3 Key Concepts in Machine Learning and Data Analysis

Katharina Morik
Mirko Bunse

Abstract: Throughout this book, machine learning is employed in order to enhance the knowledge about the structure of our universe and the understanding of particle interactions through large data-producing physical experiments. This chapter gives an overview of the structure of machine learning as a scientific discipline. Since we cannot detail the methods and their foundations, we add pointers to relevant textbooks and survey papers. Our goal is to raise awareness about the theoretical basis of machine learning so that the software that machine learning generously offers is always used with the appropriate caution.

3.1 Overview of the Field of Machine Learning

Machine Learning (ML) is the field of Artificial Intelligence (AI) that builds or enhances a model of some phenomenon. We will give a short overview here, which shows the structure of this large field. The levels of machine learning and their related theories that we discuss are:

- The learning tasks that are most often given by objective functions to be optimized are also well-known in statistics, see Section 3.1.1.
- Algorithms and libraries follow paradigms of processing, see Section 3.1.2. They define pipelines of steps in the overall learning as described in Section 3.1.3. The particular step of feature selection is illustrated by the minimum redundancy and maximum relevance method in Section 3.1.4.
- Optimization methods, in particular Newton-Raphson and Stochastic Gradient Descent, are explained in Section 3.2.

The structural view is completed by pointing at the theoretical questions of machine learning (in Section 3.3). For the classes of learning methods, we selected to present tree models (including the ensemble of trees) in Section 3.4 and deep neural networks in Section 3.5 because these are the most common methods in the analysis of physical data.

https://doi.org/10.1515/9783110785968-003

3.1.1 Learning Tasks

Many approaches to machine learning can be described as finding a predictive function $h : \mathcal{X} \to \mathcal{Y}$ where the learning task is specified by $\mathcal{X}$, the domain of the input data, $\mathcal{Y}$, the domain of the output or label, and the risk measure that quantifies the quality of the prediction. Instantiating the general scheme of learning tasks gives us the definition of the most important classes of learning tasks.

Definition 1. ***Unsupervised Learning***
Given a set of observations $D_U = \{\vec{x}_i \in \mathcal{X} : 1 \leq i \leq N\}$
and a function $ext : \mathcal{C} \to 2^{\mathcal{X}}$ *returning all objects* $\vec{x} \in \mathcal{X}$ *that are mapped to* c,
find concepts $\{c_j \in \mathcal{C} : 1 \leq j \leq k\}$
such that each $\vec{x}_i \in \mathcal{X}$ *is mapped to a concept* $c_j \in \mathcal{C}$,
and such that some quality function is optimized through this mapping.

Definition 2. ***Supervised Learning***
Given a set of labeled observations $D_L = \{(\vec{x}_i, y_i) \in \mathcal{X} \times \mathcal{Y} : 1 \leq i \leq N\}$,
find a mapping $h : X \to y$
such that a quality measure is optimized.

Where the scheme of learning tasks might look simple, if we elaborate on the three parts of it, we see how broad the space is that it covers.

- First, the **input** is characterized, ranging from formulas, database records, value series, sequences, images, and linguistic input, to graphs or even other models. Often, the input data are a sample of vectors D_U with $\mathcal{X}$ being a d-dimensional random variable. Each component is a feature or attribute of a certain domain. If all features are real numbers, $\vec{x} \in \mathcal{R}^d$. The vectors might be organized into some matrix A or into a series over an index $t_0, t_1, \ldots t_T$. For graphs, in addition to the vectors for nodes, there are also edges and possibly even edge features.
  Learning tasks with just these input data are called unsupervised learning. Cluster analysis and matrix factorization are popular methods of unsupervised learning. A clustering delivers a set of concepts $c_1, \ldots, c_k$ where each concept covers a set of data points.
  A method for selecting the appropriate set of features will be described in Section 3.1.4.
- Second, the **target** is specified. If a target value or label y_i is given for every $\vec{x_i}$, we call the task supervised learning. Classification assigns a class to an example, $h : \mathcal{X} \to y, y \in \{+1, -1\}$. Regression assigns a real number to an instance, i.e., $y \in \mathcal{R}$. For time series data, the task is either a classification, i.e. the time series is an instance of a class y, or a forecast, i.e. given a series $\vec{x}_{t_0}, \vec{x}_{t_1}, \ldots \vec{x}_{t_T}$ find $\vec{x}_{T+s}$ for a time span s.

It is possible to also have more complex output [370]. In speech recognition, for instance, a value series of audio input has to be mapped to a series of words, $h : \mathcal{X} \to \mathcal{Y}$.
Since the acquisition of the labels y_i for all given data D_U might be costly, semi-supervised learning uses a small set of observations $x_1, \ldots, x_m$ together with their labels $y_1, \ldots, y_m$ in addition to a huge set of unlabeled observations $x_{m+1}, \ldots, x_N$. It was introduced in the form of directing clustering through expert given advice [127] and then became generalized for text data [340] and graphs [266].
Methods that actively determine which labeled data have the best utility for learning are called active learning. They optimize the utility of labeling for learning regarding its cost. Self-supervised learning goes even one step further in that no human annotation is needed. Instead, the learner forms learning tasks on the basis of the given data. A component of example vectors or some part of a value series is used as the label that is to be predicted. The model is tested on examples with a mask hiding the label part. A more sophisticated approach is to distract the examples, train models on the changed and the true data, and learn how to distinguish between models [75].

– Third, a **loss function, quality measure** , or **risk function** is indicated, i.e., a condition that must be valid for the learned model. $R_D : \mathbb{R}^d \to \mathbb{R}$. It is evaluated on labeled or unlabeled data D_L or D_U. This opens the floor for optimization techniques [95, 293]. The broad field of optimization ranges from evolutionary algorithms [74] to (stochastic) gradient descent [92].
In addition to minimizing the loss or maximizing the likelihood, the goal is to minimize the model complexity and prevent it from overfitting the training data. In general, regularization adds a penalty term to the loss function. This can also be coined as multi-objective optimization [269]. Quality measures and optimization will be described in Section 3.2.

Let us illustrate the three parts of a learning task by regression to see their interaction more clearly. Learning a linear regression function is a simple but widespread task. Regression functions appear in three notations: as a sum, as a matrix product, and as a scalar product:

$$\hat{y} = h_{\vec{\beta}}(\vec{x}) = \sum_{i=1}^{d} \beta_i x_i = \vec{x}^T \vec{\beta} = \vec{x} \cdot \vec{\beta}$$

We see that the parameter vector $\vec{\beta}$ affects the predictions by weighting the features x_i in a weighted sum.

To learn a model from the data set D_L, we must decide on a risk measure. For linear regression, a popular choice is the mean squared error between the predictions and

the ground-truth values.

$$\mathrm{MSE}(\vec{\beta}; D_L) = \frac{1}{N}\sum_{i=1}^{N}(y_i - h_{\vec{\beta}}(\vec{x}_i))^2$$

Due to the simplicity of the linear regression model, it is possible to find a parameter vector $\vec{\beta}^\star = (X^\top X)^{-1}X^\top Y$ that is optimal in the sense of the MSE. In general, however, finding a solution is not that easy. More complex models and loss functions require numerical optimization techniques to select an optimal parameter vector.

