Klaus D. Toennies

# An Introduction to Image Classification

## From Designed Models to End-to-End Learning

Springer

# An Introduction to Image Classification

Klaus D. Toennies

# An Introduction to Image Classification

## From Designed Models to End-to-End Learning

Springer

Klaus D. Toennies
Computer Science Department
Otto Von Guericke University
Magdeburg, Germany

ISBN 978-981-99-7881-6 ISBN 978-981-99-7882-3 (eBook)
https://doi.org/10.1007/978-981-99-7882-3

© The Editor(s) (if applicable) and The Author(s), under exclusive license to Springer Nature Singapore Pte Ltd. 2024
This work is subject to copyright. All rights are solely and exclusively licensed by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors, and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Singapore Pte Ltd.
The registered company address is: 152 Beach Road, #21-01/04 Gateway East, Singapore 189721, Singapore

Paper in this product is recyclable.

# Preface

## What the Book Is About

Labeling an image requires bridging the large conceptual distance between a picture and its semantics. First, the low-level semantics of pixel values of an image is turned into high-level semantics of features. Then, the image is labeled based on these features. The currently most versatile and successful strategy for this is end-to-end learning using a deep neural network which combines all these steps in a common trainable model. A neural network is not a black box. It is based on concepts and methods derived from designed models for feature extraction and classification. The function and behavior of network components can be understood quite well based on traditional components of a designed classification model.

This book commences by laying the groundwork for image classification through image feature extraction and model-driven classification in feature space. Neural networks are then introduced that can learn arbitrary models directly from labeled sample images. The behavior of a neural network is traced back to concepts of a designed classification model. We explicitly discuss relations between network topologies and their ability to represent components of a traditional designed model. It helps to gain insight into the potential and limitations of such networks.

End-to-end learning necessarily defines image classification as a severely underdetermined problem. Besides concepts of network design to create a universal image classifier, we will therefore discuss in detail various concepts of model training that address the solution of such an underdetermined problem.

## How Should the Book Be Used

The book resulted from a course for bachelor students in their senior year who have some background in image or signal processing. Except for Chaps. 2, 6, and 11, which were covered by two lectures each, we spent a single lecture for each chapter. Basic concepts of image analysis and machine learning, necessary to understand the subject of image classification, are introduced. End-to-end learning using deep neural networks is then presented as a versatile means that is firmly rooted in well-understood concepts for information extraction and analysis from images. Understanding these dependencies should make it easy to grasp the potential of such networks (but also its limitations). It is useful as well for network development. Because of their complexity, network model fitting may easily fail if the model is not carefully designed.

Creating suitable network architectures elevates the design process of traditional image classification to a more abstract level. Rather than specifying suitable image features and classifier models for some task, networks are designed that can learn a model for a given problem and the available training data particularly well. It requires an intuitive understanding of properties of a specific network topology or a particular training technique. We cover the topic in some depth to provide this understanding, albeit within the scope of a bachelor course. Occasionally, we will refer to more advanced methods with literature references pointing to surveys on specific topics or to a detailed description. It is intended for those who want to deepen their knowledge in one of the topics but it is not necessary for understanding this introduction.

Each subject is completed by practical exercises in Python that lets the reader experiment with major aspects of treated subjects. Exercises in the first part of the book do not require specific hardware. Exercises with neural networks in the second part use Keras/TensorFlow and will require a suitable graphic card. They have been designed in a way that the kind of GPU to carry out tensor operations is found in many gaming computers or computers of similar capacity. A free online resource for carrying out network training and inference will be listed in Chap. 1 for those who do not have access to the necessary hardware.

We will rarely present complete programs in the book. Implementations for all sorts of computer vision tasks carried out with or without networks can easily be found on the internet. Hence, we did not find it necessary to add them to this book. Furthermore, we want to encourage the reader to use programming as an alternative way of understanding this book's topics. In our view, this kind of experience is less sustainable when just copying programs. Instead, we provide pointers and some program snippets that facilitate the search for suitable modules, classes, or functions in Python/Keras.

If used in classroom, the practical exercises or slight variations of them should let the student experience consequences of design decisions by means of example. Used for self-study, programming should also help to gain a better understanding of the

underlying theoretical concepts. The exercises are also meant to familiarize the student with building blocks in Python/Keras for the various classification models.

Carrying out the programming tasks requires basic understanding of syntax as well as data and control structures in Python. If missing, it can be acquired by taking any of the many free Python courses found on the internet. However, in our experience, a good knowledge of an object-oriented programming language (C++, Java, etc.) and/or of an interpreter language such as Matlab is sufficient for acquiring the necessary Python background on the fly while doing the practical exercises.

## How the Book Is Structured

The book has two different parts:

- Model-driven image classification to extract features from images and to design and train classification models that operate on these features
- Mapping of feature-based classification and feature extraction to universally trainable neural networks

Feature extraction from images comprises established methods. They were selected to motivate and understand trained feature extraction by neural networks later in the book. Classification models are selected to illustrate the connection between Bayesian decision making and discriminative classifiers. In several steps, it leads from Bayesian classifiers to logistic regression.

In the second part, neural networks are introduced for parameter optimization for a linear logistic regression task. We show that mapping from arbitrary feature distributions to a log-linear distribution is possible even for a rather simple fully connected neural network. Convolutional networks are then presented as a means to make this approach practically usable. Model design and training are discussed with reference to the underlying concepts that were developed in the first part of the book.

Magdeburg, Germany Klaus D. Toennies

# Acknowledgments

The book resulted from a series of lectures given by the author at the FDIBA (German Engineering Faculty) of the Technical University Sofia. It is part of an ongoing, DAAD-funded collaboration between FDIBA and the Otto-von-Guericke-University Magdeburg and belongs to a broader commitment of the DAAD and several German universities to support the German faculty at TU Sofia in Bulgaria. I sincerely like to thank Prof. Vassil Galabov, Prof. Stefan Stefanov, Prof. Velko Ilchev, and Maria Kalcheva from the FDIBA and PD Dr. Fabian Neuhaus and Michelle Bieber from the Otto-von-Guericke Universität for their ongoing support. Teaching was funded by the DAAD project 57680396. The practical exercises were developed and carried out in a recently established AI-Lab funded by the DAAD project 57676052.

Finally, I wish to thank all the students who took part in the course. Many showed a keen interest in the subject and continued either in research or industry to work with subjects in computer vision or machine learning. This has been and still is a motivation to carry on the work.

# Contents

# Chapter 1
# Image Classification: A Computer Vision Task


**Abstract** Computer vision retrieves knowledge from pictures. Image classification is an important part of this. In this chapter, you will learn that image classification is an integral part of many problems from computer vision and influences the solutions of many more. We will discuss why image classification is a difficult problem and what are the causes for it. Finally, image classification will be formulated as sequence of steps to find a mapping from an image to a label that represents the meaning of the image. We will introduce the concept of a feature space to represent image characteristics and define classification as feature space partitioning.

Imagine a goal getter awaiting the attack of the opposing team. She is watching her own team and her opponents, she identifies attackers, predicts movement of attackers, defenders, and the ball, and hopefully comes to the right conclusions (Fig. 1.1). All of these tasks primarily rely on visual input comprising the detection of relevant actors and objects, their identification and their tracking. If these tasks were to be carried out by a robot equipped with a camera, they were examples of typical computer vision tasks. *Computer vision* extracts semantics from images comprising a broad spectrum ranging from the extraction of distances between camera and a depicted point in the picture to the recognition of objects in images.

Computer vision methods can be broadly categorized into *early vision* and *high-level vision* methods. See, e.g., Szeliski (2022) for an in-depth treatment of the various subjects of computer vision; the books of Howse and Minichino (2020), and Solem (2012) explain the subject by using Python as programming language which will be used in this book as well. Early vision extracts information from images without knowledge about depicted objects. Examples are stereo vision methods to reconstruct depth of the visible surface or motion detection at every point in the image. High-level vision methods extract object-specific semantics. These methods require prior knowledge about object appearance together with data from the images. Examples are the classification of depicted objects in still images or the tracking of objects in a video sequence.

Within this book, we will concentrate on the classification of objects in still images. It is a core problem of high-level computer vision. Mastering problems associated with object classification will provide you with a theoretical background

© The Author(s), under exclusive license to Springer Nature Singapore Pte Ltd. 2024

K. D. Toennies, *An Introduction to Image Classification*,
https://doi.org/10.1007/978-981-99-7882-3_1

**Fig. 1.1** A goal getter has to decide rapidly on perceived visual information. (UEFA Women's Champions League, Olympique Lyonnais—FC Barcelona, photo by Steffen Prößdorf, licensed under the Creative Commons Attribution-Share Alike 4.0 International license)

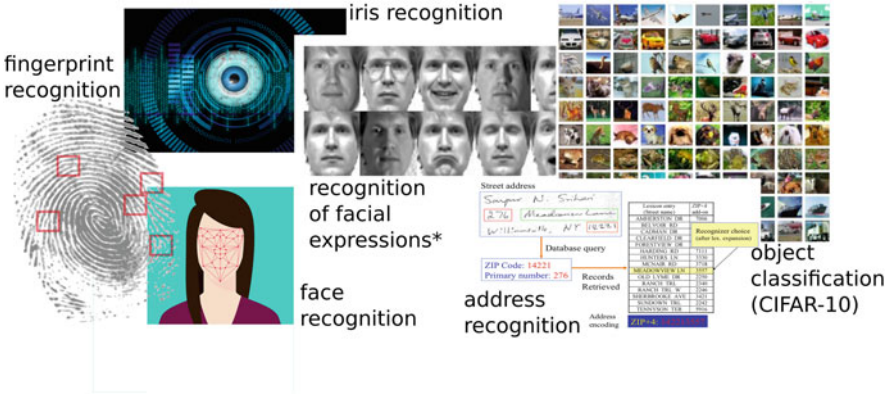



**Fig. 1.2** Some applications for image classification. (*this image is taken from Sajid Ali Khan et al. (2014), licensed under the Creative Commons Attribution 4.0 International license)

and a set of tools that will be also useful when you delve deeper into other fields in computer vision.

Object classification as a standalone problem is part of many vision-based decision systems. Examples are (see also Fig. 1.2):

- The identification of persons based on image-based biometric information (fingerprints, face photos, iris photos, etc.).


**Fig. 1.3** Necessary tasks for autonomous driving include detection and tracking of relevant objects in the path of a car. (Picture by user Eschenzweig, licensed under Creative Commons Attribution 4.0 International Public License, https://en.wikipedia.org/wiki/File:Autonomous-driving-Barcelona.jpg)

- The identification of objects in vision-based driver assistance systems or for autonomous vehicles (e.g., the identification of traffic signs, lane markers, cars, pedestrians, or obstacles of any kind)
- The identification of persons by a surveillance system
- The identification of facial expressions or gestures
- The classification of names, addresses, amounts, and the like on letters or forms
- The classification of letters and words in handwritten texts

The list is incomplete but you see that image classification is a worthwhile undertaking on its own with many practically relevant applications.

Furthermore, the classification of objects in images is part of many other computer vision tasks:

- In *object detection*, regions of interest (ROIs) need to be detected and labeled. The ROI is usually a box aligned with the coordinate axes which are submitted to image classification.
- In *object tracking*, ROIs of identified objects are tracked in a video sequence. Again, these ROIs need to be computed first and then labeled (see Fig. 1.3 for an example that requires object detection and tracking). The video sequence provides additional information that can be used to impose constraints on the two tasks (for instance, that objects do not move erratically and that they do not change their label).
- In *semantic segmentation* every pixel in an image has to be labeled to belong to one of a number of predefined classes (see Fig. 1.4). It is similar to image

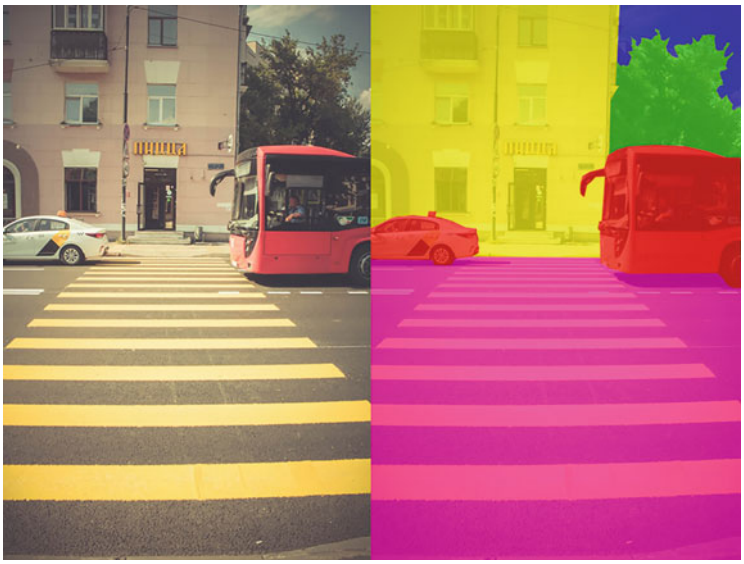


**Fig. 1.4** Semantic segmentation classifies every pixel in an image. Different to segmentation in general, every pixel has a meaningful label (such as vehicle, building etc.). (Picture from user B. Palac, licensed under the Creative Commons Attribution-Share Alike 4.0 International license)

classification but labeling needs to be carried out for every pixel. The best suitable ROI to determine image characteristics may be different for every pixel.
- *Instance segmentation* segments every instance of an object class in the image. Classification happens, similar to semantic segmentation, at a pixel level.

Hence, if image classification has been solved satisfactorily, an important part of many other problems has been solved as well.

Even some computer vision problems that do not inherently contain a classification task may profit from solutions in image classification. In stereo vision, for instance, a major problem is to identify ROIs in the two stereo images that depict the same content (the same parts of the same objects). This is essentially a two-class image classification (same or different content). Some of the methodology used there adopts concepts from image classification (such as the extraction of discriminative features). A similar problem arises when corresponding ROIs in a video sequence shall be detected in motion detection. Hence, the number of applications that apply knowledge from image classification is even higher, comprising, among others,

- depth and motion detection in robot vision or for autonomous vehicles
- motion detection in vision-based surveillance systems.

In summary, understanding concepts and methods used for image classification will provide you with a sound base from which you can explore many other fields of computer vision. And it is not just computer vision. Since part of the solution uses

machine learning methodology, you will also get a first glimpse at machine learning using conventional as well as neural-network-based methodology.

## 1.1 What Is Image Classification and Why Is It Difficult?

For image classification we assume an image that depicts only one object and some background. Since we want to label the image that contains the object, it justifies using the term *image classification*. Multiple-object-classification, where the image may contain objects from several classes, exists as well. Although this is a much harder problem, basic solution strategies are equal or at least very similar. Hence, for an introductory text such as this we will stick to the labeling of an image with a single object label.

Let us take an image of which we know that it depicts a single object and some background. How do we get from here to a solution? Well, we have to assume

- that objects of a certain class (e.g., the class "car") have a number of characteristic attributes that are not shared by objects from other classes (e.g., the class "truck").
- that these attributes are visible in the image.
- that a method exists to extract them from the image.

All these three items are highly problematic.

Let us start with object attributes. Surely, objects have characteristics but a single object attribute (e.g., that a lamp may emit light) may translate to a lot of different shapes and appearances of this object. Object classes are human-defined. For artificial objects (such as cars) class labels originate from the use of the object. A car is a car because it can be used as a car, i.e., it transports passengers but not too many of them, goods but not too many of them, and uses an engine fueled by, e.g., gasoline, to move.

While there is something like "form follows function" which ensures that shape and appearance of an object are related to its use, it allows for many different appearances of members of the same class (see Fig. 1.5). If the functionality of a class is less specific, appearance variation will be even higher. The lamp mentioned above, e.g., is anything that may emit light. Switched off, it stays a lamp. In this case, appearance does not reflect the class-defining characteristic (see Fig. 1.6). Furthermore, a very large number of different lamp designs exist which leads to substantial variation in visual appearance. This is not just true for lamps but for many other objects in the world. Nonetheless, humans are well able to recognize all but the most exotic designs of a lamp.

The problem exists for natural objects as well. We name things based on our own view of the world. It is partially based on appearance—which is good for image-based classification—and partially based on use and/or behavior of the objects—which is not so good as it may not correlate well with appearance.

These examples reflect the basic problem: Appearance characteristics may vary widely among members of a class since object characteristics are often just loosely

**Fig. 1.5** Describing the object "car" by its appearance is difficult. Characteristics may vary between car makes and visible characteristics depend on acquisition parameters of the image

**Fig. 1.6** A lamp is a lamp because is able to emit light. This property is not easily derived from the appearances of different lamps

related to object appearances. Since all classification methods that we will encounter throughout the book just model appearance and do not infer functionality from it, object appearance may vary more within a class than between classes.

For example, even the most intelligent classifier will not be able to infer the simple concept of stereo vision as means to perceive depth and visual information from face images. It will just model the appearance of two eyes and decide on visual


**Fig. 1.7** Edges are important image features to characterize objects. However, edges neither exclusively describe the object nor do they completely outline an object. There is always a certain amount of background clutter to be expected and parts of the object may be hidden or have insufficient contrast to the background or other objects

similarity. Since classification methods are often based on the assumption that similar appearance is equivalent to similar semantics, a classifier will have difficulties to understand the difference between a human face with one eye (which may be possible due to the projection angle or hiding objects) and one with three eyes. This should be kept in mind, especially later in the book when we talk about classifiers that learn everything by example.

The next assumption is that appearance is visible in the image. Let us assume that we are lucky in our object classification problem and know that objects of a certain class (e.g., pears) have always the same shape and that this shape is different from objects (e.g., apples) of another class. We could still run into problems as the image does not depict the complete object but only its visible surface in a projection.[1] Hence, the depicted object shape depends on the projection direction and the distance of the object to the camera (see again the cars in Fig. 1.5). Furthermore, a potentially characteristic color and brightness of the object may change with scene-specific illumination.

The final assumption is that visible attributes can be extracted from the image. Again, there are problems to be dealt with. Since the image contains an unknown amount of background with unknown appearance, it is difficult to decide what of the appearance of the image stems from the object and what not (see Fig. 1.7). An unknown amount of unidentified background characteristics (sometimes called *scene clutter*) will be present and submitted to the extractor. Furthermore, image attributes are low-level features such as color, brightness, or gradients of the two, whereas visible object attributes are object edges, corners, textures, or silhouettes. They are related to low-level image features but the relationship is not simple. An

[1] Computer vision may infer object class from the 3-d shape of an object. A recent example containing a good review of current approaches is Yang and Wang (2019). However, shape reconstruction requires several pre-processing steps to extract object features from images (see again Yang and Wang (2019) for the kind of data that is required to object recognition). Hence, we will restrict ourselves to classification from 2-d projections.

example are textures that are defined by brightness variation within a texture patch of which the boundary is unknown to the feature extractor.

As a first conclusion, object classification based on characteristic appearance attributes seems to be unsolvable. Attributes are difficult to detect and to extract. Intra-class variation may be high compared to inter-class variation. Fortunately, good solutions exist nonetheless and we will explore ways to arrive at such solution within this book.

## 1.2 Image Classification as a Structured Process

Image classification searches for a mapping $f$ from images $i$ to labels $y$, i.e., $y = f(i)$. The optimal function produces the fewest errors when applied to images with known labels (the *training data*). The restriction to a set of possible functions regularizes the problem. It ensures that an optimal function can be found with a limited amount of computation time and a limited number of samples in the training data. For inference, the function is generalized, i.e., it is used as a label predictor for images with unknown labels (this data is often called *unseen data* as it had not been seen when searching for the optimal mapping $f$).

The mapping is most certainly complex given all the problems mentioned in the previous section. Hence, regularization, i.e., defining a suitable set of functions from which to select an optimal one is not easy. Image classification thus often requires a preprocessing step where images $i$ are transformed into a form that is simpler to classify. The transformation attempts to remove as many adverse influences on the appearance as possible.

This process is called *feature extraction*. An adverse influence is one that causes changes of visible object appearance due to the imaging procedure. Examples are image noise, illumination effects, or the change of the projection angle for mapping from 3-d to 2-d. Ideally, extracted image features do not include any such influences.

A further goal of feature extraction is to remove redundant information. Redundancy may be either independent of the analysis task (the kind of redundancy that is also removed in lossless compression) or it may be a task-specific redundancy (features that are not relevant for a specific classification task). Redundance removal reduces computation costs and will improve, as we will see, the generalization ability to label unseen data.

Removing irrelevant and redundant information is the first step to map image features to class labels. The kind of feature extraction depends on the type of classification that follows. Generally, two different strategies for object classification can be distinguished:

1. Matching an appearance model with image features (see Fig. 1.8).
2. Mapping a feature vector to a class label.

The first group of methods comprises approaches such as *template matching*, *active shape and appearance models*, *component models* and the like (see Hashemi


**Fig. 1.8** Template matching slides a template over the image and searches for the location where it fits best. Each class requires its own template. If more than one of them has a good fit at some location, the classifier decides according to the quality of fit for each class

et al. (2016) for a recent review on template matching, Gao et al. (2010) for a review on active appearance models, and the influential paper of Felzenszwalb et al. (2009) on part-based object recognition). A model of the expected appearance of an object class is matched with a subset of the image. The model has been generated from sample objects of this class.

Class membership is decided based on a fitting function. Object models of different classes are matched with the image and the model that best fits the data is the searched class. Advantages of this strategy are:

- that methods solve the detection problem together with the classification problem and
- that the adequateness of the appearance model can be partially evaluated by just inspecting the model.

A major disadvantage is that inter-class variation is not easily distinguished from intra-class variation since the value of the fitting function is the result of a combination of the two. Furthermore, each class requires its own appearance model that needs to be matched with the image. Hence, matching of appearance models is most useful if few classes had to be distinguished and the intra-class variation is less than inter-class variation. In order to ensure the latter, a careful feature selection should exclude any unnecessary within-class variation that later influences the fitness value. Since we are dealing with spatially organized appearance models, spatial relationships between features need to be retained in feature computation.

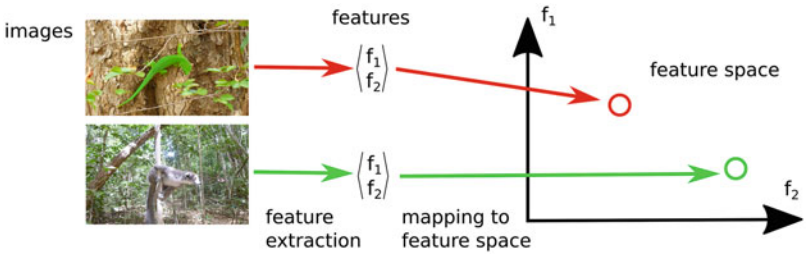



**Fig. 1.9** For classification in features space, features are extracted from the image and mapped to feature space. Classification then requires to partition feature space so that images depicting objects from different classes are found in different partitions

All this is different for the second strategy that generates a multidimensional *feature vector* from the image. The goal is to map everything important in an image to a set of independent features. As features are assumed to be independent from each other, they may be arbitrarily ordered in the feature vector as long as all images use the same order. If, for instance, an image is labeled based on average color $c$ and average brightness $b$, this may be represented either by a vector ($c$,$b$) or a vector ($b$, $c$). As long as one of the two choices is used for all images it does not make a difference for later classification. Features map the image to a specific location in $n$-dimensional *feature space*, where $n$ is the number of features of every image (see Fig. 1.9).

Classification is then a machine learning task in feature space. Sample images with known labels (the already mentioned training data) are mapped to feature space. A classifier is trained in this space to separate samples according to their label. Different to the matching approach above, the number of classes to be distinguished may be almost arbitrarily large provided that enough relevant features can be extracted and that they cluster well in feature space. However, it is more difficult to find out whether the selected features are appropriate for the problem. Features are usually not as descriptive as in the small example above (color, brightness). Rather, they are the result of several processing steps that generate a more and more abstract representation of the original image content.

Since classification in $n$-dimensional feature space is more versatile, we will explore this approach in this book. We will combine methods from image analysis to generate meaningful features with those from machine learning to classify images from these features.

## 1.3 Python Implementation Details

Many of the methods that we will discuss in the book can be explored experimentally. We will use Python to demonstrate image classification methods. Each chapter will conclude with one or two small programming projects in Python where you can carry out these experiments by yourself.

If you have no experience in Python programming, we suggest that you complete one of the many free courses on Python from the internet. You should be familiar with notation, basic data structures, and control structures before jumping into the exercises. Necessary modules, objects, and methods in Python that are specific to image classification tasks will be explained in the text. It is intended to let you gradually develop programming skills to solve image classification tasks.

### *1.3.1 Python Development Environment*

Python is a free interpreter for object-oriented programming. Basic data structures and control structures of the Python interpreter are supplemented by a large number of installable packages for specific operations (such as the numerical Python package NumPy to define arrays and operations on arrays). Although Python programs can be written using any text editor, it may be more efficient to install an integrated developer environment (IDE) to develop, test, and debug Python programs.

For program development, we suggest to use the free IDE Spyder (https://www.spyder-ide.org/) installed via Conda and the Anaconda navigator. Conda is an open-source package and environment management system that runs on Windows, macOS, and Linux (https://docs.conda.io/en/latest/). Anaconda navigator (https://www.anaconda.com) is a graphical user interface to install and launch packages using Conda (see Fig. 1.10). Since Anaconda can have different environments, each with its own packages installed, it is advisable to start with creating your own environment and install necessary packages. The advantage is that you can simply abandon an environment when something goes wrong without compromising the base environment. Spyder is not pre-installed under Anaconda but you can install it through the Anaconda interface. Necessary packages for the programming projects that are probably not pre-installed are

- OpenCV (opencv) for computer vision resources
- TensorFlow (tensorflow-gpu, https://www.tensorflow.org/) for the TensorFlow package with GPU support

These and any other packages can be installed after opening the environment menu (see Fig. 1.10 on the right). All other packages from which we will import Python modules should be already installed. If not, search under the packages that are not installed after opening the Anaconda environment page and install them.

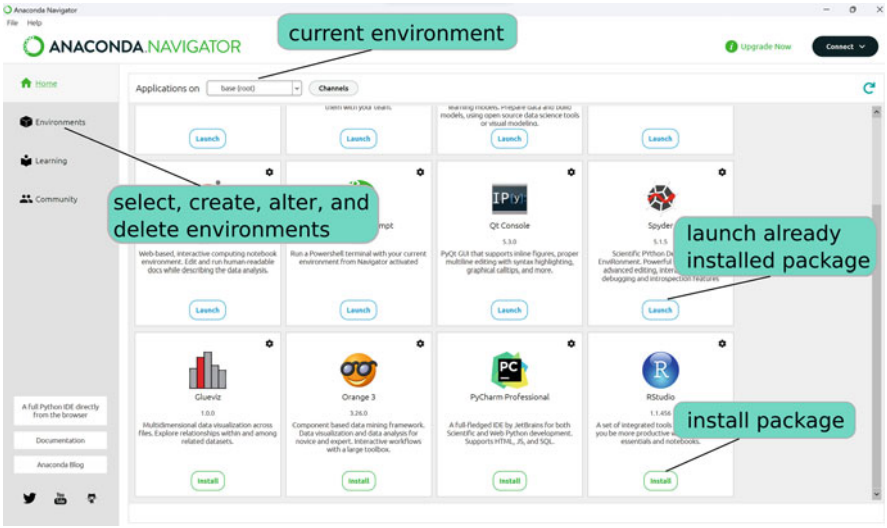



**Fig. 1.10** View of Anaconda as launchpad for various packages

If Anaconda blocks too many resources on your computer, you may use Miniconda instead. It uses command line operations instead of the GUI to install and manage packages. Anaconda and Miniconda take care of all the dependencies when later installing further packages, e.g., for carrying out neural network training and inference on a graphics processing unit (GPU).

Programming exercises in the first six chapters of this book do not require specific hardware. Network solutions for exercises in the remaining chapters need to be carried out on a suitable GPU. We will use the TensorFlow package with GPU support for this. Requirements for running TensorFlow on a GPU can be found on the internet (in short: TensorFlow supports only NVIDIA graphic cards with CUDA compute capability greater than 3.5). Expect problems, however! Since Conda, Spyder, and Tensorflow are independently developed open-source projects, it may be—depending on your hardware and your operating system—a bit of a headache, to find the best combination of packages that are supported by your GPU and the currently installed firmware on the graphics card.

If your computer does not fulfill these requirements, you may alternatively use the Google Colaboratory instead (https://colab.research.google.com/). It uses Jupyter notebooks (https://jupyter.org/) for writing code directly in your browser. The code is executed on external hardware. Jupyter notebooks may be installed standalone but they are also part of the Anaconda distribution. Resources by the Google Colaboratory that are offered free of charge are limited but should be sufficient for the programming exercises in Chaps. 7–12.

### 1.3.2 Basic Modules

A number of modules will be used for carrying out the image analysis tasks. If they are not already installed, you will need to install them before starting your programming project (e.g., through the Anaconda GUI). Necessary modules are the following:

- NumPy (https://numpy.org/) defines a number of objects and methods to work with arrays (arrays are not a basic Python data type). Numerical operations offered in NumPy include, e.g., a number of vector and matrix operations. If NumPy is installed, it needs to be imported by

```
import numpy as np
```

  It allows to access imported classes and functions via "np". For example, the NumPy method to create an array of size (M,N) of data type "unsigned byte" filled with zeros is

```
array = np.zeros((M,N),dtype='uint8')
```

- Matplotlib (https://matplotlib.org/˙) is a module with classes and functions to create visualizations. We will often use it to visualize graphs and output images. We will import a submodule `pyplot` of this module by

```
import matplotlib.pyplot as plt
```

  Similarly, to MATLAB, methods in this module gradually construct the visual output by adding attributes such as axes, titles, etc.
- SciKit-learn (https://scikit-learn.org/stable/) is a library for machine learning and data analysis. We will use this library to experiment with different machine learning methods. Functions and classes are imported by

```
from sklearn import function, Class, ...
```

- SciKit-image (https://scikit-image.org/) is an image processing library that contains most of the functions that we need for extraction and reduction of features from images. There is some overlap with OpenCV (see below) so that it is often a matter of choice which of the two to use. The programmer should be aware of the different ordering of channels of color images, when using OpenCV and SciKit-

image together. OpenCV uses a BGR (blue-green-red) order, while SciKit-image uses a RGB order. A function or class from SciKit-image is imported by

```
from skimage import function, Class,...
```

SciKit-learn and SciKit-image are add-on packages for SciPy (https://svn.scipy.org/scikits.html). As a beginner, you will mostly need the two kits. Some basic image processing functions from the SciPy package need to be imported (e.g., image convolution) since functions already in SciPy are not copied into the add-on packages. SciPy is also a valuable resource, if you delve deeper into machine learning as it contains a wealth of algorithms from linear algebra, statistics, optimization theory, and signal processing.

- OpenCV (https://opencv.org/) is an extensive library of computer vision methods. It is worthwhile to explore the many implemented image analysis methods since almost all established methods have found their way into this library. The module to import OpenCV is called `cv2`. Usually, we need just a small subset of classes or methods for our projects. Hence, just these will be imported by

```
from cv2 import function, Class,...
```

where `Class` and `function` are classes and functions in OpenCV. After their import, they can be accessed and used in your program.

- TensorFlow is a library to build, train, and use networks. It offers classes to define tensors and operation on it. Keras is an application programming interface (API) on top of TensorFlow and other machine learning libraries. It has been integrated as a submodule in the TensorFlow module. Being an API, it bundles TensorFlow objects and operations to new objects that represent the different network components. We will use Keras objects and functions whenever possible and resort to TensorFlow objects only, if absolutely necessary. TensorFlow is imported by

```
import tensorflow as tf
```

If classes or functions from Keras shall be imported, it is done by

```
from tensorflow.keras import function, Class,...
```

### 1.3.3 Some Basic Operations on Images

**Reading and Writing an Image:** Matplotlib, OpenCV, and SciKit-image all have a function `imread()` to read an image from disk (in SciKit-image all input and output operations are found in the submodule `skimage.io`). Unfortunately, the semantics of the three functions is slightly different. Please read the descriptions on the respective documentation before using any of the three functions.

For example, reading the image "test.png" from the directory "./Images" using the function `imread()` in the Matplotlib can be implemented as

```
import matplotlib.pyplot as plt
image=plt.imread('./Images/test.png')
```

The format of the image would be automatically detected and the image will be returned as a NumPy array. Further parameters of the function are explained in the Matplotlib documentation.

Writing `image` to disk at "./Images/output.png" can be done by the function `imsave()` from Matplotlib:

```
plt.imsave('./Images/output.png',image)
```

Again, the function accepts arguments to further qualify the output. The corresponding function (with slightly different semantics) in SciKit-image has the same name, in OpenCV it is `imwrite()`.

**Display of an Image**: A function to display an image on screen using the Matplotlib is

```
plt.imshow(output_image)  # prepare display of output_image
plt.axis("off")           # turn off the display of axes
plt.show()                # show pending image
```

Similar functions exist in the OpenCV and the SciKit-image module.

## 1.4 Exercises

### *1.4.1 Programming Project P1.1: Reading and Displaying MNIST Data*

MNIST is a data set of handwritten digits that has long been used in computer vision and machine learning. We will use it extensively for experimenting with different classification functions. Write a program that selects ten sample images for each label of the MNIST data and display the 10 × 10 images in a common frame. You will need to import NumPy to map the input to an array and `matplotlib.pyplot` to display the created image.

A function to access the MNIST data is included in the `sklearn.datasets` module. Assuming that NumPy is imported as `np`, the data can be accessed by

```
from sklearn.datasets import fetch_openml
mnist = fetch_openml('mnist_784')  # returns a bunch object
                                   # with data and labels
data = np.array(mnist.data)        # extract data from mnist
label = np.array(mnist.target)     # extract labels from mnist
```

Two separate NumPy arrays are created that contain the handwritten digits and corresponding labels (0. . .,9). The array `data` has dimensions 70.000 × 784. Each of the 70.000 digits is represented by a vector of length 784. Use the NumPy function `reshape()` to turn them into gray level images of size 28 × 28 pixels. The array `label` has dimension 70.000 and contains the labels for each sample. Write a function that selects the first ten digits for each label from the data and display them as an image of size 280 × 280 pixels. Use Matplotlib functions for display.

### *1.4.2 Exercise Questions*

- What is the role of a model in computer vision methods?
- What are the advantages of using a computer vision method to obtain comprehensible results?
- Let us assume that you should develop a method to detect pedestrians attempting to cross a street in front of a car and that are captured by a dashboard camera. What appearance characteristics would you expect that help to separate pedestrians from other objects? Note that the camera acquires a sequence of images and the spatial relationship between camera, car, and depicted objects is not arbitrary.
- For the problem in the previous question: What adverse influences may cause unwanted appearance variation and what are the causes for this?

- Why would it be advantageous to select the smallest possible region of interest (ROI) around an object when this ROI shall be labeled?
- What could be a means to reduce shadow artifacts in street scenes when the number of cars shall be counted in images from a stationary camera?
- Why is classification in feature space more versatile than template matching for classification? What could be a reason to use template matching nonetheless for a classification task?
- What for is a feature vector used in a computer vision method? Why should it be generated from images?
- Let us assume that images to be labeled have different number of features. One group of images has features (color, brightness, saturation), another group of images has features (color, saturation), and the last group has color, brightness as features. How could a feature vector look for all images that makes use of as much information as possible.

## References

Ali Khan, S., Hussain, A., Basit, A., & Akram, S. (2014). Kruskal-Wallis-based computationally efficient feature selection for face recognition. *The Scientific World Journal, 2014*, 672630. https://doi.org/10.1155/2014/672630

Felzenszwalb, P. F., Girshick, R. B., McAllester, D., & Ramanan, D. (2009). Object detection with discriminatively trained part-based models. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 32*(9), 1627–1645.

Gao, X., Su, Y., Li, X., & Tao, D. (2010). A review of active appearance models. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews), 40*(2), 145–158.

Hashemi, N. S., Aghdam, R. B., Ghiasi, A. S. B., & Fatemi, P. (2016). Template matching advances and applications in image analysis. arXiv preprint arXiv:1610.07231.

Howse, J., & Minichino, J. (2020). *Learning OpenCV 4 computer vision with Python 3: Get to grips with tools, techniques, and algorithms for computer vision and machine learning*. Packt Publishing Ltd.

Solem, J. E. (2012). *Programming computer vision with python: Tools and algorithms for analyzing images*. O'Reilly Media.

Szeliski, R. (2022). *Computer vision: Algorithms and applications*. Springer Nature.

Yang, Z., & Wang, L. (2019). Learning relationships for multi-view 3D object recognition. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV2019)* (pp. 7505–7514).

# Chapter 2
# Image Features: Extraction and Categories


**Abstract** We introduce feature-based image classification. Extracted features reduce the data of an image by several orders of magnitude. Ideally, they separate object appearance characteristics from image acquisition artifacts and retain just the former. Artifacts originating from image acquisition and strategies for reducing their impact are presented.

Feature extraction is a two-step process. Spatially distributed primary image features represent the image content in a reduced fashion. They are turned into secondary features suitable for classification in feature space.

Various examples of features will be discussed. Simple features are textures, edges or corners. They are generated by first filtering the image content followed by a decision criterion to extract features from the result. Advanced features such as histograms of oriented gradients or SIFT extract more descriptive characteristics from the image and are closely related to appearance features of depicted objects.

Secondary features are feature vectors that represent each image by the same number of corresponding feature values in a common feature space. Mapping from primary to secondary features will remove some of the spatial information from the features. Different means that retain spatial information to various extents will be presented.

Imagine a color mugshot picture with 80 × 100 pixels. It requires a storage space of 80 × 100 × 3 = 24.000 bytes. Given that a character is represented by one byte it translates to about 3.000–5.000 words or 10–15 pages of text. Clearly, even the most accurate description of the information in the mugshot will not require this much space. Hence, a substantial part of the data in the picture will either be redundant or irrelevant for the purpose. Extracting relevant information from an image is one of the two reasons for feature extraction. The second reason is to reformat the data in a way that can easily be processed by a machine learning method.

Image feature extraction for classification in feature space comprises methods to remove redundant or irrelevant information and to create a one-dimensional feature vector from the remaining information. It is often a two-step process with several subprocesses in each step (see Fig. 2.1). The aim of the first step is to separate relevant object characteristics from image acquisition artifacts such as the ones

© The Author(s), under exclusive license to Springer Nature Singapore Pte Ltd. 2024

K. D. Toennies, *An Introduction to Image Classification*,
https://doi.org/10.1007/978-981-99-7882-3_2

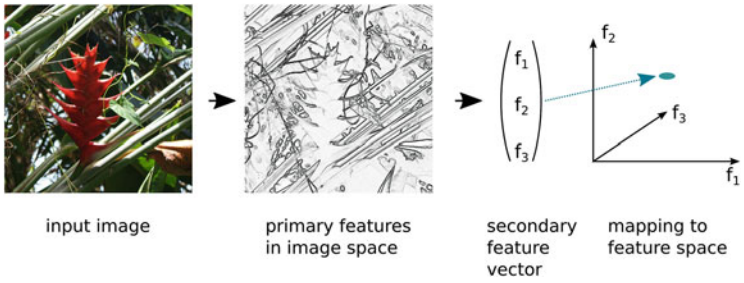



**Fig. 2.1** Feature extraction from images is often a two-step process. Primary features are generated in image space. They are then transformed into secondary features which are representable in feature space

discussed in the previous chapter. The resulting features are still spatially organized in the same way than the original input. If it is a 2-d image they will be features in 2-d space, if it is a 3-d scene they will be distributed in this 3-d space. Let us call these features *primary features* defined in image space. In a second step, they will be mapped to a 1-d feature vector which we will call vector of *secondary features*.

Secondary features represent an image by values of an $n$-dimensional feature vector in feature space. Remember the example from the previous chapter where we described an image by average brightness $b$ and color $c$. Features are not always as descriptive as color and brightness. However, each image is represented by the same number of features. Features at the same position in the feature vector must have the same meaning. It would be unacceptable to describe one image by a vector ($c$,$b$), another by a vector ($b$,$c$), and a third image by a vector ($c$,$b$,$s$) with $s$ being saturation, if the three images shall be labeled by the same classifier in feature space. If the two conditions for a proper feature vector are met, any image is represented by a single location in $n$-dimensional feature space.

It is to some extent a matter of experimentation to determine just what kind of features and what kind of methods to remove redundant or irrelevant information are the most useful. It will depend on the particular classification task. Analyzing this task will help:

- Knowledge on the image acquisition helps to determine what kind of acquisition-related artifacts are to be expected. Often, image acquisition happens under restricted conditions or in a restricted environment. Some artifacts may therefore not occur and need not to be considered. If, for instance, only indoor scenes are to be labeled and the illumination does not change, variation due to different illumination need not to be taken into account.
- Knowledge about the classification task helps to decide what kind of features carry relevant object characteristics. If, for instance, handwritten characters are to be labeled that differ by local configuration of parts with different curvatures, then this kind of shape information should be preserved when generating features.

Even with this a priori analysis, enough choices will be left. Features will differ in their resilience with respect to image acquisition artifacts and their efficiency to represent characteristics relevant for classification. Some popular choices that also stand for different feature selection strategies will be discussed in the following sections.

## 2.1 Image Acquisition Artifacts

Pixel values (intensity or color) as primary features will cause a lot of unwanted feature variation within each class if the picture is a 2-d projection of a 3-d scene. Examples for such acquisition-induced variation on pixel values are (see Fig. 2.2):

- Color changes due to different illumination or different orientation of the object surface with respect to the illumination source
- Intensity changes due to different surface orientations with respect to the illumination source or due to variation of illumination within or between pictures
- Color and intensity change due to shadowing
- Variation due to noise

Furthermore, the shape of the objects may change as well:

- Different projection angles of the same object result in different shapes.
- Different distances of the camera from the scene will change the size of the projected objects.

Ideally, primary features would remove all these sources of variation. Remaining variation is due to differences among depicted 3-d objects. Variation is assumed to be less within object classes than between classes. Hence, objects from different classes should be found at different locations in feature space. A good classifier partitions the feature space accordingly and assigns class labels of a sample based on this partitioning.

Unfortunately, it is impossible to differentiate, e.g., sources of intensity variation due to image acquisition from those due to intra-class appearance variation in pixel values. Removing all image acquisition related influences would remove most or

**Fig. 2.2** All images depict cars. However, a lot of appearance variation is not caused by the different car makes (which would be class-specific variance) but by different projection angles, distances to the depicted car, and lighting conditions

even all of the object appearance features. Hence, primary features will attempt to mainly represent object characteristics with as few artifacts as possible. Even though this causes a higher intra-class variation it is often the only possible compromise if prior image analysis (such as the deduction of object distance, surface curvature, or the localization of light sources) shall be avoided.

Simple primary features to be discussed in the following sections are edges and textures in a picture. They are the base for generating more complex features such as HOG or SIFT features (to be discussed later) that are less dependent on image acquisition artifacts.

## 2.2 Using Pixel Values as Features

Sometimes, pictures are barely affected by artifacts. Examples are the recognition of printed or handwritten characters or digits, the identification of signatures, or the automatic recognition of symbols in maps. In this case, a 2-d document is mapped to a 2-d picture. Lighting conditions of the scanning process are controlled, artifacts stemming from different projection angles, different lighting conditions, or partially hidden object surfaces are not expected. Noise may be present but at a comparatively low level so that the signal-to-noise ratio should be excellent. In cases such as these, pixel values may be taken as primary features.

Taking pixel values, i.e., intensity for gray scale images or RGB-tuples for color images, as primary features for scanned documents is appropriate due to the simplicity and strength of the signal. Printed or handwritten symbols, characters, or digits usually contrast well against the background. A pixel may belong to one of just two classes (part of the object or background) with subclasses for the foreground class (digits, letters). Hence, the spatial distribution of foreground pixels is an indicator for the kind of subclass to which the object belongs.

Examples are data sets of handwritten digits such as the MNIST data set where a region of interest contains a single digit that needs to be determined (Deng (2012), see also Fig. 2.3). The MNIST data set contains 70.000 samples of handwritten digits of approximately the same size and centered in the image. Image resolution is equal for all samples. Classification of MNIST data is often used as a first test for a

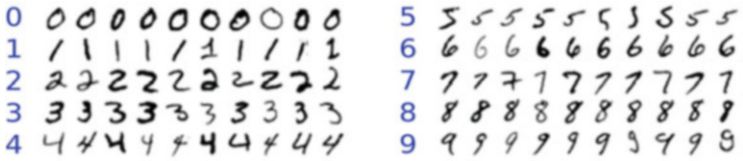


**Fig. 2.3** Examples from the MNIST data base. Each image contains a single digit and no noise. Digits are centered, have the same size and the same color. Since the objects are 2-d, the images contain no projection artefacts. In this case, pixel values can be used as primary features (although other primary features may be better suited for classification)


**Fig. 2.4** A similar but more difficult task for handwritten letter recognition. The detector defined regions of interest that contain the letter but not centered as exactly as the MNIST digits. Interpreting pixel locations as object features will thus introduce acquisition-related artefacts

classifier although it is, as we will see in the exercises, not particularly difficult. Hence, it is mainly useful to analyze and compare rather basic solutions to a classification problem.

In order to generate secondary features, the pixel values can be written row by row in a one-dimensional feature vector. Since the number of pixels per row and column is the same for all samples of the MNIST data, it fulfills the first condition for secondary features as the number of features will be equal for each sample.

The second condition is met as well because the semantics for each entry in the feature vector is the same for all samples. A specific entry simply indicates that the pixel value of this entry has been taken from a very specific location in the picture. For the MNIST data, where all images have $28 \times 28$ pixels, an entry 58 would refer to a pixel in the second row and the second column. A semantics that depends on location is adequate since spatial distribution of foreground pixels relative to each other defines the image class (one of the digits 0–9). Using the absolute position in image coordinates instead of a relative position may cause artifacts but acquisition-induced feature variation will be low as digits are centered.

The latter is not always the case, however. If images are created by determining ROIs in an image (e.g., for separating letters or signatures in a handwritten text, see Fig. 2.4), different ROIs may exhibit different sizes and aspect ratios. Images need to be scaled to a common size to meet the first condition. It needs to be assured that corresponding pixels in the scaled images have similar semantics for meeting the second condition. It is often assumed automatically after rescaling the images but may introduce a substantial amount of variation. Imagine two signatures of the same person of which one is a bit squeezed horizontally, for instance, to fit into a box on the form on which the signature has been written. The rescaling will cause equivalent parts of the signature to appear at different locations which may be interpreted by the classifier as different shapes.

Taking pixel values as secondary features preserves all the information of the primary features. Consequently, the feature vector will contain redundant information. Furthermore, some of the non-redundant information will be irrelevant for the classification task. Redundant or irrelevant information may be removed

- by subsequent feature reduction (see Chap. 3),
- by reducing information in the primary features, and
- by choosing a mapping from primary to secondary features that removes (some of the) redundant or irrelevant information.

A rather simple means for information reduction is to downscale the samples (e.g., from 28 × 28 to 14 × 14 pixels). It retains spatial relationships between foreground pixels in the picture and may not affect the classifier performance if major characteristics of the written digits are preserved.

## 2.3 Using Texture Features

Textures describe characteristics of repetitive pattern on the visible object surface (see Fig. 2.5). Textures are represented by agglomerative measures of intensity variation within a texture patch. It acknowledges that patterns often repeat in some stochastic manner and that several patterns may overlay each other within a patch. However, most texture measures discriminate surface appearances even if repetitiveness is difficult or not at all to detect.

It is an advantage over an alternative way to describe textures by determining building blocks of a repetitive pattern. Distinguishing different textures by syntactical analysis based on such building blocks may be the more exact way to determine

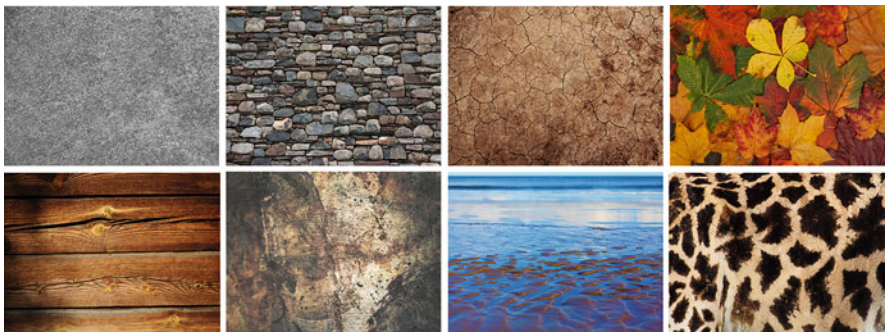


**Fig. 2.5** Examples for different textures. Textures consist of different basic structures that are arranged more or less randomly. Base structures can have their own semantics (like the leaves at the top right), they can occur at different scales (even within a single texture), and they can differ greatly in their characteristics. Textures of projected 3-d scenes also vary depending on distance, camera orientation, and projection direction. A good texture measure should be able to deal with all this

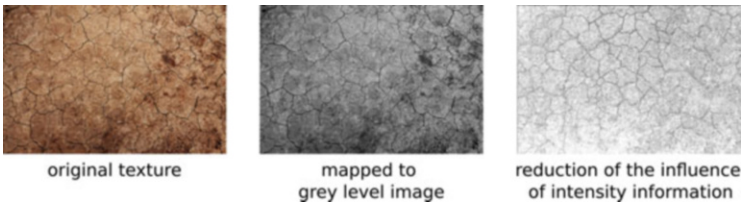



**Fig. 2.6** Texture measures often disregard color and intensity information for making the feature less dependent on illumination artifacts. The example shows that major characteristics of the original texture are still visible after preprocessing

underlying texture characteristics. However, it is often difficult or even impossible to determine the underlying syntax so that statistical analysis of local texture components is the only feasible solution. This kind of problem frequently arises when describing image properties. Hence, the possibly more effective and certainly deeper analysis of syntactical relationships between local image components is foregone in favor of a simpler more efficient treatment of local properties.

Texture patterns are recognizable based on intensity changes (see Fig. 2.6). Hence, texture measures are mostly based on intensity gradients. If textures are computed from color images, intensity is computed by first transforming the RGB color space into an appropriate color space (e.g., HSV) and then taking the intensity value. Often, just computing intensity as length of the RGB vector or a similarly simple approximation is sufficient. Although textures in color images can be computed separately for each color channel as well, the information gain is often low as the correlation between color channels is high.

Texture patterns are mainly distinguished based on location and orientation of high intensity gradients. Consequently, most texture measures are invariant with respect to illumination artifacts. Making them invariant to rotation relative to the image coordinate system is possible but would reduce the expressiveness of the feature. Fortunately, pictures used for classification often come in few (or just a single) canonical views. Hence, rotational differences of different textures in an object-defined coordinate system can be represented by assuming a fixed relation between object and image coordinate system. An example would be the classification of pedestrians from a dashboard camera of a car. A person is assumed to be a pedestrian if this person moves in an upright position. It is of interest to a driver assistance system of the car if it moves in a direction that is more or less perpendicular to direction of motion of the car. Hence, just two projections of the person are relevant (upright from the left or right side).

Texture measures treat the spatial intensity variation as a local feature that is averaged over the texture patch. Various kinds to measure and agglomerate local variation exist resulting in different texture measures, see Humeau-Heurtier (2019) or Armi and Fekri-Ershad (2019) for two recent surveys. In the following, three of them will be presented. They were selected because of their popularity in texture-based image classification. They also serve as examples for different concepts to define and compute textures.

### 2.3.1 Haralick's Texture Features

The texture features of Haralick (1979) treat a texture as result of a stochastic process represented by orientations and frequencies of intensity changes in a 2-d patch. The distribution function is represented by a set of *co-occurrence matrices* $COC_{d,\alpha}(i_1,i_2)$ that are derived from the patch. $COC_{d,\alpha}(i_1,i_2)$ measures how likely intensities $i_1$ and $i_2$ occur at a distance $d$ and angle $\alpha$ in the patch (see Fig. 2.7). If the patch would be of homogeneous intensity, $COC_{d,\alpha}(i_1,i_2)$ would be zero for all $i_1 \neq i_2$. If the patch contains a texture, non-zero off-diagonal elements of the co-occurrence matrix will exist that represent local intensity changes of the texture. The higher the frequency of the texture is in direction $\alpha$, the larger would be the intensity change. Hence, textures of different repetition frequency would lead to non-zero values at different distances from the diagonal in the co-occurrence matrix.

Co-occurrence matrices for pixels at different distances $d$ are created for two reasons:

- to account for suppression of noise. The noise amplitude is usually higher than the signal amplitude at high frequencies.
- to account for the discrete nature of the intensity scale. A low repetition frequency measured at a distance $d = 1$ may not be observable as change of intensity at such a short distance.

However, distances rarely exceed 2 or 3 pixels since the correlation between pixel intensities decreases fast with increasing distance. Co-occurrence matrices with large values for $d$ will increasingly subject to random intensity fluctuation.

Different orientations of texture patterns are not represented in a single co-occurrence matrix. The effect of measuring intensity difference of a low frequency pattern in the direction of the intensity gradient is indistinguishable from that of a higher frequency in a direction that differs from that of the gradient. Hence, co-occurrence matrices for different angles $\alpha$ can be generated (see Fig. 2.8 for an example).

The resulting co-occurrence matrices estimate distributions of intensity change. Parameters of these distributions comprise the texture feature vector. In his original publication, Haralick suggested 14 different features to be computed from all

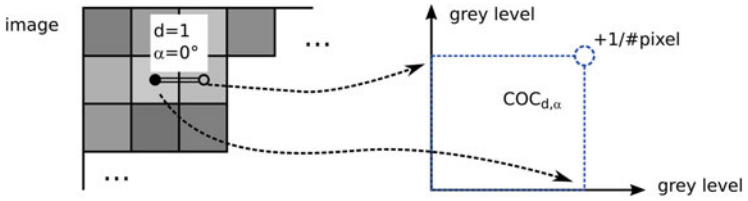



**Fig. 2.7** A co-occurrence matrix $COC_{d,\alpha}$ for a distance $d$ and an orientation $\alpha$ relative to the $x$-axis is generated by incrementing gray level co-occurrences of pixels that are $d$ pixels apart in direction $\alpha$ and dividing the result by the total number of pixels. $COC_{d,\alpha}(i_1,i_2)$ is then the a priori probability how likely intensities $i_1$ and $i_2$ occur at this distance and angle

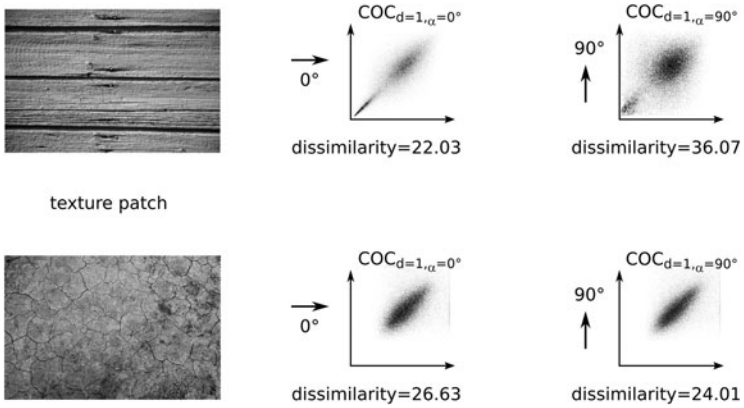



**Fig. 2.8** Computation of co-occurrence matrices for two different textures. Co-occurrence has been computed in direction of the $x$-axis ($\alpha = 0°$) and in direction of the $y$-axis ($\alpha = 90°$). One of Haralick's texture features is dissimilarity of adjacent pixel values, defined as $\sum_{i=0}^{255}\sum_{j=0}^{255} COC_{d,\alpha}(i,j) \cdot |i-j|$. While the first texture has a distinct orientation, which is also evident in the co-occurrence matrices and dissimilarity values, this is not the case for the second texture

normalized co-occurrence matrices. Each feature describes an aspect of the distribution function. Examples are moments, dissimilarity, entropy, expected value, or variance of the distribution. Many application programs compute just a subset of Haralick's texture features. Hence, a developer who uses such a program should first check just what features were generated, especially when comparing his or her results with that of others also working with Haralick's texture features.

Co-occurrence matrices are not rotationally invariant. As directions are computed relative to the image coordinate system, a rotation of the camera with regard to the texture will cause an artefactual variation of the co-occurrence matrix and, consequently, of features computed from it. However, using the distance only when computing co-occurrences would remove information about relative orientations in the pattern (see Fig. 2.9 for an example). It may make discrimination by texture features more difficult. Hence, the artefactual variation of a direction-sensitive pattern is usually accepted since orientation is one of the discriminative characteristics of a texture. This is true for many features that extract orientation from an image patch.

Features from all normalized co-occurrence matrices are finally aggregated in a single feature vector. Hence, if the complete image is taken as a single patch, this primary feature already fulfills the two conditions for secondary features.

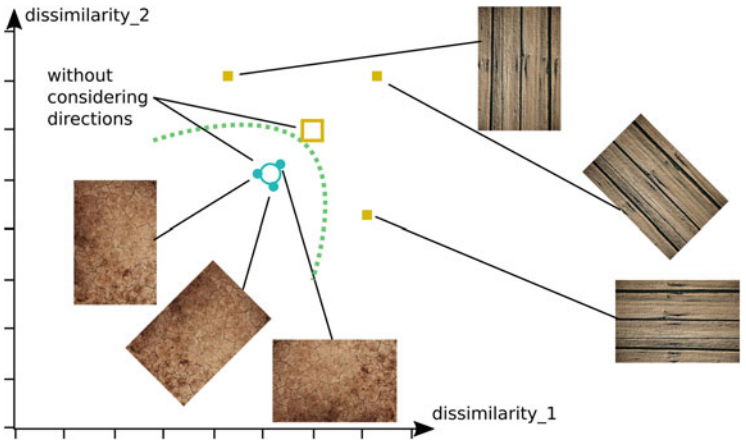



**Fig. 2.9** If a texture is rotated relative to the image coordinate system, texture features will include acquisition-specific variation. In this example, a feature vector is created from two dissimilarity values computed from co-occurrence matrices with $\alpha = 0°$ and $\alpha = 90°$. A rotated texture will result in a different position in feature space. However, just averaging over different angles $\alpha$ results in positions in features space for the two different textures that are much closer to each other since the discriminating characteristic of a preferred orientation in one of the textures is lost. Separation by texture features will be more difficult

**Classes and Functions in Python**

A gray level co-occurrence matrix (GLCM) and its properties can be computed using functions from the submodule `skimage.features`. The function `graycomatrix()` computes a gray level co-occurrence matrix from an image and `graycoprops()` extracts some of Haralick's features from it. The following sequence computes the property "dissimilarity" from an image `img`:

```
from skimage.feature import graycomatrix, graycoprops
glcm = graycomatrix(img, distances=[1], angles=[0], levels=256,
          symmetric=True, normed=True)
img_dissimilarity = graycoprops(glcm, prop='dissimilarity')
```

The function `graycomatrix()` returns a NumPy 4-d array with co-occurrence matrices `glcm(:,:,dist,angl)` for every angle and distance in the two lists `distances` and `angles`. The parameter levels contain the number of gray levels of `img`. If `symmetric` is **`True`**, the output will be symmetric to the diagonal, i.e., `glcm(i,j,k,l)=glcm(j,i,k,l)`. If `normed` is **`True`**, the output are normalized histograms.

GLCM properties that can be derived by `graycoprops()` from a GLCM $G$ from an image with $K$ gray levels are

(continued)

- Contrast $\sum_{i=0}^{K-1}\sum_{j=0}^{K-1} G(i,j)(i-j)^2$
- Dissimilarity $\sum_{i=0}^{K-1}\sum_{j=0}^{K-1} G(i,j)|i-j|$
- Homogeneity $\sum_{i=0}^{K-1}\sum_{j=0}^{K-1} \frac{G(i,j)}{1+(i-j)^2}$
- Angular second moment $\sum_{i=0}^{K-1}\sum_{j=0}^{K-1} G(i,j)^2$
- Energy $\sqrt{\sum_{i=0}^{K-1}\sum_{j=0}^{K-1} G(i,j)^2}$
- Correlation $\sum_{i=0}^{K-1}\sum_{j=0}^{K-1} G(i,j) \frac{(i-\mu_i)(j-\mu_j)}{\sqrt{\sigma_i^2\sigma_j^2}}$

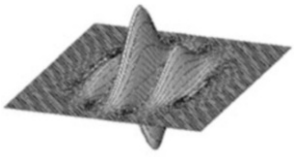
### 2.3.2 *Gabor Filter Banks*

While Haralick's texture features and other functions to characterize features by parameters of a density function describe the frequency of different patterns indirectly, Gabor filters use waveforms as matched filter to determine them directly. A Gabor filter bank is a set of wavelets with different orientations and frequencies that are matched with the texture patch (see, e.g., Roslan and Jamil (2012)). Gabor filters gained popularity as it has been shown that they approximate sensitivity with respect to different texture patterns in animal vision.

A Gabor filter is a windowed Fourier base function

$$G(f,\alpha,\sigma_1,\sigma_2) = s(f,\alpha)e(\alpha,\sigma_1,\sigma_2), \tag{2.1}$$

where $s$ is the so-called signal carrier and $e$ is the envelope (see Fig. 2.10). The signal carrier $s$ is a Fourier base function with a given frequency $f$ and orientation $\alpha$. The envelope $e$ is an equally rotated Gaussian function with variances $\sigma_1$ and $\sigma_2$ that



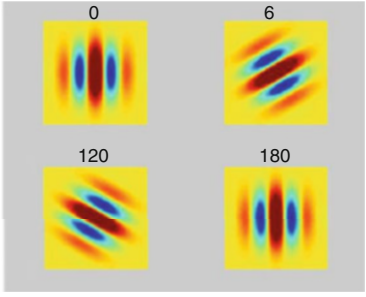



**Fig. 2.10** A Gabor filter is a windowed Fourier base function. A texture feature is computed by convolving this filter with the texture patch and return the average response. Gabor filters with different orientations and frequencies extract scale- and orientation-sensitive characteristics of a texture

determine how quickly the carrier signal is attenuated in direction $\alpha$ and perpendicular to it. Multiplying the signal carrier with the envelope produces a bandpass filter. If the envelope is centered at the origin, it is a lowpass filter on the wave represented by $s$. Hence, the Gabor filter acts as a local, matched filter with high responses in regions where the image content is similar to that of the filter.

The texture measure for a single Gabor filter is the average response from all pixels in a texture patch. It is generated by convolving the texture patch with a Gabor filter and averaging over the result. The more prominent and widespread the pattern of this filter is in an image, the higher the output value will be.

Texture features, defined by Gabor filters, are computed from a bank of filters and represent occurrence and prominence of repetitive patterns of different frequency and orientation. They can directly be used as secondary feature because the two conditions (equal number of features for every sample with feature-wise equivalency of semantics) are met.

Similar to Haralick's features a repetitive nature of patterns is assumed which means that spatial information about specific patterns, that occur only in a part of the texture patch, is not preserved.

**Objects and Functions in Python**

A function that creates and executes a Gabor filter bank is not contained in the modules listed in Sect. 1.3. However, OpenCV provides a function getGaborKernel() and a function filter2D() that convolves an input image img with an arbitrary filter function. A loop can be implemented that creates Gabor filters, filters the image in a NumPy array img, computes the average response from the filter, and submits the result to another NumPy array feature:

```
import numpy as np
from cv2 import getGaborKernel, filter2D
i=0
feature = np.array((24,)) # one-dimensional feature vector
for theta in range(4):   # orientations of the filters
  theta = theta / 4. * np.pi
  for sigma in (1, 3):  # standard deviation of the Gaussian
    for wavelength in (1.0, 2.0, 3.0): # wavelength of the Gaussian
        filter = getGaborKernel((21, 21), sigma, theta,
                    wavelength, 0.5, 0,
                    ktype=cv2.CV_32F)
        result = filter2D(img,-1,filter)
        feature[i] = np.sum(result)
        i += 1
```

A similar function for generating Gabor filters exists in `skimage.filters`. There is no general convolution function in `skimage`. Instead, the function `convolve()` from the SciPy submodule `scipy.ndimage` can be used.

### 2.3.3 *Local Binary Pattern*

Local binary patterns (LBP), presented by Ahonen et al. (2004), code textures based on orientation (but not frequency) of patterns in a patch. A local binary pattern compares the value of each pixel with that of pixels in its neighborhood. In the original paper, the neighborhood consisted of the 8-neighbors of this pixel. Other neighborhood systems stretching over larger distances have been used as well (see Fig. 2.11).

If the 8-neighborhood is used, every neighbor is represented by a single bit in a byte. The bit stores whether the corresponding neighbor pixel has a value higher or equal than that of the current pixel (1) or not (0).

