore, in each hidden layer, the number of hidden units has to be chosen. A network can also have multiple output units, for example in the case of multi-class classification, multiple binary classification problems or regression with n-dimensional target values.

3.2.3 Gradient Descent and Error Backpropagation

Up to now, only the architecture of neural networks and the process to compute the output has been described. The key problem in neural networks is the number of parameters (W, b) and

hidden layers are counted).

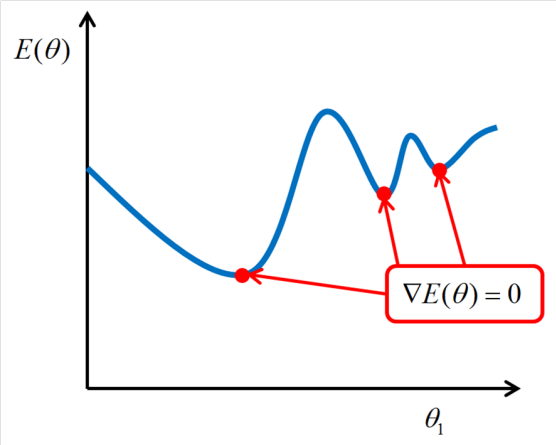


Figure 3.5: Example for a non-convex error function, if the parameter space is one-dimensional. The two red points on the right label *local minima*, whereas the red point on the left denotes the *global minimum*.

how to find suitable parameter values for a given problem [13]. Though there are many ways to train a neural network, this work focuses on training a network using *error backpropagation* and *gradient descent*. This approach is able to deal with various output-unit activation and error functions, where the error function measures the discrepancy between the desired output and the output of the model [13]. The objective of the training is to find parameter values for the weights and biases, which minimize the chosen error function $E(W, b)$. Because the choice of the error function can affect the properties and the performance of the network, it is discussed more detailed in Section 3.2.4 [51].

For better readability, we use $\theta = [W, b]$ to denote the parameters of the network, so $E(W, b) = E(\theta)$. The error function $E(\theta)$ is a smooth continuous function of all weights and biases and can be seen as a surface in the parameter-space, as shown in Figure 3.5 (for the case of a one-dimensional parameter space) [13]. While the aim of training is to find a parameter configuration which minimizes the error function, this cannot be achieved by finding an analytical solution for the equation $\nabla E(\theta) = 0$[2]. This is because the error function is a non-convex one which has many *local minima* beside the *global minimum* [13]. As stated in [13], iterative numerical procedures can be used to find a sufficiently good solution. Most techniques choose an initial value for the parameter-vector $\theta^{(0)}$ and perform iterative steps (to find a solution) of the form

$$\theta^{(k+1)} = \theta^{(k)} + \Delta\theta^{(k)} \tag{3.9}$$

where k denotes the iteration step [13]. While there are different attempts to find the update-value of $\Delta\theta^{(k)}$, the *gradient descent algorithm* presented in this section uses gradient information to do this. In particular, a single iteration step involves a small step in the direction of the

[2]The nabla operator ∇ denotes the vector of the partial derivative operators. For example if we have a three-dimensional parameter space with $(\theta_1, \theta_2, \theta_3)$, the nabla operator is defined as $\nabla = (\frac{\partial}{\partial\theta_1}, \frac{\partial}{\partial\theta_2}, \frac{\partial}{\partial\theta_3})$ [16].

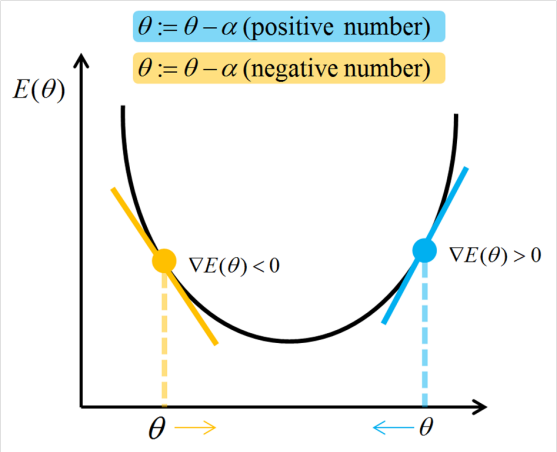


Figure 3.6: An example which illustrates the basic idea of the gradient descent algorithm. Two different initial values for the parameter are labeled with distinct colors (the left one is highlighted in 'orange' and the right one in 'blue'). Both parameters are updated in the direction of the greatest rate of decrease of the error function, towards the minimum.

negative gradient of the form

$$\theta^{(k+1)} = \theta^{(k)} - \alpha \nabla E(\theta^{(k)}) \tag{3.10}$$

where $\nabla E(\theta)$ points in the direction of the greatest rate of increase of the error function and $\alpha > 0$ is called *learning rate* [13]. These steps of changing the parameters of the neural network are performed until a minimum is reached (preferably the global minimum or a solution close to it). The basic idea of the gradient descent algorithm is illustrated in Figure 3.6, where one update step of two different initial parameter values is illustrated. In both cases, the parameter θ is moved in the direction of the greatest decrease of the error function. While the 'orange' one is moved to the right, because $\nabla E(\theta^{(k)}) < 0$, the 'blue' parameter is moved to the left, because $\nabla E(\theta^{(k)}) > 0$. Since α controls the step size of the algorithm, a learning rate, which is too small, can lead to a slow convergence of the algorithm. On the other hand, if the learning rate is too large, the gradient descent algorithm can diverge [9, 51].

After each update step, it is necessary to compute $E(\theta)$ for the new parameter vector $\theta^{(k+1)}$ [9]. In the simple *batch gradient descent* method this is done with respect to the whole training set, so every parameter update involves the whole training set being processed in order to evaluate $E(\theta)$ [9]. In contrast, the *stochastic gradient descent* method uses only one training example at a time to evaluate $E(\theta)$, where the data example is either chosen by cycling through the training set in sequence or selecting at random [9]. Another approach is *mini-batch gradient descent*, where $E(\theta)$ is evaluated by using B data examples. If $B = 1$ this method is equal to stochastic gradient descent, whereas with B set to training set size it is equal to batch gradient descent. More details can be found in [13] and [9].

Since the evaluation of the gradient of an error function in a feed-forward network is necessary in every update step of the gradient descent algorithm, it is important to do this effi-

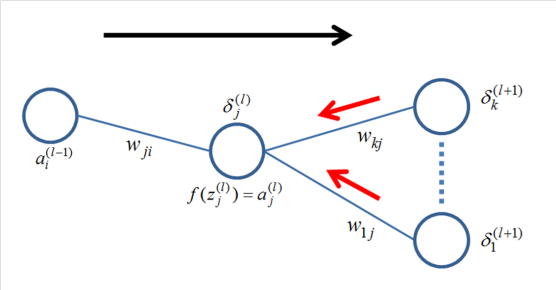


Figure 3.7: Illustration of computing the error $\delta_j^{(l)}$ for the unit j in the (hidden-) layer l. While the black arrow indicates the direction of forward propagation, the red arrows denote the backward propagation of the errors δ.

ciently [13]. *Error backpropagation* is a technique, which provides such a computational efficient way for evaluating such derivatives [51]. Because a detailed discussion of this technique would go beyond the scope of this work, only the main ideas of error backpropagation can be discussed. The idea is to compute an 'error term' $\delta_i^{(l)}$ for each unit i in layer l in order to evaluate derivatives. This error term can be seen as how much this unit is 'responsible' for any errors in the output [13]. For the output layer, $\delta_i^{(n_L)}$ can be computed directly by measuring the difference between the network-activations and the true target values. The error terms for the hidden units on the other hand can not be computed directly and are therefore obtained by propagating the errors δ backwards through the network. This procedure of backpropagation is illustrated in Figure 3.7. In the end, the activations and error terms are used to compute the required derivatives. In Algorithm 3.1, the steps of error backpropagation are shown in a pseudo-algorithmic form. More details about error backpropagation can be found in [13, 51, 56].

3.2.4 Output- and Error-Functions

The backpropagation algorithm allows to compute the gradient of various error functions and the gradient descent method uses these gradients to perform parameter updates [13]. The objective of this training procedure is to find a parameter configuration which minimizes the given error function. Since the choice of the error function is interrelated with the selection of the output-layer activation function σ and therefore the problem setting, the choice of the activation function is discussed too [13]. Although the choice of the error function, with respect to the particular activation function, can be statistically motivated, a statistical discussion is omitted because this would go beyond the scope of this work. Statistical issues can be found in [13].

For standard regression problems, the output-layer activation function is the identity $\sigma(x) = x$, which is also called *linear activation function*, due to the fact that the output is then a simple linear combination of the inputs [13]. The linear activation function does not constrain the output to a specific range and is therefore suitable for regression problems. Assuming that the training data set is given as $\{x_1, x_2, ..., x_n\}$, with the corresponding output vectors as $\{y_1, y_2, ..., y_n\}$ and

Step 1: Perform a feed-forward pass to find the activations of all hidden and output units. In other words, apply an input vector x_i to the network and do forward propagation

Step 2: Evaluate the 'error term' $\delta_i^{(n_L)}$ for every output unit

Step 3: Backpropagate all 'error terms' δ through the network by using the 'backpropagation-formula'

$$\delta_j^{(l)} = f'(z_j^{(l)}) \sum_k w_{kj}^{(l+1)} \delta_k^{(l+1)}$$

where $f'(\cdot)$ is the inverse of the activation function of unit j in layer l and z_j denotes the total weighted sum of inputs to unit j in layer l ($z^{(l)} = W^{(l-1)} a^{(l-1)} + b^{(l-1)}$).

Step 4: Use the activation $a_i^{(l-1)}$ and the error term $\delta_j^{(l)}$ to obtain the gradient of the error function with respect to the weight vector w_{ij}. The formula to obtain the derivative is: $\dfrac{\partial E(W,b)}{\partial w_{ji}^{(l)}} = \delta_j^{(l+1)} a_i^{(l)}$

Step 5: Repeat by going to step 2

Algorithm 3.1: Pseudo-code illustrating the steps of error backpropagation.

the output of the network is defined as $h_{W,b}(x)$, the error function can be defined as:

$$SSE(\theta) = \frac{1}{2} \sum_{i=1}^{n} \|h_{W,b}(x_i) - y_i\|^2 \tag{3.11}$$

which is known as the *sum-of-squares error function* [13].

For binary classification problems, the activation function of the output unit is chosen to be the *logistic sigmoid function*

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3.12}$$

where the two classes are denoted as $\{0, 1\}$ [3] [13]. If $y = 1$ is defined as label for the class C_1 and $y = 0$ denotes class C_2, then the output $h_{W,b}(x)$ can be interpreted as the class membership probability for C_1. The class membership probability for C_2 is then given as $1 - h_{W,b}(x)$. In this case, the error function can be chosen as a *cross-entropy error function*:

$$CE(\theta) = -\sum_{i=1}^{n} \{y_i \ln h(x_i) + (1 - y_i) \ln(1 - h(x_i))\} \tag{3.13}$$

[3] It is also possible to choose the *hyperbolic tangent function*, where it is necessary to choose $\{-1, 1\}$ representing the two class labels (because the output of the network is no longer constrained to $0 \leq h_{W,b}(x) \leq 1$, but to $-1 \leq h_{W,b}(x) \leq 1$) [13].

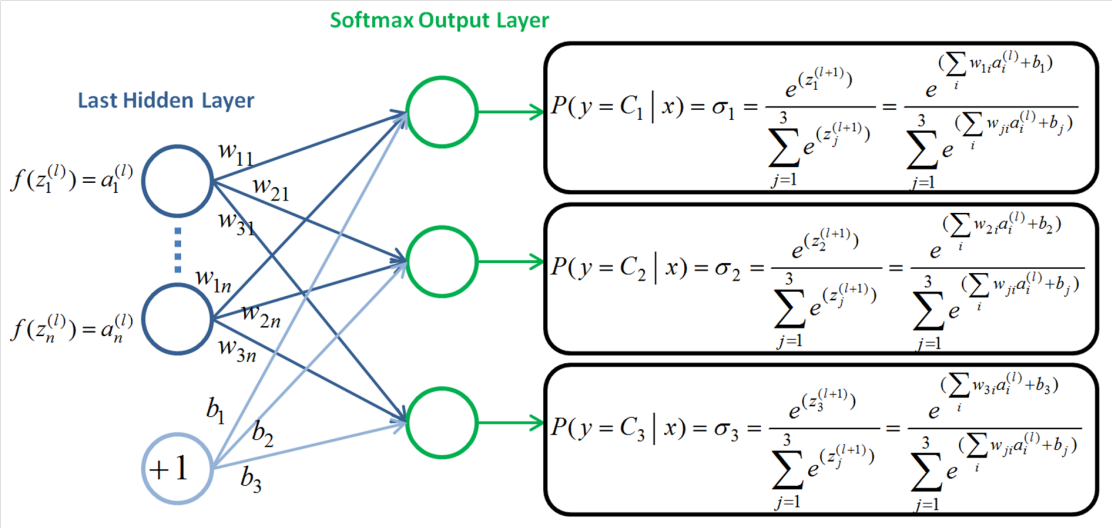


Figure 3.8: Illustration of a network using the *softmax activation function* σ_i in the last layer. For a better overview, only the hidden layer next to the output-layer is shown, though a neural network can have multiple hidden layers.

where $h(x_i)$ denotes $h_{W,b}(x_i)$ [13]. Since a network can have more than one output-unit, this activation function can be used for every output unit to deal with multiple binary classification problems [4] [13]. In this case, each output unit can be seen as a binary class label.

In the case of multi-class classification problems, each input example is assigned to exactly one of k classes [13]. Since every output unit is associated with one class, in a k-class classification problem, the network has k output units. Because each output unit is assigned to a specific class and the network outputs are interpreted as the class membership probabilities ($p(y = j|x)$, where $j = 1, ..., k$), the activation function must fulfill the constraints $0 \leq \sigma(x)_i \leq 1$ and $\sum_{i=1}^{k} \sigma(x)_i = 1$, where $\sigma(x)_i$ is defined as the output of the output-unit i [13]. Therefore the *softmax activation function* can be used in the last layer:

$$\sigma_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} \tag{3.14}$$

where k is the number of classes (respectively output-units) and z_i denotes the total weighted sum of inputs of the output-unit i [13]. Considering the example, shown in Figure 3.8, with three classes $\{C_1, C_2, C_3\}$, where an input vector x is fed to the neural network, which has three output units. Each output unit is associated with a certain class. If the output values are $(0.1, 0.7, 0.2)$, this can be interpreted as the probability of '0.1' that the input x takes the class label C_1, '0.7' that it takes the class label C_2 and '0.2' that it takes the class label C_3. The final prediction can be obtained by choosing the class with the highest probability allocated. In

[4] If the classes are mutually exclusive, the problem setting is called multi-class classification. On the other hand, if the classes are not mutually exclusive, the setting is called multiple binary classification problem and it would be appropriate to use the logistic sigmoid function [13].

this example, the final prediction would be C_2. For the multi-class classification problem, a reasonable error function is the *multi-class cross-entropy error function*:

$$MCE(\theta) = \sum_{i=1}^{n} \sum_{j=1}^{k} \delta_{y_i j} \ln \sigma_j \tag{3.15}$$

where $\delta_{y_i j}$ is the Kronecker delta [13]:

$$\delta_{y_i j} \begin{cases} 0 & \text{if } y_i \neq j \\ 1 & \text{if } y_i = j \end{cases} \tag{3.16}$$

3.2.5 Avoid Overfitting - Regularization

The number of input and output units is usually determined by the data set and the desired prediction task, whereas the number of hidden units and the number of hidden layers can be chosen [13]. The number of hidden units controls the number of adjustable parameters and therefore the capacity of the model. A first approach to control the complexity of the model would be to choose an optimal number of hidden-units, which avoids both underfitting and overfitting [9].

A more common way to avoid overfitting, as recommended in [9], is to choose the number of hidden units just large enough and add a *regularization term* to the error function:

$$\tilde{E}(\theta) = E(\theta) + \lambda \sum_{i} (\theta_i)^2 \tag{3.17}$$

where λ is called *regularization coefficient* and controls the relative importance of these two terms (and therefore the complexity of the model). The regularization term is also called *weight decay*, because it encourages weights of the model to decay towards zero [9]. This adjustment of the training criterion controls the complexity of neural networks and aims to improve the generalization error. Adding a regularization term of the form $\lambda \sum_i (\theta_i)^2$ is called L2-regularization, whereas in L1-regularization the term $\lambda \sum_i |\theta_i|$ would be added to the error function [9].

Early stopping is another way to control the complexity of the model and to avoid overfitting. The idea of this technique is to use a validation set to measure the performance of the model during the training process [7]. Considering the gradient descent method, which was introduced in a previous section, the value of the error function is iteratively reduced in the training process. Up to a point, the measured error on the validation set will decrease, while after that point, the measured error will increase due to overfitting on the training data. The idea is now to use the measured performance to stop training prior to convergence (for instance when the validation error reaches its smallest value) [9].

To summarize, early stopping can be seen as "stop before reaching a bad solution", adding a regularization term to the error function as "move to a good solution" and choosing an optimal number of hidden units can be interpreted as "allow only good solutions" [7, 9, 13].

3.2.6 Auto-Encoders

So far, only the supervised learning setting, including labeled training examples, has been discussed. Though deep architectures theoretically have advantages, the training of deep neural networks is more complicated than the training of shallow ones, as stated in [7]. Starting from random initializations (of weights and biases) and using conventional supervised learning algorithms, the solutions of deeper networks performed worse than solutions obtained for networks with one or two hidden-layers [10]. In 2006, Hinton et al. [39] discovered, that much better generalization results could be obtained, if each layer is *pre-trained* using an unsupervised learning algorithm, instead of using random initialization. The idea of this approach is to train each layer unsupervised one after another, where the output of a trained layer is used as input of the subsequent layer in the training procedure [7]. This idea of pre-training is known as *greedy layer-wise unsupervised learning*, which is discussed in more detail in Section 3.2.9. After this unsupervised learning procedure, the network is trained using the conventional supervised learning algorithm, where this supervised training step is also known as *fine-tuning*. Though the results in [39], where unsupervised pre-training improved the performance of neural networks, have been obtained with Restricted Boltzmann Machines[5], subsequent experiments using *auto-encoders* yield similar results [10]. In this work, *auto-encoders* are described and used for unsupervised pre-training of the neural network.

In Figure 3.9 the basic structure of an auto-encoder is shown, considering an unlabeled training set $\{x_1, x_2, ...\}$, where $x_i \in \mathbb{R}^n$, the auto-encoder can be seen as an unsupervised learning algorithm, which sets the target values equal to the inputs (i.e. $h_{W,b}(x_i) = x_i$) [7]. Since the input values are interpreted as target values, the auto-encoder can be seen as a neural network trained to output a reconstruction of the input. The first part of the auto-encoder is called *encoder*:

$$a^{(2)} = f(W^{(1)}x + b^{(1)}) \tag{3.18}$$

where $f(\cdot)$ is the activation function of the encoder [6]. The second part of the auto-encoder is called *decoder*:

$$h_{W,b}(x) = g(W^{(2)}a^{(2)} + b^{(2)}) \tag{3.19}$$

where $g(\cdot)$ is the activation function of the decoder. Typical choices for the encoder and decoder activation functions are the sigmoid, the hyperbolic tangent or the identity function ($g(x) = x$) [6]. If the inputs are not constrained to lie in the range of $[0, 1]$ (respectively $[-1, 1]$), it is a natural choice to use a non-linear activation function in the encoder and linear activation function in the decoder [6]. This configuration of an auto-encoder, where the outputs are unbounded and can therefore produce values greater than 1 or less than 0 as well, is called *linear decoder* [7].

The parameters $\{W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}\}$ of the model are trained simultaneously on the task of reconstructing the original input as well as possible [6]. Error backpropagation and the gradient descent algorithm are usually applied to train the auto-encoder and to find suitable values for the model-parameters $\{W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}\}$ (e.g. using the sum-of-squares error function). To avoid learning the simple identity function, various constraints can be added to the

[5]The Restricted Boltzmann Machines (RBMs) are generative models, where in [39] the pre-trained RBM-layers were stacked together to form a so-called Deep Belief Net (DBN). Descriptions of RBMs and DBNs can be found in [7], [6] or [48]

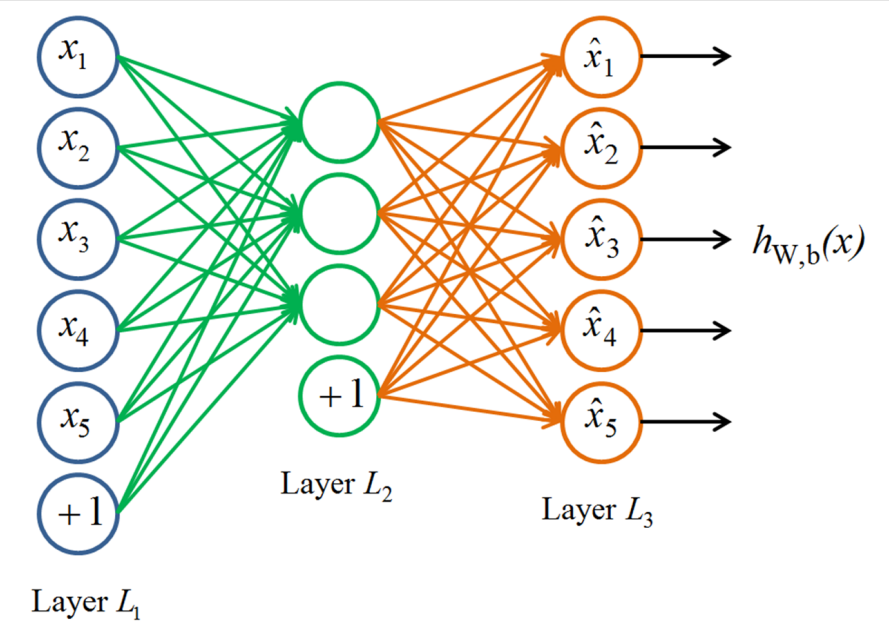


Figure 3.9: Basic structure of an auto-encoder. In the hidden layer L_2 the activation function $f(\cdot)$ and in the output layer L_3 the activation function $g(\cdot)$ is used, as described in the text. The first part of the network is called *encoder* (drawn in green) and the second part of the auto-encoder is known as *decoder* (highlighted in orange).

network [7]. The main idea of these constraints is to force auto-encoders to learn useful representations, respectively meaningful features, of the data. One approach consists of constraining the hidden-layer to have a limited number of hidden units [6]. For example, if 20-by-20 pictures are used as inputs ($x_i \in \mathbb{R}^{400}$) and the hidden layer L_2 is constrained to have only 100 units, the network is forced to learn a compressed representation of the input [6]. Since this simple auto-encoder must try to reconstruct the 400-dimensional input from the hidden-unit activations $a^{(2)} \in \mathbb{R}^{100}$, it learns a low dimensional-representation of the data. Nevertheless there are alternative ways of constraining the auto-encoder to learn meaningful features, which allow to use an *over-complete*[6] representation, as stated in [6]. These alternatives are also known as *regularized* auto-encoders and are discussed below [6].

3.2.7 Regularized Autoencoders

One way of constraining the auto-encoder to learn useful representations even with an over-complete basis, is to add a *sparsity* constraint [6]. The idea of sparse coding was originally introduced by Olshausen and Field in 1996 [65]. It is an unsupervised method, which learns a set of basis functions $\{\phi_1, \phi_2, ..., \phi_k\}$, sometimes called dictionary, so that each input vector x

[6] If the number of hidden units exceeds the dimension of the input space, the representation is denoted as *over-complete*.

of the training set can be represented as:

$$x = \sum_{i=1}^{k} a_i \phi_i \tag{3.20}$$

under the constraint that the a_i are mostly zero ("sparse"). For example, the input x_1 could then probably be represented as a linear combination of only three basis functions (e.g. $x_1 = 0.8 \times \phi_2 + 0.3 \times \phi_7 + 0.5 \times \phi_{14}$). In the context of *sparse auto-encoders*, sparsity is defined as having few hidden unit activations not close to zero or having few non-zero hidden unit activations [6]. Thus the sparsity constraint should enforce the auto-encoder to learn parameters $W^{(1)}$ and $b^{(1)}$, which give sparse features $a^{(2)} = f(W^{(1)}x + b^{(1)})$. The idea is to learn a set of basis-functions, which represent the input more compactly and give a higher-level representation than the raw input-values [7].

There are several ways to introduce sparsity into the representation learned by the auto-encoder. One approach is to penalize the hidden unit biases, to make these additive offset parameters more negative. This involves the risk that the weights could compensate for the bias, which in turn could lead to problems concerning the numerical optimization of parameters [9]. Another method involves directly penalizing the hidden unit activations with a *L1-penalty* to the error function:

$$E_{sparse}(\theta) = E(\theta) + \alpha \sum_{i=1}^{n} a_i \tag{3.21}$$

where n is the number of units in the hidden layer and α is called the *activation regularization coefficient* [9]. As stated in [6], there are also other variations of directly penalizing the hidden unit activations. For instance it is feasible to penalize the average activations (e.g. averaged over the training set or a mini-batch) and enforce them to tend towards a fixed target, instead of pushing the activations towards zero. In this case the average-activations are penalized if they differ from the chosen target [6].

Another way of forcing the auto-encoder to learn interesting structure in the data is to use *denoising auto-encoders*. Instead of introducing a sparsity constraint, the denoising auto-encoder is trained to reconstruct the original input from a corrupted version of it. Therefore the original input x is corrupted into $\tilde{x}$ (e.g. setting a random subset of the input-values to zero), as described in [80] and [81]. The corrupted version is used as the input of the auto-encoder, whereby the original uncorrupted inputs are defined as the target. The difference to the basic auto-encoder is that the denoising auto-encoder learns a function of $\tilde{x}$ rather than x. The main idea of this approach is that the learning algorithm has to learn meaningful and "robust" features (structure in the input distribution) in order to undo the effect of the corruption process. An informal reason for introducing this "robustness criterion" is that good representations should capture stable structures and characteristics of the input and partially destroyed inputs should lead to this robust representation [80]. Further information about denoising auto-encoders can be found in [80] or [81].

3.2.8 Problems in Training Classical Neural Networks

As mentioned in Section 3.2.6, deep networks, where the parameters are initialized randomly and the network is trained subsequently with the conventional supervised (gradient based) training method, give poor results [7]. Within this conventional technique, the parameters of the network are initialized randomly to break the symmetry between hidden units of the same layer [7]. There are several reasons, why this learning algorithm did not work well for deep neural networks, whereby this list is not intended to be exhaustive:

- This training algorithm relies on labeled training data. Since for some problems a vast amount of labeled data is not available, it is difficult to fit the parameters of a complex model and to avoid overfitting with a purely supervised training algorithm [7].

- Since training neural networks involves solving a non-convex optimization problem, gradient based training probably finds only local optima. As the architecture gets deeper, it gets more difficult to achieve good generalization because the number of local optima increases, as discussed in [7], [33] and [49].

- The weights in the lower layers change more slowly, due to the fact that the magnitude of the gradients (derived using backpropagation) rapidly decreases as the depth of the network increases. This problem is known as *diffusion of gradients* and can lead to a poor tuning of the lower layers, which may be another reason for the poor results, as stated in [7].

In the subsequent sections it is described, how deep neural networks attempt to handle these problems, respectively how they overcome them.

3.2.9 Stacked Autoencoders

In this section it is discussed, how auto-encoders can be trained (unsupervised pre-training and supervised fine-tuning) and stacked together to form a deep neural network.

In this work, *stacked auto-encoders* are used to implement the idea of *greedy layer-wise unsupervised learning*, also known as pre-training. The main idea of this pre-training stage is to shift the weights of the network in the "right direction", before applying the conventional supervised learning algorithm, to find a better local optimum [10,33]. Besides finding a better starting point of the parameters for fine-tuning, better models can be obtained by using additional, unlabeled data, since unsupervised learning does not rely on labeled data [49]. The unsupervised pre-training procedure is explained for sparse auto-encoders with an example, starting with the auto-encoder shown in Figure 3.9. At first, this auto-encoder is trained using error backpropagation and gradient descent to optimize the sparse error function $E_{sparse}(\theta)$, without the need for labels, as discussed in a previous section. Afterwards, the last layer of this trained network

[7] If all units of one layer would be initialized with the same parameter configuration (input and output weights), all units would compute the same output. As a result of this, all units would remain identical since the same gradient (and therefore the same update) would be computed for all units. In the end, all hidden units would learn the same function, thus wasting capacity. [9].

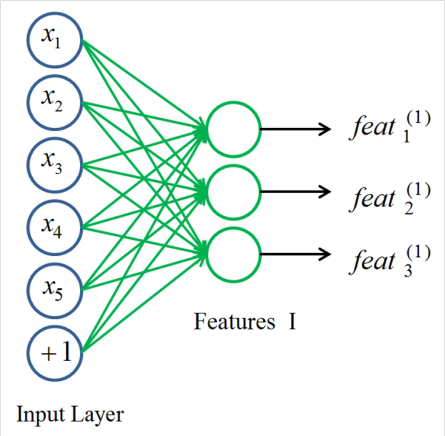


Figure 3.10: First hidden layer: This network is obtained if the last layer of the (trained) autoencoder shown in Figure 3.9 is removed. If the inputs are fed to this network, the outputs $feat^{(1)(i)}$, also known as features, should give a better representation of the inputs x_i.

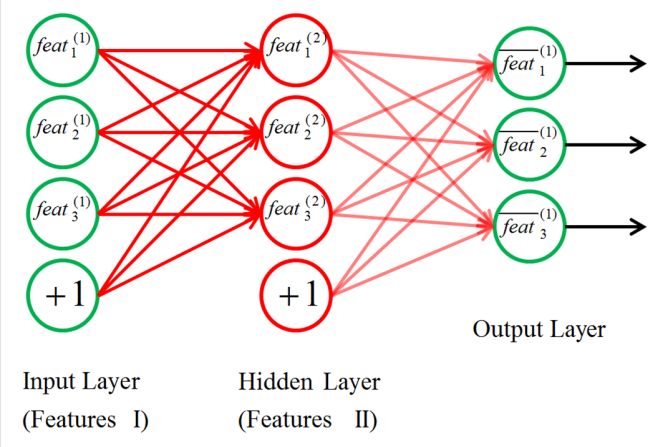


Figure 3.11: Second hidden layer training: The second auto-encoder, trained with the features obtained from the first auto-encoder. The parameters of the first auto-encoder are not changed throughout the training procedure of this second one. This way, the layers are trained *greedily, layer-wise*.

(the decoder) is removed, only the trained hidden layer with its parameters $\{W^{(1)}, b^{(1)}\}$ (the encoder) is kept and a network as illustrated in Figure 3.10 is obtained [3]. If an input sample x_i is fed into the network, the activations of the hidden layer $a^{(2)}$ should give a better representation of the example than the original input [7]. These activations are also called features and are denoted as $feat_j^{(l)(i)}$ in this example, to label the feature j in the l-th hidden layer for the input x_i. As can be seen later, the l-th auto-encoder corresponds to the l-th hidden layer in the final network.

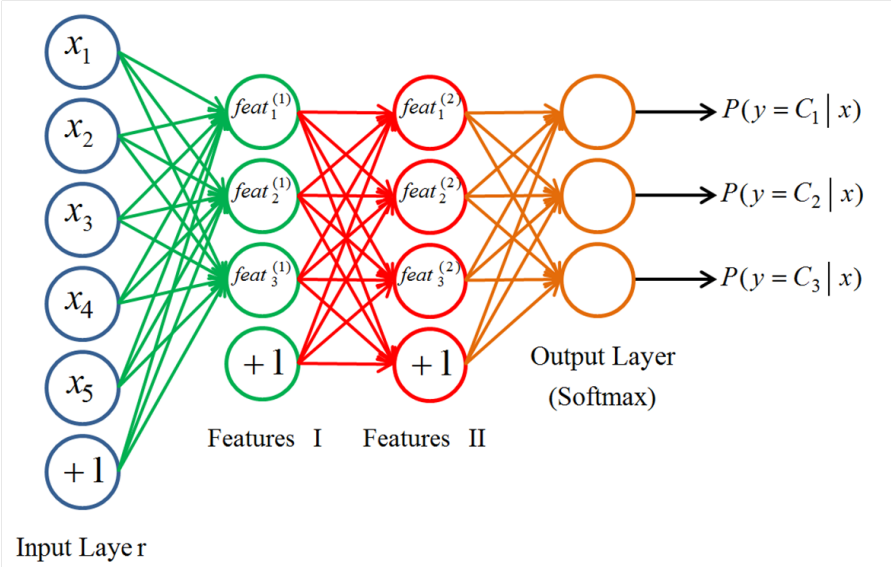


Figure 3.12: The final network formed by stacking the pre-trained hidden layers together and adding a final output layer (softmax-layer). Finally, *fine-tuning* of the whole network is performed.

The original features are fed to the network shown in Figure 3.10, to obtain the primary features $feat^{(1)(i)}$ for the input x_i. These features are new representations of the original input-samples and they are used as inputs for the next sparse auto-encoder, which is shown in Figure 3.11 [39]. This auto-encoder is trained the same way as the first one, where the resulting secondary features $feat^{(2)(i)}$ can be seen as an even higher-level representation [7]. The last layer of the second auto-encoder is removed again and the primary features are fed into this second sparse auto-encoder to obtain the secondary feature representations. In this way, the hidden layers of a deep neural network can be pre-trained one after another, where only the network parameters of the currently trained layer are updated and the output of the currently trained layer is used as input for the subsequent auto-encoder [33]. To form the final network, at first the pre-trained hidden layers are stacked together. Afterwards, the final layer is added (e.g. a softmax-layer in case of a multi-class classification problem) and the whole network is trained in a supervised fashion (with labeled data) [49]. This second training stage, where error back-propagation and gradient descent are used to update the parameters of all layers (and not only the weights of the final layer), is called *fine-tuning* [49]. In the concrete example considered in this section, the two pre-trained hidden layers are stacked together and a softmax-layer is added to form the final network, as can be seen in Figure 3.12.

3.2.10 Deep Neural Network: Hyper-Parameters

Though there are different attempts how to design and train neural networks, in this section the hyper-parameters corresponding to gradient-descent training of stacked auto-encoders are

described (according to [8, 9, 13, 49, 51]). The following list is not intended to be exhaustive.

- The *learning rate*, described in Section 3.2.3, controls the step size of the optimization algorithm and is the most important hyper-parameter, as stated in [9]. Following the explanation in [8], the optimal learning rate is usually close to the largest learning rate which does not cause divergence of the training criterion.

- It is possible to decrease the learning rate as the training process progresses. Since Bengio et al. [9] claim that the benefit of doing this is typically very small, the learning rate has been chosen to be constant over training iterations in this work.

- Another variant of the optimization procedure is to perform the current update step dependent on a weighted average between the current and the past gradients (instead of only using the instantaneous gradient). The *momentum* β controls how the moving average of the last gradients is weighted for the current optimization step (e.g. $\bar{g} = (1 - \beta)\bar{g} + \beta g$, where $\bar{g}$ is the weighted average and g the current gradient).

- The *mini-batch size* B of the mini-batch gradient descent algorithm, which is described in Section 3.2.3, is typically chosen to be between one and several hundred. The impact of B is also computational: a higher value leads to a longer training time, because more examples have to be "visited" for the same number of update steps. On the other hand, a higher value leads to a more 'deterministic' and robust optimization procedure.

- The *number of training iterations* T determines the maximum number of update steps which are performed during training. If early stopping is used for regularization, this parameter is not set explicitly, but will be found dynamically during the training procedure.

- The *number of hidden-units* has to be determined for each hidden layer in a neural network. This parameter controls the capacity of the model, whereby it is most important that the number of hidden units is not too small, as stated in [9]. If the number of neurons is larger than the 'optimal' value, the generalization performance is not affected much. As discussed in Section 3.2.5, regularization is especially necessary if the number of hidden units is larger than the number of input units. Additionally, it has been found out that in general all hidden layers should have the same size, but this may be data-dependent [49].