The risk measure is typically defined as the average value over example-wise loss values. These loss values are specified through a loss function $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$, which assigns a score to each individual prediction $\hat{y}$. Namely, the (empirical) risk R_D, which is to be minimized over $\vec{\beta}$, is defined as

$$R_D(\vec{\beta}) = \frac{1}{N}\sum_{i=1}^{N}\ell(h_{\vec{\beta}}(x_i), y_i) \tag{3.1}$$

For instance, the mean squared error for linear regression is based on the loss function $\ell_{\mathrm{MSE}}(\hat{y}, y) = (\hat{y} - y)^2$. A different choice is the zero-one loss with $\ell_{01}(\hat{y}, y) = 0$ if $\hat{y} = y$ and $\ell_{01}(\hat{y}, y) = 1$ otherwise. This latter choice results in R_D estimating the probability of misclassifications and is therefore a suitable measure for classification tasks. Other choices of ℓ include the hinge loss, the Huber loss, and many others [327].

Independent of which particular loss function is employed, we want the resulting risk to be as small as possible, so that the predictions $\hat{y}$ are as close to the true outcomes y as possible. However, we must keep in mind that the empirical risk R_D is only a substitute for a greater goal: a minimum *expected* risk R, which is valid for the entire data distribution.

$$R = \int_{\mathcal{X}\times\mathcal{Y}} \mathbb{P}(x, y) \cdot \ell(f_\beta(x), y)\,\mathrm{d}x\,\mathrm{d}y \tag{3.2}$$

This greater goal stems from the fact that we want to predict future data that is unlabeled. In other words: we intend to learn prediction functions that generalize from the observed data D_L to any data set from the same distribution. The difference between R and R_D becomes apparent when we split the data into training and validation sets: if the model class $\mathcal{H}$ is powerful enough to memorize the training set, so that $R_D = 0$, we typically observe a validation error that is greater than zero.

One lesson from observing the difference between R and R_D is that we typically want to prevent our models from a mere memorization of the training set D. To this end, we can employ regularization techniques that impose additional constraints on the model structure. The objective of a regularized optimization task, with a regularization function $r : \mathbb{R}^d \to \mathbb{R}$ and a regularization strength $\lambda \in \mathbb{R}$, is:

$$\vec{\beta}^\star = \arg\min_{\vec{\beta}} R_D(\vec{\beta}) + \lambda r(\vec{\beta}) \tag{3.3}$$

The function r typically does not depend on the training data D but on structural properties such as sparseness, that are desired in an anticipated solution. We substantiate this discussion with two exemplary regularizers, namely the L1 and the L2 norms. The L2 norm, also known as the Euclidean norm, is a popular penalty term that imposes small parameter values. It has the additional benefit of facilitating optimization by being strongly convex. In neural networks, the L2-norm regularization is also called "weight decay".

$$r_{\mathrm{L2}}(\vec{\beta}) = \| \vec{\beta} \|_2 = \sqrt{\sum_{i=1}^{d} \beta_i^2} = \sqrt{\vec{\beta}^T \vec{\beta}} = \sqrt{\vec{\beta} \cdot \vec{\beta}}$$

Picking up the linear regression model, the L2-norm regularization leads to the popular ridge regression technique:

$$\min_{\vec{\beta}} \sum_{i=1}^{N} (y_i - \sum_{j=1}^{d} x_{ij} \beta_j)^2 + \lambda \sum_{j=1}^{d} \beta_j^2$$

The L1 norm, also known as the Manhattan distance, promotes a different structure for the solution. Instead of imposing small parameter values, it promotes some parameters to be close to zero while leaving other parameters intact. In a linear model, L1-norm regularization can thereby perform a selection of features simultaneous to learning the prediction model.

$$r_{\mathrm{L1}}(\vec{\beta}) = \| \vec{\beta} \|_1 = \sum_{i=1}^{d} \beta_i = \vec{\beta}^T \vec{1}$$

Regarding linear regression models, L1-norm regularization leads to the popular LASSO regression technique. For linear dynamical systems (LDS), L1 regularization has been enhanced by a reparameterization approach based on an estimation of time-variant dynamics [310].

Bayesian statistics realizes regularization through a given prior probability distribution that decreases the complexity of the model. The SVM even selects the model that minimizes the error and the model complexity at the same time [378, 381], as measured by the VC dimension (see Section 3.3.1) [379]. Viewing machine learning as data compression under minimum description length [325, 326] has further led to frequent set mining in very large data volumes [384].

3.1.2 Processing Paradigms of Machine Learning

Data may be given as a large data set or may come in as a stream of data. The very large data sets that do not fit into the memory of a single machine require distributed processing. Computing on large distributed compute clusters has led to the programming paradigm of **map & reduce**. A small example illustrates the idea. In the map step, a function is applied to each element of a list, e.g., $map(+1)[1, 2, 3]$ delivers $[2, 3, 4]$. In

the reduce step, a function is applied to the overall list, e.g., $reduce(+)$ then delivers $[2 + 3 + 4]$ [135].

Computing on streams does not allow the algorithm to look at a data point more than once, in the extreme case. The potentially infinite stream moves through the algorithm, which processes a data point and lets it go. A data structure to handle these streams, however, needs to be finite like any other data structure. Therefore, it is often necessary in practice to employ algorithms that only approximate the true properties of the data stream, for instance, in approximate counting [259]. Novel algorithms for learning from streaming data are proposed in Chapter 3 in Volume 1. An open-source library of learning algorithms for massive data streams is MOA [83].

Astrophysical data are *big data*. The notion of big data indicates a very large *volume* of data arriving with high *velocity* and in a large *variety* of types, which need to be handled. Open-source software such as Apache Hadoop "allows for the distributed processing of large data sets across clusters of computers using simple programming models".[1] The Hadoop Distributed File System (HDFS) offers a high throughput of data via parallel and distributed data management. More generally, computer science has developed architectures for big data. The lambda architecture combines the storage of large data sets in a batch layer with a real-time component, the speed layer [261]. The kappa architecture stores the historical data in a way that processes them in a data streaming manner [161]. Chapter 6 presents ways of storing large astroparticle data in more detail. For a framework for learning from data streams see [87]. It has been exploited for astroparticle analyses [88]. A study of big data management and processing for the Cherenkov telescope FACT shows the overall pipeline of streaming data analysis [278].

3.1.3 Machine Learning Pipelines

In general, a data science pipeline starts with the most demanding step, the mapping from a scientific question to a learning task. Most often, the scientific question is split into several ones, each with its own learning task. Chapter 1 describes the interplay of theory development and data analysis in terms of epistemology.

Sampling from all observations the data to be analyzed follows the scientific concern by, say, structuring the observations according to certain concepts. In neutrino detection, for instance, we might be interested in muon, electron, or tau neutrinos and thus form separate learning processes for these concepts. This is also true for simulated data. If one class is dominating, we might change the given distribution. Section 5.2.2 shows how active class selection samples disproportionately from a skewed distribution in order to achieve a sound classification.

1 http://hadoop.apache.org.

The given data format often needs to be transformed for learning. For instance, standard representations of time-stamped data can be transformed in several ways to allow for a successful training of specific learning tasks [277]. The overall process of learning needs to be documented with all meta-parameters. Machine learning frameworks such as RapidMiner[2] offer reproducible, adaptable, and easy-to-understand processes. A complete learning pipeline for the successful learning of neutrino recognition in the IceCube experiment has been developed with RapidMiner [331].