A local binary pattern stores information about directions of gray level changes relative to the image coordinate system. Hence, a specific binary pattern code may represent edges passing though this pixel. A homogeneous region can be represented as well. In a perfect image without noise, the LBP pattern would be 11111111 since all pixels have the same intensity. In the case of noise, a random number of pixels randomly distributed will have smaller intensities and the pattern would be a random sequence of 0s and 1s. Compared to the two texture features presented in the previous sections, preferred orientations that may occur in a texture do result in a less prominent signal but they can be discriminated nonetheless. Figure 2.12 shows an example of this.

In principle, the pattern may also represent more than one edge passing through a pixel. If, for instance, two edges meet at a right angle in a pixel, a pattern could be 00111111. This ability is limited, however, as the extent of local contrast is not reflected in the pattern. If too many edges meet, i.e., if too many different texture frequencies overlay each other, patterns will exhibit a similar kind of randomness than patterns in homogeneous regions.

The low information content of such random distribution of 0s and 1s was the reason for the definition of uniform binary patterns (see Pietikäinen (2010)). Any pattern that contains at most two transitions from 0 to 1 or from 1 to 0 is called uniform (such as the patterns 00111111 or 01000000). There are 58 such uniform

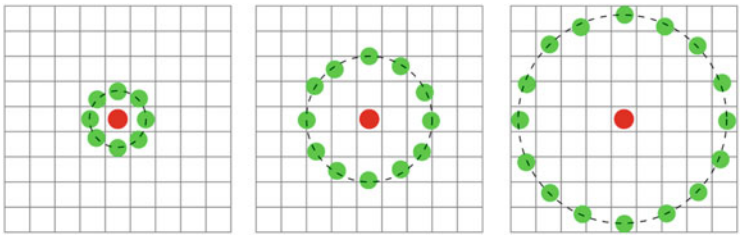


**Fig. 2.11** Originally, just the 8-neighbors of a pixel were considered for computing the LBP. Adjacencies stretching across larger distance can be used as well. It may result in a larger number of nodes, i.e., a longer code but may be less susceptible to noise. Image by user Xiaxi, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=11743214

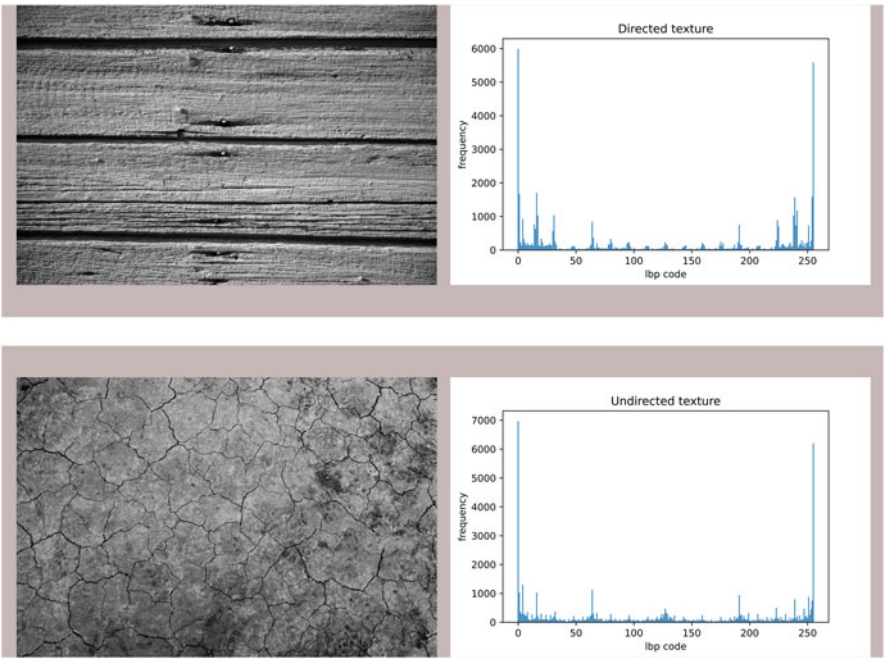



**Fig. 2.12** LBP histograms for two different textures. Differences are clearly visible but since codes are not as easily attributable to directions as, e.g., for Haralick's texture features, they are not as prominent

patterns. A uniform pattern represents the existence of an edge or line passing through the pixel. The remaining 198 non-uniform patterns are mapped to a single code 59.

Patterns for each pixel are primary image features and are not used themselves as texture measure. A texture measure is generated by computing the histogram of the local binary patterns in a texture patch (see again Fig. 2.12). It summarizes different orientations and their frequencies in the patch. It also reduces the influence of noise. The histogram of patterns meets the two conditions for secondary features and may be used for a subsequent classification in feature space.

In its simplest form the texture patch could be computed from the image itself. For adding location information, the image is partitioned into non-overlapping cells (in the original paper the authors used a cell size of 16 × 16 pixels). LBP histograms are computed for each cell and then concatenated to a single feature vector.

The description above is just the basic algorithm for LBP feature computation. With time, various extensions have been presented that improve robustness, speed of computation, direction sensitivity and the like.

**Classes and Functions in Python**
Local binary patterns are computed by the function `local_binary_pattern()` in `skimage.features`. For a given radius `rad` and a number of directions `n_dir` they can be generated from an image in the NumPy array `img` by first computing the pattern for every pixel and then taking the histogram of patterns using the NumPy function `histogram`:

```
import numpy as np
from skimage.features import local_binary_pattern

n_bin= 256 # number of bins for eight directions
n_dir= 8   # number of directions
rad = 2.0 # distance between pixel pairs

lbp   = local_binary_pattern(img, n_dir, rad, method='default')
feature = np.histogram(lbp,n_bin)
```

Other methods (e.g., uniform local binary patterns) can be used by changing the method parameter.

### *2.3.4 Using Texture Measures for Image Classification*

Texture measures such as the ones above have routinely been used to classify a variety of pictures. Not all of these pictures contain a texture in the traditional sense. Several different repetitive patterns may exist in different parts of the picture or the depicted object may not exhibit a repetitive pattern at all. Texture features can still be discriminative, if some orientation preference and/or prominent basic frequency is observed for objects of a class.

An example for texture-based classification is face detection. The frontal view of a face has certainly distinctive characteristics (the eyes, the mouth, the nose) and they are recognizable by intensity changes. For a ROI enclosing a face, these intensity changes are not arbitrarily oriented and positioned relative to each other. Hence, they do contain some kind of regularity that can be picked up by texture features.

Unfortunately, most pictures also contain background with unknown, arbitrary texture. Furthermore, orientation and frequencies of foreground or background patterns will vary in different parts of the image (e.g., the nose is oriented vertically, while the mouth is oriented horizontally). Texture measures are able to represent such different constituents of a texture but the signal will be weaker compared to image patches with just one texture. If regularity is represented by a few intensity changes—which will be the case in our face example—it will degrade the performance of a texture feature as input for a classifier.

It can be counteracted by partitioning the picture into cells before computing texture measures in each cell. The strategy has been applied for computing LBP features but may be applied to other patch-based textures as well. Even if the picture contains different textures at different locations, many cells will contain fewer or just a single texture. The texture signal in the cells will be stronger which improves a subsequent classification.

A secondary feature from cells is created by concatenating the texture features of all cells. It may still create unnecessary feature values in the feature vector if some of the parts of the picture contain irrelevant background textures. However, these features can be detected and removed in a subsequent reduction step that reduces the dimensionality of feature space (see Chap. 3).

## 2.4 Using Edges and Corners

Textures describe objects by their surface interior. Another discriminative characteristic is the object's shape. If the object is two-dimensional, its shape is given by the object boundary in the picture. Examples are the recognition of signatures, text, or digits in scanned documents. Shape features of projected 3-d objects are more difficult to detect. An object's shape is partially defined by the arrangement of surface curvature changes on the object's visible surface. Determining shape features of the complete 3-d object requires projections from all directions that contain different numbers of visible surface edges (sometimes called aspects of the object) together with knowledge about the projection geometry. The number of aspects of an object in a projection is usually finite and small since two projections show the same aspect if they contain the same projected 3-d surface edges. Nonetheless, classification that relies on the complete knowledge about the 3-d object shape is often impossible (because not all views are present) or impractical (because of high computational costs). Classification based on shape features thus often relies on visible shape features and implicitly or explicitly uses separate classification models if different aspects may be visible in different images.

Shape from a 2-d projection of a 3-d object is particularly simple to extract if it is defined by high-curvature locations. Examples are many manufactured objects. Their shape can be approximated by a combination of planar surfaces (i.e., zero curvature) that meet at edges of infinite curvature.

When projected to 2-d, edges of the visible surfaces are often mapped to edges in the picture. These are either silhouette edges caused by the contrast between the object and background at an object edge or illumination edges caused by the change of the angle between illumination source and surface normal at a 3-d object edge (see two examples in Fig. 2.13). Shadow edges and silhouette edges from objects with no or little curvature changes cause edges in the picture as well. Although they are not edges of the 3-d object, they may still represent object characteristics. Detecting these edges in the picture produces a discriminative representation of an object's shape and substantially reduces information in the picture.

**Fig. 2.13** The shape of objects is a discriminative feature. Manufactured objects such as the window on the left often have a shape defined by a few straight lines with a configuration that is recognizable in the projection of the 3-d scene. The shape of natural objects such as the bird on the right may be more difficult to describe but their silhouette is often characteristic for the shape

The mapping is not perfect, however. Silhouette and shadow edges will have varying contrast depending on the background that is hidden by the object. The strength of illumination edges depends on the kind of illumination which adds a non-object-specific component to the feature. Hence, some of the edges may not be detected when post-processing removes pixels for which the edge response is not strong enough (see Fig. 2.14). It does not hamper classification if the classifier is powerful enough to just require a sufficiently high percentage of object edges as feature.

Some extraneous edges from highly specular surfaces may be detected as well. They may mirror other objects or produce specular highlights. Fortunately, most object surfaces are diffuse reflectors so that these extra edges do not occur often. A more critical source of artifactual edges is background clutter. As for textures, some edges may originate from background pixels. The more structured the background is and the larger its share is in the image the less likely can background features be treated as a kind of uniform feature noise. In summary, a certain number of artifacts have to be expected when edges are used as features. Still, many 2-d edges in a picture should be related to the characteristics of the 3-d object.

Edges are computed from intensity gradients. Edge attributes at a pixel location are their location, their orientation, and their local contrast (their "strength"). Only the latter is susceptible to illumination variation.

As for textures, location and orientation of an edge depend on the orientation of the object relative to the camera. Fortunately, in many applications just a few different canonical views occur when pictures of an object are taken. For a car, for instance, these views could be projections from the front, the side, or the back (and in some situations from the top). Furthermore, direction and orientation of projection are often restricted which further restricts variation of apparent shape in the picture.

It is possible to reduce shape information even further if corners are computed instead of edges. We can make similar assumptions for 3-d object edges than for 3-d surfaces and approximate a shape by a set of planar surfaces that meet at straight lines. Again, it is often a good approximation for manufactured objects. In this case, a shape is representable by a set of corners where different straight edges meet. Corners of the visible surface keep their "cornerness" when projected to a 2-d picture.

**Fig. 2.14** Edge images do not always show the silhouette of an object. Weak contrast to the background, shadows, and edges from background clutter result in artifacts. Edge based features thus need to be successful in the presence of such artifacts

For reconstructing the shape information from corners, their location and incident edges are needed. Corner locations can be defined by coordinates in the image coordinate system. Representing information about incident edges requires to attach a list of corners at the other end for each incident edge. Finding incident edges is difficult as it requires edge tracking that guarantees that all edge points of an edge are found.

Even if incident edges for all corners were found, mapping it to secondary features is a problem. While corner locations could be easily mapped to a feature vector, the most natural representation of corners incident to an edge would be a graph or a connectivity matrix. Turning this into a feature vector in some meaningful way is difficult. Thus, primary corner features are seldom used for classification in feature space. They are useful, however, for indicating points of interest in a picture and are used in this capacity for some of the advanced primary features such as SIFT (Sect. 2.6).

### 2.4.1 Edge Detection

Edge detection depends on intensity gradients in a picture. It is estimated through convolution with two difference operators: one in the direction of the $x$-axis and the other in the direction of the $y$-axis. They estimate the partial derivatives in these directions, i.e., the two components of the gradient vector $\nabla f$ of an image $f$ (see Fig. 2.15 for an example).

Since non-zero length gradients may also result from noise, it is assumed that, a priori or by preprocessing, the noise level is substantially lower than the signal level. Edges can then be discriminated from noise by thresholding the gradient length.

Thresholding may break edges in low-contrast regions because the noise level may be locally higher than the signal level. Further attributes to discriminate noise from edges are needed. Hence, edge detection is a two-step process:

- Noise suppression and computation of intensity gradients
- Edge detection

In the early times of computer vision, quite a number of edge detectors have been presented using various criteria to distinguish edges from noise (a survey on the topic is Sun et al. (2022)). We will exemplify the concepts of such detectors by explaining a single such detector, the still often used Canny edge detector, see Canny (1983). Despite its simple edge model with a few parameters it produces quite satisfactory results. Canny edge detection consists of several steps:

- Noise-reduced gradient length computation
- Non-maximum suppression
- Detection of unconditional edge pixels
- Detection of conditional edge pixels

In the original publication, the author suggested to use a convolution with the derivative of a one-dimensional Gaussian along the gradient direction to compute the gradient length. It was shown to produce the strongest response at edges under some assumptions about edges in the depicted scene. As it is computationally expensive, it

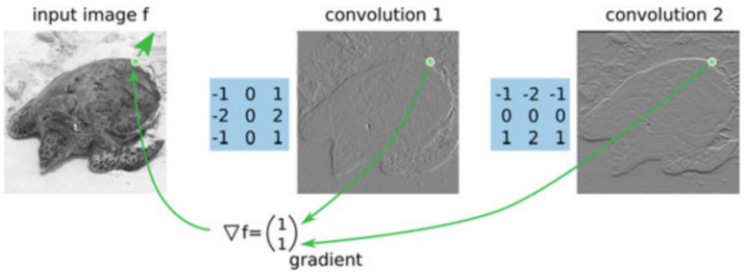



**Fig. 2.15** The result of convolving the input image with two difference operators (here the two kernels of the Sobel operator) is the image gradient. The direction of the gradient is orthogonal to the edge and the length of it gives the intensity change in this direction

is usually replaced by convolution with the partial derivatives of a 2-d Gaussian function.

A 2-d Gaussian with standard deviation $\sigma$ in $x$- and $y$-direction centered at (0,0) is

$$g(x,y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2+y^2}{2\sigma^2}\right). \tag{2.2}$$

Its partial derivatives with respect to $x$ and $y$ are

$$g_x(x,y) = \frac{-x}{2\pi\sigma^4} \exp\left(-\frac{x^2+y^2}{2\sigma^2}\right) = \frac{-x}{\sigma^2} g(x,y) \tag{2.3}$$

and

$$g_y(x,y) = \frac{-y}{\sigma^2} g(x,y). \tag{2.4}$$

The two functions are suitably cropped to fixed-length convolution kernels and normalized. The gradient length is then used as edge indicator.

Taking the two partial derivatives of the Gaussian is equivalent to smoothing the image with a Gaussian function before applying a difference operator. The variance $\sigma^2$ of the Gaussian function is a user-specified parameter. Increasing the variance leads to a better noise reduction as it increases the region over which a weighted average is computed. If the variance is too high, however, the region may cover different edges and their signals fuse leading to artifacts. Selecting a good value for variance thus depends on the scene. A higher variance value is acceptable in a scene with just a few details and few edges compared to one with many edges.

The purpose of subsequent non-maximum suppression is to remove pixels that are not the local maximum along the gradient direction. The local maxima are conveniently found by applying a second-derivative-operator, such as the Laplacian or a Laplacian-of-Gaussian (LoG) filter, and then searching for zero crossings in the result. If values of two neighboring pixels have different signs or if the value of one of them is zero, a zero crossing has been found. Non-maximum-suppression sets gradient length values for all pixels to zero where no zero-crossing has been found.

The remaining pixels are edge candidates. An edge candidate is declared *unconditional edge pixel* if its gradient length exceeds some threshold $t_1$. An edge candidate is declared *conditional edge pixel* if its gradient length exceeds some threshold $t_2$. Unconditional and conditional edge pixels together form the set of pixels from which edges are selected (see Fig. 2.16). This kind of double thresholding is known as *hysteresis thresholding*. Unconditional edge pixels have to be found first. Subsequently, all conditional edge pixels that are connected via a sequence of conditional edge pixels to at least one unconditional edge pixel are selected. Searching among conditional edge pixels is usually done by some connected-component-search on the edge candidates.

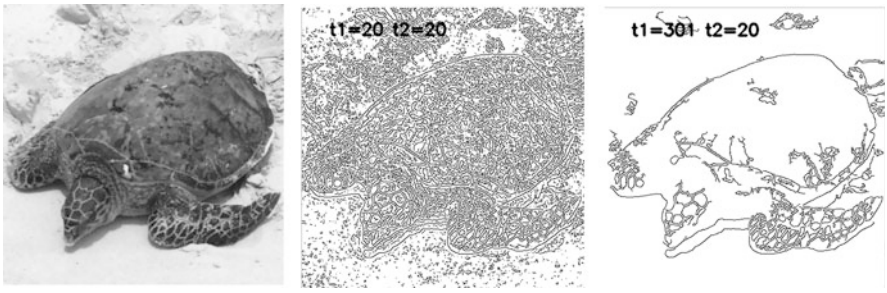



**Fig. 2.16** Canny edge detection with different thresholds for unconditional edge pixels ($t_1$). If this threshold is too low, too many edges are found. For a very high threshold, relevant edges are still found since low-contrast edge pixels are connected to unconditional edge pixels via other conditional edge pixels (defined by threshold $t_2$)

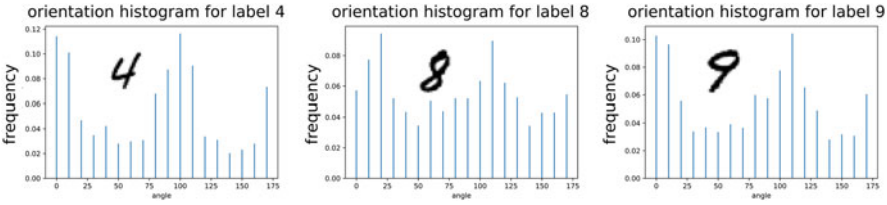



**Fig. 2.17** Class-specific orientation histograms for the digits 4, 8, and 9 from the MNIST data set. Different occurrence frequencies for the digits are clearly visible. The difference is higher for dissimilar digits (e.g., 4 and 9) than for digits with similar shape (e.g., 8 and 9)

The threshold $t_1$ for unconditional edge pixels should be high for being sure that no noise pixel is selected. The threshold $t_2$ may be low since edge candidates with gradient length greater than $t_2$ are selected only if they are connected to an unconditional edge pixel. It even has to be low for not missing edge pixels at locations with weak contrast (e.g., if two regions with similar intensity meet at the edge).

The result is a set of edge pixels where few true edges are missed and few false edges are included. Local orientation and location are the two attributes of edge pixels that characterize the shape of depicted objects. Local orientation is given by the gradient direction of an edge pixel and location by its coordinates. Both are primary image features.

For edge-based classification, we assume that object edges in a picture have characteristic features and that background edges are distributed either randomly or in a way that is similar for all pictures to be classified. Then, creating an *orientation histogram* is a simple way to map local orientation to secondary features of a feature vector. If most of the edges belong to the object, the orientation histogram represents the object's shape (Fig. 2.17 shows an example with class-specific average orientations for digits from the MNIST data). It is a shape representation that neither requires the complete extraction of an object outline nor the removal of edges that do not pertain to the object. However, the more of the object

outline is extracted and the fewer artifactual edges are contained in the image the better becomes the shape representation.

If location information needs to be represented as well, the histogram can be extended to a 3-d histogram $h(\alpha,x,y)$, where $\alpha$ is the binning in different directions and $(x,y)$ is a 2-d binning for location. Since location relative to the image coordinate system is sensitive to small displacements of the camera position, a simpler solution using image partitioning may be chosen. It is the same concept that we already know from computation of LBP features: The picture is partitioned into $n \times n$ cells, orientation histograms are computed for every cell, and then concatenated to form the feature vector.

**Classes and Functions in Python**

Some edge filters are included in `skimage.filters`. The two partial derivatives of the Sobel filter (an older but popular edge filter) in horizontal and vertical direction are computed by `sobel_h()` and `sobel_v()`. The Canny edge detector is contained in `skimage.feature`. The following sequence computes a gradient histogram from a NumPy array `img` at locations where the Canny edge operator has found edges:

```
import numpy as np
from skimage.filters  import sobel_h, sobel_v
from skimage.feature import canny

# edge detection (returns a boolean array)
#   sigma – standard deviation of the Gaussian
#   low/high_threshold – the two thresholds
edges = canny(img, sigma=2.0, low_threshold=20,
              high_threshold=100)

# compute partial derivatives
edges_x = sobel_v(img)
edges_y = sobel_h(img)

# compute gradient angles
grad_angles= 180.*np.arctan2(edges_y,edges_x)/np.pi

# angles count only at edge locations
edge_angles = grad_angles[edges]

# compute orientation histogram
orientation_histogram = (np.histogram(edge_angles,
                         bins=n_bins))[0]
```

### 2.4.2 Corners

Although corners are locations of high curvature in edge pictures, most corner detectors compute them from the original image. Edge computation is already noise-sensitive and computing a derivative along an edge (the curvature) would make the result even more dependent on noise. Furthermore, it is not guaranteed that all edges have been detected and corners may be missed.

Corners in an intensity image are locations where the *aperture problem* does not exist. An aperture problem exists if the appearance in a neighborhood (the aperture of a hypothetical camera) around a pixel does not change if this neighborhood is moved to nearby pixels. An aperture problem in the interior of a homogeneous region exists no matter in which direction the neighborhood is moved. At an edge, the problem exists if the region is moved in the direction of the edge. At corners any translation of the region would cause a change of appearance (see Fig. 2.18).

A function to compute whether an aperture problem exists is the *Harris corner detector* (Harris and Stephens (1988)). It assumes an aperture with radius $r$ around a pixel at $(x,y)$ that contains pixels $(u,v)$. It computes differences of intensities $I$ for a displacement $\Delta x,\Delta y$ at every pixel (this is the *autocorrelation* between these two regions):

$$S(x, y, \Delta x, \Delta y) = \sum_{u,\, v \in nbs(x,\, y)} w(u, v)(I(u + \Delta x, v + \Delta y) - I(u, v))^2. \tag{2.5}$$

We approximate the intensity at the displaced region using a Taylor expansion

$$I(u + \Delta x, v + \Delta y) \approx I(u, v) + \Delta x I_x(u, v) + \Delta y I_y(u, v), \tag{2.6}$$

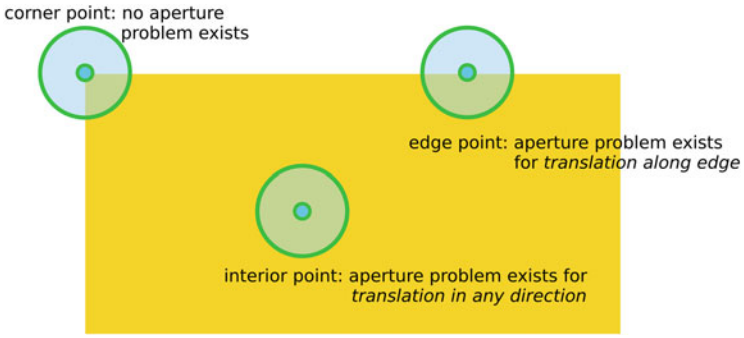



**Fig. 2.18** An image acquired by a camera shows a projection of a scene within a limited region, the aperture (green circles). An aperture problem exists if translation of the camera results in a new image that is indistinguishable from the original image. Corners in the scene are locations where no such aperture problem exists irrespective of the translation direction

where $I_x$ and $I_y$ are the partial derivatives of the intensity $I$ with respect to $x$ and $y$. The difference between all pixels in the neighborhood can then be described by

$$S(x, y, \Delta x, \Delta y) = \sum_{u,\, v \in nbs(x,\, y)} w(u, v) \big(I_x(u, v)\Delta x + I_y(u, v)\Delta y\big)^2. \tag{2.7}$$

Putting this in matrix form and reformulating it, we have

$$S(x,y,\Delta x,\Delta y) = \sum_{u,\, v \in nbs(x,\, y)} w(u,v)(\Delta x\ \Delta y) \begin{pmatrix} I_x^2(u,v) & I_x(u,v)I_y(u,v) \\ I_x(u,v)I_y(u,v) & I_y^2(u,v) \end{pmatrix} \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix} \tag{2.8}$$

$$= \sum_{u,\, v \in nbs(x,\, y)} w(u,v)(\ \Delta x \quad \Delta y\ )\mathbf{H}(x,y) \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix} \tag{2.9}$$

with

$$\mathbf{H}(x,y) = \begin{pmatrix} I_x^2(u,v) & I_x(u,v)I_y(u,v) \\ I_x(u,v)I_y(u,v) & I_y^2(u,v) \end{pmatrix}. \tag{2.10}$$

The vector $(\ \Delta x \quad \Delta y\ )$ and its transpose are independent of the local intensity structure. Hence, the matrix $\mathbf{H}$ represents the local structure at $(x,y)$ in the image. It is called the *structure tensor* at $(x,y)$. Inspection of the two eigenvalues $\lambda_1$ and $\lambda_2$ with $\lambda_1 > \lambda_2$ of $\mathbf{H}$ reveals:

- If $\lambda_1$ and $\lambda_2$ are both low, the autocorrelation function is flat. The aperture problem exists for arbitrary translation directions, and $(x,y)$ is an interior point.
- If $\lambda_1$ is high and $\lambda_2$ is low, the autocorrelation function contains a ridge. The aperture problem exists for a certain translation direction. Hence, $(x,y)$ is an edge point of an edge in this direction.
- If $\lambda_1$ and $\lambda_2$ are high, the autocorrelation function contains a sharp peak. No aperture problem exists and $(x,y)$ is a corner point.

Hence, the lowest of the two eigenvalues is taken as indicator for a corner ($\lambda_2$ is called *cornerness*). Pixels are corner points if the cornerness exceeds some threshold (see Fig. 2.19 for two examples of corner detection). The threshold itself depends on overall and local contrast in the depicted scene and is user-defined.

Corner points are points of interest in a scene and thus useful primary image features. Turning them into secondary features, e.g., by creating a histogram of corner features, is difficult. The only feature readily available is the corner location. More interesting are orientation and number of incident edges at such a corner. This requires that these edges were extracted and that the corners at the other end of each edge are known. It is more difficult than the use of locally available information such as the edge orientation in the previous section.

More efficiently, corners can be taken as points of interest. Local image characteristics, i.e., edges, around them are then used as features. Information

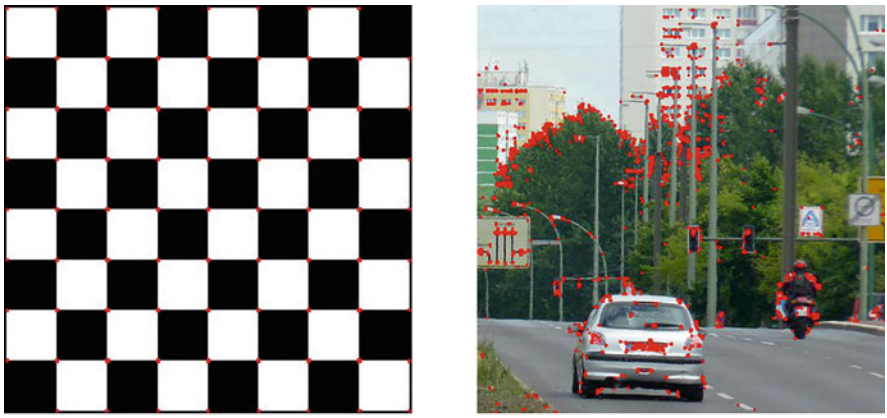



**Fig. 2.19** Corner detection on an ideal and a real image: (**a**) All corners on the checkerboard were detected. (**b**) In the street scene, corners of the car, the motorcycle, and the traffic lights were detected

representation in a scene by corners is more compact than by edges. Thus, the local characteristics need to be richer for corners than for edges in order to reach a similar discriminative power. It can be done by interpreting the points of interest as "visual words" from a vocabulary that is defined by the characteristics at the points of interest. This strategy will be explored further in Sect. 2.6 on SIFT features.

**Classes and Functions in Python**

The Harris corner detector is included in `skimage.feature`. With `corner_harris()` a cornerness array is produced and with `corner_peaks()` coordinates of the peaks on this image can be computed. For a NumPy array `img`, cornerness and corner locations can be computed as follows:

```
from skimage.feature import corner_harris, corner_peaks
corners = corner_harris(img)  # compute cornerness array
peaks = corner_peaks(corners) # compute a list of array indices of
                              # local maxima in the input array
```

The two functions have a number of parameters that control the sensitivity of corner detection and extraction of peaks. Further corner detectors are included in `skimage.features` as well.

## 2.5 HOG Features

Histograms of oriented gradients (HOG, see Dalal and Triggs (2005)) combine concepts from texture and edge features together with a cell decomposition of images to create secondary features. The image is partitioned into cells. Gradient orientation histograms are computed in each cell. Concatenated histograms from all cells make up the feature vector.

The beauty of using HOG features lies in the fact that no edge detection is necessary. If a partition contains object edges, they will be part of the histogram. If it comprises the interior of an object surface, the interior texture will be part of the histogram. A sequence of processing steps is necessary to arrive at high quality features since noise and varying local contrast will influence the appearance of these features as well as their prominence relative to noise and artifacts (see Fig. 2.20):

1. Normalization of the RGB-values
2. Gradient computation and partitioning of the image
3. Computation of gradient histograms
4. Local histogram normalization
5. Concatenation of gradient histograms in a common feature vector

Initial normalization is sometimes omitted as the local normalization in step 4 achieves a similar effect. If used, either an exponential or logarithmic normalization is carried out. The former is based on a heuristic from psychophysics that states

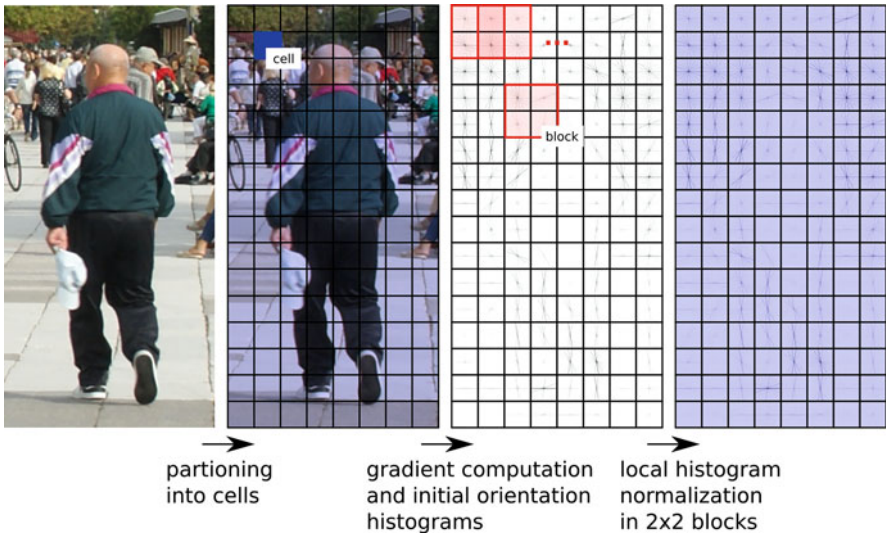



**Fig. 2.20** Major steps for computation of HOG features: Partitioning of the image into $n_c \cdot n_c$ cells, computation of gradients and gradient histograms per cell, normalization of histograms in blocks with $n_b \cdot n_{\mathbf{b}}$ cells. Preferred orientations can be seen in the cells and represent the shape of depicted objects

that perceived intensity increases exponentially (with exponent $< 1$) with true intensity (Steven's power law). The latter states that it increases logarithmically with true intensity (Weber–Fechner law).

Gradient computation is carried out on intensity that is computed from the RGB image or (less often) separately on color channels. It is done by simple difference operators $(-1\ 0\ 1)$ and $(-1\ 0\ 1)^{\mathrm{T}}$ in the directions of $x$- and $y$-axis. No noise reduction is carried out. Noise effects are assumed to be reduced by creating the histograms later. The number of cells $n_{\mathrm{c}} \cdot n_{\mathrm{c}}$ is problem-dependent. Cells should be large enough to capture relevant object detail and small enough to represent location dependencies. In the original paper on the classification of pedestrians, a $128 \times 64$ pixel ROI was partitioned into $8 \times 8$ cells of size $16 \times 8$ pixels. The elongated shape of the cells was intentional as for pedestrians more edges are expected in horizontal direction than in vertical direction. Different cell shapes may be appropriate for other problems.

For computing gradient histograms, the number of bins has to be specified. Usually, the bins capture angles from 0° to 180°. Using a 360° range instead discriminates between inward and outward pointing gradients. It makes sense, if dark objects on bright backgrounds have a different meaning than bright objects on dark background. The assumption is rarely true, not even for object edges caused by shading as this depends on the position of the light source. However, the discrimination of edges from different object coloring would require a 360° range.

The number of bins $n_{\mathrm{bin}}$ is again user-specified. Using too few bins results in loss of discriminative power. Using too many bins increases sensitivity to rotation and to noise artifacts. In the original paper, 9 bins were used for a 180° range.

When creating the histogram, the number of votes that each pixel submits depends on the length of the gradient. It reduces the influence of noise since it is generally assumed that the signal strength is on average much larger than the noise level. In the simplest case, the number of votes is just the gradient length but other linear and non-linear mappings have been used as well. The votes are then assigned to the one bin into which the gradient angle falls and its closest neighbor (see Fig. 2.21). The number of votes assigned to each of these two bins depends on the distance of the gradient angle to the bin boundary.

In the normalization step, a number of overlapping $n_{\mathrm{b}} \cdot n_{\mathrm{b}}$ blocks are created (in the original paper these were $2 \times 2$ blocks). Gradient histograms of the cells in the $n_{\mathrm{b}} \cdot n_{\mathrm{b}}$ blocks are concatenated and normalized. This local normalization reduces the influence of contrast differences within a picture and between pictures.

As a result, $n_{\mathrm{b}} \cdot n_{\mathrm{b}} \cdot n_{\mathrm{bin}}$ features have been computed for each of the $(n_{\mathrm{c}} - n_{\mathrm{b}} + 1)$ $(n_{\mathrm{c}} - n_{\mathrm{b}} + 1)$ cells that are concatenated into a feature vector of length $(n_{\mathrm{c}} - n_{\mathrm{b}} + 1)$ $(n_{\mathrm{c}} - n_{\mathrm{b}} + 1) \cdot n_{\mathrm{b}} \cdot n_{\mathrm{b}} \cdot n_{\mathrm{bin}}$.

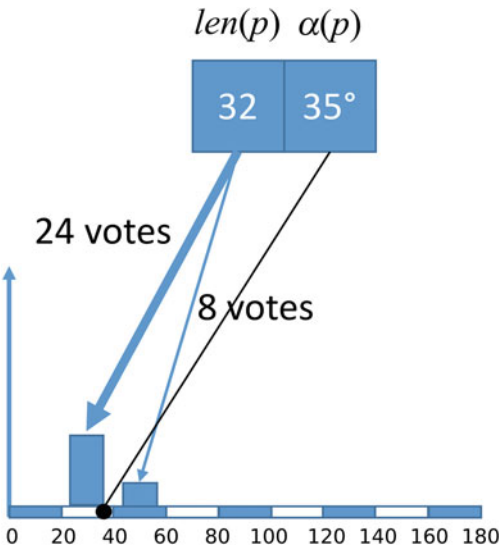



**Fig. 2.21** The number of votes from pixel $p$ for the gradient histogram depends on its length len($p$). Votes are added to the bin, in which $\alpha(p)$ falls and to its closest neighboring bin (in this case the bin 40°–60° since the other neighboring bin 0°–20° is further away from $\alpha(p)$). The number of votes assigned to each of the two bins depends on the distance of $\alpha(p)$ to the respective bin centers

**Classes and Functions in Python**
HOG features can be computed by using the function `hog()` from `skimage.feature`. The following is a sketch to compute HOG features for a NumPy array `img`

```
import numpy as np
from skimage.feature import hog

# compute HOG features and a visualization
# (used, e.g., for Fig 2.21)
#   orientation – number of orientations
#   pixels_per_cell – size of a cell
#   cells_per_block – blocks for normalization
#   visualize – if set, produces the visualization in "hog_image"
#   multichannel – if set, expects a multichannel image (e.g., RGB)
feature, hog_image = hog(img, orientations=9,
                         pixels_per_cell=(32, 32),
                         cells_per_block=(2, 2), visualize=True,
                         multichannel=True)
```

## 2.6 SIFT Features and the Bag-of-Visual-Words Approach

SIFT (scale invariant feature transform, Lowe (1999)) has a lot in common with HOG and was in fact presented earlier than HOG. It relies on local gradient histograms as features as well. The main differences are that features are computed only at

points of interest and that they are scale- and rotation-invariant. Hence, it combines concepts from corner detection with those from HOG.

The motivation behind key-point-based feature computation is that existing redundancies in the picture produce a lot of uninteresting information. Information should only be extracted at points of interest.

The scale of local context at a point of interest will vary. For instance, the scale that determines features at a corner of a homogeneous house wall will be much larger than that of the tip of a small leaf. Both scales will change when the distance between camera and object changes. SIFT extracts features from a local neighborhood with a size that depends on local scale. A large-scale object such as a house will have them extracted from a larger neighborhood than a small-scale object like a leaf.

Feature computation needs to compute the local scale of a point of interest first before computing features at this point. SIFT features can be, as we will see, used for classification but are also powerful in tracking tasks where corresponding points in an image sequence shall be found. Since position and orientation of camera and/or object may change in such sequence, not only scale invariance but also invariance relative to rotation in the local image coordinate system should be guaranteed. Rotational invariance makes sense as well for classification tasks if the rotation angle of the camera in relation to the depicted object is unknown and may vary.

### *2.6.1 SIFT Computation*

Computing SIFT is carried out in a sequence of steps

1. Computation of key points (the points of interest) at a given scale
2. Repetition for a sequence of scales
3. Determination of optimal scale for a key point
4. Key point reduction
5. Computation of a local orientation at each key point
6. Computation of rotation corrected gradient histograms at each key point

Key points at a given scale are blobs, i.e., small regions contrasting to their neighborhood, and corners. Detection of key points is done by convolution with a DoG (Difference of Gaussian) operator. The result of a DoG($\sigma$,$k\sigma$) operator is the difference from convolutions with Gaussian functions with different variances $\sigma$ and $k\sigma$. The result is similar to the convolution with a LoG operator and emphasizes corners and blobs.

In order to reduce noise effects, a series of DoG operators DoG($\sigma$,$k\sigma$), DoG($k\sigma$, $k^2\sigma$),. . . is applied (see Fig. 2.22). The number of different scales—called octaves as they are integer powers of the base $k$—is a user-specified parameter. For each location the highest response is selected. It will be the best compromise between noise reduction and optimal fitting of the filter to image detail.

For finding the optimal resolution, a Gaussian pyramid (a sequence of filtered and down-sampled images) is created. The process is repeated for each scale. Again, the

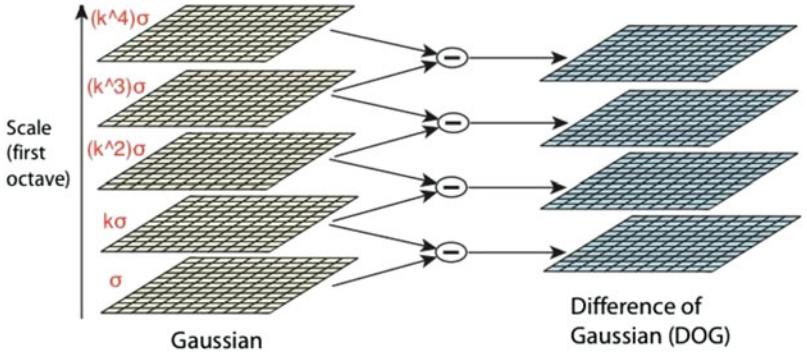



**Fig. 2.22** Key points are computed by a sequence of DoG filters applied to the image

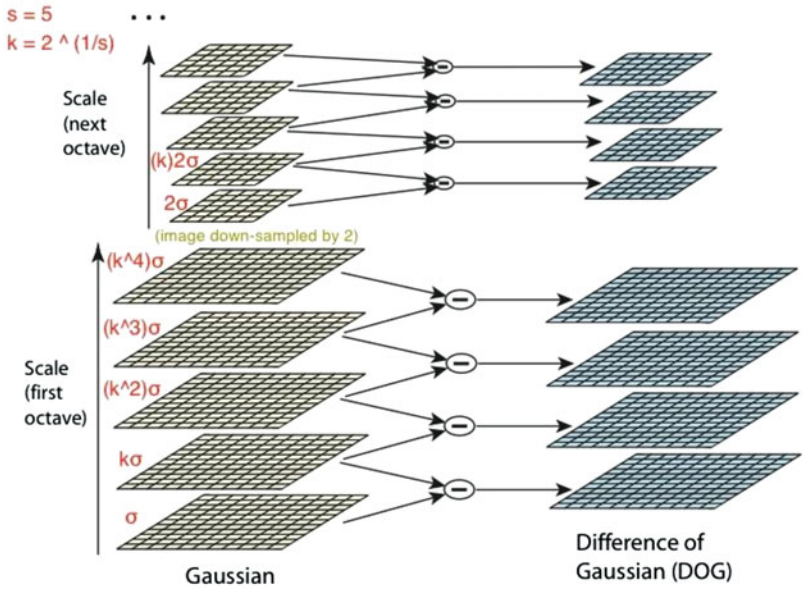



**Fig. 2.23** Key point responses are computed at several levels of resolution (octaves). The maximum response across different resolutions is used as key point strength. The resolution where the response is maximum defines the scale of this key point

maximum response is searched for each sampling resolution and finally reported as strength at this particular key point (see Fig. 2.23). If the key point represents a location with a lot of detail, its resolution needs to be high since neighboring object detail will quickly degrade the filter response. In the opposite case, the optimal resolution will be lower as with every step upward in the Gaussian pyramid noise will be reduced, thus strengthening the signal from the DoG filter. The optimal resolution will be kept and later defines the extent of the local neighborhood from which features are derived.

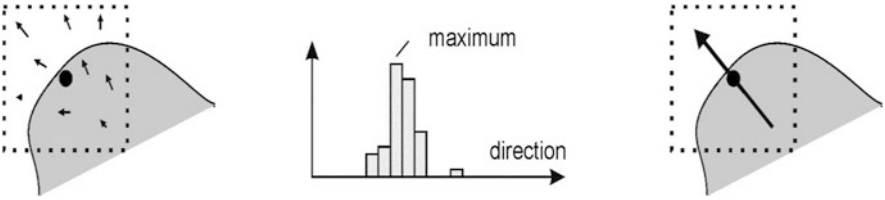



**Fig. 2.24** A local direction for each key point is defined as the orientation that has the maximum number of entries in the local gradient histogram

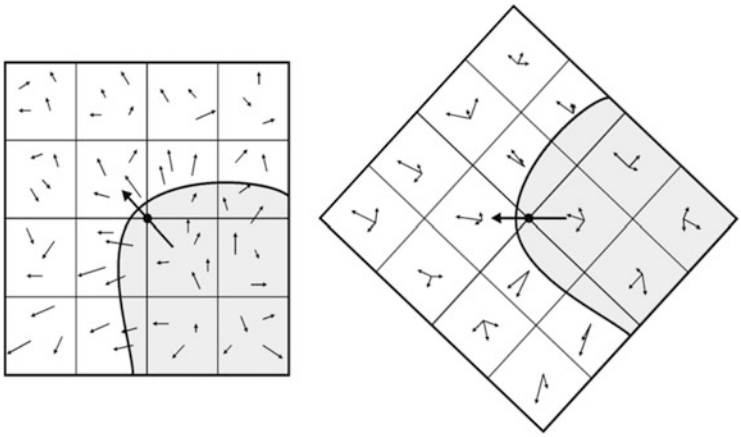


**Fig. 2.25** The local orientation is used to correct gradient directions in the vicinity of the key point. The resulting gradient histogram is then invariant with respect to rotations

In the next step, key points are reduced. Key points with low filter response are removed as well as those with similar gradient direction. The latter indicates locations at edges rather than at corners or blobs.

Next, a local orientation is estimated for every key point to make features rotation-invariant. A local neighborhood is defined for the key point and a gradient histogram is computed. The local orientation is the direction that corresponds to the bin with the most entries in the histogram (see Fig. 2.24).

This direction defines the orientation of a local neighborhood of $4 \times 4$ cells around the key point. The size of the cells depends on the scale computed before. For each cell, an orientation-corrected gradient histogram is computed. The concatenated cell histograms are the feature vector of this key point (see Fig. 2.25 on computation details and Fig. 2.26 for an example).

Key point features are great for determining corresponding points in subsequent frames of a video sequence in an object tracking task but they are not useful as secondary features for a classification task. We could apply the strategy already used for Gabor textures or HOG features and compute average features of the image or an


**Fig. 2.26** Computation of SIFT features from the image (**a**). Image (**b**) shows feature locations, whereas image (**c**) depicts scale and orientation of the features as well

image partitioning. However, the inhomogeneous distribution of key points will be lost although this may be an important characteristic of the depicted object.

### *2.6.2 From SIFT to Secondary Features: Bag of Visual Words*

A SIFT feature vector concatenates all orientation histograms from a $4 \times 4$ neighborhood around a key point. A histogram of occurrences of different feature vectors would produce too many different codes that occur too infrequently. Hence, an intermediate step is necessary that maps feature vectors to a prespecified and much smaller number of codes. These codes are called *visual words* (Sivic and Zisserman (2003)).

The basic idea is similar to that behind the LBP concept in that a picture's meaning is represented by presence and frequency of its visual words. However, the concept of a visual word is more powerful than LBP codes as the definition of visual words depends on a more elaborated feature space and on the frequency of how often such features occur.

As the expression "word" indicates, the concept stems from text analysis. A bag-of-words (BoW) approach assumes that the meaning of a text is defined by the frequency of words occurring in the text. Words are counted only if they are relevant. The position of the words relative to other words, i.e., the grammar, is not used for interpretation. Relevant words are subsets of nouns, verbs, and adjectives. The analysis is much simpler to carry out than one that considers grammar. It can be expected to be less accurate since grammar contributes decisively to the expressiveness of a text. Hence, the BoW approach has been found to be most effective to distinguish rather broad classes of texts (e.g., spam vs. no spam).

The bag-of-visual-words approach (BoVW, sometimes called bag of features, BoF) transfers the concept to image classification. It assumes that an image is made up of a number of visual words that occur at positions of interest. SIFT provides

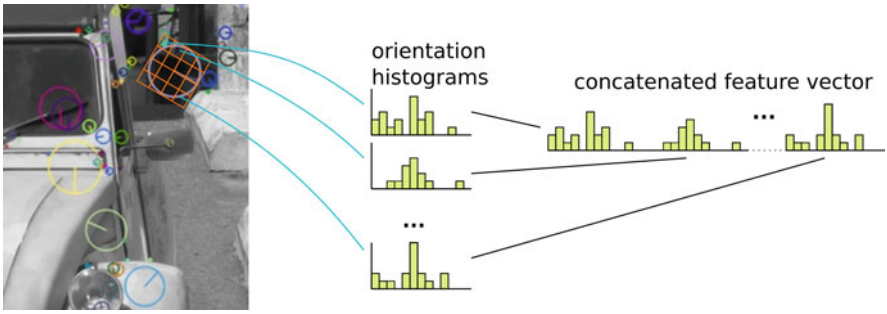



**Fig. 2.27** Every key point has 16 direction-corrected orientation histograms associated with it that are concatenated to form the feature vector for this key point

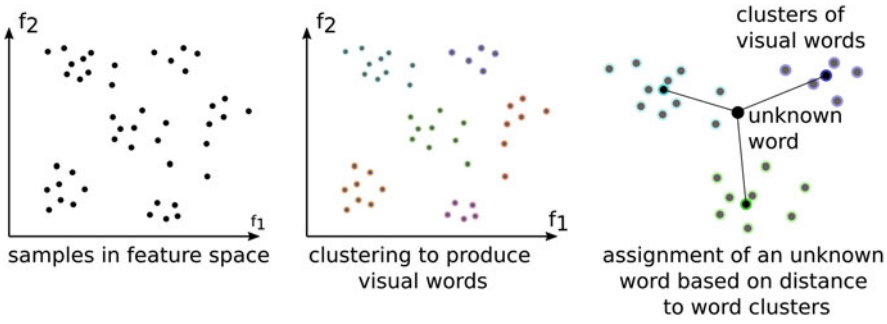



**Fig. 2.28** A dictionary is created from sample patches in images that are mapped to feature space. Clustering in feature space produces the words which are then represented by the respective cluster centers. An unknown word may be assigned to the closest cluster center

these positions of interest by generating features only at key points. Every key point feature vector defines a position in high-dimensional feature space (see Fig. 2.27).

Instead of taking the position as word itself, a *dictionary* is created from features of sample images. Key points with similar features are assumed to have the same meaning. If the meaning is relevant for a classification task, these kinds of key points should occur more often in the sample images. Hence, a feature vector represents a relevant visual word if it is part of a cluster of key points in feature space.

A dictionary of meaningful and relevant visual words can then be created by unsupervised clustering in feature space (see Fig. 2.28). The user specifies the number $k$ of visual words in the cluster. A clustering algorithm (e.g., $k$-means clustering, where $k$ indicates the number of clusters) then produces $k$ cluster centers in feature space. The number of words is application-specific and is often found by experimentation. A dictionary should be large enough to express differences between classes but not too large for not introducing too many meaningless words.

Once the dictionary exists, words of an image to be classified are found by mapping features at key points to entries in the dictionary. It is done by searching

for the closest cluster center to this feature vector, i.e., the most similar visual word in the dictionary. Different assignment strategies can be used:

- In the simplest case, the closest visual word in feature space is assigned to the unknown feature vector.
- Distance to the mean feature vector of the dictionary may be used to make a weighted assignment (the further away from the mean of the dictionary entry the lower the weight).
- The unknown feature vector may not be assigned if it is too far away from the mean vector of the closest dictionary entry.
- The unknown feature vector may be assigned to the *m* closest dictionary entries (with weights depending on distances).

In any case, the result is a (weighted) assignment of key point locations in the image to visual words in the dictionary. Subsequently, a histogram of occurrences of visual words will be built forming the secondary feature vector for the image that is then submitted to a classifier.

Similar to other histogram-based mappings to secondary features, spatial information is lost. It can be partially recovered when SIFT features are computed in image partitions with histograms from all partitions being concatenated.

**Classes and Functions in Python**

A function to compute *SIFT features* can be imported from OpenCV. The class `SIFT_create` has functions to create SIFT features and extract key points. For a gray level image represented by a NumPy array `img`, key points are generated by calling the class constructor. Key points are extracted by calling the function `detect()` of the SIFT object:

```
import numpy as np
from cv2.xfeatures2d import SIFT_create
# SIFT_create sets the attributes of SIFT
#   nFeatures – number of best features to retain
#   nOctaveLayers – number of scales (Default=3)
#   contrastThreshold – minimum contrast to be exeecded
#                       for a key point (Default=0.0133)
#   edgeThreshold – minimum curvature for an edge
                  (Default=10.0)
#   sigma – standard deviation of Gaussian at base scale
sift_features = SIFT_create(nFeatures=100, nOctaveLayers=3,
                            contrastThreshold=0.0133,
                            edgeThreshold=10.0, sigma=1.6)
# extract key points from "img", None is submitted since no
# (optional) mask restricts the search area for key points
keypoints = sift.detect(img, None)
```

(continued)

For computing the *dictionary of visual words* from a set of samples in feature space, the *k*-means clustering algorithm can be used. It is included in OpenCV as well. The following is adopted from an example of the OpenCV documentation to create cluster centers from a 2-d NumPy array `features [n_vectors,n_features]` containing `n_vectors` feature vectors of length `n_features`

```
import cv2
# set number of clusters, i.e., visual words in the dictionary
n_words=1000

# set termination criterion for iterative clustering (in this case
# a combination of fit and number of iterations
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER,
            10, 1.0)

# set number of attempts (the number of times, the clustering is
# repeated; the best result is returned)
attempts=10

# set flags in this case to initialize cluster centers at random in
# feature space
flags = cv2.KMEANS_RANDOM_CENTERS

# apply k-means
compactness,labels,centers = cv2.kmeans(features, n_words,
                                        None,criteria,
                                        attempts, flags)
```

There is an unused parameter in this function to which the value **`None`** is submitted.

The return values are a compactness measure that rates the quality of clusters (the more compact the better), a label for each feature vector indicating the cluster to which it belongs and a list of cluster centers. The latter is used to determine distances of an unknown word to each visual word in order to create the histogram of visual words from an image.

## 2.7 Exercises

### *2.7.1 Programming Project P2.1: Orientation Histograms*

- Extend the wrapper function to read MNIST data from exercise P1.1 (Chap. 1) to partition the data into training and test data.
- Then write a function to carry out the Canny edge detection on the data.

– Implement a function that computes gradients from the data, then use the output from the Canny edge operator as mask to compute an orientation histogram from gradients at edge locations found by the Canny edge detector.
– Compute average features for all labels in the training and test data. Plot the data using functions from matplotlib.pyplot.

Splitting the data into training and test can be done before reshaping the MNIST data by the function `train_test_split()` in the submodule `sklearn.model_selection`. If data and labels were mapped to two arrays data and target (see P1.1), the split into training and test is done by

```
from sklearn.model_selection import train_test_split
...
# n_train is the number of data sets for training (e.g., 60000)
# the remainder is mapped to the test data
X_train, X_test, y_train, y_test = train_test_split(data,
                          target,
                          train_size=n_train)
...
```

For computing the Canny edge detector and the histogram, you may use the Python snippet from Sect. 2.4.1 in a loop over all training and test data sets.

For creating the histograms for each label, you need the function `nonzero()` from NumPy, for example:

```
import numpy as np
 ...
idx=np.nonzero(y_train==cur) # example to find the indices with
                             # label "cur" the training data
```

Look up the function in the NumPy reference manual to understand how to use the entries of `idx` for accessing data with label `cur`. Create separate histograms for each label.

Plotting a histogram `hist` for label `cur` with `nbins` bins can be done by the following sequence of functions from `matplotlib.pyplot`:

```
import matplotlib.pyplot as plt
plt.bar((hist[1])[1:nbins+1], hist[0])# the first element is the
                                      # occurrence frequency, the
                                      # second the bin boundaries
plt.title('Histogram of label '+str(cur)) # title of the plot
plt.xlabel('angle')                       # axes labels
plt.ylabel('frequency')
plt.show()                                # show pending plot
```

### 2.7.2 *Programming Project P2.2: A Cluster Measure in Feature Space*

Ideal features would place samples of the same class close together in feature space and would have samples from different classes far apart from each other. A simple, albeit rather rough estimate for this quality is the ratio of average variability between classes to that within classes.

Write a function that estimates and returns this ratio. Use it to assess the quality of the features computed for the MNIST data from project P2.1.

Computing the average variance for some class consists of:

- Computation of a mean feature vector for each class. To access the feature vector for a certain class with label `cur`, the NumPy function `nonzero()` (see project P2.1) can be used. For computing the mean, the NumPy function `mean()` can be used. As it allows to compute the mean over selected axes, it does not require a loop (look it up in the NumPy manual, if you are not familiar with this function).
- The mean feature vector over all classes is computed in a similar way but now by summing over all feature vectors.
- Computing the mean variance for a class consists of two steps:
  - First, the variance for each feature is computed by subtracting the class mean from each feature and taking the square. The variance is computed by summing over all `n_samples` samples of this class and dividing by `(n_samples-1)`. Use the broadcasting property for NumPy arrays to avoid loops for the computation as it speeds up this operation substantially. If you are not familiar with broadcasting, you should look this up in the NumPy manual. We will use broadcasting several times in the programming projects for efficient programming.
  - Finally, a mean within-class variance is computed by averaging the variances of the features in the mean variance vector features.

The within-class variability is then computed as weighted average of the mean within-class variances of all classes.

Computing a between-class variability is similar. First, a mean feature vector for all samples is computed. Then, the variance to the mean feature vectors for the different classes is computed. The different feature variances in this vector are then averaged which gives the between-class variability.

The ratio between the two variabilities shall be returned.

### 2.7.3 Exercise Questions

- What is the difference between primary and secondary features?
- Why is the detection of handwritten digits a good example of using the pixel values themselves as secondary features?
- Why can Haralick's texture measures capture features representing correlations over larger neighborhoods, even if the co-occurrence matrices are only formed over short distances?
- Why is it possible to represent shape properties of an object in features without segmenting the object? How can this be done and under what conditions does it work?
- Which texture properties are computed by a Gabor filter bank? Why is it useful to average these properties over the entire region of interest?
- What is the advantage of having the larger of the two Canny Edge detector thresholds high? What would be a possible disadvantage if it is too high?
- What is the advantage of partitioning the region of interest into sub-images before feature calculation?
- Why should gradient lengths be normalized as a texture feature? For HOG, why is this normalization not necessary during preprocessing?
- What are the differences between SIFT and HOG? Explain whether or not HOG is suited to track objects in a video sequence.
- Why is SIFT more suitable for classification if it is combined with a BoVW strategy?
- How does a BoVW approach arrive at the secondary features suitable for classification?

## References

Ahonen, T., Hadid, A., & Pietikäinen, M. (2004). Face recognition with local binary patterns. In *8th European Conference on Computer Vision, ECCV 2004. Part I* (Vol. 8, pp. 469–481). Springer, Berlin Heidelberg.

Armi, L., & Fekri-Ershad, S. (2019). Texture image analysis and texture classification methods—A review. arXiv preprint arXiv:1904.06554.

Canny, J. F. (1983). *Finding edges and lines in images*. Massachusetts Inst of Tech Cambridge Artificial Intelligence Lab.

Dalal, N., & Triggs, B. (2005). Histograms of oriented gradients for human detection. In *2005 IEEE Conference on Computer Vision and Pattern Recognition (CVPR2005)* (Vol. 1, pp. 886–893).

Deng, L. (2012). The MNIST database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Processing Magazine, 29*(6), 141–142.

Haralick, R. M. (1979). Statistical and structural approaches to texture. *Proceedings of the IEEE, 67*(5), 786–804.

Harris, C., & Stephens, M. (1988, August). A combined corner and edge detector. *Alvey Vision Conference, 15*(50), 10–5244.

Humeau-Heurtier, A. (2019). Texture feature extraction methods: A survey. *IEEE Access, 7*, 8975–9000.

Lowe, D. G. (1999). Object recognition from local scale-invariant features. In *IEEE International Conference on Computer Vision (ICCV 1999)* (Vol. 2, pp. 1150–1157).

Pietikäinen, M. (2010). Local binary patterns. *Scholarpedia, 5*(3), 9775.

Roslan, R., & Jamil, N. (2012). Texture feature extraction using 2-D Gabor Filters. In *2012 IEEE Intl Symp on Computer Applications and Industrial Electronics (ISCAIE)* (pp. 173–178).

Sivic, J., & Zisserman, A. (2003). Video Google: A text retrieval approach to object matching in videos. In *IEEE International Conference on Computer Vision (ICCV2003)* (Vol. 3, pp. 1470–1470).

Sun, R., Lei, T., Chen, Q., Wang, Z., Du, X., Zhao, W., & Nandi, A. K. (2022). Survey of image edge detection. *Frontiers in Signal Processing, 2*, 826967.

# Chapter 3
# Feature Reduction

**Abstract** Image information has already been reduced when constructing the feature vector from primary image features. The reduction is based on heuristic rules about what is necessary to classify the image. Feature reduction, discussed in this chapter, reduces the dimension of feature space. It is based on definitions about redundancy of a feature and its relevance to the classification task. Dimension reduction of feature space is often necessary because the initial information reduction from feature extraction is not based on such kind of definition. Hence, often rather too many features are selected for not removing necessary information before classification.

Unsupervised feature reduction reduces feature space based on sample distributions without knowledge of sample labels. Supervised feature reduction requires labeled samples. Examples for the two strategies will be presented preceded by a definition of what is meant by redundancy and relevance into the two classes. Advantages and disadvantages of the different redundancy and relevance definitions will be discussed.

The feature vector extracted from an image still contains redundant information since secondary features are selected based on rules that are often but not always true (e.g., "just edge information is relevant"). For not accidentally removing relevant information rather too many than too few features are often selected for the feature vector. Feature reduction that we will discuss in this chapter will use the secondary feature vector and various definitions of redundancy and relevance to reduce information further before submitting a feature vector to a classifier (see Fig. 3.1 for two examples). As this text is meant to be a primer on image classification, just a subset of popular techniques will be presented. Reviews on the topic that treat the subject in more detail can be found in Khalid et al. (2014) or Zebari et al. (2020).

Redundancy and relevance are two different concepts. Information is redundant if it can be completely explained by other information. Hence, a reversible mapping exists to remove and reconstruct redundant information. In order to remove redundancy, the kind of mapping needs to be defined first. As the number of possible mappings is indefinitely large, it is usually restricted to redundancy with respect to a specific set of mapping functions.

© The Author(s), under exclusive license to Springer Nature Singapore Pte Ltd. 2024

K. D. Toennies, *An Introduction to Image Classification*,
https://doi.org/10.1007/978-981-99-7882-3_3

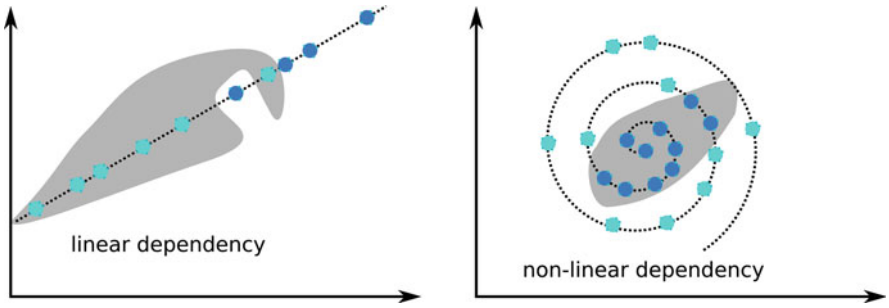



**Fig. 3.1** Finding and removing redundant information requires to find a mapping on a lower dimensional feature space that completely explains the training samples. The optimal feature space may be a linear or non-linear embedding in the original space. The mapping is usually a family of parameterizable functions where optimal parameters are searched given labeled samples. Redundancy refers to the training samples only. It may be possible that the sample distribution of complete population does not contain such redundance and feature reduction based on training samples makes successful classification impossible

A set of functions for which an analytical solution exists is that of linear mappings applied to the features. Features are linearly correlated if they can be expressed by a linear combination of each other. Detection and removal of linear correlations among features requires a representative set of samples $\mathbf{x}_i$ with features $(x_{i,1},\ldots x_{i,N})$ but no knowledge about their associated labels $y_i$. This kind of feature reduction is called *unsupervised*.

Removal of redundant information seems to be unproblematic as just superfluous data is removed and a perfect restoration of the original data is possible. It has to be taken with a grain of salt, however. Using samples to detect redundancy means that it is established for these samples only. It may well be possible that some relevant characteristics of the unknown data are not captured by the samples. Hence, sample-driven redundancy removal may affect the classification ability from a reduced set of features. The chance of this oversimplification increases with the complexity of the mapping (linear vs. non-linear) and with the sparsity of sample distributions in feature space (see again Fig. 3.1).

Opposed to redundant information, relevant information is necessary to fulfill a certain task which in our case is classification. Removal of irrelevant information makes the mapping irreversible. Determining relevancy always requires a definition of this task first. In its simplest version the definition does not require more than a sample data set and can be carried out in an unsupervised fashion. Lossy compression is an example from image processing for this kind where distinguishability to a human observer is used as criterion. We will see that a similar strategy can also be used for feature reduction.

A more efficient way to remove irrelevant features is to rate the ability to discriminate different image classes based on the reduced set of features (see Fig. 3.2). In this case, class labels need to be known for all samples in order to

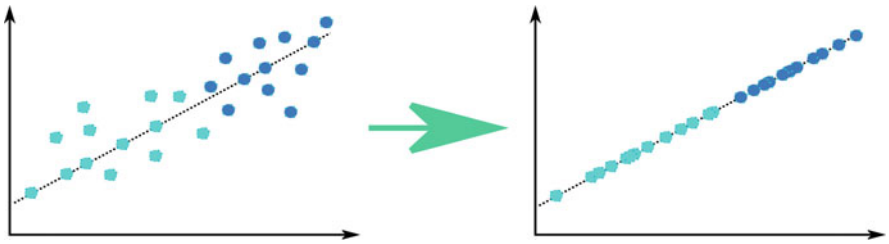


**Fig. 3.2** Irrelevant information is defined based on the ability to classify the samples. If features can be removed without compromising this goal, the resulting lower-dimensional features space may simplify classification as it will be more densely populated. As for the removal of redundancy, establishing relevancy depends on sample images. The number of trainings samples has to be high enough to allow a robust prediction of relevancy for the complete population of samples

determine whether the removal of a feature impedes the classification ability. This kind of feature reduction is called *supervised*.