- If a regularization term is added to the error function to avoid overfitting, the *weight decay regularization coefficient* λ controls the importance of the regularization term (e.g. L2-regularization) and therefore the complexity of the model. Since L2-regularization plays the same role as early stopping, Bengio et al. [9] suggest to drop L2-regularization if early stopping is used. A discussion of regularization can be found in Section 3.2.5.

- As stated in [3], there is no clear recommendation in the literature which sparsity-constraint should be used and the L1-regularization of the activations seems the most natural choice [9]. Therefore, in this work sparse auto-encoders are used for pre-training of the hidden-layers, whereby the L1-penalty, which is described in Section 3.2.7, is applied to the hidden unit activations to achieve sparsity. The *activation regularization coefficient* determines the relative importance of the sparsity penalty term in the overall error function.

- If denoising auto-encoders are used for pre-training, hyper-parameters in relation to the corrupting process have to be determined, as explained in Section 3.2.7. For example, if the input has d dimensions and we would like to set a fixed number νd of randomly selected components to zero (this is called *masking noise*), the amount of destruction ν has to be chosen.

- The *non-linear activation function*, which is discussed in Section 3.2.1, of the neurons has to be chosen too. In this work the 'hyperbolic tangent' function is used, because they are common choices and there seems to be no clear recommendation which non-linear function is preferable in a specific situation [73].

- The *weights initialization scaling coefficient* r controls the range of the uniform distribution $U(-r, r)$, which is used to initialize the weights of the units. To avoid hyper-parameters related to this initialization, Bengio [9] recommends to use formulas to choose r, which are dependent on the number of inputs ('fan-in') and outputs ('fan-out') of a unit. This is based on the idea, that neurons with a higher number of inputs should have smaller weights.

3.3 Random Recursive Support Vector Machines for Classification

Linear Support Vector Machines have become a popular method for classification tasks in computer vision, but depending on the problem setting it is possible that they do not exhibit a good performance [13]. While their main advantage is the simplicity of training a linear model, linear SVMs often fail to solve more complex problems, which is the reason behind using non-linear kernels [82]. This in turn leads to another challenge: Finding the best kernel for a specific task, which is still an open problem. Current deep learning methods, such as neural networks, can find more compact representations, but require solving a difficult, non-convex optimization problem [82]. In contrast, the idea of the model presented in this section, is to combine the simplicity of the linear Support Vector Machine with the power of a deep architecture. [82]. In this section, a description of the Random Recursive Support Vector Machine (R^2SVM) is given.

The R^2SVM is a deep non-linear classifier which builds upon a layer-by-layer architecture [82]. Random projection is used as the core stacking element, where the layers of the model are built by linear SVMs. In every layer, the weak predictions of the linear SVM are randomly projected back into the original feature space, whereby these projected predictions are then added to the original data. This can be regarded as moving data from different classes apart, by recursively transforming the original data manifold [82]. As can be seen in Figure 3.13, this leads to better linear separability in higher layers. A linear separating hyperplane in a higher layer corresponds to non-linear separating hyperplanes in the lower layers. In this way, non-linear classification can be achieved without using kernels, as stated in [82].

3.3.1 Structure and Recursive Transformation

We begin this section by stating some definitions: We consider a training set that contains *N* tuples $(d^{(i)}, y^{(i)})$, where $d^{(i)} \in \mathbb{R}^D$ is the *i*-th input sample and $y^{(i)} \in \{1, ..., C\}$ is the corre-

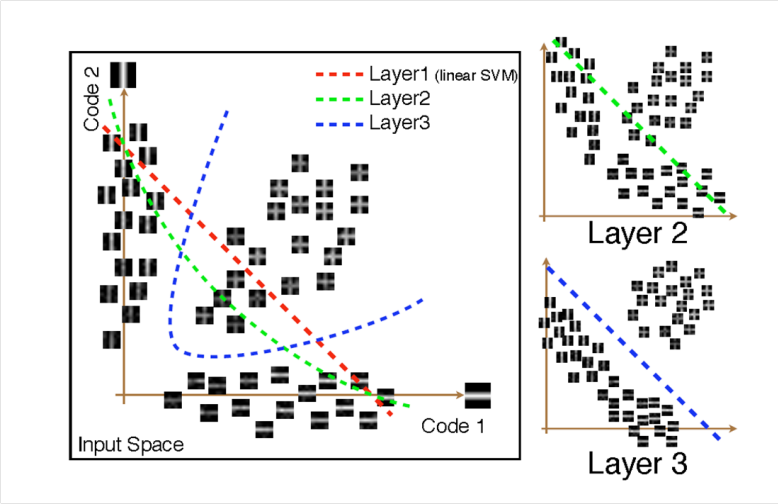


Figure 3.13: A conceptual example of Random Recursive SVM. The method transforms the data manifolds in a stacked way, to find a linear separating hyperplane in a higher layer. This linear hyperplane corresponds to non-linear hyperplanes in the lower layers. [82, p.2]

sponding class label. Furthermore, we define $\Theta \in \mathbb{R}^{D \times C}$ as the classification matrix, where $o^{(i)} = \Theta^T d^{(i)}$ is the vector of scores for each class of the sample $d^{(i)}$ and $\hat{y}^{(i)} = argmax(o^{(i)})$ is the prediction of the *i*-th sample, if we want to make final predictions. From now on, the index $.(i)$ is dropped for better readability.

As can be seen in Figure 3.14, each layer takes the vectors of scores from all lower layers and the output of the previous layer (starting with $o_0 = \emptyset$ and the original input data $x_1 = d$ in the first layer) [82]. For reasons of comprehensibility, the process in the first layer is explained. The original input data d is fed to the linear SVM in the first layer, which gives a new vector of scores, o_1. In general, o_1 would be better than a random guess, but far from a perfect prediction [82]. Next, a random projection matrix $W_{2,1} \in \mathbb{R}^{D \times C}$, which is sampled from $N(0, 1)$, is used to project the output o_1 into the original feature space, in order to modify the original features. Mathematically, the modified feature space can be described as follows:

$$x_2 = \sigma(d + \beta W_{2,1} o_1) \tag{3.22}$$

where x_2 denotes the modified feature space, d the original data, $W_{2,1}$ is the random matrix of the first layer and o_1 the output of the linear SVM [82]. β is a parameter that controls how much to shift the original feature space (if β is large, the original data samples are moved heavily), and σ is the sigmoid function, which introduces non-linearity just like in neural networks. The sigmoid function prevents the layered structure from degenerating into a trivial linear model, controls the scale of the resulting features and prevents over-confidence of the random projection on some data points (as the predictions of the lower-layers are imperfect) [82].

Next, the modified feature space x_2 and the vector of scores o_1 are fed to the second layer, where x_2 is used to train the linear SVM. Then the above described process, respectively the procedure explained in Section 3.3.2 starts again.

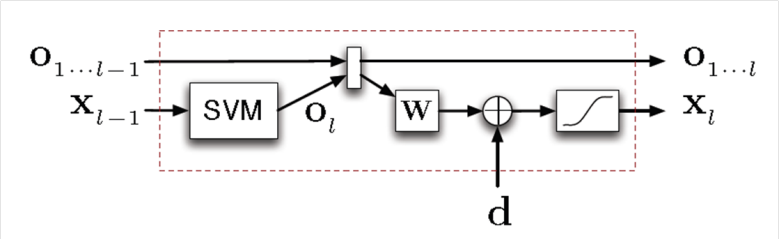


Figure 3.14: Details of an R^{2}SVM layer. [82, p.4].

The aim of the random projection (introduced in Equation 3.22) is to move data samples from different classes in different directions, so that the resulting features are, with a certain probability, better linearly separable [82]. It is important to note that, if the feature space D is relatively large, the column vectors of W_l are more likely to be approximately orthogonal [45]. Aside from that, the column vectors of W_l correspond to the per-class bias applied to the original data sample d, if the output o_l was close to the ideal e_c (where e_c is the one-hot encoding[8], representing class c). These characteristics lead to the property, that the data samples of different classes are pushed away from each other with high probability [82].

To illustrate this theoretical explanation, we consider an example, where there are four data samples, each with three features ($d \in \mathbb{R}^{3\times4}$) and a target variable with two classes. Additionally, there is a random projection matrix $W_l = (a_1 a_2)$ and the output $o_l = (b_1 b_2 b_3 b_4)$, where $a_i \in \mathbb{R}^3$ and $b_i \in \mathbb{R}^2$. It is also assumed, that the columns of W_l are orthogonal and the b_i-vectors are perfect predictions:

$$b_1 = b_2 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, b_3 = b_4 = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{3.23}$$

The predictions are projected back into the original feature space:

$$W_l * o_l = (a_1 a_2) * (b_1 b_2 b_3 b_4) = (a_1 a_2) * \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix} = (a_1 a_1 a_2 a_2) \tag{3.24}$$

If we use this projection to move the original samples, the data of different classes obviously would be shifted in different directions.

3.3.2 Training of R^{2}SVM

The training of the R^{2}SVM is done in a purely feed-forward way [82]. In every layer, a conventional linear SVM is trained and then used to compute the input of the next layer, as the addition of the original data samples and the random-projection of previous layers' outputs. This is then passed through a sigmoid function, as mentioned above. It is important to note that every layer

[8] One-hot encoding is a specific way to represent states or numbers, where legal combinations are only those with a single high bit and all the others low (e.g. 1 is '001', 2 is '010', 3 is '100')

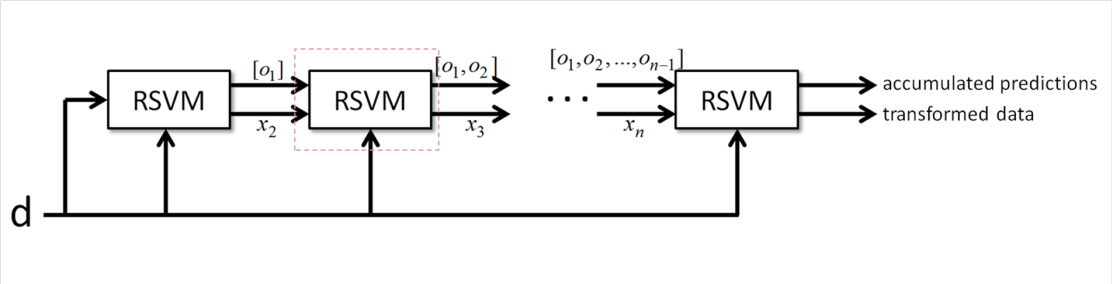


Figure 3.15: The layered structure of R^{2}SVM.

uses the original data and the outputs from all lower modules, to compute the input of the next layer, instead of using only the output of the immediately lower module (see Figure 3.15):

$$o_l = \theta_l^T x_l \tag{3.25}$$

$$x_{l+1} = \sigma(d + \beta W_{l+1}[o_1^T, o_2^T, ..., o_l^T]^T) \tag{3.26}$$

where θ_l is the classification matrix of the linear SVM trained with x_l and W_{l+1} is the concatenation of l random projection matrices $[W_{l+1,1}, W_{l+1,2}, ..., W_{l+1,l}]$ (one for each lower layer), where every random matrix is sampled from $N(0, 1)$ [82].

3.3.3 Motivation of Random Projections

First we consider a two class problem, which is non-linearly separable. The Lemma in [82, p.5] states, that if we have a training set (where we know the class labels of all samples) and a linear SVM solution on this training set, it is possible to add an offset to each class, in order to move apart the data samples and to guarantee a better solution of the linear SVM trained on this translated training set. In other words, the linear SVM trained on the translated training set would achieve a better prediction accuracy than the other linear SVM (trained on the 'original' training set).

This Lemma motivates the transformation of the original features, to achieve (better) linear separability under the guidance of SVM predictions. But since there are only noisy predictions during testing time, a deterministic choice of the offset would suffer from over-confidence in the labels and would possibly lead to overfitting [82]. This leads to the choice of random weights to avoid degenerated results and to improve the performance. Random weights enable the opportunity to use the information of the predictions and simultaneously de-correlate this information with the original data input, as stated in [82]. Vinyals et al. [82] claim

> "...randomness achieves a significant performance gain in contrast to the "optimal" direction given by Lemma 3.1 (which degenerates due to imperfect predictions), or alternative stacking strategies such as concatenation..." [82, p.5].

Here, the random projection matrices are sampled from a zero-mean Gaussian distribution. It would be possible to use a biased sampling depending on the 'optimal' direction, but the degree of bias would be difficult to determine and may be data-dependent [82].

3.3.4 R^{2}SVM: Hyper-Parameters

The R^{2}SVMs depend upon the following parameters:

- β: This parameter controls the degree, with which the original data samples are shifted [82].

- C is the regularization parameter of the linear SVM. Since C is an important parameter for the linear SVM [13] [82], it is vital to affirm that C is an important parameter of the R^{2}SVM approach, too. While over-fitting may be the result if C is too high, under-fitting may occur if C is chosen too small.

- The number of layers, n, is an important parameter as well, since the results in [82] suggest a strong influence on the accuracy.

Lastly, the random projection matrices W should be mentioned. For each layer a random matrix is created, whose elements are sampled from $N(0, 1)$ [82]. Though the projection parameters are randomly sampled from a normal distribution, the choice of the distribution can be seen as the selection of a hyper-parameter.

3.4 K-means Deep Learning Algorithm for Unsupervised Feature Learning

In this section, an unsupervised feature learning method is introduced which aims to learn high-level feature representations with unlabeled data only. These learned features can then be used for machine learning tasks such as classification. The method introduced in [22], respectively in [23], is based upon two different stages: an algorithm to learn selective features and an algorithm to combine these features into invariant features. While selective features can be seen as linear filters which respond to a particular input pattern (e.g. an object at a specific location and orientation), the invariant features respond to a broader range of patterns[9] [22]. In [22], these features are called *simple cells* and *complex cells*, where the simple cells correspond to the selective features and the complex cells to the invariant features. The algorithms for learning simple and complex cells can be used to train alternating layers of selective and invariant features, which form a (deep) architecture of multiple layers.

[9]For better understanding of the difference between these two type of features, we consider an image with a horizontal line in the middle. A selective feature would only respond to a line exactly in the middle, but not if this line would be slightly shifted to the top. In contrast, an invariant feature would also respond to this shifted line, because of its invariance to small transformations.

3.4.1 Learning Selective Features (Simple Cells)

The first algorithm in the unsupervised learning system learns "simple cell"-features, whereby these features are represented by linear filters (which respond to a particular input pattern). Before running the learning algorithm on the dataset, it is recommended in [22] and [23] to apply the following preprocessing: After brightness- and contrast-normalization is carried out, whitening is performed to remove the remaining correlations between nearby pixels. This preprocessing steps are recommended to receive better simple cell-features, whereby the linear filters are learned by a *k*-means-like method which is described in the following [23].

The classic *k*-means clustering algorithm finds cluster centroids to group the dataset into d groups, where the centroids are chosen in such a way that the distance between the data examples and the nearest centroid is minimized [23]. This procedure can be seen as a way of constructing a dictionary $D \in \mathbb{R}^{n \times d}$, where n is the feature dimension and d the number of vectors (alias centroids), so that each data vector $x^{(i)} \in \mathbb{R}^n, i = 1, ..., m$ can be mapped to a code vector that minimizes the error in reconstruction, where m is the number of examples [23]. In this first stage of the unsupervised learning system, a modified version of *k*-means is used to learn a dictionary D according to:

$$\underset{D,C}{minimize} \qquad \|DC^{(i)} - x^{(i)}\|_2^2 \tag{3.27}$$

$$subject\ to \qquad \|D^{(j)}\|_2 = 1, \forall j \tag{3.28}$$

$$and \qquad \|C^{(i)}\|_0 \leq 1, \forall i \tag{3.29}$$

where $C^{(i)}$ can be seen as a cluster assignment for the i-th data vector $x^{(i)}$ (with $C \in \mathbb{R}^{d \times m}$), $D^{(j)}$ is the j-th centroid of the dictionary D and $DC^{(i)}$ is the reconstruction of the vector $x^{(i)}$ [23]. The constraint $\|C^{(i)}\|_0 \leq 1$ means that the vector $C^{(i)}$ is only allowed to contain a single non-zero value, where this non-zero entry is otherwise unconstrained [22]. Furthermore $\|D^{(j)}\|_2 = 1$ constrains each centroid in D to have unit length and therefore avoids the possibility to rescale $D^{(j)}$ and the corresponding $C^{(i)}$ without effect [23].

There are common problems of the *k*-means algorithm like empty clusters, as stated in [23]. To avoid empty clusters in this *k*-means-like algorithm that is used to learn the simple cell filters, the centroids are initialized from a normal distribution and then normalized to unit length. Since the input data is whitened and therefore rescaled to a spherical-like distribution, Coates et al. [23] denote this initialization procedure as a good starting point. After the initialization step, the centroids are iteratively updated as illustrated in Figure 3.16. In every iteration step, each data example contributes to the update of exactly one centroid (the nearest one, where the scalar product is greatest), as depicted in Figure 3.16. In this example the computations, which are necessary to perform one update step of all centroids, are shown on the right-hand side, whereby $X \in \mathbb{R}^{d \times m}$ is defined as a matrix containing the data examples column-wise. On the left-hand side of Figure 3.16 one single update step is illustrated geometrically if we consider only one example and one centroid. Coates et al. [23] state, that typically ten iterations are enough to reach convergence.

Briefly speaking, after preprocessing the input data, a dictionary $D \in \mathbb{R}^{n \times d}$ is learned by the above described *k*-means-like algorithm, whereby the centroids represent the linear filters learned at this stage. The responses $s^{(i)}$ of the simple cell features for the i-th data example are

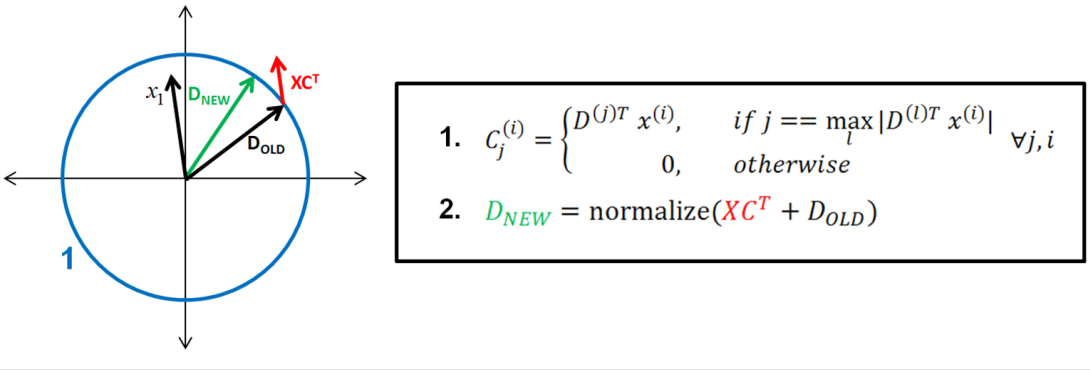


Figure 3.16: Illustration of one iteration-step of the k-means algorithm. On the right-hand side the required computation steps can be seen. For better understanding, the update step is drawn geometrically for an example with one centroid and one attached example on the left-hand side. In practice the update of one centroid can be influenced by more than one data example.

defined as $s^{(i)} = g(a^{(i)}) = g(D^T x^{(i)})$, where $g(\cdot)$ is a non-linear activation function [22]. The authors in [22] used $g(a) = |a|$ for the first layer and $g(a) = a$ for the second layer of simple cells in their experiments.

3.4.2 Learning Invariant Features (Complex Cells)

The second stage in the unsupervised learning system learns "complex cell"-features, where *max-pooling* is used to construct these invariant features as a combination of the responses of lower-level simple cells [22]. Here, *pooling* is defined as an aggregation operation, where multiple simple cell features are used to compute a single complex cell feature, as illustrated in Figure 3.17. Considering a vector of simple cell responses $s^{(i)}$, a complex cell feature is then given as:

$$c_j^{(i)} = \max_{k \in G_j} s_k^{(i)} \tag{3.30}$$

where G_j is a set of simple cells defined for the j-th complex cell c_j [22]. c_j pools over this defined set of simple cells and is therefore an invariant feature which is invariant to the changes exhibited by the patterns defined by the simple cells in its group G_j.

As stated in [22], the simple cells within a group G_j should, in some sense, be similar to one another. While there are different techniques to define the groups[10], in this system the groups are constructed, after the simple cell features have been trained, by using the *linear correlation* of simple cell responses $\mathbb{E}(a_k a_l)$ as a similarity metric [22]. In the end each group should contain simple cells that are similar on the basis of this metric. Because the inputs are whitened, as mentioned above, it is not necessary to estimate the correlations from data to get

[10] One way to define the groups is to hard-code them ahead of time, like in convolutional neural networks. Another possibility to define groups is to fix the groups ahead of time, but adapt the training algorithm in such a way that the received simple cell filters D share a certain kind of similarity within each group [22].

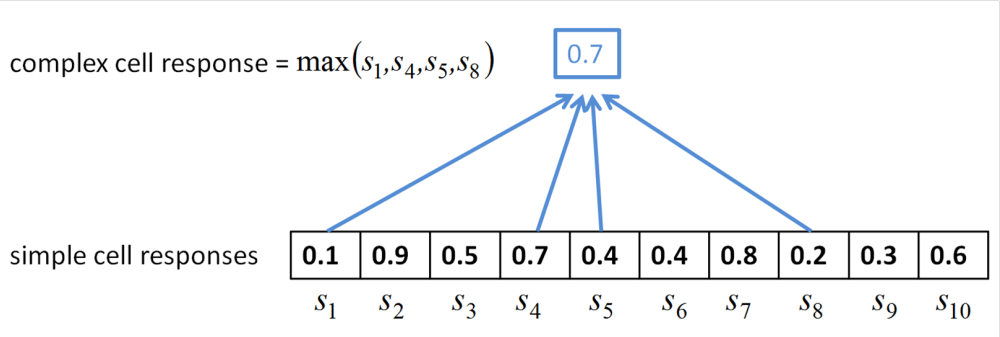


Figure 3.17: Example for a complex-cell feature illustrating the concept of *max-pooling*.

the similarities [22]. By contrast it is possible to compute the similarity directly from the filter weights:

$$\mathbb{E}(a_k a_l) = \mathbb{E}(D^{(k)T} x^{(i)} x^{(i)T} D^{(l)}) = D^{(k)T} D^{(l)} \tag{3.31}$$

In [22] Coates et al. do not use the similarity, but the dissimilarity between features, which is defined as:

$$d(k,l) = \|D^{(k)} - D^{(l)}\|_2 = \sqrt{2 - 2\mathbb{E}(a_k a_l)} \tag{3.32}$$

to find the groups. To construct a single group G_1, a randomly selected simple cell filter $D^{(k)}$ is chosen as the first member of this new group [22]. This first simple cell added is also called *seed point*. In the next step, for each simple cell $D^{(l)}$, which has not been added to G_1 until now, the dissimilarity $d(k,l)$ is computed [22]. If $d(k,l)$ is less than some limit τ, $D^{(l)}$ is added to the group G_1. In the next step additional cells are added to G_1 if any of the cells already in the group are closer than τ to the currently examined cell. This process continues until the two most dissimilar cells in the group show a dissimilarity larger than Δ, or no more cells are within τ of any member of the group [22]. This procedure, which is used to construct a single group, is shown in Algorithm 3.2.

To avoid ending up with many groups containing only a small number of simple cells, which would therefore not be especially invariant [22], it is recommended in [22] to use a specific strategy: While many groups are generated (e.g. several thousand), only a random subset of the largest groups is ultimately kept. Since the computational cost is extremely small in comparison with the simple cell learning procedure, it is no major drawback in terms of computational cost to use this strategy (generate more groups than will be kept) [22]. In addition, this statement is supported by the fact that the procedure to construct a group can be executed in parallel for a large number of randomly chosen seed cells, where this parallelization speeds up the group building process even more [22].

3.4.3 Stacking Layers of Simple and Complex Cells

So far the two stages of the unsupervised learning algorithm have been defined. The trained simple and complex cell features can be stacked together to form a deep architecture of alternating layers of selective and invariant features [22]. In this section, it is described how the learning

Input: the dictionary D
Output: a new group G

```
G = {};
Select a random simple cell filter D^(k) as seed;
Add D^(k) to the new group G;
while (the dissimilarity bewteen the two cells furthest apart in the group) < Δ do
    for all simple cells not in G do
        compute min[d(k,l)], the minimum dissimilarity of current simple cell D^(l) and
        all members of G;
    end
    for all simple cells not in G do
        if min[d(k,l)] < τ then
            add D^(l) to G;
        end
    end
end
return G;
```

Algorithm 3.2: Pseudo-code illustrating the construction of a single group for the complex cell features

procedures can be used in a concrete example to train an architecture with four layers in total, two of each type, with the architecture which is used in this example shown in Figure 3.18.

We consider a training-dataset with n 32-by-32 images, where n is the number of examples. Initially, 8-by-8 pixel patches are extracted from the 32-by-32 images to obtain a dataset[11] to train the first-layer simple cell features which have a feature dimension of 64 (8-by-8 filters). Here the number of first-layer filters which are trained is chosen to be 5000, so the resulting dictionary D has a dimension of $\mathbb{R}^{64\times5000}$. Then the complex cell learning procedure is applied to this dictionary to obtain 100 pooling groups $G_1, G_2, ...G_{100}$, where 100 is the number of complex cells chosen for this layer. These trained simple cell and complex cell features are then applied to the 32-by-32 images in a way which is illustrated in Figure 3.18. Each image is divided into 16 non-overlapping 8-by8 patches and the simple cell features (alias the linear filters D) are used to extract the responses $s_i^{(p)} = g(D^{(i)T}x^{(p)})$ of each patch, where $x^{(p)}, p = 1, ...16$ are the 16 sub-patches for one image. Next the complex cell feature responses $c_j^{(p)} = \max_{k\in G_j} s_k^{(p)}$ are computed for every patch. In other words, after the 5000 simple cell responses are computed for each patch, 100 complex cell responses are computed out of these simple cell responses for every patch. Since there are 16 patches in our example, the new representation of a single image is an array of 16×100 complex cell responses c.

These complex cell responses c are then used to train another layer of simple cells with the *k*-means like algorithm, whereby the complex cell responses can be seen as a new training

[11] Considering a dataset with 1000 32-by-32 images, the resulting dataset has 16000 8-by-8 pixel patches, if 16 non-overlapping patches are cut out of each image.

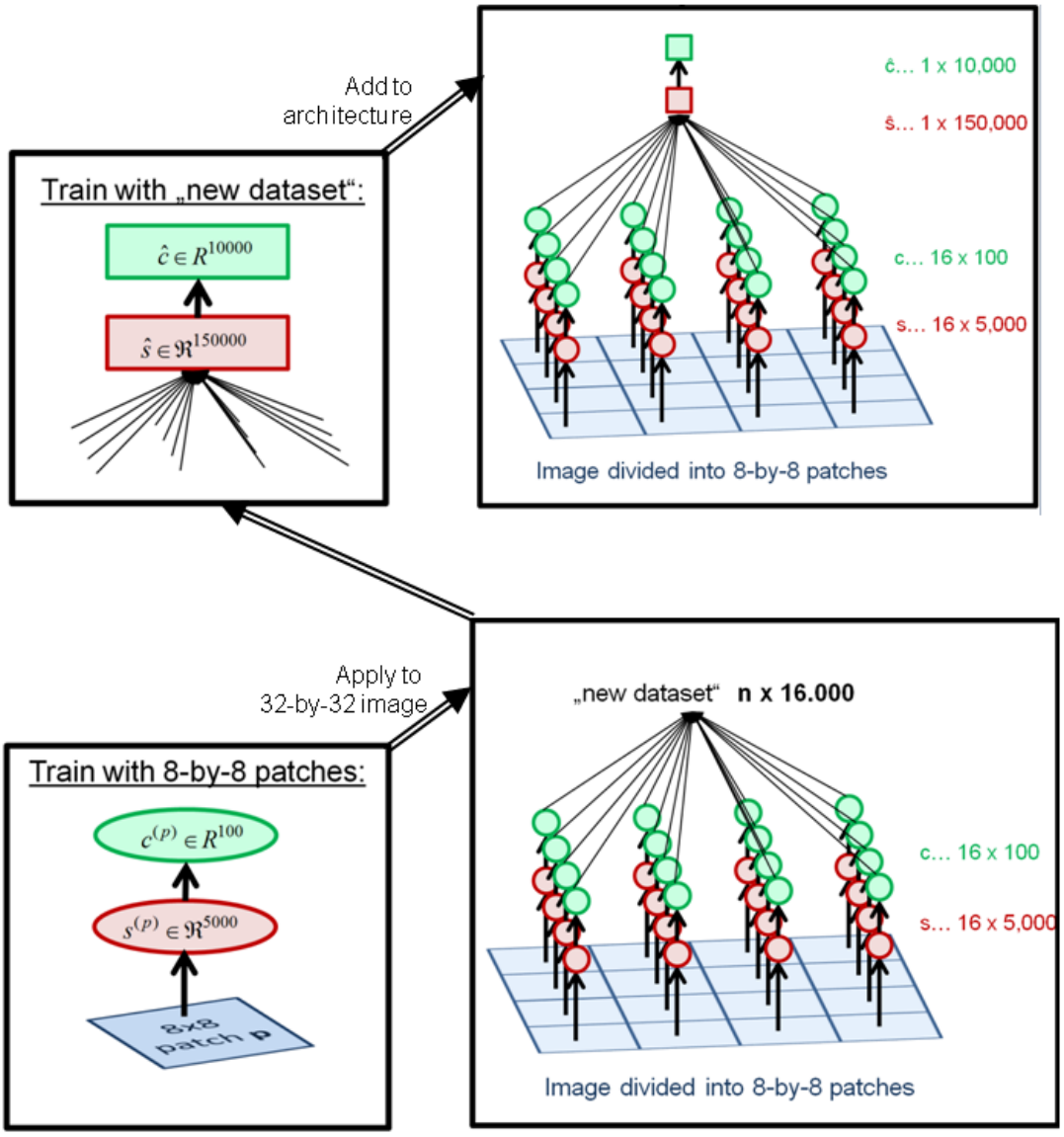


Figure 3.18: Illustration of the architecture of the unsupervised feature learning method. In the bottom left-hand corner the training of the first two layers is shown. In the bottom right-hand corner the resulting architecture and the ”new dataset” received by feeding the images to the system after the mentioned training step is depicted. In the upper left-hand corner the training of the third and forth layer is shown. The final architecture can be viewed in the upper right-hand corner. The *simple cells* are highlighted in red, whereas the *complex cells* are drawn in green. A more detailed explanation can be found in the text.

dataset with the dimensions $\mathbb{R}^{n \times 16000}$. In this second layer of simple cells (respectively the third layer in total) 150000 filters are trained with that new dataset. The resulting dictionary is denoted as $\hat{D}$ and the corresponding simple cell responses as $\hat{s} = \hat{D}^T c$. Again, the complex cell learning procedure is applied to the dictionary $\hat{D}$ to find pooling groups $\hat{G}$ and the corresponding complex cells $\hat{c}$, as shown in Figure 3.18. In this example, the number of groups for the second-layer complex cell features (respectively the forth layer in total) is defined as 10000.

Now that the complete architecture has been trained, the original n 32-by-32 images can be fed to the system which outputs 10000 features. Since these features should be meaningful high-level feature representations of the images, they can be used as input for an arbitrary classification method (e.g. SVM). While we use only four layers in total in this example, the above described learning procedures could be applied to build much deeper networks too. It is examined in Chapter 5 and Chapter 6, if these features are useful to increase the performance of a classifier.

Though an alternative could be to train the first-layer simple cells directly from the 32-by-32 images instead of using 8-by-8 patches, this turns out not to be successful, as shown in [22]. The features obtained by training the simple cells directly from the 32-by-32 images seem to exhibit a lower quality than the features obtained by training with smaller patches, as stated in [22] and [23]. As the size of the image increases, the *k*-means like algorithm attempts to learn "too complex features", which means that the learned centroids become empty or singletons. In other words, the *k*-means like algorithm fails to learn meaningful features as the dimensionality increases [23].

3.4.4 K-means Deep Learning Algorithm: Hyper-Parameters

In this section the hyper-parameters of the unsupervised feature learning method are described. Beside the hyper-parameters of the architecture, parameters of the learning procedure itself can be identified.

- The *number of layers* is a hyper-parameter which determines the depth of the architecture. Coates et al. [22] used four layers in total, two of each type, in their experiments.
- The *patch-size* of the small training sub-patches has to be chosen, where the size of the "original"-images restricts the number of possible choices. In [22], a *patch-size* of 8-by-8 is used.
- The *number of simple cells* in every layer should be set as large as possible in terms of computational resources, since the performance of the system grows with the number of features learned and training too few features causes under-fitting [22] [23].
- Another hyper-parameter is the *number of complex cells* in every layer. Unfortunately Coates et al. [22] give no clear recommendation on how to choose this value and how it impacts the performance of the approach.
- In a way, the choice of the non-linear activation function which is applied to the responses of the simple cell features, can be seen as a hyper-parameter of the architecture. As mentioned in Section 3.4.1, Coates et al. [22] use $g(a) = |a|$ for the first layer of simple cells,

which is a reasonable choice, since reversed polarity does not matter in their experiments. Apart from this information, Coates et al. [22] provide no clear recommendation as to which activation function should be used.

- τ controls the number of simple cell filters added to one group. As mentioned above, in every iteration simple cells are added to the group if the dissimilarity is not greater than τ. The higher this value is chosen, the higher is the possibility for adding more simple cells to each group. Coates et al. [22] used $\tau = 0.3$, respectively $\tau = 1.0$, in their experiments.
- Δ is a value which determines when to stop the construction process of a group. The lower this parameter is chosen, the higher is the possibility for stopping the construction procedure earlier. Coates et al. [22] used a value of $\Delta = 1.5$ in their experiments.
- The whitening parameter ϵ, which is added to avoid numerical instabilities, should also be chosen carefully. As stated in [23], a too small value can cause high-frequency noise to be amplified.

3.5 Summary

In the beginning of this chapter, a general introduction to deep learning is given. Beside the focus on this general part, this chapter also provides an explanation of three deep learning approaches. Neural networks are made up of single *neurons*, which are connected with each other. Auto-Encoders are stacked together to form a deep neural network, where an unsupervised pre-training stage (applied before supervised fine-tuning) is used to improve the performance of the deep network. The R^2SVM is a deep non-linear classifier which builds upon a layer-by-layer architecture, where the layers of the model are built by linear SVMs. The training of an R^2SVM is done purely supervised. The K-means deep learning algorithm on the other hand is a purely unsupervised feature learning method which aims to learn high-level feature representations with unlabeled data only.

CHAPTER 4

Implementation

In this section, the implementation of the aforementioned methods is described. Basically, *MATLAB* [58] is used as the primary language to implement the approaches which were discussed in the previous sections. Beside *MATLAB*, the software tools and frameworks *LIBSVM* [18], *LIBLINEAR* [34], *Torch7* [24] and *Pylearn2* [37] are used to obtain the desired functionality. In addition, other choices, for instance which kernel to choose for a non-linear support vector machine, are motivated and explained, too.

MATLAB is used as the primary language, which means that all methods are called from MATLAB, even if the method itself is written in another coding language. This way, the examination of hyper-parameters can be done centralized and the results can be visualized in a straightforward manner.

Furthermore, the *Map Reduce Framework* [30] is used to support the evaluation process, whereby this framework allows to organize the 'evaluation-pipeline' in a graph. Once built, each hyper-parameter setting can be sent as *request* to this graph and the framework automatically handles issues like computation scheduling, reusing of computations and data persistence.

Another issue is to verify the correctness of the implementation, where it has to be said that extensive verification of all tools and frameworks used would go beyond the scope of this work. In contrast, the *exploratory testing technique*[1] is used to detect as many faults as possible [5]. Furthermore, the neural networks are implemented with two different frameworks (*Torch7*, *Pylearn2*), which means that they can be compared with each other in order to detect errors. As these frameworks are not similar to use (they are even based on different programming languages), it is not likely that the results which are received are similar, if only one of the two implementations contains one or more errors or if both contain different errors. While this strategy is applicable for the neural network implementations, it cannot be used for the other approaches, since they are implemented only once. Nevertheless, the results of those methods (e.g. accuracy) can be compared with known results in literature in order to verify their correctness.