What is most important for successful machine learning is the features of the observed items. Selecting the features that ease learning is a first step. There are three types of feature selection. First, *filter approaches* like the t-test [168] or SAM-statistics [372] compute a scoring function on features, disregarding feature interplay. Second, *wrapper approaches* [230] train a learner with possible feature sets and evaluate each feature set by the accuracy of the embedding learning. Each feature set evaluation demands a cross-validated training of the used learning algorithm. Third, some learning algorithms provide the user with an *implicit feature ranking* that can easily be exploited for feature selection. Such embedded approaches use the weight vector of a linear SVM [381], or the frequency of feature use of a Random Forest (RF) [97]. They are aware of feature interplay and are faster than wrappers but depend on the learning algorithm used. In Section 3.1.4, we describe a general and efficient method of feature selection.

Processes of extracting features from the raw data are often tailored to particular scientific questions. Chapters 7 and Section 8.4.1 present this for particular astrophysical data.

Unsupervised learning may deliver features for a succeeding supervised learning. Unsupervised learning delivers pseudo-labels that are used to optimize an overall cost function as in supervised learning. This approach has also been put forward for neural networks [114].

Such a two-step procedure has been taken to the extreme of one-shot learning [249]. *One-shot learning* adapts given knowledge, which may have been learned before, to a small set of examples. A Bayesian approach works especially well for image data since there are regions of interest that could be interpreted as shapes and characteristics of appearance, e.g., a set of textures. First, pictures of a set of categories are presented to train a prior probability density function. The probabilities of shape and appearance for categories are the given knowledge that forms the space of features. Then, only one labeled example is needed to correctly classify all the instances that are close to it in the learned feature space. This is one type of one-shot learning.

Extracting the features that allow classifier learning with high accuracy can be automatized as a process, where the outer loop of learning and its evaluation with respect to a quality measure embeds an inner loop that creates novel features on the basis of given data. Autonomous feature extraction relies on a well-structured space

2 https://rapidminer.com.

of possible features. Shape and texture have been structured image classification. For value series, a structure over base transformations and shapes of curves has been developed and used for autonomous feature extraction via an evolutionary algorithm [270]. There, features are represented as trees of methods. The evolutionary algorithm creates new features by adding methods to a tree or combining trees. It optimizes over populations of method trees, and in that way performs feature selection.

Some learning algorithms provide the user with an implicit feature ranking that can easily be exploited for feature selection. Such embedded approaches use the weight vector of a linear Support Vector Machine (SVM) [381] or the frequency of feature use of a Random Forest (RF) [97]. As has already been underlined by Tom Mitchell, the hidden layers in a neural network discover useful intermediate representations [274]. This corresponds to autonomous feature extraction with evolutionary algorithms in that the nested loop is the neural network's backpropagation, the outer learning is the last layer, and the hidden layers create possible representations. The supervised training of feed-forward networks is considered representation learning because they extract patterns from the data in the hidden layers and optimize them such that the learned weights strengthen the relevant local patterns [183].

Finally, we might want to optimize the learned model itself. In contrast to the terminology in physics, the term quantization in machine learning refers to compressing a model by turning real-valued numbers into binary or integer ones. Binarized Neural Networks (BNN) quantize the weights now to binary values [207]. A quantization scheme for Tensor Flow maps real numbers to a binary representation and uses integer-only arithmetic during inference and floating-point arithmetic during training, again for saving resources [214]. An approach that fully trains Markov Random Fields (MRFs) using only integer values has been developed in order to save energy [308]. It allows learning and inference on ultra-low power devices that use integer-only arithmetic.

3.1.4 Minimum Redundancy Maximum Relevance (MRMR)

The features span the space in which concepts can be learned. Too many features bring with them the curse of high dimensionality. Having many features might slow down learning. Hence, the goal of feature selection is to find a subset of features that allows predicting the target concept well and has minimal redundancy. Correlation-based feature selection (CFS) [190] iteratively adds the feature which has the best ratio between predictive relevance of the feature and its correlation with the already selected features. Both, predictiveness and correlation, are measured by the entropy-based symmetrical uncertainty:

$$SU(f_i, f_j) = \frac{2 \cdot IG(f_i|f_j)}{H(f_i) + H(f_j)} \tag{3.4}$$

where the information gain IG of feature f_i with regard to feature f_j is divided by the sum of the entropies H of f_i and f_j. Ding and Peng [144] generalized CFS with the capability for handling numerical variables calling it *Minimum Redundancy Maximum Relevance FS* (MRMR). For numerical features, the F-test is used. It reflects the ratio of the variance between classes and the average variance inside these classes. For a continuous feature X and a nominal class variable $y \in Y$ with C classes, both from a data set with n examples, it is defined as

$$F(X, Y) = \frac{(n - C) \sum_c n_c (\bar{X}_c - \bar{X})^2}{(C - 1) \sum_c (n_c - 1)\sigma_c^2} \tag{3.5}$$

with per-class-variance σ_c^2 and n_c the number of examples in class $c \in \{1, .., C\}$. The redundancy of a numerical feature set is measured by the absolute value of Pearson's correlation coefficient

$$R(X, Y) = \frac{Cov(X, Y)}{\sqrt{Var(X) \cdot Var(Y)}} \tag{3.6}$$

or its estimate

$$r(X, Y) = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_i (x_i - \bar{x})^2 \sum_i (y_i - \bar{y})^2}}. \tag{3.7}$$

In order to stabilize the feature selection method, its variance is reduced by bagging, i.e., the feature selection algorithm is executed in parallel on different subsamples of the data, thus delivering an ensemble of feature sets [96]. A fast implementation of an ensemble of MRMR feature selection, which is well-suited for high-dimensional data, has been created [344]. Features that are selected earlier and by many sets in the ensemble are combined to become the one selected feature set.

3.2 Optimization

By now, we have expressed the task of learning a prediction model from data as the task of minimizing an empirical risk function. In the following, we exemplify numerical optimization by the two most popular methods, stochastic gradient descent and the Newton-Raphson method. Both methods search for an optimal parameter vector $\vec{\beta}^\star$ by updating an intermediate estimate $\vec{\beta}^{(k)}$ over multiple iterations, as described in Algorithm 3.1. The two algorithms differ in their choice of search directions $\vec{p}^{\,(k)}$, see line 3 in this algorithm.

Algorithm 3.1: A canonical algorithm for numerical optimization.