Before going into detail about supervised and unsupervised feature reduction, the subject of "representative set of samples" should be discussed. Any feature reduction attempts to find a generally applicable mapping from a high-dimensional feature space to a low-dimensional space. Generally applicable means that the properties of this mapping apply to the population of all samples that may occur in the context of the intended application (e.g., the recognition of pedestrians in images).

"All samples" usually translates to infinitely many samples that surely never can be acquired and treated. Hence, a finite subset is taken to be representative for the unknown population. If the behavior of a method developed on this subset should result in the expected behavior for unseen samples, the subset needs to be unbiased. Unbiased samples set should share all properties of the population in the same manner.

Just what properties are relevant is usually unknown. The only way to ensure unbiasedness is to rely on an unbiased selection of samples. This again can be difficult. A developer may fail to understand what makes up a population and may unconsciously make biased choices. For the pedestrian recognition example mentioned above, this could mean that variations due to different lighting, different weather, or different genders of the pedestrians were neglected when choosing samples. A biased sample database is not only a reason for poor performance for feature reduction. Classification itself may be affected as well. Hence, recognizing and removing bias in the sample database is an important subject in itself. We will not treat it in this introductory text but will stress the importance every time when the subject comes up. Surveys on the topic that treat the different sources of bias in depth are Mehrabi et al. (2021) and Fabbrizzi et al. (2022).

## 3.1 Unsupervised Feature Reduction

If dimension reduction of feature space is to be carried out without knowing class labels, it needs to be assured that removed dimensions do not carry indispensable information. Using feature variance as criterion is a means to make this decision. Feature removal based on variance may happen in the original feature space or in a transformed space that maximizes dimension reduction for a given variance threshold.

### *3.1.1 Selection Based on Feature Variance*

A feature is clearly redundant for a classification task when all samples have the same feature value. If it varies only little from this value, it is probably not a relevant feature. The distinction between redundancy and relevance is important here. Relevancy based on variance is just a heuristic assumption. It is based on the observation that most features will contain a certain amount of noise and artifactual variation. Relevant differences between feature values of two samples should exceed this noise level. Redundancy, however, is a well-defined property.

Noise is caused by measurement errors in the original image. For instance, the measurement of gradient angles in HOG or SIFT features is affected by image noise which will lead to unwanted orientation variation. Other variation may be caused by image artifacts. In the pedestrian example above, such artifacts may be due to intensity gradients in the image background.

For a successful classification, variation due to different object class membership should be higher on average than variation due to noise and artifacts. Being an average, relevant variation in specific cases may still be in this range. However, at this level the different sources of variation are indistinguishable. Hence, non-zero variance features may be removed if the variance is below this relevance level (see Fig. 3.3).

Unfortunately, the relevance level is unknown. Fixing it at a specific threshold is always a matter of experimentation. A poorly chosen relevance level may lead to

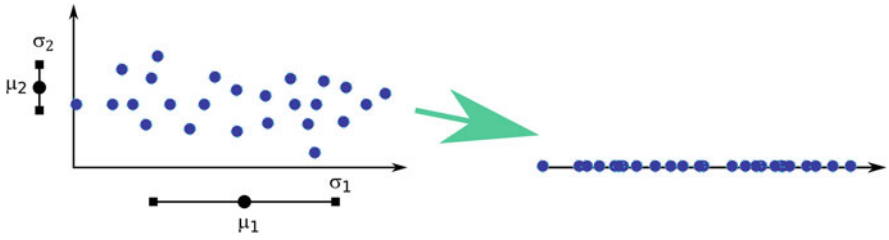



**Fig. 3.3** A simple way to define relevancy is to require a threshold where noise is higher than feature variation and to remove features with standard deviation below this threshold

poor performance of a later classification due to missing features (if it is too high) or due to a sparsely populated feature space with too many dimensions (if it is too low).

Variance is estimated from samples. Given a set of sample feature vectors $\mathbf{x}_i$, $i = 1,...M$ with each feature vector $\mathbf{x}_i = (x_{i,1} \ldots x_{i,N})$ having $N$ features, estimated expected values are

$$\mu_j = \frac{1}{M} \sum_{i=1}^{M} x_{i,j}. \tag{3.1}$$

The vector of expected values is $\boldsymbol{\mu} = (\mu_1 \ \ldots \ \mu_N)^{\mathrm{T}}$. The expected values are then used to estimate the variance $\sigma^2$ along each axis:

$$\sigma_j^2 = \frac{1}{M-1} \sum_{i=1}^{M} \left(x_{i,j} - \mu_j\right)^2. \tag{3.2}$$

Features are removed if their associated variance falls below a relevance threshold $r$. As stated above, choosing a relevance threshold may be difficult. It is even possible that different axes have different relevance thresholds because the feature vector consists of different information types. For instance, inter-class, intra-class variation, and variation due to noise can be very different for image intensity or gradient orientation features.

### 3.1.2 Principal Component Analysis

Feature removal in the original feature space based on variance is not the most efficient way to determine redundant features. Imagine two features $f_1$ and $f_2$ that vary for different samples but which have always the same value $f_1 = f_2$. A lot of variation would be measured along the two feature axes although the features are clearly redundant.

In this example, the features are linearly correlated but it does not show up in the individual variance values for $f_1$ and $f_2$. *Principal component analysis* (PCA) is a means to detect and remove such linear correlation.

Linear correlation is determined by computing the covariance between any two features of the feature vector. If two dissimilar features have a non-zero covariance, part of the variation of one of them depends on the variation of the other. PCA will first estimate covariances between all pairs of features and then find a transformation of feature axes so that the covariance between any two transformed features is zero.

Covariances between pairs of features of an $N$-dimensional feature vector are represented by an $N \times N$ covariance matrix $\boldsymbol{\Sigma}$. First, the expected values are estimated from samples as in Eq. 3.1 and new feature vectors $\widehat{\mathbf{x}}$ are generated by subtracting the vector of expected values:

$$\widehat{\mathbf{x}}_i = \mathbf{x}_i - \boldsymbol{\mu}. \tag{3.3}$$

Then, the covariance matrix is estimated by

$$\boldsymbol{\Sigma} = \frac{1}{M} \sum_{i=1}^{M} \widehat{\mathbf{x}}_i (\widehat{\mathbf{x}}_i)^{\mathrm{T}}. \tag{3.4}$$

The matrix contains the variances for each feature on its diagonal (compare the values of Eq. 3.2 for the diagonal with those computed by Eq. 3.4). All off-diagonal elements are covariances.

The aim is to find a transformation that produces new features where all off-diagonal elements of the covariance matrix are zero. It is done by an eigen decomposition of $\boldsymbol{\Sigma}$ which delivers a set of $N$ eigenvectors and $N$ associated eigenvalues. Eigenvectors have normal length. Two different eigenvectors are orthogonal to each other. We now arrange all $N$ eigenvectors in a $N \times N$ eigenvector matrix $\mathbf{E}$. If we insert the corresponding eigenvalues in a diagonal matrix $\boldsymbol{\Lambda}$, the following is true:

$$\mathbf{E}\boldsymbol{\Sigma} = \boldsymbol{\Lambda}\mathbf{E}. \tag{3.5}$$

Since $\mathbf{E}$ is orthonormal (length of each column or row of the matrix is 1 and scalar product between any two different eigenvectors in the matrix is 0), the inverse $\mathbf{E}^{-1}$ is just its transpose $\mathbf{E}^{\mathrm{T}}$. Hence, we can make the following transformation:

$$\mathbf{E}\boldsymbol{\Sigma} = \boldsymbol{\Lambda}\mathbf{E} \quad \leftrightarrow \quad \mathbf{E}\boldsymbol{\Sigma}\mathbf{E}^{\mathbf{T}} = \boldsymbol{\Lambda}. \tag{3.6}$$

Now, we replace the covariance matrix $\boldsymbol{\Sigma}$ by Eq. 3.4 and arrive at

$$\mathbf{E}\boldsymbol{\Sigma}\mathbf{E}^{\mathbf{T}} = \mathbf{E}\left(\frac{1}{M}\sum_{i=1}^{M} \widehat{\mathbf{x}}_i \widehat{\mathbf{x}}_i^{\mathrm{T}}\right)\mathbf{E}^{\mathbf{T}} = \frac{1}{M}\sum_{i=1}^{M} \mathbf{E}\widehat{\mathbf{x}}_i [\mathbf{E}\widehat{\mathbf{x}}_i]^{\mathrm{T}} = \boldsymbol{\Lambda}. \tag{3.7}$$

Since $\mathbf{E}$ is an orthonormal matrix, $\mathbf{x}_i^{\mathrm{new}} = \mathbf{E}\widehat{\mathbf{x}}_i$ is just a projection of the features $\widehat{\mathbf{x}}_i$ on a new basis that is formed by the eigenvectors in $\mathbf{E}$. Because of Eq. 3.4, the left-hand side term in Eq. 3.7 computes the covariance matrix of the transformed features $\mathbf{x}_i^{\mathrm{new}}$ which is given by $\boldsymbol{\Lambda}$. This matrix is linearly decorrelated and the eigenvalues along the diagonal are the variances of the transformed features in $\mathbf{x}_i^{\mathrm{new}}$ (see Fig. 3.4).

Feature reduction uses a truncated mapping $\mathbf{E}'\widehat{\mathbf{x}}_i$ where rows of $\mathbf{E}$ are removed if one of the following conditions is met:

- The corresponding eigenvalue is 0. In this case the feature is redundant.
- The corresponding eigenvalue is lower than a relevancy threshold. In this case, the feature variance is deemed to be too low for discriminating inter-class variation from variation due to noise and artifacts.

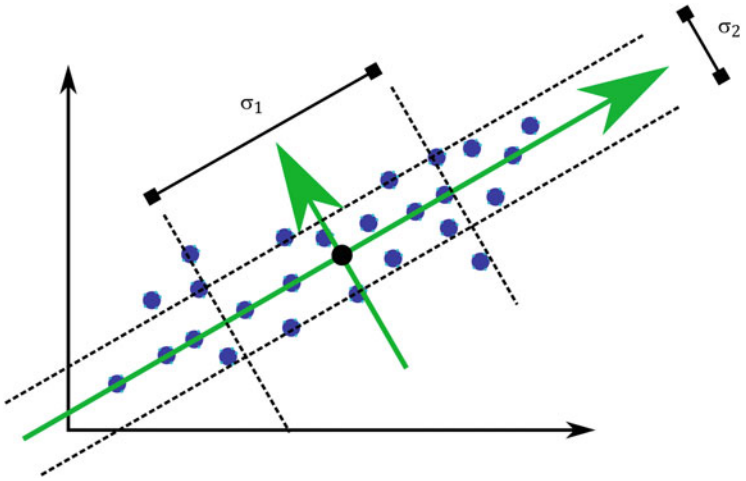



**Fig. 3.4** The PCA computes eigenvectors of the covariance matrix of feature vectors. These eigenvectors are the coordinate axes of a transformed system in which features are no longer linearly correlated. The eigenvalue $\lambda_i$ that corresponds to an eigenvector $\mathbf{e}_i$ is the variance $\sigma_i^2$ along this axis (the standard deviation $\sigma_i$ is then the square root of the eigenvalue)

Projection on the remaining $k$ rows in $\mathbf{E}'$ reduces the number of features from $N$ to $k$.

For PCA-based feature reduction, several heuristics have been proposed to choose relevant features given that eigenvalues are ordered:

- Choose features with the highest eigenvalues up to the point that $x$ % (e.g., 95%) of the total variance is covered.
- Choose the first $k$ highest features.
- Treat the sequence of accumulated variances as points of a continuous function over a line from (0,0) to ($N$,max_variance) and choose the maximum over this line.

All criteria are purely heuristic and do not depend on the underlying characteristics of the data with respect to the classification goal. The argumentation regarding the proper choice of a relevancy threshold from the previous section is true here as well. Other approaches turn the reduction into a supervised method and treat $k$ as trainable hyperparameter, e.g., Malhi and Gao (2004).

**Classes and Functions in Python**
Various functions for feature selection are contained in the submodule `sklearn.feature-selection` of SciKit-learn. The function `VarianceThreshold()` is an implementation of the function described here.

(continued)

Principal component analysis is a class in `sklearn.decompositions`. It may be used to reduce the dimensionality of features `features` as follows:

```
from sklearn.decompositions import PCA
N_COMPONENTS = 100

# create PCA object (the reduced number of components N_COMPONENTS
# is submitted to the constructor
pca=PCA(n_components=N_COMPONENTS)

# fit to features from the training data set
pca.fit(features)

# apply the transformation and return the N_COMPONENTS first
# features
red_features=pca.transform(features)
```

## 3.2 Supervised Feature Reduction

Supervised feature reduction uses classification quality in a dimension-reduced feature space as decision criterion. This can be done directly by computing classification accuracy on the sample database with different features or indirectly by searching for a feature space that optimizes certain characteristics that are assumed to simplify the classification task. In the first case, the result is always connected to a specific classification method. This is not so for the second case. The characteristics used for reduction of features are just loosely related to the performance of a classifier. Hence, a superior performance of a specific classifier on the reduced features is not guaranteed.

### *3.2.1 Forward and Backward Feature Selection*

Feature selection does not use any kind of transformation of feature space although it may be carried out in already transformed feature space (e.g., by PCA). Feature selection uses a non-optimal heuristic to select a subset of all features based on performance of some classifier.

*Forward feature selection* starts classification with a single feature and then keeps adding features until a sufficient classification quality has been reached. For finding the first feature, all features are used as single-feature-classifier and the best-performing feature is selected. It is then combined with a feature from the set of remaining features for training a double-feature-classifier. Again, the best-performing feature is added. The greedy selection procedure is continued until a sufficient classifier performance is reached.

Besides requiring a classifier for the feature selection criterion, forward feature selection is time-consuming. As many classifiers have to be trained and tested for every feature selection as there are unused features to select from. Efficient implementations make use of properties of the classifier (Fauvel et al. (2015)) or predict performance gain by newly added features (Reif and Shafait (2014)). However, selecting always the best-performing feature does not guarantee an optimal choice since possible dependencies between features are not considered. Even for linearly decorrelated data such dependencies may exist.

A strategy with similar properties is *backward feature selection*. In this case, we initially train a classifier with the complete set of features and remove features step by step. It will be always the feature that causes the highest performance gain or the lowest performance loss until performance falls below a predefined threshold. It may come as a surprise that a performance gain is expected when removing features. After all, it removes information for the classifier. However, too many features may be just as bad as too few features for classifier training (this is called *curse of dimensionality*). The classifier training needs to determine a function in feature space that maps samples to classes on the base of samples with known class membership. For a high-dimensional feature space with few samples, it may be difficult to robustly find such a function the generalizes well. Hence, removing features may improve classifier performance.

### 3.2.2 Linear Discriminant Analysis

Feature selections based on classification accuracy may not generate an optimal set of features because different features are correlated. *Linear discriminant analysis* (LDA) produces a feature transformation that removes some of this correlation. LDA has been used for feature reduction (an example for feature reduction is in Song et al. (2010)) as well as for unsupervised classification. In feature reduction, sample variance after feature transformation is used to decide which of the transformed features to remove.

For carrying out LDA, an optimal distribution of features in a transformed feature space has to be defined based on class membership of training samples. It requires a transformation of the coordinate system similar to PCA before irrelevant features are removed in the transformed space.

The goal of feature reduction is a distribution where samples of the same class are close together and samples from different classes are far from each other. The first can be characterized by the average *within-class scatter* of feature vectors and the second by the *between-class scatter* between classes. In order to compute within-class scatter, feature vectors $\mathbf{x}_i^c$ of length $N$ from samples of class with $y_i = c$ are corrected for their expected value (similar to Eq. 3.3 but now with expected value computed only from samples of class $c$) to form new features $\widehat{\mathbf{x}}_i^c$. All $M_c$ feature vectors $\widehat{\mathbf{x}}_i^c$ are written as rows in a common matrix $\mathbf{M}_c$ of size $N \times C$, where $C$ is the number of classes.

The covariance matrix for a class $c$ is then

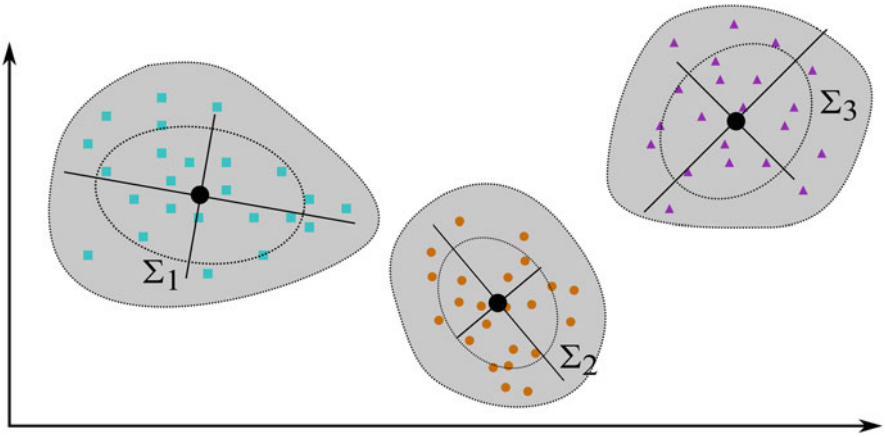



**Fig. 3.5** The within-class scatter $\mathbf{\Sigma}_c$ is computed separately for each class $c$ and then averaged over all classes

$$\mathbf{\Sigma}_c = \frac{1}{M_c - 1}\mathbf{M}_c\mathbf{M}_c^{\mathrm{T}}. \tag{3.8}$$

As in PCA, eigenvectors of $\mathbf{\Sigma}_c$ represent axes of decorrelated features. The corresponding eigenvalues are the variances along these axes. Hence, $\mathbf{\Sigma}_c$ is a representation of variation within a class $c$ (hence, the term within-class scatter, see also Fig. 3.5).

It is computed for all classes $C$ and then averaged:

$$\mathbf{S}_w = \sum_{c=1}^{C} p_c \mathbf{\Sigma}_c. \tag{3.9}$$

The weights $p_c$ are the a priori probabilities for samples to belong to class $c$. They are estimated as share of samples from this class in the training data.

Since each of the summed covariance matrices has its own, independent set of eigenvectors, the average within-class scatter $\mathbf{S}_w$ is not a covariance matrix. It still captures variation along an average of axes from the covariance matrices of the different classes and may hence be used to represent average within-class scatter.

Between-class scatter is computed as covariance between expected value vectors of the $C$ classes (see Fig. 3.6). Given expected value vectors $\boldsymbol{\mu}_c$ for each class $c$, the expected value vector for the distribution of cluster centers is

$$\boldsymbol{\mu} = \sum_{c=1}^{C} p_c \boldsymbol{\mu}_c. \tag{3.10}$$

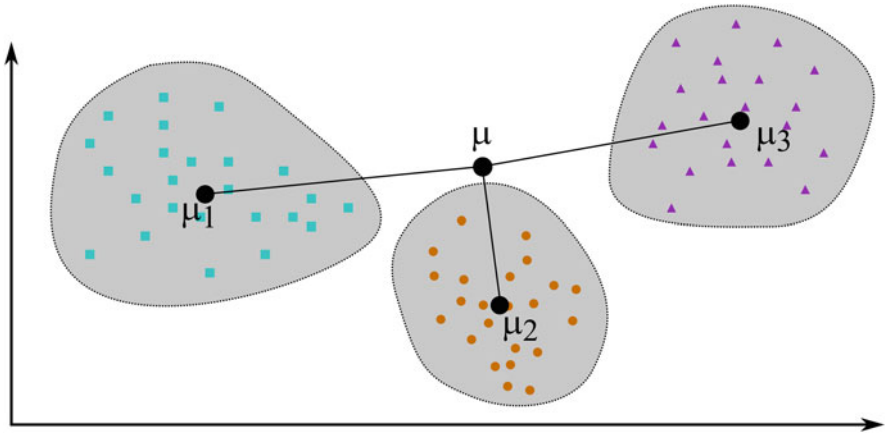



**Fig. 3.6** The between-class scatter is defined as covariance of the expected value vectors $\boldsymbol{\mu}_i$ for each class $i$ with regard to the expected value vector $\boldsymbol{\mu}$ of all feature vectors

The expected values from the class clusters are weighted by their a priori probability in order to deliver the correct expected value of all samples.

Finally, the between-class scatter matrix $\mathbf{S}_b$ is computed as covariance of weighted cluster means:

$$\mathbf{S}_b = \sum_{c=1}^{C} p_c \left[ (\boldsymbol{\mu}_c - \boldsymbol{\mu})(\boldsymbol{\mu}_c - \boldsymbol{\mu})^{\mathrm{T}} \right]. \tag{3.11}$$

As the goal is to maximize between-class scatter and to minimize within-class-scatter, a matrix $\mathbf{S}$ that combines the two matrices needs to invert one of them:

$$\mathbf{S} = \mathbf{S}_b \mathbf{S}_w^{-1}. \tag{3.12}$$

A transformation of feature space that maximizes the trace $tr(\mathbf{S})$ delivers axes along which between-class scatter is maximized and within-class scatter is minimized. An eigen decomposition of $\mathbf{S}$ delivers this transformation because the resulting diagonal matrix concentrates all variation in the diagonal values of the matrix. Summing up the diagonal for computing the trace is thus maximum. Feature selection in the transformed coordinate system may then remove axes with variances that are below some relevance threshold.

LDA is a simple way to compute an optimal linear transformation in feature space for supervised feature reduction. Feature selection is based on class membership but not on classification quality. It makes the method independent of a specific classification method.

LDA does not guarantee an optimal selection of features for a given classifier. Two reasons for this are that LDA does not consider non-linear correlations and that

its approximation of average within-class scatter covariance is accurate only if corresponding eigenvectors of all within-class covariance matrices are equal.

**Classes and Functions in Python**
Linear discriminant analysis is a class in `sklearn.discriminant_analysis`. It may be used to reduce the dimensionality of features `features` with labels `labels` in a similar way than the PCA class in `sklearn.decompositions`

```
from sklearn.discriminant_analysis import (
                                    LinearDiscriminantAnalysis)
N_COMPONENTS = 100

# create LDA object (the reduced number of components N_COMPONENTS
# is submitted to the constructor
lda= LinearDiscriminantAnalysis(n_components=N_COMPONENTS)

# fit to features and labels from the training data set. The number
# of classes and their a priori probability is inferred from the data
lda.fit(features,labels)

# apply the transformation and return the N_COMPONENTS first
# features
red_features=lda.transform(features)
```

## 3.3 Exercises

### *3.3.1 Programming Project P3.1: Comparing Variance in Feature Space*

After reading the MNIST data (use your function from P1.1), two different feature variants are to be generated from the training data `X_train`:

- Gray values by scaling the images from 28 × 28 to 14 × 14 and converting it to a 1-d feature vector.
- Gray values by converting the 28 × 28 image into a 784-element 1-d feature vector followed by reduction by PCA.

For each feature, the ratio between within-class variation and between-class variation shall be calculated (use your function from P2.2).

For computing the PCA, the MNIST data has to be used as feature vector and not as images (either by using your original function from P1.1 or by reshaping the data derived from partitioning the data as in P2.1). For computing the PCA, you may use the Python class described in Sect. 3.1.2.

### 3.3.2 Exercise Questions

- Why is it necessary to reduce the features further given that the information from the image has already been reduced by feature extraction? Could this be part of the extraction of features?
- What are the advantages and disadvantages of unsupervised feature reduction compared to supervised feature reduction?
- What is the significance of eigenvectors and eigenvalues of a covariance matrix for PCA? How are they used for feature reduction?
- What is the rationale for removing features with low variance in an unsupervised feature reduction? Why can this lead to problems?
- What are the disadvantages of forward feature selection?
- Why is it not possible to use the true within-class covariances in LDA?
- Why is the matrix of average within-class scatter covariances inverted to determine optimal features and the between-class scatter matrix is not? Can optimal features be found when instead the between-class scatter matrix is inverted?

## References

Fabbrizzi, S., Papadopoulos, S., Ntoutsi, E., & Kompatsiaris, I. (2022). A survey on bias in visual datasets. *Computer Vision and Image Understanding, 223*, 103552.

Fauvel, M., Dechesne, C., Zullo, A., & Ferraty, F. (2015). Fast forward feature selection of hyperspectral images for classification with Gaussian mixture models. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing, 8*(6), 2824–2831.

Khalid, S., Khalil, T., & Nasreen, S. (2014). A survey of feature selection and feature extraction techniques in machine learning. In *2014 IEEE Science and Information Conference* (pp. 372–378).

Malhi, A., & Gao, R. X. (2004). PCA-based feature selection scheme for machine defect classification. *IEEE Transactions on Instrumentation and Measurement, 53*(6), 1517–1525.

Mehrabi, N., Morstatter, F., Saxena, N., Lerman, K., & Galstyan, A. (2021). A survey on bias and fairness in machine learning. *ACM Computing Surveys (CSUR), 54*(6), 1–35.

Reif, M., & Shafait, F. (2014). Efficient feature size reduction via predictive forward selection. *Pattern Recognition, 47*(4), 1664–1673.

Song, F., Mei, D., & Li, H. (2010). Feature selection based on linear discriminant analysis. In *IEEE 2010 International Conference on Intelligent System Design and Engineering Application* (Vol. 1, pp. 746–749).

Zebari, R., Abdulazeez, A., Zeebaree, D., Zebari, D., & Saeed, J. (2020). A comprehensive review of dimensionality reduction techniques for feature selection and feature extraction. *Journal of Applied Science and Technology Trends, 1*(2), 56–70.

# Chapter 4
# Bayesian Image Classification in Feature Space


**Abstract** In this chapter, we will define image classification as probabilistic decision making in feature space. The Bayesian theorem is applied to define the most likely label for a sample based on its features. It is computed from a class-conditional feature likelihood and the a priori probability of a sample to belong to some class. We will present different means to estimate a likelihood function from labeled samples. Together with an estimate of the a priori probability this constitutes a generative model that enables computing class membership probabilities for every combination of features and classes. Classification then just requires to determine the maximum a posteriori probability of a sample for some class given its features.

Functions and parameters for the mapping from features to class probabilities are estimated from labeled training samples and tested on labeled test samples. The two types of data sets will be characterized and conditions for acceptable data for this purpose will be developed. Several benchmark data sets that we will use in the exercises in the book are described and criteria to assess the success of a trained classifier are presented.

Feature extraction as discussed in the previous chapters generates a feature vector for every image. Its elements are deemed to be relevant for the classification task. Every location in feature space has a certain probability of representing features of a sample from a particular class. The relevancy assumption predicts that the probability for finding a sample of this class at some locations is substantially higher than for samples from all other classes. Classification in feature space can thus be solved by finding the most probable class of a sample $i$ given its feature vector $\mathbf{x}_i$. All classifiers that we will discuss in the following three chapters base their decision on this concept. A sample $i$ with feature vector $\mathbf{x}_i$ is assigned to a class $c$ of $C$ classes for which its probability $P(y_i = c|\mathbf{x}_i)$ is maximum (see Fig. 4.1).

Classification solutions that estimate probabilities for all combinations of features and classes and then decide based on maximum probability use a *generative classification model*. An alternative to a generative model is a *discriminative model*. It computes a discrimination predicate that is applied to an unseen sample.

We will discuss estimation and use of generative models in this chapter. Heuristics to use a discriminative model will be treated in the following chapters.

© The Author(s), under exclusive license to Springer Nature Singapore Pte Ltd. 2024

K. D. Toennies, *An Introduction to Image Classification*,
https://doi.org/10.1007/978-981-99-7882-3_4

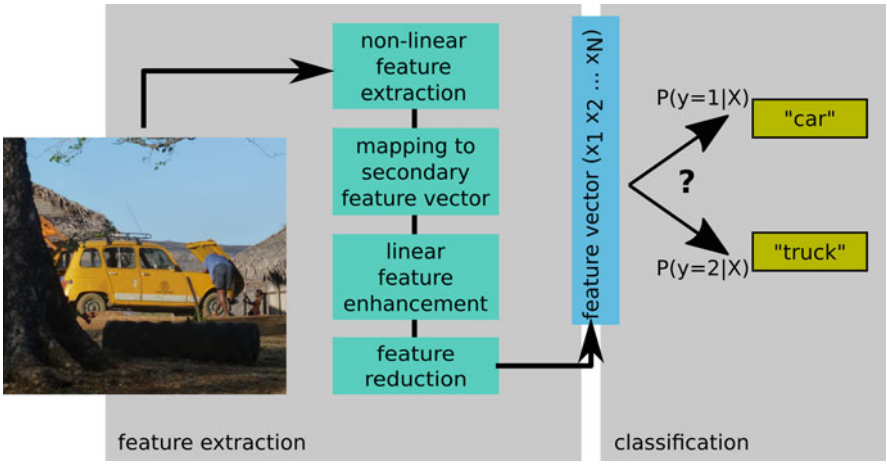



**Fig. 4.1** After completing the feature extraction step, classification decides based on the a posteriori probability $P(y = c|X)$ of a sample to belong to class $c$ given its feature vector $X$

## 4.1 Bayesian Decision Making

The expression $P(y_i = c|\,\mathbf{x}_i)$ is a conditional probability for which the Bayesian Theorem holds:

$$P(y_i = c|\mathbf{x}_i) = \frac{p(\mathbf{x}_i|y_i = c)P(y_i = c)}{P(\mathbf{x}_i)}. \tag{4.1}$$

The function $p(\mathbf{x}_i|\,y_i = c)$ is called *likelihood function* and returns the probability of observing feature $\mathbf{x}_i$ if the sample belongs to class $y_j$. $P(y_i = c)$ is the a priori *probability* of drawing a sample belonging to class $c$ irrespective of the features of this sample (hence "a priori" = beforehand, also just called "the prior"). $P(\mathbf{x}_i)$, finally, is the probability of observing a feature vector $\mathbf{x}_i$ irrespective of the class. It is called the *evidence* of this feature vector. The value of $P(y_i = c|\,\mathbf{x}_i)$ is the a posteriori *probability* because it is the probability of a sample for a class after the feature values of the sample have been considered ("a posteriori" = in retrospect, also just called "the posterior").

If all probabilities are known, classification computes $P(y_i = c|\,\mathbf{x}_i)$ for all classes $c$ and selects the one with highest a posteriori probability. Computing the evidence term $P(\mathbf{x}_i)$ is often omitted as it does not depend on the sample label. For a given sample with known feature vector it can be treated as a constant. Solutions omitting computation of the evidence are still maximizing the posterior. If $P(y_i = c|\,\mathbf{x}_i)$ is maximum for some class $y_i$, then $p(\mathbf{x}_i|\,y_i = c)P(y_i = c)$ will be maximum for this class as well. This leaves the likelihood function $p(\mathbf{x}_i|\,y_i = c)$ and the a priori probability $P(y_i = c)$ to be estimated for Bayesian classification.

Given a priori probability and likelihood function, a decision that maximizes the posterior is optimal given the image features. It does not mean that no classification errors will occur. Error-free classification would require $P(y_i = c|\,\mathbf{x}_i)$ to be zero for all classes but the correct one. It is an unlikely distribution of a posteriori probabilities since there will be always feature vectors $\mathbf{x}$ for which $P(y_i = c|\mathbf{x})$ is non-zero for more than one class $c$.

Even though the classification decision will be wrong for some samples, a decision based on maximum a posteriori probability minimizes the number of erroneous decisions. Any other location for a decision boundary between different classes would produce more errors since more samples were labeled incorrectly than the a posterior probability predicts.

Hence, optimal classification requires to estimate a priori probability and likelihood function. If the error rate for the classifier is too high, either these estimates were not good enough or the features describing the image were inadequate for the classification task.

## 4.2 Generative Classification Models

A generative model estimates the terms of Eq. 4.1 based on training data for which features and class membership are known. We will present just some simple examples for generative models in this text as they play a minor role in image classification (a more detailed presentation of the subject is Harshvardhan et al., 2020). A generative classifier determines the label with maximum a posteriori probability for unseen samples based on their features. Since a posteriori probabilities are computed for all classes, a reliability can be assigned to the decision by computing a ratio of class probability for the chosen class relative to those of other classes.

Estimation of the a priori probability and the likelihood function requires unbiased training samples $\mathbf{x}_i$ with known labels $y_i$. The a priori probability for a label $y_i$ is estimated by computing the share of samples from this class compared to all samples in this data. If the data is unbiased, it is a good predictor for the a priori probability of class membership of unseen samples.

Estimation of the likelihood function is more difficult because a complete function in feature space and not just a single scalar value is to be estimated. Furthermore, is not uncommon to have 100 or even 1000 different features in a feature vector. Given that the number of training samples rarely exceeds 10.000 per class (and is often much less), it causes a sparsely populated feature space (see Fig. 4.2). Outliers due to measurement errors or image artifacts increasingly influence any estimation that depends on the distribution of samples in feature space. Techniques to estimate the likelihood function have to be carefully selected for avoiding this.

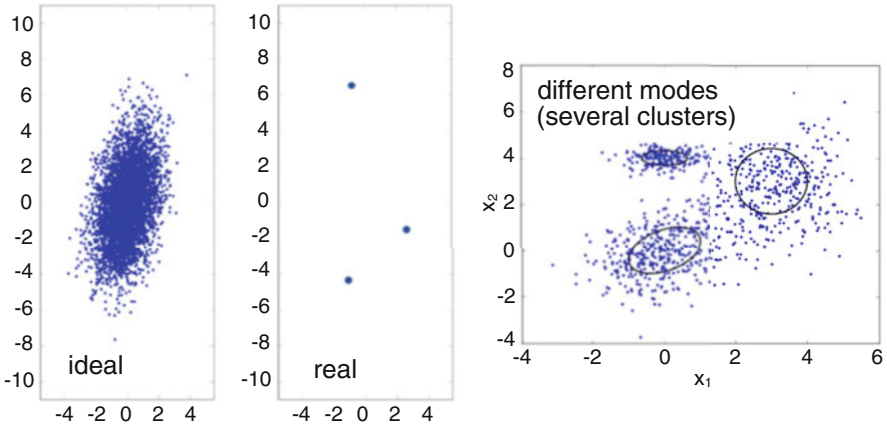



**Fig. 4.2** Ideally, sample density in feature space is high, which would allow for robust estimation of the underlying likelihood function. In reality, the number of samples is often in the range of the number of dimensions in feature space making it difficult to estimate a likelihood function. It is particularly difficult, if the likelihood function is complex, e.g., when it has different modes (corresponding to objects of the same class but with different appearance)

### *4.2.1 Likelihood Functions from Feature Histograms*

If the number of samples in the training data set is high and the dimensionality of feature space is low, the likelihood function $p$ can directly be estimated from feature histograms. Normalized histograms have to be generated separately for each class $c$. No assumptions about the nature of the likelihood functions are made since this estimate depends only on the relative occurrence of feature values for the respective class.

Direct estimation from class-specific feature histograms works only if feature space is densely packed with samples. Otherwise, either too many bins of the histogram remain empty or the number of bins of the histogram has to be set so low that this discretization no longer reflects attributes of the likelihood function. Hence, this solution may be applicable for feature spaces with one or two dimensions but quickly becomes infeasible when the number of dimensions gets higher.

*A kernel density estimator* can be used for somewhat less densely packed feature spaces (see Chen, 2017 for more details). A kernel density function is a convolution kernel that integrates to 1 (a smoothing kernel). The histogram is generated as before. Empty bins caused by sample sparsity are accepted. After computing and normalizing the histogram, it is convolved with the kernel to produce the final estimate for the likelihood function (see Fig. 4.3).

A justification to use kernels is that every sample stands for the expected value of a locally simple distribution of samples. This local distribution is defined by the kernel density function. Since little is known about the kind of local distribution, a normal distribution centered at the origin is taken as estimate. The decision is based on the *central limit theorem*. The classical version of this theorem states that a sum of

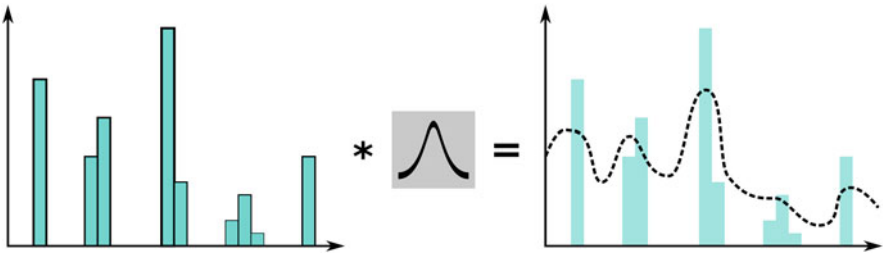


**Fig. 4.3** Kernel density estimators convolve the original histogram with a smoothing function (usually a truncated Gaussian) to arrive at a better representation of the underlying likelihood function

random variables with finite variance approaches a zero mean normal distribution if the number of summands approaches infinity. Since variation around sample features may be caused by many different influences, it motivates to rely on the central limit theorem.

Variances along the different axes in feature space are assumed to be equal so that the separability of the Gaussian enables convolution as sequence of 1-d convolution operations with a Gaussian function

$$N(x;0,\sigma) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{x^2}{2\sigma^2}\right), \tag{4.2}$$

which is truncated and renormalized to the window size of the kernel.

The value $\sigma$ determines the extent over which the original histogram is smoothed. For arriving at a reasonable estimate of the likelihood function, it needs to be large if the data is sparsely distributed. In this case, the purely heuristic assumption of Gaussianity stretches over ever larger regions the sparser the sample distribution gets. Since the assumption is less likely to be true for large kernel sizes, kernel density estimators are no longer applicable if the feature space has more than four or five dimensions.

**Classes and Functions in Python**

For kernel density estimation a class `KernelDensity` is contained in the submodule `sklearn.neigbors`. The class `KernelDensity` has a function fit() that fits the kernel to a multidimensional distribution of sample features. A simple example for using the kernel density estimator to 10 samples of 1-d features in an array `X` with feature values in the interval [0,10] is

```
import numpy as np
from sklearn.neighbors import KernelDensity
...
```

(continued)

```
# construct the KD object:
#  bandwidth is the standard deviation of the kernel
# 'gaussian' is the default, other kernels exist, see sklearn manual
kde = KernelDensity(bandwidth=0.05, kernel='gaussian')

# fit kernel estimation to the data
kde.fit(X[:, None])

# sample probabilities at a fine resolution (100 steps of d=0.1)
X_d = np.linspace(0, 10, 100)

# score_samples returns the log of the probability density at each
# value of 'X_d'
logprob = kde.score_samples(X_d[:, None])

# take the exponent to get the probability density
X_prob = np.exp(X_d)
...
```

### 4.2.2 Parametrized Density Functions as Likelihood Functions

Estimation of a likelihood function in higher-dimensional feature space requires further-reaching conditions. A rather restrictive condition is to assume independent, normally distributed features. Then, for each feature $x_j$ of a feature vector $\mathbf{x}$ the two parameters ($\mu$,$\sigma$) of a normal density are estimated from samples:

$$N(x_j;\mu,\sigma) = \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{(x_j-\mu)^2}{2\sigma^2}\right). \tag{4.3}$$

Unbiased estimates for expected value $\mu_j$ and variance $\sigma_j^2$ of features $x_{i,j}$ of $K_c$ samples with label $y_i = c$ are

$$\mu_j = \frac{1}{K_c}\sum_{\{i|y_i=c\}} x_{i,j} \tag{4.4}$$

and

$$\sigma_j^2 = \frac{1}{K_i - 1}\sum_{\{i|y_i=c\}} (x_{i,j}-\mu_j)^2. \tag{4.5}$$

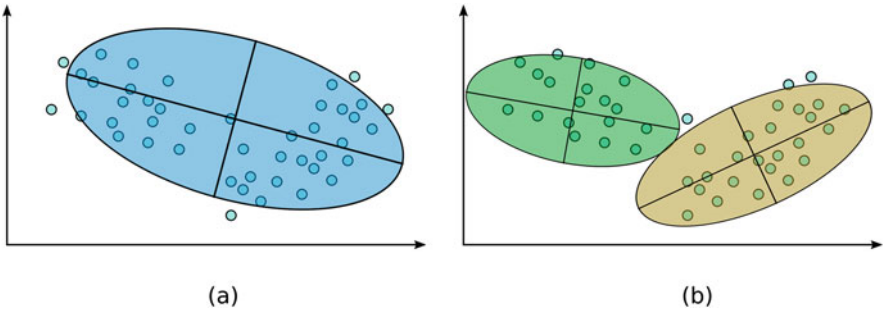



**Fig. 4.4** A distribution with several modes is not well approximated by a Gaussian. In (**a**), the highest probability is assigned to a location in features space that contains almost no samples. A representation by a sum of two Gaussians with different expected value vectors and covariance matrices as in (**b**) describes the data distribution much better

The computation of parameters $(\mu_j,\sigma_j)$ can be carried out independently for each feature. The number of dimensions in feature space does not influence the density of samples along each axis. The likelihood for a given sample is then the product of all probabilities for each of its features.

If the features are correlated, a *multivariate Gaussian density function*

$$N(\mathbf{x};\boldsymbol{\mu},\boldsymbol{\Sigma}) = \frac{1}{\sqrt{2\pi}|\Sigma|}\exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right) \tag{4.6}$$

with vector $\boldsymbol{\mu}$ of expected values and covariance matrix $\boldsymbol{\Sigma}$ needs to be estimated from the data. Covariance is estimated from samples as in Eq. 3.4 in Sect. 3.1.2.

Compared to a set of independent density functions, a multivariate Gaussian function requires more parameters to be estimated. For an $N$-dimensional feature space, a minimum of $N + 1$ samples are needed to enable the computation of a covariance matrix of full rank. Otherwise, the eigenvectors with associated non-zero eigenvalues of the matrix will span a lower-dimensional subspace. For a more robust estimate with respect to measurement noise, many more samples are needed. Hence, the use of multivariate Gaussians quickly becomes inappropriate when the number of features increases.

Despite its greater number of parameters, using a Gaussian may still be inadequate to model the likelihood function of a class (see Fig. 4.4). Remember considerations about correlation between object class and image features. Various effects may cause very different appearances of samples from the same class. It is unlikely that features from these samples cluster at a single location in feature space. Normal distributions, however, predict a single cluster per class.

An example is the appearance change when the same object is seen from different angles. If appearance is mapped to feature space, it may result in a likelihood function with different peaks (the so-called *multimodal density function*). An object of a given class "car" may have a characteristic appearance *"A"* with associated

features $f(A)$ if pictured from the front and another characteristic appearance *"B"* with features $f(B)$ if pictured from the side. $f(A)$ and $f(B)$ of different samples of class "car" may map to different clusters in feature space. In this case, the approximating density function should have two different peaks as well.

*Gaussian mixture models* (GMM) can model such multimodal density functions. A GMM is a sum of $G$ weighted Gaussians $N$ with different expected values and variances. A GMM for a likelihood function $p(\mathbf{x}|y) = p_y(\mathbf{x})$ is

$$p_y(\mathbf{x}) = \sum_{g=1}^{G} \alpha_i N\left(\mathbf{x}|\boldsymbol{\mu}_g, \boldsymbol{\Sigma}_g\right) \tag{4.7}$$

Parameters to be estimated are the weights $\alpha_i$ (that have to sum up to 1) and expected values and covariances of the Gaussians. It is done by an expectation maximization (EM) algorithm that repeats two steps until convergence:

- E-step: Use the current parameter values for the $\alpha_g$, $\boldsymbol{\mu}_g$, and $\boldsymbol{\Sigma}_g$ to compute the probability that the current GMM represents the underlying distribution of the samples.
- M-step: Use gradient ascent for changing parameters to maximize this probability.

The number of Gaussians $G$ is often user-specified although algorithms exist that find an optimal value for $G$ (see McLachlan & Rathnayake, 2014 for a review). Being a gradient ascent, the result of the EM algorithm depends on a good initialization and a good gradient estimate. A good initialization is easily found if the underlying true likelihood function is simple. A good gradient estimate requires a densely populated feature space. Hence, application of a GMM requires a feature space dimension that is even lower than that for a multivariate Gaussian model.

**Classes and Functions in Python**

A covariance matrix for a multivariate Gaussian can be estimated from samples using Eqs. 3.3 and 3.4 using the matrix multiplication in NumPy. For a NumPy array `features` of shape `(numSamples,numFeatures)` the operations are

```
import numpy as np

# estimate expected value vector
exp_val_vector = np.mean(features,axis=0)

# subtract expected value vector from all feature values
# (using broadcasting)
features_c = features - exp_val_vector
```

(continued)

```
# estimate covariance
cov_matrix = np.matmul(np.transpose(features_C), features_C)
```

Expected values and covariance matrices for a Gaussian mixture model can be estimated using the class `GaussianMixture` from `sklearn.mixture`. A simple example is

```
import numpy as np
from sklearn.mixture import GaussianMixture
# create samples, distributed around two peaks (2,2) and (8,8)
X = np.array([[1, 1], [1, 3], [3, 1], [3, 3],
              [7, 7], [7, 9], [9, 7], [9, 9]])

# construction of a GMM with two components and fit to the data
gm = GaussianMixture(n_components=2).fit(X)

# print the means and variances gives the expected result: means are
# at(2,2) and (8,8) and covariance matrices are almost identity
# matrices
print(gm.means_)
print(gm.covariances_)
```

The Gaussian mixture class has many parameters that influence the behavior and methods to access information in the object. It is suggested to read the documentation in SciKit-learn first before using it on real data.

## 4.3 Practicalities of Classifier Training

Approximations necessary to define a classifier depend on feature characteristics from samples with known labels, i.e., class memberships. Such samples are not only needed for the Bayesian classifiers presented here but also for all classifiers to follow. In this section, we will take a closer look at some databases and their use before we will continue to use sample data for defining and analyzing classifiers in the remaining chapters.

### *4.3.1 The Use of Benchmark Databases*

Image classification is an old topic in computer vision. Hence, various reference data sets for different application scenarios exist. Benchmark data sets can be very useful for the development of a classifier. There are two reasons for this:

- Benchmark data sets are usually carefully curated to represent the population for a particular classification problem. The data is selected to represent a realistic subset of pictures for some problem and the size of the database is large enough to parameterize the classifier.
- Benchmark data have often been widely used by the research community. Comparison with previous results provides valuable insights into the quality of a new development.

Hence, a first step in classifier development is to research available reference data sets. If the problem at hand is not entirely new, chances are good that one or several appropriate data sets exist. In this case, a number of publications may exist as well that use this data set. Reported results provide the developer with a benchmark for his or her own method. In the best case, the reference data set is so similar to the data of your own problem that an existing classifier may require just minor modifications such as adjustment of image size, color scales, or removal of pictures that are irrelevant for the problem at hand. Even if not, it will provide the developer with some knowledge about the performance to be expected and about potentially problematic cases for classification.

If no data set exists that reflects the scenario in which the new classifier is to be applied, benchmark data may still help to decide on potentially successful development strategies. If, for instance, children shall be detected and labeled in street scenes (e.g., in a pedestrian recognition system), and there are data sets of street scenes that contain adults, successful methods to classify the latter may be a good starting point for classifying children in such scenes.

Having benchmark data does not relieve the developer to gather own data that is representative for the problem to be solved. Even if benchmark data comprises the kind of data that is to be labeled, the actual problem may have different distribution characteristics. Such shift of distribution characteristics may not be noticed as it may not be apparent to the observer. A different performance of the classifier when applied to problem-specific data compared to that on the benchmark data may then come as a surprise.

We are not concerned with specific classification problems in this introduction but with general concepts for building a classifier. Data sets used in the examples will be thus either very simple or represent rather general classification problems. Data sets for this purpose are listed below (in the order of increasing complexity):

- *MNIST* is a large database for handwritten digits (see Fig. 4.5). It contains 60.000 samples for training and 10.000 samples for testing (see next section for details on training and test data). Samples are almost evenly distributed among the 10 digits. The samples are single channel gray level images with $28 \times 28$ pixels. The data is noise-free without scene clutter. As such, MNIST classification is fairly simple and allows the designer to concentrate on classifier aspects rather than on feature extraction.
- *CIFAR10* is a database for another 10-class problem (see again Fig. 4.5). It is a subset of the much larger database with tiny images (with 80 million images). CIFAR10 contains 60.000 color images (50.000 for training and 10.000 for

**Fig. 4.5** Examples from the MNIST database (left) and the CIFAR10 database (right). Images in both databases are tiny (28 × 28 and 32 × 32 pixels, respectively) but the CIFAR10 is a bit more realistic (and more difficult) since the images contain various amounts of background clutter and the relation between pixel value and semantics is less straightforward

testing) of size 32 × 32 pixels that depict various objects (cars, trucks and the like) with natural background. Each of the classes has exactly 6000 samples. It is more difficult to classify than MNIST because the pixel semantics is not as closely related to pixel values than in MNIST. Furthermore, pictures include background clutter. However, pictures are so small that a classifier will most likely not decide on features relevant to a human observer. Besides, these features may not be relevant, if the object were larger. Hence, while being a good database for experiments with classifiers it will not be appropriate to test object recognition in general.

- *Imagenette* and *Imagewoof* are two of several subsets that were generated from the ImageNet database. The latter is used for the 1000-class challenge for classifiers. ImageNet itself is a realistic challenge but the number of datasets and the number of classes are too large for quick initial experiments with different classifiers. The two databases contain subsets of 10 classes with each class having around 1000 samples for training and 300–400 samples for testing (70:30 train-test split). The classes of Imagenette are quite distinctive (classes such as church, golf ball, gas pump, see Fig. 4.6 for some examples), while Imagewoof has pictures of ten different breeds of dogs, which are more difficult to label (see Fig. 4.7). Images are three-channel RGB. The image size varies but the resolution is mostly better than 200 pixels along the longest side for each image. For classification, images need to be resampled to a common size (224 × 224 has been used for some neural networks but you are free to choose your own sampling rate). The two databases are more realistic examples for a classification task than MNIST and CIFAR10. Images are as large as they might be if they are a result from object detection in a picture, there is a varying amount of background clutter in the pictures, and the number of samples per class is lower than in the

**Fig. 4.6** Pictures in Imagenette are quite distinctive but there is a lot of within-class variation. Furthermore, there are large differences in resolution and aspect ratio between images

**Fig. 4.7** Imagewoof is similar to Imagenette but appearances of images from different classes are much closer to each other

previous databases while at the same time the number of pixels per image is higher.

**Classes and Functions in Python**

The MNIST and CIFAR10 data sets can be accessed using fetcher functions from `sklearn.datasets` (which also allows to fetch a large number of other real world data sets). The MNIST and CIFAR10 data are part of the OpenML repository that makes many other data sets accessible to the community (https://www.openml.org) and can be accessed through the function `fetch_openml()`

```
import numpy as np
from sklearn.datasets import fetch_openml

mnist      = fetch_openml('mnist_784') # get MNIST bunch object
mnist_data = np.array(mnist.data)    # data is a 1-d feature vector
mnist_labels= np.array(mnist.target)   # labels for each sample
```

(continued)

```
cifar10     = fetch_openml('CIFAR_10') # same for CIFAR10
cifar10_data  = np.array(cifar10.data)
cifar10_labels= np.array(cifar10.target)
```

Imagenette and Imagewoof are not contained in the repository. They can be downloaded from the GitHub repository at https://github.com/fastai/imagenette. Data is organized by label in different directories and already split into training and validation data.

### 4.3.2 *Feature Normalization*

An important assumption for all classifiers to work is that samples cluster well in feature space. A different range of feature values is among the many reasons that may cause unnecessary within-class scatter. It can be removed prior to classifier training by *feature normalization*.

The scale of different features may be vastly different. It becomes apparent when features are related to human-interpretable semantic, e.g., if one feature is an intensity ranging from 0 to 255 and another is a distance in centimeters on a sheet of DIN-A 4 paper ranging from 0 to 29.7. Such difference in scale may be present in the more abstract features as well and may cause unwanted higher variation in one direction compared to other directions in features space.

Feature normalization removes these differences (see Fig. 4.8). Different normalization schemes exist. Examples are:

- *Linear normalization* maps features to a range from 0 to 1. It requires the minimum and maximum value of the feature to be known.
- Linear mapping on an approximate range from 0 to 1. The minimum and maximum are estimated from training data and used for this kind of normalization. As these extrema are estimates, values lower than 0 or higher than 1 may occur.

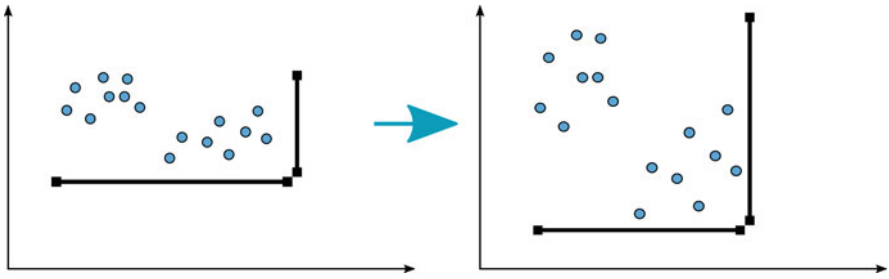


**Fig. 4.8** Normalization adjusts the scale so that features along each dimension vary in the same way and that the feature range is either 1 or has the standard deviation of 1

– *Standardization* so that the resulting features have an expected value $\mu$ of 0 and a variance $\sigma^2$ of 1. The values for $\mu$ and $\sigma^2$ are estimated from labeled data. Standardization subtracts $\mu$ from the original features and divides the result by $\sigma^2$.

In some cases, feature normalization may remove important information. This would be the case if low variance of a features means that it is of little relevance to the classification. Normalization may then enhance just noise.

### 4.3.3 Training and Test Data

The purpose of collecting labeled sample data is twofold:

1. *Training data* with known labels define an optimal mapping between feature vectors and labels. This mapping will be the classifier model.
2. *Test data* estimates how good this classifier generalizes to the unseen data.

The training data is used to specify a mapping from features to samples. For instance, training data determines a priori probability as well as means and covariance matrices of the likelihood function for each class for a Bayesian classifier assuming a multivariate Gaussian density as likelihood function. A necessary requirement is that training data is an unbiased representative for the population of all samples.

A minimum quality should be achieved (e.g., better than a previous classifier, better than some absolute performance level) when training a classifier. If results are not satisfactory, it may have two reasons:

- The features do not characterize the class membership well. It can be recognized by inspecting wrongly classified images. Inspection may reveal
  - that relevant image features have been overlooked,
  - that they have been lost when primary image features were mapped to a secondary feature vector, and
  - that they have been lost during feature reduction of secondary features.
- The *expressiveness* of the classifier is insufficient to represent the distribution of class members in feature space. This is called *underfitting*. An example is the attempt to represent a multimodal distribution by a Gaussian (which is unimodal).

Even if training produces excellent results a later application of the trained classifier in the wild may yield disappointing results. The reason is threefold:

1. The training data has been biased.
2. The training samples are too sparsely distributed in feature space and do not represent the true characteristics of the distribution.
3. The number of free parameters of the classifier is too high given the (unknown) true degrees of freedom necessary to describe the class characteristics.

*Bias* in the training data is mostly unintended. An incomplete list of bias in a sample database compared to the unseen samples of 2-d depictions of 3-d objects is

- 3-d Geometric Transformation Bias: Object rotation with respect to the camera in the samples is not representative.
- Projection Bias: Distance and projection geometry is not representative.
- Illumination Bias: Illumination in sample scenes is not representative (due to different weather conditions, time of day, season, etc.).
- Background Bias: Sample scene background distribution does not reflect reality.
- Hidden Surface Bias: Extent and location of hidden object parts differs between training data and reality.
- Object Selection Bias: Objects of a same class do not fully represent class-specific variation.

Biased training data will result in a biased classifier which in turn will perform badly when unbiased data is presented to the classifier.

Detecting bias is difficult because bias is unintended. Nobody will be interested to develop a biased classifier (unless some kind of cheating is the purpose). Hence, if unbiased data set were present, a responsible engineer would have already used it for classifier training.

But how can unbiasedness be ensured? It would require a random selection of samples from the population. Some of the challenge data for image classification (e.g., the challenge on the ImageNet data) comes close to it as these data were randomly sampled from pictures available on the internet and then grouped into classes.

For real applications, however, the classes to be distinguished are known beforehand (e.g., pedestrian recognition in an autonomous driving scenario). A random sampling of possible pictures makes little sense as most of the samples would be irrelevant for the purpose. Hence, pictures are sampled that are deemed to be relevant for the project (scenes that may be imaged from a camera in a car in our pedestrian recognition example).

Bias may then be produced by the decision just what is relevant and what would be the a priori probability of such as scene to occur. It may easily be forgotten to include whole groups of images (e.g., changing weather conditions, country-specific road scenery, or make-specific positions of the camera in the car). There is no guarantee that these sources of bias are excluded although best practice rules for data collections exist that avoid biased data collections, see, e.g., DeBrusk (2018) or Hall et al. (2022).

Too few samples or too many parameters of the classifier, the remaining two aspects, may cause the so-called *overfitting* of the classifier to the training data which is easier to detect than bias (see Fig. 4.9). An extreme example for overfitting would be a GMM with as many Gaussians in the sum as there are samples in the training data. Training would be error-free as each sample could be represented by a single Gaussian with the sample's feature vector as expected value and with zero variance. It represents the training data well but it is a very poor predictor of labels of the unseen data. Hence, the classifier does not generalize well.

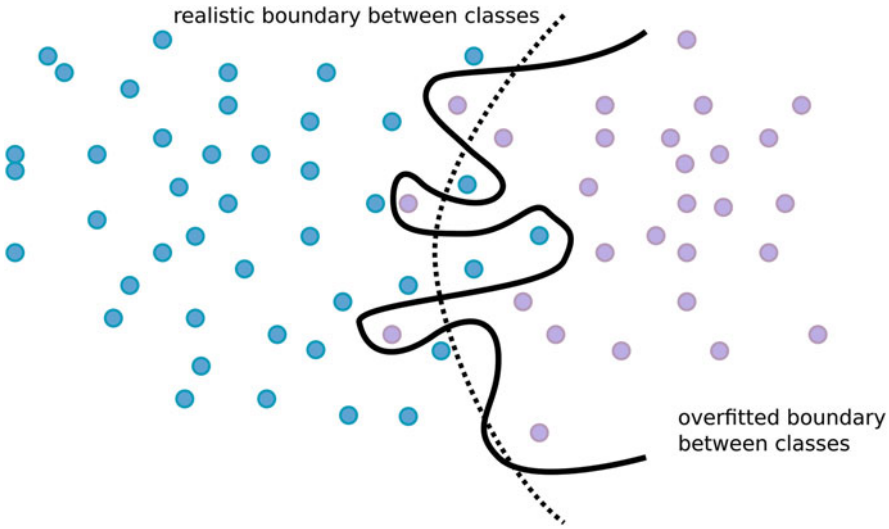



**Fig. 4.9** The locations of equal probability for a two-class problem form a surface that separates samples of the two classes. If a classification model has many parameters, the likelihood functions can be adapted in a way as to perfectly separate the training samples from the two classes. This is most probably not the best kind of separation in feature space since the unseen samples will not have exactly the same features than the training data

For testing the generalization ability, the test data set is needed. It must be independent of the training data and identically distributed (often abbreviated as i.i.d. = independent and identically distributed). Since the test data has not been used to fix classifier parameters the performance of the trained classifier on the test data predicts the performance on the population of the unseen data.

Usually, most of the data is used for training. Common split ratios are 70:30, 80:20, or even higher. It is a reasonable choice since robustness of classifier training depends on the number of available samples. Identical independent distributions of training and test data are guaranteed by random selection of samples for the two data sets and by ensuring that no sample is included in both data sets.

### *4.3.4 Cross-validation*

If the test database becomes too small, the sample distribution may be insufficiently represented by it. It also increases the probability of accidental bias when selecting test data. Hence, a small test database may decrease the capability to predict the performance of the classifier on the unseen data. *Cross-validation* is a means to deal with this problem. For $n$-fold cross-validation, the sample data set is divided into $n$ subsets of (approximately) equal size. If, for instance, the labeled data consists of 100 samples, a threefold cross-validation would create three subsets of sizes

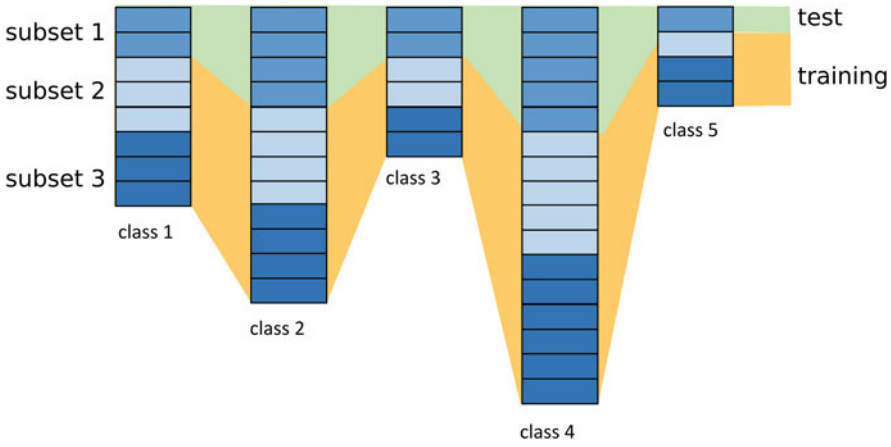



**Fig. 4.10** Example for threefold cross-validation. Sample data is divided into three subsets of approximately the same size and with ratio of samples from each class being approximately retained. A single of the three runs during cross-validation selects one of the subsets for test and uses the remaining subset for training the classifier

(33,33,34). Samples for the subsets are randomly selected and in a way that retains the ratio of samples in each class (see Fig. 4.10 for an example).

Cross-validation carries out $n$ independent training and test sessions. In each session, a different subset is used for testing. The remaining $n - 1$ subsets are used for training. Hence, each session used i.i.d. training and test data sets. The overall test quality is the average over test qualities from all sessions. If it is satisfactory, final training is carried out with all subsets.

Cross-validation makes optimal use of the sample database as all data sets have been used for training and testing without violating the independency constraint. It also enables computation of some secondary performance measures such as the mean and standard deviation of classifier performance on the different subsets. This provides some insight into the distribution of samples in feature space.

### *4.3.5 Hyperparameter*

Many classifiers have parameters that are not trained when fitting the classification model to the data. An example is the number of Gaussian functions that make up a GMM. It is user-defined and an engineer may wonder whether she or he has selected an optimal value for classifier training.

These parameters are called *hyperparameters* of the classifier. There may be a rule-of-thumb or some best-practice-rule that can be applied to find a good (initial) value for a hyperparameter. Already implemented classifiers will use this setting as default values. In other cases, the meaning of the hyperparameter is closely

associated to known properties of the classification problem, the image features, or the object appearance. Then, an (initial) parameter setting may be based on a careful analysis of the actual classification problem.

If neither is the case, hyperparameter settings are determined by training using a third i.i.d. sample set, called *validation data*. The training data set is still used for finding model parameters of the classifier. The validation set is used to find optimal hyperparameters given the model parameters. Finally, the test data set tests the generalization ability with fixed model parameters and hyperparameters.

Generating three data sets from the sample database decreases the number of training samples further. Split ratios like 70:20:10 that still use most samples for training are preferred as the number of parameters for the classification model is usually much higher than that of the hyperparameters.

### 4.3.6 Measuring the Classifier Performance

Given a labeled set of samples, the performance of a classifier is the ratio between correct decisions and all decisions. This is called the *accuracy score* of the classifier. For samples $\mathbf{x}_i$, $i = 1, M$ and associated class labels $y_i \in \{1,\ldots,C\}$ (the so-called *ground-truth*), application of the classifier cls on a sample with features $\mathbf{x}_i$ delivers a label $\text{cls}(\mathbf{x}_i)$. Computing accuracy rates the number of correct decisions $|\text{cls}(\mathbf{x}_i) = y_i|$ against the number $M$ of training samples:

$$\text{acc}(\text{cls}) = \mid \text{cls}(\mathbf{x}_i) = y_i \mid /M \tag{4.8}$$

The classification model has acquired knowledge if accuracy is higher than random guessing. For $C = 4$ with equal a priori probability for each class, for example, the expected accuracy from random guessing would be 0.25. Any accuracy above 0.25 means that the classifier has learned something about the feature-to-label relation.