[1] In *exploratory testing*, the tester explores the software in whatever way he or she wants without restrictions [5]. In this work, the information about the toolboxes and frameworks is used to test specific functionalities which are most important or ambiguously documented.

4.1 K-Nearest Neighbor

Since MATLAB is used as the primary language, we decided to use the *'knnclassify'* function provided in the 'Bioinformatics Toolbox' of MATLAB [58]. This function provides the opportunity to determine the number of neighbors K, the distance metric and the rule of how to classify the data point if no majority vote exists. In the experiments of this work, the K-Nearest Neighbor approach is evaluated using the *euclidean distance* as the distance metric. If no majority vote is available, the nearest neighbor among the tied groups is used to break the tie. Thus, the only parameter which can be varied in the experiments is the number of neighbors.

4.2 Support Vector Machines

In this work the *LIBSVM* software package is used as the implementation of support vector machines. LIBSVM is a library which is based on the programming language C/C++ and is described in detail in [18]. It can be used for SVM-classification, SVM-regression and distribution estimation. It provides a MATLAB-interface, so it can be easily integrated into our code.

4.2.1 The Kernel Choice

As mentioned in Section 2.3, the 'kernel-trick' can be used to solve non-linear classification problems with SVMs. Since different kernels lead to different transformations (from the original input space into the high-dimensional feature space), the choice of the kernel is not a simple task. The LIBSVM software package provides the following kernels: *'linear kernel'*, *'polynomial kernel'*, *'Radial Basis Function (RBF) kernel'* and the *'sigmoid kernel'*. Because the computational power of our servers is limited, we decided to use only one kernel instead of testing all these kernels in their different configurations (respectively with different hyper-parameter settings). In this work, the RBF kernel is used to solve non-linear problems.

The first reason for using the RBF kernel, is that it has only one additional hyper-parameter which has to be selected. This means that the complexity of the model selection with the RBF kernel is simpler than with the polynomial kernel (because the polynomial kernel has more hyper-parameters) [41].

The second reason is that, in comparison to the polynomial kernel, the RBF kernel exhibits fewer numerical difficulties [41]. Furthermore, the sigmoid kernel is not valid under some parameters[2] [79].

The third reason for using the RBF kernel, is that Vinyals et al. [82] compared the R^{2}SVM approach with SVMs using the RBF kernel. Therefore the RBF kernel is a reasonable choice to be able to compare the experimental results of this work with the results given in [82].

4.2.2 Training SVMs with a one-vs-one Strategy

Referring to [60] and [2], there is no clear benefit of the "one-versus-one" or the "one-versus-all" strategy and the choice depends on what the problem is. More precisely, the accuracy and

[2] There are cases in which the so-called *mercer's condition* is not fulfilled, which means that the 'kernel-trick' is not applicable for those parameter combinations [25] [53] [79].

the complexity of the training process depend on the number of classes and training samples. With respect to accuracy, [60] and [2] obtain different results: While Milgram et al. [60] come to the conclusion, that the 'one against all' strategy is more accurate than the 'one against one' approach, Allwein et al. [2] state the opposite. Furthermore, by default *LIBSVM* supports only one-versus-one classification which would lead to a more complex code for a one-versus-all strategy implementation. All these facts lead to the decision to train the non-linear SVM-models in a one-vs-one fashion. For the evaluation, this means that non-linear support vector machines, which use the one-versus-one strategy and a Radial Basis Function (RBF) kernel, are examined.

4.3 R^{2}SVM

Since the *LIBSVM* software described above is not efficient in training linear support vector machines, we decided to use the software *LIBLINEAR* [34] instead. LIBLINEAR is an open source software package from the same authors, which can be used from the command line and supports linear regression and linear support vector machines. As in LIBSVM, a MATLAB-extension is included in the LIBLINEAR package, whereby LIBLINEAR can be used in the MATLAB environment. A detailed explanation of LIBLINEAR can be found in [34].

As stated in [41], the LIBLINEAR package is faster than LIBSVM for training linear SVMs. Considering an example, where five-fold cross-validation is used for finding the best parameter-value on a dataset with 20,242 instances and a feature dimension of 47,236, LIBLINEAR takes only around 3 seconds for training the linear SVM, the training with LIBSVM takes around 350 seconds [41]. Furthermore, in [41] it is claimed that the solutions of both packages in respect of accuracy are nearly the same. In the above mentioned example, the cross-validation accuracy is 96.8% for the LIBSVM solution and 97.0% for the LIBLINEAR solution.

Since it is stated in [82], that the linear SVM classifiers are trained in a one-vs-all fashion, we decided to train the linear classifiers in a one-vs-all mode too. As mentioned above, there is no clear benefit of one strategy and the choice depends on the problem at hand. Furthermore, the *LIBLINEAR* software package implements the one-vs-all classification strategy [34], which is why the decision to train the linear classifiers in a one-vs-all fashion avoids a more complex implementation and reduces the number of potential errors in the code (because no own code has to be written and tested).

The rest of the functionality is implemented in MATLAB, where the MATLAB-interface is used to integrate the functionality of LIBLINEAR into the MATLAB code. In MATLAB, the basic architecture of random recursive support vector machines is implemented. The random matrices are created, the linear SVMs are stacked together and the predictions of the linear SVMs are used to modify the original feature space (random projection).

4.4 Extremly Randomized Trees

For this approach, the implementation used in [31], is also used in our work. In this implementation, the default values for classification, $K = \sqrt{N}$ and $n_{min} = 2$, are used, where N is the number of attributes. This is a reasonable choice, since the results obtained with the default settings of the extra-trees algorithm show no significant difference to the variant, where the best

parameters are learned by cross-validation [36]. In short, the number of trees M is the only parameter which is varied.

4.5 Neural Networks

The implementation of the neural network is done in the programming language *Lua*, respectively in *Python*. These languages are used, since the machine learning libraries *Torch7* [24] and *Pylearn2* [37] (which is based on *theano*) can be used within these programming languages. To compare these implementations with each other, the same functionality is implemented, although Torch7 is not as suitable as Pylearn2 for some parts and vice versa. The reason for implementing neural networks with two different languages and libraries is to be able to verify the correctness of the implementations, as stated previously. While Pylearn2 can be easily run on GPUs, the GPU usage is much more complicated with Torch7. Due to this and the fact that the runtime of the deep neural network approaches is much higher if they are not computed on a GPU, we decided to use only the Pylearn2 implementation for the evaluation in Chapter 5, Chapter 6 and Chapter 7.

4.5.1 Functionality

All functions are called from MATLAB, which means that the data and hyper-parameters of the neural network are initialized in MATLAB and are then passed to the Python code (Pylearn2), respectively to the implementation in Lua (Torch7) via *.mat-files*. The results of the neural network computations are then again stored in .mat-files and loaded into MATLAB. In the following, the characteristics and the possible settings of the neural network implementation are described:

- First of all, the *basic architecture* of the neural network can be determined by a parameter. For instance, $[400\ 1000\ 600\ 5]$ would create a network with an input dimension of 400 (e.g. 20-by-20 images as input), two hidden layers (where the first layer has 1,000 units and the second one 600) and five output units.

- It is also possible to chose the type of non-linear activation function which is used in the neurons of the neural network: Either the *hyperbolic tangent* or the *logistic sigmoid* function (which are described in Section 3.2.1).

- Since it is also necessary to determine the activation-function for the last layer, it is possible to chose either the identity function (alias the linear activation function) or the softmax activation function. As discussed in Section 3.2.4, the softmax activation function can be used in the case of multi-class classification problems and the linear activation function can be used for regression problems.

- The neural network is trained with *mini-batch gradient descent*, because this kind of training exhibits a better flexibility than stochastic or batch gradient descent, which are just special cases of mini-batch gradient descent (as explained in Section 3.2.3). Therefore it is possible to set the mini-batch size of the training procedure.

- The *learning rate* of the gradient descent algorithm, which is introduced in Section 3.2.3, can be determined, whereby it remains constant permanently.

- Further, the *momentum*, which is described in Section 3.2.10, can be set by a parameter.

- Two different criteria determine the time when to stop the training procedure. At first, the maximum number of training iterations is determined by a parameter which defines the maximum number of 'epochs'[3] over the given training data set. Secondly, to avoid unnecessary update steps and to save computation time, the following can be defined via parameters: If the accuracy does not improve within N epochs with a factor of d, training is stopped. The accuracy is calculated on the training set, so this stopping criterion does not correspond with the concept of early-stopping (which has been introduced in Section 3.2).

- There are three *types of training procedures*, which can be used:

  - unsupervised greedy layer-wise pre-training of the hidden layers with a subsequent supervised fine tuning step of the whole network,
  - supervised fine-tuning only or
  - unsupervised layer-wise training only (without a fine-tuning stage).

  These training procedures can be defined by a hyper-parameter in the corresponding .mat-file too.

- Since there are more hyper-parameters of the neural network which have to be determined (in comparison with the other approaches) and our computational power is limited, we decided to use specific *error functions* in our experiments (without the need to chose them explicitly), in order to reduce the complexity of the model selection. The type of error function which is used depends on the defined architecture and the type of training. For the greedy layer-wise training, the sum-of-squares error is used as the "basic" error function. Additionally L2-regularization of the weights and L1-regularization of the unit-activation is used to implement regularization, respectively the concept of sparse auto-encoders. The error function used for fine-tuning depends on the activation function of the last layer. For the linear activation function, the sum-of-squares error function is used to train the network. In contrast, if the softmax function is used in the last layer, the *multi-class cross-entropy error function*, introduced in Section 3.2, is used for fine-tuning. Briefly speaking, it is only possible to set the parameters of the error function (weight decay parameter, sparsity parameter), but not the type of the error function itself[4].

- To control the relative importance of the regularization term and the sparsity constraint in the cost function, it is possible to set the *weight decay regularization coefficient* and

[3] Here, one epoch is defined as one iteration over the whole training set. For instance, if the mini-batch size is determined as 50 and the training data contains 10,000 examples, 200 update steps would be done in one epoch.

[4] Though the code is written in a way that this functionality, of setting the error function explicitly, could be implemented easily.

an *activation regularization coefficient*. Again, these hyper-parameters are initialized in MATLAB, stored in a *.mat*-file and then loaded in our Lua-, respectively Python-implementation. More details about these hyper-parameters can be found in Section 3.2.10.

- As discussed in Section 3.2.10, the *weights initialization scaling coefficient* r is used to initialize the weights of the neural network. Since it is recommended in [9] to use formulas to chose this coefficient and Torch7 provides no simple way of changing the initialization procedure of the weights, we use the standard formula defined in Torch7: $r = \frac{1}{\sqrt{fan-in}}$.

The only difference between the Lua- and the Python-implementation, is an extension of the latter one. Since Pylearn2 provides a function which allows an easy usage of the denoising auto-encoder variant instead of sparse auto-encoders (and Torch7 does not), the denoising-autoencoder variant is only implemented once. Therefore it is possible to choose between sparse or denoising auto-encoders in the pre-training stage of the neural network only in the Pylearn2 implementation. *Masking noise* is used as the corruption process of the denoising auto-encoder, whereby the amount of destruction can be determined via a parameter. A detailed explanation of this corruption process can be found in [80] and [81].

4.5.2 Libraries

Pylearn2 is a machine learning library based on the programming language *Python*. It is built upon theano, a compiler for Python which optimizes mathematical expressions to increase the computation speed. A detailed explanation of theano can be found in [12]. Pylearn2 provides components which can be used in the code to achieve the desired functionality. Basically there are three types of components: 'Dataset', 'Model' and the 'TrainingAlgorithm' [37]. The 'Dataset' simply stores the data which is used for training, whereas the 'Model' component contains parameters and the "basic architecture" of the model (e.g. weights, biases, number of layers, etc.). The 'TrainingAlgorithm' uses the training data provided by the 'Dataset' to adapt the 'Model'. Furthermore, modularity is used in Pylearn2 to allow simple changes concerning functionality, without the need to make big changes [37].

Torch7 is a machine learning library and extends *Lua*. Lua itself is a scripting language, whereby it is implemented as a library and written in C [24]. Torch7 basically consist of eight packages, which provide various functions like optimization algorithms or neural network modules. A detailed description of all used packages would go beyond the scope of this work, but can be found in [24].

4.6 K-means Deep Learning Algorithm

Since the framework's code is written in MATLAB, we decided to implement this unsupervised feature learning method in MATLAB as well. The visualization functions of MATLAB are used to verify the correctness of the implementation. At first the original patches and the preprocessed versions of the patches are visualized to check the preprocessing procedure. Secondly, the learned simple cell filters are plotted in order to compare them with the results presented

in [22]. Additionally, the groups which are generated in the complex cell training stage are visualized to be able to compare them with the grouping in [22] (in order to verify the correctness of the training procedure) and to check the parameter-settings (τ, Δ, etc.). Furthermore, the functionality is divided into several independent functions, so the implementation is clear and built modular (e.g. the algorithm to create a single group). In the following all hyper-parameters of the implementation are described:

- The *patch-size* of the small training sub-patches can be determined, whereby the size of the 'original' images (which are fed to the network in the end to get the features) has to be considered[5].
- The *number of simple-cells* which are trained, has to be defined for every layer.
- Though Coates et al. [23] claim that typically ten iterations of centroid-updates are enough, it is possible to set the number of iterations, used for the K-means clustering algorithm presented in Section 3.4, individually.
- The *number of complex-cells* which are trained can be determined for every layer.
- τ, a hyper-parameter of the complex-cell training procedure, has to be defined for every layer.
- Δ, the second hyper-parameter of the complex-cell training process, can also be determined for every layer.
- Since Coates et al. [22] recommend to generate more groups than are needed in the end and then keep only a random subset of the largest groups, the *initial number of groups* which are trained at all and the *size of the largest-groups-subset* can be defined for each layer (beside the number of complex-cells).
- The whitening parameter *epsilon* is chosen to be $1e^{-5}$ and cannot be changed, since the visualizations of the preprocessed patches suggest that this value is a reasonable choice for our datasets.
- The nonlinear activation function for simple cells, discussed in Section 3.4.4, is chosen to be $g(a) = a$ for all simple cell layers, since it is not clear whether reversed polarity matters.

4.7 Summary

This section provides a summary of the main issues described in this section. MATLAB is used as the primary language to implement the approaches, which means that all methods are called from MATLAB, even if the method itself is written in another coding language. For

[5] If there are 32-by-32 images, the patch-size could be 8-by-8. In contrast, if there are 28-by-28 images, it would be appropriate to define the patch-size as 7-by-7 in regard to the final system architecture.

the K-nearest neighbor approach, the *'knnclassify'* function of MATLAB is used for the experiments. The LIBSVM software package (based on C/C++) is used as the implementation of SVMs. In contrast, the LIBLINEAR package is used as implementation for the R^2SVMs. For the extremely randomized trees, the implementation used in [31], is also used in the experiments of this work. The neural networks are implemented twice, using the machine learning libraries Pylearn2 and Torch7. Finally, the K-means deep learning algorithm is implemented in MATLAB.

CHAPTER 5

Comparison of Learning Approaches on the MNIST Dataset

This chapter contains the first part of the main contribution of this thesis: the comparison of learning approaches. In this chapter, the methods which are discussed in Chapter 2 (K-Nearest Neighbors, SVM, Extremely Randomized Trees) and Chapter 3 (R^2SVM, Stacked Auto-Encoders, K-means Deep Learning Algorithm) are evaluated on the *MNIST dataset* [50], where the implementations described in Chapter 4 are used for this evaluation. An overview of the evaluated methods and the evaluation criteria are shown in Table 5.1.

After the description of the dataset in Section 5.2, the choice of the used hyper-parameters is discussed in Section 5.3. Subsequently the accuracy and the runtime of the examined methods are evaluated in Section 5.4 and Section 5.5. The influence of the hyper-parameters on the performance is discussed in Section 5.6. Special Characteristics of selected models are described in Section 5.7. In Section 5.8 the effect of changing the number of used labeled examples is examined. Finally, a summary of the evaluation on the MNIST dataset is given in Section 5.9.

5.1 Evaluation Procedure

The evaluation process, which is used in this work, is shown in Figure 5.1. First of all the dataset is divided into three parts: The training, the validation and the test set. The *training set* is used to train the methods with different hyper-parameter settings. In this work, *grid-search* (this technique is discussed in Section 2.1.3 and in [9] respectively) is used to find the best hyper-parameter setting, since the analysis of the results is easier in comparison to other techniques [9]. The *validation set* is used to compare the hyper-parameter settings in terms of predictive performance of every method, where the model with the best hyper-parameter setting (i.e. the parameter setting with the highest predictive performance on the validation set) is selected for every method. Finally, the generalization performances of the selected models (one for every method) are evaluated using the *test set*. The examples of the labeled training set are

Methods	Criteria
K-nearest neighbor	The approaches shown on the left-hand side are compared in terms of the following criteria: • Accuracy • Runtime • Influence of Hyper-Parameters • Special Characteristics of Selected Methods • Influence of Supervised Training Set Size
Extremely Randomized Trees	
SVM	
R^2SVM	
Stacked Sparse Auto-Encoder	
Stacked Denoising Auto-Encoder	
K-means Deep Learning Algorithm	

Table 5.1: This table gives an overview about the methods, which are evaluated in this work, in the left-hand column. The right-hand column contains the criteria which are used to compare these methods.

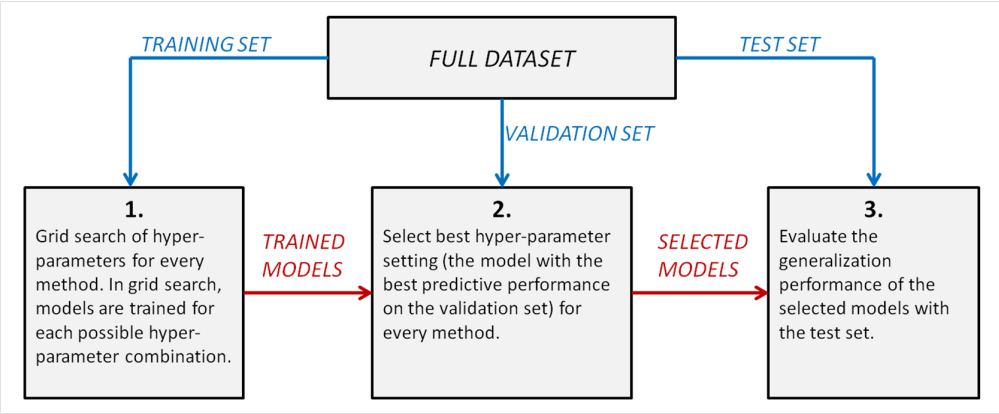


Figure 5.1: Illustration of the evaluation process used in this work. While the "model-flow" is denoted as red, the "data-flow" is highlighted in blue.

also used as unsupervised dataset (by ignoring the labels), since three of the evaluated methods[1] require unsupervised images. This process of model selection and calculating the generalization performance is also discussed in Section 2.1.3.

5.2 MNIST Dataset

The *Modified NIST (MNIST) dataset* is a database of 28-by-28 pixel handwritten digit images and is a subset of the NIST database [50]. The MNIST training set comprises 30000 images from the *NIST's Special Database 1 (SD-1)*, 30000 images from the *NIST's Special Database 3*

[1]To be exact, stacked sparse auto-encoders and stacked denoising auto-encoders use unsupervised data in the pre-training stage. On the other hand, the K-means deep learning algorithm is trained solely with the unsupervised dataset.

(a) (b)

Figure 5.2: Randomly selected examples from the *MNIST database of handwritten digits*. (a) Examples taken from the training set. (b) Test set samples.

(SD-3) and the test set has a size of 10000 images (5000 per database). As stated in [50], it was necessary to build the new dataset *MNIST* out of the NIST's databases, because the results of experiments must be independent of the split of the complete dataset into training and test set, in order to be able to draw reasonable conclusions. This argument is plausible, since originally SD-3 was designated as a training set, SD-1 as a test set, and SD-3 is composed of examples which are easier to recognize than the ones in SD-1[2] [50].

The MNIST training set contains images from about 250 writers, while the images of the MNIST test set originate from other writers. The original bi-level (black and white) images from the NIST database were normalized to fit into a 20-by-20 pixel image, where the aspect ratio was kept for all examples during this resizing process [50]. Since an anti-aliasing technique was used within this normalization procedure, the resulting images also contain grey levels. The size-normalized examples were centered in the final 28-by-28 image by computing the center of mass of the pixels (for the 20-by-20 image) and translating the image so the center of mass is located at the center of the final 28-by-28 image [50]. Randomly selected images from the *MNIST dataset* are shown in Figure 5.2.

In this chapter, the MNIST set is divided into a training set of 50000, validation set of 10000 and a test set of 10000 examples.

5.3 Parameters of the Algorithms

In the experiments with the MNIST database no preprocessing like normalization of the brightness, contrast normalization or whitening is performed. Instead, a simple rescaling to $[0, 1]$ of all examples is done, as recommended in [41]. Since the pixels lie in the range of $[0, 255]$, this is achieved by dividing the data by 255.

[2]The reason for this is that these two databases of the handwritten digits were collected among different writers [50].

As stated in [9], the values for the hyper-parameters have to be chosen carefully and can have an important effect on the results. The selection of the interval (in which the hyper-parameters are examined) can therefore be seen as a hyper-hyper-parameter. Within this interval, it is recommended in [9, 41] to choose regularly spaced values in the exponential domain (e.g. $2^{-5}, 2^{-3}, ..., 2^{1}, 2^{3}$), since the ratio between different values is a better indicator of the impact of the hyper-parameters than the absolute value. More information of this issue can be found in Appendix A.1. In the following it is described which hyper-parameters are examined for every method within the evaluation process.

5.3.1 Approach 1: K-Nearest Neighbors

Since the Euclidean distance is fixed as the distance metric, the number of neighbors K is the only hyper-parameter which is varied. On the one hand, theoretically the optimal value is rather in the order of n than in the order $\log n$ (where n is the number of features) as discussed in [57]. On the other hand, LeCun et al. [50] state that the optimal value for the MNIST dataset is $k = 3$, though the feature dimension of the examples is $n \in \mathbb{R}^{784}$. To consider both research results in our experiments, we use the values

$$k = \{3, 9, 27\}. \tag{5.1}$$

5.3.2 Approach 2: Extremely Randomized Trees

As described in Section 4.4, the default parameters, which are recommended in [36] for classification problems, are used. Therefore, the number of trees M is the only parameter which is varied in this work. Since Geurts et al. [36] use $M = 100$ in their experiments and state at the same time that the convergence behavior may depend on the problem, the following values are examined:

$$M = \{8, 16, 32, 64, 128\}. \tag{5.2}$$

5.3.3 Approach 3: SVMs

As discussed in Section 4.2, the RBF kernel is used for the non-linear SVMs in our experiments. Therefore two hyper-parameters have to be determined for the SVM approach: C and γ. Taking into account the parameter choices in [35,41,71,72][3] and computational considerations, we have decided to examine the following hyper-parameter values:

$$C = \{10^{-4}, 10^{-2}, 10^{0}, 10^{2}, 10^{-4}\} = \{0.0001, 0.01, 1, 100, 10000\} \tag{5.3}$$

$$\gamma = \{10^{-4}, 10^{-2}, 10^{0}, 10^{2}, 10^{-4}\} = \{0.0001, 0.01, 1, 100, 10000\} \tag{5.4}$$

5.3.4 Approach 4: R^{2}SVM

For the R^{2}SVM approach, the values for three hyper-parameters, which are discussed in Section 3.3.4, have to be determined: the number of layers n, the regularization parameter C and β

[3] For instance, in [35] the cost parameter C was examined in the range $[2^{-2}, 2^{14}] = [0.25, 16384]$ and γ in the range $[2^{-5}, 2^{7}] = [0.03125, 128]$.

(which controls the degree with which the original data samples are shifted). First of all, the number of layers has to be chosen. Vinyals et al. [82] state the following

> "For the R^{2}SVM, in most cases the performance asymptotically converges within 30 layers." [82, p.7].

Therefore it is reasonable to use 50 layers for our experiments. Further, the values which are examined for the regularization parameter C have to be determined. Since the results are sensitive to the selection of C, as explained in Section 3.3.4, and in order to simplify the comparison with the conventional non-linear SVM approach, we have decided to examine the same values as specified for the non-linear SVMs. Finally, three different values of β are taken into consideration in this work, though $\beta = \frac{1}{10}$ is used for all experiments in [82]. To sum up, the following values are chosen for the hyper-parameters of the R^{2}SVM approach:

$$n = \{50\} \tag{5.5}$$
$$C = \{10^{-4}, 10^{-2}, 10^{0}, 10^{2}, 10^{4}\} = \{0.0001, 0.01, 1, 100, 10000\} \tag{5.6}$$
$$\beta = \{0.1, 0.5, 1\}. \tag{5.7}$$

5.3.5 Approach 5: Stacked Sparse Auto-Encoders

As discussed in Section 3.2.10, for neural networks it is necessary to select more hyper-parameters (in our work eight or more) than for other approaches like K-Nearest Neighbors, SVMs, R^{2}SVMs or Extremely Randomized Trees. This leads to a high number of combinations, since grid-search is used to evaluate the hyper-parameter setting. For stacked sparse auto-encoders, 324 different combinations are examined with the MNIST dataset. A summary of the settings can be seen in Table 5.2. A detailed discussion about the choice of the hyper-parameters is provided in Appendix A.2.

5.3.6 Approach 6: Stacked Denoising Auto-Encoders

For the stacked denoising auto-encoders, 216 combinations are computed for the MNIST dataset in this work. The settings which are used in the experiments are nearly the same as for the stacked sparse auto-encoder approach, except that denoising auto-encoders do not have an *activation regularization coefficient*. Instead, the amount of destruction ν has to be chosen, where in the experiments of this work, *masking noise* is used as the corruption process (this destruction technique is explained in Section 3.2.10). For the MNIST database two values of the amount of destruction are examined:

$$\nu = \{0.25, 0.5\} \tag{5.8}$$

5.3.7 Approach 7: K-means Deep Learning Algorithm

The hyper-parameters of this method, which is described in Section 3.4, are summarized in Table 5.3. In this work the hyper-parameters are determined based on the settings and values used in [22, 23]. Therefore the basic structure of the architecture described in Section 3.4.3 is used in the experiments of this work. A summary of the settings for the K-means deep learning

Characteristic	**Setting**
Activation function for neurons	Hyperbolic tangent function
Activation function of last layer	Softmax activation function
Cost function for pre-training	Sum-of-squares error function + L2-regularization of the weights + L1-regularization of the unit-activation
Cost function for fine-tuning	Multi-class cross-entropy error function
Type of training	Unsupervised pre-training and supervised fine-tuning
Stopping criterion	Stop if output of the cost function does not improve upon a factor of 0.1% within 10 epochs. Maximum number of epochs: 1000.
Mini-batch size	100
Initialization of weights	Initialize weights with the uniform distribution $U(-r, r)$, where $r = \frac{1}{\sqrt{fan-in}}$
Number of hidden-units	$\{1000, 2000\}$
Number of hidden-layers	$\{2, 3, 4\}$
Momentum	$\{0, 0.5\}$
Weight decay regularization coefficient	$\{0.003, 0.1, 3\}$
Activation regularization coefficient	$\{0.003, 0.1, 3\}$
Learning rate	$\{10^{-5}, 10^{-4}, 10^{-3}\} = \{0.00001, 0.0001, 0.001\}$

Table 5.2: Summary of settings for the stacked sparse auto-encoder approach.

algorithm is given in Table 5.3. An in-depth discussion of the hyper-parameter selection can be found in Appendix A.3.

The K-means deep learning algorithm is trained purely unsupervised and provides no prediction of the class of images. Instead, this method aims to transform the input images into a higher level of representation, where the outputs (of the finally trained architecture) are ideally useful features which can be used as input for a conventional classifier. One objective of this work is to evaluate whether the performance is improved, if the classifier works with these features as input instead of the raw input images as input. Therefore, the pipeline shown in Figure 5.3 is used to build the final classifier. First, the K-means deep learning algorithm is trained purely unsupervised. Subsequently, the labeled training set is fed to the trained architecture, where the outputs are used to train extremely randomized trees with $M = 64$ (extremely randomized trees are discussed in Section 2.4 and Section 5.3.2). Finally predictions can be done with this trained pipeline.

Characteristic	Setting
Patch-size	7-by-7
Number of layers	4 in total (2 of each type)
Activation function	$g(a) = a$
Number of simple cells	*1. Layer:* $\{12800\}$ *2. Layer:* $\{6400, 12800\}$
Number of complex cells	$\{128, 256\}$ for both layers.
Initial number of groups	$10 \times number\_of\_complex\_cells$
Size of the largest-groups-subset	$2 \times number\_of\_complex\_cells$
τ	*1. Layer:* $\{0.5, 0.7, 0.9\}$ *2. Layer:* $\{1.0, 1.2\}$
Δ	1.5 for both layers.

Table 5.3: Summary of settings for the *K-means deep learning algorithm*.

5.4 Comparison of Accuracy

In this section, the accuracies of the methods, obtained on the MNIST dataset, are presented and compared. An overview is given in Table 5.4, where the test-accuracies (the accuracies computed on the test set) of all evaluated methods are shown. The baseline of 10.00% denotes the revealed accuracy of a simple model which classifies all examples randomly (in the test set). Since the test set of the MNIST database exhibits 10 classes, the baseline is 10.00%. This baseline helps to assess the quality of the evaluated models.

Accuracy is a way to measure and compare the generalization performance of the methods. It is calculated as the number of correct predicted examples divided by the total number of examples:

$$Accuracy = \frac{\#correct}{\#correct + \#incorrect} \tag{5.9}$$

While the stacked denoising auto-encoder exhibits the best test accuracy with 98.55%, the sparse auto-encoder approach has the third highest test accuracy (98.07%). On the other hand, the SVM approach exhibits the second highest test accuracy of 98.29%. The K-means deep learning algorithm does not help to improve the performance of the extremely randomized tree approach, since the test accuracy decreases from 90.31% to 80.03%.

The confusion matrix of the stacked sparse auto-encoder approach is illustrated in Figure 5.4, where the general structure of this confusion matrix is representative of the confusion matrices of all methods. This confusion matrix indicates that there are no classes of the MNIST dateset which are especially difficult to detect or distinguish from one another.

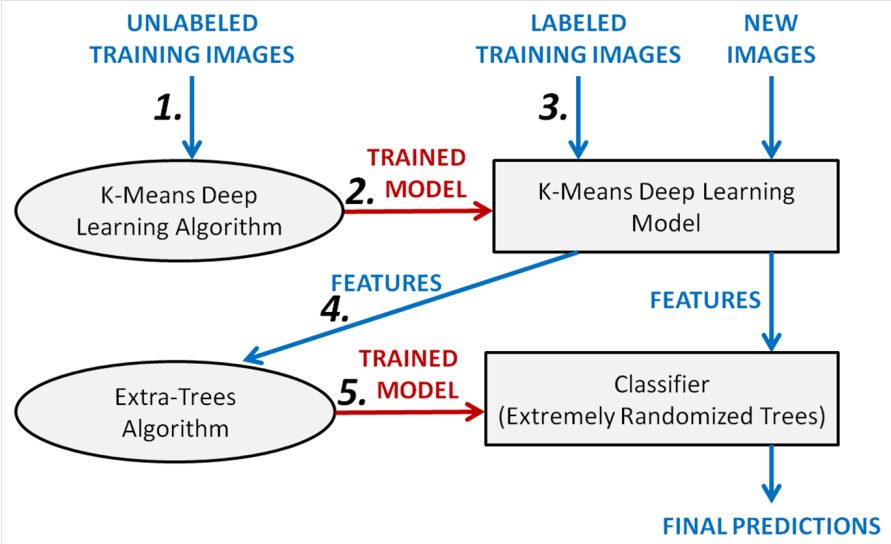


Figure 5.3: Illustration of the pipeline which uses the K-means deep learning algorithm and the extremely randomized tree approach. The ”model-flow” is highlighted in red, the ”data-flow” is denoted as blue and the particular steps of the training procedure are numbered. After the training of the pipeline has finished, predictions can be made for new images.

5.5 Comparison of Runtime

Since the computations are performed in parallel on six different machines, it is possible that the runtime of our computations is influenced by operations of other users. Additionally, the GPU is used for the computations of the stacked denoising auto-encoders and the stacked sparse auto-encoders, where the other approaches are executed on the CPU. An overview of the programming languages, in which the evaluated methods are implemented, can be found in Appendix A.4. Nevertheless a rough tendency of the training time can be identified. Beside a summary of the training time, Table 5.5 gives an overview of the time needed for prediction, too. The times shown in this table denote the average time (mean and median) needed to train (respectively predict) with one setting of a method. The approaches are arranged according to the mean training time, in ascending order. One exception is the K-nearest neighbor approach which has no explicit training phase and therefore no training time, as explained in Section 2.2.

The distribution of the training time is illustrated in Figure 5.5 for each method. This figure contains the so-called *box-plot* which draws one ”box” for every method. An explanation of box-plots is given in Appendix A.5.

As can be seen in Table 5.5, the R^2SVM approach and the extremely randomized trees are the fastest methods in respect to training time: both need about half an hour (mean value) to train a model with one setting. The K-means deep learning algorithm needs about five hours (mean value) to train one model. Though the stacked sparse auto-encoders are trained on the GPU, it takes nine hours (mean value) to train one model. The mean training time of the SVM approach, which is executed on the CPU, is about nine hours, too. The stacked denoising auto-

Method	Test Accuracy (MNIST)	# Varied Parameters	# Settings
Stacked Denoising Auto-Encoder	98.55%	6	216
SVM	98.29%	2	25
Stacked Sparse Auto-Encoder	98.07%	6	324
K-Nearest Neighbor	96.92%	1	3
R^2SVM	93.36%	2	15
Extremely Randomized Trees	90.31%	1	5
K-means Deep Learning Algorithm	80.03%	5	48
Baseline	10.00%	-	-

Table 5.4: This table gives an overview of the obtained test-accuracies. The selected models (the ones with the highest validation accuracy) were applied on the MNIST test set. Beside the accuracy, the table summarizes the number of varied parameters and the number of settings (i.e. the number of trained models) for each method.

encoder approach exhibits a mean training time of fifteen hours, which is nearly twice as long as the mean sparse auto-encoder runtime and 25 times longer than the mean value of the fastest method (R^2SVM).

The time to calculate the predictions for the whole test set (10000 examples) is also depicted in Table 5.5, where the fastest approaches are Extremely Randomized Trees, R^2SVM and the auto-encoder approaches, which compute the predictions within half a minute. It takes two minutes to calculate the predictions of the K-means deep learning algorithm, which means that predictions of about 83 examples are calculated within one second. Though the theoretical part of this work in Chapter 2 may give the impression that the prediction time of the K-nearest neighbor approach is higher than the prediction time of the SVM (because the SVM uses only the support vectors for prediction, while the K-nearest neighbor approach uses the whole dataset, as mentioned in Section 2.2), our experimental results show the opposite: the prediction runtime of the SVM is, with eleven minutes, the longest one among all examined approaches. In contrast, it takes seven minutes to calculate the predictions of the K-nearest neighbor approach.

The box-plot of Figure 5.5 coincides with the results presented in Table 5.5, but shows that the data values of the auto-encoder approaches, the SVM and the K-means deep learning algorithm are widely distributed in comparison to the other approaches. In Section 5.6 it is examined, if this is due to the influence of the hyper-parameters.