Input: an initial guess $\vec{\beta}^{(0)} \in \mathbb{R}^d$ and a risk function $R_D : \mathbb{R}^d \to \mathbb{R}$
Output: a minimizer $\vec{\beta} \in \mathbb{R}^d$ of R_D

1 $k \leftarrow 1$ **while** *a stopping criterion is not met* **do**
2 Choose a search direction $\vec{p}^{(k)} \in \mathbb{R}^d$, using $\vec{\beta}^{(k-1)}$ and R_D
3 Choose a step size $\alpha^{(k)} \in \mathbb{R}$, using $\vec{p}^{(k)}$, $\vec{\beta}^{(k-1)}$ and R_D
4 $\vec{\beta}^{(k)} \leftarrow \vec{\beta}^{(k-1)} + \alpha^{(k)} \cdot \vec{p}^{(k)}$
5 $k \leftarrow k + 1$
6 **end**
7 **return** $\vec{\beta}^{(k-1)}$

3.2.1 Stochastic Gradient Descent

Machine learning tasks are typically characterized by large numbers of training examples that need to be handled with limited resources. This profile needs to be addressed by the numerical optimization method, which takes out the learning according to Equation 3.3. Stochastic Gradient Descent (SGD) is a family of optimization methods that is particularly suitable if N, the number of training examples, is large.
We consider the popular mini-batch variant of SGD, which chooses each search direction $\vec{p}^{(k)}$ based on a mini-batch of $b \ll N$ examples with random indices. Since each of these mini-batches $I_k = \{1 \le i_j \le N : 1 \le j \le b\}$ is a random draw from the complete data set D, it follows that the estimates $\{\vec{\beta}^{(k)}\}$ of the SGD algorithm are a stochastic (Markov) process. This behavior is in contrast to complete-batch algorithms, which access all instances of D in each iteration and therefore produce a deterministic sequence of estimates. Another characteristic of SGD is that only the gradient ∇R_D of the objective function is accessed, but not any higher-order derivative of R_D. Therefore, the per-iteration cost of the mini-batch SGD is very cheap, i.e., it depends linearly on b but does not depend on N. The search direction of this variant is given by the following equation, where all gradients are taken with respect to the parameter vector $\vec{\beta}$:

$$\vec{p}^{\,(k)}_{\text{SGD}} = -\nabla R_{I_k}(\vec{\beta}^{(k)}) = -\frac{1}{N}\sum_{j=1}^{b} \nabla \ell(f_{\vec{\beta}^{(k)}}(x_{i_j}), y_{i_j}) \tag{3.8}$$

Accessing only b examples in each iteration causes a considerable amount of noise in the parameter updates. This issue can be handled by appropriate choices for the step sizes, e.g. by decaying values $\alpha^{(k)} < \alpha^{(k-1)}$. However, since small step sizes can slow down the learning process, there are several other techniques that address gradient noise more directly.

For instance, momentum-based SGD variants alter the parameter update rule of the canonical optimization algorithm presented above. Namely, these variants alter line 5 of Algorithm 3.1 to the following assignment, introducing an additional step size

$\gamma^{(k)} \in \mathbb{R}$:

$$\vec{\beta}^{(k)}_{\text{momentum}} \leftarrow \vec{\beta}^{(k-1)} + \alpha^{(k)} \cdot \vec{p}^{(k)} + \gamma^{(k)} \cdot (\vec{\beta}^{(k-1)} - \vec{\beta}^{(k-2)}) \tag{3.9}$$

The additional step size $\gamma^{(k)}$ weights the contribution of another direction, which is the difference between previous updates. Thereby, the parameter updates are "stabilized" in the sense that they maintain earlier search directions to the degree that is configured through the scalar sequence of step sizes $\{\gamma^{(k)}\}$. Momentum-based techniques are widely applied in practice. For instance, the famous optimization method Adam [227], popular for learning deep neural networks, is based on two orders of momentum to achieve even more stability.

The suitability of SGD for large N stems from the fact that the number of iterations and the per-iteration cost do not depend on N [93]. This property is in contrast to *full-batch* algorithms, where the per-iteration cost is indeed proportional to N. Since the per-iteration cost of SGD is comparably cheap, it scales well with large data sets.

Beyond its desirable scaling behavior, SGD has two more advantages over full-batch algorithms: first, it optimizes not only the empirical risk R_D but also the expected risk R directly. In this regard, SGD is better aligned with the actual goal of learning a generalized prediction model. Second, SGD can optimize functions that are non-convex if these functions are well-behaved in less strict terms [93].

3.2.2 Newton-Raphson Optimization

The Newton-Raphson method [293] differs from SGD in two central aspects. First, it uses not only gradient information but also the second derivative of the objective function R_D. Thereby, it is able to assess the curvature of the search space in exchange for a cost that is quadratic in the number of parameters. Second, is the batch variant of the Newton-Raphson algorithm, i.e., each iteration computes the full derivatives of R_D using all examples in D. Due to these differences, the per-iteration cost of Newton-Raphson is considerably higher than the per-iteration cost of SGD. However, since full derivatives capture considerably more information than an SGD update, fewer iterations are typically needed. Newton-Raphson can therefore outperform SGD in terms of the computational resources that it needs to find an accurate solution if the number of training examples N and the number of parameters d are both sufficiently small. Optimization tasks with a small N and a small d are not very typical in the empirical risk minimization framework of Equation 3.1, but such tasks do have their relevance in other aspects of data analysis such as in the deconvolution problem put forward in Chapter 10.

Complying with the canonical optimization algorithm from Algorithm 3.1, the Newton-Raphson method takes out multiple iterations, starting from some initial guess $\vec{\beta}^{(0)}$. To find a search direction $\vec{p}^{(k)}$, the method evaluates a local second-order Taylor

approximation $\widehat{R}_D{}^{(k)}$ of the actual objective function R_D:

$$\widehat{R}_D^{(k)}(\vec{\beta}) \;=\; \frac{1}{2}\,\vec{\beta}^{\top} H\,\vec{\beta} \;-\; \vec{\beta}^{\top}(H\,\vec{\beta}^{(k)} - \vec{h}), \tag{3.10}$$

where $\vec{h} = \nabla R_D\left(\vec{\beta}^{(k)}\right)$ is the full gradient and $H = \nabla^2 R_D\left(\vec{\beta}^{(k)}\right)$ is the Hessian of the actual objective R_D at the latest estimate $\vec{\beta}^{(k)}$. The minimum of this local approximation can be computed analytically. It defines the search direction of the Newton-Raphson method:

$$\vec{p}\,{}^{(k)}_{\text{Newton-Raphson}} \;=\; -H^{-1}\vec{h} \tag{3.11}$$

SGD and Newton-Raphson are examples of a broad research field that covers numerical optimization algorithms. The interested reader can find additional information on batch algorithms in reference [293]. Stochastic algorithms are covered in the survey by Bottou and colleagues [93].

3.3 Theories of Machine Learning

Machine learning research aims at answering the following questions:

- Which guarantees can be given regarding the error? (Error bounds)
- Which model class is best suited for the problem? (Model selection)
- How many examples are needed? (Sample complexity)
- How does a model scale in terms of runtime, memory, and energy, if the number of examples and the number of dimensions increase? (Resource bounds)

At the most abstract level, machine learning works on *formulas* like that of regularized error minimization of a regression function from Equations 3.1 and 3.3 above. In terms of these functions, the learning tasks are specified. Proving the error bounds of learning models is the shared subject of machine learning and statistics. As has already been stated, also the field of optimization also plays a role. Almost every paper at a machine learning conference such as ICML or ECML PKDD presents or at least substantially bases its results on a proof of error bounds. For an introduction, we recommend the books on statistical and probabilistic approaches of machine learning [192, 284].[3] Another school of theory at this abstract level is the computational learning theory which investigates learnability on the basis of the representations of the feature and the hypothesis space. In Section 3.3.1 the main idea is shown with some hints for further reading.

Given a physical problem and experimental data, no class of learning theories is a priori well suited. As the *no free lunch theorem* points out, every selection of a learning method comes along with its requirements on the one hand and its theoretical guarantees on the other [388, 389].

3 For a series of video lectures on foundations of machine learning, taught by Ulrike von Luxburg, see: https://www.youtube.com/playlist?list=PL05umP7R6ij2XCvrRzLokX6EoHWaGA2cC.