Besides an overall accuracy score, a detailed look helps to understand class-specific differences. If the number of classes is not too high, a *confusion matrix* helps to understand how reliably samples of the different classes are correctly labeled. A confusion matrix has as many rows $i$ and columns $j$ as there are classes (see Fig. 4.11). The rows indicate ground truth labels and the columns indicate predicted labels by the classifier. An entry $(i,j)$ of the matrix contains the number of samples with true class $i$ that have been classified as class $j$. Correct classifications are found on the diagonal of the confusion matrix. Confusion matrices enable a quick assessment of the overall accuracy of the classifier and of classes that are particularly difficult to classify.

The confusion matrix is also the base for two class-specific scores (see again Fig. 4.11):

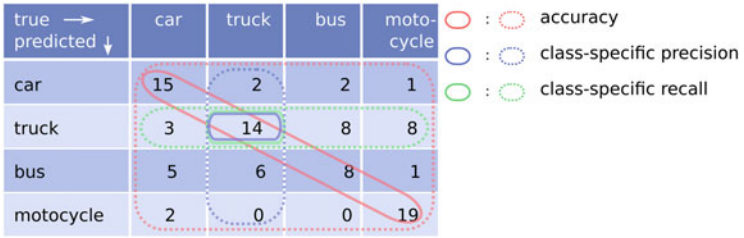



**Fig. 4.11** Confusion matrix for a four-class problem. Accuracy, class-specific precision, and recall scores can be obtained from the confusion matrix

- *Precision score p* for a class with class label $c$ measures the ratio of correctly labeled samples of a class $c$ to all samples with this label. For instance, if 15 of 20 samples that were labeled as "truck" truly are trucks and the remaining 5 samples were incorrectly labeled, then $p = 0.75$ for this label.
- *Recall score r* for a class $c$ measures, how many members of a class were correctly labeled. If a database contains 25 samples with label "truck" but just 15 of them were recognized as trucks, the recall is $r = 0.60$.

### 4.3.7 Imbalanced Data Sets

Precision and recall scores take care of class-specific differences but a single value, such as accuracy, is used for model parameter optimization. Relevant class-specific differences from such single quality criterion can be expected, if data sets are imbalanced. An *imbalanced data set* means that samples from some classes occur much more often than samples from other classes (see also Haixiang et al., 2017).

An extreme example would be a two-class problem, where 99% of the samples belong to class 1 and only 1% belong to class 2. A classifier that simply used the a priori probability would assign all samples to class 1 with a quite satisfactory overall accuracy of 0.99. It may not be satisfactory in view of the application, however. Imagine that class 1 stands for "non-defective" and class 2 for "defective" in a visual part inspection system. The ratio of 99:1 could be quite realistic in such scenario. The cost incurring from declaring a defective part as non-defective, however, will probably be much higher than that for the opposite case.

This can be factored in when looking at the class-specific precision and recall values by computing the *F1 score* for each class

$$F1(p, r) = \frac{2pr}{p + r}. \tag{4.9}$$

Subsequently, a weighted average of all $F1$ scores from all classes is computed.

If changing the quality criterion is insufficient for arriving at a good classifier for imbalanced data, the classifier itself can be changed. The main reason for preferring the majority class is the difference in the a priori probability. This behavior could be changed by manipulating this probability. If the a priori probability is assumed to be equal for each class even if it is not, all classes were treated equally. Just the class-specific likelihood function decides on classification. However, intentionally choosing an incorrect a priori probability is not particularly elegant. It is better to make this change explicit by factoring a *cost of error* for each class into the equation. The model is then trained to minimize the average cost of error instead of maximizing the a posteriori probability.

**Classes and Functions in Python**
The different scores listed above (and many more) are contained in the submodule `sklearn.metrics`. For a set of ground truth labels `y_gt` and predicted labels `y_predict` of a two-class problem, they are computed as follows:

```
from sklearn.metrics import (accuracy_score, precision_score,
              recall_score, f1_score, confusion_matrix)

ac = accuracy_score(y_gt, y_predict)  # these are all single scores
pc = precision_score(y_gt, y_predict)
rc = recall_score(y_gt, y_predict)
f1 = f1_score(y_gt, y_predict)

cm = confusion_matrix(y_gt, y_predict) # this is a 2-d matrix
```

The accuracy score and the confusion matrix work with multiclass problems as well. Precision, recall, and f1 score for multiclass problems deliver values for class label 1 vs. the remaining class labels. Scores and confusion matrix are normalized by default. This can be changed by submitting a parameter setting `normalize=False` with the call of the function, e.g.,

```
ac = accuracy_score(y_gt, y_predict, normalize=False)
```

## 4.4 Exercises

### *4.4.1 Programming Project P4.1: Classifying MNIST Data*

Write a program that

- reads the MNIST data (as in P1.1),
- reduces the features by PCA to the first 40 features (use the program snippet from Sect. 3.1.2),
- computes expected values and covariances for each class on the reduced features,
- uses this for classifying the data, and
- reports the accuracy of your result.

In order to compute class-specific vectors of expected values and covariances, you will need to partition the MNIST data by label. This can be done using the function `nonzero()` in the module NumPy. For example, for a feature array `features` of shape `(M,N)` with `M` samples and `N` features per sample and corresponding array `labels` containing a label for each sample, the following snippet extracts indices of entries in labels with `cur_label`, computes the number of entries, and then extracts features with this label:

```
loc = np.nonzero(labels==cur_label)
nmb = np.shape(loc)
features_cur = features[loc[0],:]
```

Class-specific expected values and covariance matrices can then be computed by matrix multiplication as explained in Sect. 4.2.2.

Since the MNIST data is balanced, the a priori probability is the same for every digit. Hence, the likelihood is proportional to the a posteriori probability. For computing the likelihood function in Eq. 4.6 you will need the NumPy matrix multiplication (see example for computation of covariances), as well as functions to compute determinant and inverse of the matrix. The latter two are included in the submodule `linalg` of NumPy. For a square matrix `mat`, the two operations are

```
determinant = np.linalg.det(mat)
inverse_mat = np.linalg.inv(mat)
```

Since the features from PCA are linearly uncorrelated (although this is true only for the complete data set but not the individual classes), the covariance matrices for the different classes should be close to diagonal matrices and easily invertible. However, for generalizing the function for use with other features, it is good to check, whether the computation of the determinant throws an exception (happens if matrix is too large) or indicates that the matrix is singular before attempting to invert it.

In order to classify the data, the likelihood has to be computed for each sample and each class. For efficient computation, loops should be avoided when computing the likelihood. Ideally, the output would be a NumPy array with shape `(M,C)`, where `M` is the number of samples and `C` the number of classes. Each entry contains

the likelihood of this sample and class. It can be achieved by using the broadcasting ability when working with NumPy arrays.

The class with highest likelihood will be your prediction. A function to find the argument of a NumPy array is `np.argmax()`. It takes the axis-argument to specify along which axis the index with maximum value shall be returned.

For computing the accuracy between ground truth and your predictions, you may use the functions in the SciKit-learn package presented in Sect. 4.3.7.

### *4.4.2 Exercise Questions*

- What is meant by the likelihood function? When can it be used directly (i.e., without knowledge of other probabilities) for classification based on features?
- How could classification be affected if the training data are not representative of the population? How can this non-representativeness be determined?
- What happens if a Gaussian kernel density estimator with large variance is applied to a sparse sample distribution? Why is this a problem?
- Outline a situation where you would expect a likelihood function to not be a normal distribution.
- What would be possible reasons if training a classifier produced a low precision on the training data? How might this be remedied?
- What is the difference between the parameters and hyperparameters of a model? What is the consequence for testing if hyperparameters have to be determined as well?
- Why is cross-validation often used to validate a classifier? What criteria would you use to determine the number of subsets (folds) of a cross-validation?

## References

Chen, Y. C. (2017). A tutorial on kernel density estimation and recent advances. *Biostatistics & Epidemiology, 1*(1), 161–187.

DeBrusk, C. (2018). The risk of machine-learning bias (and how to prevent it). *MIT Sloan Management Review*.

Haixiang, G., Yijing, L., Shang, J., Mingyun, G., Yuanyue, H., & Bing, G. (2017). Learning from class-imbalanced data: Review of methods and applications. *Expert Systems with Applications, 73*, 220–239.

Hall, M., van der Maaten, L., Gustafson, L., Jones, M., & Adcock, A. (2022). A systematic study of bias amplification. arXiv preprint arXiv:2201.11706.

Harshvardhan, G. M., Gourisaria, M. K., Pandey, M., & Rautaray, S. S. (2020). A comprehensive survey and analysis of generative models in machine learning. *Computer Science Review, 38*, 100285.

McLachlan, G. J., & Rathnayake, S. (2014). On the number of components in a Gaussian mixture model. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, 4*(5), 341–355.

# Chapter 5
# Distance-Based Classifiers


**Abstract** Distance to classified samples of a training database can be used to determine labels of an unknown sample. Depending on the type of classifier, a distance-based classifier is related to Bayesian inference in different ways. Two popular methods exemplifying various concepts will be presented in this chapter.

A nearest centroid classifier computes distances of an unseen sample to the class means in feature space. The closest distance to one of the class means decides on the label. Under certain assumptions about the likelihood function, this maximizes the posterior probability of a sample for this label given its features.

A kNN classifier, however, bases its decision for a label on the $k$ closest samples to an unseen sample. It directly approximates the a posteriori probability of a sample to belong to a class given its features. The quality of the estimate depends on the density of labeled training data. Efficient determination of the closest samples during inference requires some preprocessing to exclude far-away samples from the search. We will explain how this can be done by representing the feature space by a binary space partitioning tree that is created in a set-up phase prior to inference.

Generative classifiers, discussed in the previous chapter, completely describe the probabilistic relationship between features and class labels. They are rarely used for image classification because

- the number of features is high (100s to 1000s or even more),
- the number of samples is low (sometimes less than a few hundred per class), and
- the distribution within a class is often multimodal so that simple parametrized likelihood functions are too inaccurate.

A discriminative model requires less information from samples. It either estimates the a posteriori probability of a class given a sample's features or executes a predicate on the features that delivers the sample's class. Exemplary methods of the first kind will be presented in this chapter and those of the second kind will be presented in the next chapter.

© The Author(s), under exclusive license to Springer Nature Singapore Pte Ltd. 2024

K. D. Toennies, *An Introduction to Image Classification*,
https://doi.org/10.1007/978-981-99-7882-3_5

## 5.1 Nearest Centroid Classifier

The *nearest centroid classifier* (NCC) is also known as *nearest mean classifier* or *minimum distance classifier*. It is simple to define and simple to apply to unseen samples. Given a distribution of training samples in feature space, an expected feature vector $\boldsymbol{\mu}_c$ is estimated for each class $c$ by averaging over all samples of this class. This already completes classifier training.

### *5.1.1 Using the Euclidean Distance*

Euclidean distances of an unseen sample with features $\mathbf{x}_i$ are computed to all mean vectors $\boldsymbol{\mu}_c$. The sample is assigned to the one class to which it is the closest (see Fig. 5.1):

$$y_i = \underset{c}{\operatorname{argmin}} \|\mathbf{x}_i - \boldsymbol{\mu}_c\|. \tag{5.1}$$

The decision maximizes the a posteriori probability of the sample to belong to this class for specific kinds of likelihood functions. An example are the following conditions:

1. The a priori probability is equal for all classes.
2. The likelihood function for each class $c$ is a normal distribution $N(\boldsymbol{\mu}_c,1)$ of features with expected value $\boldsymbol{\mu}_c$ and determinant of the covariance matrix $\det(\boldsymbol{\Sigma}) = 1$.
3. Features are linearly decorrelated.

Condition 2 and 3 result in $\boldsymbol{\Sigma}$ being the identity matrix for all classes.

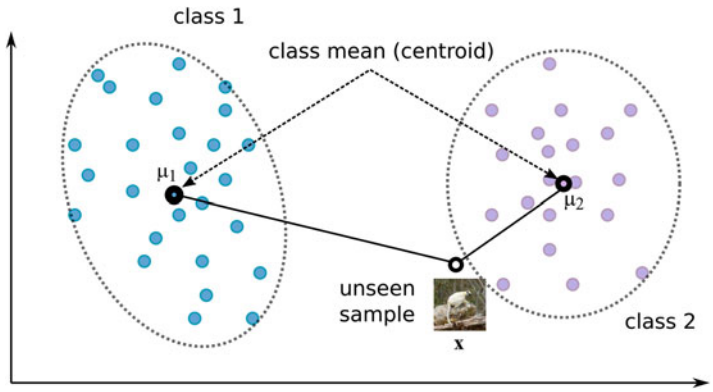



**Fig. 5.1** The nearest centroid classifier computes the means (the centroids) of all classes in feature space and assigns an unknown sample according to distances to the class means

In this case, the a posteriori probability of all samples irrespective of their class is the likelihood function $p(\mathbf{x}|y)$ scaled by the ratio between a priori probability $P(y)$ and evidence $P(\mathbf{x})$ which are both constant for all samples:

$$P(y|\mathbf{x}) = \frac{P(y)}{P(\mathbf{x})} p(\mathbf{x}|y). \tag{5.2}$$

The likelihood function in $N$-dimensional feature space for some class $c$ given the assumptions above is

$$\begin{aligned} p(\mathbf{x}|y=c) &= \frac{1}{\sqrt{(2\pi)^N |\boldsymbol{\Sigma}|}} \exp\Big(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_c)^{\mathrm{T}} \boldsymbol{\Sigma}^{-1} (\mathbf{x}-\boldsymbol{\mu}_c)\Big) \\ &= \frac{1}{\sqrt{(2\pi)^N}} \exp\Big(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_c)^2\Big). \end{aligned} \tag{5.3}$$

The logarithm of the likelihood (the *log-likelihood*) for an unseen sample with features $\mathbf{x}$ to belong to some class $c$ decreases with half of the squared distance to $\boldsymbol{\mu}_c$ minus a correction value for normalization:

$$\ln p(\mathbf{x}|y=c) = -\frac{1}{2}\|\mathbf{x}-\boldsymbol{\mu}_c\|^2 - \ln\sqrt{(2\pi)^N}. \tag{5.4}$$

Taking the logarithm of the a posteriori probability in Eq. 5.2 and using Eq. 5.4 we arrive at

$$\ln P(y=c|\mathbf{x}) = -\frac{1}{2}\|\mathbf{x}-\boldsymbol{\mu}_c\|^2 - \alpha \tag{5.5}$$

with

$$\alpha = \ln\sqrt{(2\pi)^N} - \ln P(y) + \ln P(\mathbf{x}) \tag{5.6}$$

as a constant since $P(y)$ and $P(\mathbf{x})$ are equal for all samples. As the logarithm monotonously increases, minimizing $\|\mathbf{x}-\boldsymbol{\mu}_c\|$ maximizes $P(y=c|\mathbf{x})$.

The a posteriori probability $P(y=c|\mathbf{x})$ can be computed from the squared distance $\|\mathbf{x}-\boldsymbol{\mu}_c\|^2$ if the evidence $P(\mathbf{x})$ for obtaining feature $\mathbf{x}$ is estimated from the training data. Comparison with a posteriori probabilities for other classes gives an indication of the reliability of a decision.

Assuming equal a priori probabilities as well as independent features with normalized variance seems to be quite rigid. However, we will see below that it is at least approximately true for many distributions.

Having the same a priori probability for all classes means a balanced data distribution. Each class is assumed to occur as often as all other classes. If it is

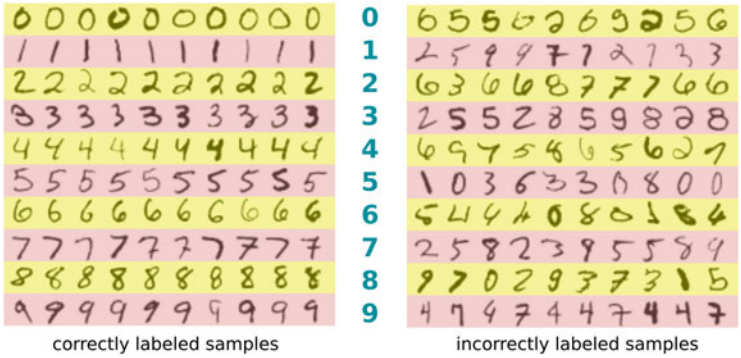



**Fig. 5.2** Classification of the MNIST data using a nearest centroid classifier with pixel values as features. The standard deviation within classes averaged over all feature axes was between 24 and 27 gray values except for class "1" that had a lower variation of 21. It hints at similar co-variance matrices. The accuracy of this simple classifier is already about 80%

true or if existing class imbalance has been counteracted for arriving at a satisfactory classifier performance, condition 1 for using NCC is satisfied. Class-specific a priori probabilities can be included in the equation as well. In this case, the logarithm of class-specific values $P(y_c)$ is subtracted when computing Eq. 5.5. As expected, the subtraction moves the decision boundary (the location where class memberships for different classes are equally likely) in the direction of the class with lower a priori probability.

Condition 2 asks for features with normalized variance. It is often achieved during feature preprocessing by feature standardization (see Sect. 4.3.2). Reduction techniques such as PCA map to linearly decorrelated features. It does fulfill condition 3 only, if all class-specific distributions have the same covariance matrix. Otherwise, it produces a kind of average covariance. It may still be adequate if likelihood covariances do not vary much between classes.

The example above is just one of the several likelihood functions that produce a decision based on maximizing the a posteriori probability. For other rotationally symmetric likelihood functions that monotonically decrease with distance from the expected value, NCC maximizes the a posteriori probability as well.

In summary, NCC is a simple but often appropriate model for classification in feature space that assumes rotationally symmetric likelihood functions. Classification of the MNIST data using pixel values as features using this simple classifier already produced an accuracy of around 80% (see Fig. 5.2)

**Classes and Functions in Python**

A class to use the nearest centroid classifier is contained in the submodule `sklearn.neighbors` of the SciKit-learn module. It may be used as follows to fit the classifier to `N` features `X` with shape `(M,N)` with

(continued)

corresponding labels `y` with shape `(N)` and then classify an unknown sample `X_unknown` with shape `(M)`:

```
from sklearn.neighbors import NearestCentroid
...
ncc= NearestCentroid() # constructor to instantiate classifier
                       # object
ncc.fit(X, y)                      # compute means
print(ncc.predict([X_unknown])) # predict class
...
```

### 5.1.2 *Using the Mahalanobis Distance*

If the condition on the likelihood function does not hold but it is still a normal distribution, NCC may still be used if the Euclidean distance is replaced by the *Mahalanobis distance*. It measures distances in multiples of standard deviations (see Fig. 5.3). Hence, a Mahalanobis distance of 1 of a feature value from the expected value means that it is one standard deviation away from the expected value. The Mahalanobis distance of a feature vector $\mathbf{x}$ from the expected value $\boldsymbol{\mu}$ is

$$d_m(\mathbf{x}, \boldsymbol{\mu}) = \sqrt{(\mathbf{x} - \boldsymbol{\mu})^{\mathrm{T}} \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu})}. \tag{5.7}$$

This distance is related to the a posteriori probability in a similar fashion than the Euclidean distance.

Independent covariance matrices for features of each class have to be computed using the Mahalanobis distance. Hence, computational costs and requirements regarding the density of training samples in feature space are as high as for the estimation of a multivariate Gaussian distribution of a generative model.

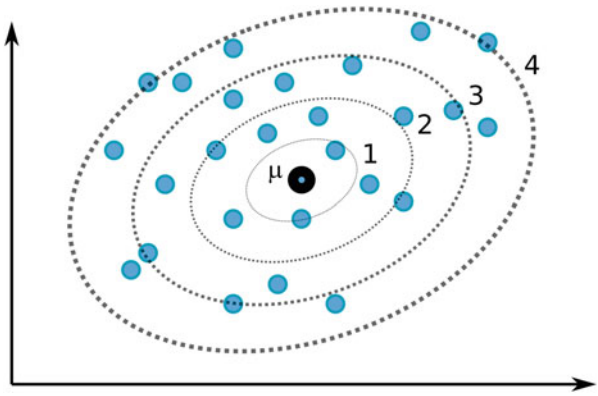



**Fig. 5.3** The Mahalanobis distance measures distances in multiples of standard deviations. If the sample distribution were a normal distribution, equal distances refer to equal probabilities of this feature combination to occur

For increased efficiency and in the case of a sparsely populated feature space, the Mahalanobis distance often uses a common averaged covariance matrix instead. Using the Mahalanobis distance extends the number of likelihood functions for which NCC determines the label with maximum a posteriori probability to those that can be completely described by expected value and covariance and where all likelihood functions are equal. It is similar to linear decorrelation of features by PCA followed by NCC based on the Euclidean distance.

**Classes and Functions in Python**
The Mahalanobis distance may be computed by using matrix inversion and matrix multiplication from NumPy. For a feature vector `f` and an invertible covariance matrix `cv`, we have

```
import numpy as np

# linalg is a submodule with functions from linear algebra
# linalg.inv inverts a square matrix
out_f = np.matmul(f,np.linalg.inv(cv))
```

There is also a function in SciPy that takes the feature vector and the inverse of the covariance matrix.

## 5.2 The kNN Classifier

A minimum distance classifier requires the likelihood functions to be unimodal, monotonically decreasing, and rotationally symmetric. Unimodality is not always to be expected for the distribution of features that stem from pictorial information about 2-d or 3-d objects.

A solution is to estimate the a posteriori probability just for a specific sample. The *k-nearest-neighbor (kNN) classifier* does just this. Instead of the complete histogram for a generative model, just a single bin is created around the location of an unseen sample in feature space. Its size is just large enough to allow a good estimate of a posteriori probabilities for the different classes for this sample. It is a *local neighborhood* around the sample $\mathbf{x}$ with a size that includes the $k$-nearest neighbors to $\mathbf{x}$ (see Fig. 5.4).

The advantage over the histogram approach in Sect. 4.2.1 is an adaptive neighborhood size that depends on local sample density. However, the kNN classifier is not a generative model as it does predict a sample density only at specific locations $\mathbf{x}$ in feature space.

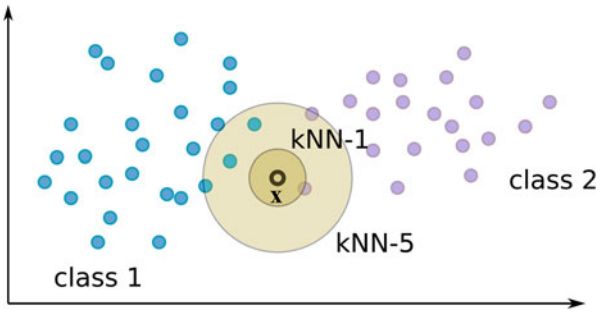



**Fig. 5.4** The k-nearest-neighbor classifier defines a neighborhood around an unknown sample **x** that is just large enough to include the $k$ closest samples of the training data set

### 5.2.1 *Why Does the kNN Classifier Estimate a Posteriori Probabilities*

The distribution of training samples in feature space represents that of unknown samples of the population. It represents the likelihood functions for all $C$ classes weighted by their frequency of occurrence in the training data. For bias-free training data, it is an approximation of the a posteriori probability weighted by the evidence at each location in feature space.

The distribution of training samples is sparse, however. For estimating $P(y = c|\mathbf{x})$ at some location **x**, the local neighborhood nbs(**x**) around **x** is defined. The likelihood functions $p(\mathbf{x}|y = c)$ are assumed to be linear in nbs(**x**). In this case, the following estimate can be computed from samples $\mathbf{x}_i$ with labels $y_i$ that are in nbs(**x**)

$$p(\mathbf{x}|y=c) \approx \frac{\sum_{\mathbf{x}_i \in \text{nbs}(\mathbf{x})} |y_i = c|}{M_c}, \tag{5.8}$$

where $|y_i = c|$ is the count of samples with label $c$ and $M_c$ is the total number of samples in the training data with label $c$. The a priori probability $P(c)$ is

$$P(c) \approx \frac{M_c}{M}, \tag{5.9}$$

where $M$ is the total number of samples in the training data. The evidence for feature vector **x** (i.e., its probability to occur) is approximately

$$P(X) \approx \frac{|\text{nbs}(\mathbf{X})|}{M}. \tag{5.10}$$

Hence, an estimate of the a posteriori probability $P(y = c|\,\mathbf{x})$ is just the ratio of neighbors belonging to class $c$ to all samples in the neighborhood (see Fig. 5.5)

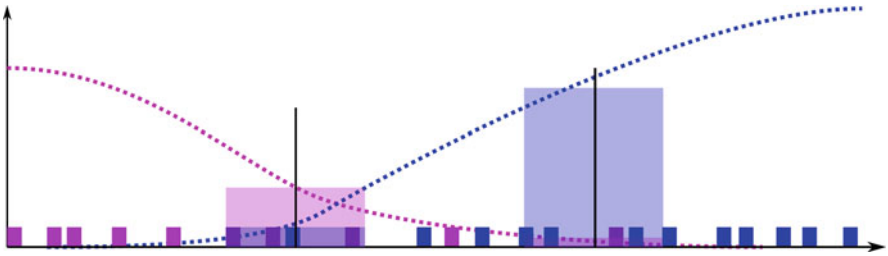


**Fig. 5.5** The unknown (dashed) likelihood function can locally be estimated by binning over a neighborhood with $k$ neighbors and computing the ratio of class members in this neighborhood to the number $k$. The estimate becomes better for larger values of $k$ as long as the likelihood function is linear in this neighborhood

$$P(y=c|\mathbf{x}) \approx \frac{\sum_{X_i \in \text{nbs}(\mathbf{x})} |y_i = c|}{|\text{nbs}(\mathbf{x})|}. \tag{5.11}$$

The approximation quality depends on the size of the local neighborhood and the local sample density. The larger the size, the less likely is $p(\mathbf{x}|\, y=c)$ locally linear in this neighborhood. From this point of view, a small neighborhood is preferred. However, a smaller neighborhood will contain fewer samples.

The adaptive neighborhood size of the kNN classifier is quite appropriate. Low density regions in feature space correspond to low probabilities of the likelihood function and are more likely to have slowly, approximately linear changing function values. The opposite is true for high density regions that may include highly non-linear modal peaks of the likelihood function.

The smallest reasonable neighborhood size for a kNN classifier is $k = 1$. This is called a *nearest neighbor classifier*. It provides a very rough estimate for the a posteriori probability for some class which is either 1.0 (if this sample belongs to this class) or 0.0 (for all other classes). A better estimate results from higher values of $k$. If it is too high, however, the approximation worsens. In the extreme case of $k = M$ the neighborhood always includes all samples from the training data. It then approximates a priori instead of a posteriori probabilities for class labels of an unseen sample.

The best value for $k$ is a compromise that avoids bias toward a priori probability of class membership while not losing accuracy due to violation of the linearity assumption. In reality, fairly small values are commonly used (see Fig. 5.6 for an example with $k = 5$ on the MNIST data). Higher values indicate a densely packed feature space where a generative model would be more informative.

As $k$ is a hyperparameter, finding an optimal value may be carried out by a grid search with different values being tried on training data and evaluated on validation data. Various more advanced methods to learn an optimal value of $k$ from the training data exist as well, e.g., Zhang et al. (2017) or Gong and Liu (2011).

Since the kNN classifier is able to estimate a posteriori probabilities for arbitrary likelihood functions it is more versatile than NCC. However, for a very sparsely


**Fig. 5.6** Classifying MNIST with a kNN classifier ($k = 5$) using pixel values as input features produces an accuracy of 97% (compare this to the 80% in Fig. 5.2 when using a nearest centroid classifier on the same features; compare also the incorrectly labeled digits in the two figures)

populated feature space, it may overfit to the training data (an example is Tamatjita & Mahastama, 2016).

### 5.2.2 *Efficient Estimation by Space Partitioning*

Training a kNN classifier is just a mapping of training samples to feature space. However, such naïve implementation would be computationally expensive. Each classification of an unseen sample requires to inspect the complete set of training samples for finding the $k$ closest neighbors.

The search for nearest neighbors becomes simpler if far-away samples are not included in the search. During a set-up phase, the feature space is partitioned in a way that this exclusion can be carried out by inspecting just a small subset of samples. Feature space is partitioned repeatedly by hyperplanes into two subspaces until sample distribution in each subspace is sufficiently simple. Partitioning is documented in a binary space partitioning tree (BSP tree, see Fuchs et al., 1980 or any computer graphics textbook).

Binary space partitioning for efficient kNN inference often uses a specific BSP tree, the *k-d tree*. The hyperplanes of a $k$-d tree partition a $t_k$-dimensional space always parallel to one of the $t_k$ coordinate axes of the space. The algorithm is initialized and started by the following steps:

- Initialize the $k$-d tree (define root and assign feature space to root).
- Set feature space to current subspace and $k_{\text{actual}}$ to the first axis of feature space.
- Call *partitioning*( ) with current subspace.

The function *partitioning*( ) recursively creates new subspaces until none of the subspaces contains samples. It carries the following steps as long as the current subspace contains samples:

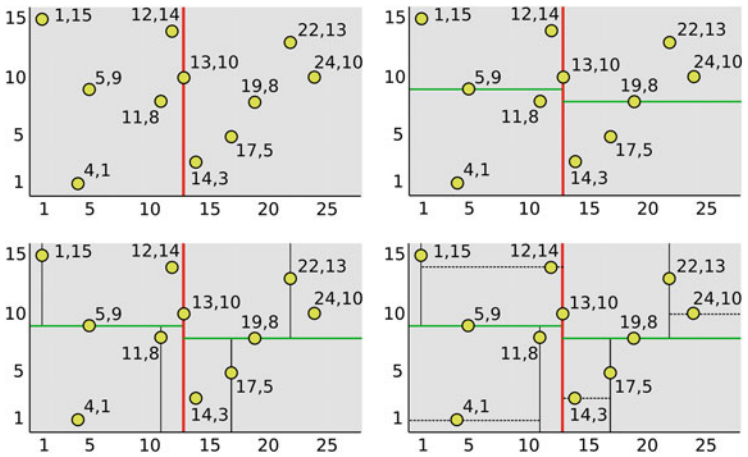



**Fig. 5.7** A *k*-d tree partitions a *k*-dimensional space along hyperplanes through samples that are coplanar to the co-ordinate axes until no samples in one of the subspaces exist

- Select the next coordinate axis $k_{\text{actual}}$.
- Sort samples of the subspace along $k_{\text{actual}}$
- Determine the median $m$ of the sorted samples along $k_{\text{actual}}$
- Partition the subspace at $k_{\text{actual}}$ in two new subspaces with the sample on the partition boundary.
- Document the new partition boundary in the *k*-d tree by creating two child nodes to the current node.
- Assign median sample to the current node.
- Assign samples with coordinates smaller than $m$ to the left child node and all others to the right child node.
- Call *partitioning*( ) with the two new subspaces and the next axis of feature space.

Finally, all samples are nodes of the *k*-d tree and the subspaces are empty (see Fig. 5.7 for an example in 2-d feature space).

For determining the nearest neighbor of a sample **x**, the tree is traversed from the root to a leaf (see Fig. 5.8). The rule is to descend to the left if the coordinate in the node is smaller than the coordinate of **x** that corresponds to the axis of this node and to the right otherwise.

The distance between **x** and the sample represented by the leaf is a first guess for the nearest neighbor. It may not be the closest neighbor since the partitioning order along coordinate axes for creating the *k*-d tree has been arbitrary. Hence, the path is traced back. At each node it is determined whether its coordinates are closer to **x**. The shortest distance and sample coordinates are updated if necessary. If neighboring subspaces represent samples with distance closer than the current shortest distance, the corresponding subtrees are traversed.

The computational cost depends on the depth of the *k*-d tree. For $M$ samples it is in the range of $O(\log M)$ which is a substantial reduction compared to the naïve

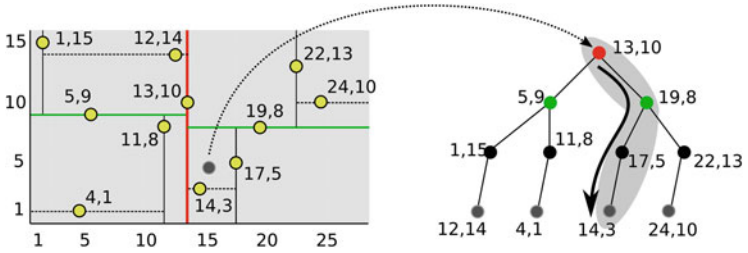



**Fig. 5.8** Finding the nearest neighbor for an unknown sample requires to traverse the tree from the root to one of its leaves. Traversal is always to the left, if the value of the co-ordinate of the unknown sample is less than that of the separating hyperplane and to the right otherwise. The closest neighbor is one of the nodes along this path

approach where always all $M$ samples need to be inspected. Extending this to find the $k$-nearest neighbors requires that during tree traversal a list of $k$ neighbors is kept and updated. The computational cost is then $O(k \log M)$.

Creation of the tree is expensive, however. For each partition the list of samples needs to be sorted along the current axis. Sorting a list of $M$ values carries a cost of $O(M \log M)$. Tree construction is faster if the sample for each partitioning is selected at random but then the tree is no longer balanced. The computational costs for determining nearest neighbors would increase.

**Classes and Functions in Python**

The kNN classifier is part of the submodule `sklearn.neighbors` in SciKit-learn. An unknown sample `X_unknown` can be classified by a kNN classifier fitted to an array of feature vectors `X` and corresponding labels `y` in the following way:

```
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X, y)
print(knn.predict([X_unknown]))
```

Fitting comprises the creation of the $k$-d-tree and takes some time (this is the default, other space partitioning functions may be selected, see documentation of SciKit-learn on nearest neighbor algorithms).

## 5.3 Exercises

### 5.3.1 Programming Project P5.1: Features and Classifiers

Compare the performance of different features with different classifiers on the MNIST data. For features, use

- The gray values themselves in a reduction of the image to 24 × 24 pixels (from its original size 28 × 28). Use the function `resize()` from `skimage.transform` to resize the image. The number of features is 568.
- The HOG features. Use a 4 × 4 pixel window, 2 × 2 blocks, and 9 directions to compute HOG features. The number of features is 568. You will have to apply the HOG transform separately on each sample.
- PCA on either of the 2 feature types above and reduction to 100 features each.

For classifiers use NCC and the kNN classifier. Compute accuracies for all combinations of features and classifiers.

What is the impact of the two different feature types? How does PCA influence the results? Comparing different features and different classifiers: What has the larger influence on the result? Can you guess, why this is so?

### 5.3.2 Exercise Questions

- Under what assumptions does a nearest centroid classifier (NCC) yield the most likely class membership according to the Bayesian theorem?
- Why is the use of the Mahalanobis distance an extension of the NCC? Give a condition under which the distance calculation with use of Mahalanobis distance is not a correct solution according to the Bayesian theorem.
- What is the commonality between an NCC based on the Mahalanobis distance and a classification by a kNN classifier? What is a key difference between the two approaches?
- What are the criteria used to determine the value of $k$ of a kNN classifier?
- Why is (naive) prediction by a kNN classifier so costly? What does the computational cost depend on?
- What is the underlying strategy of nearest neighbor prediction using a previously constructed $k$-d tree? How is the computational cost reduced and why is this so?

## References

Fuchs, H., Kedem, Z. M., & Naylor, B. F. (1980). On visible surface generation by a priori tree structures. In *Proceedings of the 7th Annual Conference on Computer Graphics and Interactive Techniques* (pp. 124–133).

Gong, A., & Liu, Y. (2011). Improved KNN classification algorithm by dynamic obtaining K. In *International Conference on Advanced Research on Electronic Commerce, Web Application, and Communication, ECWAC 2011, Guangzhou, China, April 16-17, 2011. Proceedings, Part I* (pp. 320–324). Springer.

Tamatjita, E. N., & Mahastama, A. W. (2016). Comparison of music genre classification using nearest centroid classifier and k-nearest neighbours. In *2016 IEEE International Conference on Information Management and Technology (ICIMTech)* (pp. 118–123).

Zhang, S., Li, X., Zong, M., Zhu, X., & Wang, R. (2017). Efficient kNN classification with different numbers of nearest neighbors. *IEEE Transactions on Neural Networks and Learning Systems, 29*(5), 1774–1785.

# Chapter 6
# Decision Boundaries in Feature Space


**Abstract** Decision boundaries partition feature space so that samples from different classes are in different compartments. Classification applies a predicate on the features of an unknown sample that returns one of the compartments.

Linear decision boundaries for two-class problems will be presented first. The predicate is the sign of a linear equation applied to features of a sample. The decision boundary is a parametrized linear equation where, ideally, the prediction for all samples from training data is true. Support vector machines extend the concept by adding a condition to search for a boundary that is furthest away from all samples. Conditions will be presented, where the distance from the decision boundary is related to the conditional a posteriori probability of a sample to belong to a class given its features. Logistic regression is then introduced as means to find an optimal decision boundary under these conditions. Finally, extensions to multiclass (multinomial) problems and to non-linear decision boundaries are presented.

The chapter is concluded with a section on model ensembles. Instead of using ever more complex classifiers, ensemble models achieve a performance gain by combining simple classifiers. Two different strategies for this, bagging and boosting, are introduced.

Decision boundaries as classifiers are even further away from a generative classification model than the distance-based classifiers discussed in the previous chapter. Inference based on decision boundaries is fast. Once a decision boundary is computed from training data, label prediction consists of evaluating the trained predicate on the unseen data. It carries just a constant computational cost.

Computing a decision boundary requires less information from training data compared to the models discussed in the previous two chapters. It is particularly useful for a high-dimensional feature space or if just a few training samples exist. Both are a rule rather than an exception when images are to be labeled.

Decision boundaries are defined for two-class problems (also called *binary* or *binomial classification*). The concept can be extended to a multiclass problem by reformulating them as a series of binary classification problems. A multitude of different strategies exists to create decision boundaries from training samples. We

© The Author(s), under exclusive license to Springer Nature Singapore Pte Ltd. 2024

K. D. Toennies, *An Introduction to Image Classification*,
https://doi.org/10.1007/978-981-99-7882-3_6

will limit ourselves to methods that are closely related to the kind of decision making in neural networks that will be discussed later in the book.

## 6.1 Heuristic Linear Decision Boundaries

Decision boundaries for binary problems assume that samples of the two classes are distributed in feature space such that they can be separated by a predicate which is simple to evaluate. Parameters of the predicate are determined based on labeled training data. Linear decision boundaries are an example for such a predicate. Inference consists of the evaluation of a linear equation in feature space with parameters derived from labeled training data.

### *6.1.1 Linear Decision Boundary*

Classification of an unseen sample for a binary classification problem by a linear decision boundary consists of determining on which side of a decision boundary the sample is located. This boundary is a line, plane, or hyperplane in feature space. A sample with feature vector $\mathbf{x} = (x_1\ x_2\ \ldots\ x_N)^{\mathrm{T}}$ is on the boundary if

$$w_0 + w_1 x_1 + \cdots + w_N x_N = 0. \tag{6.1}$$

The weight vector $\mathbf{w} = (w_1\ \ldots\ w_N)^{\mathrm{T}}$ together with the bias weight $w_0$ represents the hyperplane. Extending the weight vector to $\widehat{\mathbf{w}} = (w_0\ w_1\ \ldots\ w_N)^{\mathrm{T}}$ and the feature vector to $\widehat{\mathbf{x}} = (1\ \ x_1 \ldots x_n)^{\mathrm{T}}$, a sample with features $\widehat{\mathbf{x}}$ sits on the decision boundary, if the scalar product is $\widehat{\mathbf{w}}^{\mathrm{T}}\,\widehat{\mathbf{x}} = 0$. For $\widehat{\mathbf{w}}^{\mathrm{T}}\,\widehat{\mathbf{x}} > 0$ the sample is located in the direction of the normal on the hyperplane (i.e., in direction $\mathbf{w}^{\mathrm{T}}$) and for $\widehat{\mathbf{w}}^{\mathrm{T}}\,\widehat{\mathbf{x}} < 0$ it is located in the opposite direction. Once suitable weights $\widehat{\mathbf{w}}$ have been determined, inference just evaluates the plane equation

$$c(\widehat{\mathbf{x}}) = \widehat{\mathbf{w}}^{\mathrm{T}}\,\widehat{\mathbf{x}} \tag{6.2}$$

for an unseen sample with features $\widehat{\mathbf{x}}$ and assigns it to one of the two classes $y = \{-1,1\}$ according to the sign of $c$.

A simple heuristic for finding suitable weights $\widehat{\mathbf{w}}$ based on training data $X = \{\mathbf{x}_1,\ldots,\mathbf{x}_M\}$ is to repeatedly

- select a sample from $X$,
- plug it into Eq. 6.2,
- make a correction for $\widehat{\mathbf{w}}$ if $c(\widehat{\mathbf{x}}_i)y_i < 0$, i.e., if the sign of the computed class is different from that of the true class (see Fig. 6.1).

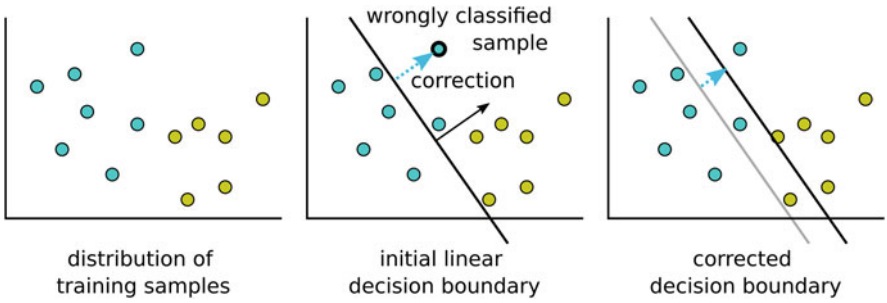



**Fig. 6.1** For optimization, a linear decision is initially placed arbitrarily in feature space. Samples from the training data are randomly selected and the decision boundary is corrected if the selected sample is wrongly classified

The correction step pushes the current hyperplane toward the sample $\widehat{\mathbf{x}}$:

$$\widehat{\mathbf{w}}^{\text{new}} = \widehat{\mathbf{w}}^{\text{old}} + \alpha \cdot c(\widehat{\mathbf{x}})\widehat{\mathbf{x}}. \tag{6.3}$$

The parameter $\alpha$ is a *learning rate* that specifies by how much the hyperplane should be pushed toward the wrongly classified sample. The process is repeated as long as wrongly classified samples are found in the data set or if no further improvement is observed.

The process is similar to a gradient descent optimization. The difference is that optimization is conditional on the sign of $c(\widehat{\mathbf{x}})y$ and the optimization goal is different for $y = 1$ and $y = -1$ resulting in different gradients for these two goals. The conditionality is necessary as the two goals (minimize or maximize the hyperplane equation, respectively) make sense only for incorrectly classified samples. If correctly classified samples were included, the gradient ascent realized by Eq. 6.3 would push the boundary further away from the samples even if the termination condition for this subset of samples is already satisfied. It would let the weight vector oscillate since samples from the other subset would push in the opposite direction.

Even though it is not a gradient descent optimization of a common loss function, similar requirements for carrying out the optimization exist. Sample selection needs to be random without replacement. Going once through the data set is called an *epoch*. Several epochs are needed for arriving at the final result. As it cannot be guaranteed that the methods terminate based on a perfect classification, setting a maximum number of epochs is a necessary auxiliary termination criterion.

### *6.1.2 Non-linear Decision Boundaries*

Results may be unsatisfactory even for perfectly separable training data if the searched decision boundary is non-linear. Such decision boundary, e.g., a

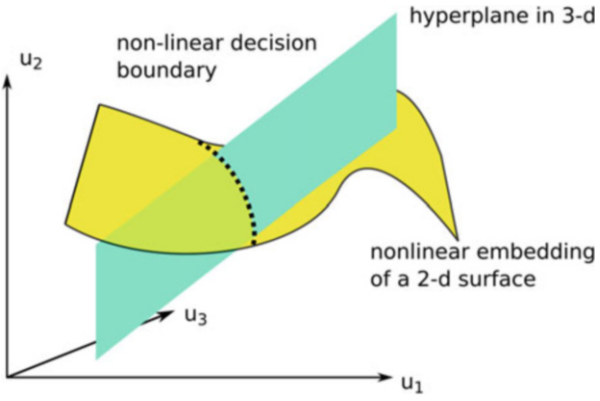



**Fig. 6.2** A non-linear decision boundary can be defined by embedding the original feature space (yellow) non-linearly in a higher-dimensional space. A linear decision boundary in this space (green) is non-linear at the intersection with the embedding (dashed line)

polynomial of degree higher than 1, can be used instead but computing its parameters and evaluation of the predicate on sample features become more difficult.

An elegant solution is to embed the original features space as a non-linear manifold in a higher-dimensional space so that each linear hyperplane in this space corresponds to a non-linear decision boundary in the original feature space (see Fig. 6.2).

An example for a 2-d feature space $(x_1,x_2)$ and a polynomial of degree 2 is the following embedding:

$$u_1 = x_1, \quad u_2 = x_2, \quad u_3 = x_1 x_2, \quad u_4 = x_1^2, \quad u_5 = x_2^2.$$

A hyperplane in the five-dimensional feature space corresponds to a polynomial of degree 2 in the original space. It comes at the cost of a substantial increase in dimensions. It may be acceptable for the example above but the strategy quickly becomes unsuitable if the number of dimensions in the original space and/or the degree of the polynomial decision boundary increases.

### *6.1.3 Solving a Multiclass Problem*

A decision boundary provides a predicate for binary classification problems. It is not easily extended to more than two classes as a hypersurface separates just two subspaces. A strategy for extension to multiclass problems without changing the basic concept is to reduce it to a sequence of binary problems, e.g.,

– by a *one-vs.-all approach* that creates $C$ binary classification problems for a $C$-class-task (see Fig. 6.3). Each of the $C$ classes is tested against samples from all remaining classes. The final decision is made based on some score from the individual classification results. An example would be to gather all positive

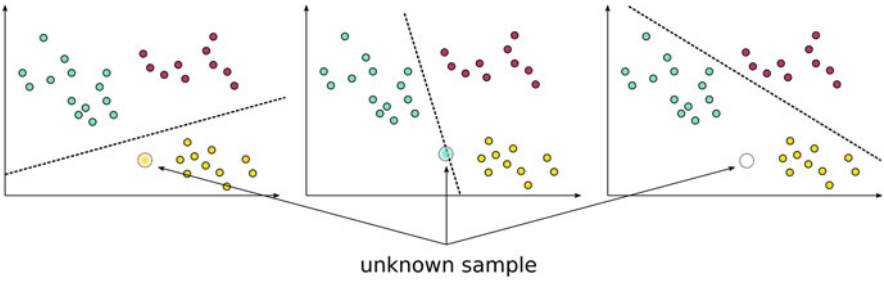



**Fig. 6.3** Example of the one-vs.-all classification, where the sample has twice been classified positively. The final decision will be the yellow class as this decision found the sample further away from the decision boundary

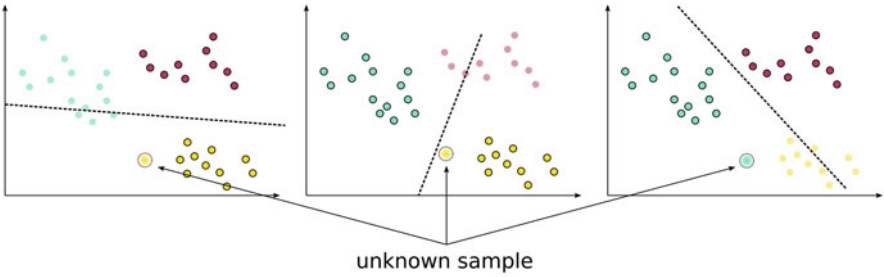



**Fig. 6.4** In a one-vs.-one approach, the classification is carried out as a series of binary decisions between all possible pairs of classes. In this example, the selection is by majority vote. The yellow class was selected

results and assign the sample to the one class to which it had the furthest distance from the decision boundary.

– by a *one-vs.-one approach* that creates $C \cdot (C-1)$ pairwise classification problems for each possible pair of classes and assigns the one class to the sample that has not been ruled out by any of these tests. Another possibility to come to a decision is a majority vote on selected classes (see Fig. 6.4).

The two approaches become expensive if the number of classes increases. Furthermore, choosing the best approach is impossible unless additional experiments with the different approaches have been carried out (then, the choice is just an additional hyperparameter).

### 6.1.4 Interpretation of Sample Distance from the Decision Boundary

Intuitively, a decision for a class may be assumed more likely to be correct the further a sample is away from the decision boundary. However, is this really so? Three conditions have to be met:

- There must be some relation between a posteriori probability of a sample to belong to a class given its features and the distance from the decision boundary.
- A decision boundary defined by locations of equal a posteriori probability for the two classes has to be linear.
- The boundary determined by training from samples has to be just this linear boundary.

The latter two conditions can, at the most, only partially be true for the classifier presented here because it relies on so many heuristic assumptions. However, sometimes the first condition holds so that distances from the heuristically found hyperplane can be assumed to be at least somewhat close to the desired estimate of the maximum a posteriori probability.

If we have a binary classification problem with classes $y = \{-1,1\}$, the probability $P(y)$ for a class $y$ is given by the Bernoulli distribution

$$P(y) = \begin{cases} p^y(1-p)^{1-y} & \text{if } y \in \{-1,1\} \\ 0 & \text{otherwise} \end{cases}, \tag{6.4}$$

with $p = P(y|\mathbf{x})$ being the a posteriori probability for label $y$. The odds (a.k.a. chance) for one of the two classes is the ratio of the two probabilities

$$\text{odds}(1) = \frac{P(y)}{1-P(y)} = \frac{p}{1-p}. \tag{6.5}$$

Using odds for an event instead of the probability has the advantage that the proportionality factor (the evidence) from the Bayesian Theorem cancels out.

Under some conditions the odds can be mapped to a linear expression that is related to the distance from the decision boundary. If, for instance, we assume that the two likelihood functions are Gaussians with the same covariance matrices and different expected value vectors $\boldsymbol{\mu}_1$ and $\boldsymbol{\mu}_2$, the a posteriori probabilities of a sample with features $\mathbf{x}$ are

$$P(y=1|\mathbf{x}) = \frac{1}{\sqrt{(2\pi)^m \cdot \det(\boldsymbol{\Sigma}_1)}} \exp\Big(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_1)^{\mathrm{T}}\boldsymbol{\Sigma}_1^{-1}(\mathbf{x}-\boldsymbol{\mu}_1)\Big) \times \frac{P(Y=1)}{P(\mathbf{x})}. \tag{6.6}$$

$$P(y=-1|\mathbf{x}) = \frac{1}{\sqrt{(2\pi)^m \cdot \det(\boldsymbol{\Sigma}_2)}} \exp\Big(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_2)^{\mathrm{T}}\boldsymbol{\Sigma}_2^{-1}(\mathbf{x}-\boldsymbol{\mu}_2)\Big) \times \frac{P(Y=0)}{P(\mathbf{x})}. \tag{6.7}$$

The odds for $y = 1$ is the ratio of these two probabilities

$$\begin{aligned}\text{odds}(1) &= \frac{p(\mathbf{x}|y=1)}{p(\boldsymbol{x}|y=-1)} \\ &= \frac{\sqrt{(2\pi)^m \cdot \det(\boldsymbol{\Sigma}_2)}}{\sqrt{(2\pi)^m \cdot \det(\boldsymbol{\Sigma}_1)}} \frac{P(y=1)}{P(y=-1)} \frac{\exp\Big(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_1)^{\mathrm{T}}\boldsymbol{\Sigma}_1^{-1}(\mathbf{x}-\boldsymbol{\mu}_1)\Big)}{\exp\Big(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_2)^{\mathrm{T}}\boldsymbol{\Sigma}_2^{-1}(\mathbf{x}-\boldsymbol{\mu}_2)\Big)}\end{aligned} \tag{6.8}$$

which after some rearrangement of terms gives

$$\text{odds}(1) = k \cdot \exp\Big(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_1)^{\mathrm{T}}\boldsymbol{\Sigma}_1^{-1}(\mathbf{x}-\boldsymbol{\mu}_1) + \frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_2)^{\mathrm{T}}\boldsymbol{\Sigma}_2^{-1}(\mathbf{x}-\boldsymbol{\mu}_2)\Big) \tag{6.9}$$

with

$$k = \frac{\sqrt{(2\pi)^m \cdot \det(\boldsymbol{\Sigma}_2)}}{\sqrt{(2\pi)^m \cdot \det(\boldsymbol{\Sigma}_1)}} \frac{P(y=1)}{P(y=-1)} = \frac{\sqrt{\det(\boldsymbol{\Sigma}_2)}}{\sqrt{\det(\boldsymbol{\Sigma}_1)}} \frac{P(y=1)}{P(y=-1)}. \tag{6.10}$$

As the two covariance matrices are equal, it further simplifies to

$$k = \frac{P(y=1)}{P(y=-1)}. \tag{6.11}$$

The logarithm of the chance is

$$\log \text{odds}(1) = \log k - \frac{1}{2} \times \Big[(\mathbf{x}-\boldsymbol{\mu}_1)^{\mathrm{T}}\boldsymbol{\Sigma}_1^{-1}(\mathbf{x}-\boldsymbol{\mu}_1) - (\mathbf{x}-\boldsymbol{\mu}_2)^{\mathrm{T}}\boldsymbol{\Sigma}_2^{-1}(\mathbf{x}-\boldsymbol{\mu}_2)\Big]. \tag{6.12}$$

The logarithmic odds are called *logits*. In the specific case that $\boldsymbol{\Sigma}_1 = \boldsymbol{\Sigma}_2 = \boldsymbol{\Sigma}$ and since the covariance matrices are symmetric, the logits can be written as linear equation:

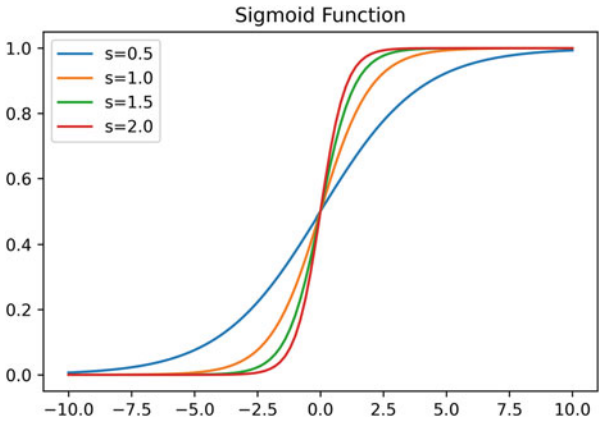



**Fig. 6.5** Sigmoid functions for different scale factors

$$\log \text{odds}(1) = \log k - \left(2\mathbf{x}^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}(\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1) + \boldsymbol{\mu}_1^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_2\right). \tag{6.13}$$

Under these assumptions the decision boundary is indeed linear with optimal $\mathbf{w} = \boldsymbol{\Sigma}^{-1}(\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1)$ and $w_0 = \boldsymbol{\mu}_1^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_2$. The distance $\widehat{\mathbf{w}}^{\mathrm{T}}\widehat{\mathbf{x}}$ from the boundary is the scaled logarithm of the odds:

$$\log \text{odds}(1) = \widehat{\mathbf{w}}^{\mathrm{T}}\widehat{\mathbf{x}}, \tag{6.14}$$

where $\widehat{\mathbf{w}}$ depends on the covariance matrix and the distance between $\boldsymbol{\mu}_1$ and $\boldsymbol{\mu}_2$.

Given this, the a posteriori probability can be computed from the inverse $\exp\left(\widehat{\mathbf{w}}^{\mathrm{T}}\widehat{\mathbf{x}}\right)$ of the logits:

$$P(y = 1|\mathbf{x}) = \frac{\exp\left(\widehat{\mathbf{w}}^{\mathrm{T}}\widehat{\mathbf{x}}\right)}{1 + \exp\left(\widehat{\mathbf{w}}^{\mathrm{T}}\widehat{\mathbf{x}}\right)} = \frac{1}{1 + \exp\left(-\left(\widehat{\mathbf{w}}^{\mathrm{T}}\widehat{\mathbf{x}}\right)\right)}. \tag{6.15}$$

This function is called *sigmoid function* (or *logistic function*, see Fig. 6.5). It is usually written as

$$S(d) = \frac{1}{1 + \exp(-sd)}, \tag{6.16}$$

where $d$ is the signed distance to the decision boundary and $s$ is the scale factor mentioned above. If the assumptions about the likelihood functions are correct and the determined estimate for the decision boundary is the true boundary, applying the sigmoid function to the computed (scaled) signed distance with proper value for $s$ delivers the a posteriori probability.

Before we continue, let us recapitulate what we have done as it will play a major role in the remainder of this book when interpreting classifier models. We defined a

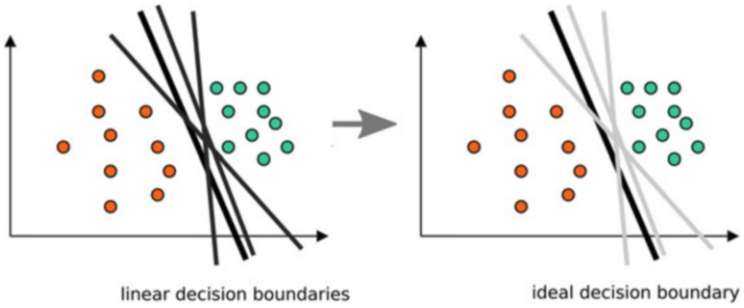



**Fig. 6.6** Many decision boundaries may be possible to separate samples from different classes. The heuristics does not provide a means to select an ideal decision boundary from it

probability distribution for a binary classification task from the odds of one class being the truth versus the other class. Under some assumptions the logarithm of this chance is a linear function of sample features. It is called a *log-linear model* (or model with log-linear odds). If, under these assumptions, the decision boundary is the locus with odds equal to 1, it allows direct interpretation of distances from a linear decision boundary as conditional class probability of a sample given its features.

The first condition bears similarities to constraints used for the nearest centroid classifier and, as stated in Sect. 5.1, it can sometimes be assumed to be at least approximately true. However, multimodal likelihood functions or training data that is only separable by a non-linear decision boundary already show that there are relevant exceptions to it.

The second condition is not fulfilled by a heuristically chosen decision boundary. It seems plausible that linearly separable training data indicate that the optimal decision boundary is somewhere between samples from different classes. However, an arbitrary hyperplane between the samples, as produced by the method above, diverges from the optimal hyperplane in an unknown way.

In summary, heuristic linear decision boundaries are simple to generate from training samples but with a number of problems associated with it:

- The optimization is difficult to characterize as it does not contain a model about the best decision boundary given the training data (see Fig. 6.6).
- The extension to non-linear decision boundaries becomes awkward for a high-dimensional feature space.
- The extension to multiclass problems is not straightforward. It is not clear what is the best way to do this.

## 6.2 Support Vector Machines

Support vector machines (SVM, see the tutorial of Fletcher, 2009 for a gentle introduction or the paper of Mammone et al., 2009) extend the concept used in the previous section in three ways:

- The solution is unique.
- Optimization depends on a common loss function.
- Non-linear decision boundaries may be defined in the original feature space.

A linear SVM determines a subset of training samples (the *support vectors*) in feature space that define a hyperplane with maximum margin of separation. The margin of separation is a corridor around the hyperplane that is free of training samples (see Fig. 6.7).

The motivation behind this is similar to that of using kernel density estimators. If training samples are unbiased, they can be taken as expected value of subsets of the sample population in a local neighborhood around each sample. If this is true, locations close to the decision boundary may be occupied by unseen samples that may be erroneously classified. If the localization assumption guarantees that subsets of unseen samples represented by the training data do not overlap, the error rate of the classifier is completely determined by samples closest to the decision boundary. Hence, maximizing the margin of separation should minimize the error rate.

The argumentation requires that training samples are perfectly separable. We will see later that it can be extended to cases where training samples from different classes overlap. Similar to the argumentation in the previous section, distance from the decision boundary can be interpreted as conditional class probability if requirements listed in Sect. 6.1.4 are met. Intuitively, selecting a hyperplane that is furthest away from samples of different classes is more likely close to the unknown optimal hyperplane than an arbitrary hyperplane.

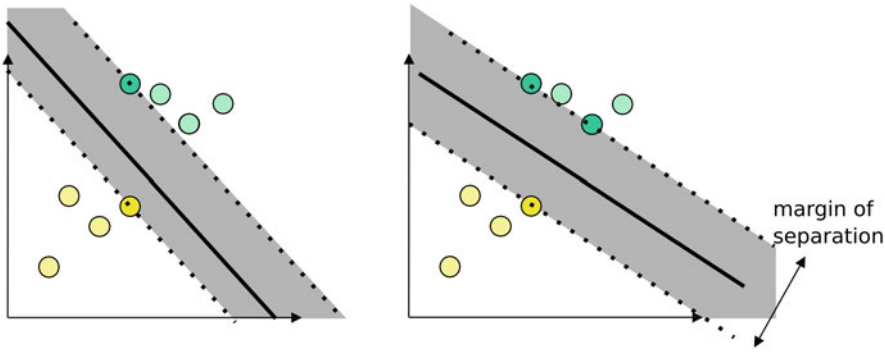



**Fig. 6.7** The margin of separation defines a corridor around a decision boundary that does not contain samples from the training data. Different decision boundaries that separate the classes will have different margins of separation. The samples that are on the boundary of this corridor are the support vectors

### 6.2.1 Optimization of a Support Vector Machine

Let us assume a binary classification problem for which training data exists. For each training sample $\widehat{\mathbf{x}}_i = (1\ x_{i,1} \ldots x_{i,N})$ its class label $y_i \in \{-1,1\}$ is known. The goal is to find a hyperplane in feature space $H(\widehat{\mathbf{w}}) : \widehat{\mathbf{w}}^{\mathrm{T}}\widehat{\mathbf{x}} = 0$ that separates samples from the two classes while maximizing the distance of all samples to $H(\widehat{\mathbf{w}})$. It can be formulated as a constrained optimization problem. The constraint ensures correct classification. It is

$$y_i\left(\widehat{\mathbf{w}}^{\mathrm{T}}\widehat{\mathbf{x}}_i\right) \geq 1 \tag{6.17}$$

for all $\widehat{\mathbf{x}}_i$ of the training data. The constraint enforces strict separation of samples, i.e., no sample from the training data may sit on the hyperplane. An arbitrary value larger than 0 may be chosen on the right-hand side of Eq. 6.17 since the weight vector $\mathbf{w}$ may be arbitrarily scaled.

The optimization goal is to find weights $\widehat{\mathbf{w}}$ so that the hyperplane $H(\widehat{\mathbf{w}})$ fulfills the constraint of Eq. 6.17 and has maximum distance to all samples. The maximum distance is

$$\widehat{\mathbf{w}}_{\mathrm{opt}} = \underset{\widehat{\mathbf{w}}}{\mathrm{argmin}} \left( \frac{\left|\widehat{\mathbf{w}}^{\mathrm{T}}\widehat{\mathbf{x}}_i\right|}{\|\mathbf{w}\|} \right). \tag{6.18}$$

The closest distance to the decision boundary depends on the samples that are the closest to this boundary. These are the support vectors $\mathbf{x}_{\mathrm{s}}$.

The constraint of Eq. 6.18 will be $y_{\mathrm{s}}\left(\widehat{\mathbf{w}}^{\mathrm{T}}\widehat{\mathbf{x}}_{\mathrm{s}}\right) = 1$ for all support vectors. Hence, the distance of samples $\mathbf{x}_s$ from the decision boundary is

$$r_{\mathrm{s}} = \frac{1}{\|\mathbf{w}\|} \tag{6.19}$$

on either side of the hyperplane (see Fig. 6.8). The width of the corridor around $H(\widehat{\mathbf{w}})$ is $\frac{2}{\|\mathbf{w}\|}$. It is maximized if the following expression is minimized:[1]

$$\mathbf{w}_{\mathrm{min}} = \underset{\mathbf{w}}{\mathrm{argmin}} \frac{1}{2}\|\mathbf{w}\|^2 = \underset{\mathbf{w}}{\mathrm{argmin}} \frac{1}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w} \tag{6.20}$$

subject to the constraint

[1] Here, the square of the Euclidean distance is used as regularizing penalty term (hence it is called L2-regularization).

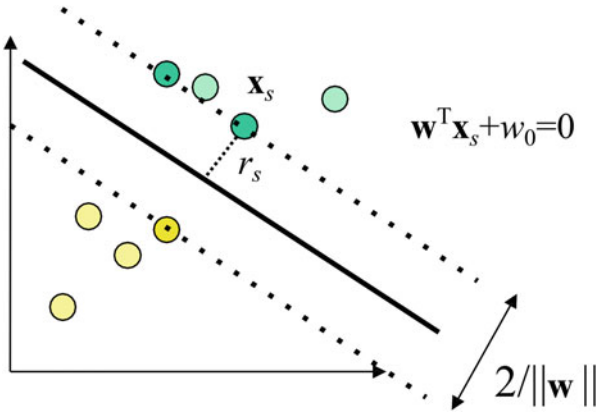



**Fig. 6.8** The margin of separation is defined by the length of the weight vector $\mathbf{w}$

$$y_i\left(\widehat{\boldsymbol{w}}^{\mathrm{T}}\widehat{\mathbf{x}}_i\right) - 1 \geq 0. \tag{6.21}$$

For finding the optimum, the Lagrangian formalism is used to define an unconstrained optimization problem. The desired solution is at a local minimum in the subspace defined by Eq. 6.21. It is usually not a minimum of Eq. 6.20 so that the solutions are saddle points in the space of $\widehat{\mathbf{w}}$.