5.6 Influence of Hyper-Parameters

In this section, the characteristics of every method, like the influence of the hyper-parameters on the runtime and validation accuracy (the accuracy is computed on the validation set), are discussed in detail. Since the computations are performed in parallel on six machines which

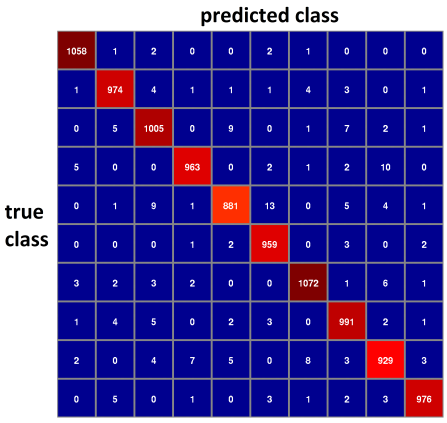


Figure 5.4: The confusion matrix of the sparse auto-encoder with the highest validation accuracy. The rows denote the known classes, while the columns depict the predicted classes. For instance, the number 13 in row 5 and column 6, is a count of observations known to be in class "4" but predicted to be in class "5". The first row, respectively column corresponds to the class *"1"*, while the tenth to the class *"0"*.

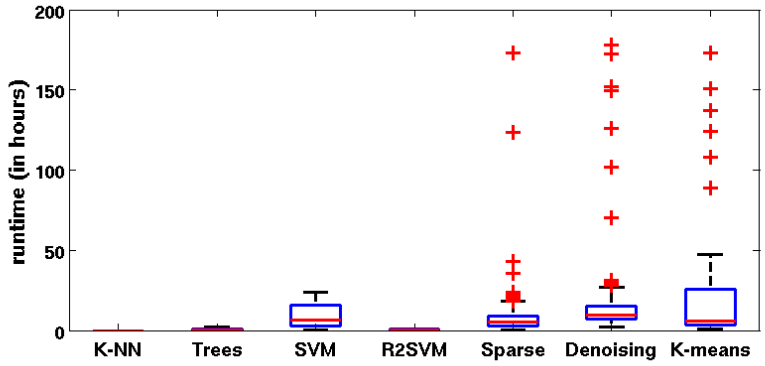


Figure 5.5: This box-plot illustrates the training-times of the methods, where each distribution is computed by considering all training-times of a method (for each setting one training-time).

are not reserved exclusively for our computations, calculations of other users can influence the runtime of our computations. Therefore the results presented in this section in relation to runtime have to be treated with this in mind.

Method	Runtime for Training in hours (mean / median)	Runtime for Prediction (mean)
K-Nearest Neighbor	none	7 minutes
R^2SVM	0.6 / 0.6	26 seconds
Extremely Randomized Trees	0.7 / 0.3	5 seconds
K-means Deep Learning Algorithm	5.2 / 5.6	2 minutes
Stacked Sparse Auto-Encoder	9.0 / 5.5	30 seconds
SVM	9.3 / 6.4	11.4 minutes
Stacked Denoising Auto-Encoder	15 / 9.6	30 seconds

Table 5.5: This table gives an overview of the average time needed to train one setting of a method, respectively the average time needed to calculate the predictions for the validation set. The runtime is listed for each approach and is the mean and median of the time over all models of a method. The only exception is the K-nearest neighbor approach which exhibits no training stage.

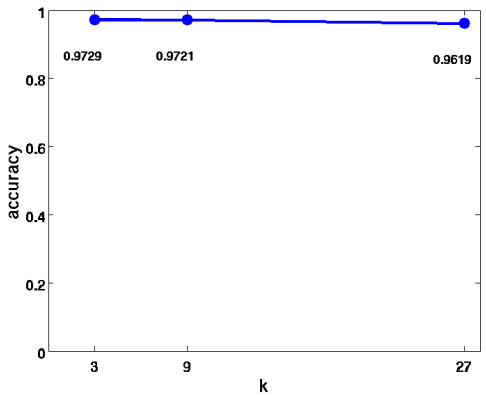
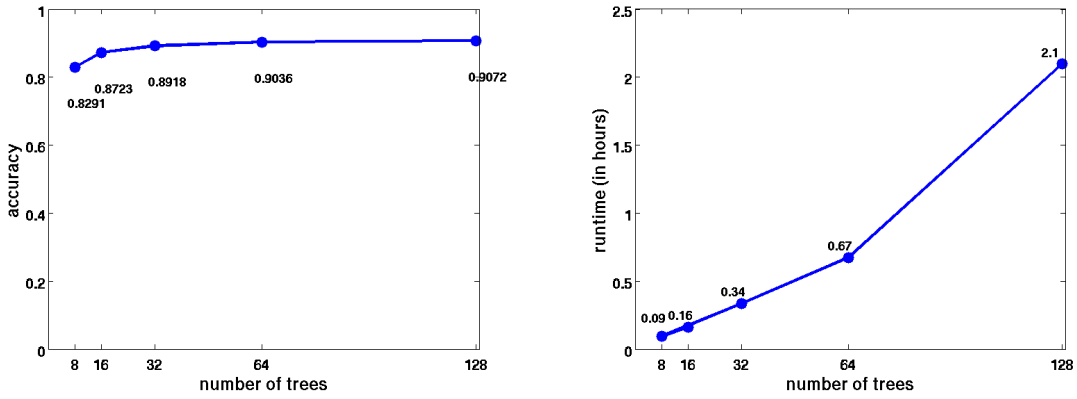


Figure 5.6: Approach 1 - Plot which illustrates the influence of the hyper-parameter k of the K-nearest neighbor approach on the validation accuracy. The accuracy is plotted against the hyper-parameter k.

5.6.1 Approach 1: K-Nearest Neighbors

The higher k is chosen the lower the accuracy gets.

As can be seen in Figure 5.6, the higher k is chosen the lower the accuracy gets, where the highest validation accuracy is achieved with $k = 3$. To be exact, the accuracy decreases from 97.29% ($k = 3$) to 96.19% ($k = 27$). In terms of prediction-runtime, no clear tendency can be recognized. The time to calculate the predictions is for all values of k about seven minutes



(a) (b)

Figure 5.7: Approach 2 - This figure contains box-plots concerning the extremely randomized tree approach. (a) In this graphic, the validation accuracy is plotted against the number of trees. (b) This plot illustrates the influence of the number of trees on the runtime.

(between 413 and 438 seconds). Since the K-nearest neighbor approach has no explicit training phase, there is no training-runtime which can be examined.

5.6.2 Approach 2: Extremely Randomized Trees

A higher number of trees leads to a higher accuracy, where the accuracy converges within 128 trees. The runtime increases linearly with the number of trees.

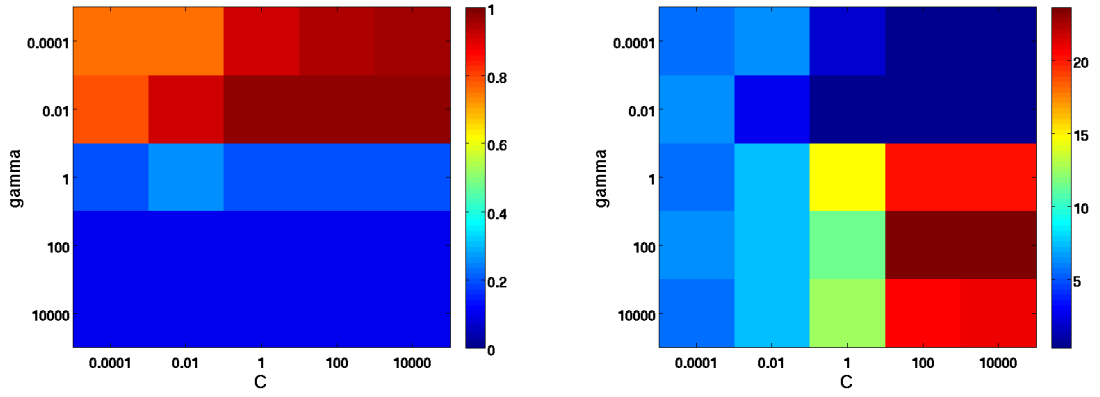
In Figure 5.7(a), the number of trees is plotted against the obtained validation accuracy. This plot coincides with the statement in Section 2.4.2, that a higher number of trees leads to a higher accuracy. More precisely, the accuracy converges within the examined values, which can be seen in Figure 5.7(a). This convergence behavior is consistent with results presented in [36, 63].

The time needed for training in the extremely randomized tree approach is plotted against the number of trees in Figure 5.7(b). In this graphic it can be seen, that the runtime increases linearly with the number of trees chosen.

5.6.3 Approach 3: SVM

A lower γ leads to an increased accuracy and a shorter training time. A higher value of C tends to increase the accuracy and the training time. In terms of accuracy, the choice of γ is more crucial than C.

The SVM exhibits two hyper-parameters, where the influence of C and γ in terms of accuracy is shown in Figure 5.8 (a). This figure illustrate the importance of the hyper-parameters, where the choice of γ is much more critical than the choice of C. A bad choice of γ can result in a very bad model: for the MNIST dataset, choices of γ of 1 or higher lead to an accuracy of



(a) (b)

Figure 5.8: Approach 3 - Two figures which illustrate the relation between accuracy and runtime. (a) The two hyper-parameters of the SVM are plotted against the accuracy. A dark red area denotes a high accuracy and a dark blue area highlights low accuracy. (b) The two hyper-parameters of the SVM are plotted against the runtime. Red areas denote a long runtime and dark blue areas highlight a short runtime.

about 20% or lower. In contrast, high values of C tend do give better results than lower values, though the influence of C is not as high as the influence of γ. This circumstance can be seen in Figure 5.8(a), too.

The influence of the hyper-parameters C and γ on the runtime is depicted in Figure 5.8(b). It can be seen that the higher C is, the longer is the time needed to train the SVM. This correlation can be explained by the fact that a higher C tends to increase the model complexity of the SVM, as mentioned in Section 2.3.6. Furthermore, the higher γ is chosen, the longer is the training time. Again, the reason for this is that the model complexity of the SVM is increased as γ is getting higher (as explained in Section 2.3.6).

Furthermore, Figure 5.8 shows that the training time is longer for a model with a lower accuracy, while the runtime is shorter for models with a high accuracy.

While a higher C tends to a model with fewer support vectors, a higher γ tends to decrease the number of support vectors. A detailed discussion about the influence of the hyper-parameters on the number of support vectors can be found in Appendix A.6.

5.6.4 Approach 4: R^{2}SVM

The parameter-selection is not as sensitive as the parameter-choice for SVMs. Both a higher C and a higher β lead to a higher accuracy.

Since the number of layers is chosen to be fixed (as explained in Section 5.3.4), there are only two hyper-parameters of the R^{2}SVM approach which can be examined. First of all, the results show that the parameter-selection of the R^{2}SVM approach is not as sensitive as the parameter-choice for SVMs. While the standard deviation of the validation accuracy of SVMs is 38.59%,

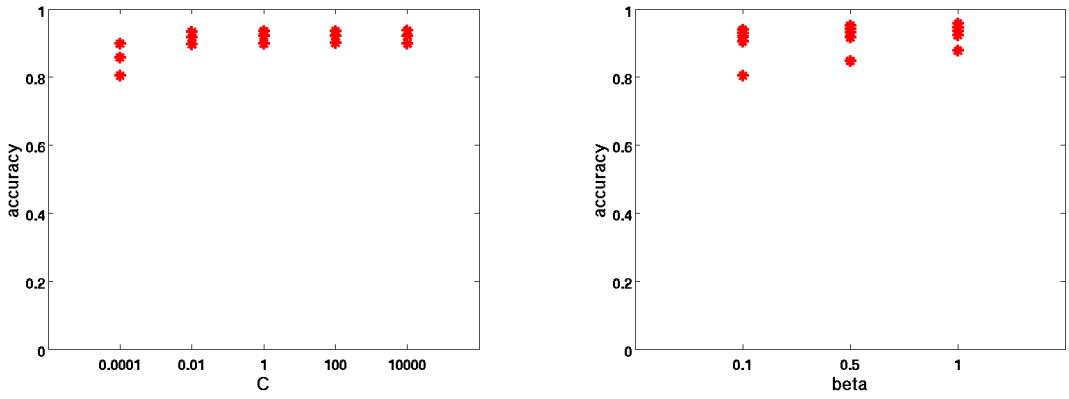


Figure 5.9: Approach 4 - An illustration of the influence of the hyper-parameters of R^{2}SVM on the accuracy. (a) In this graphic, the hyper-parameter C is plotted against the accuracy. (b) A plot showing the influence of β on the accuracy.

it is only 3.11% for R^{2}SVMs. In other words, for SVMs, the different hyper-parameter settings result in a validation accuracy in the range from 10.64% to 98.35%. In contrast, the range of R^{2}SVMs stretches from 82.40% to 93.75%.

Though the parameters do not have such a high influence as the parameters of the SVM approach, clear tendencies can be detected. The higher the hyper-parameter C is chosen, the higher is the validation accuracy of this model, where C is plotted against the accuracy in Figure 5.9 (a). Figure 5.9 (b) shows a plot of β and the validation accuracy, where it can be seen that a higher value of β leads to a higher accuracy. While Vinyals et al. [82] found the value of $\beta = 0.1$ to work well in their experiments, in our work the model with the highest accuracy exhibits a value of $\beta = 1$. This mismatch may appear due to the fact that in our work the images are directly fed as input to the R^{2}SVM approach, while Vinyals et al. [82] feed extracted features as input to the R^{2}SVMs. In terms of runtime, no clear tendency (respectively influence) can be identified, neither for C nor for β.

5.6.5 Approach 5: Stacked Sparse Auto-Encoders

In respect of accuracy, the *activation regularization coefficient* and the *learning rate* exhibit the strongest influence. Both a higher number of layers and a higher number of hidden units lead to a shorter training time.

The choice of hyper-parameters of the stacked sparse auto-encoder approach has a strong influence on the performance, which can be seen at the high standard deviation of the validation-accuracies of 40.43% (the range stretches from 10.00% to 98.08%). Furthermore, 21 out of 324 settings cause divergence of the optimization algorithm (of the stacked sparse auto-encoder), where the hyper-parameters which are the same for all these settings are the learning rate (0.001)

and the number of hidden-units (2000). At the same time, 58.03% of all models have an accuracy below 20%, which is due to the influence of the *activation regularization coefficient*, as illustrated in Figure 5.10(a).

The *activation regularization coefficient* exhibits the strongest influence among all hyper-parameters, which can be seen in Figure 5.10(a). The box-plot in this figure shows that the lowest value, which is examined (0.003), yields the best performance. This means that if the *activation regularization coefficient* is chosen too high, the term which penalizes the hidden unit activations (with a *L1-penalty*) dominates the error function and therefore the gradient descent optimization algorithm (i.e. the update steps, respectively the gradients, are dominated by this regularization term). Though the influence of the *number of hidden-units* is superimposed by the *activation regularization coefficient*, it can be seen in Figure 5.10(b), that the higher number of hidden-units tends to lead to better accuracy. This coincides with the findings in [9] that it is most important to chose the number of hidden units large enough.

In Figure 5.10(c), the *learning rate* is plotted against the accuracy. Though the *activation regularization coefficient* dominates over the *learning rate*, it can be seen that a higher learning rate leads to an improved accuracy. To stress the influence of the learning rate, a second box-plot is shown in Figure 5.10(d), where only models which are trained with an *activation regularization coefficient* of 0.003 are taken into account. Together with the statement above concerning settings which cause divergence, this result coincides with the recommendation in [9] to use the highest possible value of the *learning rate* which causes no divergence. An explanation for this effect is that a higher learning rate is more likely to skip local minima (vice versa, a smaller learning rate more likely gets stuck in local minima). Among the other hyper-parameters (*number of hidden-layers*, *momentum*, *weight decay regularization coefficient*) no clear relations or influences in terms of accuracy can be identified.

The influence of the *activation regularization coefficient* on the runtime is depicted in Figure 5.10(e). It can be seen that a higher value leads to a shorter training time of the stacked sparse auto-encoder. A reasonable explanation for this correlation is, that a higher value changes the optimization criterion in such a way that the optimization procedure converges more quickly (but in a local minimum), since a higher coefficient also leads to models with bad performance. Furthermore, a higher *learning rate* leads to a shorter training time, where this relation is shown in Figure 5.10(f). This correlation can be explained by a higher number of update steps which are needed to reach the final minimum if the *learning rate* is decreased, since the *learning rate* controls the step size of the optimization algorithm, as described in Section 3.2.3.

The box-plot in Figure 5.10(g) illustrates, that a higher number of hidden-layers increases the training time of the network. On the one hand, this can be explained by the fact, that a higher number of layers generates the necessity to pre-train more hidden-layers. On the other hand, an increased number of hidden-layers leads to an increased number of network-parameters which have to be tuned in the subsequent fine-tuning training stage. In Figure 5.10(h) it is shown, that a higher number of hidden-units yields a longer runtime. Again, this can be explained by the fact, that a higher number of units have to be trained both in the pre-training and in the fine-tuning stage, as mentioned in [9]. Among the other hyper-parameters (*momentum*, *weight decay regularization coefficient*) no clear influence on the runtime can be detected.

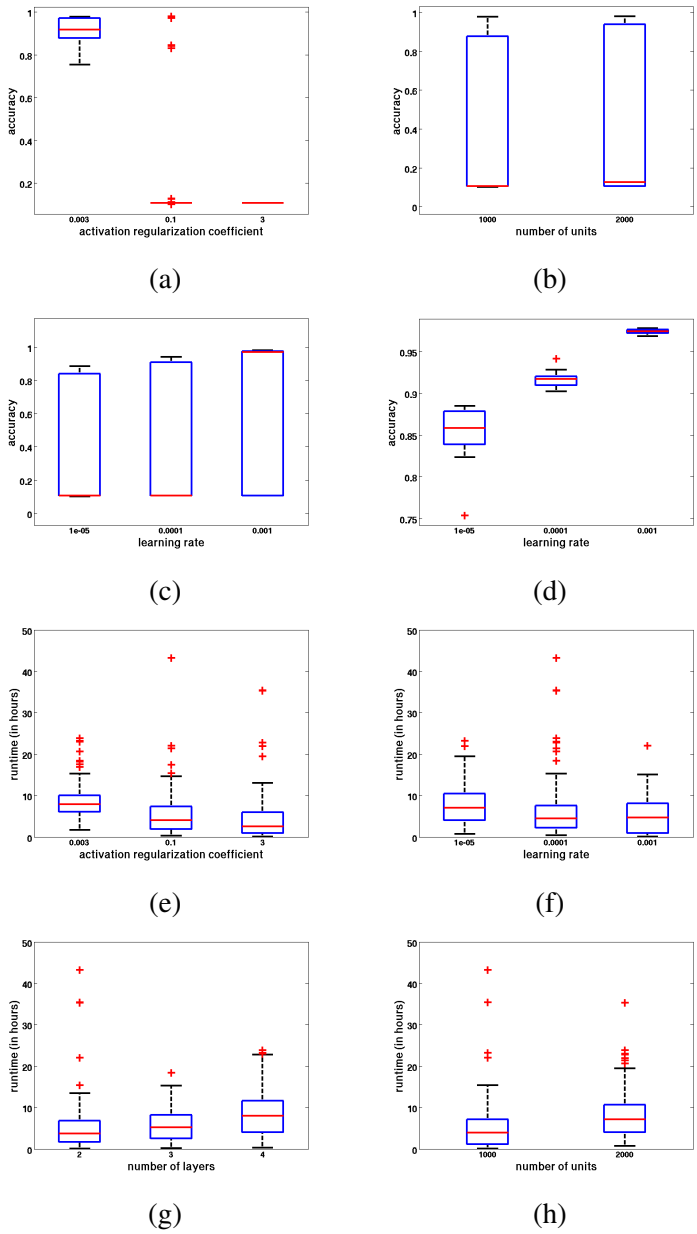


Figure 5.10: Approach 5 - In this figure box-plots show the influence of (a) the *activation regularization coefficient*, (b) the *number of units* and (c) the *learning rate* on the validation accuracy. (d) To improve the illustration of the influence of the *learning rate* on the accuracy, this box-plot only uses models which are trained with an *activation regularization coefficient* of 0.003. (e) The *activation regularization coefficient*, (f) the *learning rate*, (g) the *number of hidden-layers* and (h) the *number of hidden-units* is plotted against the runtime.

A clear influence of the number of layers and the number of hidden-units on the prediction runtime can be identified, where the runtime is within 4 minutes for all stacked sparse auto-encoders. Both a higher number of hidden-units and a higher number of hidden-layers tends to increase the prediction runtime. Among the other hyper-parameters (*learning rate*, *momentum*, *weight decay regularization coefficient*, *activation regularization coefficient*), no clear influence on the prediction runtime can be identified.

5.6.6 Approach 6: Stacked Denoising Auto-Encoders

The parameter-selection is not as sensitive as the parameter-choice for sparse auto-encoders. A higher *learning rate* leads to a higher accuracy. In respect of runtime, the number of hidden-units and the number of layers show the strongest influence.

For the stacked denoising auto-encoder approach, the influence of the parameters on the accuracy is not as strong as with the stacked sparse auto-encoder approach, which can be seen in the standard deviation of the validation-accuracies (5.38%). Additionally, no parameter setting causes divergence of the optimization process, which also suggests that the denoising auto-encoders are more stable than the sparse auto-encoder approach.

As stated in [9] and shown in Figure 5.11(a), a higher learning rate leads to a higher accuracy. In contrast to the sparse auto-encoder approach, it can be seen in Figure 5.11(b) that the usage of a momentum increases the performance of the network. Though an influence of the momentum can be identified, it is not as strong as the influence of the learning rate, as can be seen when comparing Figure 5.11(a) and Figure 5.11(b). Among the other hyper-parameters (*weight decay regularization coefficient*, *number of hidden-units*, *number of hidden-layers*, ν) no clear influence on the accuracy can be identified.

In Figure 5.11(c) and Figure 5.11(d) it can be seen, that both a higher number of layers and an increased number of hidden-units increases the training time of the stacked denoising auto-encoder approach. This result coincides with the results presented for the sparse auto-encoders, as discussed in Section 5.6.5. For the rest of the parameters (*momentum*, *weight decay regularization coefficient*, *learning rate*, ν) no clear influence on the runtime can be identified.

5.6.7 Approach 7: K-means Deep Learning Algorithm

The hyper-parameter τ of the first layer exhibits the highest influence on the accuracy and the runtime.

Only the parameters of the K-means deep learning algorithm are examined in this section. The K-means deep learning algorithm does not help to improve the performance of the randomized tree approach on the MNIST dataset. Possible reasons are inappropriate hyper-parameters or a low number of unlabeled training examples (50000).

In terms of accuracy, the hyper-parameter τ of the first layer has the highest influence on the accuracy. With $\tau = 0.5$, the validation accuracy is below 20% for all models, which is shown in Figure 5.12(a). In contrast, the highest accuracy is achieved with $\tau = 0.9$. In Figure 5.12(b), only models are considered which are not using $\tau = 0.5$ in their first layer. This helps to better

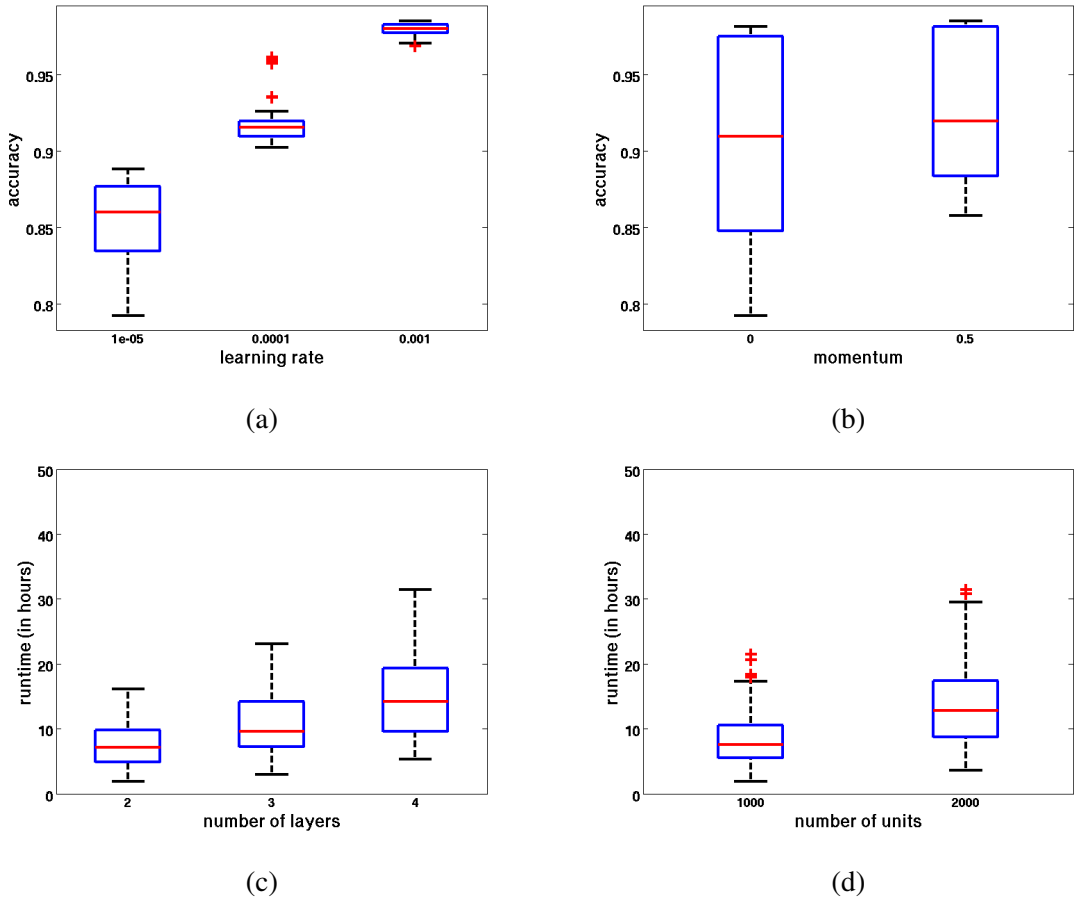


Figure 5.11: Approach 6 - In this figure, (a) the *learning rate* and (b) *momentum* are plotted against the accuracy. Furthermore, (c) the *number of hidden-layers* and (d) the *number of hidden-units* is plotted against the runtime.

illustrate the influence of τ of the second layer on the accuracy. In this box-plot it can be seen that a value of $\tau = 1.0$ in the second layer yields a better distribution at first glance. However, the validation- accuracies of $\tau = 1.2$ are more widely spread than the validation-accuracies of $\tau = 1.0$, which is why the highest validation- accuracy is obtained with $\tau = 1.2$.

Figure 5.12(c) shows the influence of the *number of simple cells in the second layer* on the accuracy, where a higher number of simple cells tends to give a higher accuracy. For the number of complex cells a similar behavior as for the influence of τ of the second layer can be identified. As shown in Figure 5.12(d), 256 complex cells in the second layer have a better distribution of the validation accuracy. Yet again, the validation-accuracies for 128 complex cells (in the second layer) are more widely spread and therefore lead to a higher maximum of the validation accuracy. Also in Figure 5.12(c) and Figure 5.12(d) only models are considered which are not using $\tau = 0.5$ in their first layer to improve the illustrations. For the *number of*

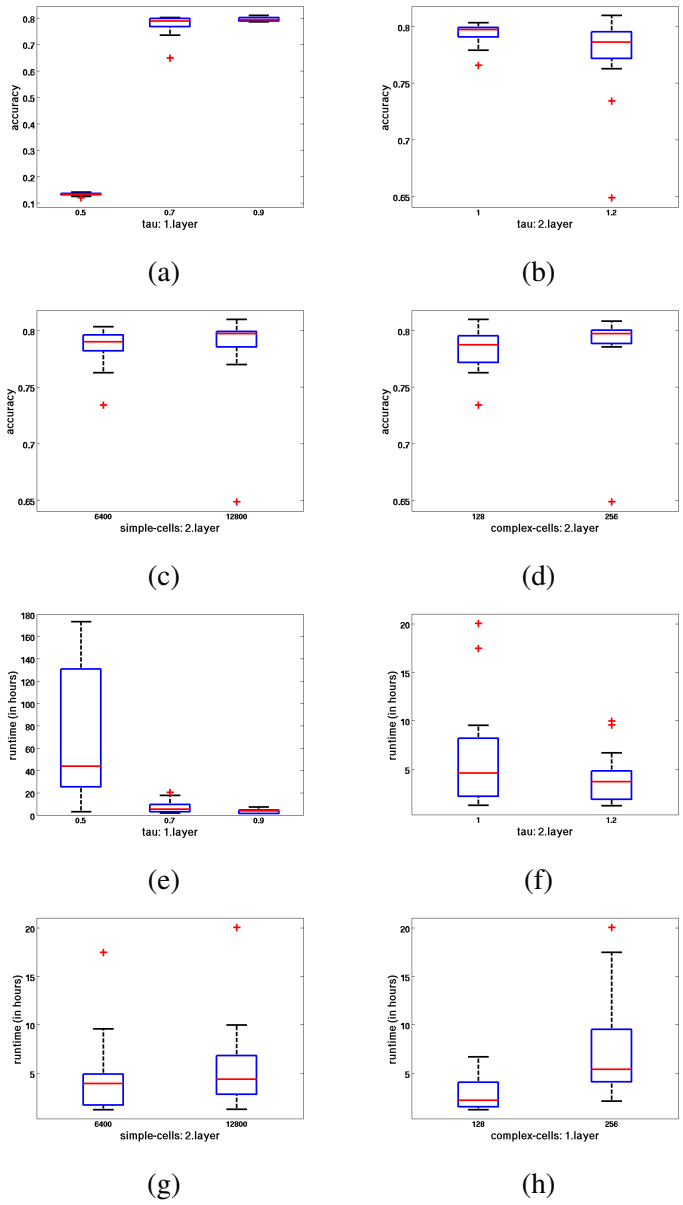


Figure 5.12: Approach 7 - In (b), (c), (d), (f), (g) and (h) only models are taken into account which are not using $\tau = 0.5$ in their first layer. An explanation for this is given in the text. In this figure box-plots show the influence of (a) τ of the first layer, (b) τ of the second layer, (c) the *number of simple cells in the second layer* and (d) the *number of complex cells in the second layer* on the accuracy. (e) τ of the first layer, (f) τ of the second layer, (g) the *number of simple cells in the second layer* and (h) the *number of complex cells in the first layer* are plotted against the runtime.

complex cells in the first layer, no clear correlation in terms of accuracy can be identified.

The strongest influence on the runtime of the K-means deep learning algorithm comes from the hyper-parameter τ of the first layer. As illustrated in Figure 5.12(e), a higher value of τ leads to a shorter runtime. τ of the second layer exhibits a similar behavior, though the influence is not as high as the parameter τ of the first layer, as illustrated in Figure 5.12(f). This influence on the training time can be explained when looking at the complex cell growing procedure. It is more likely that in each iteration more simple cells are added to one group, if a higher value of τ is used. Additionally, a higher value of τ allows to add simple cells into the same group which exhibit a higher dissimilarity than a lower value of τ. Therefore the probability, that the termination criterion is fulfilled earlier, is higher if τ is chosen to be higher. An explanation of the complex cell growing procedure is given in Section 3.4.2.

In Figure 5.12(g), the influence of the *number of simple cells in the second layer* on the runtime is shown. A higher number of simple cells leads to a longer training time, which is reasonable since more simple cells have to be updated during the training stage (a description of the training is given in Section 3.4.1). Also, a higher *number of complex cells in the first layer* leads to a longer runtime, which can be seen in Figure 5.12(h). Since more complex cells have to be built in the training procedure if this hyper-parameter is chosen to be higher, this behavior is reasonable, too.

To provide further in-depth information, a discussion of the influence of the hyper-parameters on the mean number of simple cells in one complex cell pooling group is given in Appendix A.7.

5.7 Relevant Observed Characteristics of Selected Methods

In this section, characteristics of selected methods are discussed in detail, where the model with the best hyper-parameter setting is examined for each method. As mentioned in Section 3.1, the main focus of deep learning is to automatically discover abstractions from low-level features to high-level representations. Therefore, the learned features of the neural network approaches and the simple and complex cells of the *K*-means deep learning algorithm are visualized and examined.

5.7.1 Characteristics of R^{2}SVM

The accuracy converges within 30 layers.

In Figure 5.13, the test accuracy is plotted against the number of layers of the R^{2}SVM, where the output of every layer is used to predict the examples of the test-set. In this way, the progress of accuracy as the number of layers increases can be illustrated. In this graphic it can be seen, that the accuracy converges within 30 layers, which coincides with the results presented in [82].

5.7.2 Characteristics of Stacked Sparse Auto-Encoders

The features in the higher layers are more class-specific than the features in the lower layers.

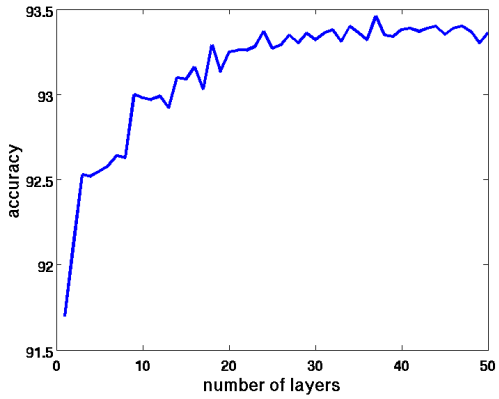
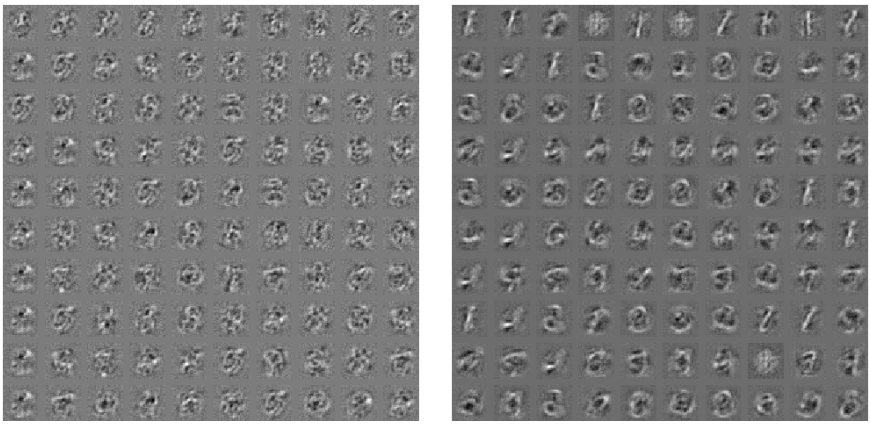


Figure 5.13: This plot illustrates the trend of the accuracy as the number of layers increases, where the whole MNIST training set is used to train the R^2SVM and the MNIST test set is used to compute the accuracy.

The model with the highest validation accuracy exhibits three hidden-layers and 2000 units per hidden-layer. Therefore, the first hidden layer consists of 2000 units, where every unit has 784 input-weight parameters, which can be visualized as 28-by-28 pixel filters, as done in [21, 52, 81]. The units of the subsequent hidden layers cannot be visualized like the neurons of the first hidden-layer, because every unit has 2000 input-weight parameters. This is why the higher layer features are visualized as a linear combination of the bottom layer filters, as explained in [52]. In Figure 5.14(a), the filters of the first hidden-layer of the stacked sparse auto-encoder are shown, where the filters are not selected randomly. Instead, for each digit-class of the MNIST dataset, all data examples of this class are fed to the network to compute the activations of the units. Then the mean activation of every unit is calculated and the units with the highest mean activation (of this class) are visualized. In Figure 5.14, every row corresponds to the visualized filters of one class.

The features of the second hidden-layer are shown in Figure 5.14(b) and are more class-specific than the filters of the first hidden-layer. For instance, if the filters of the first row (which exhibit the highest mean activation for class ”1”) are compared with each other, it can be seen that the first hidden-layer filters are much more generic than the features of the second layer: While for the first layer filters no clear common pattern can be identified, the features of the second layer exhibit a bright vertical line in the center. This means that in the second layer, the generic patterns of the first layer are combined in a way that more class-specific features are obtained for the hidden-units of this subsequent second layer.