Within a selected general class, there are criteria that help to select a particular method. The most important characteristic of an algorithm is the distinction between batch and online, or distributed and centralized processing. For the development of an application, efficient *algorithms* are to be studied. Worst-case complexity and proven tight accuracy bounds are known for diverse algorithms of the same model class. The competitions of implementing frequent set mining, a task of finding all frequent co-occurrences of database items [163], are well known. 12 varying levels of implementations were reported with their different resource consumption, e.g., memory usage, runtime, and the compression of the resulting model.

Today, we go even further to the level of *implementation on particular hardware*. In earlier days, learning algorithms were tailored from CPU to GPU, as discussed in [309]. Hardware is now particularly designed for the low-latency and high-throughput computational demands of machine learning. The non-von Neumann architectures In-Memory Computing (IMC)/ Processing-In-Memory (PIM) are being extensively researched [78]. Currently, FPGAs are frequently used machine learning accelerators. Different implementations of an algorithm for a von-Neumann architecture and for FPGAs have been carefully explored [107]. Also, the optimization for a fast execution on a particular architecture received interest [231], especially computing on multicore architectures (see Section 6.4 in Volume 1). For convolutional neural networks in particular, their abstract description in terms of the Open Neural Network Exchange Format (ONNX) allows them to synthesize an implementation on the high-level interface of FPGAs and optimize it [176].

Recently, the connection of machine learning algorithms with hardware architectures has become even closer in that a given learning algorithm is not only adjusted to a given architecture; the learning algorithm itself takes care of possible hardware failures within its training procedure. This is particularly relevant for hardware that trades in accuracy for energy savings. In-memory computation, for instance, saves energy but may deliver wrong results. A novel max-margin optimization binarized neural networks succeeded in optimizing the bit-error tolerance [110].

The level of model classes with its ties to statistical learning theory is important, but machine learning investigates more levels of abstraction in collaboration with other fields of computer science. Algorithms for a model class for distributed or streaming learning are based on theoretical computer science work as in [129, 130]. The level of implementations on a specific hardware links machine learning with computational architectures [260]. Particular hardware has even been designed especially for machine learning [217]. Machine learning investigates and contributes to all the levels: from the model class over the algorithms to implementations and even computer architectures. Learning is orthogonal to the hierarchical levels of computer science.

Tab. 3.1: An example of the concept class that consists of conjunctions of Boolean literals.

c_1:	*rainy*	*AND*	*not play golf;*
c_2:	*rainy*	*AND*	*play golf;*
c_3:	*not rainy*	*AND*	*not play golf;*
c_4:	*not rainy*	*AND*	*play golf;*
c_5:	*sunny*	*AND*	*not play golf;*
c_6:	*sunny*	*AND*	*play golf;*
c_7:	*not sunny*	*AND*	*not play golf;*
c_8:	*not sunny*	*AND*	*play golf;*
c_9:	*rainy*	*AND*	*not sunny;*
c_{10}:	*rainy*	*AND*	*sunny;*
c_{11}:	*not rainy*	*AND*	*not sunny;*
c_{12}:	*not rainy*	*AND*	*sunny;*

3.3.1 Computational Learning Theory

In addition to the statistical theory of machine learning, a distribution-independent theory known as the Probably Approximately Correct (PAC) learning, offers bounds of learnability [375]. It is based on the idea of hypothesis spaces. Which hypotheses can be expressed in a particular formal system of representation? An easy concept class is the conjunctions of Boolean literals. Literals include *rainy*, *sunny*, and *play golf*. The hypothesis space $\mathcal{C}$ would then be the set of concepts c_1 to c_{12}.

Learning identifies within the hypothesis space those concepts that are consistent with the examples, i.e., there is no logical contradiction between the true hypothesis and all the examples. In other words, a concept is a hypothesis that is determined by the set of instances that it covers. If there is an example in contradiction with the hypothesis, it is counted as an error. For all domains, we shall have different literals that define the concepts. This is not what the learning theory cares about. The theory is to state whether the class of all concepts that can be expressed in this representation is learnable from examples. Learnability is defined with respect to the number of computing steps that are necessary to identify the target concept in the worst case.

Definition 3. *A concept class $\mathcal{C}$ is* PAC-learnable *by a learning algorithm $\mathcal{A}$ using hypothesis space $\mathcal{H}$, if for all $c \in \mathcal{C}$, distributions $\mathcal{D}$ over the instance space X, ϵ such that $0 < \epsilon < 1/2$, δ such that $0 < \delta < 1/2$, $\mathcal{A}$ will with probability at least $(1 - \delta)$ output a hypothesis $h \in \mathcal{H}$ such that the $error_{\mathcal{D}}(h) \leq \epsilon$, in time that is polynomial in $1/\epsilon$, $1/\delta$ and in the size of the instance space and in the size of the concepts space complexity.*

PAC learning polynomially bounds the number of computation steps needed in the worst case to learn a classifier for a class of concepts. A proof of PAC learnability usually first shows that each target concept in $\mathcal{C}$ can be learned from a number of examples, which is polynomially bounded by ϵ and δ. This lower polynomial bound of the number

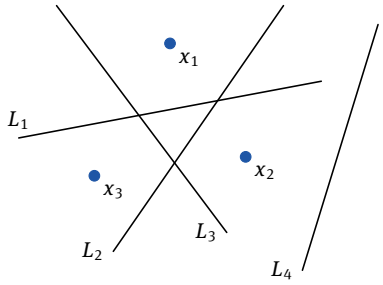

Fig. 3.1: There exists a set of three points $\{x_1, x_2, x_3\}$ for which any binary labeling can be separated by a single, straight line $L \in \{L_1, L_2, L_3, L_4\}$. Therefore, the VC dimension of straight lines is at least three.

of examples is called the sample complexity of the learning algorithm [191, 349]. It shows that the time for processing one example is also polynomially bounded. The overall idea is described more elaborately in [273] and many proofs can be found in [286].

PAC learning often measures the model complexity in terms of the *Vapnik Chervonenkis (VC)* dimension [379]. A PAC learning algorithm is then required to learn a concept class in time that is polynomial in the VC dimension of the hypothesis space.

Definition 4. *A set $\mathcal{H}$ of hypotheses shatters a set D of examples if each subset of $\mathcal{H}$ could be separated by a $h \in \mathcal{H}$. The* VC-dimension *of a set of hypotheses $\mathcal{H}$ equals the maximum number d of examples in D that could be shattered by $\mathcal{H}$.*

We illustrate this by 2-dimensional data and the hypotheses in the form of separating planes as shown in Figure 3.1. *One* set of three points could be shattered by straight lines, but there is *no set* of four points that could be shattered by straight lines. Hence, for the straight line hypothesis space, the VC-dimension is 3.

For the proof of the VC-dimension d the following has to be shown:

- There exists one set D with d points that could be shattered by $\mathcal{H}$.
  $VCdim(\mathcal{H}) \geq d$
- There does not exist a set D' with $d + 1$ points that could be shattered by $\mathcal{H}$.
  $VCdim(\mathcal{H}) \leq d$

The VC-dimension denotes the model complexity and hence allows us to select the least complex model that still learns the target concept. It also gives a hint to the confidence we can have in a learning result. A large VC-dimension indicates a large confidence. There is even a learning method that exploits the VC-dimension by regularizing its internal optimization such that it guarantees a unique and optimal learning result. This method is the support vector machine [380].