For the constrained minimum, the gradients of Eqs. 6.20 and 6.21 must point in the same direction although their lengths may be different. This is expressed by

$$J(\mathbf{w}, w_0, \alpha) = \frac{1}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w} - \sum_{i=1}^{M} \alpha_i \left[y_i\left(\mathbf{w}^{\mathrm{T}}\mathbf{w}_i + w_0\right) - 1\right], \tag{6.22}$$

where the Lagrange multipliers $\alpha_i$ account for the different gradient lengths.

Finding saddle points of $J$ is a necessary condition for finding the optimal weight vector $\widehat{\mathbf{w}}_{\mathrm{opt}}$. Computing partial derivatives with respect to $(\mathbf{w}, w_0, \alpha)$ and setting them to 0 results in the following expression to be optimized:

$$-\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j \mathbf{x}_i^{\mathrm{T}}\mathbf{x}_j + \sum_{i=1}^{N}\alpha_i. \tag{6.23}$$

This expression is interesting as the sample vectors occur solely in a scalar product. It will later help to find an extension to non-linear decision boundaries. Optimizing Eq. 6.23 results in

$$\mathbf{w}_{\mathrm{opt}} = \sum_{i=1}^{M} \alpha_i y_i \mathbf{x}_i \tag{6.24}$$

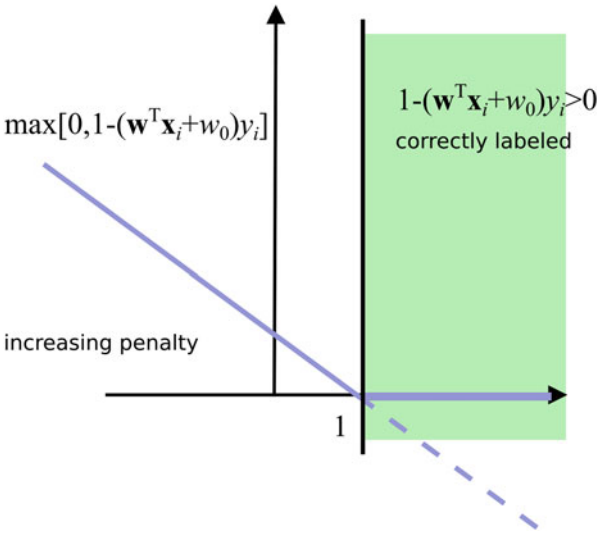



**Fig. 6.9** The penalty term has an influence only, if a sample is on the decision boundary or on the wrong side of it. It increases with the distance of the wrongly classified sample from the decision boundary

and an additional constraint for the bias $w_0$. The vector $\mathbf{w}_{\text{opt}}$ is a weighted combination of the training samples. It can be shown that Lagrange multipliers are zero for all correctly classified samples which are not on the boundary of the empty corridor around $H$. Hence, the vector $\mathbf{w}_{\text{opt}}$ is represented by the (very few) support vectors.

The computing time for the optimization depends on the number of support vectors and that of summands in the double sum of Eq. 6.23. Hence, finding the support vectors can be time-consuming if the number of samples in the training data is large. Inference is quick, however, as it requires a single evaluation of the hyperplane equation with features of an unseen sample.

### 6.2.2 *Soft Margins*

In reality, classes are rarely linearly separable. Hence, several methods to relax this condition were presented. The first solution introduced a slack variable which moved every sample of the training data depending on its class in or against the direction of the hyperplane's normal by a certain amount. Data became separable if the overlap was less than this amount.

Another approach, that does not require the difficult estimation of expected overlap between classes, is the introduction of *soft margins*. Arbitrary overlap is accepted but it gets increasingly penalized (see Fig. 6.9):

$$\widehat{\mathbf{w}}_{\text{opt}} = \underset{\mathbf{w}, w_0}{\operatorname{argmin}} \frac{1}{M} \sum_{i=1}^{M} \max\left[0, 1 - \left(\mathbf{w}^{\mathrm{T}}\mathbf{x}_i + w_0\right) y_i\right] + \lambda\left(\frac{1}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w}\right). \tag{6.25}$$

The value of $\lambda$ (in the Python implementation, the variable is called $C$) determines how serious an excess of the hyperplane (as indicated in the second term of this equation) is for the computation of the optimal weight vector.

The expression $\|\mathbf{w}\|^2 = \mathbf{w}^{\mathrm{T}}\mathbf{w}$ is again the L2-regularizer that we already encountered in Sect. 6.2.1. Minimization of Eq. 6.25 will produce weights that are similar to each other since the square of the Euclidean distance penalizes large weights. Hence, it will prevent single features to overly influence the classification model. Using a L1-regularizer that computes the sum of absolute differences instead of the Euclidean distance will have a different impact in that it will prefer a weight vector with many weights being zero. Hence, it produces a sparse model where few weights make up the classification model. L2-regularization will be most helpful when a complex decision boundary needs to be trained supported by a sufficient number of training samples. L1-regularization will be helpful when a sparse model is sought because the number of features is high given the number of training samples.

Optimization is similar to that without soft margins. The weight vector is still a weighted sum of support vectors. Because of the overlap of the classes in feature space, a potentially large number of samples are misclassified. Since Lagrange multipliers are zero only for samples that are correctly classified, the number of support vectors can get very large which increases the computation costs for weight optimization.

### 6.2.3 Kernel Functions

Support vector machines compute a linear decision boundary. They can be extended to non-linear decision boundaries by embedding the original space as curved manifold in a higher-dimensional space as in Sect. 6.1.2.

However, Eqs. 6.23 and 6.24 hint at a way to avoid the mapping in higher-dimensional space while still producing the same result. In Eq. 6.23 the term to be optimized contains the feature vectors only as scalar product between two vectors. The scalar product of mappings of original features $\mathbf{x}$ to higher-dimensional space $\Phi(\mathbf{x})$ for two feature vectors would then be $\Phi(\mathbf{x}_1)\Phi(\mathbf{x}_2)$. If now a so-called *kernel function* $k$ on the original features exists, such that

$$k(\mathbf{x}_1, \mathbf{x}_2) = \Phi(\mathbf{x}_1)\Phi(\mathbf{x}_2) \tag{6.26}$$

the explicit mapping to higher-dimensional space in order to optimize Eq. 6.23 is not necessary. The result can be achieved as well by optimizing $-\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j k\left(\mathbf{x}_i, \mathbf{x}_j\right) + \sum_{i=1}^{N}\alpha_i$ instead.

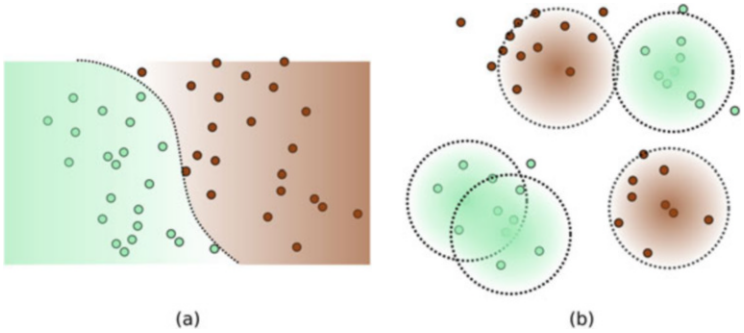



**Fig. 6.10** Comparison between (**a**) polynomial and (**b**) radial kernels. Support vectors of polynomial base function define distances from a non-linear decision boundary, while support vectors of radial base functions describe distances from cluster centers

Kernel functions like this do indeed exist. An example is the polynomial kernel for polynomial boundary surfaces of degree $d$ (see also Fig. 6.10a)

$$k_d\left(\mathbf{x}_i, \mathbf{x}_j\right) = \Phi_d(\mathbf{x}_i)\Phi_d(\boldsymbol{x}_i) = \left(1 + \mathbf{x}_i^{\mathrm{T}}\mathbf{x}_j\right). \tag{6.27}$$

For computing class membership in the higher-dimensional space, the kernel function can be used as well using Eq. 6.24

$$\begin{aligned}\mathbf{w}^{\mathrm{T}}\Phi\left(\mathbf{w}_j\right) + w_0 &= \left(\sum_{i=1}^{M} \alpha_i y_i \Phi\left(\mathbf{x}_i^{\mathrm{T}}\right)\right)\Phi\left(\mathbf{x}_j\right) + w_0 \\ &= \left(\sum_{i=1}^{M} \alpha_i y_i \Phi\left(\mathbf{x}_i^{\mathrm{T}}\right)\Phi\left(\mathbf{x}_j\right)\right) + w_0 = \left(\sum_{i=1}^{M} \alpha_i y_i k\left(\mathbf{x}_i, \mathbf{x}_j\right)\right) + w_0.\end{aligned} \tag{6.28}$$

The number of non-zero support vectors in the sum increases with the implied dimensionality of the kernel function but the computation itself never has to be carried out in this space. Instead, the Euclidean distance between the sample feature vector and the weighted sum of support vectors is replaced by a non-Euclidean distance implied by the kernel function.

Kernel functions have been intensely investigated for classifying data sets, see, e.g., Kecman (2005). From the different kernel functions that have been used, a popular kernel is the *radial base function* (a.k.a. Gaussian kernel):

$$K_\sigma = \exp\left(-\frac{\left\|\mathbf{x}_i - \mathbf{x}_j\right\|^2}{2\sigma^2}\right). \tag{6.29}$$

Whereas a polynomial kernel is able to produce ever more complex decision boundaries between classes, radial base functions define support vectors that represent clusters in the training data (see Fig. 6.10). It is so because the value of $K_\sigma$ decreases fast with increasing difference between a feature vector and a support vector. Hence, only the closest support vector plays a role when determining the distance of a sample. The value of $\sigma$ determines the extent of the local neighborhood around a particular support vector.

Radial base functions may be more appropriate than a polynomial decision boundary if data of some class is assumed to be distributed over several clusters (multimodal distribution). An example are pictures of class samples from different views (see Fig. 6.10b).

There has been a lot of research regarding the use of kernels to increase the expressiveness of a classifier (see the book of Schölkopf & Smola, 2018). If well selected it represents a projection in a space where the (unknown) optimal decision boundary is linear. Hence, it helps to get close to a situation where distances to the decision boundary may rightfully be interpreted as a posteriori probabilities after applying the sigmoid function to it.

The larger expressiveness comes at a cost, however, as it may cause overfitting to the training data. A developer needs to carefully balance expressiveness against the generalization ability of the classifier. It is a difficult task because the true distribution of samples in feature space as well as changes by applying kernel functions cannot be visualized in some meaningful way. It depends on experience and experiments to decide on an appropriate kernel for a given task.

### 6.2.4 *Extensions to Multiclass Problems*

Just as in the previous section, support vector machines define predicates for binary classification problems. If a multiclass problem shall be solved, the one-vs.-all approach or the one-vs.-one approach from Sect. 6.1.3 is used.

**Classes and Functions in Python**
Support vector machines are a class of the SciKit-learn submodule `sklearn.svm`. The class is imported by

```
from sklearn.svm import SVC
```

An object for solving a binary classification problem is constructed, fitted to training data `X_train` with labels `y_train` , and used to predict labels for test data `X_test` by

(continued)

```
svm_clf = SVC(C=1,kernel='rbf')   # C is the parameter for soft
                                  # margins kernel='rbf' selects
                                  # radial base functions (see
                                  # sklearn documentation for
                                  # further kernels)
svm_clf.fit(X_train, y_train)     # find support vectors
y_pred = svm_clf.predict(X_test)  # make predictions for test data
```

The class can be adapted by a number of parameters (see SciKit-learn documentation). For instance, to influence the way, multinomial problems are solved, the parameter `decision_function_shape='ovr'` (default) or `='ovo'` selects between one-vs.-rest and one-vs.-one strategy.

Remember that fitting the SVM to the data, especially when solving a multinomial problem, can be quite time-consuming when the set of training data is large.

## 6.3 Logistic Regression

Support vector machines got us a step closer to the interpretation of distances as a posteriori probabilities. The search for a decision boundary that is furthest from all training samples is likely to be closer to the optimal hypersurface than an arbitrary separating hypersurface. Furthermore, the possibility to use kernel functions may produce new features with a log-linear distribution of odds.

However, it is still a purely heuristic approach. *Linear logistic regression* finds an optimal decision boundary for a log-linear distribution of odds (see Christensen, 2006 for a detailed treatment or Ghojogh & Crowley, 2019 for a short tutorial).

### *6.3.1 Binomial Logistic Regression*

Binomial logistic regression solves a two-class problem for classes $y = \{0,1\}$ for a log-linear model with odds $l\_\text{odds}(1)$ for $\widehat{\mathbf{x}}$ having the label $y = 1$:

$$l\_\text{odds}_{y=1}(\widehat{\mathbf{x}}) = \widehat{\mathbf{w}}^{\mathrm{T}}\widehat{\mathbf{x}}. \tag{6.30}$$

It searches for a sigmoid function that optimally represents class memberships of labeled training data (see Fig. 6.11).

Optimal values for $\widehat{\mathbf{w}} = (w_0\ w_1 \ldots .w_N)^{\mathrm{T}}$ are searched based on a set of training samples $X = \{\mathbf{x}_1,\ldots,\mathbf{x}_M\}$ with associated labels $y=\{y_1,\ldots,y_M\}$. Optimization is done numerically by gradient descent to minimize a loss function. The probability for a sample with features $\mathbf{x}_i$ to belong to class $y = 1$ given a current estimate for $\widehat{\mathbf{w}}$ is

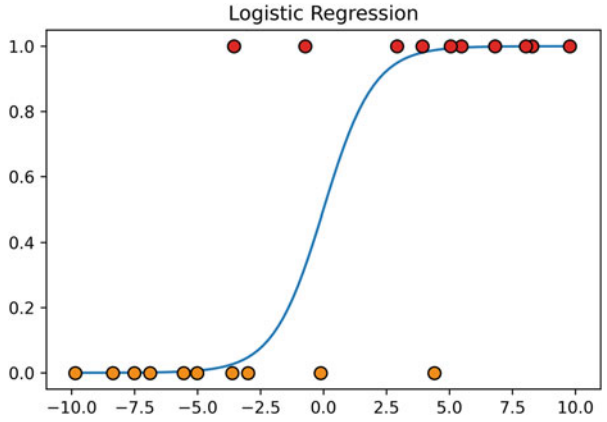



**Fig. 6.11** Logistic regression fits a sigmoid function to the data so that the average discrepancy between sample class probability of a binary classification problem (1 or 0, depending on whether the sample belongs to class 1—red—or not—orange) and the predicted probability is minimum

$$P_{\widehat{\mathbf{w}}}(y=1|\mathbf{x}_i) = \frac{1}{1+\exp\left(-\left(\widehat{\mathbf{w}}^{\mathrm{T}}\widehat{\mathbf{x}}_i\right)\right)}. \tag{6.31}$$

The loss $l$ for a single sample $y$ describes the discrepancy between the true class of a sample (either 0 or 1) and the predicted outcome from the sigmoid function (see Fig. 6.12):

$$l(y;\widehat{\mathbf{w}}) = \begin{cases} -\ln P_{\widehat{\mathbf{w}}}(y|\mathbf{x}_i) & \text{for } y=1 \\ -\ln\left(1-P_{\widehat{\mathbf{w}}}(y|\mathbf{x}_i)\right) & \text{for } y=0 \end{cases}. \tag{6.32}$$

This can be formulated more compactly as

$$l(y;\widehat{\mathbf{w}}) = -y\ln P_{\widehat{\mathbf{w}}}(y|\mathbf{x}_i) - (1-y)\ln\left(1-P_{\widehat{\mathbf{w}}}(y|\mathbf{x}_i)\right). \tag{6.33}$$

Summing up discrepancies for all samples gives the loss function $\mathrm{CE}(\widehat{\mathbf{w}})$ for the current parameter vector $\widehat{\mathbf{w}}$:

$$\mathrm{CE}(\widehat{\mathbf{w}}) = \sum_{i=1}^{M} -y_i\ln P_{\widehat{\mathbf{w}}}(y_i|\mathbf{x}_i) - (1-y_i)\ln\left(1-P_{\widehat{\mathbf{w}}}(y_i|\mathbf{x}_i)\right). \tag{6.34}$$

This loss function is known as (*binary*) *cross-entropy*. It measures the deviation between the distributions of the ground truth labels and those inferred from features of the training samples.

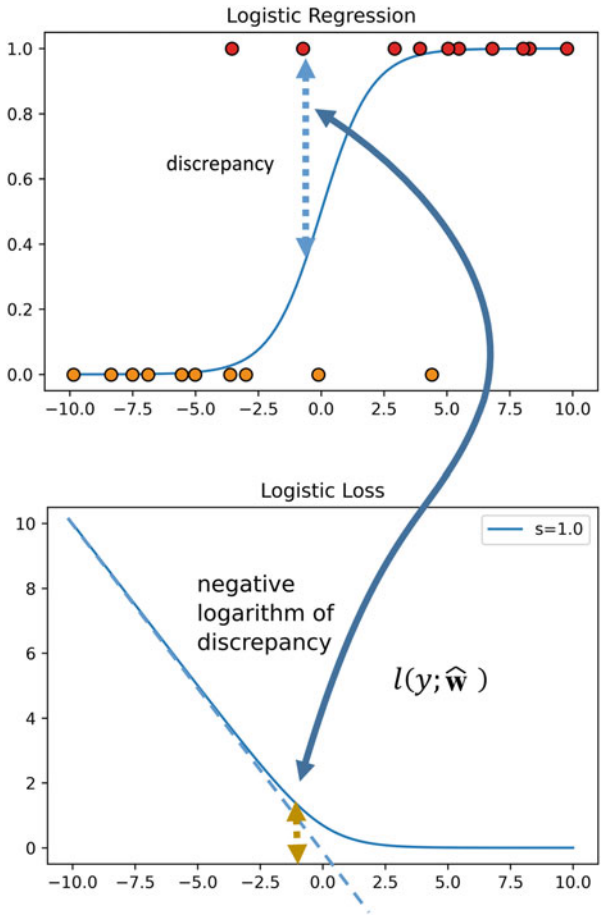



**Fig. 6.12** The loss for a single sample depends on the discrepancy between its ground-truth label and the predicted a posteriori probability

### 6.3.2 *Multinomial Logistic Regression*

Different to support vector machines, logistic regression extends easily to multiclass problems. For $C > 2$ classes a *pivotal class* $c = 0$ is selected and odds of all other classes relative to the pivotal class are computed:

$$l\_\text{odds}_{y=c}(\widehat{\mathbf{x}}) = \widehat{\mathbf{w}}_c^\mathrm{T}\widehat{\mathbf{x}}. \tag{6.35}$$

For each class $c \neq 0$ a separate parameter vector $\widehat{\mathbf{w}}_c$ is estimated from training samples. Each of the functions $l\_\,\text{odds}_{y=c}$ evaluates a different hyperplane equation. However, it is a bit difficult to interpret the solution as a distance to a hyperplane since each class except for $c = 0$ defines its own hyperplane to test a sample against the pivotal class. This is related to linear discriminant analysis that we used for supervised feature reduction in Sect. 3.2.2 and the functions are called *linear*

*discriminant functions*. Each test computes a scaled cosine between $\widehat{\mathbf{w}}_c$ (the weight vector including the weight $w_0$) and the extended sample feature vector $(1\ x_1\ x_2\ \ldots\ x_N)$. The decision depends on this cosine.

The probability of a sample $i$ with feature vector $\mathbf{x}_i$ to belong to some class $c \neq 0$ is then computed by rating the odds for this class against all other odds:

$$P_{\widehat{\mathbf{w}}_c}(y=c|\mathbf{x}_i) = \frac{\exp\left(\widehat{\mathbf{w}}_c^{\mathrm{T}}\widehat{\mathbf{x}}\right)}{1+\sum_{i=1}^{C-1}\exp\left(\widehat{\mathbf{w}}_i^{\mathrm{T}}\widehat{\mathbf{x}}\right)}. \tag{6.36}$$

The probability for $c = 0$ is

$$P_{\widehat{\mathbf{w}}_0}(y=0|\mathbf{x}_i) = 1 - \sum_{i=1}^{C-1} P_{\widehat{\mathbf{w}}_c}(y=c|\mathbf{x}_i) = \frac{1}{1+\sum_{i=1}^{C-1}\exp\left(\widehat{\mathbf{w}}_i^{\mathrm{T}}\widehat{\mathbf{x}}\right)}. \tag{6.37}$$

For finding optimal values for $\widehat{\mathbf{w}}_c$ the *multinomial cross entropy* (also known as *categorical cross entropy*) is used as loss function. If $\mathbf{m}_i$ is an indicator vector of length $C$ with $m_{i,c} = 1$ if the ground truth class for a sample $\mathbf{x}_i$ is $c$ and 0 elsewhere, the loss for this sample is

$$l(\mathbf{x}_i, m_i; \widehat{\mathbf{w}}) = -m_{i,0}\ln\left(1 - P_{\widehat{\mathbf{w}}_0}(y=0|\mathbf{x}_i)\right) - \sum_{c=1}^{C-1} m_{i,c}\ln P_{\widehat{\mathbf{w}}_c}(y=c|\mathbf{x}_i). \tag{6.38}$$

The multinomial cross entropy sums over all samples in the training data yielding

$$\mathrm{CE}(\widehat{\mathbf{w}}) = \sum_{i=1}^{M}\left[-m_{i,0}\ln\left(1 - P_{\widehat{\mathbf{w}}_0}(y=0|\mathbf{x}_i)\right) - \sum_{c=1}^{C-1} m_{i,c}\ln P_{\widehat{\mathbf{w}}_c}(y=c|\mathbf{x}_i)\right]. \tag{6.39}$$

For $C = 2$ the computation of probabilities as well as the cross entropy turns into Eqs. 6.33 and 6.34.

We will see later that logistic regression can be carried out by a neural network as well. Mostly for reasons of simplicity, an overparametrized version of the model is used there. Since odds can be defined using any denominator, it makes no difference whether the odds of an event against a pivotal event is noted as $x$:1 or scaled by an arbitrary factor $s$ as $sx$:$s$. If the scale factor is chosen to be $s = \exp\left(\widehat{\mathbf{w}}_0^{\mathrm{T}}\widehat{\mathbf{x}}\right)$ using an additional set of parameters $\widehat{\mathbf{w}}_0$, Eq. 6.37 changes to

$$P_{\widehat{\mathbf{w}}_0}(y=0|\mathbf{x}_i) = \frac{\exp\left(\widehat{\mathbf{w}}_0^{\mathrm{T}}\widehat{\mathbf{x}}\right)}{\sum_{i=0}^{C-1}\exp\left(\widehat{\mathbf{w}}_i^{\mathrm{T}}\widehat{\mathbf{x}}\right)} \tag{6.40}$$

and probabilities for all classes $c = 0,\ldots,C-1$ are expressed by

$$P_{\widehat{\mathbf{w}}_c}(y=c|\mathbf{x}_i) = \frac{\exp\left(\widehat{\mathbf{w}}_c^{\mathrm{T}}\widehat{\mathbf{x}}\right)}{\sum_{i=0}^{C-1}\exp\left(\widehat{\mathbf{w}}_i^{\mathrm{T}}\widehat{\mathbf{x}}\right)}. \tag{6.41}$$

This expression called *softmax function*. Overparameterization comes at a cost, however, as the unnecessary extra variables produce infinitely many solutions for the optimization problem. Besides extra computation it causes problems for some numerical optimizers.

### 6.3.3 Kernel Logistic Regression

Linear logistic regression has the advantage over the two heuristic approaches discussed before that it guarantees to find an optimal decision boundary. Hence, distances from the decision boundary for a binary problem can rightfully be transformed to conditional class membership probabilities. The extension to linear discriminant functions for a multinomial logistic regression extends this to multiclass problems without having to resort to an indirect treatment such as the one-vs.-one and the one-vs.-all approach for heuristic decision boundaries.

However, the relation between sample distances and a posteriori probability is only true for a log-linear model. Although examples exist that can be represented by such model, it is still a serious restriction as we already noted in Sect. 6.1.4.

Just as support vector machines, logistic regression can also make use of kernel functions (see Zhu & Hastie, 2002). Observing Eqs. 6.40 and 6.41, we see that—just as for support vector machines—the sample vectors $\widehat{\mathbf{x}}$ appear solely as part of a scalar product. Similar to SVMs the weights $\widehat{\mathbf{w}}$ may be expressed as a linear combination of $\widehat{\mathbf{x}}$ since the sample vectors span the feature space. Hence, we have

$$\widehat{\mathbf{w}}_c^{\mathrm{T}}\widehat{\mathbf{x}}_i = \left(\sum_{j=1}^{M}\alpha_j\widehat{\mathbf{x}}_j^{\mathrm{T}}\right) = \sum_{j=1}^{M}\alpha_j\widehat{\mathbf{x}}_j^{\mathrm{T}}\widehat{\mathbf{x}}_i. \tag{6.42}$$

If we define a transformation $\Phi$ into higher-dimensional space and have a kernel function $k$ with $k\left(\widehat{\mathbf{x}}_i,\widehat{\mathbf{x}}_j\right) = \Phi(\widehat{\mathbf{x}}_i)^{\mathrm{T}}\Phi\left(\widehat{\mathbf{x}}_j\right)$, this operation becomes

$$\widehat{\mathbf{w}}_c^{\mathrm{T}}\widehat{\mathbf{x}}_i = \left(\sum_{j=1}^{M}\alpha_j\widehat{\mathbf{x}}_j^{\mathrm{T}}\right) = \sum_{j=1}^{M}\alpha_j k\left(\widehat{\mathbf{x}}_i, \widehat{\mathbf{x}}_j\right) \tag{6.43}$$

and the parameters to be estimated are now the $\alpha_j$. Logistic kernel regression may be carried out using the same kernel functions as for SVM. It is again the task of the developer to select kernels that transform features into a linearly separable space.

**Classes and Functions in Python**
Logistic regression is a class of `sklearn.linear_model`. Object construction, model fitting, and inference using the fitted model are functions that may be sequentially called. An example for training data `X_train` with labels `y_train` that predicts labels for samples in `X_test` is

```
...
from sklearn.linear_model import LogisticRegression
...
logr_clf = LogisticRegression(random_state=0).fit(X_train,
                                                  y_train)
y_pred  = logr_clf.predict(X_test)
...
```

The default setting automatically detects whether a binomial or multinomial problem is to be solved. For multiclass problems, the overparametrized version with softmax function as decision criterion is the default. The default can be overridden using the keyword `multi_class` in the constructor call.

## 6.4 Ensemble Models

Image classification discussed so far consists of a sequential processing pipeline with several linear and/or non-linear transformations of the image information. Ideally, although not being guaranteed, the application of the sigmoid function (for a binary classification problem) in the final step works on features with log-linear odds and produces the a posteriori class probability given the input image (see Fig. 6.13).

Discriminative classifiers based on complex models increase the chance that the non-linearly transformed input data has log-linear odds. However, classification based on such a complex model increases computational costs for model training and inference. It also may lead to overfitting of the model to the training data.

Bagging and boosting are two alternatives to work with simple classifiers while still being able to classify samples with highly non-linear decision boundaries and potentially multimodal likelihood functions. Instead of using a single, powerful classifier the concepts implement two different ways to combine results from several simple classifiers (see Brownlee, 2021).

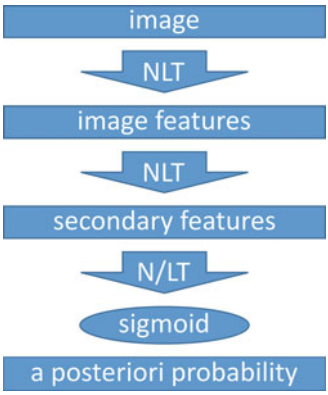



**Fig. 6.13** Binary classification as a sequential processing pipeline first generates image features and then secondary features via several non-linear transformations (NLT) which are further processed by linear or non-linear transformations before being submitted to the sigmoid function

### *6.4.1 Bagging*

*Bagging* stands for bootstrap aggregation. It implements ensemble learning from independent classifiers and it may outperform a single, more complex classifier. A comparison between the performance of a single classifier and an ensemble of classifiers is found in Liang et al. (2011).

Each of $K$ classifiers $c_k$ is trained separately to classify a subset of the data. Subsets are randomly selected from the training data and may overlap, i.e., the intersection of any two subsets is not empty. It can be assumed that at least some of the subsets have simpler distributions than the complete training data set. Hence, a simpler classifier should produce satisfactory results for a subset. Since subsets are still unbiased samplings from the training data, each classifier will learn something different about the overall distribution of samples in feature space. It still requires a classifier that is powerful enough to express characteristics of class-specific distributions of samples in feature space.

All $K$ classifiers are often of the same kind (e.g., linear support vector machines). They vary by their different parameterization stemming from training on different subsets of the training data.

During inference, all classifiers are applied to an unseen sample (see Fig. 6.14). Each classifier delivers a vote for a class label. The class with most votes wins and the sample is assigned to this class. Voting can be weighted. In this case, each of the classifiers is applied to the complete training data set and its performance is used as a weight when voting for a class.

Non-linear decision boundaries can be created by this approach even if each classifier is linear. The non-linearity is introduced by the voting procedure in decision making. The approach is particularly useful for sparse sample distributions in high-dimensional feature spaces. The simple linear classifiers then reduce the danger of overfitting to the training data, while potential non-linearities of the sample distribution (e.g., by multimodal likelihood functions) are addressed by the voting scheme.

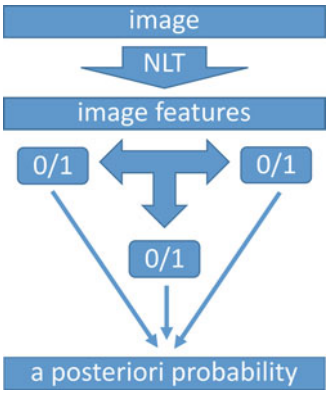



**Fig. 6.14** Different classifiers are used independently of each other for bagging. Each classifier reaches its own decision. Decisions are then combined to an estimate of the a posteriori probability for class membership

**Classes and Functions in Python**
Bagging is a class in `sklearn.ensemble`. It may be used to classify samples `X_test` based on labeled training data in `(X_train,y_train)` as follows:

```
from sklearn.ensemble import BaggingClassifier # the bagging
                                               # classifier
from sklearn.svm import SVC  # the base classifier

# constructor (earlier versions use keyword 'base_estimator'
# instead of 'estimator=')
bagg_clf = BaggingClassifier(estimator=SVC(), n_estimators=10,
                             random_state=0)

bagg_clf.fit(X_train, y_train) # fit the model to the data
y_pred = clf.predict(X_test)   # make prediction
```

### 6.4.2 *Boosting*

A *boosting* scheme uses a set of classifiers as well. It differs from bagging in that the classifiers are always applied to the complete training data set. The classifiers are trained differently because they are applied sequentially with each new classifier concentrating on the deficiencies of earlier trained classifiers. During inference, classification results from all classifiers are combined. The influence on the final result of a single classifier $c_k$ defines its performance quality.

A well-known example for a boosting algorithm is *AdaBoost* (adaptive boosting). It spawned a wealth of other boosting schemes (see Ferreira & Figueiredo, 2012 for a review) and we will use it here for a binary classification problem to demonstrate how this sequential learning takes place. AdaBoost is trained on a training data set $X = \{\mathbf{x}_1,\ldots,\mathbf{x}_M\}$ with known class membership $y_i = \{-1, 1\}$ for each sample $\mathbf{x}_i$. It

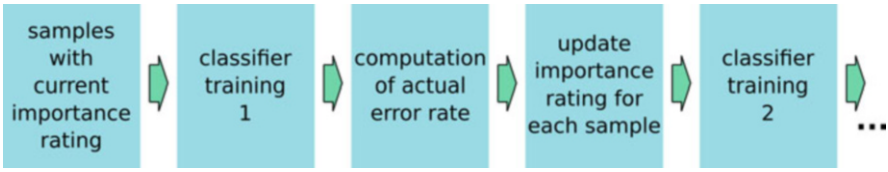



**Fig. 6.15** Boosting is a process where repeatedly a classifier is trained on the complete training data set. After each classification, sample importance is weighted so that samples have higher importance when they were not labeled correctly by the previous classifiers

trains a sequence of base classifiers $c_1,\ldots,c_K$. During inference, an unseen sample is classified as a weighted result from all base classifiers. The weights $\alpha_k$ represent the performance of classifier $c_k$.

Although training happens always on the complete training data set, samples influence classifier training differently for each new classifier. An importance value $D_k(i)$ is assigned to each training sample $i$ that changes during training. High-importance samples have a stronger influence on the computation of the optimization criterion of a classifier than low-importance samples. A sample gets higher importance if previous classifiers performed poorly on it. Hence, the next classifier will concentrate on these samples (Fig. 6.15).

Initially, equal importance $D_1(i) = \frac{1}{M}$ is assigned to all samples $i$. Then, classifiers are iteratively trained. If the current classifier is $c_k$ with parameter vector $\mathbf{w}_k$, it is trained by minimizing the importance-weighted error rate $\varepsilon$:

$$\mathbf{w}_k^{\text{opt}} = \underset{W_k}{\operatorname{argmin}} \varepsilon(c_k; \mathbf{w}_k) \tag{6.44}$$

with

$$\varepsilon(c_k; \mathbf{w}_k) = \sum_{i=1}^{M} D_k(i) I(y_i \neq c_k(\mathbf{x}_i; \mathbf{w}_k)). \tag{6.45}$$

The error rate is now used to compute a weight $\alpha_k$ for the trained classifier $c_k$

$$\alpha_k = \frac{1}{2} \ln \frac{1 - \varepsilon\left(c_k; \mathbf{w}_k^{\text{opt}}\right)}{\varepsilon\left(c_k; \boldsymbol{w}_k^{\text{opt}}\right)}. \tag{6.46}$$

The weight increases from 0 to infinity for error rates between 0.5 and 0 (higher error rates than 0.5 do not exist as for an error of, e.g., 0.7 the class assignment can simply be reversed yielding an error rate of 0.3).

Finally, new importance values are computed for each sample before training the next classifier $c_{k+1}$:

$$D_{k+1}(i) = \frac{D_k(i)\exp\left(-\alpha_k y_i c_k\left(\mathbf{x}_i; \mathbf{w}_k^{\text{opt}}\right)\right)}{Z}. \tag{6.47}$$

The exponent for a sample $i$ increases with the weight $\alpha_k$ of the previous classifier if the sign for the predicted class $c_k\left(\boldsymbol{x}_i; \mathbf{w}_k^{\text{opt}}\right)$ differs from the ground truth $y_i$. Hence, wrongly labeled samples receive a higher importance than correctly labeled samples. The value of $Z$ is a normalization constant that sums over importance values of all samples.

If all classifiers are trained, the label of an unseen sample with features $\mathbf{x}_\text{u}$ is inferred from a weighted combination of all classifier results for this sample

$$c(\mathbf{x}_\text{u}) = \text{sign}\left(\sum_{k=1}^{K} \alpha_j c_k\left(\mathbf{x}_\text{u}; \mathbf{w}_k^{\text{opt}}\right)\right). \tag{6.48}$$

AdaBoost develops a sequence of classifiers where each new classifier concentrates on the so far unsatisfactorily solved parts of the classification problem. Increasing the number of classifiers increases the expressiveness of the combined classifier and keeping the base classifiers simple avoids overfitting to the training data.

Choosing the number of base classifiers is a design decision. It may be trained as well. It is then a hyperparameter that requires a separate validation data set besides training and test data.

**Classes and Functions in Python**

AdaBoost is a class in `sklearn.ensemble`. It may be used to classify samples `X_test` based on labeled training data in `(X_train,y_train)` as follows:

```
from sklearn.ensemble import AdaBoostClassifier # AdaBoost class
from sklearn.svm import SVC           # base classifier

adab_clf = AdaBoostClassifier(estimator=SVC(), # constructor
               n_estimators=10,    # N.B.: earlier versions
               random_state=0)     # use 'base_estimator='
                                   # instead of 'estimator'
adab_clf.fit(X_train, y_train)     # fit the model to the data
y_pred = clf.predict(X_test)       # make prediction
```

## 6.5 Exercises

### 6.5.1 Programming Project P6.1: Support Vector Machines

Import the CIFAR10 data set from the OpenML repository using `fetch_openml()` from `sklearn.data`, reshape the data to a sequence of images of size $32 \times 32 \times 3$ ($32 \times 32$ pixels in 3 color channels). Take only the first 5000 samples from the data set with a 4000:1000 split for training and test data (the subset is still pretty balanced and results in faster fitting of the SVM).

Create four sets of features from it that you will compare:

1. Raw data as feature vector with length 3072
2. Raw HOG features with $4 \times 4$ pixel window, $2 \times 2$ blocks, and 9 directions resulting in a feature vector with length 1764
3. PCA on the original input data and selection of the top-200 features
4. Computation of HOG features followed by PCA, again selecting the top-200 features

Create a baseline result by classifying the data using the kNN classifier and compute the accuracy.

Then, use SVM for classification with different kernels and weight values for the soft margin on the raw data. What produces the best results? Compare the results with that of the kNN classifier. What does it say about the two classifiers? Then, compare results on raw HOG features. What does it say about the class-specific characteristics? Then, use PCA on the raw data and the HOG features with SVM. Look up the explained variance by the PCA for the two features (by summing up the variable `explained_variance_ratio_` of the PCA object) and discuss the results in view of this.

### 6.5.2 Programming Project P6.2: Label the Imagenette DATA I

Imagenette (see Sect. 4.3 on reference data sets) is a small subset of the data used for the ImageNet challenge. It has much fewer data than MNIST or CIFAR10. Images generally have higher resolutions. Resolutions and aspect ratios of different images are different for different images. The data can be found in a GitHub repository on the network (search terms "imagenette," "github," "download" or see Sect. 4.3.1). Training and validation data is found in two different directories. Within each directory, data is organized by label in ten subdirectories.

Copy the Imagenette data and write your own function to read it into separate arrays for training and testing. Resize all images to a common size of $224 \times 224$ pixels (it will change the aspect ratio of most images but you have to live with this in order to create images of equal size and aspect ratio). Create a corresponding label

array that assigns the label "0" to samples from the first subdirectory in training and testing, the label "1" to samples from the second subdirectory, and so on.

Then, develop your own classifier. Decide which features should be extracted, whether and how they should be reduced, and what classifier should be applied. Report the accuracy of your classifier (anything above 0.5 is quite good).

### *6.5.3 Exercise Questions*

- Why must the best decision boundary be selected from several possible decision boundaries?
- Compare the computational cost for model fitting and prediction between a kNN classifier and a SVM. In terms of cost, what are the respective advantages and disadvantages?
- What does the sigmoid function have to do with the a posteriori probability of class membership?
- Why and under what circumstances is the decision boundary selection criterion used in SVMs good for classification?
- What disadvantages arise if you have chosen a soft margin for SVM that is too large? How can these disadvantages be detected during training or test?
- What are the possibilities for extending a classification by SVM to any number of classes? Which approach would you choose if a large number of classes (e.g., the 100 classes of the CIFAR100 competition) are to be distinguished? Justify your answer.
- What kind of distributions cannot be represented by a linear SVM? Why is this so?
- What is the advantage of using kernel functions and why can you use them? What disadvantage does this have over a linear SVM?
- Decision boundaries found by logistic regression are a discriminative model. For the generation of new samples from the distribution, they may be used similarly than a generative model. Explain, how this could be done.
- Why does bagging with multiple classifiers have an advantage over a very expressive classifier?
- Does bagging with linear classifiers yield a non-linear classifier? Give reasons for your answer.
- Why does boosting require each classifier to be generated sequentially?
- Does boosting with linear classifiers yield a non-linear classifier? Justify your answer.

# References

Brownlee, J. (2021). *Ensemble learning algorithms with Python: Make better predictions with bagging, boosting, and stacking*. Machine Learning Mastery.

Christensen, R. (2006). *Log-linear models and logistic regression*. Springer Science & Business Media.

Ferreira, A. J., & Figueiredo, M. A. (2012). Boosting algorithms: A review of methods, theory, and applications. In *Ensemble machine learning: Methods and applications* (pp. 35–85). Springer.

Fletcher, T. (2009). Support vector machines explained. Tutorial paper (pp. 1–19).

Ghojogh, B., & Crowley, M. (2019). Linear and quadratic discriminant analysis: Tutorial. arXiv preprint arXiv:1906.02590.

Kecman, V. (2005). Support vector machines—An introduction. In *Support vector machines: Theory and applications* (pp. 1–47). Springer.

Liang, G., Zhu, X., & Zhang, C. (2011). An empirical study of bagging predictors for different learning algorithms. In *Proceedings of the AAAI Conference on Artificial Intelligence* (Vol. 25, No. 1, pp. 1802–1803).

Mammone, A., Turchi, M., & Cristianini, N. (2009). Support vector machines. *Wiley Interdisciplinary Reviews: Computational Statistics, 1*(3), 283–289.

Schölkopf, B., & Smola, A. J. (2018). *Learning with Kernels: Support vector machines, regularization, optimization, and beyond*. MIT Press.

Zhu, J., & Hastie, T. (2002). Support vector machines, kernel logistic regression and boosting. In *Third International Workshop on Multiple Classifier Systems, MCS 2002 Cagliari, Italy, June 24–26, 2002, Proceedings 3* (pp. 16–26). Springer.

# Chapter 7
# 7 Multi-Layer Perceptron for Image Classification

**Abstract** Artificial neural networks are trainable models that model arbitrarily complex decision boundaries for image classification. A network consists of nodes and edges represented as graph. Networks for image classification are cycle-free and information flows from an input layer through the nodes along the edges to an output layer. The input layer receives features extracted from an image producing a prediction at the output layer. A simple network of this kind is the single-layer perceptron which receives features extracted from an image and carries out a binary classification. We will show that it solves the classification problem by linear logistic regression.

Adding layers to the network extends the trainable model to non-linear decision boundaries. We will explain why a single extra layer is sufficient to train an arbitrary mapping from input to output. It makes a multi-layer perceptron a universal solver that can find an optimal non-linear mapping for multinomial logistic regression. We will explain why adding further layers increases the efficiency of such networks. Parameter optimization in the network is done by the backpropagation algorithm that carries out a gradient descent on the network parameters. Finally, the Adam optimizer will be presented as a popular gradient descent method.

The idea to simulate the neural system of humans or animals to arrive at an intelligent system is older than computers themselves. It is attractive as the neural system consists of few different components—various kinds of neurons—with functionality that is understood well, at least at some global level[1]: Neurons are interconnected and transmit information to other neurons if some action level is exceeded. The neuron itself consists of cell body (the soma), an axon, and dendrites (see Fig. 7.1). The axon extends from the soma. It transmits signals through terminal buttons at the end of the axon to dendrites of other neurons. Transmission between nerve cells happens through synapses which are small gaps between dendrite and an axon terminal button of another neuron. A signal is generated by a neuron if the

[1] If you want to dig deeper, see, e.g., Bear et al. (2020) for an extensive introduction on the current view on the human neural system.

© The Author(s), under exclusive license to Springer Nature Singapore Pte Ltd. 2024

K. D. Toennies, *An Introduction to Image Classification*,
https://doi.org/10.1007/978-981-99-7882-3_7

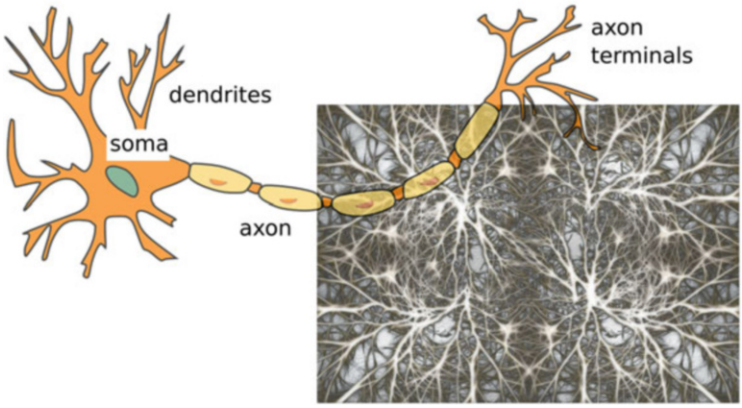



**Fig. 7.1** A single neuron is just a tiny component in a complex neural network consisting of billions of interconnected neurons. (Picture created from public domain pictures at pixabay.com)

electric potential in a neuron exceeds a threshold. Transmission may excite or inhibit the action potential in connected neurons. The extent of excitation or inhibition is learned, i.e., connections between neurons may be strengthened or weakened. The neural system then causes reactions to sensory input.

Creating an intelligent system seems to be just a question of scale if the function of a neuron network can be simulated. For simulation, the network topology is mapped to a graph. Nodes represent neurons that are connected by directed edges. Signal transmission between two neurons happens along these edges. Strengthening or weakening of a connection between nodes is represented by edge weights that modify the transmitted signal. Connections with negative weights are inhibitory and those with positive weights are excitatory. Neuron action sums the input from all connected neurons and transmits it to connected neurons, if this sum exceeds some threshold.

In practice, creating an intelligent system as artificial neural network is still difficult. To date no such system has been presented that has all or even most of the main attributes of a true artificial intelligence system (sometimes called *strong AI*):

- Being self-aware.
- Being able to reflect its own decisions.
- Being able to transfer learned knowledge from one domain to another.

Even the most advanced solutions train rather than learn knowledge. A trained system is not able to decide by itself how this knowledge should be represented, adapted, or applied. The inability to model such a learning system using the brain as a blueprint is caused by the sheer complexity of the neural system and the learning process in humans or animals:

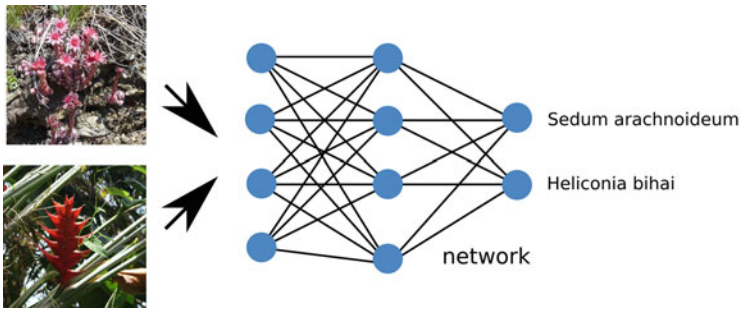



**Fig. 7.2** Artificial neural networks copy the basic concept of transmitting signals along dendrites from one cell to the next. The network structure is much simpler, however, and the tasks to be solved by the network are much more restricted

- The neural network is very large. The human brain contains about 100 billion neurons with $10^{15}$ connections. Most computers simply do not have enough memory to represent this massive amount of information.
- Network connections are highly complex. Although transmission between pairs of neurons is unidirectional, the network contains cycles resulting in numerous feedback loops. Hence, time of transmission is an additional component that adds to the overall complexity.
- The neural network comes pre-trained as some of its functionality is inherited. Little is known about how much of our behavior depends on inheritance.
- Knowledge representation in the brain seems to use various models that again are not well understood. Hence, it cannot be verified whether trained knowledge representation in an artificial network resembles that in humans or whether it exhibits a similar behavior when used to drive a decision.
- Learning is a very long process with multi-sensory input. It takes many months, for instance, for a baby to understand the three-dimensionality of the world. During this time, the baby constantly receives input from different senses. This massive training is impossible for applications with artificial networks.

There have been and still exist large research projects attempting to analyze and model the neural system in order to understand to what extent it may be used as a blueprint for an artificial intelligent system. However, artificial neural networks used in practice are extreme simplifications of the human neural system. They are far from a system whose behavior cannot be distinguished from human intelligence (this would indeed be strong AI). Hence, the term "neural" is sometimes omitted when talking about artificial networks.

As far as our image classification system is concerned, we will work with such simple networks (see Fig. 7.2). We will see that the downscaling still retains some aspects of universal learning and that it delivers good results even for difficult classification tasks. However, we will also see that the resulting network is not intelligent in the sense of strong AI. The kind of generalizations that our artificial neural network will be able to make are rather simplistic and explain little about

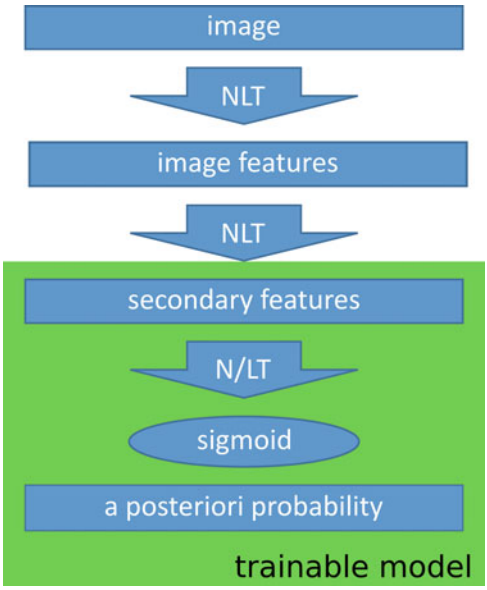



**Fig. 7.3** In a first step, we will replace the designed classifier by a universal trainable model

underlying class specifics that caused a certain appearance. It will not be possible, for instance, to infer from pictures of 3-d objects the concept of 3-d to 2-d projection. Understanding limitations like this is important. State-of-the-art artificial neural networks can solve difficult problems and they are a great tool for many computer vision tasks but one should not expect too much intelligence from them just because of this.

In our case, the sensory input are pictures. The network is trained to make a class decision based on this input. Our networks will not have cycles. Classification is a single feedforward step during which information from the image is transmitted to some output nodes that represent the classification result.

Remember, that traditional classification consists of a feature extraction step followed by an application of a classifier to the features. For networks, we will find the same separation. In this chapter, we start with networks representing classifiers (see Fig. 7.3). We will extend it to learn the appropriate features in the following chapters.

The motivation to have a universally trainable classifier is the large diversity of existing traditional classifiers.[2] Each of the many different concepts for traditional classifiers implements another assumption about the approximation of conditional class probabilities. Developing a classifier thus has two stages. First, the developer decides on the most appropriate classifier concept for a given task. Then, parameters of the chosen classifier are trained from samples. A classifier model represented by a

[2] Machine learning textbooks (e.g., Raschka et al. (2022) for a comprehensive book using Python as programming language) present a much larger diversity of models than the few that were presented in the first part of this book.

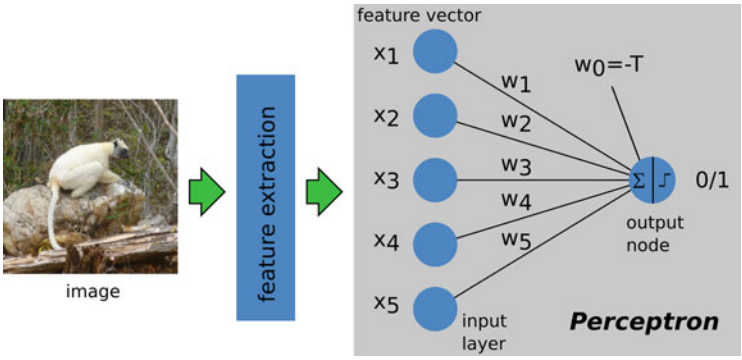



**Fig. 7.4** A perceptron takes a feature vector as input and generates output by thresholding a weighted sum of the input. Weight optimization for the original perceptron is done by heuristic rules similar to the ones presented for linear decision boundaries in Sect. 6.1. Feature extraction is not part of the perceptron

neural network, however, will combine both stages in a trainable model. It may be turned into different specialized systems based on training from different sets of labeled samples.

## 7.1 The Perceptron

A simple artificial neural network to solve a classification problem is the *perceptron*. It has been presented by Rosenblatt (1957) and used an even older idea about the simulation of a neuron (the McCulloch-Pitts cell). A perceptron consists of a single neuron that receives input from features in a feature vector. The system is represented by a layered, directed graph (see Fig. 7.4). In the input layer, every feature is represented by a node that feeds the feature value to the next layer. All input nodes are connected by directed edges to a single node in the output layer.[3] Feature values are transmitted through the graph from the input layer to the output layer to create a decision for the classification problem.

[3] The layer count of such a hierarchical network depends on whether the input nodes are counted or not. Input nodes are often not counted since they do not modify the input. Under this definition, a perceptron consists of a single layer.

### 7.1.1 Feedforward Step

The purpose of the node in the output layer is to gather information from the input nodes and to produce a decision. It is done by two nested functions. The first function produces a weighted sum of the feature vector $\mathbf{x} = (x_1 \ldots x_N)^{\mathrm{T}}$:

$$f_{\mathrm{W}}(\mathbf{x}) = \sum_{i=1}^{N} w_i x_i. \tag{7.1}$$

Weights $w_i$ are trained from samples with known label.

The second function is called *activation function* and produces the output value. In the original formulation of the perceptron, it is a threshold on $f_{\mathrm{W}}$:

$$g(f_{\mathrm{W}}(\mathbf{x})) = \begin{cases} 0 & \text{if } f_{\mathrm{W}}(\mathbf{x}) < T \\ 1 & \text{if } f_{\mathrm{W}}(\mathbf{x}) \geq T \end{cases}. \tag{7.2}$$

It is easily seen that this is a linear decision boundary with bias term $w_0 = -T$. Integrating $w_0$ into $\mathbf{w}$ to $\widehat{\mathbf{w}} = (w_0\ w_1 \ldots w_N)^T$ and extending $\mathbf{x}$ accordingly to $\widehat{\mathbf{x}} = (1\ \ x_1 \ldots\ \ x_N)^T$ we have the familiar equation

$$f_{\widehat{\mathbf{w}}}(\widehat{\mathbf{x}}) = \widehat{\mathbf{w}}^T \widehat{\mathbf{x}} \tag{7.3}$$

with the neuron firing if $g\left(f_{\widehat{\mathbf{w}}}(\widehat{\mathbf{x}})\right) \geq 0$. In the original publication, weights $\widehat{\mathbf{w}}$ were determined using a heuristic similar to the one presented in Sect. 6.1. Pairs $(\mathbf{x}_i, y_i)$ from a set of training samples $X = \{\mathbf{x}_1, \ldots, \mathbf{x}_M\}$ are picked at random and the decision boundary is corrected if the prediction differed from the ground truth.

### 7.1.2 Logistic Regression by a Perceptron

Even though the result from a perceptron is not different from training a linear decision boundary, its structure allows for a number of interesting generalizations by changing basic hyperparameters of the network.

If the activation function is the sigmoid function, the classification model becomes a linear logistic model. For training the model, gradient descent on the binary or categorical cross entropy can be used (see Fig. 7.5).

The output of the regression network using an activation function $g_{blg}$ for a binary logistic model is now

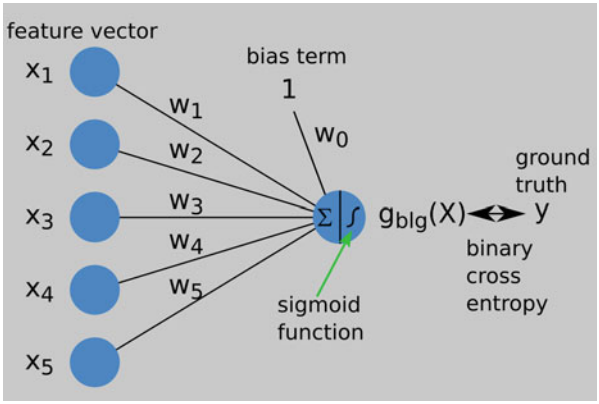



**Fig. 7.5** If the threshold in the original perceptron is replaced by the sigmoid function and binary cross entropy is the loss function to be optimized, the perceptron carries out a logistic regression for the input features

$$g_{blg}(\widehat{\mathbf{x}}, \widehat{\mathbf{w}}) = \frac{1}{1 + \exp\left(-\left(\widehat{\mathbf{w}}^T \widehat{\mathbf{x}}\right)\right)} \tag{7.4}$$

and the loss function to be optimized is the binary cross entropy

$$l\left(g_{blg}(\widehat{\mathbf{x}}, \widehat{\mathbf{w}})\right) = -y \ln g_{blg}(\widehat{\mathbf{x}}, \widehat{\mathbf{w}}) - (1-y) \ln\left(1 - g_{blg}(\widehat{\mathbf{x}}, \widehat{\mathbf{w}})\right). \tag{7.5}$$

For carrying out a single step of gradient descent, partial derivatives of $l$ with respect to the weights in $\widehat{\mathbf{w}}$ have to be computed. Using the chain rule, we have

$$\frac{\partial}{\partial w_i} l\left(g_{blg}(\widehat{\mathbf{x}}, \widehat{\mathbf{w}})\right) = l'\left(g_{blg}(\widehat{\mathbf{x}}, \widehat{\mathbf{w}})\right) \frac{\partial}{\partial w_i} g_{blg}(\widehat{\mathbf{x}}, \widehat{\mathbf{w}}). \tag{7.6}$$

The derivative of the cross-entropy function $f(x) = -y \ln x - (1-y) \ln (1-x)$

$$f'(x) = -\frac{y}{x} + \frac{1-y}{1-x} \tag{7.7}$$

and thus

$$l'\left(g_{blg}(\widehat{\mathbf{x}}, \widehat{\mathbf{w}})\right) = -\frac{y}{g_{blg}(\widehat{\mathbf{x}}, \widehat{\mathbf{w}})} + \frac{1-y}{1 - g_{blg}(\widehat{\mathbf{x}}, \widehat{\mathbf{w}})}. \tag{7.8}$$

The partial derivative of the activation function with respect to weight $w_i$ is

$$\frac{\partial}{\partial w_i} g_{blg}(\widehat{\mathbf{x}}, \widehat{\mathbf{w}}) = \frac{\partial}{\partial w_i} \frac{1}{1 + \exp\left(-\left(\widehat{\mathbf{w}}^T \widehat{\mathbf{x}}\right)\right)} = \frac{\partial}{\partial w_i} \frac{1}{1 + \exp\left(-f_{\widehat{\mathbf{x}}}(\widehat{\mathbf{w}})\right)} \tag{7.9}$$

with

$$f_{\widehat{\mathbf{x}}}(\widehat{\mathbf{w}}) = \widehat{\mathbf{w}}^T \widehat{\mathbf{x}} = \sum_{i=0}^{N} w_i x_i. \tag{7.10}$$

Applying the chain rule gives

$$\frac{\partial}{\partial w_i} \frac{1}{1 + \exp\left(-f_{\widehat{\mathbf{x}}}(\widehat{\mathbf{w}})\right)} = \frac{\exp\left(-f_{\widehat{\mathbf{x}}}(\widehat{\mathbf{w}})\right)}{\left(1 + \exp\left(-f_{\widehat{\mathbf{x}}}(\widehat{\mathbf{w}})\right)\right)^2} \frac{\partial}{\partial w_i} f_{\widehat{\mathbf{x}}}(\widehat{\mathbf{w}}), \tag{7.11}$$

where the first term on the right-hand side is just the derivative of the sigmoid function which may be written as

$$g'_{blg}(\widehat{\mathbf{x}}, \widehat{\mathbf{w}}) = g_{blg}(\widehat{\mathbf{x}}, \widehat{\mathbf{w}})\left(1 - g_{blg}(\widehat{\mathbf{x}}, \widehat{\mathbf{w}})\right). \tag{7.12}$$

Derivation of the sum in $f_{\widehat{\mathbf{x}}}(\widehat{\mathbf{w}})$ with respect to one of its summands $w_i$

$$\frac{\partial}{\partial w_i} f_{\widehat{\mathbf{x}}}(\widehat{\mathbf{w}}) = x_i. \tag{7.13}$$

Finally, we arrive at

$$\frac{\partial}{\partial w_i} l\left(g_{blg}(\widehat{\mathbf{x}}, \widehat{\mathbf{w}})\right) = l'\left(g_{blg}(\widehat{\mathbf{x}}, \widehat{\mathbf{w}})\right) g'_{blg}(\widehat{\mathbf{x}}, \widehat{\mathbf{w}}) x_i. \tag{7.14}$$

Hence, first the derivative *l'* of the loss function is computed and weighted with the derivative of the activation function. The result is then weighted separately with each of the input values to produce the partial derivative.

Gradient descent then changes the weights by

$$w_i^{\text{new}} = w_i - \lambda \frac{\partial}{\partial w_i} l\left(g_{blg}(\widehat{\mathbf{x}}, \widehat{\mathbf{w}})\right) \tag{7.15}$$

with $\lambda$ being a learning rate.

Gradients for multinomial regression may be computed as well. Instead of a single output neuron the layer will have as many output neurons as there are classes to be distinguished (see Fig. 7.6). Every input node is connected to every output node. An edge from an input node $i$ to output node $j$ carries a trainable weight $w_{ij}$. The feedforward step now computes for every sample an output vector by

$$G_{mlg}\left(\widehat{\boldsymbol{x}}_i, \widehat{\mathbf{W}}\right) = \widehat{\mathbf{W}}^T \widehat{\mathbf{x}}_i \tag{7.16}$$

where the columns of matrix $\widehat{\mathbf{W}}$ contain weights $w_{ij}$ for class $j$.

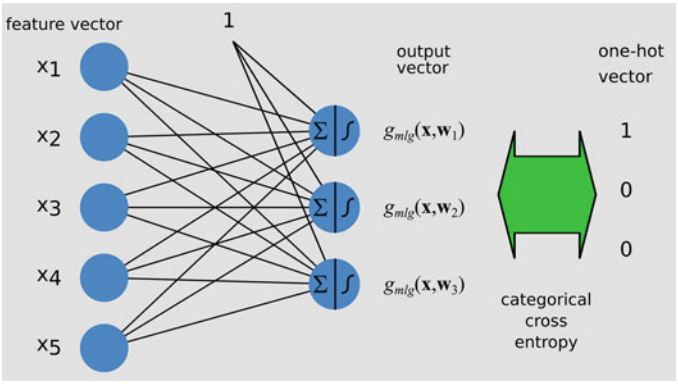



**Fig. 7.6** For multinomial classification, each class is represented by a fully connected perceptron. Output is a vector with as many elements as there are classes. It is compared with a ground truth vector that is zero everywhere except for the class that is predicted as ground truth (the so-called one-hot-vector; in this example, it is class 1)

The overparameterized version for multinomial regression preferred as it leads to a simple network topology (no pivotal class required). The activation function is the softmax function, introduced in Sect. 6.3.2:

$$g_{mlg}(\widehat{\mathbf{x}}, \widehat{\mathbf{w}}_c) = \frac{\exp\left(\widehat{\mathbf{w}}_c^T \widehat{\mathbf{x}}\right)}{\sum_{i=0}^{C-1} \exp\left(\widehat{\mathbf{w}}_i^T \widehat{\mathbf{x}}\right)}, \tag{7.17}$$

where $\widehat{\mathbf{w}}_c$ is the column $c$ of $\widehat{\mathbf{W}}$ that contains weights $(w_{0,c}, w_{1,c}, \ldots, w_{N,c})$ for class $c$. The output for all classes $c$ forms a vector that is now compared with the ground truth. The latter is represented by a *one-hot-vector* with as many entries as there are classes. All entries but the one for the correct class are zero. The entry for the correct class is set to one. The one-hot-vector and the vector of activations of the output layer are compared using categorical cross entropy (also called multinomial cross entropy, see Sect. 6.3.2) as loss function.

The complete training consists of repeated application of the following steps:

- Feed training samples to the network (the so-called *feedforward step*).
- Compute the decision of the classifier.
- Backpropagate from the output by computing the derivative of the loss function and that of the activation function.
- Distribute the derivatives among the different incoming edges.
- Correct the weights using the computed gradient.

This kind of gradient computation by backpropagating an error term (the loss) through the network will work for all of the networks that we will discuss in this book.