The correlation matrices, shown in Figure 5.15, indicate the relation between the units in a layer of the network and the classes. They can be seen as an indicator for which classes the units are ”responsible”. A higher value denotes a strong correlation between the feature and the class, while a lower value indicates a weak correlation. The correlation matrix in Figure 5.15(a) shows the correlation between the classes and the features of the first hidden-layer, while the correlation matrix of the second hidden-layer is illustrated in Figure 5.15(b). Both the maximum correlation



(a) (b)

Figure 5.14: This graphic shows the features of the first and second hidden-layer of the trained (stacked sparse auto-encoder) network. In every row, the filters with the highest mean activation (of the corresponding class) are visualized, where the first row corresponds to the class ”1” and the last row to the class ”0”. (a) Visualization of first hidden-layer features. (b) The filters of the second hidden-layer, visualized as a linear combination of the first layer filters.

and the number of clear clusters increases with the number of layers. This is a reasonable explanation why the model with the best performance on the validation set exhibits three hidden-layers instead of two. The correlation matrix of the output-layer, illustrated in Figure 5.15(d), can be seen as the confusion-matrix and shows that all classes are predicted uniformly and no pair of classes is interchanged significantly often. In summary, the correlation matrices suggest that the features in the higher layers are more class-specific than the features in the lower layers, which coincides with the filter-visualizations in Figure 5.14.

Additionally, this observation is supported by an another way to visualize the features, where this visualization is provided in Appendix A.8.

5.7.3 Characteristics of Stacked Denoising Auto-Encoders

The features are more generic than the features of the stacked sparse auto-encoder approach. The features in the higher layers are more class-specific than the features in the lower layers.

The model with the highest validation accuracy exhibits four hidden-layers and 2000 units per hidden-layer. Again, the features of the first hidden-layer can be visualized as 28-by-28 pixel filters and the higher layer features are visualized as a linear combination of the lower layer filters, as described in Section 5.7.2.

The filters visualized in Figure 5.16(a) are the features of the first hidden-layer and look similar to the filters learned in [80] on the MNIST training set, where the model with the highest validation accuracy (in the experiments of this work) is trained with an amount of destruction of $\nu = 0.25$. A phenomenon that can be recognized for the second hidden-layer filters with

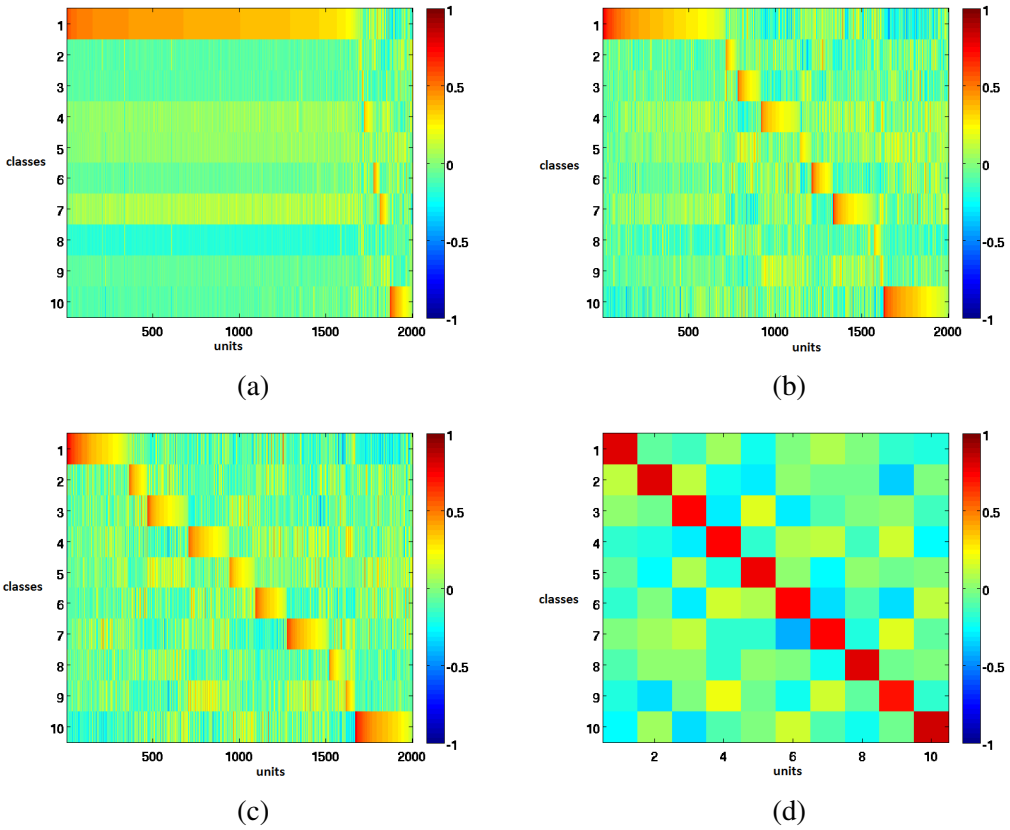
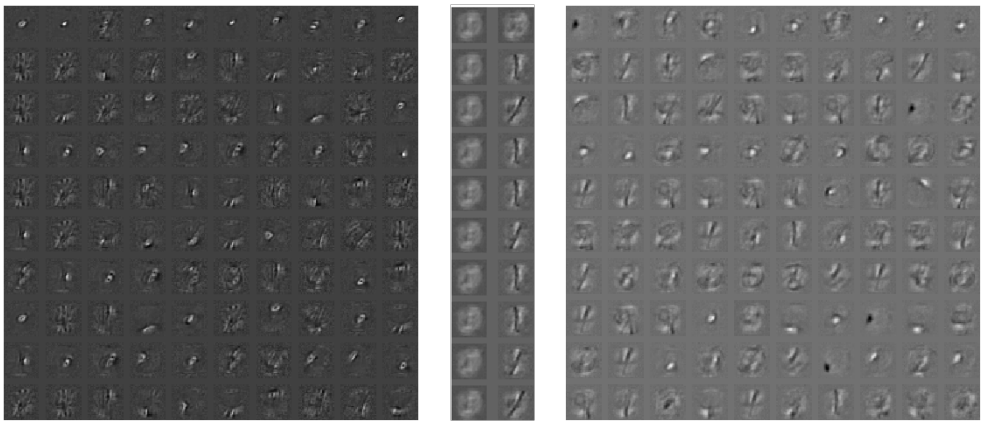


Figure 5.15: Illustration of the correlation matrices, which indicate the correlation between the units of a layer and the classes (”10” labels the class with the zero-digits). High values are denoted in red and indicate a strong correlation, while low values are highlighted in blue and indicate a weak correlation. Clustering and a subsequent re-ordering step is applied to improve the visualization of the matrices. (a) Correlation matrix of the first hidden-layer. (b) The correlation matrix of the second hidden-layer. (c) Correlation matrix of the third hidden-layer. (d) Correlation matrix of the output-layer.

the highest mean activation is the following: Approximately ten units have the highest mean-activation for all classes and are not class-specific, where a subset (the two highest) of these filters is shown in Figure 5.16(b)[4]. Since these filters exhibit the highest mean-activation for all classes, the units with the eleventh to the twentieth highest mean activation are visualized for the second hidden-layer in Figure 5.16(c). If this figure is compared with Figure 5.14(b), it can be seen that the second layer filters of the stacked denoising auto-encoder approach are much less class-specific than the second layer features of the stacked sparse auto-encoder.

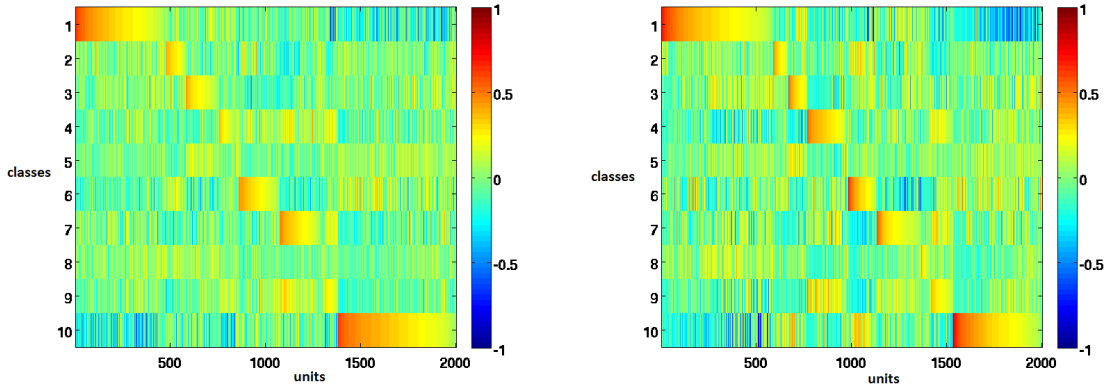
[4]This subset is a good representation for the features which exhibit the highest mean-activation for all classes, because all these filters look like the ones shown in Figure 5.16(b).



(a) (b) (c)

Figure 5.16: This graphic shows the features of the first and second hidden-layer of the trained (stacked sparse auto-encoder) network. In every row, the filters with the highest mean activation (of the corresponding class) are visualized, where the first row corresponds to the class ”1” and the last row to the class ”0”. (a) The filters of the first hidden-layer. (b) For every class, the two second hidden-layer features with the highest mean activation are shown in this graphic. (c) Second hidden-layer features with the eleventh to the twentieth highest mean activation.

This statement is supported by the correlation matrices illustrated in Figure 5.17(a) and Figure 5.17(b). In contrast to the correlation matrices of the stacked sparse auto-encoder, the stacked denoising auto-encoder shows two differences. Firstly, the number of clear clusters in the first layer is higher for the denoising auto-encoder, as can be seen in Figure 5.17(a). At the same time, the clusters of the denoising auto-encoder are not as clearly distinguishable from the background as the clusters of the sparse auto-encoder approach. This is a hint that the first layer features of the denoising auto-encoder can cover a wider variation of image-characteristics and are at the same time not as class-specific as the first layer features of the sparse auto-encoder. Secondly, the denoising auto-encoder exhibits fewer differences between the correlation matrices of different layers than the sparse auto-encoder. Thirdly, the clusters of higher layers with high correlation are not as clearly distinguishable from the background as the clusters of the sparse auto-encoder approach, as can be seen in Figure 5.17(b). This means that the features, which are learned by the denoising auto-encoder, are not specialized in one class, but instead exhibit a high correlation for multiple classes and are more generic than the features of the stacked sparse auto-encoder approach. For instance, Figure 5.17(b) shows that the units which exhibit a high correlation for class ”4”, also have a high correlation for class ”9”. This in turn means that it is likely that only a combination of these features makes it possible to distinguish between different classes.



(a) (b)

Figure 5.17: This figure contains the correlation matrices of the first and the fourth hidden-layer features of the stacked denoising auto-encoder approach. (a) The correlation matrix of the first hidden-layer. (b) Correlation matrix of the fourth hidden-layer.

5.7.4 Characteristics of K-means Deep Learning Algorithm

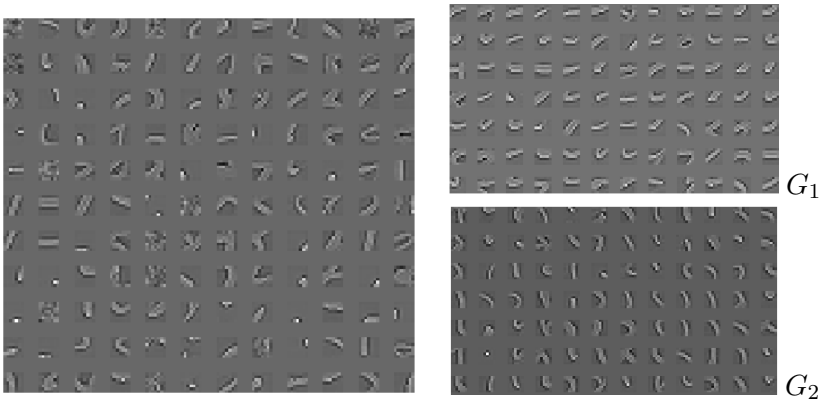
The K-means deep learning algorithm does not learn class specific features to the same extent as the neural networks on the MNIST dataset.

The model with the highest validation accuracy exhibits 12800 first layer simple cells, 12800 second layer simple cells, 128 first layer complex cells and 128 second layer complex cells. In Figure 5.18(a), a random subset of the first layer simple cells, which are learned by the *k*-means clustering algorithm, is depicted. Figure 5.18(b) shows sets of simple cells that belong to two pooling groups, which are learned by the complex cell learning procedure of the first layer. While the upper group G_1 contains edge-like filters, which are oriented horizontally, G_2 exhibits filters, which are oriented vertically.

As discussed in Section 5.4 and Section 5.6.7, the K-means deep learning algorithm does not help to improve the performance of the randomized tree approach on the MNIST dataset. This observation coincides with the correlation matrix shown in Figure 5.19: In comparison to the correlation matrices of the neural network approaches, the features of the K-means deep learning algorithm exhibit a much lower correlation and the clusters do not contrast as clearly with the background as the clusters in the correlation matrices of the neural network approaches. This means that the K-means deep learning algorithm does not learn class specific features to the same extent as the neural networks on the MNIST dataset.

5.8 Influence of Supervised Training Set Size

In this section, the impact on the performance of the methods when the number of labeled examples is varied, is evaluated. At the same time, the number of unlabeled examples remains the same. This means that the whole MNIST training set is used as the ”unlabeled training set”,



(a) (b)

Figure 5.18: This figure gives an overview of the learned features in the first layer of the K-means deep learning algorithm. (a) Randomly chosen first layer simple cells. (b) The simple cells of two pooling groups.

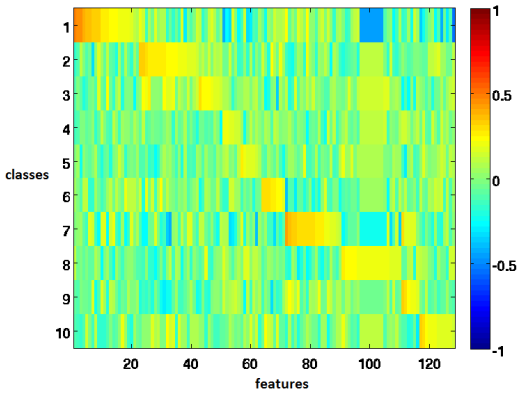


Figure 5.19: The correlation matrix of the K-means deep learning algorithm. This matrix denotes the correlation between the classes of digits and the output of the model trained by the K-means deep learning algorithm.

while only a subset of the MNIST training set is used as the ”labeled training set”. The following values are chosen to determine the size of the labeled training subset:

$$\{8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096\} \tag{5.10}$$

where the values indicate the number of examples used per class. For each method, the hyper-parameter setting with the highest validation accuracy (found in Section 5.6) is used for all experiments of this section. The influence of the number of labeled examples on the test accuracy is illustrated in Figure 5.20, where each method is highlighted in its own color. The correlation is the same for all approaches: a higher number of labeled training examples increased the test

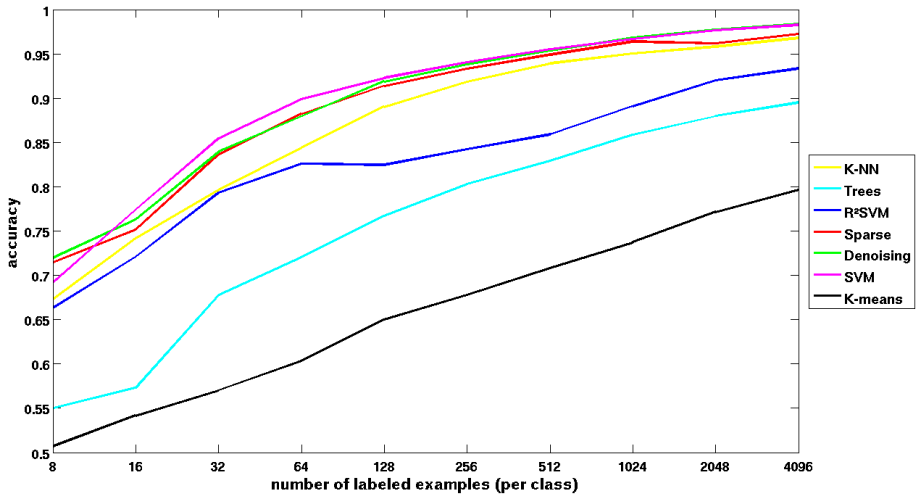


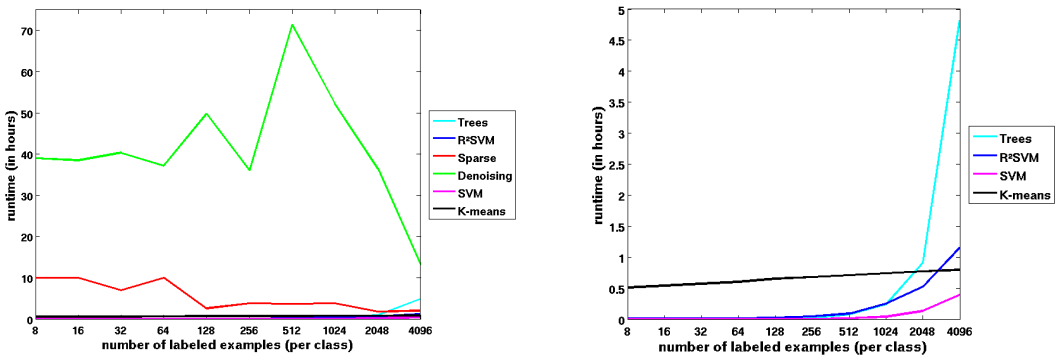
Figure 5.20: This graphic illustrates the correlation between the test accuracy and the number of labeled examples (which are used to train the model) for all methods.

accuracy. Furthermore, no curve in Figure 5.20 shows a clear convergence. This indicates that a higher number of patches would improve the accuracy.

Figure 5.20 shows that the influence of the number of examples used is strongest for the extremely randomized tree approach. If the number of labeled examples per class increases from 8 to 4096, the accuracy increases from 55.05% to 89.52%, an absolute increase of 34.47% (and a relative increase of 62.6%). As mentioned in Section 5.4, the K-means deep learning algorithm does not help to improve the performance of the extremely randomized trees. As can be seen in Figure 5.20, this is true for all examined numbers of labeled examples. This leads to the fact, that the K-means deep learning algorithm is the worst method among all evaluated approaches in terms of accuracy on the MNIST dataset, independent of the number of labeled training examples. An influence of the number of unlabeled training examples on the performance is not examined in this work, but the results indicate that training the K-means deep learning algorithm with 50000 patches is not enough.

Figure 5.21(a) shows the influence of the number of labeled training examples on training time for all methods. In contrast, Figure 5.21(b) shows only the runtime of the extremely randomized trees, the R^2SVM, the SVM and the K-means deep learning algorithm, where it can be seen that a higher number of labeled examples tends to increase the runtime of these four approaches and the extremely randomized trees shows the strongest dependence. The R^2SVM and the SVM approach exhibit the second and the third strongest influence of the number of labeled examples on the runtime (among these four methods). For the K-means deep learning algorithm, the number of labeled examples has nearly no influence on the training time[5]. Since

[5]This is reasonable, since the number of labeled training examples does only affect the training time of the extremely randomized trees.



(a) (b)

Figure 5.21: This graphic illustrates the correlation between the test accuracy and the number of labeled examples (which are used to train the model) for all methods. While (a) comprises the training times of all evaluated approaches, (b) illustrates the runtime of selected methods.

the K-nearest neighbor approach does not exhibit an explicit training phase, it is not shown in Figure 6.17.

No clear influence of the number of labeled examples on the training time can be identified for the sparse and the denoising auto-encoder in Figure 5.21(a). This is reasonable, since both neural network approaches are computed on the GPU and the advantage of the GPU lessens, as the number of training examples (and therefore the size of the data structures) decreases. Furthermore, only unlabeled training examples are used for the pre-training of the networks, which means that only the subsequent fine tuning stage is influenced by the number of labeled examples. The large differences between the training runtimes of the denoising auto-encoder (in comparison with the other approaches) can be explained by the fact that the servers are not reserved exclusively for our computations, which means that the activities of other users influence the runtime of our experiments.

Furthermore, the number of support vectors (of the SVM) increases, as the number of training examples is increased, which is discussed in detail in Appendix A.9.

5.9 Summary

In this section, the main results of this chapter are summarized. The stacked auto-encoder approaches and the SVMs achieve a test accuracy above 98%, but at the same time exhibit the longest training time among all evaluated methods. On the other hand, the R^2SVMs and the extremely randomized trees show a shorter training time, but a lower test accuracy than the aforementioned approaches. Beside these general observations, the results show that the K-means deep learning algorithm does not help to improve the performance of the extremely randomized tree approach on the MNIST dataset.

The results of the SVM approach show that γ exhibits the strongest influence on the performance. Concerning the K-means deep learning algorithm, τ of the first layer exhibits the

strongest influence on the accuracy among all hyper-parameters of this method. The *activation regularization coefficient* of the sparse auto-encoder exhibits the strongest influence on the accuracy among all hyper-parameters. In contrast, for the denoising auto-encoder, the *learning rate* shows the strongest influence on the accuracy. Moreover, the results indicate that the denoising auto-encoder approach is more stable than the sparse auto-encoder approach.

For the R^2SVM, the accuracy converges within 30 layers. Higher layer features of the stacked sparse auto-encoder approach show a higher correlation and are more class-specific than the features of lower layers. Furthermore, the features of the stacked denoising auto-encoder are more generic (less class-specific) than the features of the sparse auto-encoder.

Finally, a higher number of labeled training examples increases the accuracy of all evaluated methods, while a higher number increases the training time of all methods except both auto-encoder approaches and the K-nearest neighbor method (because this approach exhibits no explicit training phase).

CHAPTER 6

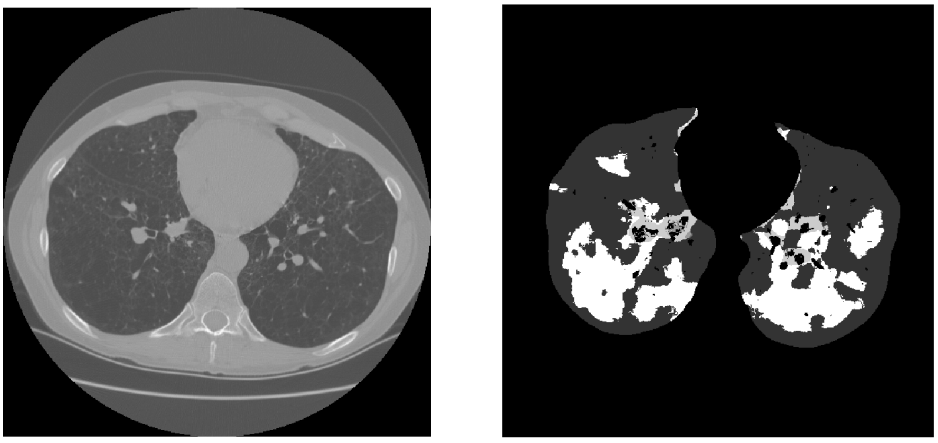
Comparison of Learning Approaches on the PULMO Dataset

In this chapter the same methods which are examined in Chapter 5, are evaluated on the *PULMO dataset*. To be exact, the methods which are discussed in Chapter 2 (K-Nearest Neighbors, SVM, Extremely Randomized Trees) and Chapter 3 (Stacked Auto-Encoders, R^2SVM, K-means Deep Learning Algorithm) are evaluated on the *PULMO dataset* [40], where the implementations described in Chapter 4 are used again for this evaluation. An overview of the evaluated methods and the evaluation criteria are shown in Table 5.1. Furthermore, the same evaluation procedure as in Chapter 5, which can be seen in Figure 5.1, is used. First, an explanation of the PULMO dataset is given in Section 6.1. After a discussion of the choice of the parameters is provided in Section 6.2, an evaluation of the accuracy and runtime of the above mentioned methods is done in Section 6.3 and Section 6.4. While the impact of the hyper-parameters on the performance is analyzed in Section 6.5, the effect of changing the number of labeled examples for training is examined in Section 6.7. In the end, a summary of the evaluation on the PULMO dataset is provided in Section 6.8.

6.1 PULMO Dataset

The PULMO dataset is comprised of 120000 28-by-28 patches in total, which have been sampled from 128 Computed tomography (CT) volumes of the thorax. The volumes are provided by the Lung Tissue Research Consortium (LTRC). The LTRC is a project with the aim to collect comprehensive data including a CT imaging database that contains volumetric high-resolution CTs of the thorax [40]. The volumes are annotated by expert radiologists and contain six different classes: [40, 85]:

- **Original:** Part of the CT scan which cannot be classified into the other five classes. Contains structures like the heart or bones and corresponds to ”background”.



(a) (b)

Figure 6.1: Example of the CT data provided by the LTRC. (a) One 512-by-512 slice of a volume. (b) The corresponding annotation of the slice, where the classes are highlighted in different grey levels.

- **Normal:** Healthy pulmonary tissue which contains small vessels and bronchioles [85].
- **Ground glass:** This class denotes areas of increased attenuation in the lung, where the vessels and bronchioles remain apparent. This increased homogeneous opacity is not disease-specific and can occur due to fibrosis, interstitial cellular infiltration, edema or tissue compression [4, 85].
- **Reticular:** Tissue which is characterized by abnormal irregular linear opacities of pulmonary parenchyma. This may appear due to parenchymal fibrosis [4, 67, 85].
- **Honeycombing:** The CT shows clustered cyclic air spaces that are separated from the airways and characterized by well-defined walls. The cysts have variable sizes from microscopic to several millimeters [85]. This appearance corresponds to the end stage of fibrosing lung disease [4].
- **Emphysema:** Regions of decreased attenuation due to enlarged air-spaces and destroyed alveolar walls. These changes mean a destruction of the pulmonary parenchyma [4, 85].

An example for one slice of a volume and its corresponding annotation can be seen in Figure 6.1. The slices have a size of 512-by-512, where one volume comprises about 500 slices (the number of slices varies). The pixel values lie in the range of $[-2000, 4095]$. Since the objective of this work is to compare machine learning methods in terms of classification, an appropriate dataset has to be defined. Therefore 28-by-28 patches are cut out from the three-dimensional volumes, where the patches are lying in the *axial plane*. For each volume, 3000 patches are selected randomly per class, so 384000 patches are created altogether for each class. The only exception is the class *Emphysema* with 245000 patches, since there are not enough annotations in

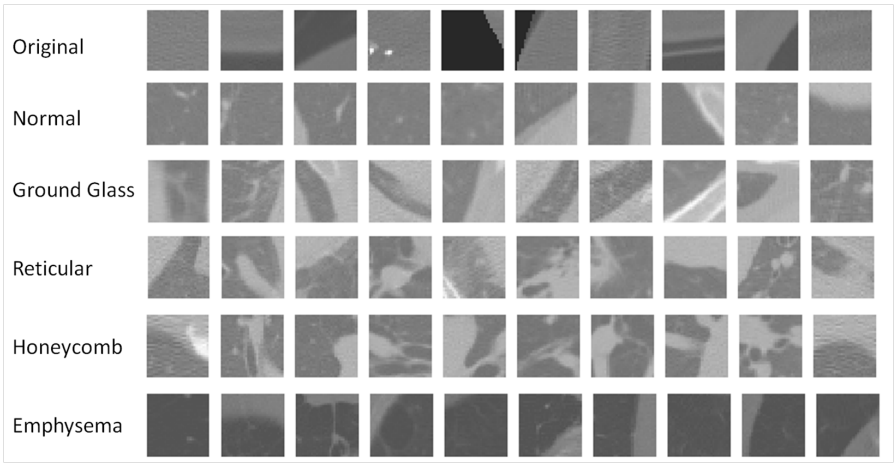


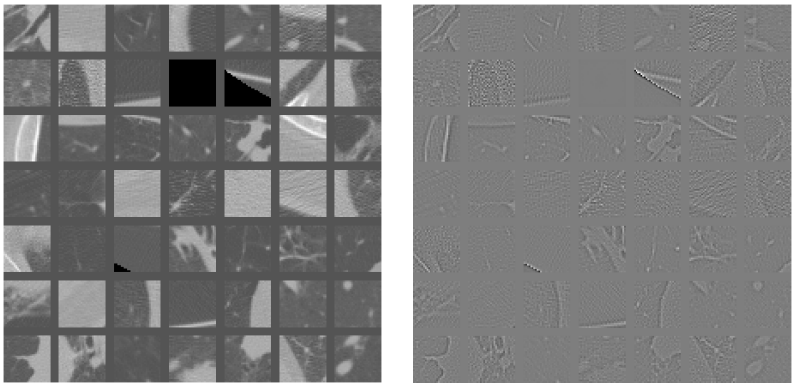
Figure 6.2: A section of the PULMO dataset. One row corresponds to one class, where ten patches are shown for each class.

every volume (for some volumes fewer patches are created). From this vast amount of patches, 20000 examples are chosen randomly for every class, in order to create the final database of 120000 28-by-28 patches. The label of every patch is determined by the label of the center-pixel, which means that each patch could contain image-parts of more than one class. Finally, the dataset is randomly divided into three parts: the training, validation and test set. The training set is composed of 16000 examples per class and has therefore an overall size of 96000 patches. Both the validation and the test set have an overall size of 12000 examples, with 2000 patches from each class. Due to the fact that three methods (stacked sparse auto-encoders, stacked denoising auto-encoders, K-means deep learning algorithm) need unsupervised data for training, the whole training set (comprising 96000 examples) is also used as an unsupervised dataset by ignoring the labels. In Figure 6.2, patches of the PULMO dataset can be seen.

6.2 Parameters of the Algorithms

In the experiments with the PULMO dataset, a preprocessing pipeline is used which consists of feature standardization and a subsequent whitening stage. First, feature standardization is performed, where the mean and the standard deviation is computed for every dimension along all data examples of the training set. Subsequently, the following is done for every image: the mean is subtracted from each dimension and every dimension is divided by its standard deviation. This feature standardization is often used to transform raw data into an appropriate input for machine learning methods, as stated in [9, 21]. After these normalization steps, ZCA-whitening is applied to remove correlations between input values[1], where whitening is a common step in deep learning work, as stated in [21]. The resulting patches, if this preprocessing pipeline is ap-

[1] As mentioned in Section 2.1.5, a small constant ϵ is added to avoid numerical instabilities. In this work, $\epsilon = 10^{-5} = 0.00001$ is used.



(a) (b)

Figure 6.3: Example showing the effect of the preprocessing pipeline used for the PULMO dataset. (a) Original patches of the PULMO dataset. (b) The same patches, but after preprocessing.

plied to the images of the PULMO dataset, are shown in Figure 6.3(b), where the corresponding original patches are illustrated in Figure 6.3(a). One exception in terms of preprocessing is the K-means deep learning algorithm. Since this approach already includes a preprocessing stage, no preprocessing is performed on the images for this method.

In the experiments concerning the PULMO dataset, the intervals and ranges of the hyper-parameters are selected according to the choices in the experiments of the MNIST dataset. In order to get results in this section which are comparable to the ones in Chapter 5, the same hyper-parameters are used here, if possible. In the following it is explained which hyper-parameters are used for the methods on the PULMO dataset.

For the K-Nearest Neighbor algorithm, the extremely randomized trees, the SVMs, the R^2SVM and the K-means Deep Learning Algorithm, the same values as in the MNIST experiments are used for the hyper-parameters. The values of these methods can be found in Section 6.2.

6.2.1 Approach 5: Stacked Sparse Auto-Encoders

As described in Section 5.4 and Section 5.5, the high mean runtime (nine hours) to train a sparse auto-encoder on the MNIST dataset and the high number of settings (324) leads to a total runtime of about two months on our machines. This is the reason for evaluating a reduced number of settings on the PULMO dataset (to reduce the total runtime). So we decided to use the same configuration as described in Section 5.3.5, with the exception of the following hyper-parameters:

$$\text{learning rate} = \{10^{-4}, 10^{-3}\} = \{0.0001, 0.001\} \tag{6.1}$$

$$\text{weight decay regularization coefficient} = \{0.003, 0.1\} \tag{6.2}$$

$$\text{activation regularization coefficient} = \{0.003, 0.1\} \tag{6.3}$$

Method	Test Accuracy (PULMO)	# Varied Parameters	# Settings
Stacked Sparse Auto-Encoder	63.95%	6	96
SVM	56.49%	2	25
Stacked Denoising Auto-Encoder	55.18%	6	96
R^2SVM	49.00%	2	15
K-means Deep Learning Algorithm	37.76%	5	48
Extremely Randomized Trees	31.72%	1	5
Baseline	16.67%	-	-
K-Nearest Neighbor	15.33%	1	3

Table 6.1: This table gives an overview of the obtained test-accuracies. The selected models (the ones with the highest validation accuracy) were applied on the PULMO test set. Beside the accuracy, the table summarizes the number of varied parameters and the number of settings (i.e. the number of trained models) for each method.

6.2.2 Approach 6: Stacked Denoising Auto-Encoders

Such as for the sparse auto-encoders, the number of settings (216, as described in Section 5.4) and the mean training time on the MNIST dataset (fifteen hours) are also high. Since the total runtime for this method (on the MNIST dataset) is about two months on our machines, the number of settings is reduced for the stacked denoising auto-encoder approach, too. The same configuration which is discussed in Section 5.3.6 is also used for the PULMO dataset, with the exception of the following hyper-parameters:

$$\text{learning rate} = \{10^{-4}, 10^{-3}\} = \{0.0001, 0.001\} \tag{6.4}$$

$$\text{weight decay regularization coefficient} = \{0.003, 0.1\} \tag{6.5}$$

6.3 Comparison of Accuracy

In this section, the test-accuracies of the evaluated methods, received on the PULMO dataset, are presented and compared. Table 6.1 shows the test accuracy, the number of varied hyper-parameters and the total number of trained models for each method. The baseline of 16.67% denotes the revealed accuracy of a simple model which classifies all images in the test set randomly.

For the PULMO dataset, the stacked sparse auto-encoder exhibits a test accuracy of 63.95% and therefore the highest test accuracy among all evaluated methods. As for the MNIST dataset, the SVM approach is the method which achieves the second highest test accuracy (56.49%), though fewer models (25) are trained than for the denoising auto-encoder approach (96) which achieves a test accuracy of 55.18%. The R^2SVM exhibits a test accuracy of 49.00%, where

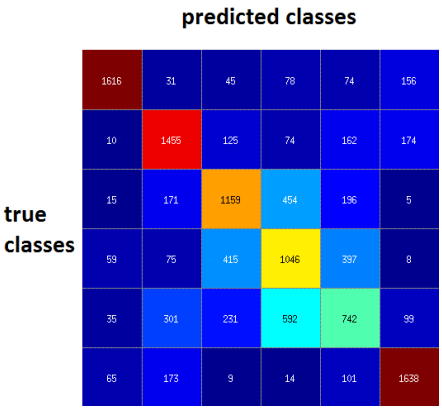


Figure 6.4: The confusion matrix of the sparse auto-encoder with the highest validation accuracy. The rows denote the known classes, while the columns depict the predicted classes. For instance, the number 301 in row 5 and column 2, is a count of observations known to be in class 5 (”Honeycombing”) but predicted to be in class 2 (”Normal”). The first row, respectively column corresponds to the class *”Original”*, the second to *”Normal”*, the third to *”Ground Glass”*, the fourth to *”Reticular”*, the fifth to *”Honeycombing”* and the sixth to the class *”Emphysema”*.

the results in Section 6.6.1 suggest, that a higher number of layers would lead to an increased accuracy. The extremely randomized trees do not perform well (31.72%) in comparison to the other approaches, which coincides with the results presented in Section 5.4. But in contrast to the results in Section 5.4, the K-means deep learning algorithm helps to improve the performance of the extremely randomized trees 37.76%. More precisely, the test accuracy of the extremely randomized tree approach is improved by 7.37% (from 30.39% to 37.76%)[2], which means a relative improvement of 24.3%. The K-nearest neighbor approach fails to solve the classification problem on the PULMO dataset, since the best model exhibits a test accuracy of 15.33% which is lower than the baseline (16.67%).