The VC-dimension allows us to write a lower bound on the sample complexity. The theorem has been proven for all learners and concept classes [149]. It is given below according to Mitchell [274].

Theorem 1. Lower bound on sample complexity. *Consider any concept class C such that $VCdim(C) \geq 2$, any learner $\mathcal{A}$, and any $0 < \epsilon < \frac{1}{8}$, and $0 < \delta < \frac{1}{100}$. Then, there exists a distribution $\mathcal{D}$ and target concept in $\mathcal{C}$ such that if $\mathcal{A}$ observes* fewer *examples than*

$$max \left[\frac{1}{\epsilon} log(1/\delta), \quad \frac{VCdim(\mathcal{C})-1}{32\epsilon} \right] \tag{3.12}$$

then $\mathcal{A}$ outputs a hypothesis h having error $error_{\mathcal{D}}(h) > \epsilon$ with probability at least δ.

The theorem states that with fewer examples, no learner can PAC-learn every target concept in $\mathcal{C}$. This very general bound for all learners is turned into more specific bounds, when the VCdim is known for the model class.

PAC learning has investigated the learning of neural networks from its very beginning [256] and succeeded in showing that neural networks are capable of approximating arbitrary functions [224]. As is sketched in Section 3.5, the theoretical analysis that gives us tight bounds and a deep understanding of deep learning is still an active research area.

3.4 Tree Models

Decision trees are one of the most successful learning methods: often applied, based on probabilistic theory, and easy to implement. They recursively divide the feature space into smaller and smaller hyper-rectangles until there are only members of one class in the hyper-rectangle, which makes it a leaf node, or until some other stopping criterion is met [319]. While usually used for classification in supervised learning, decision trees may also model regression tasks. Training a tree, where each node covers a set of examples, is performed by selecting the best feature for splitting the current node, creating sub-nodes for each feature value, and passing the examples to those fitting their feature value until a node covers only examples of the same class or has a clear majority of one class. The learned model classifies a previously unseen example by passing it according to its feature values to its leaf.

Selecting the splitting feature X_j often follows the information gain criterium. The probability p_+ that an example belongs to class + is the entropy I:

$$I(p_+, p_-) = (-p_+ \log p_+) + (-p_- \log p_-)$$

A feature X_j with k values divides a set of examples $\mathbf{X}$ into k subsets $\mathbf{X}_1, \ldots, \mathbf{X}_k$. For real-valued features, the numerical values are partitioned into some intervals, so that these intervals are handled along with the discrete features. Binary decision trees always

split into a left and a right branch. For numerical values, a threshold t is used $x_j \leq t$. The best feature X_j or the best threshold t is selected based on a quality criterion such asinformation gain:

$$Information(X_j, \mathbf{X}) := -\sum_{i=1}^{k} \frac{|\mathbf{X}_i|}{|\mathbf{X}|} I(p_+, p_-)$$

The information gain is the difference between the entropy of the examples with and without the segmentation by X_j. It is calculated with respect to the particular set of examples at each sub-tree.

The higher the information gain, the closer a feature is to the root. In this sense, the place in the tree seems to indicate feature importance. However, the order of selected features is *not* a precise feature weighting algorithm. In particular, correlated features along a path in the tree do not share the importance but add it. Hence, they violate the condition that the sum of weights of alternative features must be constant independently of the actual number of alternative features used. This condition is important since it guarantees that a set of alternative features is not more important than a single feature [271].

Decision tree learners are not robust with respect to the order in which examples arrive. Hence, in their original form, they are not applicable to data streams. For a data stream setting, the VeryFastDecisionTree (VFDT) approach has been introduced [146]. They are efficient in that the runtime is proportional to the number of features; moreover, the examples of the stream are not stored but processed just once. In the beginning, a sample of observations at a node is kept. For these, the split criterion is evaluated. The difference between the top features is required to be larger than some ϵ. Reading in additional examples and evaluating the split criterion is continued until the Hoeffding bound is reached. From then on, only aggregated statistics are stored at a node. The Hoeffding bound guarantees that a sum or difference of independent random variables most likely will not deviate more than a constant from the expectation value. Here, it states that with probability $1 - \delta$, after seeing n examples, the feature which receives an evaluation (e.g., information gain) that is ϵ larger than that of the next best feature is indeed the best split criterion also for future examples. ϵ can be calculated:

$$\epsilon = \sqrt{\frac{R^2 ln(1/\delta)}{2n}}$$

with R being the range of feature values, e.g., the log of the number of classes.

3.4.1 Ensemble Methods

The idea of training many decision trees in parallel and letting each tree vote for a class, hence delivering the majority vote as result, is known famously as a *Random Forest* (RF) [97]. The ensemble of many simple models achieves a higher robustness

than a complex model that covers the same observations. Since ensembles are trained in parallel, their runtime is also an advantage. The algorithm for training the random forest is shown in Algorithm 3.2. The application of the random forest delivers the

Algorithm 3.2: The Random Forest algorithm.

Input: number of decision trees l and n examples and desired number of features k
Output: $h_1 \ldots h_l$ mapping X to Y

1 **forall** *the* $i = 1 \ldots l$ *decision trees in the forest* **do**
2 Sample n examples without removal
3 choose $k \ll K$ features from the K given ones randomly,
4 split the set of examples at the node according to the best split, given by, e.g., the information gain
5 **if** *the resulting nodes have enough examples of the same class* **then**
6 stop and output h_i
7 **else**
8 go to line 3
9 **end**
10 **end**

classification

$$h(x) = sign\left(\frac{1}{l}\sum_{i=1}^{l} h_i(x)\right)$$

The procedure is a kind of a bootstrap aggregation—bagging for short. The statistical bootstrap method draws samples and fits models to each of them. The output of bagging is the averaged output of all the models or the majority vote. In any case, the variance of the prediction over the data is decreased. Among the many successful applications of Random Forests are those in astrophysics, namely the IceCube experiment [16, 18, 331] and several Imaging Air Cherenkov Telescopes [48, 258, 294].

Another ensemble method is boosting, first introduced as AdaBoost [169]. Like the Random Forest, it also consists of several learners and decreases the variance of the learning result. In contrast to bagging, boosting is an incremental method that directs the training to areas of the example space that are difficult to learn. It is based on PAC learning, where it has been shown that it is sufficient to have hypotheses that are only slightly better than pure random on the training data because these can be boosted to become arbitrarily correct. The algorithm of AdaBoost is given in Algorithm 3.3. On the one hand, the weak classifiers are weighted by α. On the other, the weights of the examples w_m are updated such that the ones misclassified by $h_m(x)$ receive more impact by $exp(\alpha_m)$. Hence, the next weak classifier $h_{m+1}(x)$ will fit the previously not

Algorithm 3.3: The AdaBoost algorithm.

Input: N examples and the desired number of decision trees M

Output: a prediction rule $h : \mathcal{X} \to \mathcal{Y}$

initialize example weights: $w_{1,i} \leftarrow 1/N \;\; \forall \; i = 1, 2, \ldots N$

forall $m = 1, 2, \ldots M$ **do**

  train a weak classifier $h_m(x)$ using the weighted examples

  compute the error $error_m$ on all data

  $\alpha_m \leftarrow log((1 - error_m)/error_m)$

  $w_{m+1,i} \leftarrow w_{m,i}\, exp(-\alpha_m y_i h_m(x_i)) \;\; \forall \; i = 1, 2, \ldots N$

end

return $h(x) = sign\left(\sum_{m=1}^{M} \alpha_m h_m(x)\right)$

well-covered areas of the example space. There are several applications of boosted decision trees in physics. See Chapter 8.