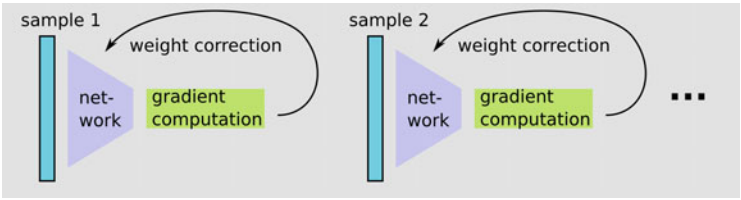



**Fig. 7.7** Stochastic gradient descent computes the loss after feeding a single sample through the network and corrects the weights accordingly. A run through all samples is called an epoch. Several such runs are necessary for optimization to converge

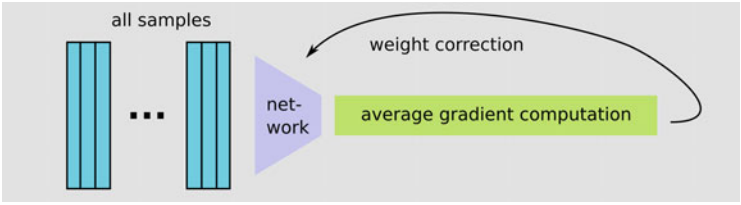



**Fig. 7.8** In batch gradient descent, the complete set of training samples is submitted to the network. An average gradient is computed and used for weight correction. An epoch consists of a single correction step

### *7.1.3 Stochastic Gradient Descent, Batches, and Minibatches*

Gradient descent often behaves unsatisfactorily for more complex networks so that there are many variants to correct the weights.[4]

A reason for the erratic behavior of gradient descent even for a simple network is the way loss is computed. We computed it for every single sample and used the gradient to correct weights. The true goal, however, is to minimize the overall loss from all samples of the training data. Using gradients from the loss of single samples is just a probabilistic estimate. It is called *stochastic gradient descent* (SGD, see Fig. 7.7). SGD requires to go several times (called *epochs*) over the training samples. A lot of random fluctuations can be expected during an epoch since each correction is based on a rather rough approximation of the true gradient.

For smoother descent, *batch gradient descent* (BGD) may be carried out (see Fig. 7.8). A comparison of loss curves for SGD and BGD is depicted in Fig. 7.9.

For BGD, the complete training data set $\widehat{\mathbf{X}} = \{\widehat{\mathbf{x}}_1, \ldots, \widehat{\mathbf{x}}_M\}$ is fed to the network and an output vector $\tilde{\mathbf{y}} = (y_1 \ldots y_M)^T$ is computed by (for binomial classification)

[4] A reader, who is unfamiliar with gradient descent methods in general, might want to read some introductory texts. A good resource is Ruder (2016).

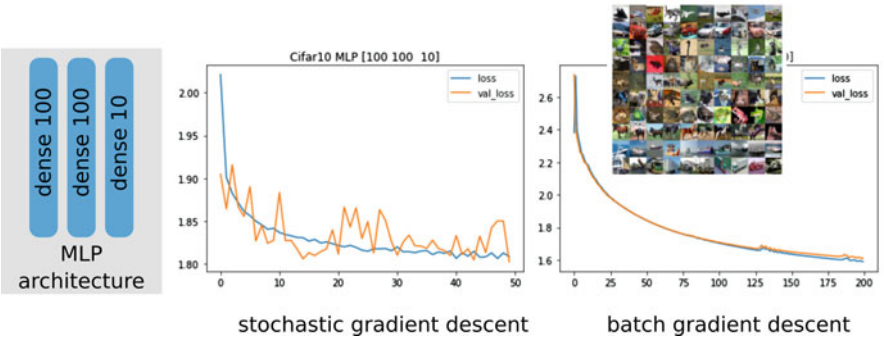



**Fig. 7.9** Comparison of loss function for stochastic (SGD) and batch gradient descent (BGD) for a simple MLP on the CIFAR-10 data. The loss function decreases more slowly but also more smoothly for BGD. The reason for the former is that just a single correction is made per epoch for BGD, whereas for this data set 50,000 corrections per epoch are made for SGD. The architecture of the network is depicted on the left. It consists of two fully connected hidden layers with 100 nodes each and an output layer with as many nodes as there are classes

$$\tilde{\mathbf{y}} = g_{blg}\left(\hat{\mathbf{X}}, \hat{\mathbf{w}}\right) = \frac{1}{1 + \exp\left(-\left(\hat{\mathbf{w}}^{\mathrm{T}}\hat{\mathbf{X}}\right)\right)} \tag{7.18}$$

or (for multinomial classification) with $\tilde{\mathbf{y}}_c$ being a column vector of a $M$x$C$ matrix $\tilde{\mathbf{Y}}$ of all $M$ samples and $C$ classes

$$\tilde{\mathbf{y}}_c = g_{mlg}\left(\hat{\mathbf{X}}, \hat{\mathbf{w}}_c\right) = \frac{\exp\left(\hat{\mathbf{w}}_c^{\mathrm{T}}\hat{\mathbf{X}}\right)}{\sum_{i=0}^{C-1}\exp\left(\hat{\mathbf{w}}_i^{\mathrm{T}}\hat{\mathbf{X}}\right)}. \tag{7.19}$$

The matrix $\hat{\mathbf{X}}$ consists of $M$ columns, one for each data set in the training data. Each column contains the features of a sample. An average cross entropy is computed from the vector $\tilde{\mathbf{y}}$ or the matrix $\tilde{\mathbf{Y}}$, respectively, as loss and backpropagated to compute average gradient.

Before we continue the discussion, one aspect of the two equations above should be looked at further. In both cases, information has been stored by one- and two-dimensional arrays. The flow of information through the layer interprets these arrays as vectors and matrices and realizes the propagation by vector-matrix operations on the input. Hence, a graph with nodes and edges is just a visualization of network topology and information transfer. The developer of a system thinks in weight matrices and input matrices instead. Network training and inference is done by defining proper arithmetic operations on these vectors and matrices. This will be true for all the networks that we will discuss in this book. For networks with more than one layer discussed in the next section, it will be realized by concatenation of matrix operations.

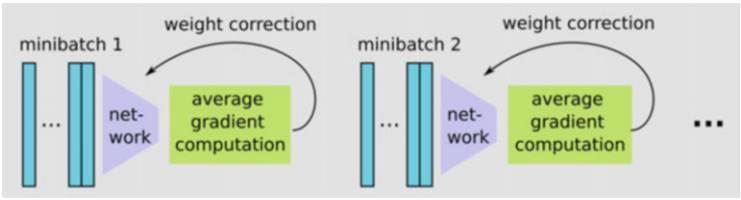



**Fig. 7.10** Minibatches are a compromise between stochastic gradient descent with single samples and batch gradient descent. A minibatch is a subset of the training samples. The training data is separated into disjoint subsets. An epoch consists of correction steps with all subsets

Networks with node layers and input layers that have more than one dimension and that we will introduce in the next chapter require a representation of weights and input information by arrays with more than two dimensions. This is mapped to tensors. Tensors are a generalization of matrices to collections of a number of arbitrary dimensions with associated arithmetic operations. All operations in arbitrary, cycle-free neural networks can be represented by operations between tensors.[5] For network design, the engineer defines tensors for information representation and the proper sequence of arithmetic operations on it. Popular packages for network design and operation such as TensorFlow, PyTorch, or Theano already come with a predefined tensor class including basic arithmetic operations so that development consists of constructing or extending objects of this class and defining operations that realize an individual network design.

Modern solutions to train such a network load the complete matrices or tensors on a graphics card that is specialized for carrying out fast matrix/tensor operations by the graphic processing unit (GPU). Securing sufficient memory on the graphics card may become a major issue especially for larger (and deeper) networks. In such case, the use of *minibatches* is a compromise between batch gradient and stochastic gradient descent (see Fig. 7.10).

A minibatch is a subset of the training data that is fed to the network for computing a mean loss from all samples in the minibatch. Gradient descent with minibatches is still stochastic as each minibatch is a random selection of samples from the training database. Random fluctuations should be less than for single sample SGD, however, as the estimate of the average loss becomes better more samples are included in the minibatch (see Fig. 7.11). The random fluctuations may even be beneficial as they introduce a kind of perturbation in the gradient descent that may allow the system to escape a local minimum.

Except for some toy examples, using minibatches is the rule as it is often impossible to load the complete batch to the GPU. Common practice for network training is to select the largest possible minibatch given a network and the memory restrictions of the GPU and train with these minibatches. Again, several epochs will

[5] Even recurrent networks that contain cycles are representable in this way if the flow of information is approximated by cycle-free graphs, see Boden (2002).

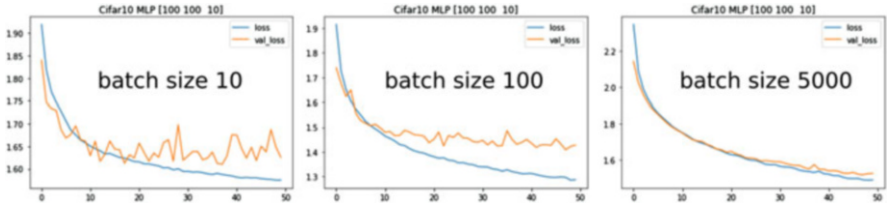



**Fig. 7.11** Gradient descent for the network and data depicted in Fig. 7.8 with different sizes for the minibatches. For smaller minibatches the behavior resembles that of SGD with single samples. For very large minibatches it resembles that of BGD

be necessary for training and some fluctuation caused from the randomness of sample selection will be observed. Bias from sample selection for each minibatch is avoided if samples are randomly shuffled for every new epoch.

**Classes and Functions in Python**

Equal to traditional models, there are three phases to generate, fit, and use a model. Python defines models as objects of a class. Fitting and using the model are functions of this class. Hence, the familiar sequence for classifier models will be found in the implementation here as well.

```
model = model_constructor(<params>) # in this case a neural
                                    # network
model.fit(X_train, y_train,<params>) # determine weights of the
                                    # network
Y_pred = model.predict(X_test)  # apply the trained network to
                                # predict labels of unknown
                                # data
```

The model construction, however, is a complex process since many different networks with different layers, layer properties, nodes per layer etc. can be constructed and many different functions to optimize the model can be employed. Although it is possible to put all this information in one, very long parameter list, the implementation of model construction is easier understood, if first an empty model is created, then the topology (how many layers, what kind of layers, how many nodes) is defined together with layer properties for each layer (activation function, etc.), before finally specifying the parameters for model training. This is the usual structure of building a model using Keras/TensorFlow.

Hence, the call to the constructor is replaced by the following sequence:

(continued)

– Create an empty network by calling a constructor with no input parameters.
– Add model layers to the network by calling a model function `add()` for every layer.
– Define training properties by calling the model function `compile()`.

In order to create a single-layer perceptron, we need the class `Sequential` from the `tensorflow.keras.models` submodule and the classes `InputLayer`, `Flatten`, and `Dense` from the submodule `tensorflow.keras.layers`. `Sequential` is a class for strictly sequential networks, i.e., nodes of one layer are connected only to nodes of the next layer. An object of the class `InputLayer` adds an input layer with the shape of the input data. An object of the class `Flatten` adds a layer that flattens multidimensional input (our images) to a 1-d feature vector needed by the perceptron. Objects of the class `Dense` are network layers where every node of one layer is connected to every node of the next layer.

A single-layer perceptron with 784 input nodes and 10 output nodes, with softmax activation and Adam optimizer with default parameters, that is trained to classify the MNIST data is.

```
import numpy as np
from tensorflow.keras.datasets import mnist      # MNIST through Keras
from tensorflow.keras.models import Sequential  # model class

# layer classes
from tensorflow.keras.layers import InputLayer, Dense, Flatten

# utils contains function to make a one-hot vector from labels
import tensorflow.keras.utils as utils

from sklearn.model_selection import train_test_split

# load dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# split training data into training and validation data
X_train, X_val, y_train, y_val = train_test_split(X_train,
                  y_train, test_size=0.2, random_state=42)

# normalize data and make sure it is of type float32
X_train = X_train.astype('float32')/255.0
X_val = X_val.astype('float32')/255.0
X_test = X_test.astype('float32')/255.0
```

(continued)

```
# one-hot encoding using Keras' numpy-related utilities
y_train = utils.to_categorical(y_train, 10)
y_val = utils.to_categorical(y_val, 10)
y_test = utils.to_categorical(y_test, 10)

# construct empty sequential network
model = Sequential(InputLayer(input_shape=(28, 28)))

# add layer to map the input to a 1-d feature vector
model.add(Flatten(),name='flatten')

# add layer 'd01' with 10 fully connected nodes to the input layer
model.add(Dense(10, activation='softmax',name='d01'))

# print a summary of the constructed model
model.summary()

# set model training parameters
model.compile(loss='categorical_crossentropy',
              metrics=['accuracy'], optimizer='adam')

# train the model for 100 epochs using minibatches of size 256
model.fit(X_train, y_train, batch_size=256, epochs=100,
          validation_data=(X_val, y_val))

# predict labels of the test data
y_pred = model.predict(X_test)

# print loss and accuracy on the test data
score = model.evaluate(X_test, y_test, verbose=0)
print('Test loss:', score[0],' accuracy:', score[1])
```

This example includes all the basic operations when training and testing a network. Compilation, fitting, prediction, and evaluation may be applied to any sequential network model that we will discuss in the following chapters. The function `model.predict()` returns for each sample a prediction for the one-hot-vector. The class label can be extracted using the NumPy function `argmax()`. Prediction and scoring on the test data are usually applied on a trained model where the validation data is used to determine optimal hyperparameters (e.g., the number of epochs for training). It was added here just for simplicity.

## 7.2 Multi-Layer Perceptron

A single-layer perceptron solves linear binomial or multinomial classification problems. Often, a more complex topology is needed that can, at least in principle, model arbitrary decision boundaries. Similar to traditional methods, the network is

extended by implementing a non-linear transformation of features. Ideally, it maps a non-linear problem to a linear problem. Although such transformation always exists, traditional methods define the non-linear transformation without knowing the distribution of samples in feature space. There is no guarantee that a selected transformation maps to linearly separable features. The beauty of using networks is that arbitrary mappings can be represented by it. An optimal mapping can be trained from examples. Hence, the network is a universal solver for classification problems that minimizes the number of erroneous decisions.

This sounds too good to be true and indeed it is. The statement above is true under conditions that usually cannot be met. However, it is still an enormously important result as it paves the way for a generic trainable classification architecture with properties that are directly related to the performance of the classifier. Hence, it is no longer the intuition or the ingenuity of the engineer that produces a good classifier but simply a matter of proper training from samples. Engineering is still necessary but this relates to minimizing adverse effects because constraints for an optimal classifier are not met.

### *7.2.1 A Universal, Trainable Classifier*

For arriving at a universal classifier, two problems need to be solved:

- A non-linear mapping from original features to linearly separable features has to be defined that guarantees linearity of the result for arbitrary distributions.
- A network has to be designed that can represent this mapping.

The answer to the first problem is simple: Such a mapping is always possible. For the multinomial case, log-linear odds *log_lin_odds* for each class with respect to the pivotal class have to be generated. If *log_lin_odds* is a monotonous non-linear function, then an inverse $log\_lin\_odds^{-1}$ exists which is the desired non-linear mapping. If it is not monotonous, it is partitioned into piecewise monotonous functions and inverted separately for each interval. The non-linear function is then piecewise continuous with a finite number of discontinuities.

To solve the second problem, we need a network topology which is capable to represent an arbitrary non-linear function. It turns out that this can be realized for almost all functions by adding a second layer to the perceptron (see Fig. 7.12). This layer is called a *hidden layer* as it is not directly observable.

The *universal approximation theorem*[6] predicts that a two-layer perceptron with suitable non-linear activation function can represent any function, if the number of neurons in second layer is allowed to be arbitrarily large. The theorem has first been

[6]There are different variants that state this theorem for different network topologies, see, e.g., Scarselli and Tsoi (1998).

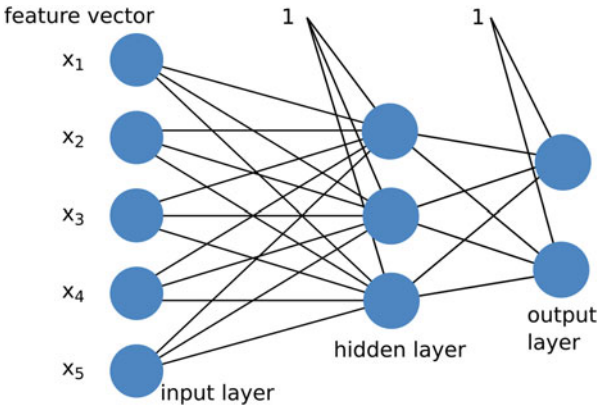



**Fig. 7.12** A multi-layer perceptron (MLP) adds one or more hidden layers to the perceptron architecture. Since the output of each perceptron is a non-linear function of the input, this adds a non-linear mapping to the network

proved for the sigmoid function as activation function and later for a number of other popular non-linear activation functions.

A very nice sketch of a visual proof of this theorem can be found in Nielsen (2015). Using the sigmoid function as activation function, the basic idea in this sketch is to represent an arbitrary function with finite support by a piecewise approximation with a finite number of constant functions. For a one-dimensional function, a constant function with support width $d$ at some location $x$ is approximated by subtraction of two sigmoid functions that are shifted by $x$ and $x + d$, respectively, where the second function has negative weights.[7] This can be modeled by two nodes in the layer with weights $f$ and $-f$. The value of $f$ is constant in this interval (see Fig. 7.13).

A sequence of such functions with support $[0,d]$, $[d,2d]$,. . ., $[(n\text{-}1)d,nd]$ can now be represented by twice as many nodes as there are intervals (see Fig. 7.14a). The extent of the support $[0,nd]$ for the function to be approximated is increased by increasing the number of nodes. The extent $d$ of the approximating functions is decreased by increasing the number of nodes as well. Hence, if the number of nodes is allowed to be arbitrarily large, the support for the function may be arbitrary large with arbitrarily high approximation accuracy. The opposite is true as well. If, for practical reasons, the number of nodes in the layer is limited, an approximation is still possible albeit with lower accuracy.

Extension to higher-dimensional input feature vectors does not change the argumentation (see Fig. 7.14b). It just requires superposition of several subtracted sigmoid function pairs that are rotated in the input space. It increases the number of necessary nodes for a given level of accuracy.

Hence, given enough training samples without artifacts in the features, the optimal mapping for classification can be learned from the samples. This is because the log-linear regression modeled by the connections between the hidden layer and

[7] It is an approximation that can be arbitrarily exact by increasing the scale factor in the sigmoid function.

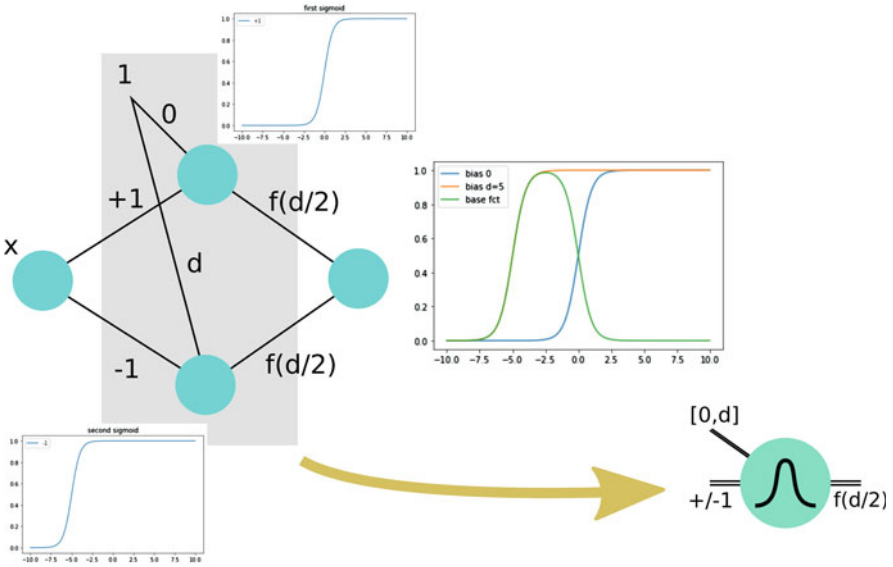



**Fig. 7.13** An input function $x$ can be mapped on an arbitrary function since it is possible to construct an approximation by a sequence of shifted functions. These are each a subtraction of two sigmoid functions that are shifted by some $d > 0$ (in this case $d = 5$). Smaller values for $d$ result in a better approximation of an arbitrary mapping but require more nodes in the hidden layer. The trainable weight at the output of the hidden layer learns the function value $f(d/2)$ in the interval $[0,d]$

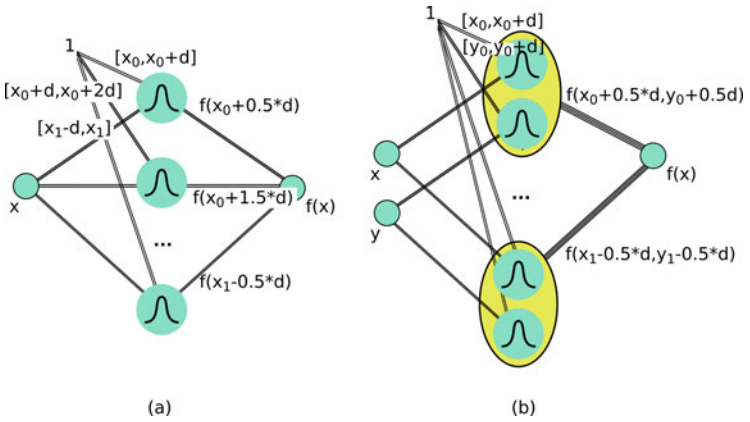



**Fig. 7.14** (**a**) A sequence of double nodes with finite support maps the input function $x$ on an arbitrary function $f(x)$ that is described by a sequence of its function values. Since every node pair in the hidden layer submits a non-zero value only in its support region, every node pair stands for one function value. (**b**) For multi-dimensional input, super nodes are created that consists of as many node pairs as the input has dimensions

the output layer improves when the output of the hidden layer approaches linearity. Hence, optimizing the cross entropy as loss function will guide weights between input layer and hidden layer toward producing a linear function.

### 7.2.2 Networks with More Than Two Layers

Constraints for a universal classifier are still difficult to fulfill. Even for fairly low-dimensional feature vectors the number of nodes that are necessary for a good approximation becomes prohibitively large.

Let us assume, for example, that we have a feature vector of length 10 with feature values normalized to the range [0,1] which should be approximated by cells with size $d = 0.01$ in each direction. Ten different pairs of sigmoid function are used to approximate the output function along the 10 dimensions. Hence $100^{10} = 10^{20}$ nodes are required in the layer that are fully connected to the 10 features in the input layer resulting in $10^{21}$ weights to be determined.

Weight computation by loss minimization from training samples in such a high-dimensional space will be only partially successful as the number of training samples is much lower than the number of weights. Hence, the optimization result covers just a subspace of feature space. Even in this subspace, the result will be far from optimal as the sample density is low and outliers may adversely influence results. Computed weights will thus not generalize well to the entirety of the unseen data.

For practical applications, the decision on the number of nodes in a network will be a compromise between the ability to represent a linear problem at the last layer and the ability to learn weight values from samples. A heuristic but often successful strategy for this is to add further hidden layers. For many complex problems, adding layers has been found to be more successful than adding nodes in a layer. Part of the reason is that the solution presented for proving the universal approximation theorem does not guarantee efficiency. Surely, some non-linear functions can be approximated by fewer nodes and in a different way. If the network topology supports this, a good approximation quality can be trained with much fewer nodes than the proof predicts.

Let us assume, for example, that the function to be estimated can be represented by a combination of simpler base functions. In this case, the network is trained to produce these base functions in the first layer that are then combined by nodes in a second layer (see Fig. 7.15). Since a base function is simpler than the function to be estimated, it can be represented by fewer nodes in the first layer. Non-linear combination in the second layer then requires to learn an additional set of weights between the two hidden layers. In total, the number of nodes and node connections will still be lower than in a single-layer topology. This is particularly advantageous for multidimensional input as the number of required nodes for a single-layer topology increases exponentially with the number of dimensions.

The set of functions to be estimated needs to be restricted in some way for this approach to work. Some kind of correlation as well as a restricted range of functions

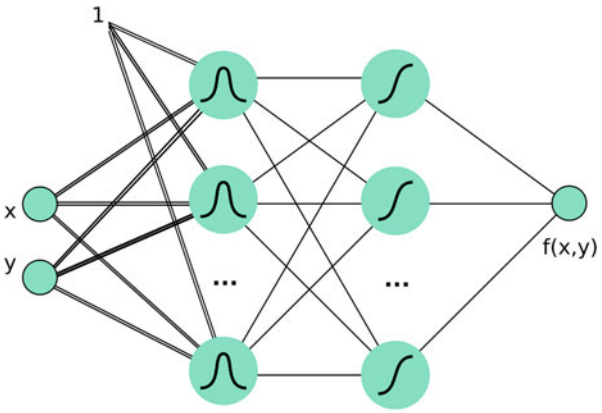



**Fig. 7.15** Instead of interpreting the output of the first hidden layer as function values of the function to be estimated, they can be interpreted as base functions that are combined to form the final function in a second layer of the perceptron

can often be assumed since features, generated from real world entities, are not entirely random. Similar features often have similar meaning. This regularity is exploited in a multi-layer topology with fewer nodes and trainable weights. For a strictly sequential networks with nodes of a layer being connected only to nodes of the next layer, network training should generate a sequence of approximations where the next layer always corrects for deficiencies of the previous layer. This should be particularly effective if the activation functions and the class-specific likelihood functions are smooth. In this case, the first layer may already produce a set of good approximations that is then refined by recombination in the next layer.

Such a *multi-layer perceptron* (MLP) with a restricted number of nodes does not guarantee that input features are mapped on output features with log-linear odds. However, it provides the means to train an approximation based on a limited set of training samples. Regarding the partitioning of feature space for classification, it is actually advisable to prefer deeper networks with fewer nodes. A representation of a complex partitioning in a single layer is possible but may lead to overfitting. If, for instance, the strategy above would be used, nodes could represent just small regions in feature space (see Fig. 7.16). It would be difficult for a sparse distribution of samples to infer the unknown distribution of the entirety of unknown samples. Hence, the network would probably overfit to the training data. Using several hidden layers allows to define less complex partitions at every layer. It restricts the set of representable shapes of the partitions but it is simpler to derive representative base partitions that can be recombined in later layers.

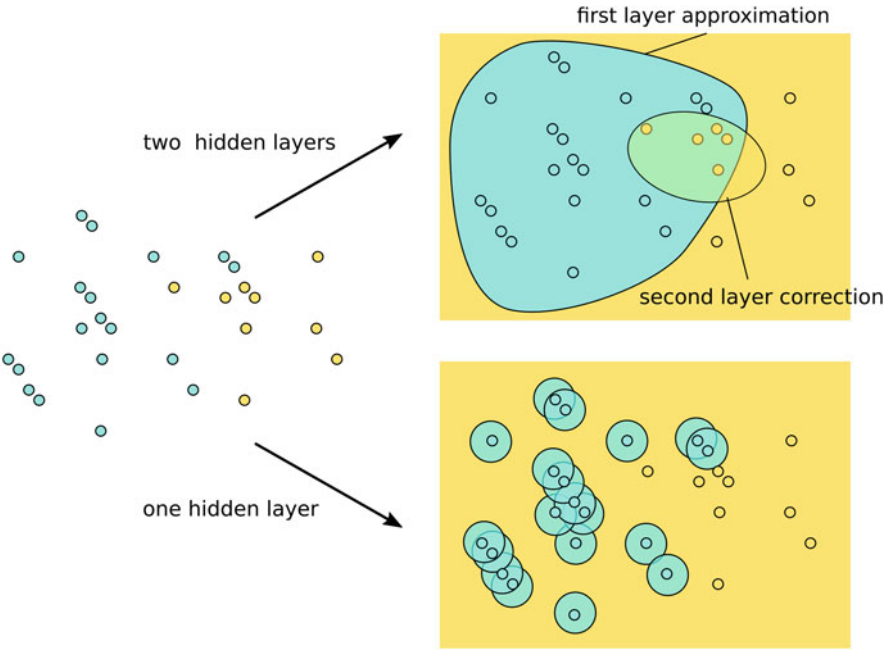



**Fig. 7.16** Sketch of a partitioning of feature space for a binomial classification problem. If a two-hidden-layer topology is chosen, the concave shape of the foreground class is modeled by a convex shape on the first layer that is then corrected on the next layer. For a single hidden layer, the concave shape is created by combining many simple convex shapes that do not generalize well

## 7.3 Training a Multi-Layer Perceptron

Weights to train a MLP are adapted by gradient descent. Independent computation of partial derivatives as presented earlier would make this a very costly undertaking. Fortunately, the structure of the network enables to extend backpropagation from single-layer perceptrons to MLP.

### *7.3.1 The Backpropagation Algorithm*

To understand how the backpropagation algorithm works, let us start with a very simple MLP with a single input node, just one node per layer, and with three layers (see Fig. 7.17).

The value at the input node is $x$. In each of the three layers, the activation function is $a$ and its derivative is $a'$. If, for instance, the activation function is $a(x) = sigmoid(x)$, then $a'(x) = sigmoid(x)(1-sigmoid(x))$. The final result is compared to the

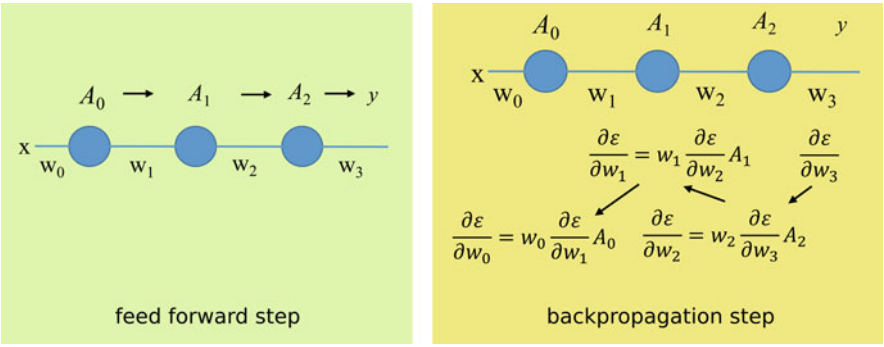



**Fig. 7.17** Simple multi-layer network with one node per layer. The input is fed through the network in the feedforward step. The resulting loss is then backpropagated for computing the partial derivatives along the way

expected output using some loss function $l$ (e.g., the categorical cross entropy) with derivative $l'$.

For computing the output, the input is fed through the network until the last node is reached resulting in a sequence of intermediate results $A_i$ at layer $i$:

$$A_0(w_0, x) = a(w_0 x) \tag{7.20}$$

$$A_1(w_0, w_1, x) = a(w_1 A_0(x)) = a(w_1 a(w_0 x)) \tag{7.21}$$

$$A_2(w_0, w_1, w_2, x) = a(w_2 A_1(w_0, w_1, x)) = a(w_2 a(w_1 a(w_0 x))). \tag{7.22}$$

The output $A_2(w_0, w_1, w_2, x)$ of the last layer is then subjected to the loss function

$$l(A_2(w_0, w_1, w_2, x)) = l(a(w_2 a(w_1 a(w_0 x)))). \tag{7.23}$$

The partial derivatives are now computed backward beginning with $w_2$. Applying the chain rule to $l(A_2(w_0, w_1, w_2, x))$

$$[l(A_2(w_0, w_1, w_2, x))]' = l'(A_2(w_0, w_1, w_2, x)) A_2{}'(w_0, w_1, w_2, x). \tag{7.24}$$

Since $A_2{}'(w_0, w_1, w_2, x) = a'(w_2 A_1(w_0, w_1, x))$, we apply the chain rule a second time arriving at

$$\begin{aligned} \frac{\partial l}{\partial w_2} &= l'(A_2(w_0, w_1, w_2, x)) a'(w_2 A_1(w_0, w_1, x)) A_1(w_0, w_1, x) \\ &= D_2(w_0, w_1, w_2, x) A_1(w_0, w_1, x). \end{aligned} \tag{7.25}$$

For computing this partial derivative, we re-use intermediate results for $A_1$ and $A_2$. For the next partial derivative $\frac{\partial l}{\partial w_1}$, we apply the chain rule again to Eq. 7.25 arriving at

$$\begin{aligned}\frac{\partial l}{\partial w_1} &= l'\left(A_2(w_0, w_1, w_2, x)\right) a'\left(w_2 A_1(w_0, w_1, x)\right) a'\left(w_1 A_0(w_0, x)\right) A_0(w_0, x) \\ &= D_2(w_0, w_1, w_2, x) a'(w_1 A_0(w_0, x)) A_0(w_0, x) \\ &= D_1(w_0, w_1, w_2, x) A_0(w_0, x),\end{aligned} \tag{7.26}$$

where we re-use the quantity $D_2$ from Eq. 7.25. For the last partial derivative, we again apply the chain rule and, re-using $D_1$, arrive at

$$\frac{\partial l}{\partial w_0} = D_1(w_0, w_1, w_2, x) a'(w_0 x) x. \tag{7.27}$$

It should now be clear why the algorithm is called backpropagation algorithm. First, the input $x$ is propagated forward through the network and $A_0$, $A_1$, and $A_2$ are computed. Then, the derivative of the loss is computed and used as weight to compute the derivatives with respect to the weights $w_2$, $w_1$, and $w_0$. The derivative for a weight in some intermediate layer is always weighted with the derivatives computed before it. Hence, the information is propagated backward through the network until the input layer is reached. Increasing the number of layers does not change this procedure.

If more than one node exists in a layer, the sum rule of differentiation needs to be applied. Hence, computation of $D$ involves summing up over all nodes that are connected to this node. With the following notation for layer $k$:

- $A_{k,j}$—result at node $j$ of layer $k$.
- $D_{k,j}$—intermediate result at node $j$ of layer $k$ during backpropagation.
- $w_{k,i,j}$—weight at the edge connecting node $j$ of layer $k$ with node $i$ at layer $k + 1$.

the algorithm first computes all values for $A_{k,j}$ for all layers $k = 1,K$ in the feedforward step and stores them with the nodes.

Then, the derivative of the loss is computed for all output nodes $j$, $j = 1,M_K$:

$$\begin{aligned}&\text{for } j = 1, M_K \text{ do} \\ &\quad D_{K,j} = l'\left(A_{K,j}\right) a'\left(A_{K,j}\right).\end{aligned}$$

With this the intermediate results $D_{k,\ j}$ at earlier layers are computed by backpropagation

$$\begin{aligned}&\text{for } k = K - 1, 1 \text{ do} \\ &\quad \text{for } j = 1, M_k \text{ do} \\ &\qquad D_{sum} = \sum_{i=1}^{M_{k+1}} D_{k+1,i} w_{k,i,j} \\ &\qquad D_{k,j} = a'\left(A_{k,j}\right) D_{\text{sum}}.\end{aligned}$$

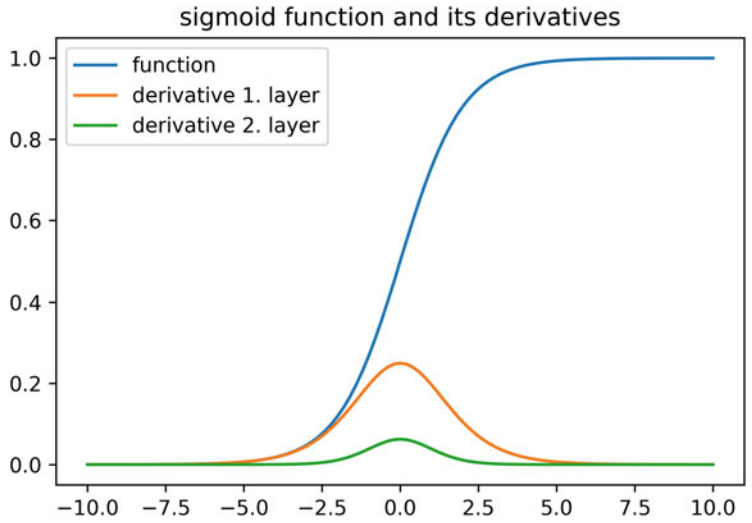



**Fig. 7.18** The maximum of the derivative of the sigmoid function is 0.25. Since partial derivatives are computed by multiplying the input at a layer with derivatives from previous layers, the partial derivatives decrease quickly during backpropagation

Finally, the partial derivatives are

$$\frac{\partial l}{\partial w_{k,i,j}} = D_{k,j} A_{k-1,j}. \tag{7.28}$$

Weights are then updated by moving against the direction of the gradient by a step size defined by the learning rate. With backpropagation, networks of arbitrary depth can be trained as long as they do not contain cycles.

Given the argumentation about finding the best non-linear mapping of input features, networks are preferred that are deep rather than wide. However, MLPs seldom have more than three or four hidden layers because too many layers cause slow convergence of the optimization. The effect is called *vanishing gradient* and becomes more serious the deeper a network is.

The absolute value of partial derivatives decreases from layer to layer through backpropagation because

- at every new layer $k-1$ the intermediate results $D_{k,\,i}$ from the previous layer are distributed over all incoming edges.
- the new value $D_{k-1,\,j}$ results from multiplication with the derivative $a'(A_{k,\,j})$ with the summed results from the $D_{k,i}$. Since the maximum of the derivative of the sigmoid function is 0.25, this decreases the value for derivatives with every layer (see Fig. 7.18).

The first problem cannot be solved in a strictly sequential network and restricts the number of layers that can be trained. The latter can be eased by changing the activation function. Remember that the sigmoid function or the softmax function is only needed for computing the probabilities of the log-linear model of the last layer. Universal approximation ability has also been proved for other activation functions. A commonly used function is the rectified linear unit (ReLU)[8] which is defined by

$$\mathrm{ReLU}(x) = \begin{cases} x & \text{for } x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{7.29}$$

The derivative for $x > 0$ is 1 and that for $x < 0$ is 0. For positive input, sequential multiplication of the derivative with intermediate results from the previous layer does not change the value. Although this is not true for the negative part, ReLU activation is an often-selected choice for all but the last layer.

**Classes and Functions in Python**
Construction, training, and use of a multi-layer perceptron is not much different to a single-layer perceptron except that several denser layers are added. Only the last layer has sigmoid/softmax activation. All other layers use activation function such as ReLU. For example, a sequence to add three layers with the input layer having 784 features would look like this.

```
# data preparation and construction of empty network 'model'
...
# add layers
model.add(Dense(100, input_shape=(784,),activation='relu',
                name='d01'))
model.add(Dense(100, activation='relu', name='d02'))
model.add(Dense(10, activation='softmax',name='d03'))
# compile and fit the model
...
```

### 7.3.2 *The Adam Optimizer*

The step size for the correction step using the partial derivatives in Eq. 7.28 depends on the learning rate. Determining an optimal learning rate can be a problem, however. Theoretically, gradient descent requires infinitely many infinitesimally

[8] Subtraction of a shifted ReLU from another ReLU produces a function that is similar to the sigmoid function. For universal approximation this can be used as a base function instead of the sigmoid function. The remainder of the argumentation is similar to that when using the sigmoid function.

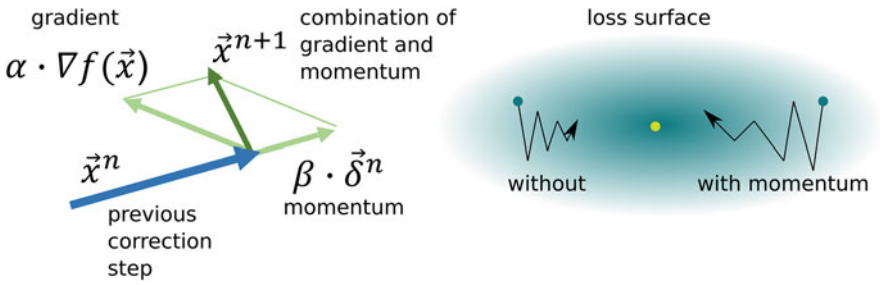



**Fig. 7.19** The introduction of a momentum term reduces oscillation effects by adding a certain percentage of the previous change (the momentum) to the change predicted by the current gradient

small steps. For practical purposes, the learning rate is finite. If it is too large, the minimum may be missed and the search oscillates around the minimum. For complex loss functions such as the high-dimensional loss function of neural networks this is often noticeable as erratic behavior of the loss values during optimization. Choosing a small learning rate instead slows down convergence. If learning rate and gradient are too small it may even cause premature termination.

In consequence, various adaptive schemes have been adopted. Simple adaptation decreases an initial learning rate by a fixed factor. More advanced methods use momentum terms that drive optimization in a direction that is also influenced by previous changes (see Fig. 7.19, for different adaptation schemes see Ruder, 2016). Adding momentum to gradient descent causes earlier weight changes to influence the current weight change.

A common adaptation scheme for network training is *Adam* (*ada*ptive *moment* estimation, see Kingma & Ba, 2014) that uses an adaptive momentum and an adaptive learning rate. The adaptive momentum vector $\mathbf{m}_t$ depends on the previous momentum $\mathbf{m}_{t\text{-}1}$ and the current gradient of the loss function (for notational simplicity we put all the weights of a network at iteration $t$ into a common weight vector $\mathbf{w}_t$):

$$\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\nabla l(\mathbf{w}_t). \tag{7.30}$$

Earlier changes have a lesser influence on the momentum. The value of $\beta_1 < 1$ is user-determined. The authors of the original paper suggested $\beta_1 = 0.9$. The momentum vector is bias-corrected

$$\widehat{\mathbf{m}}_t = \frac{\mathbf{m}_t}{1-\beta_1}, \tag{7.31}$$

as the estimate of an average direction of change for $\mathbf{w}$ increases with the number of summed gradients.

The learning rate of the Adam optimizer is adaptive as well and depends on accumulated squared partial derivatives $g_{t,ii}$:

$$g_{t,ii} = \beta_2 g_{t-1,ii} + (1-\beta_2)\frac{\partial l}{\partial w_i}(\mathbf{w}_t)^2, \tag{7.32}$$

where $\frac{\partial l}{\partial w_i}(\mathbf{w}_t)$ is the partial derivative of the loss function $l$ with respect to $w_i$ for the current weight estimate $\mathbf{w}_t$. Squared partial derivatives from previous steps are weighted by $\beta_2$ against the current lengths. Hence, older gradients will have a lesser influence than newer gradients. The user-defined value of $\beta_2 < 1$ has been set to $\beta_2 = 0.999$ in the original publication. Setting $\beta_2$ to such a high value means that random fluctuations of gradients—not uncommon for stochastic gradient descent—have little influence on the learning rate.

As the accuracy of the estimated accumulated variance in Eq. 7.32 depends on $t$, it is bias-corrected as well by

$$\widehat{g}_{t,ii} = \frac{g_{t,ii}}{1-\beta_2^t}. \tag{7.33}$$

The bias-corrected values are used to construct a matrix $G_t$ that contains the specific learning rates $\widehat{G}_{t,ii}$:

$$\widehat{\mathbf{G}}_t = \begin{pmatrix} \frac{1}{\widehat{g}_{t,11}+\varepsilon} & \cdots & 0 \\ \cdots & \ddots & \cdots \\ 0 & \cdots & \frac{1}{\widehat{g}_{t,nn}+\varepsilon} \end{pmatrix}. \tag{7.34}$$

Here, $n$ is the number of weights of $\mathbf{w}_t$. Hence, the learning rate will decrease with larger accumulated variances of squared partial derivatives. The parameter $\varepsilon$ is a small value to avoid division by 0 and has been set to $\varepsilon = 10^{-8}$ by the authors.

Finally, momentum and learning rate are combined to give the following update rule:

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \widehat{\mathbf{G}}_t \cdot \widehat{\mathbf{m}}_t. \tag{7.35}$$

Here, $\eta$ controls the influence of the momentum on the correction.

## 7.4 Exercises

### *7.4.1 Programming Project P7.1: MNIST- and CIFAR10-Labeling by MLP*

Construct a multi-layer perceptron to classify the MNIST data. Use the original input values and orientation histograms as input. Try to find the smallest possible network

(the fewest nodes, the fewest layers). Determine suitable minibatch sizes and number of epochs that are necessary for convergence. What does this tell you about the problem complexity. Discuss the results in comparison to traditional MNIST classifiers from previous exercises.

Adapt the model to classify the CIFAR10 data. What is the best accuracy that you can get and with what kind of network? What does it tell you about the differences between the two data sets?

The two data sets can be accessed using Keras classes:

```
from tensorflow.keras.datasets import mnist, cifar10
(X_train1, y_train1), (X_test1, y_test1) = mnist.load_data()
(X_train2, y_train2), (X_test2, y_test2) = cifar10.load_data()
```

The data comes in the original shape (gray level images for MNIST and RGB images for CIFAR10) and has to be reshaped and normalized for directly using the pixel values as feature.

### *7.4.2 Exercise Questions*

- What would be the advantages and disadvantages of a universal estimator as a solution to an image classification problem?
- Why was the perceptron not considered an advance for image classification?
- What would happen if a linear activation function were used for multi-layer networks?
- Which activation function is used for the last layer in a classification network? Why?
- You observe an erratic behavior of the loss function during training. What are possible reasons for this?
- What is the role of the loss function in a classification network? What is the loss function measured by categorical cross entropy?
- Why does ReLU alleviate the problem of a vanishing gradient? Why is it not used in the last layer of a classification network?
- Why is overfitting a much bigger problem in networks than in conventional classification methods? Why does overfitting still have to be accepted?

## References

Bear M, Connors B, Paradiso MA (2020). Neuroscience: Exploring the brain, enhanced edition. .

Boden, M. (2002). A guide to recurrent neural networks and backpropagation. *The Dallas Project, 2*(2), 1–10.

Kingma, D. P., & Ba, J. (2014). *Adam: A method for stochastic optimization*. arXiv preprint arXiv:1412.6980.

Nielsen, M. A. (2015). *Neural networks and deep learning*. Determination Press.

Raschka, S., Liu, Y. H., Mirjalili, V., & Dzhulgakov, D. (2022). *Machine learning with PyTorch and Scikit-learn: Develop machine learning and deep learning models with python*. Packt Publishing.

Rosenblatt, F. (1957). *The perceptron, a perceiving and recognizing automaton project Para*. Cornell Aeronautical Laboratory.

Ruder, S. (2016). *An overview of gradient descent optimization algorithms*. arXiv preprint arXiv:1609.04747.

Scarselli, F., & Tsoi, A. C. (1998). Universal approximation using feedforward neural networks: A survey of some existing methods, and some new results. *Neural Networks, 11*(1), 15–37.

# Chapter 8
# Feature Extraction by Convolutional Neural Network


**Abstract** An optimal mapping from pixel values to class labels by a MLP requires too many nodes and layers for successful training. Convolutional neural networks extend the concept by adding a different type of layer that restricts network expressivity without compromising classifier performance. Part of the network is a sequence of convolutional building blocks that extracts features from images which are submitted to fully connected layers for classification.

Feature extraction in a convolutional building block happens by feeding the input through a sequence of convolution layers followed by a pooling step. Convolution layers are applied to spatially organized maps and learn a set of convolution kernels followed by a non-linear activation function to extract intermediate features from the input map. Nodes in a convolution layer are connected to a fixed neighborhood of input nodes and weights of connections to the neighborhood are equal for all output nodes. The pooling layer reduces the spatial resolution of the feature map.

The resulting network is deeper than the MLP but adds fewer weights per layer. We describe the effects of a deeper network on training and conclude with an initial experiment with a trained network.

A multi-layer perceptron is a universal approximator. Hence, it should be able to approximate the mapping to log-linear features at the last layer from any input, even from the pixels themselves. It actually works as we saw in the previous chapter by example of MNIST classification. However, if the semantic is less closely related to pixel values, the classification performance will be poor. Even for a rather simple problem such as the classification of the CIFAR10 data with its few pixels and many samples it will not be possible to achieve reasonable accuracy using a multi-layer perceptron (just try it!).

The poor performance is caused by the extremely large number of nodes necessary to approximate a complex mapping. It may be somewhat reduced by adding more layers instead of adding more nodes in the layers. However, the number of required nodes will still be high and lead to overfitting. A large number of layers will worsen the problem of vanishing gradients.

A solution to this dilemma is to define a different kind of network that makes use of known characteristics of information in images. It can safely be assumed that the

© The Author(s), under exclusive license to Springer Nature Singapore Pte Ltd. 2024

K. D. Toennies, *An Introduction to Image Classification*,
https://doi.org/10.1007/978-981-99-7882-3_8

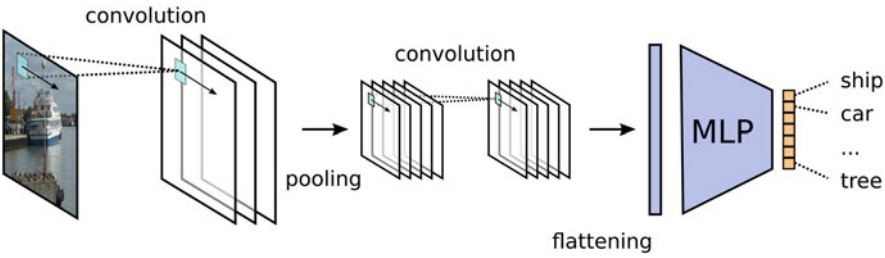



**Fig. 8.1** Topology of a convolutional neural network. Convolution layers learn convolution kernels that extract features. Pooling reduces the scale. After mapping the output to a one-dimensional feature vector, the features are classified by a multilayer perceptron (MLP)

correlation between pixel pairs decreases with distance between them. Large-scale correlations, if they exist, will relate to larger subsets of connected pixels. It may further be assumed that the semantics of pixels or pixel aggregates change little if they are shifted, scaled, or rotated. All this can be used to reduce the number of connections between nodes in a sequential network. Such network is no longer fully connected. *Convolution layers* implement the concepts of a limited range of correlation between pixels and shift invariance, whereas *pooling layers* add scale invariance to this (see Fig. 8.1).

## 8.1 The Convolution Layer

For processing of image features, spatial relationships between pixels need to be retained. Hence, the input layer transfers a spatially organized pixel map to the network. It enables a simple definition of a spatial neighborhood around each pixel.

### *8.1.1 Limited Perceptive Field, Shared Weights, and Filters*

The first convolution layer in a neural network is connected to the input layer. It has often the same number of nodes than the input layer. In this case, an output node in the convolution layer is uniquely associated with a corresponding node in the input layer. A node in the convolution layer is connected to its associated node in the input layer and its neighboring nodes. The size of the neighborhood is a hyperparameter of the network and defines what is called the *perceptive field* of the node (see Fig. 8.2). All nodes of the convolution layer have the same neighborhood size.

The concept of a *limited perceptive field* stems from cognitive neuroscience. It has been found that early processing in the retina and the visual cortex is organized in layers where the original spatial organization of the photoreceptor cells is retained. Furthermore, neurons are connected only to a limited number of neurons in the

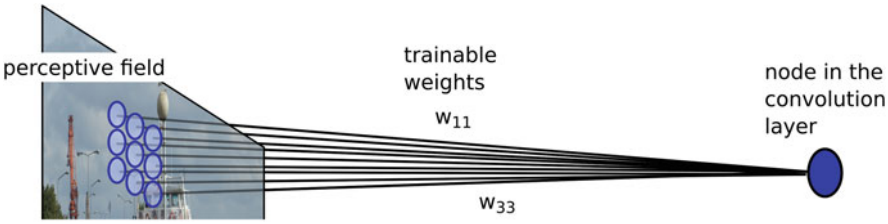



**Fig. 8.2** Nodes in the first convolution layer often have the same spatial dimensions than the input data. Hence, every node has a corresponding location in the image. A node is not fully connected but receives input from a neighborhood around its corresponding location in the input image. This is the perceptive field of this node

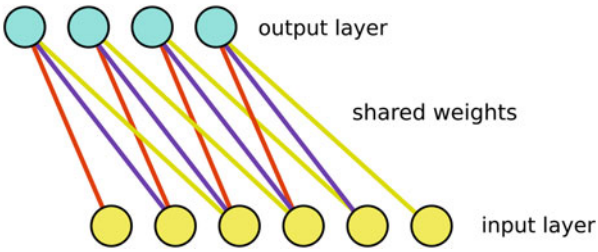



**Fig. 8.3** The shared-weights principle requires that corresponding edges between input and output layer (those with the same color in this picture) always have the same weights

previous layer. It has also been found that this kind of organization enables extraction of primitive features such as blobs, edges, lines, and corners.

Applying this to the design of an artificial neural network is not new. In the 1980s, the neocognitron was presented (Fukushima et al., 1983). It processes visual input through a number of spatially organized layers with decreasing resolution and node connections with limited perceptive field. Similar to the perceptron, nodes are connected by edges with trainable weights. It was shown that this network was able to extract image features to solve a simple classification task (handwritten digits).

Besides a limited perceptive field, the *shared weights* principle is another concept from the neocognitron that has been adopted in a modern convolutional neural network. Connections from nodes of the convolution layer to nodes of its perceptive field in the input layer have the same weight for all nodes (see Fig. 8.3). This implements a convolution with a kernel of fixed size (the perceptive field). Weights are the values of the convolution kernel.

Training a convolution layer will result in a linear filter with fixed-size kernel followed by the application of a non-linear activation function such as ReLU or the softmax function. It maps the input to an output *feature map* (also called *activation map*). The feature map is sometimes named after the kind of activation function (e.g., rectified linear unit feature map) to differentiate it from the output before applying the activation function (which is then called *raw output* or *raw feature map*). The input map is the image (if it is the first convolution layer) or the feature map from a previous convolution layer.

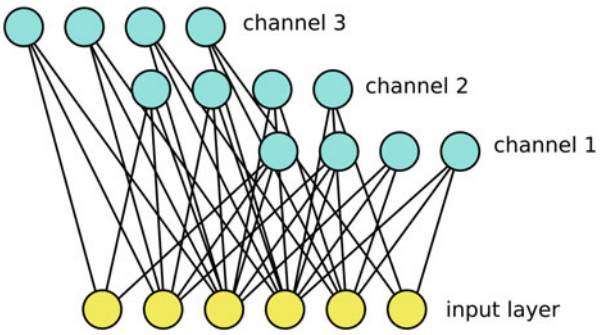



**Fig. 8.4** Input nodes are connected to several channels in the output layer. Each channel learns its own set of shared weights. Hence, each channel is a filter that extracts features from the input independent of all other channels

The result is a single feature map. It could be, for instance, a filter that highlights blobs of a certain size in the image. Since the diversity of low-level image characteristics will be larger than a single filter, a convolution layer has usually several filters. It is still called a single convolution layer but with an extra dimension which is called *channels*, *filters,* or *filter kernels* of the convolution layer (see Fig. 8.4). The number of channels is a hyperparameter. Its value depends on the richness of information that can be gained from the input and will be higher for larger perceptive fields.

The shared weights principle of convolution layers provides shift invariance, i.e., one of the three types of invariances that we already mentioned. It reduces degrees of freedom in the network without compromising the ability to extract image features. Rotational invariance, the second type of invariance, cannot be represented by a convolution layer. Rotationally invariant features need be represented by several differently rotated versions of a filter. When we later inspect filters of a trained network, we will see that this is indeed happening.

Network training independently adapts weights for the different channels. Hence, it may produce redundant features since two different filters may converge to the same or similar weights during training. Thus, the number of channels will be larger than necessary in order to be sure that all relevant information in the input is represented by a convolution layer.

Each channel is connected with nodes from the input layer in the same way. For a perceptive field of size $k \times k$ with $n$ channels, the number of weights to be trained will be $n \cdot k^2$. It is still less than the number of connections for a fully connected network, where the number of weights is the product of the number of input nodes with the number of nodes in the layer. Imagine an image of size 200 × 200 and a fully connected layer with 1000 nodes (which already constitutes a substantial information reduction given the 40,000 input nodes). Forty million weights had to be trained, while a convolution layer with perceptive field of 11 × 11 and 100 channels had just 12,100 trainable weights.

Fewer weights mean reduced capacity to represent information contained in the image. It is an advantage rather than a problem, however, as the reduced capacity stems from two assumptions:

- There is only a weak correlation between distant pixels.
- The information from the pixel position relative to the image coordinate system is irrelevant.

The former just puts experimental results from the cognitive sciences about perception into practice. Relationships between distant pixels are mostly irrelevant at a pixel scale even if they may be relevant at a coarser scale. Coarse scale correlations across larger distances will be addressed by building a multi-scale network with several convolution layers.

Invariance with respect to the image coordinate system is true in most cases. Moving an object in the image coordinate system does not change its semantics. Relative positions between object details are relevant, however. For pixels in the receptive field, this is representable by proper filter weights. For larger neighborhoods, this will again be addressed by building a multi-scale network from convolution layers.

### *8.1.2 Border Treatment*

Border treatment is a problem when carrying out a convolution. At the image border, the perceptive field exceeds the extent of the image. Hence, some of the input is undefined. For convolution in general, several solutions exist for the problem from which two are common in neural networks (see Fig. 8.5):

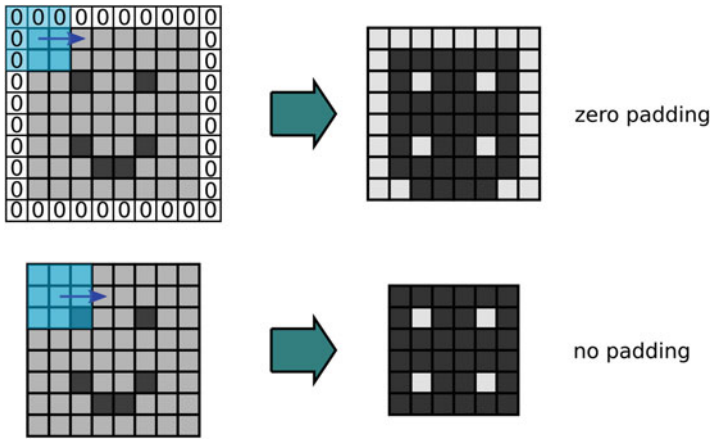



**Fig. 8.5** If zero padding is used, a band of zeros is padded around the image so that the perceptive field for every position in the picture is defined. The output has the same size than the input. If no padding is used, only pixels for which the perceptive field is completely inside the image produce an output. The output image gets smaller

- *Zero padding*: A band of nodes is padded to the input image and each node value in this band is set to zero. The width of the band is half the width of the perceptive field. Hence, the size of the convolution layer is equal to that of the input layer.
- *No padding*: The convolution is carried out only for nodes with fully defined perceptive field. It does not create artifacts but the size of the convolution layer is smaller than that of the input layer.

### 8.1.3 Multichannel Input

Convolution layers can accept multichannel input such as color images. If, for instance, a color image is represented as 3-channel RGB image, it is mapped to three-channel input nodes. Each channel of the input layer is connected to every channel in the convolution layer. The number of weights is now $k^2 \cdot n_1 \cdot n_2$, where $n_1$ is the number of channels in the input layer and $n_2$ that of the convolution layer. Training will find an independent set of convolutional kernels for each combination of the three color channels.

### 8.1.4 Stride

So far, shifting from one node to the next in the convolution layer corresponds to a shift by the same distance in the input layer. Although this is probably the most often implementation of a convolution layer, the shift may also be different between the two layers. The number of nodes to be shifted in the input layer for a shift by one node in the convolution layer is called *stride*.

A stride of two, for instance, would mean that a shift by one node in the convolution layer corresponds to a shift of the perceptive field by two nodes in the input layer (see Fig. 8.6). Hence, the convolution layer combines a convolution with a subsampling by a factor of two. In the first convolution layer, it is sometimes used to reduce input resolution. In intermediate layers it is a way to build a

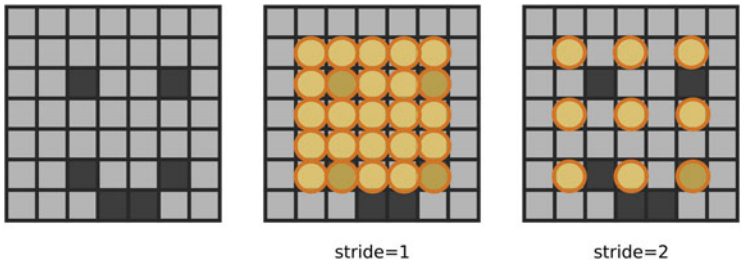



**Fig. 8.6** The stride indicates how many pixels are skipped when computing the convolution. A stride>1 combines the convolution with a subsampling of the input

multi-resolution representation of the image. Hence, it is a way to introduce scale as a characteristic of features.

**Classes and Functions in Python**
Convolution layers are classes of `tensorflow.keras.layers`. A 2-d convolution layer is an object of the class `Conv2d` that takes the number of filters and the size of the filter kernels as input. Parameters that you will often need are the strides, the type of padding, the activation function, and a name for the layer. If an input layer has not been specified as extra layer and it is the convolution layer that is directly connected to the input, there is also an argument `input_layer` where its shape may be submitted.

The following sequence adds two convolution layers to a model `model`.

```
from tensorflow.keras.layer import Conv2D
...
# add first layer of network, include input_shape (useful only, if
# no input layer has been specified); padding='same' is
# zero padding, padding='valid' would be no padding
model.add(Conv2D(cnn_layers[i], kernel_size=(3,3),
                 strides=(1,1), padding='same',
                 activation='relu', input_shape=(28,28),
                 name='conv01'))
# add second layer
model.add(Conv2D(cnn_layers[i], kernel_size=(3,3),
                 strides=(1,1), padding='same',
                 activation='relu', name='conv02'))
```

## 8.2 Convolutional Building Blocks

A convolution layer combines trainable convolutions with the execution of an activation function on the result. The activation function introduces non-linearity which—similar to the multi-layer perceptron—increases the expressiveness of a stack of layers. This kind of feature extraction has its equivalent in traditional image classification. Initial image features are aggregated and non-linearly transformed there as well to create new features that extract relevant image information and cluster well in feature space.

Since we have two different goals—reduce features and create more meaningful features—combining several convolution layers is done on two levels:

- A *convolutional building block* (CBB) with several convolution layers learns meaningful features at a given resolution and generates a condensed representation with lower resolution.

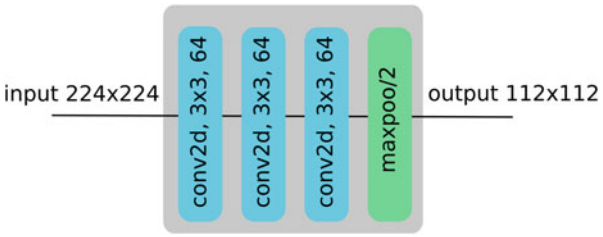



**Fig. 8.7** A convolutional building block consists of several convolution layers followed by a pooling layer that reduces the spatial resolution of the feature maps

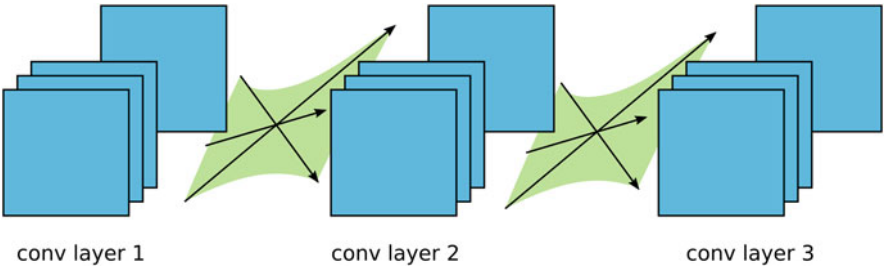



**Fig. 8.8** A convolutional building block contains usually several convolution layers. Between layers every output channel of the previous layer is connected to every input channel of the next layer. Hence, the input of a layer is a weighted sum of features from the previous layer. Weights will be trained as part of the network optimization

- A sequence of CBBs creates ever more meaningful features that are increasingly less dependent on local neighborhoods in the original image.

A convolutional building block consists of a sequence of convolution layers followed by resolution reduction step, e.g., by pooling, see below (see Fig. 8.7).

### *8.2.1 Sequences of Convolution Layers in a CBB*

Channels of two subsequent convolution layers are fully connected. It is a concept that we already know from multi-channel input to the first convolution layer. Every kernel on the next layer learns a weighted combination of activations from filters in the previous layer (see Fig. 8.8). The perceptive field with respect to the original input is then a combination of the perceptive fields of the two layers. If, for instance, the perceptive field is $5 \times 5$ for each layer, the second layer "sees" $9 \times 9$ pixels of the input layer through all possible combinations of filters in the first layer.

### 8.2.2 Pooling

The last layer of a CBB reduces the spatial resolution. A means to do so is to use a stride>1 in a convolution layer. Often, a specialized pooling layer is preferred as it enables the specification of the kind of interpolation for resolution reduction.

Pooling maps $s \times s$ nodes of a pooling region in a convolution layer to a single node. The value for $s$ is a hyperparameter. If set to $s = 2$, it halves the resolution for each pooling step. Different kinds of pooling exist:

- *Max-pooling* uses no interpolation and selects always the largest value in the pooling region (see Fig. 8.9a). Max-pooling implements the winner-takes-it-all strategy known from cognitive neuroscience. It introduces local translation invariance to feature processing.
- Average pooling computes the average in the pooling region (see Fig. 8.9b). It is easier to invert than max-pooling but may cause attenuation of high-resolution signals.
- Softmax pooling is a compromise between max-pooling and average pooling. It computes the softmax function over the pooling area and submits this to the output layer.