The confusion matrix of the stacked sparse auto-encoder approach is illustrated in Figure 6.4, where the general structure of this confusion matrix is representative for the confusion matrices of all methods except the K-nearest neighbor approach. The K-nearest neighbor approach predicts a vast amount of examples as *”Original”*, which means that the first column of the confusion matrix contains only values around 1800. In contrast, the confusion-matrices of all other approaches exhibit high values in the rows and columns ”3”, ”4” and ”5”, which can be seen in Figure 6.4. This indicates, that the classes *”Ground glass”*, *”Reticular”* and *”Honeycombing”* are very hard to distinguish from one another.

[2]Since a tree with $M = 64$ is used (as described in Section 5.3.7), we compare against the test accuracy of the extremely randomized tree with 64 trees.

Method	Runtime for Training in hours (mean / median)	Runtime for Prediction (mean)
K-Nearest Neighbor	none	33 minutes
R^2SVM	1.4 / 1.2	50 seconds
K-means Deep Learning Algorithm	3.7 / 1.7	50 seconds
Extremely Randomized Trees	4.7 / 4.7	30 seconds
Stacked Denoising Auto-Encoder	6.3 / 5.3	30 seconds
Stacked Sparse Auto-Encoder	7.3 / 6.3	30 seconds
SVM	101.7 / 97.3	43 minutes

Table 6.2: This table gives an overview of the average time needed to train one setting of a method on the PULMO training set, respectively the average time needed to calculate the predictions for the PULMO validation set. The runtime is listed for each approach and is the mean and median of the time over all models of a method. One exception is the K-nearest neighbor approach that exhibits no training stage.

6.4 Comparison of Runtime

Table 6.2 gives an overview about the average time which is needed to train one model of a method. Furthermore, the mean time needed to calculate the predictions for the PULMO validation set is given, too.

With the exception of the K-nearest neighbor approach, which exhibits no explicit training stage, the R^2SVMs have the shortest mean training time among all evaluated approaches. The K-means deep learning algorithm exhibits a mean training time of 3.7 hours, where the reduced runtime in comparison with the MNIST runtime can be explained by the fact that an optimized version of the complex cell training procedure is used for the PULMO dataset[3]. The extremely randomized tree approach, on the other hand, shows a significant higher mean runtime for the PULMO dataset (4.7 hours) than for the MNIST data. The stacked denoising auto-encoder and the stacked sparse auto-encoder exhibit a mean runtime of six, respectively seven hours. The difference in comparison with the mean runtimes on the MNIST training set are possibly caused by the activities of other users on the machines where the computations are executed. The SVMs exhibit by far the longest mean runtime among all evaluated methods, where this caused by the number of training examples (respectively by the number of support vectors), as discussed in Section 6.7.

A box-plot of the training times of all evaluated methods is given in Figure 6.5, where for each method the distribution of the runtime is plotted. The results in this figures coincide with the results presented in Table 6.2. Additionally, this figure illustrates that the training times of the SVM approach are widely distributed in comparison to the other methods.

[3]The logic is the same in both ”versions” of the complex cell training procedure. The only difference is, that for the PULMO dataset some tricks like vectorization were used to optimize the code.

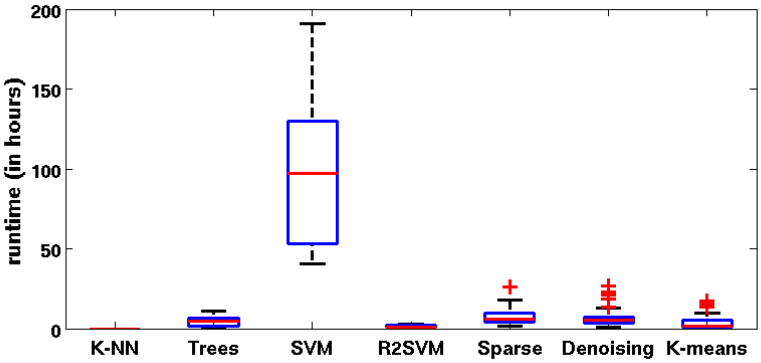


Figure 6.5: This box-plot illustrates the training-times of the methods, where each distribution is computed by considering all training-times of a method (for each setting one training-time).

Except for the K-nearest neighbor and the SVM approach, all methods exhibit a mean prediction time inside of one minute. As for the MNIST dataset, the mean time of the SVM (43 minutes) to predict the PULMO validation set (16000 examples) is longer than the mean prediction time of the K-nearest neighbor approach (33 minutes).

6.5 Influence of Hyper-Parameters

In this section, the influence of the hyper-parameters on characteristics of the methods, like validation accuracy and runtime, is examined and discussed in detail.

6.5.1 Approach 1: K-Nearest Neighbors

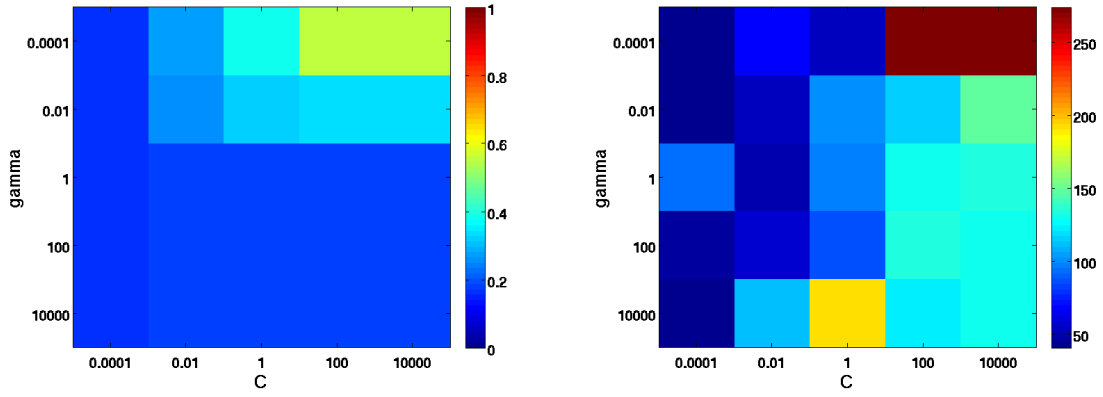
No clear influence of the hyper-parameter k can be identified.

In relation to validation accuracy, no clear influence of the hyper-parameter k can be identified. The validation-accuracies for all three values of k range between 15.28% and 15.72%, which means that k does not help to increase the accuracy of the model above the frequency-baseline of 16.67%. In terms of prediction runtime, no clear tendency can be identified, which coincides with the results presented in Section 5.6.1.

6.5.2 Approach 2: Extremely Randomized Trees

A higher number of trees leads to a higher accuracy, where the accuracy converges within 128 trees. Both the training and prediction runtime increase linearly with the number of trees.

There is only one hyper-parameter of the extremely randomized tree approach that is varied within our experiments on the PULMO dataset. The validation accuracy converges within the



(a) (b)

Figure 6.6: Approach 3 - This figure illustrates the influence of the SVM hyper-parameters C and γ on the (a) accuracy and (b) the time to train the SVM, where the runtime is plotted in hours.

examined values for the number of trees: While the relative increase is 5.2% (28.73% to 30.23%) when the number of trees is increased from 16 to 32, the relative increase is only 0.99% (31.3% to 31.61%) when the number of trees is doubled from 64 to 128. As for the MNIST dataset, both the training and prediction time of the extremely randomized tree approach increase linearly with the number of trees. The results presented in this paragraph coincide with the results discussed in Section 5.6.2 and in [36, 63]. Additionally, a Figure illustrating the influence of the number of trees on the accuracy and the prediction runtime is provided in Appendix B.1.

6.5.3 Approach 3: SVM

A higher value of C tends to increase the accuracy and the training time. A lower γ leads to an increased accuracy, but does not affect the training time.

Since the SVM approach exhibits two hyper-parameters, Figure 6.6(a) illustrates the influence of C and γ on the accuracy, where the colors denote the value of the accuracy. The green color highlights a model with a higher accuracy, while a dark blue color denotes a model with a lower accuracy. In this figure it can be seen, that a higher C tends to increase the validation accuracy of the SVM. In contrast, a higher value of γ leads to a lower accuracy. These results are in accordance with the results for the MNIST database, except that for the PULMO dataset the influence of γ is not stronger than the influence of C in terms of accuracy.

The influence of C and γ on the runtime can be seen in Figure 6.6(b), where a higher C leads to a higher training runtime. This correlation can be explained by the fact that a higher C tends to increase the model complexity of the SVM, as mentioned in Section 5.6.3. On the other hand, no clear influence of gamma on the runtime can be identified for the PULMO dataset.

A higher C tends to decrease the number of support vectors in the final SVM. This coincides with the results for the MNIST dataset concerning the influence of C on the number of support

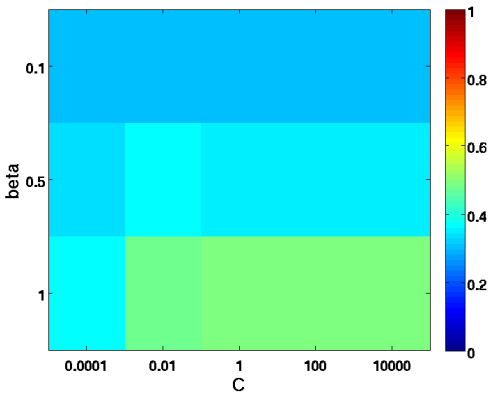


Figure 6.7: Approach 4 - Illustration of the influence of the R^{2}SVM hyper-parameters C and β on the validation accuracy.

vectors. In comparison with the results on the MNIST database, the results concerning γ differ: no clear influence of γ on the number of support vectors can be identified.

The high mean prediction runtime of the SVM approach of 43 minutes can be explained by the high mean number of support vectors (94762). Furthermore, a lower number of support vectors tends to increase the validation accuracy. Finally, the correlation between the validation accuracy and the training runtime, identified in Section 5.6.3 for the MNIST database, cannot be detected for the PULMO dataset.

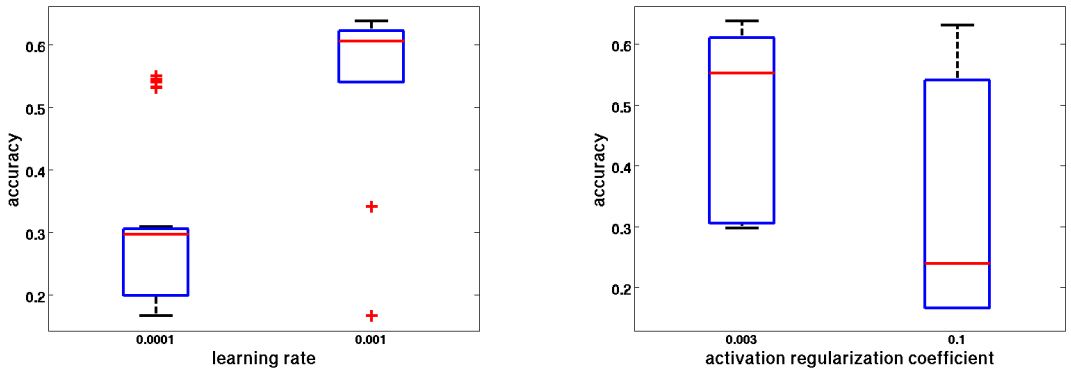
Illustrations which support the findings described in this paragraph, can be found in Appendix B.2.

6.5.4 Approach 4: R^{2}SVM

A higher value of β leads to a higher accuracy.

The influence of C and β on the validation accuracy is depicted in Figure 6.7, where it can be seen that a higher value of β leads to a higher validation accuracy. While this correlation coincides with the result for MNIST (discussed in Section 5.6.4), the result for C differs: on the PULMO dataset, no clear tendency can be identified. In relation to runtime, no clear correlation can be found, neither for C nor for β.

Finally, it can be stated that the influence of the SVM hyper-parameters is stronger than the influence of the R^{2}SVM hyper-parameters, though the difference is not as significant as for the MNIST dataset (details are provided in Appendix B.3).



(a) (b)

Figure 6.8: Approach 5 - This figure illustrates the influence of (a) the *learning rate* and (b) the *activation regularization coefficient* on the validation accuracy.

6.5.5 Approach 5: Stacked Sparse Auto-Encoders

In respect of accuracy, the *learning rate* and the *activation regularization coefficient* exhibit the strongest influence. Both a higher number of hidden-layers and a higher number of hidden-units increase the training time.

Though the hyper-parameters of the stacked sparse auto-encoder exhibit a weaker influence on the validation accuracy on the PULMO dataset than on the MNIST examples, the influence is significant. The range of the obtained validation-accuracies stretches from 16.67% to 63.80% and exhibits a standard-deviation of 18.55%. This result supports the statement that the choice of the stacked sparse auto-encoder hyper-parameters strongly influence the performance of the finally obtained network. The lower standard-deviation on the PULMO dataset can be explained by the fact, that we reduced the number of settings from 324 (MNIST) to 96 (PULMO) for computational reasons. In contrast to the results presented in Section 5.6.5, the *learning rate* shows the strongest influence on the validation accuracy among all hyper-parameters, as depicted in Figure 6.8(a). This figure illustrates that a higher *learning rate* leads to a higher validation accuracy, which coincides with the statements in [9]. The parameter with the second highest influence on the validation accuracy is the *activation regularization coefficient*, where 0.003 leads to a better accuracy than 0.1, as can be seen in Figure 6.8(b). Furthermore, the usage of the *momentum*, a higher number of hidden-units and a higher number of layers all tend to increase the accuracy. More details can be found in Appendix B.4. The weight parameter is the only hyper-parameter of the stacked sparse auto-encoder approach that shows no influence on the validation accuracy.

A higher learning rate slightly decreases the runtime. Both a higher number of hidden-layers and a higher number of hidden-units increase the time needed for training. A lower value of the *activation regularization coefficient* tends to decrease the runtime, which can be seen when comparing the mean runtime of all models that use 0.003 (7.7 hours) with the mean runtime

of all models which use 0.1 (6.9 hours). In contrast to the results presented in Section 5.6.5, a higher *weight decay regularization coefficient* tends to decrease the training time on the PULMO dataset. For the *momentum*, no clear influence on the training runtime can be identified. Figures supporting these observations can be found in Appendix B.4.

Though the prediction runtime is below 90 seconds for all networks, a clear influence of the number of layers and the number of hidden-units can be identified. A higher number of hidden-units as well as a higher number of hidden-layers tends to increase the prediction runtime. Among the other hyper-parameters (*learning rate*, *momentum*, *weight decay regularization coefficient*, *activation regularization coefficient*), no clear influence on the prediction runtime can be identified.

6.5.6 Approach 6: Stacked Denoising Auto-Encoders

The parameter-selection is not as sensitive as the parameter-choice for sparse auto-encoders. The *learning rate* has the highest influence on the accuracy. Both a higher number of hidden-units and a higher number of hidden-layers tends to increase the training runtime

The validation-accuracies of the stacked denoising auto-encoder models exhibit a standard deviation of 11.17%, where the range stretches from 27.00% to 55.83%. This indicates that the influence of the hyper-parameters is weaker in comparison to the sparse auto-encoders and that the denoising auto-encoders are a more stable approach than the sparse auto-encoders. This coincides with the results for the MNIST dataset (discussed in Section 5.6.6), though the difference on the PULMO dataset is not as clear as for the MNIST examples.

The *learning rate* has the highest influence on the validation accuracy, as illustrated in Figure 6.9(a). In this graphic it can be seen that a higher learning rate tends to increase the validation accuracy. Though the influence of the *momentum* on the accuracy is not as strong as the influence of the *learning rate*, the usage of the *momentum* increases the validation accuracy. This correlation is depicted in Figure 6.9(b). In contrast to the results in Section 5.6.6, the validation accuracy of the stacked denoising auto-encoder is both influenced by the number of hidden-layers and the number of hidden-units (considering the PULMO dataset). In Figure 6.9(c), it can be seen that a higher number of layers increases the accuracy. On the other hand, an increased number of hidden-units does not help to improve the accuracy, but does not affect the generalization performance much, as illustrated in Figure 6.9(d). This coincides with the statement in [9] that it is most important to choose the number of units large enough and higher values than the optimal number do not hurt the generalization performance much. Another difference in comparison with the MNIST results is that the *corruption parameter* $\nu = 0.25$ shows a better median accuracy (51.47%) than $\nu = 0.5$ (48.17%) on the PULMO dataset. For the *weight decay regularization coefficient*, no clear influence on the validation accuracy can be identified.

In terms of the time needed to train one model, correlations can only be identified for the number of hidden-units and the number of hidden-layers. Both a higher number of hidden-units and a higher number of hidden-layers tends to increase the training runtime. An explanation for this relation can be found in Section 5.6.5.

With reference to the prediction runtime (all runtimes lie within 90 seconds), the same correlations as for the sparse auto-encoders can be detected: Both a higher number of hidden-units

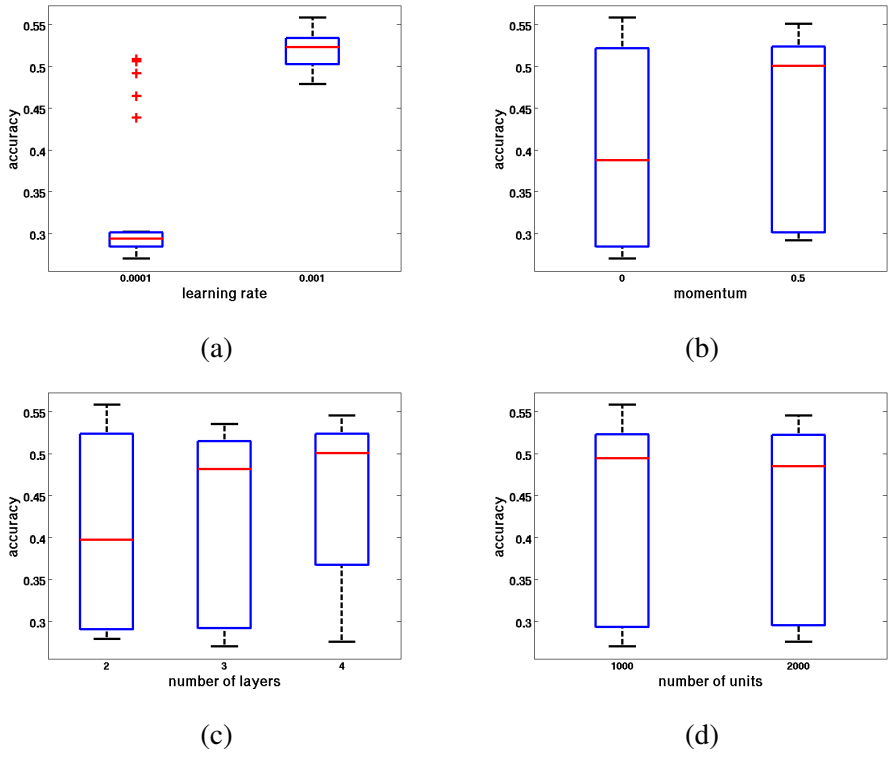


Figure 6.9: Approach 6 - This figure illustrates the influence of (a) the *learning rate*, (b) the momentum, (c) the number of hidden-layers and (d) the number of hidden-units on the validation accuracy.

and a higher number of hidden-layers tend to increase the prediction runtime. Among the other hyper-parameters (*learning rate*, *weight decay regularization coefficient*, *momentum*, ν) no clear influence on the prediction runtime can be identified.

6.5.7 Approach 7: K-means Deep Learning Algorithm

The hyper-parameter τ of the first layer exhibits the highest influence on the accuracy and the runtime.

While the K-means deep learning algorithm does not help to improve the performance on the MNIST dataset, it helps do improve the performance of the extremely randomized tree approach on the PULMO dataset, as mentioned in Section 6.3. A possible reason for this could be the number of unlabeled patches: While 96000 patches are used to train the K-means deep learning algorithm on the PULMO dataset, only 50000 patches are used within the MNIST experiments.

In Figure 6.10(a) the validation accuracy is plotted against the τ of the first layer. As for the MNIST dataset, τ of the first layer shows the highest influence on the accuracy among all evaluated hyper-parameters. In order to highlight the influence of the number of simple cells

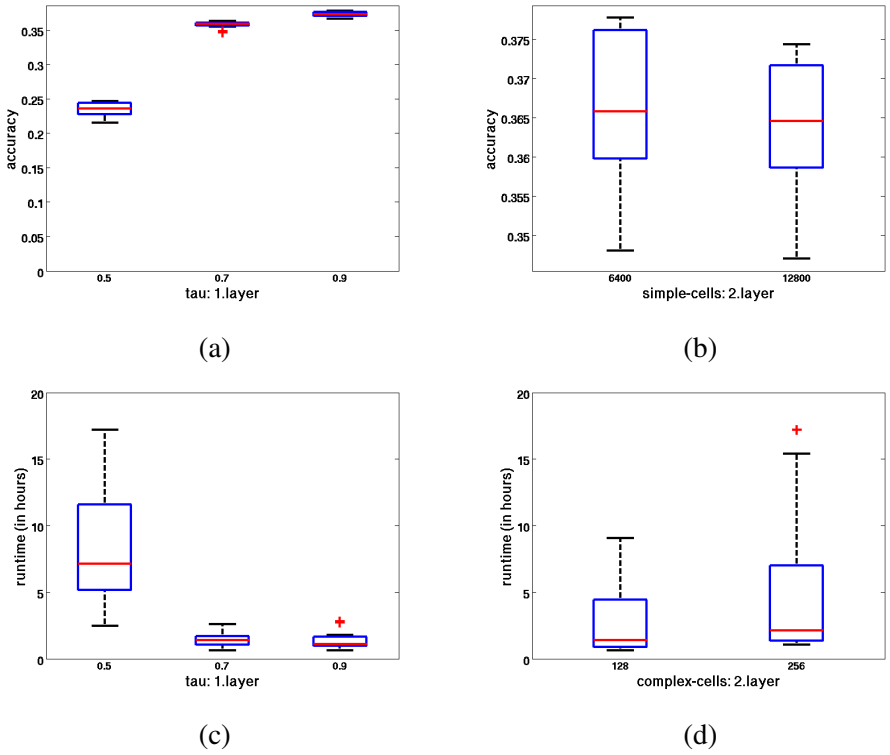


Figure 6.10: Approach 7 - In this figure box-plots show the influence of (a) τ of the first layer and (b) the number of simple cells in the second layer on the validation accuracy. In (b), only models are considered which are not using $\tau = 0.5$ in their first layer to highlight the influence of the hyper-parameter. Furthermore, (c) τ of the first layer and (d) the number of complex cells in the second layer are plotted against the runtime.

in the second layer on the validation accuracy, in Figure 6.10(b) only models are considered which are not using $\tau = 0.5$ in their first layer. In this figure it can be seen that 6400 cells tend to increase the accuracy in comparison to 12800 cells, though the influence is weaker than the influence of τ of the first layer. Among the other hyper-parameters (τ of the second layer, number of complex cells in the first and second layer), no clear influence on the validation accuracy can be identified.

In terms of training runtime, τ of the first layer shows the highest influence, where this can be seen in Figure 6.10(c). A higher τ of the first layer leads to a shorter runtime. This result coincides with the result presented in Section 5.6.7. In Figure 6.10(d), the number of complex cells in the second layer is plotted against the runtime. It can be seen that a higher number of complex cells tends to increase the training time, though the influence is not as high as the influence of τ of the first layer. Furthermore, both a higher number of complex cells in the first layer and a higher number of simple cells in the second layer lead to a longer training runtime. The only hyper-parameter which shows no influence on the training time on the PULMO dataset,

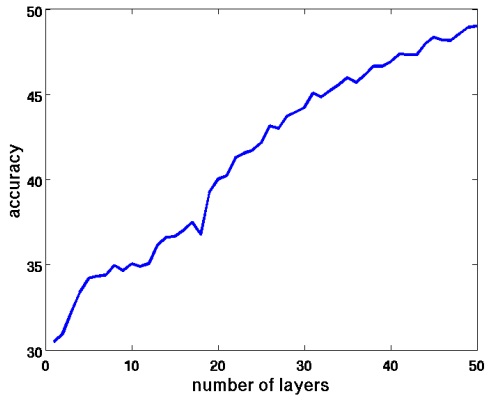


Figure 6.11: This plot illustrates the trend of the accuracy as the number of layers increases, where the whole PULMO training set is used to train the R^2SVM and the PULMO test set is used to compute the accuracy.

is τ of the second layer.

To provide further in-depth information, a discussion of the influence of the hyper-parameters on the mean number of simple cells in one complex cell pooling group can be found in B.5.

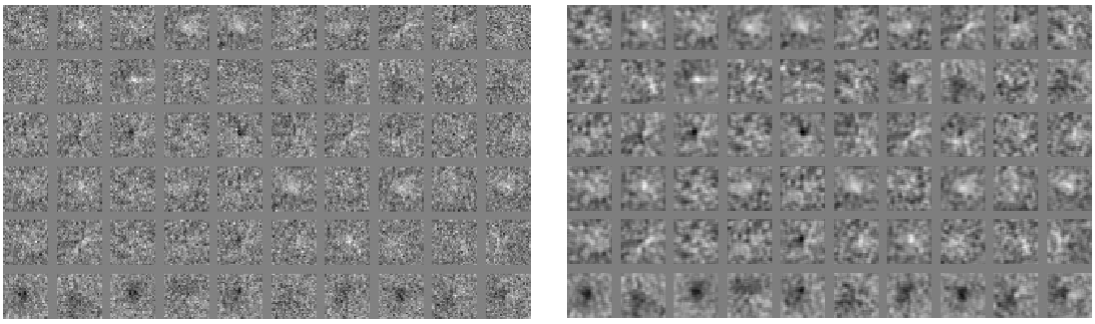
6.6 Relevant Observed Characteristics of Selected Methods

In this section, the best models (the models with the highest validation accuracy) of selected methods are examined, where special characteristics are discussed in depth. Besides the R^2SVM approach, the learned features of the neural network approaches and the simple and complex cells of the K-means deep learning algorithm are visualized and examined.

6.6.1 Characteristics of R^2SVM

The accuracy does not converge within 50 layers.

The test accuracy is plotted against the number of layers of the R^2SVM in Figure 6.11. This graphic shows how the accuracy develops as the number of layers increases, while the other hyper-parameters remain the same. In contrast to the results presented in Section 5.7.1 and in [82], Figure 6.11 does not show a clear convergence of the R^2SVM within the chosen number of layers (50). This illustration suggests, that a higher number of layers than 50 would probably increase the final accuracy of the R^2SVM approach on the PULMO dataset.



(a) (b)

Figure 6.12: This graphic shows the features of the first and second hidden-layer of the trained (stacked sparse auto-encoder) network. In every row, the filters with the highest mean activation (of the corresponding class) are visualized, where the first row corresponds to the class *"Original"*, the second to *"Normal"*, the third to *"Ground Glass"*, the forth to *"Reticular"*, the fifth to *"Honeycombing"* and the last row to the class *"Emphysema"*. (a) Visualization of first hidden-layer features. (b) The filters of the first hidden-layer, where the images have been smoothed with a Gaussian filter (3-by-3, $\sigma = 2$).

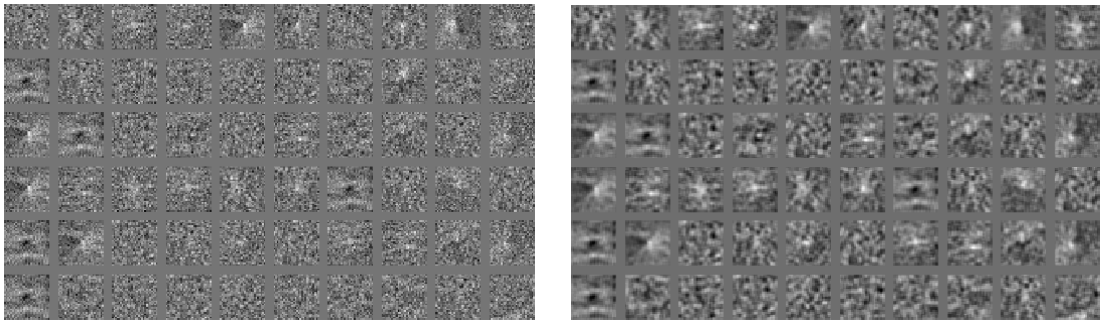
6.6.2 Characteristics of Stacked Sparse Auto-Encoders

The stacked sparse auto-encoder approach learns dataset specific features. The features in the higher layers are more class-specific than the features in the lower layers.

The model with the highest validation accuracy exhibits a structure with two hidden layers and 2000 hidden-units per layer. The features are visualized as described in Section 5.7.2.

In Figure 6.12(a) the filters of the first hidden-layer are visualized, where every row corresponds to the visualized features of one class. In contrast to the filters learned for the MNIST dataset (which can be found in Section 5.7.2), the filters learned by the stacked sparse auto-encoder for the PULMO dataset appear noisy and contain no structures which are easily recognizable for the human eye. These characteristics are obtained for the second hidden-layer filters as well. These results indicate that the stacked sparse auto-encoder approach attempts to learn dataset specific features, although the filters which are learned are not "beautiful" for all datasets. Furthermore, these results show that a neural network that comprises filters which are not "beautiful" is able to achieve a good performance, since the test accuracy of the stacked sparse auto-encoder (63.95%) is the highest one among all evaluated approaches. To illustrate that the first hidden-layer filters are not only random noise, the filters of Figure 6.12(a) are smoothed with a Gaussian filter (with a size of 3-by-3 and $\sigma = 2$) and depicted in Figure 6.12(b).

As for the MNIST dataset, the second layer features are more class-specific than the first layer features, where a discussion of this issue can be found in Appendix B.6.



(a) (b)

Figure 6.13: This graphic shows the features of the first and second hidden-layer of the trained denoising auto-encoder. In every row, the filters with the highest mean activation (of the corresponding class) are visualized, where the first row corresponds to the class *"Original"*, the second to *"Normal"*, the third to *"Ground Glass"*, the fourth to *"Reticular"*, the fifth to *"Honeycombing"* and the last row to the class *"Emphysema"*. (a) Visualization of first hidden-layer features. (b) The filters of the first hidden-layer, where the images have been smoothed with a Gaussian filter (3-by-3, $\sigma = 2$).

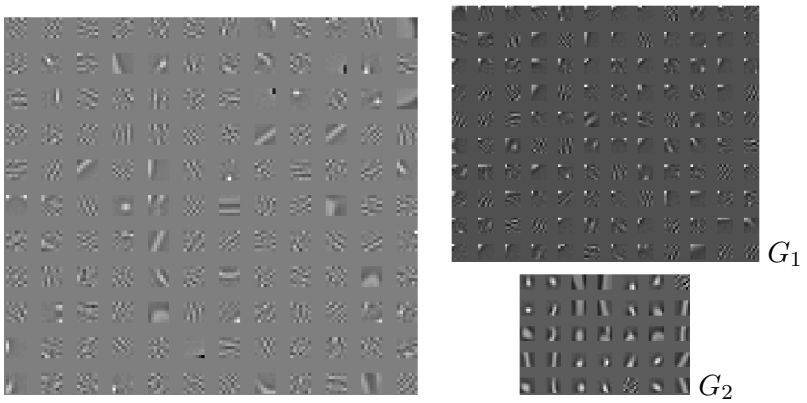
6.6.3 Characteristics of Stacked Denoising Auto-Encoders

The stacked denoising auto-encoder approach learns dataset specific features. Only a combination of the second layer features makes it possible to distinguish between different classes.

The denoising auto-encoder with the best parameter setting exhibits two hidden-layers and 1000 hidden-units per layer. The same visualization and selection procedure as for the sparse auto-encoder is used to choose and illustrate the most interesting filters, which means that every row in Figure 6.13 corresponds to one class.

Figure 6.13(a) shows the filters of the first hidden-layer, where it can be seen that the features which are learned by the stacked denoising auto-encoder approach contain no structures which are easily recognizable for the human eye and appear, according to the sparse auto-encoder filters learned on the PULMO dataset, also noisy. Therefore, a smoothed version of the first layer filters is visualized in Figure 6.13(b). These observations, respectively the difference between the first layer filters in Figure 6.13 and Figure 5.16, indicate that the stacked denoising auto-encoder approach tries to learn dataset specific features automatically (although the filters learned on the PULMO dataset are not as "beautiful" as the filters learned on the MNIST dataset).

A further in depth analysis of the correlation matrices can be found in B.7, where the main finding of this discussion is that only a combination of the second layer features makes it possible to distinguish between different classes.



(a) (b)

Figure 6.14: This figure gives an overview of the learned features in the first layer of the K-means deep learning algorithm. (a) Randomly chosen first layer simple cells. (b) The simple cells of two pooling groups.

6.6.4 Characteristics of K-means Deep Learning Algorithm

The K-means deep learning algorithm learns dataset specific features.

The K-means deep learning algorithm with the highest validation accuracy exhibits 12800 first layer simple cells, 128 first layer complex cells, 6400 second layer simple cells and 128 second layer complex cells. A random subset of the first layer simple cells is shown in Figure 6.14(a), where it can be seen that this random subset contains edge-like features which are learned for the MNIST dataset, too. On the other hand, this random subset also contains filters which look noisy and do not occur within the features learned on the MNIST dataset. This indicates that the K-means deep learning algorithm learns dataset specific features. Figure 6.14(b) shows two groups of simple cells which belong to two different complex cell pooling groups. While the filters of the group G_1 contain a bright area in the upper left corner, G_2 comprises edge-like filters which are oriented vertically. Beside these characteristics, both groups contain noisy-looking filters as well.

It can be seen that the correlation matrix illustrated in Figure 6.15, shows a generally low correlation of the features and classes (in comparison to the correlation matrix in Figure 5.19). Furthermore, the pattern of the rows ”1”, ”2” and ”6” (respectively of the rows ”3” and ”5”) are similar, which means that the K-means deep learning algorithm does not learn class-specific features. Despite these characteristics, the K-means deep learning algorithm, in contrast to the results for the MNIST dataset, helps to improve the performance of the extremely randomized trees on the PULMO dataset. Possible reasons are the usage of more unlabeled patches or a lower accuracy of the extremely randomized tree approach (without K-means) (31.72%) in comparison to the accuracy on the MNIST dataset (90.31%). This indicates that the classification task is more difficult for the PULMO dataset if the original features are used as input to train the extremely randomized trees. This in turn means that even features which exhibit a weak

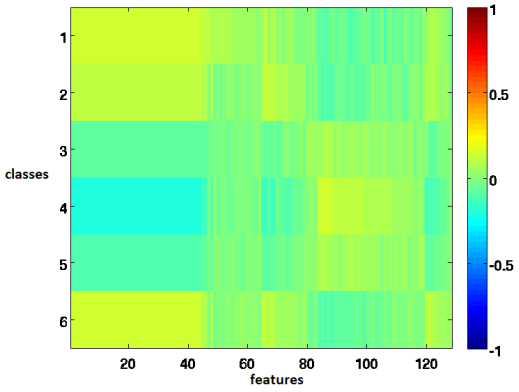


Figure 6.15: The correlation matrix of the K-means deep learning algorithm. This matrix denotes the correlation between the classes of the PULMO dataset and the output of the model trained by the K-means deep learning algorithm.

correlation help to improve the performance.