3.4.2 Implementations and Hardware Considerations

Decision trees and their ensembles have a statistical or PAC learning description as described above. This abstract level is complemented by algorithmic challenges and their implementation on diverse hardware. The highest performance has been achieved by *gradient-boosted trees*. The implementation *XGBoost* is a truly scalable algorithm using external memory and processing the training in a parallel manner on GPUs, exploiting gradient descent optimization (cf. Section 3.2.1) [119].

Other approaches optimize the application or evaluation of the learned model. This is particularly important if the training may be run on a large computing center, but the learned model is to be applied in the wild of a physical experiment like the Cherenkov Telescope Array (CTA) or IceCube. The execution of the model is then restricted in runtime as well as in memory. Traversing a large ensemble of decision trees has been developed for fast inference [242]. A probabilistic view of executing decision trees has been developed in order to optimize the data layout and enhance the cache usage [107]. The improvement is based on the systematic use of tree usage statistics. Two different implementations of decision trees are investigated, namely, an optimized if-then-else tree and an optimized path layout. These implementations exploit computing architectures better and should be considered for real-time applications under resource constraints. Section 7.3.3 in Volume 1) provides more information and explains how to generate optimized code for specific computing architectures.

Machine learning models are often compressed or quantized to use fewer resources. For decision trees, the pruning of sub-trees was put forward from the beginning on [319]. Also, the selection of ensemble members has received attention, e.g. [371], but

this is an active research field. Before deploying a tree model, it is worth determining the appropriate pruning method.

3.5 Neural Networks

Neural networks have attracted attention from their inception. Here, we present the structure of the field and indicate selected literature for further studies.

Neural networks are—most often acyclic—directed graphs with the nodes being organized in layers. The input layer consists of nodes x_i for the input features. The output layer gives the result of the network y or has several so-called *heads* each being a target in a multinomial neural network. Layers between input and output are called *hidden layers*. A neural network with hidden layers is called a Deep Neural Network (DNN) . Figure 3.2 shows a neuron with the weights of the incoming nodes that are summed up and the non-linear activation function, here, the Rectified Linear Unit, which computes $ReLU(z) = max\{0, z\}$. Other activation functions are sigmoid, tanh, and softmax. Just one such neuron is also called a perceptron. Having layers of several such perceptrons is then also called a Multilayer Perceptron (MLP). The connections between the nodes in the following layers may be such that every node of the preceding layer is connected with every node of the next layer, yielding fully connected layers. There are also types of networks with fewer connections between layers. Since the inference feeds the computed values from the incoming signals to the next hidden layer(s) until the output layer is reached, it is a feedforward neural network. If computation also includes feedback connections, the neural network is called a recurrent network.

The term "neural network" originates from the idea that the combination of a weighted sum and a non-linear activation function is reminiscent of a biological neuron cell. In fact, the neuron cells of the human brain become activated when they receive a sufficiently strong signal from their input neurons. However, a neuron cell is much more complex than the simple mathematical function that we call "neuron" here. Also, a brain is much more complex than a simple concatenation of neuron layers. Therefore, a neural network should not be misunderstood as an appropriate model of the biological brain.

Training a neural network is performed by optimizing the weights between the nodes of succeeding layers. These parameters are to be determined such that for all possible inputs, the respective true output value is returned. An output that does not fit the true label starts a *backpropagation* of the error from the last to the first layer. We consider the DNN a chain of functions. Hence, we can propagate the gradients of the loss function using the chain rule. The derivative of $f(g(x))$ is

$$\frac{\partial f(g(x))}{\partial x} = \frac{f}{\partial g}\frac{\partial g}{\partial x}$$

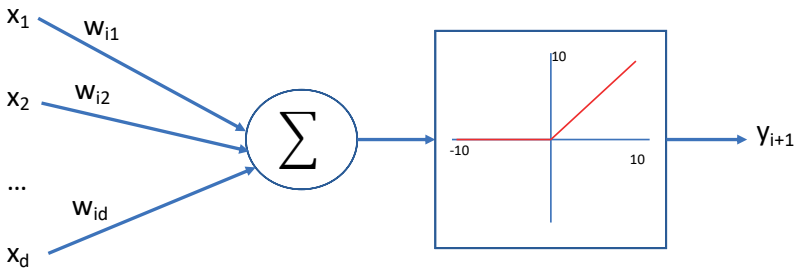

Fig. 3.2: General picture of a neuron with incoming units x_i, the activation function with the summation of the weighted incoming values, and the non-linear ReLU function together computing the resulting unit y_{i+1}.

The backpropagation algorithm stores the derivatives of f with respect to all variables $x = f(w)$, $y = f(x)$, $z = f(y)$. For x, it is

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$$

For w it is

$$\frac{\partial z}{\partial w} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w}$$

As is seen, for the three multipliers, only one needs to be calculated in addition, the others are already stored. This makes backpropagation very efficient.

Once the derivatives are calculated, the optimization uses them. Iteratively, the parameter matrix is updated until a sufficiently good matrix is found. Most often, stochastic gradient descent is used as the optimization method (cf. Section 3.2.1).

For the abstract description of neural network training with backpropagation of errors and weight updates to optimize the network, see [192]. However, this is only a small part of what DNNs are about and what makes them successful. It is the design and development of algorithms and their implementation on diverse computing platforms that produces their excellent performance. For a comprehensive description, it is very much recommended to read the book *Deep Learning* by Goodfellow, Bengio, and Courville [183].

3.5.1 Architectures of DNNs

A DNN consists of a series of layers, each of which can carry out different computations. When we speak of an *architecture*, we mean a particular series of layers, leaving aside the optimization algorithm or the data with which the architecture is trained.

The most fundamental type of layer is called the *dense layer*; it multiplies a weight matrix W to the full vector of incoming values $\vec{h}$ and possibly adds a bias term $b \in \mathbb{R}$. The weighted sum is then fed into a non-linear activation function u, such that the output of the dense layer is $u(W^\top h + b)$, as shown in Figure 3.2. The weight matrix is then optimized during the learning process, and the non-linear function enables the model to learn non-linear dependencies within the data. However, the dense weight matrix W introduces many parameters into the model, the training of which can be ineffective in terms of resource consumption.

Convolutional layers circumvent this problem by sharing parameters among pairs of inputs and outputs. Namely, a kernel of parameters that is much smaller than the input dimension, is moved over the input. The name of this type of layers stems from the *convolution* of two functions, where an input function $x(a)$ and a kernel function $w(a)$ of measurement a deliver a feature map

$$s(t) = \int x(a)w(a-t)\,\mathrm{d}a. \tag{3.13}$$

The discrete convolution over integer values t is

$$s(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a). \tag{3.14}$$

This re-use of parameters leads to fewer connections between the nodes of the preceding and succeeding layers. Moreover, the computation is straightforward to parallelize. Another important property of convolutional layers is their translational invariance. For object recognition in an image, it is not important where exactly the object is. Similarly, patterns in audio input may occur at different widths and heights but should be recognized anyhow. A DNN architecture that uses convolutional layers is called a *convolutional neural network* (CNN).