**Classes and Functions in Python**
Pooling layers are classes of `tensorflow.keras.layers`. A pooling layer to carry out max-pooling on a 2-d feature map is an object of the class `MaxPool2D`. Important parameters are the pooling size and the name of the layer. Pooling layers are added to the model after convolution layers to decrease the resolution of the feature maps. For a model `model` the following sequence adds a pooling layer:

```
from tensorflow.keras.layer import MaxPool2d
...
# pooling reduce feature maps by a factor of 2 (defined by pool_size)
model.add(MaxPool2D(pool_size=(2,2), name='pool01'))
```

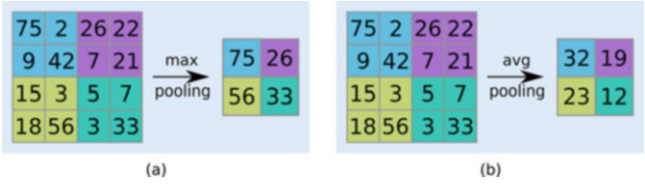



**Fig. 8.9** Comparison between max-pooling and average pooling. Strong responses before pooling do not survive as well when average pooling is used

### 8.2.3 1 × 1 Convolutions

Pooling intentionally reduces resolution of feature maps transmitted to the next layer. Possible redundancies between channels are not detected and removed. Such redundancies may be removed by adding a convolution layer with a 1 × 1 perceptive field (see Fig. 8.10). A 1 × 1 convolution weighs the input and subjects it to the activation function. If used for feature reduction, the number of channels in the next layer will be smaller than that in the current layer. It forces the network to express information of the current layer by a smaller number of channels in the next layer.

The process bears some similarity to principal component analysis presented earlier. If we assume that the activation function is linear (which, for ReLU, it is for positive input) the mapping is linear just as in PCA. Unlike PCA, the reduction is not the result of some assumption about linear correlation and its relevance for classification but can be learned from labeled samples. Because of the non-linear activation function, the reduction extends to the evaluation of non-linear relations between input features and classification output.

### 8.2.4 Stacking Building Blocks

A single building block computes relevant features at a given resolution and reduces the resolution by some factor. For feature computation at different levels of resolution, several building blocks are stacked (see Fig. 8.11).

Because of the resolution reduction, the perceptive field of later building blocks with respect to the image gets larger. Hence, the number of channels per convolution layer should increase, since a node in a feature map in a building block represents accumulated information from ever larger regions in the original image.

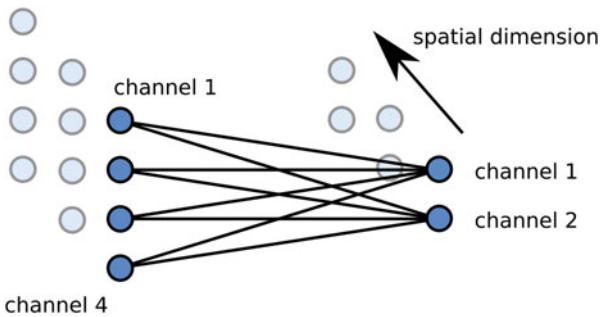



**Fig. 8.10** 1×1 convolutions between layers can be used to change the number of channels. Since channels are fully connected between layers, an arbitrary number of outputs can be connected by a trained, weighted combination of channels from the previous layer. This can be used, e.g., to reduce the number of channels

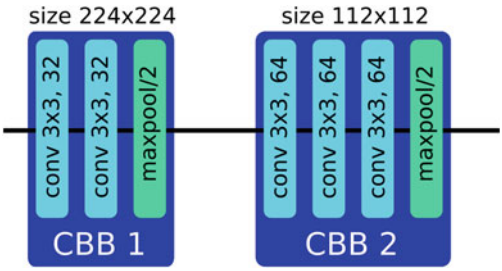



**Fig. 8.11** Convolutional building blocks (CBBs) are stacked to form the convolutional part of the neural network. Each CBB has its own number of convolution layers with each having its own perceptive field and number of channels. The notation above is common for describing complex networks, giving the size of the data submitted to a layer, the perceptive field ($3 \times 3$ in the example), the number of channels (32 and 64 in the example), and the kind of pooling

## 8.3 End-to-End Learning

So far, our network architecture is able to represent relevant image features. Just what is relevant is subject to training from samples. As the only information available from training samples is their label, we need a classifier to decide what features are best suited for a given classification task. Hence, the sequence of CBBs is combined with the fully connected layers of a MLP to form the convolutional neural network (CNN). The output from the last CBB of the network is turned into a feature vector (called *flattening* of the features, see Sect. 7.1.3 for the implementation with Keras/Python) which is then input to the MLP part of the CNN. The MLP part is often just called the fully connected or dense layers.

Weights for the complete network are learned at the same time. This is known as *end-to-end learning* of a classifier as it enables classifier training directly from pixels in an image (see Fig. 8.12). Since the convolutional part of the network is assumed to produce easily separable features, the MLP does not have to be very complex. For fairly simple problems such as the training of a classifier for the CIFAR10 reference data set or for one of the various subsets of the ImageNet data (Imagenette, Imagewoof, etc.), a MLP with just a single hidden layer with few nodes may already be sufficient. Training will automatically determine how much of the non-linear transformation is delegated to the feature computation part and how much to the fully connected classifier part.

It is much more efficient than leaving the feature extraction to a fully connected MLP. For an example, compare results for the CIFAR10 data set that was classified by a pure MLP and a CNN in Fig. 8.13. The two networks have approximately the same number of trainable weights. However, instead of using just fully connected layers, the CNN combines convolution layers with fully connected layers. Apparently, the restriction by shared weights and a limited perceptive field matched the underlying semantics in the image much better leading to a substantially improved classification accuracy.

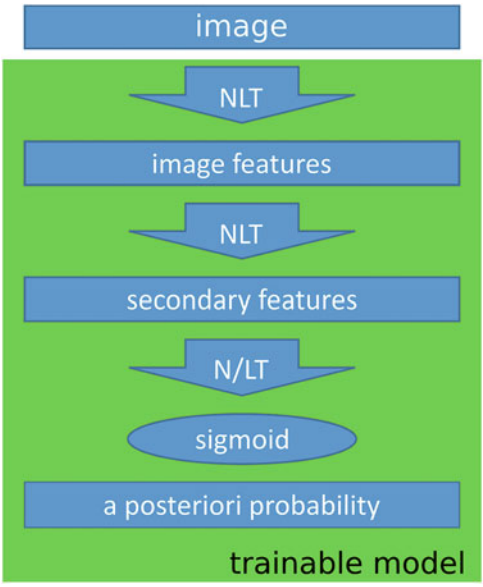



**Fig. 8.12** The combination of the fully connected MLP with convolutional building blocks creates a classifier that learns, extracts, and transforms image features as well as classifies based on transformed features

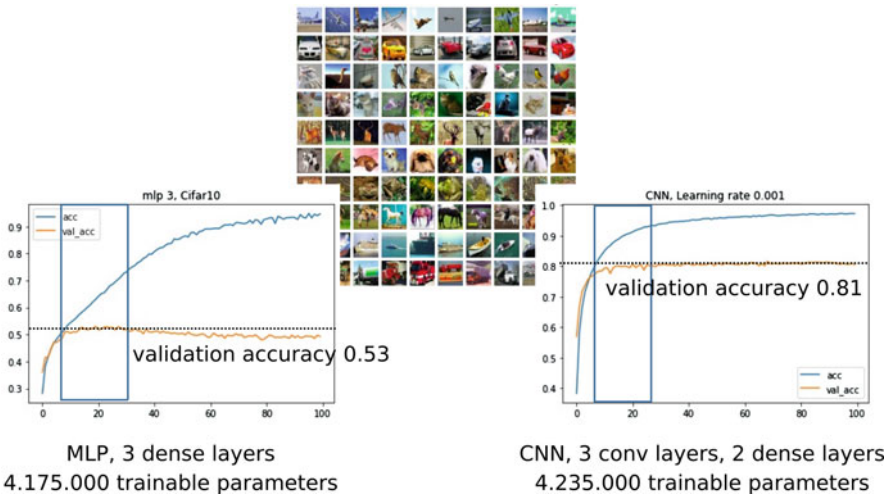



**Fig. 8.13** Adding convolutional building blocks apparently matches the underlying structure of image information better than a pure fully connected MLP. In this example for classification of CIFAR10 data, two networks with similar numbers of trainable weights but without and with convolution layers perform vastly different

### *8.3.1 Gradient Descent in a Convolutional Neural Network*

The convolutional neural network sketched above is still a sequential network. The main differences are:

- Nodes are not fully connected to the previous layer.
- Weights are shared for each channel.
- A pooling step decides what signal is transmitted to the next building block.

Backpropagation can still be carried out for gradient descent optimization. If nodes are not fully connected, weighted intermediate results are distributed to connected nodes only. Weight sharing requires that the partial derivatives of a channel in a given layer have to be averaged before being applied as a correction step. Pooling requires a specific propagation depending on the pooling scheme. For the interpolating schemes, it means that backpropagation is weighted with the interpolation weights. For max-pooling, backpropagation happens along the edge that transmitted the maximum response. In summary, optimization of a convolutional neural networks uses the backpropagation algorithm with some minor modifications.

Since CNNs have many more layers than MLPs, care has to be taken for choosing a good activation function. We already argued in the previous chapter that using the sigmoid function leads to partial derivatives close to zero for earlier layers (vanishing gradient). The problem is more serious for CNNs. Instead of 3 or 4 layers, a CNN may well have 20 layers or more. Hence, the sigmoid function is used only for the last layer in binary classification problems. For all other layers, ReLU is a preferred first choice as its gradient length for positive input is always 1.

It must be stressed, however, that the choice of an appropriate activation function alone is insufficient to ensure satisfactory convergence behavior for a deep network. Several strategies and methods devoted to enhance learning performance for deep network will be discussed in the next chapters on network training.

### *8.3.2 Initial Experiments*

For reasons explained in the introductory chapter, image classification is a core problem for high-level computer vision. Numerous methods have been developed in the last 50 years to extract proper features and apply suitable classifiers to solve this problem. Consequently, many reference data sets exist (MNIST is one of them) so that researchers can compare their developments with work from others. A particularly difficult data set is the ImageNet classification challenge. It is part of ImageNet and contains about 1.2 million images in 1000 classes.

When they were presented, convolutional neural networks beat conventional classification algorithms on the ImageNet challenge by far (Krizhevsky et al., 2012). Since then, a variety of CNNs have been developed for image classification, yielding increasingly better results. We will get to know basic architectures in the following chapters. CNNs seem to be the best option for an adaptive model for

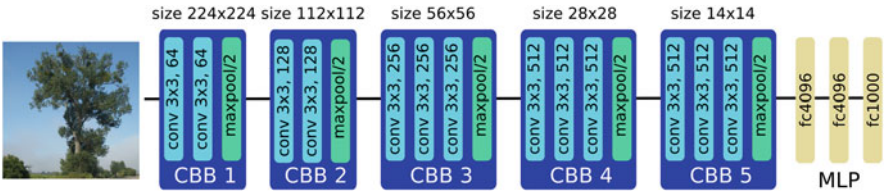



**Fig. 8.14** The VGG16 network that was presented to classify images from the 1000-class Imagenet challenge

image classification at the moment.[1] Meanwhile, a plethora of trained networks are publicly available and can be used for experiments to understand, how a network operates.

We will use the sequential VGG16 network for this purpose (see Fig. 8.14, Simonyan & Zisserman, 2014). It is no longer state of the art but it is still a strong competitor. We have chosen it as it follows the general structure of sequential networks as detailed in this chapter. VGG16 consists of 5 CBBs. The first two have two and the last three have three convolution layers. The perceptive field of all convolution layers is $3 \times 3$. Max-pooling is used for resolution reduction. The number of channels increased from 64 in the first CBB to 512 in the last CBB. The spatial resolution decreases from $224 \times 224$ to $7 \times 7$. The feature vector generated from the last CBB is fed into a 2-layer MLP with 4096 nodes in each layer and 1000 nodes for the one-hot-vector in the output layer. Counting the number of weights to be trained we see that the large majority of weights belong to the MLP part even though the convolutional part is much deeper.

Given the trained VGG16 network, a simple test is to submit an image to the network that contains an object from one of the 1000 classes and observe activations in some of the layers (see Fig. 8.15). We see that earlier layers mostly respond to differently oriented edges. It is not a surprise since a 3×3 kernel or a sequence of two of them in the first CBB has a narrow perceptive field. From all the characteristics in images that we explored in the first part of this book, edges and points are the only ones representable by such small kernels.

It is also seen that different kernels are devoted to edges with different orientations and that all representable orientations are present in the channels. Again, this is no surprise. At this level of detail it is unlikely that certain orientations should be more important than others.

For later layers a certain preference is seen for locations where the class-defining object is found. Hence, the network has learned to distinguish object detail from

[1] Recently, vision transformers have arisen as strong competitors, see Dosovitskiy et al. (2020). Their advantage is a more direct representation of relationships between components of the image. They use the transformer concept from natural language processing that commenced to replace the recurrent networks for learning dependencies between parts. It is too early for declaring them as successor to convolutional neural networks for classification but a researcher should certainly look into it.

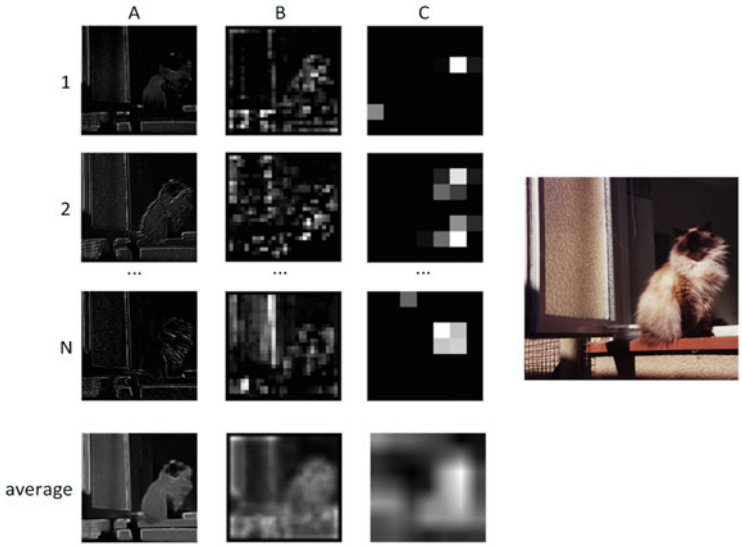



**Fig. 8.15** Comparison responses of different layers of the VGG16 network to an input image. Layer A is the output of CBB1, layer B that of CBB 3, and layer C is that of CBB 5, each time before pooling. Images are rescaled to the original size of the input image. The average over all responses is shown on the lower row. It is not used for classification but it highlights the kind of information that is processed at each layer

background. This is particularly visible if responses from channels of the last layer are averaged. Although this information is not submitted to the dense layers, it shows that the majority of nodes respond to object regions (see also Fig. 8.16). Incidentally, this is a first means to analyze the behavior of a trained network. Sometimes, when an image is misclassified, it is observed that feature extraction has failed since the network concentrates on background regions of the misclassified image instead of object regions.

In summary, these first experiments already indicate that end-to-end learning with simultaneous feature extraction and classifier training is indeed useful. It is of course not a detailed analysis. It is unclear, for instance, just what contributed to a specific decision or how features in later layers relate to image content. Such in-depth analysis requires tracking the signal from the input layer to the last layer of the convolutional part of the network. It also requires a means to generalize such tracking result. We will come back to this at the end of this book after we have learned more about the inner working of a deep convolutional neural network.


**Fig. 8.16** Depiction of the averaged activation maps of the last convolution layer before and after the final pooling step. Although this is just an average and not the transmitted features, it is seen that the highest feature values stem from regions around the object. The difference before and after max-pooling also shows the tendency of features with high value to survive the pooling step. Since pooling reduces the resolution, the high-value features now represent a larger sub-region of the image

## 8.4 Exercises

### *8.4.1 Programming Project P8.1: Inspection of a Trained Network*

In this project, we want to inspect an average over all feature maps of the last layer of a trained network. Keras provides a number of pre-trained networks trained on the ImageNet data accepting 224 × 224 pixel RGB images as input. We want to create something similar to the result depicted in Fig. 8.16 by overlaying the average activation from all feature maps on the input image.

There are several ways to extract activation maps. If it is the activation of just a single layer (as in our case), a simple means to do so is to get the trained model from the Keras repository and then copy the part up to that last feature layer. Calling `predict()` will then produce the activation of this layer. We will use the VGG16 network for this project.

The trained network is imported by.

```
from tensorflow.keras.applications.vgg16 import VGG16
# do not include the top, we do not need the MLP part
Model = VGG16(weights='imagenet',include_top=False)
```

A submodel that uses the VGG16 weights and that is cut off at layer `layer_name` (use `model.summary()` to look up the name of the last layer of the convolutional part) of VGG16 is created by

```
from tensorflow.keras.models import Model
submodel = Model([model.inputs[0]],
                 [model.get_layer(layer_name).output])
```

You can use `imread()` from `skimage` to read an arbitrary image `img` from your disk. VGG16 expects a tensor of shape `(nmb_images,224,224,3)` with RGB images. Hence, resize your image to 224 × 224 pixels (e.g., by using `resize()` from `skimage.transforms`) and rescale it to shape (1,224,224,3) using the NumPy rescaling function.

Feed the image to the truncated network. The output tensor will have the shape `(1,f1,f2,nchan)` where `f1xf2` is the resolution of the feature map and `nchan` is their number of channels. Average the activity over all channels, rescale it to the original size, and display it as an overlay with the original image. Regions that submit a lot of activity should be discernible in the image.

For creating the overlay, you can use `imshow()` from `matplotlib.pyplot`. The function has a parameter `alpha` that controls the transparency of the plotted image. Calling `imshow()` twice with two different images of the same extent (the input image and the average activation rescaled to the size of the image) and different values for the parameter `alpha` produces the overlay image when `show()` is called to render the retained depictions.

### *8.4.2 Exercise Questions*

- In what way do convolution layers restrict the use of information compared to fully connected layers? Why is this an advantage and not a disadvantage for image classification?
- Between two convolution layers, which nodes are linked together by weighted connections?
- What is the purpose of zero padding? List advantages and disadvantages of zero padding.
- What is the purpose of a pooling layer? How can it be implemented?
- What statements regarding the features generated for classification can be made from the images in slide Fig. 8.15? Give reasons for the answers.
- An optimal image for producing a label would be one that maximizes the output for this class. Look at the output of the first CBB in Fig. 8.15. Would an optimal image for class “cat” look like a cat? Justify your answer.

## References

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., & Houlsby, N. (2020). *An image is worth 16x16 words: Transformers for image recognition at scale*. arXiv preprint arXiv:2010.11929.

Fukushima, K., Miyake, S., & Ito, T. (1983). Neocognitron: A neural network model for a mechanism of visual pattern recognition. *IEEE Transactions on Systems, Man, and Cybernetics, 5*, 826–834.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems (NIPS)* (pp. 1097–1105).

Simonyan, K., & Zisserman, A. (2014). *Very deep convolutional networks for large-scale image recognition*. arXiv preprint arXiv:1409.1556.

# Chapter 9
# Network Set-Up for Image Classification


**Abstract** Given the basic topology of a deep convolutional neural network for end-to-end learning of image classification, we will discuss details to create such model for a labeling task.

Important components of a network design are the number of layers in a convolutional building block, the number of filters in each layer, the perceptive field, the kind of boundary treatment for convolution layers, and the kind of pooling. We will discuss different parameterizations for each of these hyperparameters in view of a classification task as well as strategies to gradually develop a network with a satisfactory performance. We will also take a closer look at the activation function and discuss a number of alternatives to the standard ReLU activation.

The second part of this chapter treats the data set-up. Often images come in different sizes and shapes. They need to be resampled, rescaled, and mapped to a common aspect ratio in order to be acceptable as network input.

In order to deal with the scarcity of training images in view of the large number of trainable parameters, we introduce basic data augmentation techniques to extend the number of samples by transformation of geometry and illumination of existing training samples.

Our initial experiments with a pre-trained network showed that the combination of convolutional building blocks with fully connected layers of a final multi-layer perceptron is well suited for direct end-to-end classification of pictures. In the following, we will discuss some design aspects for such a network.

## 9.1 Network Design

Even for a simple, plain vanilla convolutional neural network, a number of decisions have to be made:

- How many convolutional building blocks (CBBs) are necessary and how should they be structured?

© The Author(s), under exclusive license to Springer Nature Singapore Pte Ltd. 2024

K. D. Toennies, *An Introduction to Image Classification*,
https://doi.org/10.1007/978-981-99-7882-3_9

- How many fully connected layers are needed in the classifier part of the network?
- What is an appropriate number of channels, what kind of border treatment should be done, and how is spatial reduction in the convolutional blocks carried out?
- What kind of activation function shall be used?

We will see later that further components enhance the training abilities of the network so that even more decisions are necessary.[1] For now, we will stick to these in order to set up our own, first end-to-end classification network.

### *9.1.1 Convolution Layers in a CBB*

As for MLPs, experience tells that deeper is better when deciding on the number of layers. The total number of layers depends on the number of CBBs and the number of convolution layers in each CBB. Earlier networks had few or just a single such layer per CBB with nodes having a comparatively large perceptive field (e.g., the $11 \times 11$ perceptive field in the first layer of AlexNet). The perceptive field was large enough to capture correlations between pixel values at a given spatial resolution. Pixels that are further apart are assumed to be uncorrelated at this scale (this is the Markovian assumption often made for spatially organized data). Filters on each level learned to react on textures at this resolution.

Even though earlier networks used large perceptive fields in a CBB, the general experience is that a system of several layers with smaller perceptive fields ($3 \times 3$ or $5 \times 5$) compress the information contained in the image more efficiently. The number of channels per layer may be different but established networks such as the VGG16/VGG19 networks have the same number of channels for all layers in a CBB. It seems to work well and reflects the ability of this construction for efficient compression of input information (see Fig. 9.1 for a comparison between AlexNet from 2012 and VGG16 from 2014).

Filters with small perceptive field in a layer may not be able to represent many different textures but combining two or more such layers can do this. The main reason is that filters of every layer are fully connected to the input at the next layer (fully connected across filters, not across the feature map). If a texture feature depends on pixel correlations that extend beyond the receptive field of a filter, it may be described by a combination of less complex local aspects of it. This is efficiently representable by a multi-layer structure. The argumentation is similar to the one arguing in favor of more layers in a perceptron. Since each layer introduces its own non-linearity by its activation function, even non-linear relationships are representable.

[1] In fact, the number and diversity of hyperparameters are so large that automatic determination of suitable parameter values (including parameters about the network topology such as the number of CBBs) is a research field in its own, see, e.g., Yu and Zhu (2020) for a recent review or Dutta et al. (2018) for an example for topology optimization.

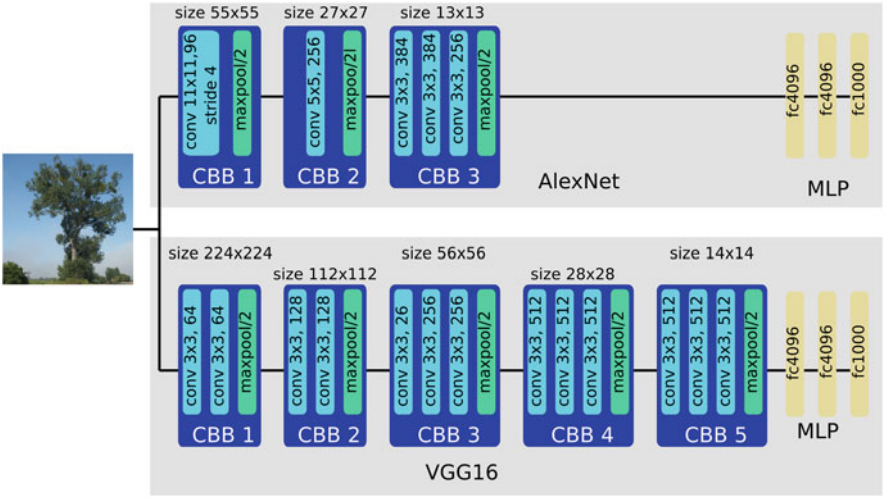



**Fig. 9.1** Comparison of the architectures of AlexNet, Krizhevsky et al. (2012), and VGG16, Simonyan and Zisserman (2014), which both won the ImageNet Large Scale Visual Recognition Challenge (in 2012 and 2014, respectively). The architecture of AlexNet, especially in the first two CBBs, was driven much more by implementing design ideas about the representation of semantics in the images than the architecture of VGG16. The more generic architecture of VGG16 seems easier to adapt to different problems as it leaves the decision on how to represent image content to network training

In principle, fewer filters per layer should be necessary in a multi-layer structure compared to a single-layer CBB with large perceptive field. It does not reduce the number of trainable weights, however, since channels are fully connected between layers. In fact, the number of weights may be even higher than for a single-layer CBB.

Channel weights are trained independently of each other. Hence, a certain amount of redundancy in the filters is to be expected, i.e., two different filters may adapt to very similar textures. Consequently, the number of filters is higher than the expected number of discernable and relevant features at this level.

When designing a network, the size of the perceptive field is often fixed and experimentation relates to the choice of the number of channels and the number of layers in a CBB. Since both are related to the represented feature diversity by a CBB, not only the number of channels but also the number of layers is smaller in the first CBBs of the network than in later CBBs.

### *9.1.2 Border Treatment in the Convolution Layer*

Classification networks mostly use “zero padding” even though the extension by pixels with value 0 creates strong artificial edges at the image boundary. Just little

extra training should suffice to recognize that these edge features are irrelevant for classification, as these borders are always at the same location and have the same shape. Hence consequences of using "no padding" are graver. Images to be classified are often the result of a detection step that singles out a small region of interest around an object. This region has often less than a few hundred pixels along each side (the image size of the CIFAR data is just $32 \times 32$, that of the ImageNet data is often reduced to $224 \times 224$, a common size of the ROI for pedestrian detection is $128 \times 64$). If "no padding" instead of "zero padding" is used, the outermost rows and columns of the data will not contribute to the features. For the smallest possible filter size of $3 \times 3$, every layer removes two rows and two columns from the input feature map. Imagine a simple network with three CBBs and two convolution layers per CBB to classify the CIFAR10 data. With "no padding," the output to the MLP would be just a $2 \times 2$ grid of channels, whereas it would be $4 \times 4$ for "zero padding." Hence, with "no padding" most of the spatial information is lost.

The situation becomes worse if larger images are classified, as they usually require more convolution layers in a CBB and a larger number of CBBs. The VGG16-net to label $224 \times 224$ images from ImageNet, for instance, has two CBBs with two convolution layers followed by three CBBs with three convolution layers each. Each CBB concludes with a $2 \times 2$ pooling. With "no padding," the final feature maps that are submitted to the MLP would consist of just a single node for each channel, whereas VGG16 with "zero padding" delivers $7 \times 7$ feature maps to the MLP.

### *9.1.3 Pooling for Classification*

Once the number of convolution layers in a building block is fixed, the next decision is how to define the resolution reduction that produces the input to the next CBB. Using a convolution layer with stride$>1$, max-pooling, average pooling, or softmax pooling each have different characteristics (see Fig. 9.2):

- Using a convolution layer with stride$>1$ is a subsampling. It has the ability to transfer high-resolution features unchanged to the next layer.
- Max-pooling is similar to the use of a convolution with stride$>1$, however with the difference that always the one feature with maximum response is submitted to the next layer.
- Average pooling filters the response prior to resolution reduction. It reduces influence from aliasing but it also reduces high-response high-detail features in the data.
- Softmax pooling is a parametrizable compromise between average pooling and max-pooling and has features of either of them depending on the specific parameter setting.

Features for classifying an image may occur at every level of resolution. Hence, max-pooling is used most often for classification networks. Keeping the exact value

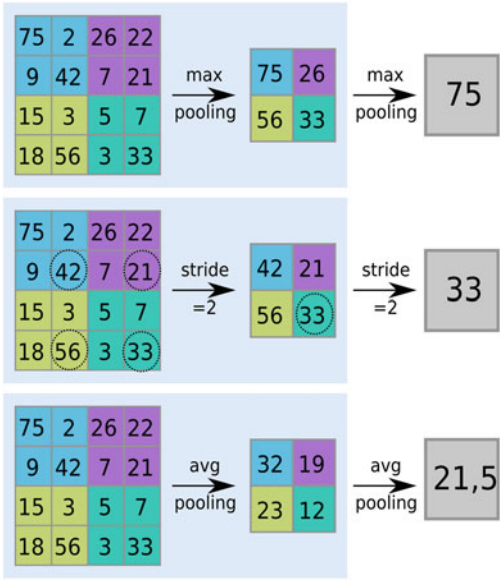



**Fig. 9.2** Max-pooling lets strong, high-resolution signal survive several pooling steps. If a convolution with stride>1 is used, this is still possible but subject to the random selection of input. In the example above, the maximum response would survive the resolution reduction, if the first instead of the second node would have been selected for signal transfer. With average pooling, the strong high-resolution signal is gradually attenuated

of the lower layer allows high-resolution features from earlier layers to be transmitted unchanged to higher layers. This may happen if the feature is important enough for classification. In such case, a channel of the last layer of the convolutional part may be exclusively or dominantly influenced by input perceived in a small region of the image. If average pooling were used instead such high-level details may be attenuated.

Similar to a convolution with stride>1 and opposed to average pooling, max-pooling may produce aliasing artifacts. It is not a concern since artifacts are likely to cancel out by the averaging effect of minibatch gradient descent during training and, if not, may be treated as irrelevant by the subsequent MLP of the network.

### *9.1.4 How Many Convolutional Building Blocks?*

The final decision in the convolutional part is about the number of convolutional building blocks. Again, bigger seems to be better. A possible reason is that each CBB extracts features from all earlier CBBs and adds new features at the current spatial resolution. Hence, a network is often designed in such a way that the final layer of its convolutional part represents image features at a low resolution (e.g., $4 \times 4$ or $7 \times 7$). The perceptive field of feature maps in this layer covers a large region at the original image resolution.

Features at this level represent low resolution characteristics in the image that depend on correlations of pixel values across large distances. High-resolution

features from earlier layers will be represented indirectly by some of the channels at the last layer of the network's convolutional part if they are relevant for classification. Especially, when using max-pooling, some channels would produce a strong signal only if a specific texture has been observed at some earlier layer.

Hence, the last layer summarizes relevant features at all resolutions. It bears some similarity to an old concept from image analysis, the linear scale space represented by a Gaussian pyramid. A Gaussian pyramid is a multiresolution representation of an image that implements the concept of a limited perceptive field separately for each resolution level. The motivation for using a multi-scale representation stems from cognitive psychology according to which humans perceive visual information at different levels of resolution in parallel. The multiresolution concept has later been extended to a non-linear scale space representation that includes an assumption about perception. It predicts that relevant scales at some image location depend on local image characteristics (realized, e.g., by anisotropic diffusion filtering). Non-linearity albeit of a different kind is included by the network topology as well, since signal transmission through resolution reduction layers may depend on non-linear operators such as convolutions with stride$>1$ or max-pooling.

As each CBB summarizes information from previous CBBs it adds to it information at the current level of resolution. It motivates the increase in the number of channels further down the network. Assuming $2 \times 2$ pooling, the amount of information from previous layers to be represented in a feature map quadruples for each new CBB. Information content may be a bit lower though as spatial reduction should filter out irrelevant information.

There are no rules for choosing the number of channels in a building block, but common practice during the development of a network is to use some computational scheme to define the number based on the channels of the first block. If this number is $b$ for building block 0, then for building block number $n$, reasonable rules are, for instance, $b \cdot 2^n$ (doubling every time), $b \cdot n$ (linear increase), or $b + n$ (adding a fixed number of channels). It is used as initial definition of numbers of channels in each layer which may be finally adapted individually for each layer during a subsequent fine-tuning phase of network development.

Since independent channel training may produce redundancy, the reduction of the number of channels in each building block may be beneficiary for efficient training. It may either be done directly by designer's choice or by introducing $1 \times 1$ convolution layers for feature reduction.

In a strictly sequential network, the number of CBBs and the number of convolution layers should not be too large in order to avoid vanishing gradients. At a first glance, this should be no problem as gradient direction rather than gradient length is important to steer the optimization process. However, smaller values for partial derivatives are more susceptible to noise or computational artifacts so that gradient directions will be less reliable.

### 9.1.5 *Inception Blocks*

There is a difference between the abovementioned Gaussian pyramid and the layered structure of convolutional building blocks. While the former may carry out image analysis at all spatial resolutions simultaneously, the latter extracts features from different spatial resolutions in different layers that are processed sequentially. High-resolution features have to pass filters in several building blocks in order to be represented as image feature in the last layer of the convolutional part of the network. This may be inefficient because of two reasons:

- A large number of convolutional building blocks will increase the problem of vanishing gradients.
- Consideration of a high-resolution feature will require that the feature survives filtration and resolution reduction in the following building blocks.

An early approach to address these two problems is the introduction of convolutions with different kernel sizes in a single convolutional building block. Such block is called an *inception block* or inception module and has first been presented by Szegedy et al. (2015). An inception module takes the output from the previous layer and passes it in parallel through convolution layers with different perceptive fields. The paper of Szegedy et al. (2015) and a follow-up publication (Szegedy et al., 2016) presented three versions of a network called GoogleLeNet that used inception modules.

An inception module always consisted of convolutions with perceptive fields of size

- $5 \times 5$
- $3 \times 3$
- $1 \times 1$.

and a $3 \times 3$ max-pooling layer with stride 1.

The three different convolutional layers are responsible to filter information from the incoming feature map at different resolutions. Max-pooling is responsible to transmit the maximum response in the incoming feature map. Hence, the inception layer extracts information from the incoming feature map in parallel at different resolutions and also allows to pass (and even emphasize) information from higher resolutions in previous layers.

Results of the four different feature extraction steps are concatenated to create an output feature map of the inception block. The main difference between the three versions of GoogleLeNet is (see also Fig. 9.3).

- Version 1 (the so-called naïve architecture) implements the concept as explained above.
- Version 2 added $1 \times 1$ convolutions for feature reduction.
- Version 3 implemented the $5 \times 5$ convolution as sequence of two $3 \times 3$ convolutions to increase the efficiency of the convolution operation.

The concept proved to be trained several times faster than a deeper purely sequential architecture while retaining the accuracy of the latter.

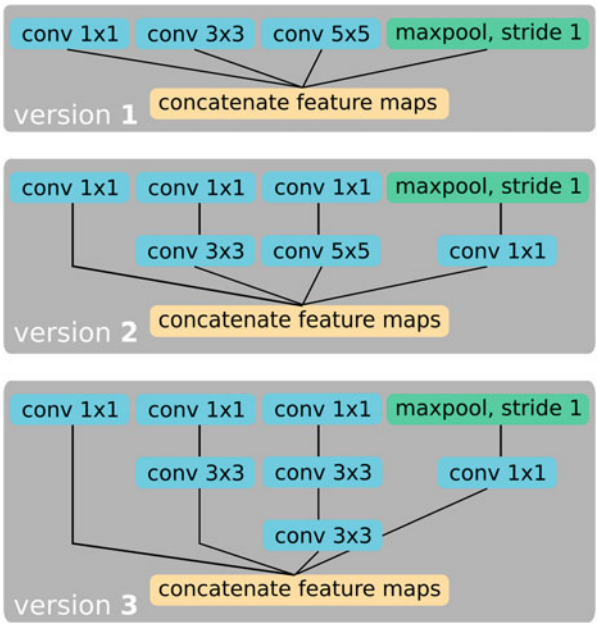



**Fig. 9.3** The first three different versions of GoogleLeNet. The second version adds $1 \times 1$ convolutions for feature reduction and the third version represents the $5 \times 5$ convolution by a sequence of two $3 \times 3$ convolutions

**Classes and Functions in Python**

A network with parallel convolution layers is no longer a sequential network. It cannot be created with the Keras class `Sequential`. However, a more general class `Model` exists in Keras that allows for a larger variety of network connections. Layer classes in `Model` use the functional interface of Python. Each layer takes a tensor as input and returns a tensor as output. For instance,

```
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, Input
features = Input(input_shape(784,))
predictions = Dense(10, activation='softmax')(features)
model = Model(inputs=features, outputs=predictions)
```

creates a tensor `features` from input with shape (784,) by calling `Input()`, then defines a layer that outputs a tensor `predictions` and finally combines this in a model. It is equivalent to

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, InputLayer
model = Sequential()
model.add(InputLayer(input_shape=(784,)))
model.add(Dense(10, activation='softmax'))
```

(continued)

An inception module such as the one in version 2 of GoogleLeNet for an input feature map `feature_in` that produces an output feature map `feature_out` can be created by the following sequence

```
from tensorflow.keras.models import Model
from tensorflow.keras.layers import (Conv2D, MaxPool2D,
                                     concatenate)
...
# 1x1 convolution with "nf1" filters
cv1 = Conv2D(nf1, (1, 1), padding='same',
             activation='relu')(feature_in)

# 1x1 reduction to "nf3_red" filters followed by a 3x3 convolution
# with "nf3" filters
cv3 = Conv2D(nf3_red, (1, 1), padding='same',
             activation='relu')(feature_in)
cv3 = Conv2D(nf3, (3, 3), padding='same',
             activation='relu')(cv3)

# 1x1 reduction to "nf5_red" filters followed by a 5x5 convolution
# with "nf5" filters
cv5 = Conv2D(nf5_red, (1, 1), padding='same',
             activation='relu')(feature_in)
cv5 = Conv2D(nf5, (5, 5), padding='same',
             activation='relu',)(cv5)

# pooling followed by feature reduction to "nfpool_red" filters
pool = MaxPool2D((3, 3), strides=(1, 1),
                 padding='same')(feature_in)
pool = Conv2D(nfpool_red, (1, 1), padding='same',
              activation='relu')(pool)

# concatenate the feature maps
feature_out = concatenate([cv1, cv3, cv5, pool], axis=3)
```

### *9.1.6 Fully Connected Layers*

The output of the last layer of the convolutional part is flattened, i.e., the channels of all nodes of the last layer are written into a single feature vector and then submitted to classification (see Fig. 9.4). The length of this vector depends on the number of channels and the number of nodes in each feature map. As the number of channels in the last CBB can be quite high (256 or 512 filters per node are quite common for a classification network) the number of values in the feature vector is often in the thousands.

We know from Chap. 7 that a MLP requires at least one hidden layer to act as a universal estimator. We also know that adding further hidden layers increases the

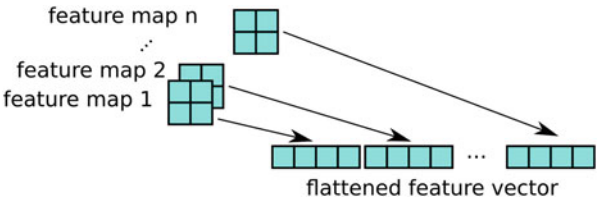



**Fig. 9.4** Flatting flattens every feature map and concatenates them to a single feature vector. Spatial and channel information is not lost since the position of a value in the feature vector indicates the feature map and its position in it

flexibility of the network to learn a suitable mapping for the linear classification in the last layer. If we now follow standard practice for constructing a MLP for $N$ input nodes, our first network design would probably consist of two hidden layers with a decreasing number of nodes and an output layer with as many nodes as classes shall be differentiated.

Given the large number of input nodes—the final convolution layer of VGG16 with $7 \times 7$ feature maps and 512 channels per node results in more than 25,000 features—it results in a rather large network (AlexNet has about 63 million, VGG16 has about 138 million trainable weights). If just 10 classes were to be distinguished from a feature vector with length 1280 after flattening and we assumed 400 and 100 nodes in the first and second hidden layer, we already have more than 500,000 weights to train in the MLP part alone (see Fig. 9.5). Fortunately, if so few classes are to be differentiated the deep convolutional part of the network can be assumed to produce almost linearly separable features. A single hidden layer with a rather small number of nodes may then be sufficient. If, in our example, we used a single hidden layer with 40 nodes, the number of weights reduces to about 50,000.

Such simplifying assumptions are often made. After all, feature extraction by the convolutional layers should produce good features as the training goal is to increase class-specific separability of samples. Still, in most deep neural networks the bulk of weights to be trained is found in the few layers of the MLP part.

### *9.1.7 The Activation Function*

If input images map to log-linear features in the last layer, the output of the next to last layer (which is sometimes called *raw output*) can be mapped to the a posteriori probability of class membership given the input information. The activation function applied to the raw output of the last layer is the softmax function (for multinomial classification) or the sigmoid function (for binomial classification).

For all other layers a non-linear activation function is chosen that alleviates the potential for vanishing gradients. The ReLU function presented in Chap. 8 fits the bill. Its advantage is a constant gradient length of one for positive input. However,

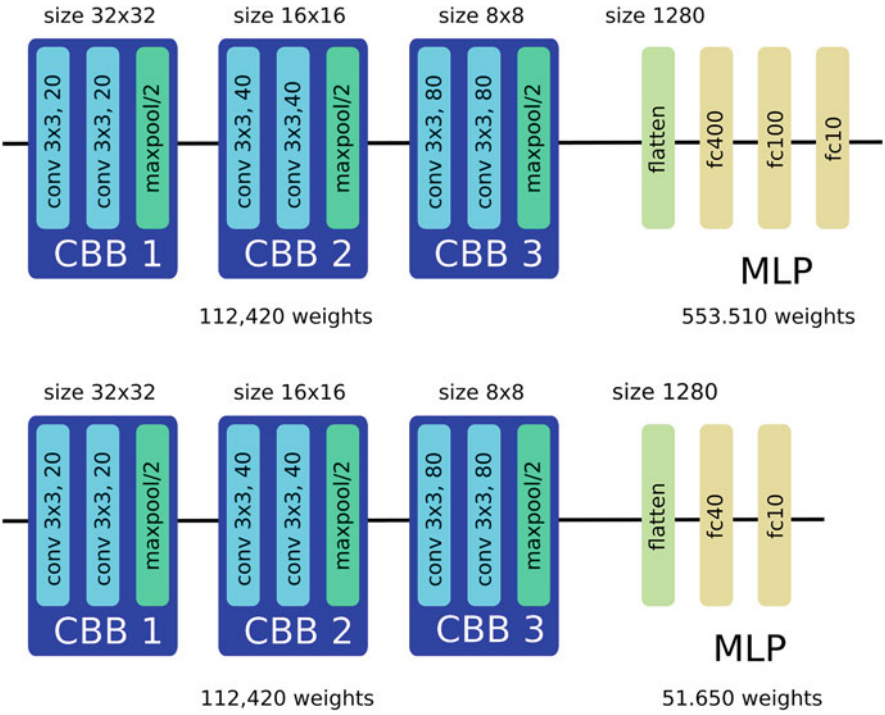



**Fig. 9.5** The majority of trainable weights is often found in the MLP part (the fully connected layers) of a CNN despite its smaller depth. It pays for a simple classification task to experiment with this part first in order to reduce the number of trainable weights (and thus make the network less susceptible to overfitting)

other activation functions exist. The search for a good activation function is still an active research field since ReLU has a number of shortcomings:

- The function is linear for all but one value. It introduces the desired non-linearity but this specific kind of non-linearity may support a peculiar distribution of samples in transformed feature space that complicates generalization from training images to unseen images.
- The gradient is large everywhere. This may lead to exploding gradients (the opposite to the vanishing gradient effect which to avoid was motivation to use ReLU).
- The function is zero for negative input with zero gradient.

The last point gave reason for developing a number of alternatives for ReLU since negative input produces a *dead neuron*. To understand what is happening let us assume that we have a simple net with one neuron per layer and ReLU activation in all neurons (see Fig. 9.6). If a neuron receives negative input, it will transmit a zero to the next layer and this neuron does not contribute to the output of the system.

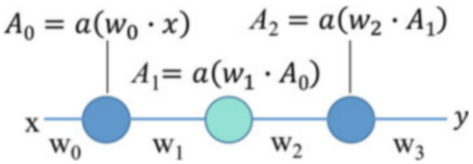



**Fig. 9.6** If activation function $a$ is the ReLU function and weight $w_1$ is negative, it will transmit a zero value to the next neuron. This will not change during optimization since gradients are computed by backpropagating the error weighted with the current weights

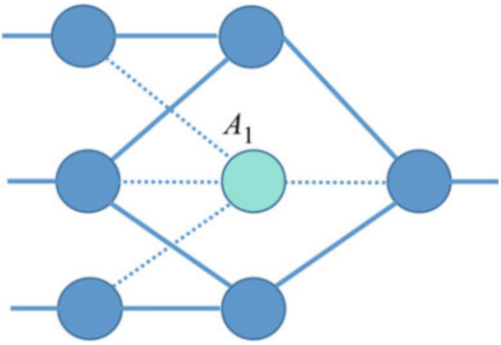



**Fig. 9.7** A dead neuron (A1) will stay dead in a more complex network than the one depicted in Fig. 9.6. If A1 transmits a zero value, weights at the dotted connections will not be updated

This will not change during optimization. Remember, that the loss from one layer is backpropagated to all nodes at the previous layer. For gradient computation, the distributed loss is weighted with the output of every node of this layer. The result is backpropagated until the input layer is reached. Hence, if a node with zero output is reached in our simple network, earlier layers will have zero-valued partial derivatives. The input to the zero-output node and all earlier nodes will not change.

The situation changes little if a layer has more nodes. Again, the output of the dead node is zero so that this node does not contribute to the final output. During backpropagation, partial derivatives of weights for connections leading to this node will be weighted with zero (see Fig. 9.7). Hence, a dead node will stay dead.

A network with dead nodes has fewer degrees of freedom as not all nodes participate in the result. It may still be acceptable considering that we argued in Sect. 9.1.4 that a layer should have enough channels to represent redundant filters. It is probably the reason why ReLU is often successful in classification networks despite this shortcoming. Still, this kind of "regularization" and its influence on the results is unpredictable. Hence, several alternatives have been suggested to replace ReLU.

An alternative is *Leaky ReLU* (LReLU, see Fig. 9.8). Instead of mapping negative values to zero, LReLU maps it to a linear function with non-zero slope

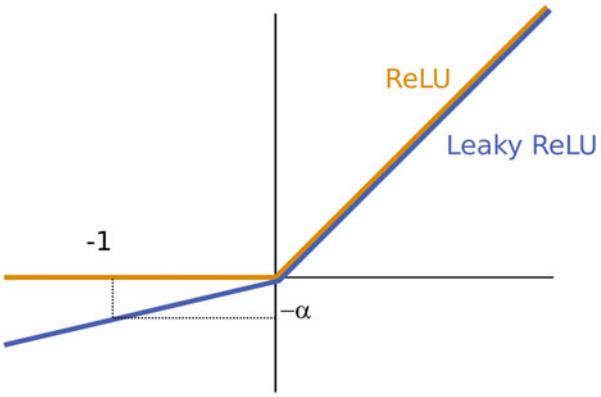



**Fig. 9.8** Comparison between ReLU and Leaky ReLU. While ReLU does not transmit a signal for negative input, PReLU transmits an inhibitory signal.

$$\text{LReLU}(x) = \begin{cases} x & \text{for } x > 0 \\ \alpha x & \text{otherwise} \end{cases} \tag{9.1}$$

with $\alpha < 1$. The slope for negative input is lower than for positive input but never zero. Although LReLU avoids the problem of dead neurons, it is less commonly used than ReLU. There are probably two reasons for this. First, LReLU introduces a new hyperparameter $\alpha$ that needs to be specified by the developer. More importantly, depending on the choice of $\alpha$ the function either approaches ReLU (for very small values for $\alpha$) or it loses its non-linearity (for values close to 1).

A variant to LReLU is parametric ReLU (PReLU). The difference to LReLU is that $\alpha$ is trained during backpropagation. It has been found to outperform ReLU if training data sets were large enough to allow for this extra parameter to be trained without causing overfitting.

Several parameterizable differentiable functions have been proposed with a similar shape than PReLU. One such function is the *swish activation* function (Ramachandran et al., 2017, see Fig. 9.9a), that has been found to improve classification results on the ImageNet data compared to the use of ReLU activation

$$\text{swish}(x) = x \cdot \text{sigmoid}(\beta x) = \frac{x}{1 + \exp(-\beta x)}. \tag{9.2}$$

The function approaches zero for $x \to -\infty$. The parameter $\beta$ controls the similarity of swish to (P)ReLU. For $\beta \to \infty$ it approaches the ReLU function and for $\beta \to 0$ it approaches the linear function $f(x) = x$. Although $\beta$ is a trainable hyperparameter, it is often just set to $\beta = 1$. In this case swish is equal to the earlier proposed *sigmoid linear unit* (SiLU).

Similar to swish is the *mish activation* function (see Fig. 9.9b):

$$\text{mish}(x) = x \tanh(\text{softplus}(x)) \tag{9.3}$$

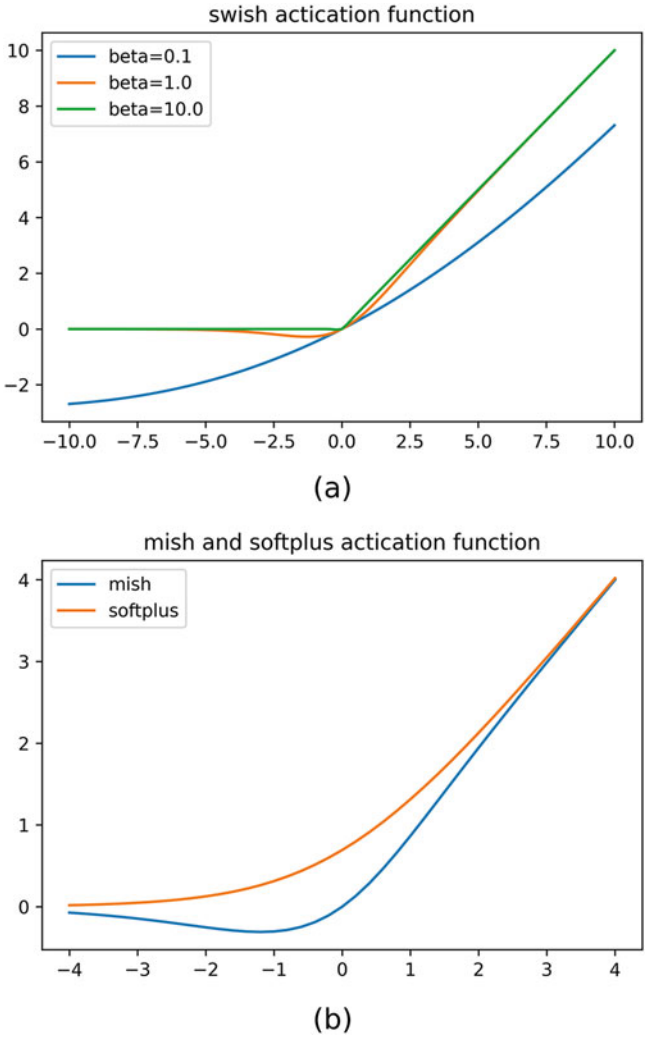



**Fig. 9.9** (**a**) Swish activation function for different parameter settings for $\beta$. (**b**) Mish activation function

where softplus$(x) = \ln(1 + e^x)$ is another smooth approximation of ReLU. The activation function has no hyperparameters, is non-linear everywhere, and was shown to outperform ReLU on the ImageNet classification data set (Misra, 2019).

The softplus function, which is part of the mish function, has been used as activation function as well. Similar to PReLU, it has the advantage of being monotonous.

These are just a few of the many activation functions that have been presented so far. They all have in common, that they behave exactly or approximately linear

delivering a derivative that is almost one for positive input. It is desired to avoid vanishing gradients as much as possible but it results in the same two shortcomings listed for ReLU above.

**Classes and Functions in Python**
Definition and adding of layers in Keras were already treated in the previous chapter. Part of it is the specification of an activation function. Besides "relu" and "softmax," that we already used earlier, Keras provides a number of further activation functions. From the ones that were mentioned in this chapter, the following can be used by submitting their string identifiers in the layer constructor:

```
sigmoid
softplus
swish
```

Use them, e.g., to add a dense layer with 10 nodes and activation `softplus`, by calling.

```
model.add(Dense(10, activation='softplus'))
```

Leaky and parameterizable ReLU activation functions need to be instantiated and then submitted to the layer constructor:

```
from tensorflow.keras.layers import Dense, LeakyReLU, PReLU

# LeakyReLU requires the alpha value to be specified
lrelu = LeakyReLU(0.1)

# PReLU takes parameters to initialize and train the alpha values as
# well as a specification whether to share alpha values across
# channels or axes of feature maps or batches; we use defaults here
prelu = PReLU()

model.add(Dense(10, activation=lrelu))
model.add(Dense(10, activation=prelu))
```

## 9.2 Data Set-Up

Training data have to be prepared in several ways. Even though end-to-end learning means that information is generated directly from images, a bit of preparation work—some of it necessary, some useful—should be carried out before presenting the data to the network.

### 9.2.1 Preparing the Training Data

Images, submitted to a classifier, are often the result of a detection step in a larger picture where rectangular regions of interest (ROI) were extracted to be labeled. Different objects may have different sizes even if they belong to the same class. The aspect ratio of their extent may be different as well (see Fig. 9.10). Very often training samples come with different spatial resolution.

The first preparation step is to ensure proper sampling so that all images have the same number of pixels per row and column. An affine transformation maps a training image to a common shape (often a square) that is then resampled to a common resolution. The affine transformation distorts the depicted object. It increases their within- and between-class variation (see Fig. 9.11) but it is still the most efficient way to deal with different sizes and aspect ratios of images. Any other strategy would have to distinguish between the different causes for changes in size and shape of depicted objects during data preprocessing.

Sampling the rescaled image changes the representation of image detail. For instance, most networks that process images from ImageNet require a resolution of $224 \times 224$ pixels even though some of the pictures have a much better resolution. However, if decisive image detail is present at a lower resolution, higher resolution input would just increase the size of the network. It unnecessarily increases the cost of network training since the system has to learn that high-resolution layer input—in this case—is irrelevant for the classification task.

When designing a new network for a certain classification task, it is worthwhile to experiment with different resolutions for network training. We want to determine the lowest resolution that still retains class-specific information from the images.

**Fig. 9.10** Images to be classified are often ROIs from object detection. ROIs will have different shapes, sizes, and resolutions

**Fig. 9.11** Rescaling and resampling produces may increase the variation within a class

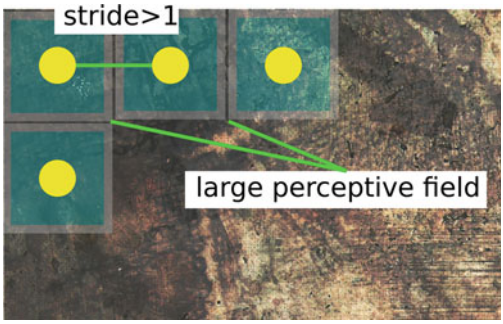



**Fig. 9.12** If the objects to be labeled contain repetitive high-resolution detail, a combination of a stride>1 with a large perceptive field captures this information while still producing a low-resolution feature map on the first layer of the network

Instead of reducing the resolution at the input layer, another option is to submit a high-resolution image as input and use a stride greater than one to reduce the resolution. The underlying assumption is that high-resolution structures are relevant to classification. Using subsampling with or without interpolation prior to submitting the image would act as low-pass filter that removes this detail. In contrast, a convolution with stride greater than one would still be able to measure such detail (see Fig. 9.12). In this case, the first convolution kernel should be not too small in order to be able to pick up such features. It may be worthwhile to define a rather large number of channels for this layer as well since high-level details may be found at different locations in the perceptive field. If necessary, features can later be reduced by a $1 \times 1$ convolution layer.

The next step of data preparation is to normalize pixel values. We already used normalization in our first experiments in Chap. 7. It is useful since learning a mapping of non-normalized feature values to normalized output values using low initial weights may result in very large gradients during the first optimization steps. It

may lead to unnecessary oscillation of the weights around the minimum instead of guiding them straight to it. For pixels, it is usually sufficient to map their values to the range [0,1] before training the network. Finally, the data is separated into training, validation, and test data.

### *9.2.2 Data Augmentation*

A deep convolutional neural network easily possesses several hundred thousand to several millions of free parameters. Training data is much scarcer and often several orders of magnitude below the number of trainable parameters. Hence, training will lead to severe overfitting.

Acquiring more data is usually not an option. However, existing training data can be used to seed the generation of new training data. This process is called *data augmentation*. Basic data augmentation produces slightly modified copies of training images and add them to the database. The modification should produce samples that are likely appearances of the same object. Hence, a newly created image by data augmentation gets the same class label than that of the image from which it was created.

Several kinds of modifications can be used, e.g.,

- *geometric modifications*: shift, scale, rotation, shear, mirroring, non-linear transformation (see Fig. 9.13),
- *modification of illumination*: contrast, color, intensity (see Fig. 9.14).

Data augmentation is specified by defining augmentation types and possibly a parameter range for each of the augmentation types (e.g., shift with a shift range of $[-10,10]$). A new sample is generated by randomly selecting parameter values within the specified range which are then used to transform an existing sample.

Data augmentation can be applied during preprocessing or during training. During preprocessing a new database is created from the original training data. Data augmentation during training applies the random transformation every time, a training sample is put into a minibatch. The former may cause problems if storage space on the computer is limited. The latter may substantially increase computation time since the transformation has to be carried out every time when a minibatch is created.

Some of the geometric transformations may cause the domain of the new image not to overlap with that of the original image (e.g., when an image is shifted). Several ways exist to fill the unknown pixel (see Fig. 9.15):

- *Constant filling* fills the image with a constant value (e.g., with zero). It does not infer information at locations where the content of the image is unknown but it creates artificial edges.
- *Nearest filling* fills the image with the closest known value in the image. It avoids artificial edges but creates structures that are unlikely to exist in reality.

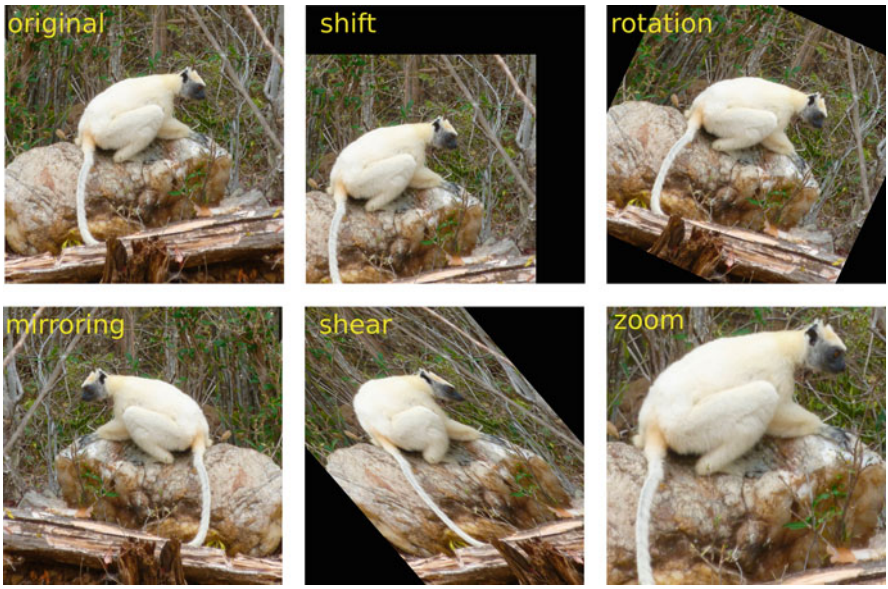



**Fig. 9.13** Different kinds of geometric transformation to create new samples by data augmentation. The probability to encounter such images in reality varies with the kind and extent of the transformation


**Fig. 9.14** Examples for changing the colors in the image. Again, some augmentations are more likely to occur in reality than others

- *Wrapping* repeats the image. It acts as if depicted objects are repeated which is unlikely but possible. It creates artificial edges at locations where the depicted object is cut by the image boundary.
- *Reflected wrapping* mirrors and repeats the image. It avoids artificial edges but the content depicted by filled pixels is less likely to exist in reality.

The kind of filling is the user's choice. Some experimentation with different methods before final network training may help to find the most appropriate filling method for a given task.

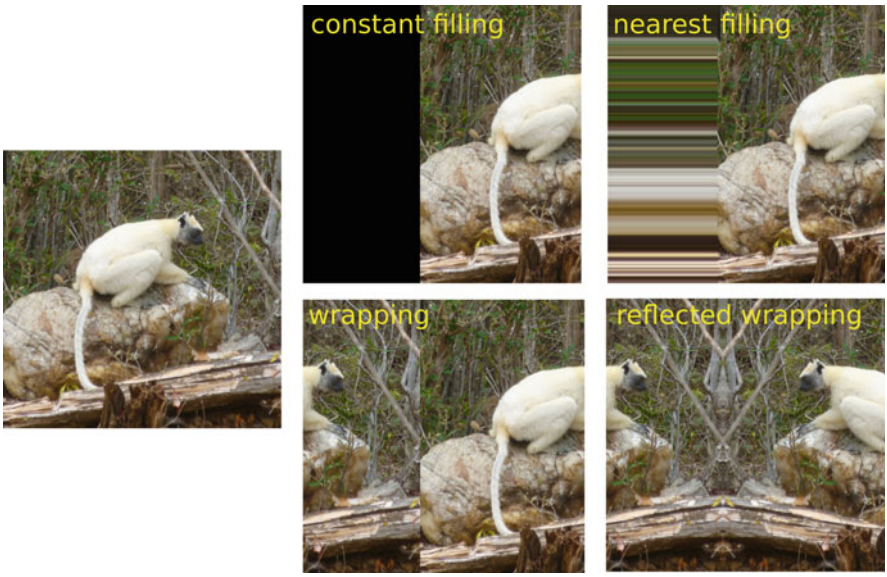



**Fig. 9.15** Different kinds of filling produce different kinds of artifacts. Most artifacts are clearly visible with an expected appearance. Reflected wrapping that produces the fewest edge artifacts creates less prominent artificial structures. It may degrade classifier performance nonetheless

Data augmentation creates arbitrary large numbers of different samples. It should be noted, however, that the set of augmented samples is biased toward the augmentation techniques used. This can lead to surprising results during training:

> Consider the case of a classification task for which only a small number of labeled data sets exist for training, validation, and test. Assume that 80% of this has been set apart for training and multiplied by data augmentation. Training and validation work quite nicely and, since augmentation created such a massive amount of data for training, the developer might assume that the system should generalize well. However, test results are disappointing.
>
> The reason is simple. Even if selection of the labeled data is assumed to be unbiased, the small number of validation samples did not reflect the unknown diversity of the unseen data good enough to generalize well. Hence, data augmentation prevented the system to overfit too rapidly to the training data but it did not help to approximate the unknown diversity in the data for predicting correct labels for the test data.

Even advanced augmentation methods that we will discuss in Chap. 11 may suffer from this problem. Variation due to data augmentation biases diversity toward the augmentation transformations whether prespecified by the developer or inferred from a model such as, e.g., Antoniou et al. (2017).

**Classes and Functions in Python**
Data augmentation at runtime can be defined if data is submitted as iterator to the network. The iterator is an object of the class `ImageDataGenerator` from `tensorflow.keras.preprocessing.image`. This class does more preprocessing than just data augmentation (rescaling the data, etc., see Keras documentation). If used for data augmentation from images read from a directory `train_directory`, it would look like this.