6.7 Influence of Supervised Training Set Size

In this section, the effect of changing the number of used labeled examples is examined, while the number of unlabeled examples remains the same. The following values are chosen to determine the size of the used labeled training subset:

$$\{8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16000\} \tag{6.6}$$

where the values indicate the number of examples used per class. For each method, the model is trained with the best parameter setting found in the experiments of Section 6.5. In Figure 6.16, the influence of the number of labeled examples used on the test accuracy is illustrated for all methods, where each color highlights the development of the test accuracy for one method. It can be seen that the performance of the K-nearest neighbor approach worsens as the number of labeled examples is increased (a test accuracy of 15.3% is obtained if 16000 examples are used for each class). At the same time, the K-nearest neighbor approach does not exhibit a good test accuracy when the number of labeled examples is low (16.5% if 8 examples are used for each class), which means that the K-Nearest neighbor approach is the worst method in terms of accuracy on the PULMO dataset. Though the performance of the extremely randomized tree approach does improve if the number of labeled examples is increased (from 24.4% to 31.3%), it is not affected much by the number of examples in comparison to the SVMs, the R^2SVMs and the neural network approaches, as illustrated in Figure 6.16. On the one hand, this can be advantageous if the number of examples is low, since the extremely randomized trees outperform all other approaches except the K-means deep learning algorithm if 512 or fewer labeled examples per class are used. On the other hand, this can be seen as a disability to make use of an increased number of labeled examples. The K-means deep learning algorithm improves

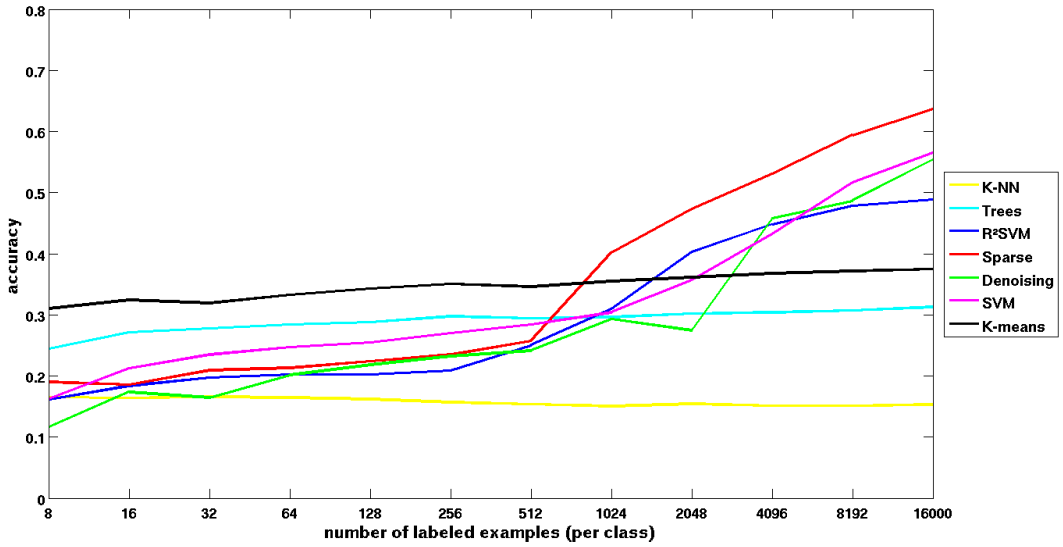


Figure 6.16: This graphic illustrates the correlation between the test accuracy and the number of labeled examples (which are used to train the model) for all methods.

the test accuracy of the extremely randomized tree approach by about 6% (independently of the number of labeled examples) on the PULMO dataset. This leads to the fact, that the K-means deep learning algorithm outperforms all other approaches, if 512 or fewer labeled examples per class are used. The sparse auto-encoder makes the most of the increased number of labeled examples and exhibits the highest increase among these methods.

The influence of the number of labeled examples on the runtime is illustrated for all methods in Figure 6.17. The training time of the SVM increases significantly, if 4096 or more labeled examples are used. The training time of the SVM is about 240 hours for 16000 examples per class. This value is not illustrated in Figure 6.17, since this would lead to an unclear illustration of the runtime of the other approaches. The training time of the extremely randomized tree approach increases significantly, if 8192 or more labeled examples per class are used for the training of the tree. As for the MNIST dataset, the sparse and the denoising auto-encoder, show no clear influence of the number of labeled examples on the training time. For the R^2SVM and the K-means deep learning algorithm, the influence of the number of labeled training examples on the training time is much lower than for the other evaluated approaches, as can be seen in Figure 6.17.

Finally, it can be stated that the number of support vectors is increased as the number of labeled examples is increased, which coincides with the result presented in Section 5.8. More details on this topic are provided in Appendix B.8.

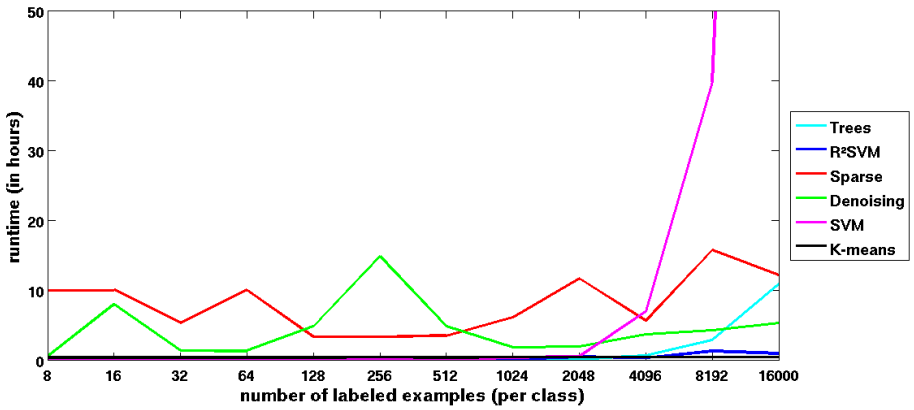


Figure 6.17: This graphic illustrates the correlation between the training time and the number of labeled examples (which are used to train the model) for all methods.

6.8 Summary

In this section, the main results of this chapter are summarized. While the stacked sparse auto-encoder approach achieves the highest test accuracy (63.95%) among all evaluated methods, the K-nearest neighbor approach completely fails to solve the classification task on the PULMO dataset. Furthermore, the SVM clearly exhibits the longest runtime, but at the same time achieves the second highest accuracy among all evaluated approaches. The K-means deep learning algorithm helps to improve the performance of the extremely randomized tree approach on the PULMO dataset.

The performance of the R^2SVM approach shows a weaker dependence on the hyper-parameters than the SVMs. The *learning rate* shows the highest influence on the accuracy, which is both true for the sparse and the denoising auto-encoder approach. For the K-means deep learning algorithm, τ of the first layer exhibits the strongest influence on the accuracy.

The accuracy of the R^2SVM approach shows no clear convergence within the chosen number of layers (50). Though the learned features of both auto-encoder approaches are noisy, both auto-encoders attempt to learn dataset-specific features. Furthermore, the results show that higher layer features of the sparse auto-encoder exhibit a higher correlation and are more class-specific than the features of lower layers.

A higher number of labeled training examples increases the accuracy of all evaluated methods with the exception of the K-nearest neighbor approach. Moreover, a larger training set size increases the runtime of all methods except both auto-encoder approaches and the K-nearest neighbor method (because this approach exhibits no explicit training phase).

The results of the SVM approach show that γ exhibits the strongest influence on the performance. Concerning the K-means deep learning algorithm, τ of the first layer exhibits the strongest influence on the accuracy among all hyper-parameters of this method. The *activation regularization coefficient* of the sparse auto-encoder exhibits the strongest influence on the accuracy among all hyper-parameters. In contrast, for the denoising auto-encoder, the *learning rate*

shows the strongest influence on the accuracy. Moreover, the results indicate that the denoising auto-encoder approach is more stable than the sparse auto-encoder approach.

CHAPTER 7

Summary and Discussion of the Results

In this chapter, the major results of Chapter 5 and Chapter 6 are summarized and compared with each other. Subsequently, practical recommendations for the methods (and their hyper-parameters), which are evaluated in this work, are given.

7.1 Main Results and Comparison

If the results in relation to accuracy of Section 5.4 and Section 6.3 are compared with each other, it can be seen that the classification task on the PULMO dataset is more difficult than the classification task on the MNIST dataset. In the following, the similarities and differences between the results on the PULMO and the MNIST dataset are highlighted.

The results of the K-nearest neighbor approach in terms of accuracy are contrary: while this method achieves a good performance on the MNIST dataset (96.92%), the performance is very bad on the PULMO dataset (15.33%). Furthermore, on the MNIST dataset the number of labeled training examples does affect the obtained test accuracy, whereby on the PULMO dataset no such correlation can be identified. Beside these contrary observations, the hyper-parameter k of the K-nearest neighbor shows no (PULMO), respectively only a slight (MNIST), influence on the accuracy.

For the extremely randomized tree approach, the second-worst test accuracy among all evaluated approaches is obtained on both datasets. Additional similarities are that the accuracy converges within 128 trees on the MNIST dataset as well as on the PULMO dataset and that the training runtime of the extremely randomized tree approach increases linearly with the chosen number of trees. Moreover, a higher number of labeled training examples tends to increase the runtime as well as the accuracy of the extremely randomized trees, where the influence of the number of examples on the accuracy is stronger on the MNIST dataset.

The SVMs exhibit the second-best test accuracy among all evaluated approaches on both datasets. Beside this general observation, it can be stated that both γ and C can have a strong influence on the performance of the SVMs: While for the MNIST dataset, γ exhibits a stronger

influence than C on the accuracy, the runtime and the number of support vectors, this is not true for the PULMO dataset. In terms of accuracy, γ and C are equally important on the PULMO dataset. In relation to the training time and the number of support vectors, C shows a stronger influence than γ on the PULMO dataset. A higher number of labeled training examples tends to increase the training time, the accuracy and the number of support vectors on both datasets.

In terms of accuracy, the R^2SVM approach is neither one of the best nor one of the worst approaches on both datasets. While the accuracy converges within the chosen number of layers (50) on the MNIST dataset, the results presented in Section 6.6.1 indicate that a higher number of layers than 50 would probably increase the accuracy on the PULMO dataset (since the accuracy does not converge within the chosen number of layers). The results on the MNIST dataset indicate that the two hyper-parameters C and β are equally important in relation to accuracy. In contrast, on the PULMO dataset, a clear influence on the accuracy can only be identified for β. The hyper-parameters of the R^2SVMs show no influence on the runtime on both datasets. A higher number of labeled training examples leads to an increased accuracy and to a longer runtime on both datasets.

The stacked sparse auto-encoder approach exhibits the highest test accuracy on the PULMO dataset and the third-highest test accuracy on the MNIST dataset. The results on both datasets show that the most important parameters in terms of accuracy are the *activation regularization coefficient* and the *learning rate*. Furthermore it can be stated, that a higher number of hidden-units as well as a higher number of hidden-layers tends to increase the runtime of the training (and of prediction). Moreover, a higher number of labeled training examples does not affect the training time (if trained on GPU) and leads to an increased accuracy on both datasets. The stacked sparse auto-encoder approach tends to learn dataset-specific features (as can be seen in Section 5.7.2 and Section 6.6.2), where the features learned on the PULMO dataset appear noisy. Furthermore, the features of higher layers exhibit a higher correlation and are more class-specific than the lower layer features.

The stacked denoising auto-encoder approach exhibits the highest test accuracy on the MNIST dataset and the third-highest test accuracy on the PULMO dataset (which is the reverse to the sparse auto-encoder). On both datasets, the learning rate shows the highest influence on the accuracy. In relation to runtime, both a higher number of hidden-units and a higher number of hidden-layers leads to an increased runtime. The standard deviation of the validation-accuracies indicate that the denoising auto-encoder is a more stable approach than the sparse auto-encoder. In other words, the influence of the hyper-parameters on the accuracy is weaker than for the sparse auto-encoder. A higher number of labeled training examples does not affect the training time (if trained on GPU) and leads to an increased accuracy on both datasets. As for the sparse auto-encoder, the stacked denoising auto-encoder approach tends to learn dataset specific features (as can be seen in Section 5.7.3 and Section 6.6.3) and learns noisy-looking features for the PULMO dataset. The features of higher layers exhibit a higher correlation and are more class-specific than the lower layer features. A behavior which can only be identified on the MNIST dataset is that the learned features are more generic (less class-specific) than the features of the sparse auto-encoder.

Finally, the K-means deep learning algorithm exhibits the worst test accuracy on the MNIST dataset and the fifth-best (or third-worst) test accuracy on the PULMO dataset. The major dif-

ference concerning the accuracy is that the K-means deep learning algorithm helps to improve the accuracy of the extremely randomized trees on the PULMO dataset, but not on the MNIST dataset (here, the accuracy decreases). The results of both datasets indicate that the hyper-parameter τ of the first layer is the most important parameter of this method, since τ of the first layer exhibits the strongest influence on the accuracy, the runtime, the number of group members in the first layer and the number of group members in the second layer (on both datasets). Comparing the learned first layer pooling groups of the MNIST and the PULMO dataset, it can be seen that the first layer pooling groups learned on the MNIST dataset contain edge-like simple cell features, while the first layer pooling groups learned on the PULMO dataset contain noisy-looking filters too. Since the K-means deep learning algorithm is trained purely unsupervised, the model learned by the K-means deep learning algorithm is not affected by the number of labeled training examples. Only the subsequent extremely randomized tree approach is affected by the number of labeled training examples, which means that if the K-means pipeline illustrated in Figure 5.3 is evaluated as a single method, the same dependencies as for the extremely randomized tree approach can be identified.

7.2 Recommendations

Based on the results presented in Chapter 5 and Chapter 6, recommendations concerning the methods evaluated in these sections are given in the following. First, the method by which the classification task is solved has to be chosen. If the computational resources are limited and the runtime is an important issue, we recommend to use the R^2SVM approach, since the results in Section 5.5 and Section 6.4 show that the R^2SVMs exhibit the shortest training time among all evaluated approaches. The only exception is the K-nearest neighbor approach, which has no explicit training phase. But because the K-nearest neighbor approach completely fails to solve the classification problem on the PULMO dataset, it is recommended to use this method only in order to figure out how complex a classification task is. Though the R^2SVMs achieve a poorer accuracy than the SVMs and the neural network approaches, it is at the same time not one of the worst approaches on both datasets.

If the computational cost is not as important as the accuracy, it is recommended to use the SVMs, the sparse auto-encoder or the denoising auto-encoder, since these approaches achieved the highest accuracy on both datasets. If the number of unlabeled training examples is high in comparison to the number of labeled training examples (e.g. 8 labeled training examples per class and 96000 unlabeled training examples) and the performance of these approaches is not satisfying, it may be a good idea to use the K-means deep learning approach, where this recommendation is motivated by Figure 6.16. On the other hand, if the number of labeled training examples is 96000 or higher, it is recommended to use either the stacked sparse auto-encoder or the stacked denoising auto-encoder approach (and not the SVM approach), independent of the number of unlabeled training examples. This can be explained by the fact that on both datasets, one neural network approach achieved a higher accuracy than the SVM and that the training runtime of the SVM is very high (240 hours) if 96000 labeled training examples are used (as described in Section 6.7), while the neural network approaches show no clear influence between the number of labeled training examples and the runtime. It has to be kept in mind that this is

true if the neural network approaches are computed on a GPU.

After the selection of the method, the hyper-parameter settings of the chosen method have to be determined, which is why method-specific recommendations are given in the following. For the extremely randomized trees it is recommended to use 100 trees, since convergence of the accuracy is observed on both datasets for this value. The results of the SVM indicate that the examined range of C is a reasonable choice. If computational resources are limited, it is recommended to use values between 0.01 and 100 first. In contrast, we advise to use 0.01 or lower values for γ, since the results show that the best performance is achieved with values in this range.

For the R^2SVM approach it is recommended to use 0.1 or higher values for β (even higher than 1) and values between 0.01 and 100 for C (as for the SVM approach). Since the results presented in Section 6.6.1 suggest that a higher number of layers than 50 would probably increase the final accuracy of the R^2SVM approach on the PULMO dataset, we recommend to use more than 50 layers in future experiments.

For the sparse and the denoising auto-encoder, the results suggest to use the highest *learning rate* which causes no divergence of the training procedure, which coincides with [9]. Moreover, we recommend to use a *momentum* within the training procedure, because the usage of a momentum tends to increase the accuracy, as described in Section 5.6.6, Section 6.5.5 and Section 6.5.6. For both neural network approaches, a higher number of *hidden-layers* tends to increase or at least does not affect the accuracy. Therefore it seems advantageous to use 4 *hidden-layers* instead of 3 or 2. In relation to hidden-units, the results of this work coincide with the results presented in [9], which means that it is most important to choose the number of hidden-units large enough (larger than the input dimension), where a higher number than the optimum does not affect the accuracy much. Since the weight decay regularization coefficient shows no influence, neither on the accuracy nor on the runtime, the tuning of this parameter has low priority. Beside these observations and recommendations, which are both true for the sparse and the denoising auto-encoder, there are also method-specific recommendations. For the *activation regularization coefficient* of the stacked sparse auto-encoder, the results indicate that 0.003 is a good starting value, where values above 0.1 should be avoided. For the stacked denoising auto-encoder, the results on the PULMO dataset indicate that 0.25 is a good starting point for the *corruption parameter* ν.

For the K-means deep learning algorithm it is most important to find the optimal value for τ of the first layer, since this parameter shows the strongest influence on the accuracy of the training time among all examined hyper-parameters. Beside this observation, we avoid to give further recommendations, because the results on the MNIST and the PULMO dataset are contrasting.

CHAPTER 8

Conclusion and Future Work

Beside a general introduction into machine learning and deep learning, seven different methods have been described, implemented and evaluated on a conventional (MNIST) and on a medical (PULMO) dataset. Three of these methods (K-nearest neighbors, SVMs, extremely randomized trees) can be categorized as conventional approaches, whereas the other four approaches (stacked sparse auto-encoders, stacked denoising auto-encoders R^2SVM, K-means deep learning algorithm) can be characterized as deep learning methods. In the following, the major findings of this work are summarized and areas of further investigations are indicated.

8.1 General Performance

With respect to the general performance of the methods, it can be stated that the stacked sparse auto-encoder, the stacked denoising auto-encoder and the SVM achieved the highest accuracy among all evaluated approaches on both datasets. At the same time, these three methods exhibit the highest training time among all evaluated approaches on both datasets. Therefore these methods are preferable if the available computational resources allow to use them. In contrast, the K-nearest neighbor approach and the R^2SVMs exhibit the shortest training time on both datasets, but achieve a poorer accuracy than the aforementioned approaches. Since the results suggest that the K-nearest neighbor algorithm fails to solve more complicated classification tasks, the R^2SVMs are preferrable in addressing medical classification tasks if the computational cost is the most important factor.

Although the K-means deep learning algorithm helps to improve the performance of the extremely randomized trees on the PULMO dataset, it is not clear if this is also true for other medical datasets, in particular because the K-means deep learning algorithm worsens the generalization performance of the extremely randomized trees on the MNIST dataset.

Besides these findings concerning accuracy and runtime, the results suggest that the K-means deep learning algorithm, the stacked sparse auto-encoder and the stacked denoising auto-encoder are all successful in learning problem-specific features.

8.2 Hyper-Parameters

Among the R^{2}SVM hyper-parameters, the results indicate that β is more important than C and that the number of layers should be chosen high enough to ensure convergence of the accuracy. For the sparse auto-encoder, the *activation regularization parameter* and the *learning rate* show the highest influence on the performance. For the denoising auto-encoder, the results indicate that the *learning rate* is the most important hyper-parameter. The results of the K-means deep learning algorithm show that the hyper-parameter which exhibits the highest influence on the accuracy and the runtime, is τ of the first layer.

8.3 Further Issues of Research

Since the results of the K-means deep learning algorithm on the MNIST and the PULMO dataset are contrasting, further in-depth analyses of the hyper-parameters would help to understand the behavior and the characteristics of this method. This should not only be done with other datasets, but also with the dataset used in this work, in order to find out what causes the worsening of the generalization performance of the extremely randomized trees on the MNIST dataset. In particular, the influence of the number of unlabeled training examples on the performance should be evaluated in future work. For the R^{2}SVM approach, a higher number of layers should be evaluated on the PULMO dataset, for the purpose of finding out if this helps to improve the accuracy. Though this work contains an extensive evaluation of the hyper-parameters, future work should also focus on examining further ranges of the hyper-parameters and the influence of the number of considered hyper-parameters on the accuracy. This will help to stress the importance of the hyper-parameters and is of particular interest for the neural network approaches, since they exhibit more hyper-parameters and a longer training time than the other methods. Finally, the methods which are evaluated in this thesis should be evaluated on further medical datasets in order to qualify, respectively confirm, the results, statements and recommendations of this work.

APPENDIX A

Additional MNIST Results

A.1 Selection of Hyper-Parameter Intervals

As discussed in Section 5.3, the ratio between different values is a better indicator of the impact of the hyper-parameters than the absolute value. For instance, if we have two neural networks with the learning rates 0.1 and 0.11, the results obtained are likely to be very similar. Further, if two neural networks with the learning rates 0.01 and 0.02 are compared, the results are probably quite different, though the absolute difference is the same in both cases [9]. In contrast, if the ratios $\frac{0.1}{0.11} = 0.91$ and $\frac{0.01}{0.02} = 0.5$ between the values are considered, it can be seen that the ratio between different values is in this case a better indicator of the impact of the hyper-parameters than the absolute value [9]. In other words, in this work the values for one hyper-parameter are chosen in such a way that the ratio between two "neighbors" is always the same.

A.2 Parameter-Selection of the Stacked Sparse Auto-Encoder

In the following, the settings of the neural network which are fixed are explained. Subsequently, the settings for the hyper-parameters, which are examined using multiple values, are discussed. A summary of the settings is given in Table 5.2.

The training of the stacked sparse auto-encoder approach, which is described in this section, consists of an unsupervised greedy layer-wise pre-training stage (with sparse auto-encoders) and subsequent fine-tuning of the whole network. For reasons described in Section 4.5.1, the activation and error functions are not varied within the experiments. As explained in Section 3.2.10, the *hyperbolic tangent* function is used as the activation function for the neurons of the neural network. The error function of the pre-training stage consists of three terms: the sum-of-squares error function, L2-regularization of the weights and L1-regularization of the unit-activations. For the subsequent fine-tuning stage, the softmax activation function with the corresponding error function (described in Section 3.2.4) are used in the last layer, since the MNIST dataset is a classification problem. To avoid unnecessary update steps and to save computation time,

the maximum number of epochs per layer (respectively in the fine-tuning stage) is fixed at 1000. Additionally the training is stopped if the objective of the error function does not improve within ten epochs by a factor of 0.1%[1]. The mini-batch size of the training (mini-batch gradient descent is used to train the network) is set to 100, because Bengio et al. [9] recommend to use a mini-batch size between 10 and several hundred (Hinton [38] recommends to use a size between 10 and 100). The last setting which is not varied within the evaluation process of this work is the initialization of the weights which is done as described in Section 4.5.1.

Beside the fixed settings, there are six different hyper-parameters where multiple values are examined. First, the basic architecture has to be determined: the number of layers and the number of units per layer. For the MNIST dataset, three different depths are chosen for the stacked sparse auto-encoder approach ($\{2, 3, 4\}$). As recommended in [9] and discussed in Section 3.2.5, the number of hidden-units per layer is chosen higher than the number of input-units and is constant across all hidden-layers. Because we want to examine the influence of the number of hidden-units per layer on the performance, two different values are selected ($\{1000, 2000\}$). The impact on the results of using the momentum or not is also evaluated. Bengio et al. [9] recommend using a learning rate between 1 and 10^{-6}, so the following three different values are used: $\{10^{-5}, 10^{-4}, 10^{-3}\}$. Since two regularization terms are used in the cost function of the pre-training stage, the *weight decay regularization coefficient* and the *activation regularization coefficient* have to be determined. For both the values $\{0.003, 0.1, 3\}$ are used.

A.3 Parameter-Selection of the K-means Deep Learning Algorithm

The *number of layers* in total is chosen to be four, two of each type. Further we have decided to use a *patch-size* of 7-by-7, since the MNIST dataset contains 28-by-28 images. Since it is not sure whether reversed polarity matters in our experiments, we have decided to use the activation function $g(a) = a$. The number of simple cells in the first layer is set to 12800, because Coates et al. [22] state, that only the usage of the activation function $g(a) = |a|$ allows to train half as many simple cell features for the first layer (6400). Though in [22] the number of simple cells in the second layer of simple cells is chosen to be 150000, we use much fewer cells ($\{6400, 12800\}$), because our computational resources are limited and the problem setting given in [22] is more complex. While Coates et al. [22] try to learn features which are able to identify an object class (from images which are containing partial views of objects or clutter at differing scales[2]), the images of the MNIST dataset contain full views of centered digits. For the same reason the number of complex cells is chosen to be $\{128, 256\}$ in the first layer, and $\{128, 256\}$ in the second layer. The corresponding parameters of the complex cell training procedure are set relative to the number of complex cells: the *initial number of groups* is defined as "ten times the number of complex cells" and the *size of the largest-groups-subset* as "two times the number of complex cells".

[1] For instance if the objective has a value of 67.0 after the 30th epoch and a value of 66.90 after the 40th epoch, training is continued, since the objective improved by a factor of more than 0.001% ($66.90 < 67 - (67 * 0.001) = 66.933$).

[2] Further details of the experiments can be found in [22].

The number of iterations of centroid-updates within the simple-cell training procedure is chosen to be 10 for all layers, as recommended in [23]. Since a different number of simple and complex cells is used in the experiments of this work in comparison to the architecture defined in [22], the parameters τ and Δ have to be reconsidered. As discussed in Section 3.4.4, Coates et al. [22] used $\tau = 0.3$ for the first and $\tau = 1.0$ for the second layer of complex cells. Here, $\tau = \{0.5, 0.7, 0.9\}$ is used for the first layer of complex cells, because a test run with $\tau = 0.3$ (as used in [22]) yields groups with very few simple cells in the first layer[3]. For the second layer of complex cells $\tau = \{1.0, 1.2\}$ is examined, since no clear recommendation is provided in [22]. Finally, $\Delta = 1.5$ is used for both layers, since the explanation in [22] for using this value is also valid for the problem setting in this work. A summary of the settings for the K-means deep learning algorithm is given in Table 5.3.

A.4 Programming Languages

Table A.1 gives an overview of the programming languages, in which the evaluated methods are implemented.

Method	**Implementation Language**
K-Nearest Neighbor	MATLAB
Extremely Randomized Trees	C
SVM	C/C++
R^2SVM	MATLAB, C/C++
Stacked Sparse Auto-Encoder	Python, C
Stacked Denoising Auto-Encoder	Python, C
K-means Deep Learning Algorithm	MATLAB

Table A.1: This table gives an overview of the main programming languages in which every method is implemented.

A.5 Details of the Box-Plot

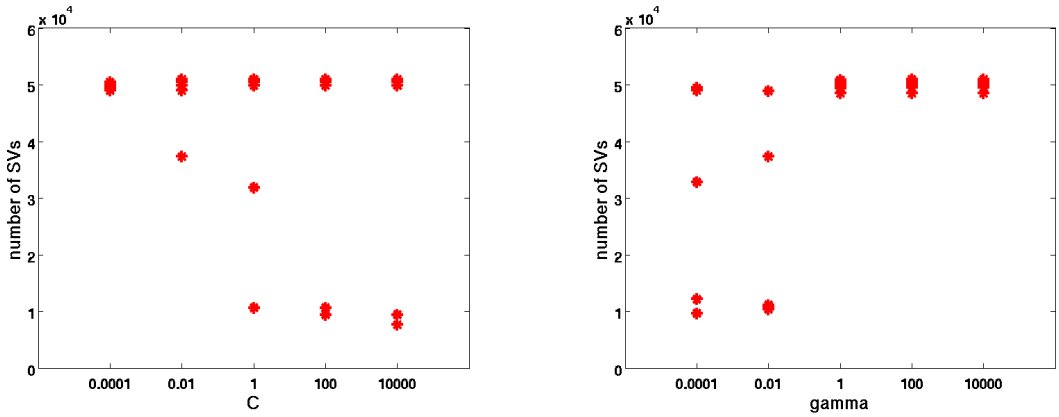
On each box, a red line denotes the median and the edges of the box are the 0.25- and 0.75-percentiles [58]. The median divides the data-values into two parts: one half of the data values are at most as high as the median and the other half is at least as high as the median [84]. Each percentile divides the data also into two parts: 25% of the data values are at most as high as the 0.25-percentile and 75% are at least as high as the 0.25-percentile (accordingly 75% of the data values are at most as high and 25% of the data values are at least as high as the 0.75-percentile) [84]. The so-called whiskers, drawn in black, include the most extreme data values

[3]The average number of simple cells was lower than 5 members per group in the first layer.

which are not recognized as outliers, where outliers are highlighted as red crosses. Points are classified as outliers, if they are larger than $q_3 + 1.5(q_3 - q_1)$ or smaller than $q_1 - 1.5(q_3 - q_1)$, where q_1 and q_3 are the 0.25- and 0.75-percentiles [58]. If the data are normally distributed, the whiskers include approximately 99.3% (+/–2.7σ) of all data [58, 84].

A.6 SVMs: Influence of Hyper-Parameters on the Number of Support Vectors

Figure A.1(a) illustrates the influence of C on the number of support vectors in the trained SVM. It can be seen that a higher C leads to a model with fewer support vectors. This is reasonable, since a higher C penalizes misclassified training examples stronger and therefore more likely leads to a narrower margin and a more complicated model. This in turn decreases the number of support vectors, because fewer training examples are misclassified (misclassified examples are used as support vectors, as discussed in Section 2.3.3). The influence of γ on the number of support vectors is shown in Figure A.1(b), where a higher γ tends to increase the number of support vectors in the final SVM. This can be explained by the fact, that a higher γ leads to a complicated margin and therefore to an increased number of support vectors in the final SVM. More precisely, a training with $\gamma \geq 1$ leads to SVMs which use nearly all training examples as support vectors, and therefore to SVMs with a bad generalization performance due to overfitting. As mentioned in Section 5.5, eleven minutes is the mean prediction runtime of SVMs for the MNIST dataset, where this can be explained by the high mean number of support vectors (40802) of SVMs. If Figure A.1 and Figure **??** are compared, it can be seen that an SVM model with a lower number of support vectors tends to have a higher validation accuracy.



(a) (b)

Figure A.1: Two figures which illustrate the relation between hyper-parameters and the number of support vectors in the final SVM. (a) The hyper-parameter C is plotted against the number of support vectors in the final SVM. (b) A plot that illustrates the influence of γ on the number of support vectors.

A.7 K-means Deep Learning Algorithm: Number of Members in a Complex Cell

The influence of τ of the first layer on the mean number of simple cells in one complex cell pooling group in the first layer is illustrated in Figure A.2(a). It can be seen that a higher τ leads to an increased number of simple cells in one group, because a higher τ leads to an increased number of simple cells that are added in one iteration of the complex cell growing procedure (which is described in Section 3.4.2). The number of group members in the second layer is mainly influenced by τ of the first layer, where this relation is illustrated in Figure A.2(b). A higher τ in the first layer tends to decrease the number of simple cells in one pooling group of the second layer. Since a higher τ in the first layer leads to complex cells which exhibit more group members and are therefore more invariant in comparison to complex cells built with a lower value of τ. This in turn may influence the second layer simple cells which are trained based on the first layer complex cell outputs. In Figure A.2(c) the *number of simple cells in the second layer* is plotted against the number of group members in the second layer. An increased *number of simple cells in the second layer* tends to increase the number of simple cells in one pooling group of the second layer. Figure A.2(d) illustrates the influence of τ of the second layer on the number of group members in the second layer. If τ is chosen to be higher, the number of group members increases, though the influence is not as strong as the influence of τ of the first layer. Again, this relation is plausible, since τ controls the number of simple cells which are added in one iteration of the complex cell growing procedure. Among the other hyper-parameters, no clear influence on the number of simple cells in pooling groups can be identified.

A.8 Stacked Sparse Auto-Encoder: Feature Visualization using Input-Images

In Figure A.3, each unit is represented as a weighted average of the input images, where the weighting factor is the activation of the unit. For the visualization of one unit, not all input images are used to create the final feature-representation. Instead, only the first 1000 images with the highest activation are used for the visualization. We decided to use the 1000 "highest-activation-input-images", since the MNIST test-set contains about 1000 images of each class. A perfect predictor for one specific class would output the highest activation for the input-images of this class. These images (all of the same class) in turn would then be used by the visualization procedure, which would highlight the class-specific property of the perfect predictor. Therefore this visualization-variant is a good indicator as to whether the features are class specific or not. Figure A.3(a) shows the features of the first hidden-layer, while the features of the third hidden-layer are visualized in Figure A.3(b). These feature visualizations indicate, that higher-layer features are more class-specific than lower-layer features.

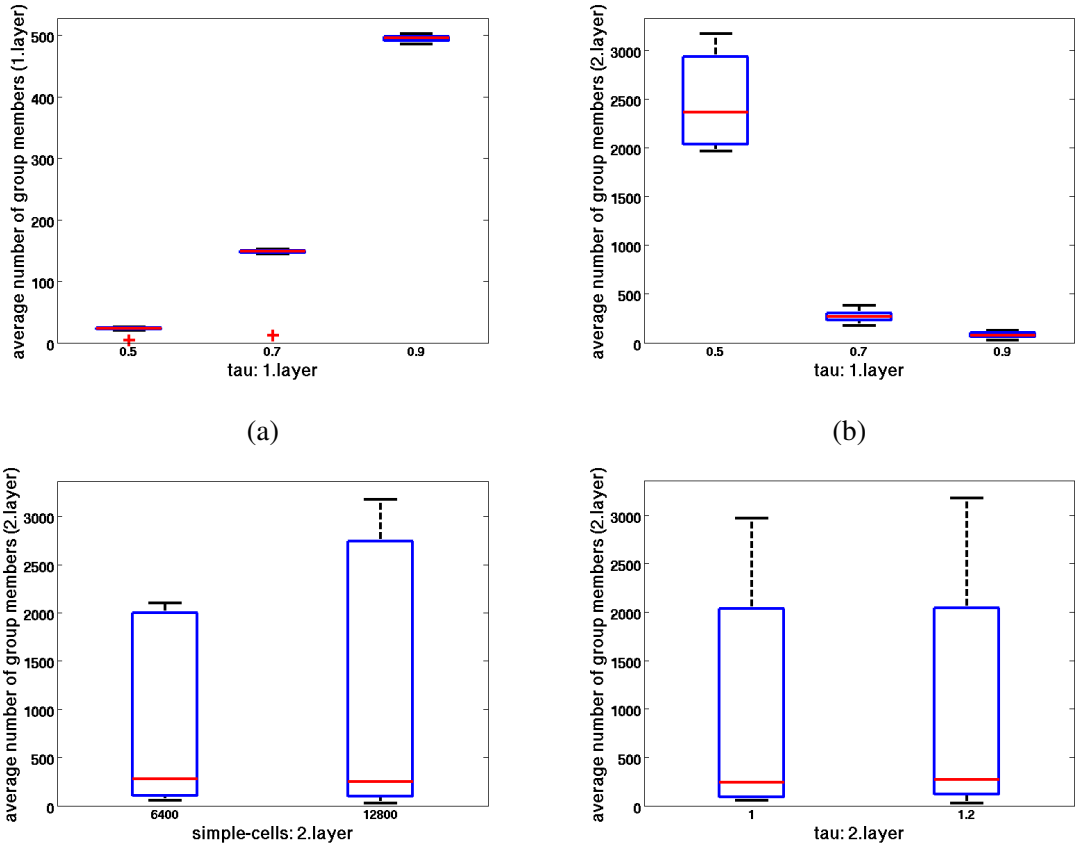
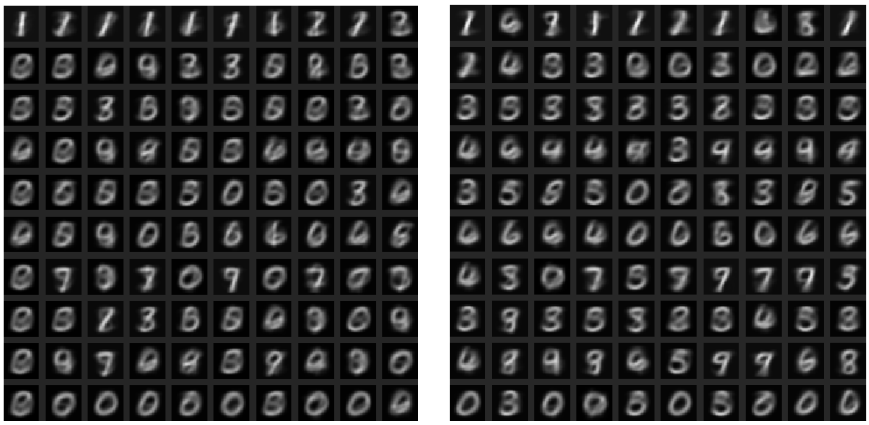


Figure A.2: This figure illustrates the influence of K-means deep learning algorithm hyper-parameters on the mean number of simple cells in one complex cell pooling group. (a) τ of the first layer plotted against the number of group members in the first layer. (b) τ of the first layer plotted against the number of group members in the second layer. (c) The influence of the number of simple cells in the second layer on the number of group members in the second layer. (d) This graphic shows the influence of τ of the second layer on the number of group members in the second layer.