A more drastic decrease in the number of connections is achieved through the *dropout* layer. This layer randomly ignores, at the succeeding layer, a certain percentage of incoming values with their weights. Dropout can be seen as a regularization of the model.

For image processing, the *pooling* layer is widely used. It summarizes the values in a rectangular neighborhood of nodes by, say, the maximum value or by the average.

The *batch normalization* layer takes the incoming values $z_1, \ldots, z_m$ and calculates mean and variance.

$$\mu = \frac{1}{m}\sum_i = 1^m z_i$$

$$\sigma = \sqrt{\frac{1}{m}\sum_i^m = 1(z_i - \mu)^2}$$

How to combine these building blocks or define new ones is a matter of active research. Several network architectures have been proposed. Starting from AlexNet [233] with

its tremendous success on ImageNet, which offers images labeled into 1000 object categories, such as keyboard, mouse, pencil, and many animals [143], the Oxford Visual Geometry Group proposed a CNN of 19 layers named VGG-19 that is well suited for the recognition of objects in images [350]. *Residual networks* structure a CNN into repeating blocks of convolutional and batch normalization, where each block adds a shortcut from the first to the last layer of the block. This allows enhancing the depth of the network without difficulties in optimization [193].

EfficientNets scales, at the same time, a learned base model in width, depth, and (image) resolution [363]. Experiments show good results for scaling two very different network architectures, namely the deep ResNet and the energy-efficient MobileNet. Bello et al. disentangle architecture and resolution again [81]. They scale the depth and the width depending on the amount of overfitting and apply a slower resolution increase than in EfficientNets. The race for better speed and accuracy in training neural networks thus continue.

3.5.2 Robustness of DNNs

Neural networks are fragile in many respects. On the one hand, little changes in the architecture may have large changes in predictive performance as a consequence. On the other hand, little changes in the data may change the classification of the neural network tremendously. Famous examples are the images that are imperceptibly changed but output a completely different class [362]. Optimizing the least changes leading to the worst accuracy has been called *adversarial attack*. Many types of perturbations have been studied, not only on the data but also on the physical objects that are perceived. In [154], some stickers were attached to a stop sign, and it was classified as "speed limit 45". Defense mechanisms were invented in order to make neural networks more robust. Recently, guarantees have been developed that prove the robustness against particular perturbations [132]. Finally, robustness might also refer to changes in the data distributions [124].

3.5.3 Deep Learning Theory

Neural networks with a single hidden layer of n nodes have been proven to be universal approximators of any measurable function, assuming activation functions $\Psi : \mathcal{R} \to [0, 1]$ that have countably many discontinuities [203]. The number of hidden nodes is not known in general but depends on the function to be approximated. In the worst case, one hidden node is needed for each configuration of the input, i.e., the number of hidden nodes is exponential if we have just one hidden layer. Applying the VCdim (see Section 3.3.1), bounds of the sample complexity of DNN were proven that depend both on the depth and the width of the network. Deeper models require less hidden

nodes in each layer. This is the advantage of deeper networks [364]. The complexity of a DNN can be shown without referring to the width of the layers, but usually depends on its depth. For $||\vec{x}|| \leq B$, d layers, and the matrices $W_1, \ldots, W_d$, the complexity has been shown with regard to the Frobenius norm at most $M_F(j)$ of W_j and m as seen in examples from [181]:

$$\mathcal{O}\left(\frac{B\sqrt{d}\prod_{j=1}^{d} M_{F(j)}}{\sqrt{m}}\right) \tag{3.15}$$

The authors then convert depth-dependent bounds into depth-independent bounds, which are based on some control over the norm of the parameter matrices.

Learning algorithms with a large capacity are capable of fitting randomly labelled data. With increasing depth, neural networks have an increasing capacity of representation, i.e. they approximate increasingly complex functions. Hence, they are capable of fitting pure noise. Does that contradict the statement that DNNs generalize? In general, machine learning should not overfit the training data, or even memorize them, but perform well on previously unseen data! Inspecting the optimization, it was found that true patterns are learned before the overfitting occurs and that dropout and other regularizations prevent the optimization from memorizing [58]. Other approaches to explaining the generalization of DNNs are the coherent gradients of similar examples pointing in the same direction [118] and the stiffness of a network, which measures the impact of the change in one example's small parameters on in the gradient step in the loss of another example. If the network's weights based on one example help to better classify another example, it generalizes well [167].

A way to characterize DNNs is by analogy with Bayesian models. It has been shown that inference with dropout of a DNN approximates a Gaussian process and, hence, that the Bayesian model uncertainty explains the dropout of DNNs [174]. Bayesian models estimate the uncertainty of a DNN model. However, they do not scale well. Hence, a Spectral-normalized Neural Gaussian Process (SNGP) has been proposed that replaces the output layer with a Gaussian process and includes weight normalization in the training [251].

3.5.4 Explanations

Explainable AI has been studied for black-box algorithms of all kinds [187]. Selecting borderline examples or showing the feature importance according to the learned model explains the learned model without looking into the training process. If, however, the explanation refers only to a surrogate model and not to the learned and deployed model itself, they actually do not explain the learning result [177]. The model agnostic methods are complemented by methods for verifying and explaining DNNs. In particular, for scientific data, where we want to model true processes, the modeling procedure itself must be trustworthy.

A survey of verification and explaining DNNs has been framed theoretically [206]. There exists a large variety of methods that explain DNNs [337]. A most prominent method is the layer-wise relevance propagation and its visualization. It shows which areas of an image have been used for classification by a trained model [275]. For image data, this helps users to understand the learned model. Layer-wise weight change helps to understand the training of DNNs [41]. The training process of DNNs can be inspected at each layer—each intermediate representation—in order to find the most influential examples and determine the classes that attributed most to the classification [305].

3.5.5 Hardware Considerations

AI accelerating hardware is a booming market. Many companies offer specialized chips, which are built into phones, tablets, and many other devices. We address several of these developments in Volume 1 of this book series.

Regarding DNNs in particular, a central aspect of accelerating hardware is energy consumption. One example of a processor dedicated to DNNs has been designed by Google for the TensorFlow software, especially for its matrix multiplications. The Tensor Processing Unit (TPU) delivers an order of magnitude better-optimized performance per Watt for machine learning.[4]

Even when using TPUs, DNNs still demand large amounts of energy. In particular, for edge computing and for the Internet of Things, using a Field-Programmable Gate Array (FPGA) as a platform is advantageous. Automatic synthesis of FPGA programs for CNNs has been developed, see, e.g., [176].

In general, Binarized Neural Networks (BNN), i.e., those that calculate with binarized weights and activation function values, consume less energy and require less memory. The use of approximate memory provides DNN training with even lower energy consumption. However, the saving comes at the price of the memory sometimes flipping a bit. A novel approach to BNN training using approximate memory includes robustness against bit errors in the optimization of BNN learning, thus combining the advantages of approximate memory and BNN directly [110].

Examples of how neural networks leverage research in astroparticle and particle physics are described in Chapter 9. Resource consumption of learning methods is a central theme of all chapters.

4 https://cloud.google.com/blog/products/ai-machine-learning/google-supercharges-machine-learning-tasks-with-custom-chip/.