A.9 SVM: Influence of Supervised Training Set Size on the Number of Support Vectors

Figure A.4 illustrates the influence of the number of labeled examples on the number of support vectors in the SVM, where the distances between the marks on the x-axis are constant and do not reflect the exponential increase of the number of labeled examples, in order to make the graphic clearer. It can be seen that the number of support vectors increases, as the number of training examples is increased, where a linear correlation can be identified. For instance, if the number of



(a) (b)

Figure A.3: This graphic shows the features of the first and third hidden-layer of the trained (stacked sparse auto-encoder) network. In every row, the filters with the highest mean activation (of the corresponding class) are visualized, where the first row corresponds to the class ”1” and the last row to the class ”0”. The features are visualized as weighted average image of the input-images. (a) The weighted average image of first hidden-layer features. (b) The visualization of the third hidden-layer.

training examples per class is doubled from 256 to 512, the number of support vectors increases from 1371 to 2218.

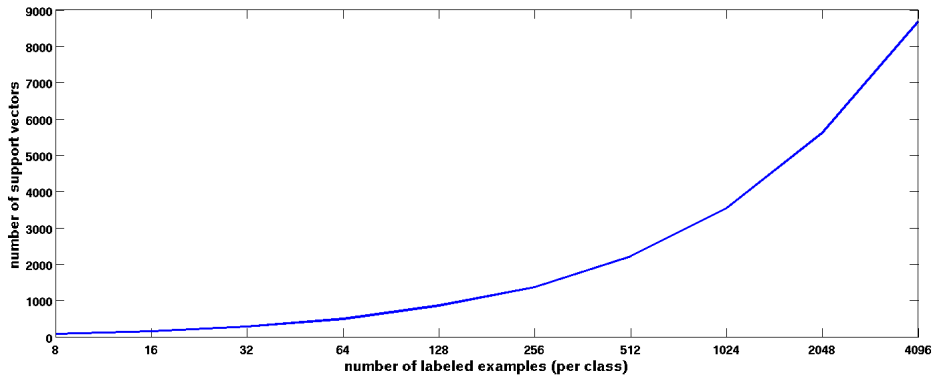


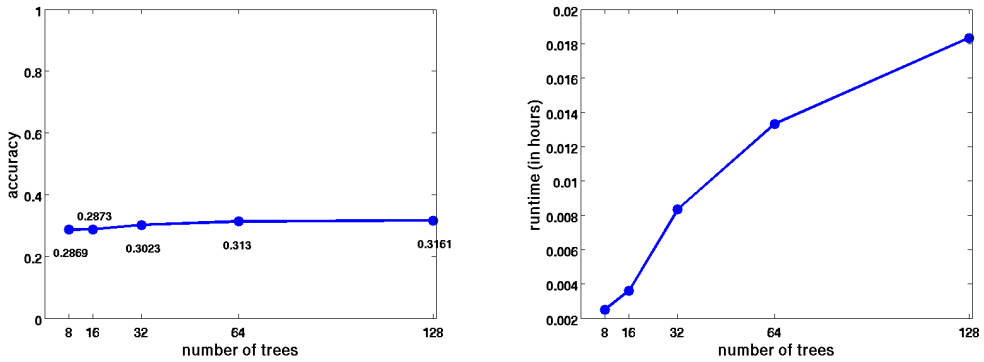
Figure A.4: This figure shows the impact of changing the number of labeled examples (of the MNIST training set) on the number of support vectors in the SVM.

APPENDIX B

Additional PULMO Results

B.1 Extremely Randomized Trees: Influence of the number of trees

Figure B.1 illustrates the influence of the extremely randomized tree hyper-parameters.

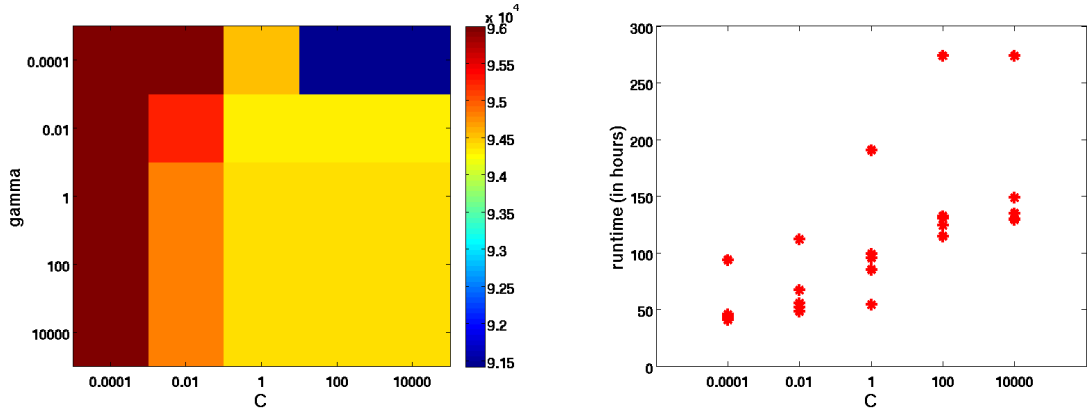


(a) (b)

Figure B.1: This figure illustrates the influence of the number of trees of the extremely randomized tree approach on the (a) validation accuracy and the (b) prediction runtime.

B.2 SVM: Influence of Hyper-Parameters

Figure B.2 illustrates the influence of the SVM hyper-parameters.



(a) (b)

Figure B.2: (c) This graphic illustrates the influence of the SVM hyper-parameters C and γ on the number of support vectors. In (d), C is plotted against the runtime, to clarify the correlation between C and the time needed for training.

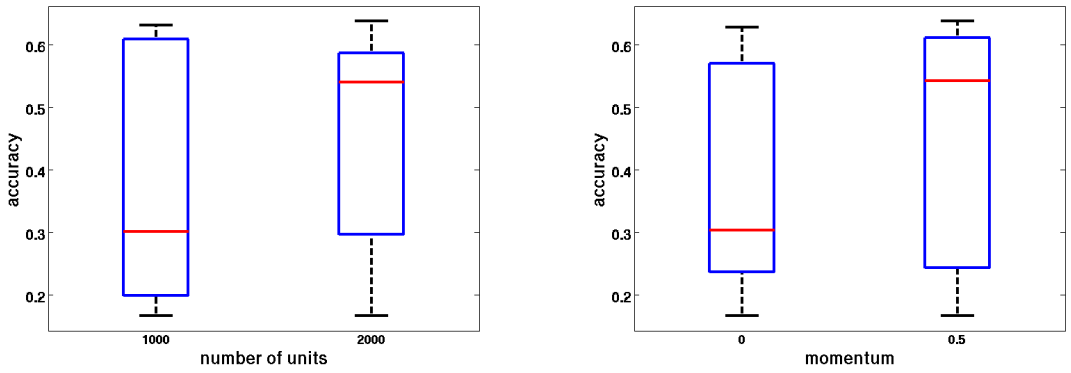
B.3 Comparison of SVM and R^{2}SVM

If the influence of the SVM hyper-parameters and the R^{2}SVM hyper-parameters are compared, the results show that the influence of the SVM hyper-parameters is slightly stronger. This can be seen when looking at the standard-deviation and the range of the validation-accuracies. The SVMs exhibit a standard deviation of the validation accuracy of 9.6%, where the range streches from 16.67% to 56.09%. The standard deviation of the validation accuracy of the R^{2}SVMs is 7.6%, where the validation accuracy lies in the range from 29.8% to 49.08%. In other words, the influence of the SVM hyper-parameters is stronger than the influence of the R^{2}SVM hyper-parameters, though the difference is not as significant as for the MNIST dataset.

B.4 Stacked Sparse Auto-Encoders: Influence of Hyper-Parameters

A higher number of hidden-units tends to increase the accuracy, as can be seen in Figure B.3(a). In contrast to the results presented in Section 5.6.5, the usage of the *momentum* tends to increase the accuracy, as depicted in Figure B.3(b). A further difference to the MNIST-results is that the number of layers influence the validation accuracy: A higher number of layers tends to increase the accuracy. This can be seen when looking at the medians of the validation accuracies for the different number of layers (two hidden-layers: 32.49%, three hidden-layers: 42.16%, four hidden-layers: 53.16%).

Figure B.4 illustrates the influence of the stacked sparse auto-encoder hyper-parameters on the runtime.



(a) (b)

Figure B.3: This figure illustrates the influence of the stacked sparse auto-encoder hyper-parameters on the validation accuracy. (a) The number of hidden-units plotted against the validation accuracy. (b) This graphic depicts the influence of the *momentum* on the validation accuracy.

B.5 K-means Deep Learning Algorithm: Number of Members in a Complex Cell

Figure B.5(a) shows the influence of τ in the first layer on the mean number of simple cells in one complex cell pooling group in the first layer. A higher value of τ tends to increase the mean number of simple cells in one pooling group in the first layer. On the other hand, Figure B.5(b) shows that a higher value of τ in the first layer leads to a decreased mean number of group-members in the second layer. Beside this strong influence of τ of the first layer, a higher τ of the second layer tends to increase the number of group-members in the second layer. Furthermore, a higher number of simple cells in the second layer leads to an increased number of group-members in the second layer, too. Among the other hyper-parameters, no clear influence on the mean number of simple cells in one complex cell pooling group can be identified.

B.6 Stacked Sparse Auto-Encoders: Correlation Matrices

While the correlation matrix in Figure B.6(a) shows the correlation between the classes and the features of the first hidden-layer, the correlation matrix of the second hidden-layer is illustrated in Figure B.6(b). It can be seen that the correlation of the second layer features is higher than the correlation of the first layer features. Furthermore, the clusters in the correlation matrix of the first layer are not as clearly distinguishable from the background as the clusters in the correlation matrix of the second layer. While the structure of the rows ”2” and ”6” (respectively of ”1” and ”4”) is similar in the correlation matrix of the first layer, the structure of these rows is more different in the correlation matrix of the second layer. This indicates that the second

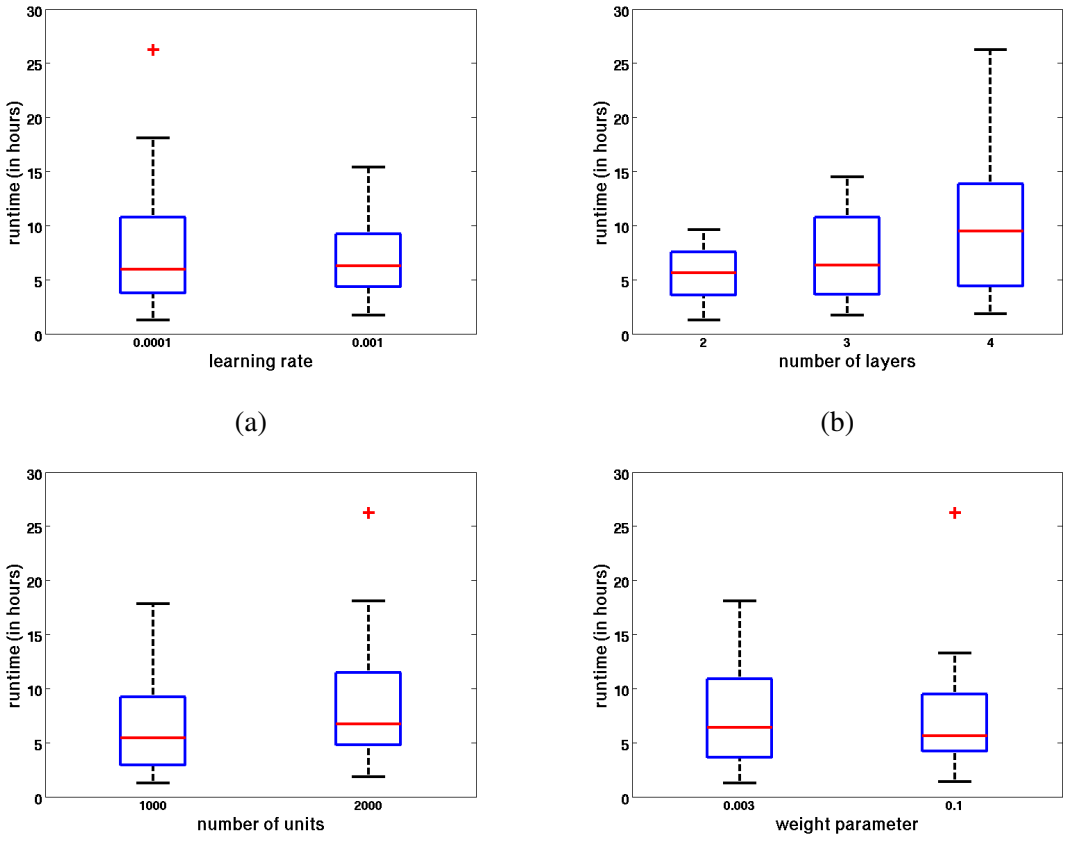
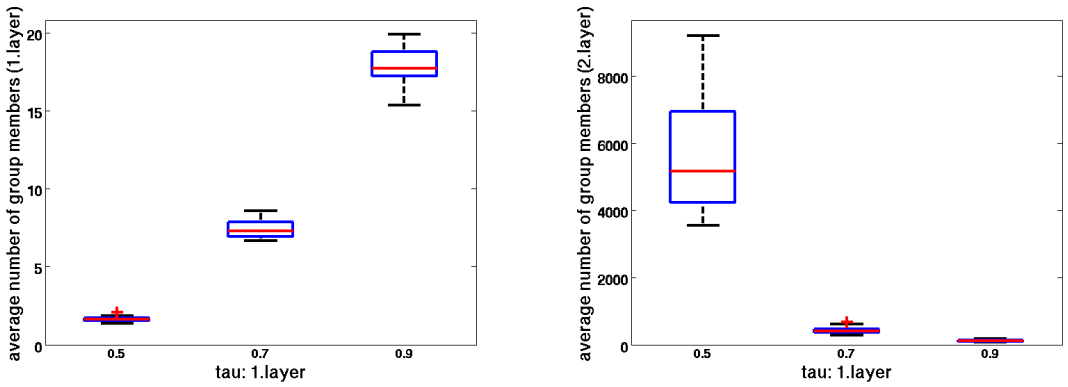


Figure B.4: (a) The *learning rate* is plotted against the time needed to train. (b) A box-plot which illustrates the influence of the number of layers on the training time. (c) The number of hidden-units plotted against the training runtime. (d) This graphic depicts the influence of the *weight decay regularization coefficient* on the time needed to train.

layer features are more class-specific than the first layer features and are therefore more useful to address the classification problem.

B.7 Stacked Denoising Auto-Encoders: Correlation Matrices

The correlation matrix of the first layer is shown in Figure B.7(a), while the correlation between the second hidden-layer features and the classes is illustrated in Figure B.7(b). If these two correlation matrices are compared with each other, it can be seen that both matrices exhibit a similar structure, though the correlation matrix of the second layer shows higher correlations than the correlation matrix of the first layer. On closer examination, it is noticeable that the structures of the rows ”2” and ”6”, the structures of the rows ”1” and ”4” and the structures of



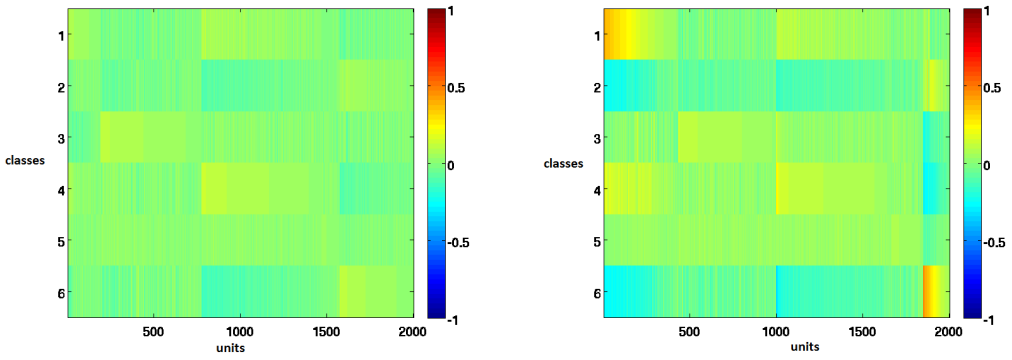
(a) (b)

Figure B.5: This figure illustrates the influence of the K-means hyper-parameters on the mean number of simple cells in one complex cell group. (a) In this box-plot, τ of the first layer is plotted against the number of group members in the first layer (b) This graphic illustrates the influence of τ of the first layer on the number of group members in the second layer.

the rows ”3”, ”4” and ”5” are in each case similar. This indicates that the features learned in the second layer are inappropriate to identify one single class, since the features exhibit a high correlation for multiple classes. This in turn means that only a combination of the second layer features makes it possible to distinguish between different classes.

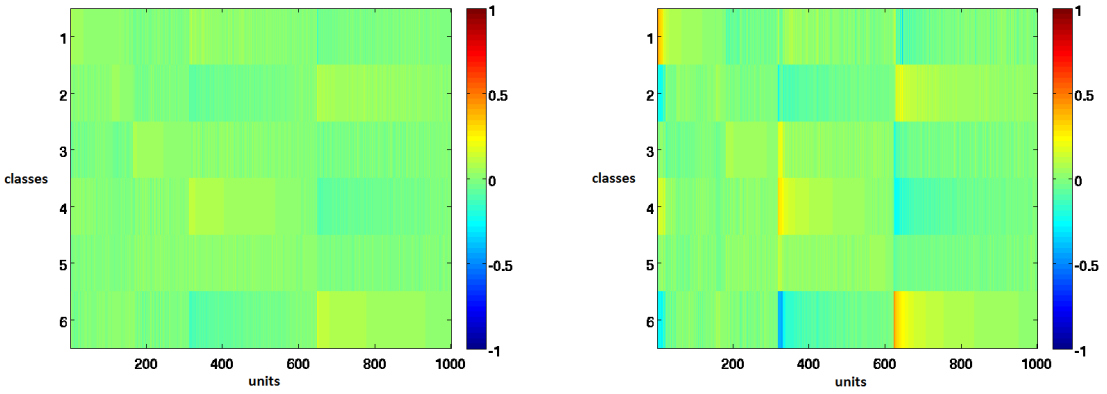
B.8 SVM Support Vectors: Influence of Supervised Training Set Size

Figure B.8 shows the influence of the number of labeled examples on the number of support vectors in the trained SVM. It can be seen that a higher number of training examples tends to increase the number of support vectors, where the number of support vectors increases linearly with the number of labeled training examples per class. More precisely, if the number of labeled training examples per class is doubled, the number of support vectors in the SVM is also doubled. For instance, if the number of labeled training examples is doubled from 256 to 512, the number of support vectors is nearly doubled from 1500 to 2956.



(a) (b)

Figure B.6: Illustration of the correlation matrices, which indicate the correlation between the units of a layer and the classes for the sparse auto-encoder. High values are denoted in red and indicate a strong correlation, while low values are highlighted in blue and indicate a weak correlation. Clustering and a subsequent re-ordering step is applied to improve the visualization of the matrices. The assignment between the numbers and the classes is given in the following: ”1”: Original, ”2”: Normal, ”3”: Ground Glass, ”4”: Reticular, ”5”: Honeycombing, ”6”: Emphysema. (a) The correlation matrix of the first hidden-layer. (b) The correlation matrix of the second hidden-layer.



(a) (b)

Figure B.7: This graphic shows the correlation matrices of the denoising auto-encoder. The colors indicate the correlation-strength, where strong correlations are highlighted in red and weak correlations are denoted in blue. (a) The correlation matrix of the first hidden-layer. (b) The correlation matrix of the second hidden-layer.

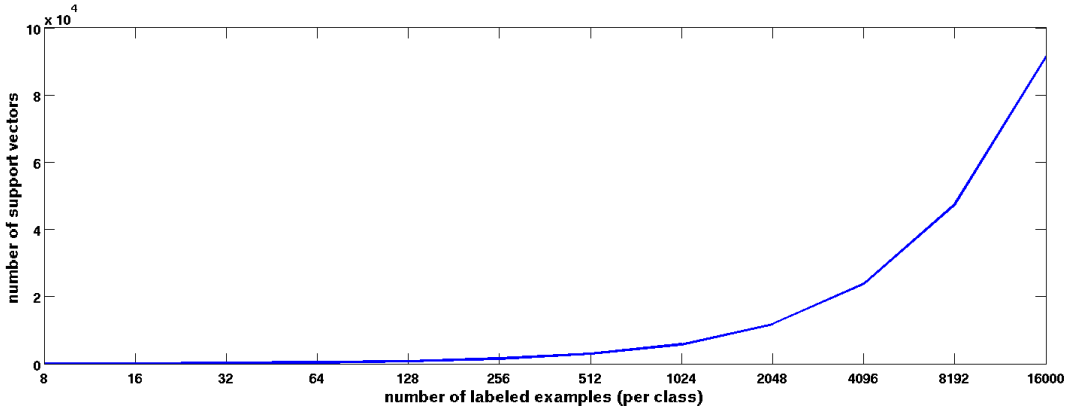


Figure B.8: This figure shows the impact of changing the number of labeled examples on the number of support vectors in the SVM.

Bibliography

[1] Hervé Abdi and Lynne J Williams. Principal component analysis. *Wiley Interdisciplinary Reviews: Computational Statistics*, 2(4):433–459, 2010.

[2] Erin L Allwein, Robert E Schapire, and Yoram Singer. Reducing multiclass to binary: A unifying approach for margin classifiers. *The Journal of Machine Learning Research*, 1:113–141, 2001.

[3] Telmo Amaral, Luís M Silva, Luís A Alexandre, Chetak Kandaswamy, Jorge M Santos, and Joaquim Marques de Sá. Using different cost functions to train stacked auto-encoders. In *12th Mexican International Conference on Artificial Intelligence (MICAI)*, pages 114–120. IEEE, 2013.

[4] JH Austin, NL Müller, Paul J Friedman, David M Hansell, David P Naidich, Martine Remy-Jardin, W Richard Webb, and Elias A Zerhouni. Glossary of terms for ct of the lungs: recommendations of the nomenclature committee of the fleischner society. *Radiology*, 200(2):327–331, 1996.

[5] Graham Bath and Judy McKay. *Praxiswissen Softwaretest: Test-Analyst und technical Test-Analyst*. dpunkt-Verlag, 2010.

[6] Y. Bengio, A. Courville, and P. Vincent. Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1798–1828, 2013.

[7] Yoshua Bengio. Learning deep architectures for ai. *Foundations and trends in Machine Learning*, 2(1):1–127, 2009.

[8] Yoshua Bengio. Deep learning of representations for unsupervised and transfer learning. *Journal of Machine Learning Research-Proceedings Track*, 27:17–36, 2012.

[9] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. In *Neural Networks: Tricks of the Trade*, pages 437–478. Springer, 2012.

[10] Yoshua Bengio, Pascal Lamblin, Dan Popovici, Hugo Larochelle, et al. Greedy layer-wise training of deep networks. *Advances in neural information processing systems*, 19:153, 2007.

[11] Kristin P Bennett and Colin Campbell. Support vector machines: hype or hallelujah? *ACM SIGKDD Explorations Newsletter*, 2(2):1–13, 2000.

[12] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010. Oral Presentation.

[13] Christopher M Bishop et al. *Pattern recognition and machine learning*, volume 1. Springer New York, 2006.

[14] Bernhard E Boser, Isabelle M Guyon, and Vladimir N Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 144–152. ACM, 1992.

[15] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[16] Klemens Burg, Herbert Haf, Friedrich Wille, and Andreas Meister. *Vektoranalysis*, volume 2. Springer, 2012.

[17] Christopher JC Burges. A tutorial on support vector machines for pattern recognition. *Data mining and knowledge discovery*, 2(2):121–167, 1998.

[18] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[19] Olivier Chapelle, Bernhard Schölkopf, Alexander Zien, et al. *Semi-supervised learning*, volume 2. MIT press Cambridge, 2006.

[20] Vladimir Cherkassky and Yunqian Ma. Practical selection of svm parameters and noise estimation for svm regression. *Neural networks*, 17(1):113–126, 2004.

[21] Adam Coates. *Demystifying Unsupervised Feature Learning*. PhD thesis, Standford University, 2012.

[22] Adam Coates, Andrej Karpathy, and Andrew Y Ng. Emergence of object-selective features in unsupervised feature learning. In *NIPS*, volume 25, pages 2690–2698, 2012.

[23] Adam Coates and Andrew Y Ng. Learning feature representations with k-means. In *Neural Networks: Tricks of the Trade*, pages 561–580. Springer, 2012.

[24] Ronan Collobert, Clément Farabet, and Koray Kavukcuoglu. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, 2011.

[25] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[26] Padraig Cunningham and Sarah Jane Delany. k-nearest neighbour classifiers. *Mult Classif Syst*, pages 1–17, 2007.

[27] Li Deng and Dong Yu. *Deep Learning: Methods and Applications*. Microsoft Research, 2014.

[28] Luc Devroye, Laszlo Gyorfi, Adam Krzyzak, and Gábor Lugosi. On the strong universal consistency of nearest neighbor regression function estimates. *The Annals of Statistics*, pages 1371–1385, 1994.

[29] Pedro Domingos. A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78–87, 2012.

[30] René Donner, Georg Langs, Dimitrios Markonis, Matthias Dorfer, and Henning Müller. Report on the anatomical structure identification and localization. Technical Report D2.5, Computational Imaging Research Lab, Medical University of Vienna, 2013.

[31] René Donner, Bjoern H Menze, Horst Bischof, and Georg Langs. Global localization of 3d anatomical structures by pre-filtered hough forests and discrete optimization. *Medical image analysis*, 17(8):1304–1314, 2013.

[32] Juergen Dukart, Karsten Mueller, Henryk Barthel, Arno Villringer, Osama Sabri, and Matthias Leopold Schroeter. Meta-analysis based svm classification enables accurate detection of alzheimer's disease across different clinical centers using fdg-pet and mri. *Psychiatry Research: Neuroimaging*, 212(3):230–236, 2013.

[33] Dumitru Erhan, Yoshua Bengio, Aaron Courville, Pierre-Antoine Manzagol, Pascal Vincent, and Samy Bengio. Why does unsupervised pre-training help deep learning? *The Journal of Machine Learning Research*, 11:625–660, 2010.

[34] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874, 2008.

[35] Holger Frohlich and Andreas Zell. Efficient parameter selection for support vector machines in classification and regression via model-based global optimization. In *IEEE International Joint Conference on Neural Networks*, volume 3, pages 1431–1436. IEEE, 2005.

[36] Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Machine learning*, 63(1):3–42, 2006.

[37] Ian J. Goodfellow, David Warde-Farley, Pascal Lamblin, Vincent Dumoulin, Mehdi Mirza, Razvan Pascanu, James Bergstra, Frédéric Bastien, and Yoshua Bengio. Pylearn2: a machine learning research library. *arXiv preprint arXiv:1308.4214*, 2013.

[38] Geoffrey Hinton. A practical guide to training restricted boltzmann machines. *Momentum*, 9(1):926, 2010.

[39] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[40] DR Holmes III, BJ Bartholmai, RA Karwoski, V Zavaletta, and RA Robb. The lung tissue research consortium: an extensive open database containing histological, clinical, and radiological data to study chronic lung disease. In *The Insight Journal, 2006 MICCAI Open Science Workshop*, pages 1–5, 2006.

[41] Chih-Wei Hsu, Chih-Chung Chang, Chih-Jen Lin, et al. A practical guide to support vector classification. Technical report, 2010.

[42] Aapo Hyvärinen, Jarmo Hurri, and Patrik O Hoyer. *Natural Image Statistics: A Probabilistic Approach to Early Computational Vision.*, volume 39 of *Computational Imaging and Vision*. Springer, 2009.

[43] Alexandre Irrthum, Louis Wehenkel, Pierre Geurts, et al. Inferring regulatory networks from expression data using tree-based methods. *PloS one*, 5(9):e12776, 2010.

[44] Ian Jolliffe. *Principal component analysis*. Wiley Online Library, 2005.

[45] Teuvo Kohonen. *Self-organizing maps*, volume 30. Springer, 2001.

[46] Andrej Krenker, J Bester, and Andrej Kos. Introduction to the artificial neural networks. *Artificial neural networks: methodological advances and biomedical applications. InTech, Rijeka. ISBN*, pages 978–953, 2011.

[47] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. *Computer Science Department, University of Toronto, Tech. Rep*, 2009.

[48] Hugo Larochelle and Yoshua Bengio. Classification using discriminative restricted boltzmann machines. In *Proceedings of the 25th international conference on Machine learning*, pages 536–543. ACM, 2008.

[49] Hugo Larochelle, Yoshua Bengio, Jérôme Louradour, and Pascal Lamblin. Exploring strategies for training deep neural networks. *The Journal of Machine Learning Research*, 10:1–40, 2009.

[50] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[51] Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer, 2012.

[52] Honglak Lee, Roger Grosse, Rajesh Ranganath, and Andrew Y Ng. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 609–616. ACM, 2009.

[53] Hyokyeong Lee and Rahul Singh. Unsupervised kernel parameter estimation by constrained nonlinear optimization for clustering nonlinear biological data. In *IEEE International Conference on Bioinformatics and Biomedicine*, pages 1–6. IEEE, 2012.

[54] Peter Lennie. The cost of cortical computation. *Current biology*, 13(6):493–497, 2003.

[55] Andy Liaw and Matthew Wiener. Classification and regression by randomforest. *R news*, 2(3):18–22, 2002.

[56] Richard P Lippmann. An introduction to computing with neural nets. *ASSP Magazine, IEEE*, 4(2):4–22, 1987.

[57] Markus Maier, Matthias Hein, and Ulrike von Luxburg. Optimal construction of k-nearest-neighbor graphs for identifying noisy clusters. *Theoretical Computer Science*, 410(19):1749–1764, 2009.

[58] MATLAB. *Version 8.0.0 (R2012b)*. The MathWorks Inc., Natick, Massachusetts, United States, 2012.

[59] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[60] Jonathan Milgram, Mohamed Cheriet, Robert Sabourin, et al. "one against one" or "one against all": Which one is better for handwriting recognition with svms? In *Tenth International Workshop on Frontiers in Handwriting Recognition*, La Baule, France, 2006.

[61] Tom Mitchell. *Machine Learning*. The McGraw-Hill Companies, Inc, 1997.

[62] Tom Michael Mitchell. *The discipline of machine learning*. Carnegie Mellon University, School of Computer Science, Machine Learning Department, 2006.

[63] LE Newstrom, Gordon W Frankie, and Herbert George Baker. A new classification for plant phenology based on flowering patterns in lowland tropical rain forest trees at la selva, costa rica. *Biotropica*, pages 141–159, 1994.

[64] Hong Wei Ng and Thi Ngoc Tho Nguyen. Mlsp 2013 bird classification challenge: An approach using bag-of-words features and extremely randomized trees. In *Proceedings of Intl. Workshop on Machine Learning for Singal Proc.(MLSP'13), Southempton, UK*, 2013.

[65] Bruno A Olshausen and David J Field. Sparse coding with an overcomplete basis set: A strategy employed by v1? *Vision research*, 37(23):3311–3325, 1997.

[66] L Breiman JH Friedman RA Olshen and Charles J Stone. Classification and regression trees. *Wadsworth International Group*, 1984.

[67] M Pasławski, E Kurys, and J Złomaniec. Differentiation of linear and reticular opacities in high resolution computed tomography (hrct) in interstitial lung diseases. In *Annales Universitatis Mariae Curie-Sklodowska. Sectio D: Medicina*, volume 58(2), pages 378–385, 2002.

[68] Nicolas Pinto, David D Cox, and James J DiCarlo. Why is real-world visual object recognition hard? *PLoS computational biology*, 4(1):e27, 2008.

[69] Sushravya Raghunath, Srinivasan Rajagopalan, Ronald A Karwoski, Michael R Bruesewitz, Cynthia H McCollough, Brian J Bartholmai, and Richard A Robb. Landscaping the effect of ct reconstruction parameters: Robust interstitial pulmonary fibrosis quantitation. In *10th International Symposium on Biomedical Imaging (ISBI)*, pages 374–377. IEEE, 2013.

[70] Rajat Raina, Alexis Battle, Honglak Lee, Benjamin Packer, and Andrew Y Ng. Self-taught learning: transfer learning from unlabeled data. In *Proceedings of the 24th international conference on Machine learning*, pages 759–766. ACM, 2007.

[71] Yuan Ren and Guangchen Bai. Determination of optimal svm parameters by using ga/pso. *Journal of Computers*, 5(8):1160–1168, 2010.

[72] Bernhard Scholkopf, Kah-Kay Sung, Christopher JC Burges, Federico Girosi, Partha Niyogi, Tomaso Poggio, and Vladimir Vapnik. Comparing support vector machines with gaussian kernels to radial basis function classifiers. *IEEE Transactions on Signal Processing*, 45(11):2758–2765, 1997.

[73] P. Sibi, S. Allwyn Jones, and P. Siddarth. Analysis of different activation functions using back propagation neural networks. *Journal of Theoretical and Applied Information Technology*, 47(3):1344–1348, 2013.

[74] Ingrid Sluimer, Arnold Schilham, Mathias Prokop, and Bram van Ginneken. Computer analysis of computed tomography scans of the lung: a survey. *IEEE Transactions on Medical Imaging*, 25(4):385–405, 2006.

[75] Martijn D Steenwijk, Petra JW Pouwels, Marita Daams, Jan Willem van Dalen, Matthan WA Caan, Edo Richard, Frederik Barkhof, and Hugo Vrenken. Accurate white matter lesion segmentation by k nearest neighbor classification with tissue type priors (knn-ttps). *NeuroImage: Clinical*, 3:462–469, 2013.

[76] Kenji Suzuki. Pixel-based machine learning in medical imaging. *Journal of Biomedical Imaging*, 2012:1–18, 2012.

[77] Kenji Suzuki, Pingkun Yan, Fei Wang, and Dinggang Shen. Machine learning in medical imaging. *Int J Biomed Imaging*, 2012:1–2, 2012.

[78] Jinshan Tang, Rangaraj M Rangayyan, Jun Xu, Issam El Naqa, and Yongyi Yang. Computer-aided detection and diagnosis of breast cancer with mammography: recent advances. *IEEE Transactions on Information Technology in Biomedicine*, 13(2):236–251, 2009.

[79] Vladimir Vapnik. *The nature of statistical learning theory*. springer, 2000.

[80] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103. ACM, 2008.

[81] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *The Journal of Machine Learning Research*, 11:3371–3408, 2010.

[82] Oriol Vinyals, Yangqing Jia, Li Deng, and Trevor Darrell. Learning with recursive perceptual representations. In *Advances in Neural Information Processing Systems*, volume 4, pages 2834–2842, 2012.

[83] Kilian Weinberger, John Blitzer, and Lawrence Saul. Distance metric learning for large margin nearest neighbor classification. *Advances in neural information processing systems*, 18:1473–1480, 2006.

[84] Christel Weiß and Berthold Rzany. *Basiswissen Medizinische Statistik*, volume 5. Springer, 2005.

[85] Vanessa A Zavaletta, Brian J Bartholmai, and Richard A Robb. High resolution multidetector ct-aided tissue analysis and quantification of lung fibrosis. *Academic radiology*, 14(7):772–787, 2007.