```
from tensorflow.keras.preprocessing.image import
                                              ImageDataGenerator
# create generators for training data with some augmentation
# (further augmentation transformations are found in the
# Keras documentation)
imagegen_train = ImageDataGenerator(width_shift_range=0.05,
                                    height_shift_range=0.05,
                                    shear_range=10.,
                                   brightness_range=[0.9,1.1],
                                    zoom_range=[0.8,1.2],
                                    vertical_flip=True,
                                    fill_mode='reflect'
                                    )
# load training data from directory (other functions of the object
# define flow of data from other sources, e.g., an array)
train = imagegen_train.flow_from_directory(train_directory,
                           class_mode ="categorical",
                           batch_size = 128,
                           target_size = (224,224))
```

The variable `train` is an iterator that can be submitted to the network to fit the model to the image data. If a second iterator `val` is defined to get validation data from another directory (of course without augmentation), the two iterators can be submitted to the `fit` function of an existing network `model` for 100 epochs by

```
model.fit(train, epochs=100, validation_data=val)
```

## 9.3 Exercises

### *9.3.1 Programming Project P9.1: End-to-End Learning to Label CIFAR10*

Create a convolutional neural network to classify the CIFAR10 data. Use the function `load_data()` from Keras (explained in project P7.1) to access the data. Use the test data for validation during fitting since we are only interested in

observing the training behavior for different network topologies (we do not want to test the classifier). Goal is to find the simplest architecture that can label the data with satisfactory accuracy (above 75% for the validation data) trained by as few epochs as possible.

Use ReLU and softmax (last layer only) as activation functions. Experiment with different network topologies by varying

- the number of CBBs,
- the number of convolution layers per CBB,
- the kernel sizes and channels in the convolution layers, and,
- the number of nodes and layers of the final multi-layer perceptron.

Do not use data augmentation. Use the Adam optimizer and `batch_size=128`. Compilation and fitting of your created network model `model` with data/label pairs `X_train, y_train` and `X_test,y_test` is then

```
# compile the sequential model, using Adam optimizer with default
# values
model.compile(loss='categorical_crossentropy',
              metrics=['accuracy'],
              optimizer='adam')
# train the model for N_EPOCH epochs
train_history = model.fit(X_train, y_train, batch_size=128,
                          epochs=N_EPOCH,
                          validation_data=(X_test, y_test))
```

So far, we have not used the return value of `fit()`. It is an object that contains a dictionary `train_history.history` with performance lists. With `<key>={'loss', 'accuracy', 'val_loss', 'val_accuracy'}`, `data=train_history.history[<key>]` returns a list with (depending on the value of `<key>`) loss/accuracy values on training/validation data for each epoch.

For fast comparison of the training performance of a given network topology, create two plots "loss" and "accuracy." The first depicts the training and validation loss and the second the training and validation accuracy. For creating the plots, functions from `matplotlib.pyplot` can be used (see Fig. 9.16 for an example):

```
plt.plot(data1) # data1: list of training data (loss/accuracy)
plt.plot(data2) # data2: list of validation data
plt.legend(str1,str2) # two strings that giving the meaning of
                      # data1 and data 2
plt.title(title_text) # title of your plot
plt_show()            # show plot
```

Try to find a network topology that avoids premature overfitting (for this simple network caused by too many trainable parameters).

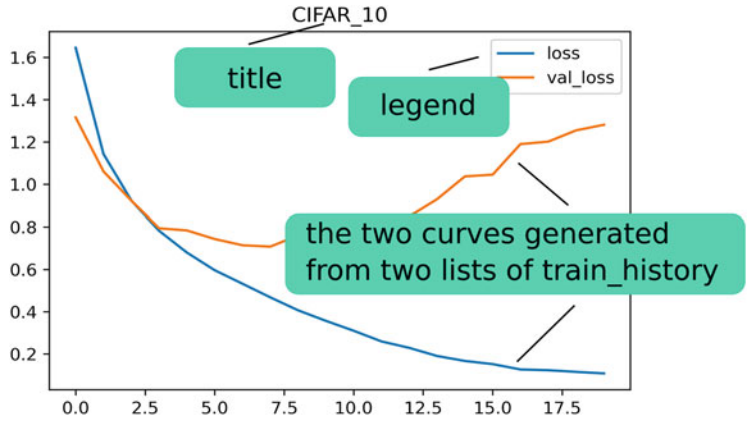



**Fig. 9.16** Example a plot of two of the lists (in this case the training and validation loss). Overfitting is immediately recognized

### 9.3.2 *Programming Project P9.2: CIFAR10 Labeling with Data Augmentation*

Extend the solution from project P9.1 to include data augmentation. Use the image data generator to create an iterator that computes data augmentation on the fly. Since the Keras function `load_data()` creates the CIFAR10 data as NumPy arrays, the iterator receives the flow from these arrays and not from a directory. The function is flow(). An example to use it with data in XX`_train`, `y_train`, `X_test`, `y_test` (in the usual form) is.

```
model.fit(datagen.flow(X_train, y_train, batch_size=128,
                       subset='training'),
          validation_data=datagen.flow(X_test, y_test),
          epochs=20)
```

Experiment with different augmentation transformations. Augmentation should defer overfitting to later epochs and decrease the loss further than training without augmentation. What augmentation transformations have the greatest impact? Is it possible to create a more complex network with better classification performance?

### 9.3.3 *Exercise Questions*

- What would be a reason to make the convolutional part of a CNN deeper rather than the fully connected part? What advantage would a deeper fully connected part have?

- Would it make sense to extend an inception module to include a $7 \times 7$ convolution? What happens if even larger perceptive fields are chosen?
- Why can dead neurons, caused from ReLU activation, even be beneficial to network training?
- Why is PReLU as activation function more susceptible to overfitting than Leaky ReLU?
- Why is it necessary to change the aspect ratio of a training image before submitting it to a classification network for training? What disadvantage has it?
- What kind of information would you expect in images for which resolution reduction for input data is done by stride$>1$ on the original resolution rather than by subsampling the image? Why should you use a larger kernel size for this first layer?
- Why can data augmentation reduce overfitting? Why does it not replace the request for more training data?
- Why does data augmentation require boundary treatment? What are the options for this and what would be the respective advantages and disadvantages?

## References

Antoniou, A., Storkey, A., & Edwards, H. (2017). *Data augmentation generative adversarial networks*. arXiv preprint arXiv:1711.04340.

Dutta, J. K., Liu, J., & Kurup, U., Shah, M. (2018). *Effective building block design for deep convolutional neural networks using search.* arXiv preprint arXiv:1801.08577.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems (NIPS)* (pp. 1097–1105).

Misra, D. (2019). *Mish: A self-regularized non-monotonic activation function*. arXiv preprint arXiv:1908.08681.

Ramachandran, P., Zoph, B., & Le, Q. V. (2017). *Searching for activation functions*. arXiv preprint arXiv:1710.05941.

Simonyan, K., & Zisserman, A. (2014). *Very deep convolutional networks for large-scale image recognition*. arXiv preprint arXiv:1409.1556.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., & Rabinovich, A. (2015). Going deeper with convolutions. In *IEEE conference on computer vision and pattern recognition* (pp. 1–9).

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., & Wojna, Z. (2016). Rethinking the inception architecture for computer vision. In *IEEE conference on computer vision and pattern recognition* (pp. 2818–2826).

Yu, T., & Zhu, H. (2020). *Hyper-parameter optimization: A review of algorithms and applications*. arXiv preprint arXiv:2003.05689.

# Chapter 10
# Basic Network Training for Image Classification


**Abstract** We review basic aspects of network training and explore the influence of different decisions on the training success.

An important hyperparameter for network training is the number of epochs during which network weights are optimized. We will discuss how early stopping weight optimization is used to avoid overfitting to the training data. Parametrization of further hyperparameters besides early stopping that we already encountered will be discussed as well. It follows a detailed discussion about basic decisions for successful network training. We cover the selection of initial weights, the choice of a suitable loss function, and the choice and parameterization of the gradient descent method. Label smoothing of the ground truth data will be discussed as well since the way in which output is generated never produces the 1- and 0-values of a one-hot-vector.

The chapter is concluded with a discussion about the interpretation of training and validation loss curves. Ideal curves would monotonically decrease until reaching a minimum. Various problems caused by network training will result in deviations from this ideal in different ways.

Because of its greater depth and the higher number of trainable weights, problems of MLP training amplify when deep networks are trained and require special treatment. While training with default parameters is a good starting point for network development, it is worthwhile to continue from there to find the best possible setting for network training of an individual network.

## 10.1 Training, Validation, and Test

The necessary amount of labeled data for a deep network is much higher because of the higher number of trainable weights. As we have seen in the previous chapter, such large database can be created by data augmentation. However, data augmentation extends the training database only. It should not be used on validation data as it would cause a network to learn selected augmentation techniques rather than

© The Author(s), under exclusive license to Springer Nature Singapore Pte Ltd. 2024

K. D. Toennies, *An Introduction to Image Classification*,
https://doi.org/10.1007/978-981-99-7882-3_10

characteristics of the true data. Test data should not be augmented as well. Otherwise, it would evaluate the generalization ability for weights and hyperparameter settings based on augmentation techniques. Hence, hyperparameter fixing using validation data will still have to use sparse data even if data augmentation is used.

Hyperparameter fixing requires a two-level process. First, network weights are optimized and then hyperparameters. The latter are often fixed by experimentation with different parameter settings. Hence, generating a trained network has an inner loop where weights are found by gradient descent and an outer loop where this process is repeated for different hyperparameter settings. We will discuss popular strategies for the process. A review that discusses settings for several types of hyperparameters and different strategies for hyperparameter optimization is Yu and Zhu (2020).

### *10.1.1 Early Stopping of Network Training*

An important hyperparameter is the number of epochs during which training continues. Remember that an epoch consists of a run through all samples that have been grouped into minibatches. The number of weight corrections in each step does not increase if data augmentation is used. In this case, an augmented exemplar is generated from each sample in the minibatch (which may also be the original sample). Optimization stops after gradient descent is carried out for a sufficient number of epochs.

Just what is meant by "sufficient number of epochs" is subject to experimentation. Training until the loss function no longer decreases is clearly not an option as this would lead to a massive overfitting to the training data. Ideally, a number of epochs should be selected such that training stops when overfitting starts. A possible criterion is a loss that still decreases on the training data but not on the independent validation data. Hence, the outer loop for finding the number of epochs consists of a test after each epoch on the validation data. Training is stopped when no improvement of the loss is observed on the validation data. This is called *early stopping* (see Fig. 10.1 for an example).

Just when to stop training is still a difficult task. Training and testing underly several influences:

- The learning rate or a complex shape of the loss function may cause oscillation around a local minimum.
- Optimization may have to pass several local minima.
- Gradient computation is an inaccurate estimate that depends on size and composition of minibatches.
- Data augmentation as well as the random shuffling to create minibatches in each epoch adds an additional stochastic component.

All this will lead to some random fluctuation of the loss curve during training as well as during validation. Since training techniques that we will discuss in this and

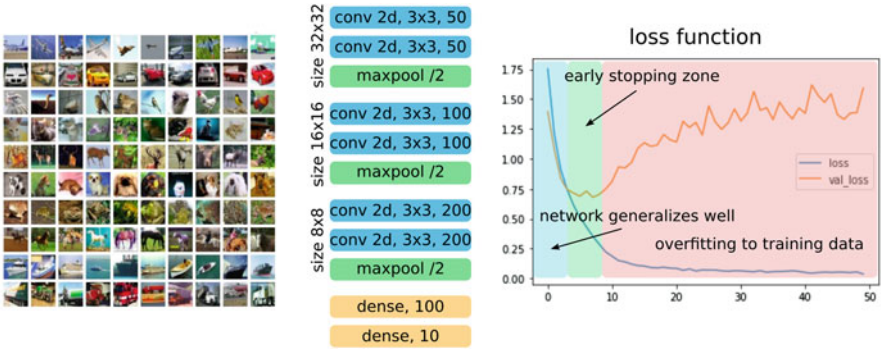



**Fig. 10.1** This is an example for early stopping for a network to classify the Cifar10 data. The loss function decreases for training and validation data for some time (blue zone). Then, the validation loss remains approximately constant for a number of epochs. This is the zone to select an epoch to stop optimization. After that, the networks overfit to the training data

the next chapter attempt to avoid overfitting as long as possible, the loss difference between training and validation data will increase slowly and may well be overlaid by a much larger variation due to influences such as the ones mentioned above.

In consequence, after a basic topology of the network (number of layers, building blocks, nodes, etc.) has been set, it is trained well beyond the epoch where the first increase between loss on training data and loss on validation data is observed. The number of epochs is then fixed after a developer's analysis of the two loss curves. Decision criteria can be, for instance, that the validation loss did not decrease for a certain number of epochs, that a smoothed validation loss curve no longer decreases, or a combination of the two.

**Classes and Functions in Python**

Keras provides a callback object for early stopping. It is submitted to the fitting function and will stop optimization when the conditions defined by the callback function are true. Parameterizable conditions include (for further parameters see the Keras documentation):

- what quantity is to be monitored (one of the logged quantities also returned in the history object),
- whether improvement of the quantity is defined as minimum or maximum of the quantity,
- for how many epochs no improvement has been observed, and
- what minimum change is necessary to indicate an improvement.

An example to monitor and act on the validation loss (with `X_train`, `y_train`, `X_val`, `y_val` as labeled training and validation data) is

(continued)

```
from tensorflow.keras.callbacks import EarlyStopping
... # import other necessary objects
... # get training and test data, define network model 'model'
# parameterize the callback function:
# – monitor validation loss (default value)
# – minimum change is 0.01
# – minimum number of epochs without improvement is 3
# – improvement is detected automatically (default value)
# – after termination restore weights corresponding to best
#   validation loss value
early_stop = EarlyStopping(monitor='val_loss', min_delta=0.01,
                           patience=3, mode='auto',
                           restore_best_weights=True)

# compile the model, with the Adam optimizer with default values
model.compile(loss='categorical_crossentropy',
              metrics=['accuracy'], optimizer='adam')

# train the model for at most 100 epochs or if early_stop
# returns True
train_history = model.fit(X_train, y_train, batch_size=128,
                          epochs=100,
                          validation_data=(X_val, y_val),
                          callbacks=[early_stop])
```

### *10.1.2 Fixing Further Hyperparameters*

A hyperparameter is anything that influences the behavior of the network. Even the network topology may be treated as configurable hyperparameter. Also, choices made to define the gradient or for carrying out the gradient descent, the choice and parameterization of the optimization algorithm, or the choice of the loss function are hyperparameters.

Some of these parameters can be integrated into the loss function (e.g., parameters of some activation functions such as the PReLU activation discussed in the previous chapter) which would delegate their optimization to the gradient descent on the training data, while others, such as the network topology, cannot. Integration into the training reduces the number of hyperparameters for which values are to be found but it increases the burden on gradient-descent-based optimization.

The number of hyperparameters that we presented so far is already large. It includes model topology hyperparameters such as:

- resolution of the input data,
- size of the perceptive field in each convolution layer (which may be different for different layers),

- border treatment for convolution layers,
- number of channels in each convolution layer (which may be different for different layers),
- number of layers per convolutional building block (which may be different for different CBBs),
- number of convolutional building blocks, and
- number of layers and nodes per layer in the MLP.

Other hyperparameters relate to the model fitting (model training) process:

- data augmentation (yes/no, and if yes, then what types and parameter ranges),
- activation functions (and possibly also their parameterization),
- gradient descent algorithm (and possibly its parameterization), and
- choice of loss function.

Further hyperparameters will be added that increase the training capacities of a given network. Many parameters are correlated in some way. Most hyperparameter settings are not found as simply as the early stopping epoch. Training needs to be repeated from scratch for every new value of a hyperparameter.

A grid search over different hyperparameter settings is possible. However, the sheer quantity of parameters prohibits such approach. Instead, developers often use default values as they may be assumed to result from extensive experimentation with similar problems. Fixing remaining hyperparameters is separated into a phase to fix the network topology and one to fix optimization parameters based on the found topology. The two phases may be repeated several times.

For finding a good network topology, a developer should first investigate existing solutions for similar problems. Possibly, the topology of this solution can be re-used or adapted to the new problem. If, for instance, a binomial classification shall be solved on regions of interest to detect pedestrians and a solution for some 10-class-problem to recognize objects in street scenes exists, taking the network topology of this solution and adapting it to the 2-class-problem is probably faster than developing a network topology from scratch.

Given an initial network topology, fixing training hyperparameters such as those in the list above will require several training runs to find appropriate settings. Parameters are usually tested independent of each other even though the assumption of independence is not always true. An example would be to search for the best activation function before experimenting with different loss functions even though the loss depends (among others) also on the activation function.

Given hyperparameter settings, experiments with adapted network topology will follow. If we started with a simple topology for the convolutional part, it is usually a good idea to experiment with the number of layers and the number of nodes in the MLP part first. Reducing the number of weights in this part as much as possible without compromising the classifier performance reduces the chance for overfitting as often the majority of trainable weights stem from the fully connected layers.

Deciding which hyperparameters to optimize is the developer's choice and depends

- on her or his experience with networks in general,
- the problem to be solved, and
- the data for which a solution is searched.

Optimizing too many hyperparameter can decrease performance, however. Since validation data is not augmented and scarce it may contain involuntary bias. Extensive hyperparameter training is just a second level training that may overfit the model to the validation data. Application to test data with all parameters fixed may then deliver disappointing results.

## 10.2 Basic Decisions for Network Training

A number of decisions have to be made for any deep network irrespective of its topology or application. These are discussed below.

### *10.2.1 Weight Initialization*

Weights and bias values in the network are initialized with small random numbers since large weights may produce large raw output values which may require large correction steps. Similar to input normalization, this kind of weight normalization reduces the chance for exploding gradients.

A random selection of weight values increases the diversity of learned filters as it lets filters contribute differently to the output. This, in turn, results in different updates for each filter. Contributions will stay different as each filter is changed to optimize its individual contribution to the output. If, instead, initial weights were all equal, the gradient descent would update all weights of a layer in the same way.

Weight values are drawn from a normal or uniform distribution. The expected value of the distribution is 0, i.e., initially a node may be excitatory or inhibitory with equal probability. Determining variance of the distribution is more difficult. If it is too high, it supports large corrections but may lead to exploding gradients. If it is too small, convergence may be slow. Variance is often specified differently for the convolutional part and the fully connected part of the network and may even be different for each layer. Variance may also depend on the number of layers and the number of nodes per layer in a network.

The definition of bias values (the $w_{0n}$ for the $n$-th layer) is less critical. Even if all bias values are just set to zero, it seems to not affect the performance of the network.

Since gradient descent does not guarantee termination, random weights initialization will guide the optimization process to a different local minimum every time the training is initialized. And, indeed, when comparing the filters of a network that is trained twice on the same data, the weights are different. This would be a significant problem for hyperparameter optimization since many hyperparameters

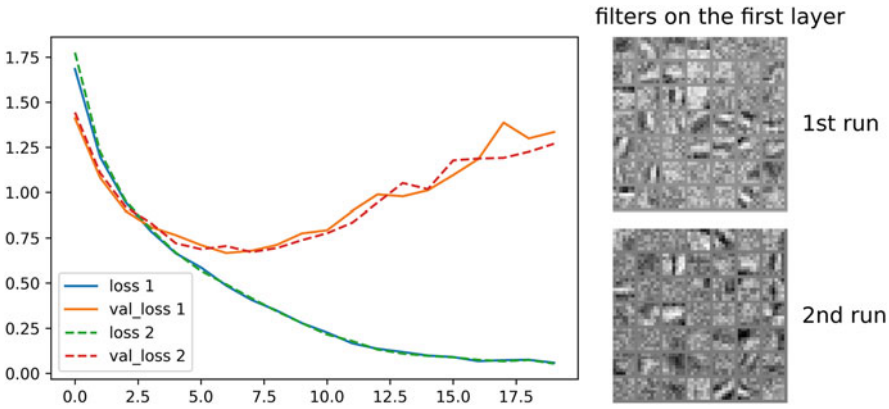



**Fig. 10.2** Two different optimization runs produce different filters which in turn results in different courses of the loss and accuracy during the epochs. However, the results are very similar

require re-training after changing its setting. However, it does not seem to have a great impact on the classifier performance. The loss from different training runs with the same network has similar values (try it!) which means that the different local minima describe the ideal classifier model equally well (see Fig. 10.2 for an example of results from different training runs).

One of the reasons for this behavior is the activation function. ReLU as well as the alternatives presented in the previous chapter are linear or almost linear over a wide range. If it were completely linear, the problem would be convex having only one minimum. Approximate linearity changes this but the local minima of the non-convex function may be similar. This can be observed experimentally when analyzing a trained network. It is possible to visualize pattern to which a filter is most sensitive (see Chap. 12 for details). Visualizations from different training runs often show that the resulting networks react on very similar patterns.

**Classes and Functions in Python**

Keras has a submodule `initializers` that contains different (random) initializers. They can be used when defining a network layer to replace the default initialization. Parameters of the initializers specify the distribution. For a normal distribution, e.g., mean and standard deviation may be submitted. Omitted parameters receive default values. If (for instance, for testing purposes) you want to initialize several times with the same weights, a parameter seed may be submitted and set always to the same value.

An example for a dense network layer that is initialized with normal-distributed 0-mean kernel weights (all weights except the bias weights) that should be the same for several runs and with zero for the bias weights is

(continued)

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.initializers import RandomNormal, Zeros
model = Sequential()

# initialize all kernel weights with normal distributed 0-mean
# values with standard deviation 0.01. If seed has the same value for
# different constructions of Dense objects, the objects will have
# the same weights.
#
# initialize bias weights with 0

model.add(Dense(100,
      kernel_initializer = RandomNormal(stddev=0.01, seed=42),
      bias_initializer  = Zeros()
          )    )
```

### *10.2.2 Loss Functions*

The goal of training is to fit the model represented to map images to labels. The optimization goal is defined by a loss function. Various loss functions exist for different network applications. For example, the loss function for identifying a bounding box around an object to be detected in an image—which is a regression task—is of a different kind than the loss function used to label a picture. For the latter, the loss function is related to finding a model that makes the highest number of correct decisions (measured, e.g., as accuracy). However, accuracy and loss are not the same and for network development, both functions should be analyzed. At best, a good gradient descent algorithm delivers a steadily declining loss. This should coincide with a steady increase in classification accuracy. If not, changing the loss function should be considered.

Customary loss functions for image classification are binary or categorical cross entropy as defined in Sect. 6.3. Examples for other loss functions used in image classification are the following:

- *Sparse categorical cross entropy* computes the loss for a ground truth that is represented by a single integer instead of a one-hot vector. It uses the same representation for predictions—a vector of label probabilities for each class—but loss is computed just for the ground truth label.
- *Weighted binary cross entropy* extends binary cross entropy by weighting the two classes differently to account for unbalanced data. It is the function of choice if the quantity to be maximized is not the accuracy but some measure that accounts for unbalanced data as well.

- *Dice coefficient* was first defined for binary problems that differentiate between a foreground and a background class. It computes the ratio of correctly labeled samples of the foreground class to all samples that are labeled as foreground. The background class is treated as uninteresting. The Dice coefficient is an appropriate loss function, e.g., for binary pixel segmentation, where each pixel is either foreground or not and just the match of correctly classified foreground pixels is of interest. The Dice coefficient is not differentiable but a differentiable approximation exists as well as extensions to more than two classes.

More loss function for image classification can be found, e.g., in Yessou et al. (2020), a general treatment of loss functions in machine learning is Wang et al. (2020). No matter what loss function is selected, a discrepancy to the desired goal (e.g., classification accuracy) should be expected. However, it is not advisable to select hyperparameters based on the goal function—which would be possible as hyperparameters are selected by experimentation rather than by gradient descent—instead of the loss function. There is no guarantee that the achieved performance extends to the classification of unseen samples.

**Classes and Functions in Python**
Keras provides several loss function objects for different purposes. For classification, binary and categorical cross entropy as well as sparse categorical cross entropy are part of this list and can be assigned by using the string identifier in the `compile()` function of a model object `model`, for instance, as

```
model.compile(loss='binary_crossentropy')
```

or

```
model.compile(loss='sparse_categorical_crossentropy')
```

In this case, the loss function objects are used with default values which is often sufficient. Alternatively, the object can be constructed and submitted to `compile()`, e.g.,

```
model.compile(loss=CategoricalCrossentropy(
                          from_logits=True))
```

which in this case tells the object that the input will not already be logits (i.e., after applying the softmax function) but the raw output of the network.

### 10.2.3 Optimizers, Learning Rate, and Minibatches

The only optimizer that we discussed in detail (in Sect. 7.3.2) is Adam. Adam belongs to a group of optimizers that adapts the learning rate automatically during optimization. Automatic adaptation relieves the user to choose learning rate decay and adaptation ranges. Why is this important? Since we use a finite learning rate, the concept of gradient descent—which requires an infinitesimal small learning rate—is just approximated.

Just what learning rate is optimal depends on the local shape of the loss function. Large learning rates are good if the loss function is locally flat around the current estimate, whereas in regions with large gradients a smaller learning rate would be preferred. Furthermore, optimization of high-dimensional and perhaps noisy loss function should result in a smooth course through feature space during optimization. It can be achieved by adding a momentum term that includes changes from previous correction steps in the current correction step.

Stochastic gradient descent (SGD) with fixed learning rates or predefined schedules to change the learning rate with and without different kinds of momentum is one way to come up with a good optimizer. SGD has been found to work well with shallow networks sometimes surpassing automatic adaptation schemes. For deep networks, various adaptive optimizers have been investigated of which Adam performed often the best (see Ruder (2016) for a review of different optimization schemes). Although this justifies our decision to use only Adam in this book, further improvements may lead to other optimizers being better suited to train deep networks.

The (initial) learning rate of the optimizer (e.g., the Adam optimizer) has a great influence on the training process. Together with the gradient length it determines the amount of change for each step during optimization. A small learning rate approximates the infinitesimally small step size of gradient descent better but it produces small correction steps as well. It increases computation time for training. It may even stop optimization early because improvements become too small for being counted as such.

Choosing a large initial learning rate will improve convergence speed but only up to a certain point beyond which convergence will deteriorate because of oscillation. Hence, experiments with the learning rate usually start with default values and then vary it in order to find the largest possible (initial) learning rate that does not cause oscillation (see Fig. 10.3).

Oscillation is recognized by observing the course of the loss values during training (the *learning curve*). Some random fluctuation can be expected from the stochastic nature of gradient estimation from minibatches but there should be at least a trend toward lower loss values. Otherwise, the learning rate is clearly too high.

Minibatches should be as large as possible as their size is the main source for random fluctuation of the loss function. For convolutional neural networks, this may become a problem. Weight and data tensors for forward and backward propagation need to be stored on the GPU for efficient computation of tensor operations. For a


**Fig. 10.3** Experiments with different learning rates for the Adam optimizer. If the learning rate is too high, the loss does not improve and the loss function oscillates. If it is too low, learning converges too slowly

fixed network topology, the number of samples in the minibatch that can be transferred to the GPU depends on the image resolution. In a large network and for images of moderate sizes (e.g., the 224 × 244 × 3 bytes of the data fed to the VGG16 network) this may quickly lead to rather small sizes of the minibatches. Instead of the several hundreds of samples from the 28 × 28 × 1 bytes MNIST data that can be processed even by small GPU we may end up with just a few samples per minibatch for larger images.

Few samples in a minibatch cause a lot of fluctuation in the loss curve. It can be only avoided if either the batch size is increased or methods are employed that smoothen the shape of the loss function. The latter will be subject of strategies presented in the next chapter. The former would either require more GPU memory, a smaller network or images with a lower resolution. It is another reason to experiment with smallest acceptable image resolution for classification.

**Classes and Functions in Python**

Minibatch sizes are specified during model fitting as we have already done several times.

For changing the optimizer, Keras provides for a selection of optimizers that can be used during compilation of the model by using the respective string identifier, e.g., for the SGD optimizer

```
model.compile(loss='categorical_crossentropy',
              metrics=['accuracy'], optimizer='sgd')
```

If parameters of the optimizer need to be changed, the optimizer object needs to be imported. The constructor of the optimizer can then be submitted to `compile()` by calling it with the parameter values. For instance, to compile the model with the Adam optimizer with an initial learning rate of 0.001:

(continued)

```
from tensorflow.keras.optimizers import Adam
model.compile(loss='categorical_crossentropy',
              metrics=['accuracy'],
              optimizer=Adam(learning_rate=0.001))
```

### 10.2.4 Label Smoothing

The ground truth for a classification task is a one-hot-vector where all entries but one are zero. The output of the network is the result of applying the softmax function to the weighted input from the last hidden layer. This function will never be zero or one. It complicates the optimization procedure. In order to reach the ground truth, the summed output from the last hidden layer would have to be either $\infty$ or $-\infty$, which will require infinitely many steps with an ever-decreasing improvement per step.

The task becomes simpler if ground truth labels are smoothed. Instead of expecting a value of 1 at the entry $c$ that corresponds to the ground truth class, a value $a < 1$ is assigned to this entry (e.g., $a = 0.8$, see Fig. 10.4). This may be interpreted as "I am $a$% sure that this sample has label $c$."[1] The remainder 1-$a$ is distributed evenly over all the other entries in the ground truth vector. Now, the values in the ground truth vector are actually achievable in the predicted output vector by a finite number of training steps.

**Classes and Functions in Python**
Label smoothing is integrated in the computation of loss functions. If, for instance, label smoothing is used with categorical cross entropy, the cross entropy object needs to be constructed with this parameterization and submitted to model compilation. An example is

```
from tensorflow.keras.losses import CategoricalCrossentropy
# re-distribute 0.1 to all non-ground-truth labels
CategoricalCrossentropy(label_smoothing=0.1)
```

and submit this instead of the string identifier "`categorical_crossentropy`" to the function `compile()`.

[1] This may even reflect reality better since labeling ground truth data is not without error.

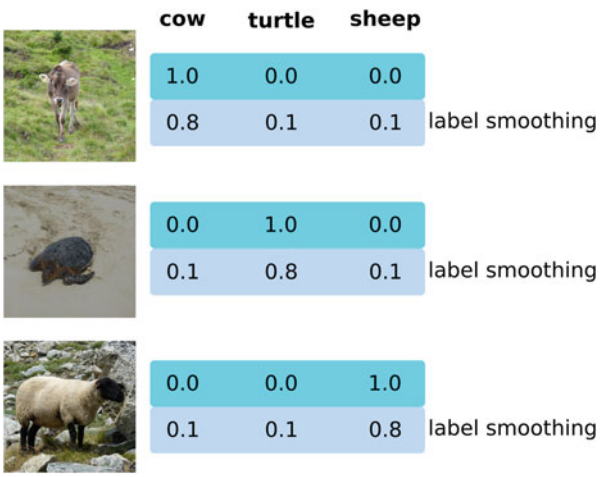



**Fig. 10.4** Label smoothing on the ground truth labels assigns a value a to the correct label for a sample ($a = 0.8$ in the example) and distributes the remainder 1-$a$ over the remaining sample classes. It simplifies the optimization and may be interpreted as uncertainty of the label assignment

## 10.3 Analyzing Loss Curves

We already saw the usefulness of tracking learning curves when we evaluated the choice of an initial learning rate or the effect of different batch sizes. Learning curves can show much more, however.

Loss curves for training and validation should monotonically decrease with little difference between the two curves. The decrease in the training curve shows that gradient descent works as expected. The similarity between the two curves shows that the model generalizes well. After stopping optimization, the loss for the validation data should be approximately the same than that for the test data.

The basic assumption for this is that all the three data sets are representative for the complete population of images of the given classification problem. Ensuring representativeness is a difficult problem in itself as we have seen before. For network development we assume that a representative set of labeled samples exists and that training, validation, and test samples are independent of each other and identically distributed (i.i.d.).

Even for i.i.d. data sets learning curves for training and validation data will not evolve in the way sketched above. They may be subject to random fluctuations, they may not decrease or even increase, and training and validation curve may differ from each other.

Some of this behavior is unavoidable. It is just a nuisance rather than an insurmountable obstacle. Some of it points at deficiencies in network design, data selection, or the training process. Hence, we will discuss frequently occurring characteristics of learning curves, potential causes, and, if necessary, remedies.

### 10.3.1 Loss After Convergence Is Still Too High

If the training curve does not decrease before a sufficiently low loss value is achieved, the network does not learn a suitable model from the data. From a certain point onward, the curve will be flat. The reason is an under-complex network.

Why does this happen? If a network is developed from scratch, the initial architecture is often simple since network training is costly. If inter- and intra-class variation is too complex to be mapped to (almost) log-linear features at the last hidden layer, training will not be able to find a function that separates the classes. We already encountered underfitting with classical machine learning method when the designed model was too simple.

The remedy is to gradually extend the network until the training loss reaches a sufficiently low value. Unless the final MLP is very simple, extension of the network first happens in the convolutional part. The aim is to compute features that are easily classified. It is preferred to extend the MLP part. Adding additional CBBs, additional layers in the CBBs, or additional channels increases the number of trainable weights less than an addition of extra fully connected layers. Hence, it is easier to avoid overfitting while still increasing the network's expressive power.

If the initial MLP is simple, e.g., just a single-layer perceptron, adding fully connected layers may be advisable as well. A single-layer perceptron assumes linearly separable features which may be too much to ask from feature computation in the convolutional part.

### 10.3.2 Loss Does not Reach a Minimum

If the losses on training and validation data are still decreasing when optimization is terminated it simply means that training stopped too early and we are still at a point on the loss surface with a non-zero gradient. It may be a characteristic of the problem. Some problems seem to have rather complex loss surfaces so that a large number of steps are necessary to reach the optimum. It may also be a problem of the learning process itself. If, for instance, the initial learning rate has been low, too many steps are needed to reach the minimum.

The remedy for this problem is simple. First, substantially increase the number of epochs to find out whether an acceptable loss value can be achieved. If this is the case, experiment with learning rates, different batch sizes, as well as with the various methods for efficient training that we will discuss in the next chapter.

It may also be the case that the network is too complex for the problem so that much of the training is spent on extracting the relevant information from the input. Reducing image resolution or removing channels, nodes, or layers from the network may help in this case.

### *10.3.3 Training and Validation Loss Deviate*

Training and validation loss are not always exactly the same since the independent validation data is not used for adapting the weights. In the first phase of training, validation loss may even be lower than training loss. It simply means that optimization has not progressed far enough for letting the model describe the training data properly. Since the amount of validation samples is usually lower than that of training samples, the current model may actually better fit the few validation samples than the many training samples.

The model gets better during network training and, apart from random fluctuations, the validation loss should be similar but a bit higher than the training loss. It is higher because label inference for unseen samples from a model cannot be better than modeling the training samples themselves.

At some point, validation loss flattens or may even increase while training loss still decreases. Overfitting of the model to the training data commences. We have already seen that early stopping may prevent overfitting.

Early stopping may not be satisfactory if the validation loss is still too high. Several strategies can be applied to avoid early overfitting. Almost all networks have a number of degrees of freedom that is magnitudes higher than the number of training samples. For a large network trained from a few samples, it is—at least in principle—possible to devote a different part of the network every single sample. Hence, training may converge in a state that no longer describes the entirety of unseen data. Making the network simpler or increasing the diversity of appearances represented by the training data are two means to avoid this.

The network can be made simpler by removing CBBs, single layers or channels from it. The diversity in the training data can be increased by adding training samples or by data augmentation (limitations of data augmentation should be kept in mind). The expected change is a training curve that decreases not as quickly with overfitting happening at a much later epoch where training and validation loss have reached lower values than before.

Another reason for deviating learning curves is an insufficient characterization of data diversity by training data. In this case, the number of training samples is much too low given the appearance variability in the images of the different classes. If it happens, training and validation learning curves may both decrease but the validation loss is much higher than the training loss. The trained model loses its ability to generalize to unseen data even before overfitting happens.

In this case, reducing the network complexity is not an option as it is not the cause. Even data augmentation may not work as it is not guaranteed that the diversity from augmentation transformations properly predicts the diversity in the unseen data. If transferring additional data from validation to training is not possible because the number of samples in the validation set is already low then acquiring new training data is often the only solution to this problem.

Sometimes, the validation loss is consistently lower than the training loss. Although seemingly unlikely it may happen when the validation data is simpler to

predict than the training data. A possible reason can be the size of the validation data set. If it contains very few samples that were selected from the complete set of labeled samples, their distribution in feature space may accidentally be simpler than the distribution of samples from the usually larger training data set. Increasing the number of samples for validation, a repeated random selection of samples from the entire labeled data set, or using cross-validation are means to deal with this.

### *10.3.4 Random Fluctuation of the Loss Curves*

Random fluctuation of the loss can make it very difficult to determine a good stopping criterion for training as it is not easy to detect the underlying trend of loss evolution. We already encountered the size of the minibatches as one possible cause for random fluctuation of the training loss. This fluctuation is reflected in the validation curve as well since validation just shows the generalization ability of the trained model given current hyperparameter settings. Hence, fluctuation in training loss will cause a validation loss that adds (minor) fluctuation of generalization errors to this training loss.

If increasing the size of the minibatches is not an option, several strategies for smoothing the loss surface have been developed of which some will be presented in the next chapter. A smoother loss surface is desirable as it reduces random fluctuation due to approximation errors of the average gradient. A smoother loss surface leads to smaller first and second derivatives. Hence, the influence of approximation errors on the weight change by a gradient descent step becomes smaller. It may also lead to less variation between single sample gradients, thus reducing approximation errors from small minibatches. Furthermore, the smooth surface will result in a smaller deviation between loss computed from minibatch gradient and that of the true batch gradient even if the former points in a different direction than the latter.

Sometimes, the training loss does not fluctuate much but the validation loss does. If no trend is observed in the validation loss curve, it points at data sets that are not identically distributed. It may happen even if basic requirements for the selection of training, validation, and test data have been observed. If, for instance, the number of labeled samples is very low, a random selection of training and validation samples may accidentally have different distribution characteristics. If data augmentation has been used, the effect may be amplified. Adding more labeled data to training and validation sets, using cross-validation, or resampling validation data from the labeled data are means to deal with this.

### *10.3.5 Discrepancy Between Validation and Test Results*

Loss on validation data after hyperparameter values have been fixed and test data should be similar, if the results generalize well to the unseen data. If test performance is substantially worse, it may have two reasons:

- overfitting of the model to the validation data
- unrepresentative test data

Overfitting may be the result from extensive hyperparameter optimization. Often, just some minor modifications of the network structure, some learning parameters, and the stopping epoch are optimized. However, the number of hyperparameters that we presented so far is already large and even more exist. Extensive experimentation or some secondary optimization process on hyperparameters may train the network in a way that "generalizes" toward the validation data. The chance of this to happen increases if just a few samples comprise the validation data set.

There is not much to be done in this case since hyperparameters have much more diverse characteristics than the weights trained in a network. It would be very difficult to optimize just those hyperparameters that do not lead to such secondary overfitting. The best solution is to increase the number of data sets used for validation and test possibly combined with a cross-validation-type strategy.

Unrepresentative test data, the second reason for differences of the loss for validation and test data, may lead to results that are substantially better or substantially worse than the performance on the validation data. Again, if test data has been randomly selected from the labeled data, it should have the same distribution characteristics than the training and validation data sets. However, a low number of samples in the data set increases the probability that the distribution characteristics of selected test data differ from that of the entirety of sampled data. In this case, increasing the size of the database is the only remedy.

## 10.4 Exercises

### *10.4.1 Programming Project P10.1: Label the Imagenette Data II*

In the programming project P6.2 an implementation of a classifier has been requested to classify the Imagenette data. Results from P6.2 serve as benchmark for the current project. Please develop a sequential neural network that reads and classifies this data. Refer to project P6.2 on where to download the data.

Training and validation images can be accessed using the Keras image data generator with the `flow_from_directory()` function of the generator object (see programming project P9.2 and Sect. 9.2.2 on how to use the image data generator in Keras).

The spatial resolution of the data is much better than that of the CIFAR10 data, whereas much fewer labeled samples exist. Hence, for extracting and labeling the data, a deeper network will be required. Data augmentation will be necessary to deal with the scarcity of labeled samples given the large number of weights to be trained in a deep network. As you know from project P6.2, Imagenette pictures all have different sizes and aspect ratios that need to be rescaled before submitting them to the network.

Normalizing the data from its input range (0...255) to (0...1) can be done by setting the parameter `rescale` when constructing the generator object. Mapping all images on a common size can be done by submitting the parameter `target_size` to the `flow_from_directory()` function. For instance, creating a generator that gets images from directory `train_directory` that normalizes the pixel values to a range (0...1) and mapping all images on a common size of 224x224 pixels is done by

```
from tensorflow.keras.preprocessing.image import (
                                        ImageDataGenerator)
# define generator
imagegen_training = ImageDataGenerator(rescale=1./255.)
# define data flow for batches of size 128
train = imagegen_training.flow_from_directory(train_directory,
                          class_mode='categorical',
                          shuffle=True,
                          batch_size=128,
                          target_size=(224,224))
```

Experiment with

- different network topologies (the general topology of VGG16 is a good starting point for this but has too many weights to be trained),
- different target sizes for the image,
- different augmentation transformations (can be integrated in the data generator, see project P9.2), and
- different learning rates, batch sizes with and without label smoothing.

The goal is to postpone overfitting as long as possible while achieving a good loss value. Since network training may take quite some time, it is important that you make a note of the intermediate results for not getting lost in all the different options.

Since it may be possible that network training needs to be carried out in several phases, you should be able to save the current state of a partially trained network after a certain number of epochs and continue with this later. This can be done by the functions `save_model()` of a model object and with the function `load_model()` in `tensorflow.keras.models`. For example, for saving a model `model` to destination `file_destination` use

```
# create 'model' and train for a number of epochs
...
# now save the model in its current state
model.save(file_destination)
```

and for loading a model from disk to a model object `model` use

```
from tensorflow.keras.models import load_model
model=load_model(file_destination)
```

The state of the model contains all parameters (weights, optimizers including state variables such as learning rate, loss function, etc.) in their current state. Hence, continuing fitting after loading the model will produce the same results as an uninterrupted training.

It may be necessary to save the state of the model after each epoch (e.g., when using resources such as the Google Colaboratory that terminates your training process if you exceed your allotted computing time). There is a callback function in `tensorflow.keras.callbacks` named `ModelCheckpoint` that you can use to do this. See the TensorFlow/Keras documentation for more details.

### *10.4.2 Exercise Questions*

- What are possible reasons for not reaching an optimal loss when an early stopping criterion is used to terminate the optimization process?
- Why are hyperparameters optimized in a heuristic, sometimes even erratic fashion? What are good strategies to avoid large computation costs?
- Why should network weights not be initialized with zero at the beginning of the training?
- What is a possible disadvantage when using sparse categorical entropy loss? What is a motivation to use it nonetheless?
- Manual schedules to change the learning rate are often monotonically decreasing with the number of iterations of gradient descent. What would happen if such scheme increases learning rate instead?
- Why may label smoothing not only lead to lower loss values but also to a higher accuracy?

## References

Ruder, S. (2016). *An overview of gradient descent optimization algorithms*. arXiv preprint arXiv:1609.047470.

Wang, Q., Ma, Y., Zhao, K., & Tian, Y. (2020). A comprehensive survey of loss functions in machine learning. *Annals of Data Science, 9*, 187–212.

Yessou, H., Sumbul, G., & Demir, B. (2020). A comparative study of deep learning loss functions for multi-label remote sensing image classification. In *IGARSS 2020-2020 IEEE international geoscience and remote sensing symposium* (pp. 1349–1352). IEEE.

Yu, T., & Zhu, H. (2020). *Hyper-parameter optimization: A review of algorithms and applications*. arXiv preprint arXiv:2003.05689.

# Chapter 11
# Dealing with Training Deficiencies


**Abstract** Basic techniques to define and use training data for network optimization may not produce the desired optimum because of the complexity of the loss function, deficiencies of the gradient descent method, and the scarcity of training data. Advanced methods and strategies deal with these problems.

First, advanced data augmentation techniques are presented to increase the robustness of the classifier. Occlusion augmentation trains the network to consider a wider range of features for classification. Adding noise makes the decision less dependent on small variation of the input data. Adversarial training does the same but for variation of features toward partitions in feature space. Generative models produce new training data from a learned distribution. Unlabeled samples may be added to training data for semi-supervised training.

Then, various methods to improve training are discussed. Transfer learning exploits existing classification problems to solve a new problem. Weight regularization as well as batch and weight normalization explore effects of normalizing weights deep in the network to arrive at a smoother loss surface. Ensemble learning uses a group of simple classifiers instead of a single complex classifier. Residual links are presented to solve the problem of a vanishing gradient in very deep networks.

Making networks deeper clearly increases capabilities to model useful characteristics based on training data. However, training suffers from a mismatch between the number of labeled samples and the degrees of freedom necessary to model such complex relationship. Substantial effort was made to alleviate problems caused by this mismatch. Several popular strategies in this regard will be discussed here. The concepts are quite diverse. Hence, we will often present just basic methodology and refer to surveys and tutorials for those of you who want to go further into the subjects.

© The Author(s), under exclusive license to Springer Nature Singapore Pte Ltd. 2024

K. D. Toennies, *An Introduction to Image Classification*,
https://doi.org/10.1007/978-981-99-7882-3_11

## 11.1 Advanced Augmentation Techniques

Training data provides a very sparse sampling of feature space. Consider, for instance, a 3-channel RGB image of size $32 \times 32$ input which results in 3072 features (the CIFAR10 data). A training database for a 10-class problem with 5000 samples per class would be considered rather generous for such images. However, it translates to roughly one sample per dimension. Hence, variation from an expected value along a dimension would be estimated by roughly two samples per dimension. Even if redundancy between pixels can be assumed so that actually a smaller subspace is spanned by the samples, the estimate of the sample distribution will not be very accurate. The sample database needs to be several orders of magnitude larger for really representing the diversity of unseen samples.

The lack of training samples has already been addressed by data augmentation in Chap. 9. The kind of augmentation presented there, however, biased the resulting samples toward very specific kinds of variation (e.g., certain types of geometric transformations). Here, we will present methods to increase the training database that do not imply such specific variation but try to make the classifier more robust in its decision.

### *11.1.1 Cutout Augmentation*

The goal of cutout augmentation (DeVries and Taylor (2017)) is to make the classifier robust to missing detail. Images are created where some parts of the image are removed (see Fig. 11.1). Often, trained classifiers seem to concentrate on a few aspects of the object to be labeled. The network will not come to the correct decision if these parts are missing even though other, probably less prominent object details are visible. Training the classifier with images, where different parts are occluded, forces the classifier to consider such less prominent details as well.

**Classes and Functions in Python**
Cutout augmentation is not contained as part of the data generator class. If you want it to use with this class, you will need to extend it to a custom class that includes this augmentation. However, for TensorFlow 2.9 or later an add-on package `keras_cv` exists that provides a number of computer vision related to building blocks. Among others, it contains advanced augmentation functions such as a random cutout layer. If the library is installed, it can be used to augment image data in an array image to add randomly distributed cutouts, e.g., like this

(continued)

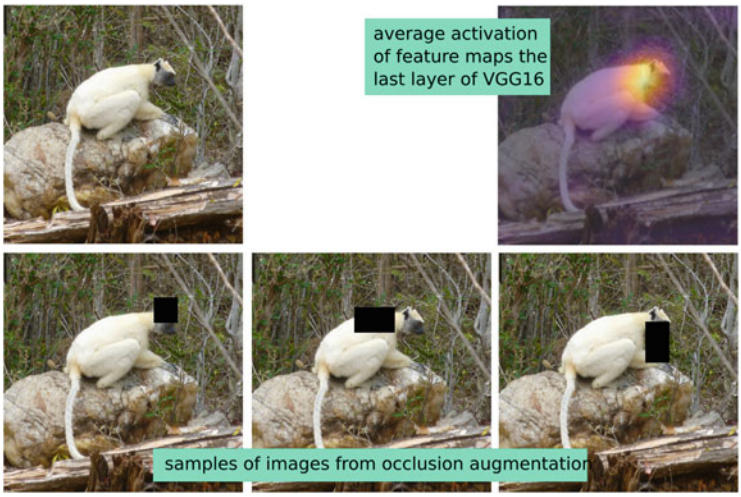



**Fig. 11.1** Cutout augmentation creates images where some part is occluded. Since many network classifiers seem to concentrate on some detail (such as the VGG16 in this example), their performance is affected if this detail is not or just partially visible. Cutout augmentation forces the network to learn correct labels even in such cases

```
from keras_cv.layer.preprocessing import RandomCutout
# define the cutout object: randomly place cutouts of which height
# and width vary randomly between [h_min,h_max] and [w_min,w_max]
# (given as percentage)
random_cutout = RandomCutout((h_min,h_max),(w_min,w_max))

# produce augmented images
augmented_images = random_cutout(images)
```

### *11.1.2 Adding Noise to the Input*

Noise is an integral part of all image acquisition. It may safely be assumed that images exist with different noise levels than those of images of the training data set. At first sight, one would assume that the added images will not contribute much to a better model. After all, the loss function averages over all samples in the training data. The distribution of training data in feature space should be robust with regard to small perturbations of the input. However, a simple example shows that this is not the case. The addition of noise often changes the output substantially (see Fig. 11.2) and may even cause misclassifications.

The reason for this amplification lies in the structural components of the network. Remember, that networks carry out a logistic regression on transformed features.

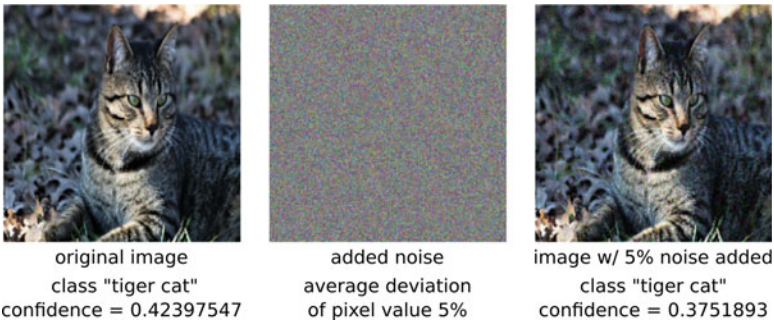



**Fig. 11.2** Adding a random variation of pixel values by about 5% reduces the confidence in the result by more than 11%. The change due to noise in the image on the right is almost invisible to a human observer (classification by a trained VGG16 network)

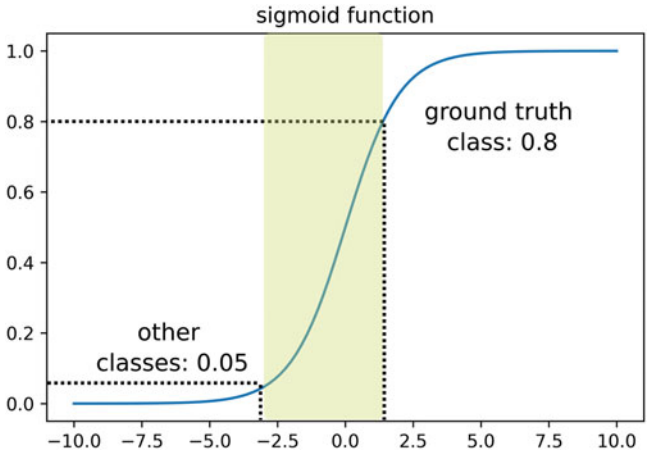



**Fig. 11.3** In this example, label smoothing for a 5-class-problem assigned a value of 0.8 for the ground truth class and values of 0.05 to all other classes to the expected output vector. This substantially reduces the range of the raw output to $[-3, +1.5]$ (instead of $[-\infty, \infty]$, if a one-hot-vector without label smoothing was used)

The raw output prior to applying softmax or sigmoid activation in the last layer is unbounded. It should be maximal for the true class and minimal for all other classes. Hence, training will attempt to let values of the raw output vector approach either $\infty$ or $-\infty$. The function that maps the bounded (and usually normalized) input vector to the raw output vector may substantially amplify the input. Label smoothing, introduced in Sect. 10.2.4, reduces this amplification of input value as it restricts the output range (see Fig. 11.3) but it cannot completely remove the effect.

The effect of adding noise under these assumptions is best understood if we assume that the mapping from input to raw output is linear. We further assume that the input is arranged as one-dimensional vector $\mathbf{x}$. Linear mapping multiplies this with a weight matrix $\mathbf{W}$ and produces the output vector $\mathbf{y}$.

$$\mathbf{y} = \mathbf{W}\mathbf{x}. \tag{11.1}$$

Let us assume that input and output dimension are equal. In this case, $\mathbf{W}$ is a square matrix. In order to map bounded, small values of $\mathbf{x}$ to very large positive or negative values in $\mathbf{y}$, the sum of eigenvalues of $\mathbf{W}$ has to be large. If a noise vector $\mathbf{n}$ is added to $\mathbf{x}$, the raw output changes to

$$\mathbf{y} + \mathbf{W}\mathbf{n} = \mathbf{W}(\mathbf{x} + \mathbf{n}). \tag{11.2}$$

Hence the noise is amplified in the same way causing the large confidence change observed in Fig. 11.2. If input and output dimensions are different, the non-square matrix $\mathbf{W}$ will have large singular values that again cause noise amplification.

The linear model sketched above is not far away from the kind of mapping in classification networks. Remember that the combination of linear matrices is again a linear matrix. Furthermore, the ReLU activation function as well as variants like swish or mish behave linear over large ranges of input values. Consequently, the advantage of being able to train deep networks has been bought at the cost of an approximate linear behavior.

Fortunately, amplification is restricted since the loss is not optimized for single samples. The loss function averages the discrepancy between network output and ground truth for all samples in the training data. It is a compromise between optimal classification of all samples. However, the low number of samples compared to the number of trainable weights still leaves the network susceptible to classification errors due to input noise.

Augmentation by adding noise alleviates the scarcity of samples. A random perturbation is added to every pixel value. A normal distribution is often used in order to create augmented images. Applying normal-distributed noise to the image enables to create arbitrarily many samples. The noise variance reflects the expected signal-to-noise ratio. Choosing higher variance values can be a reasonable choice as well since our goal is to achieve a denser representation of the underlying unknown distribution function. In this case, the addition of noise plays a similar role than kernel density estimators presented earlier: Every sample is assumed to be the expected value of a normal density function from a Gaussian mixture model and augmented images are created by drawing samples from this distribution.

Kernel density estimators have their limitations. The initial sample distribution has to be dense enough to work with sufficiently small variances as the assumption of local Gaussianity becomes less likely for a larger variance. This argumentation holds as well when adding random noise as augmentation. Hence, the selected variance should not be too high. Even it is insufficient to model the underlying true sample distribution, it will still produce better results. Under the assumption that samples of a class are on a low-dimensional manifold (a low dimensional, possibly curved subspace) in feature space, the small displacements by adding noise should result in a denser and better representation of this manifold. Training should find it easier to produce decision boundaries that generalize well to unseen samples.

**Classes and Functions in Python**
Gaussian noise can be added through an additional layer in `tensorflow.keras.layers`, e.g., for RGB images of size 224 × 224 by

```
from tensorflow.keras.layers import GaussianNoise
# constructor for model
model = Sequential(InputLayer(input_shape=(224,224, 3)))
# add Gaussian noise with variance 'noise_var' to input
model.add(GaussianNoise(noise_var, input_shape=(224,224,3)))
```

Gaussian noise will be added to all images (whether original images or augmented by other augmentation techniques)

### *11.1.3 Adversarial Attacks*

Adding noise for creating a better representation of the manifold in which samples of a given class are contained assumes that the semantics of a picture depends on all pixel values in the same way. This is generally not true. Some pixels will be more important than others for a classification. Hence, augmentation that makes the resulting classifier less sensitive to irrelevant changes of pixel values can be more effective than a broadband approach of indiscriminately adding noise to every pixel. The base of such target-oriented augmentations are *adversarial attacks* (see Akhtar and Mian (2018)) that intentionally create images that are incorrectly labeled by the classifier (see Fig. 11.4).

An adversarial attack is possible because the trained class-specific partitioning of feature space is often convoluted and does not reflect the true decision boundaries

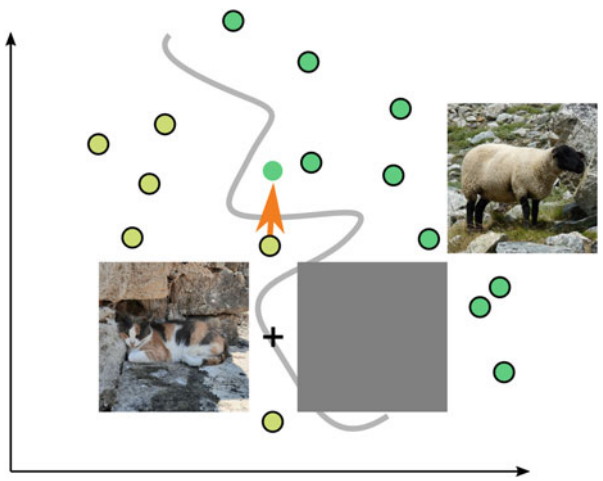


**Fig. 11.4** The strategy of an adversarial attack is to perturb the image by small changes that are invisible to a human observer. The perturbation is selected in a way to cause a change of classification. Different kinds of adversarial attacks are possible

well. Even small changes of an image may result in an incorrect class assignment. Because of the scarcity of samples, incorrect assignments may not show up in the training, validation, or test data. This makes the trained classifier susceptible to an attack that intentionally creates such images.

A disturbing aspect of this is that incorrectly classified images from an adversarial attack may look indistinguishable from correctly classified images. Apparently, the trained decision boundaries between classes are so complex that changes by fractions of an intensity level, invisible to the viewer, cause a change of class membership.

Various methods exist to create an adversary. They take advantage of the fact that the network amplifies the input for arriving at the optimal raw output as we have already seen in Sect. 11.1.2. A well-known example is the fast gradient sign method (FGSM) of Goodfellow et al. (2014). Instead of adding arbitrary noise to the image, changes are made in a way to increase the loss function. It will move the classifier away from the original decision.

The image is constructed in three steps (using the notation from Goodfellow et al. (2014)):

1. Compute the loss $J(\boldsymbol{\theta},X,y)$ for a picture. The values in $\boldsymbol{\theta}$ are the weights of the network, $X$ is the input image and $y$ is the output.
2. Compute the partial derivatives $\nabla_X J(\boldsymbol{\theta},X,y)$ of the loss function with respect to the pixels of $X$. It can be done by the same backpropagation algorithm that is used to optimize the weights. The only difference is that now the weights are fixed and the input is variable.
3. Change the image in the direction of the pixel derivatives. This is a gradient ascent on the loss function that moves the image away from the current class assignment.

In order to carry out the last step, the sign of the partial derivative is computed for every pixel and multiplied with a small quantity $\varepsilon$:

$$\tilde{X} = X + \varepsilon \cdot \text{sign}(\nabla_X J(\boldsymbol{\theta}, X, y)). \tag{11.3}$$

The difference image will have values rarely exceeding one or two gray levels. It is added to the original image.

The adversarial attack causes an invisible change that substantially changes confidence values (see Fig. 11.5 for an example with $\varepsilon = 0.5$). Why is this so? If our system were linear and had only a single output value $y$, the network would be a perceptron with weight vector $\boldsymbol{\theta}$ that is applied to the input vector $\mathbf{x}$. If noise $\mathbf{n}$ were added to $\mathbf{x}$, we have

$$\tilde{y} = \boldsymbol{\theta}^T(\mathbf{x} + \mathbf{n}) = \boldsymbol{\theta}^T\mathbf{x} + \boldsymbol{\theta}^T\mathbf{n} = y + \boldsymbol{\theta}^T\mathbf{n} = y + \varepsilon \cdot \boldsymbol{\theta}^T \text{sign}(\nabla_X J(\boldsymbol{\theta}, \mathbf{x}, y)). \tag{11.4}$$

The partial derivative in the perceptron with respect to $\mathbf{x}$ is just the weight vector $\boldsymbol{\theta}$. Hence, (11.4) simplifies to

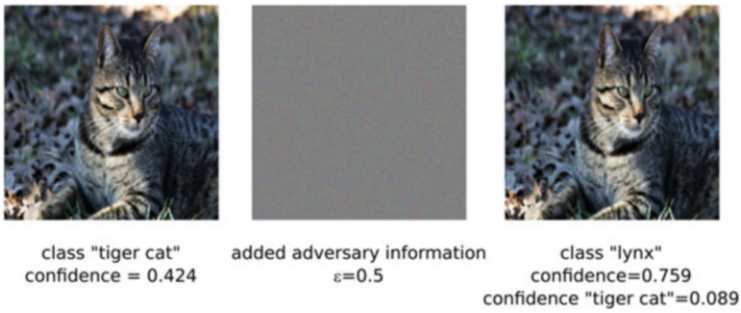



**Fig. 11.5** Adversarial attack on the image from Fig 11.2 using again the trained VGG16 network. The amount of added adversary information is less than that used for the addition in noise ($\varepsilon = 0.5$ changes pixel values by $\pm 0.5$ grey values). The change in the output vector values is much higher leading to a misclassification as class "lynx" because pixel values were intentionally changed to increase the loss

$$\tilde{y} = y + \varepsilon \cdot \boldsymbol{\theta}^T \text{sign}(\boldsymbol{\theta}\,). \tag{11.5}$$

The error is the sum of weights $\boldsymbol{\theta}$ weighted by $\varepsilon$. A change by $\varepsilon < 1$ in the input (i.e., by less than a gray value) causes a change of $\varepsilon \|\boldsymbol{\theta}\|$ at the raw output. Since the number of weights is large (as many elements as the input picture has pixels) and the regression maximizes the output, this explains the substantial change in the output. In an approximative fashion, the argumentation holds for non-linear networks as well.

### *11.1.4 Virtual Adversarial Training*

The possibility of an adversarial attack is disturbing. Being able to change class assignment by an invisible change of an image reduces the usefulness of automatic recognition that is, for instance, installed for access protection to a security area. Apart from this, it is also a general deficiency for a classifier as invisible alterations should not change an image label.

Withstanding adversarial attacks is called *virtual adversarial training*. It is an augmentation technique because it creates additional samples that, similar to the addition of noise, lead to a denser sampling of the manifold in feature space that represents an image class. Since linearity is the cause of successful adversarial attacks, the samples are created in a way to enforce non-linearity for cases where such an attack would be successful.

Virtual adversarial training is an unsupervised training technique. It forces similar images to have the same label (see Fig. 11.6). Samples $X$ are randomly selected either from an image database or from augmenting such image. The sample is fed through the network resulting in a label $\widehat{y}$. If the network is already well-trained the

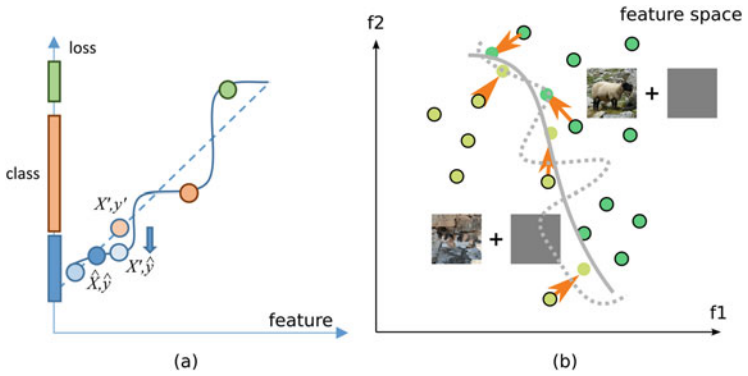



**Fig. 11.6** (**a**) Virtual adversarial training creates adversary samples and forces them to have the same label. (**b**) It improves the generalization ability of the network as images that look the same for an observer should have the same meaning (i.e., label)

label is likely to be correct. However, this does not particularly matter as the goal of virtual adversarial training is not to improve inference performance but to ensure that labeling is locally smooth in feature space.

Adversarial samples $\tilde{X} = X + \varepsilon \cdot sign[J_x(\boldsymbol{\theta}, X, y)]$ for small values of $\varepsilon$ are now created. Classification will result in labels $y'$ that are likely to be different from $\hat{y}$. Network weights are now trained to produce labels $\hat{y}$ for the adversaries as well.

Virtual adversarial training may be applied anytime during network training but it makes most sense for an already well-trained network. Such network should have almost optimal weights. Improving them to include correct labeling for the adversarial samples will "smoothen" the decision boundaries while retaining the overall inference model. In this case, it could also be called semi-supervised training as it propagates (mostly) correct labels to the constructed adversaries.

Virtual adversarial training may be applied several times in the final training phase to find the best balance between classification performance on the original training samples and smooth behavior for similarly looking samples that are erroneously assigned another class.

**Classes and Functions in Python**

(So far) virtual adversarial training is not part of Keras modules. Since creating an adversary requires access to low-level functions of TensorFlow (notably gradient computation), adversaries can be created using tensor functions directly (alternatively, the Keras backend can be used). An example for creating an adversary for an image read from disk is

(continued)

```
import numpy as np
import tensorflow as tf
IMAGE_PATH_IN = './tabby_cat.jpg' # path to input image
EPS=0.1                           # epsilon from Eq. 11.3

# load image to be attacked
img= tf.keras.preprocessing.image.load_img(IMAGE_PATH_IN,
                                   target_size=(224, 224))
# make tensor of rank 4
img= np.reshape(
      tf.keras.preprocessing.image.img_to_array(img),
      (1,224,224,3))

# load trained classification model (VGG16 in this case)
model = tf.keras.applications.vgg16.VGG16(weights='imagenet',
                                          include_top=True)

# Cast 'img' to variable (to be automatically taped by GradientTape)
tf_img = tf.Variable(tf.cast(img, tf.float32))

# 'with' is an elegant way to handle exceptions for GradientTape()
with tf.GradientTape() as tape:
    # forward propagation
    outputs = model(tf_img)
    # get label and label probability for the input image
    pred_labels = outputs[0,:]
    pred_index = (tf.argmax(pred_labels)).numpy()

    # produce one-hot vector (1000 classes of Imagenet)
    one_hot_label = tf.one_hot(pred_index, 1000)

    # compute cross entropy loss
    cce = tf.keras.losses.CategoricalCrossentropy()
    cross_entropy=cce(one_hot_label, pred_labels)

# compute gradients of cross entropy w.r.t. input image
grads = tape.gradient(cross_entropy, tf_img)

# compute adversarial image and clip it to range 0...255
tf_adv = tf.clip_by_value(tf_img + EPS*
                          tf.sign(grads[0]),0.0,255.0)
```

### *11.1.5 Data Augmentation by a Generative Model*

Adversarial training uses a single sample to generate a new sample by optimizing an output criterion assuming a certain kind of smoothness of the sample distribution in feature space. This assumption can be applied in a broader fashion by creating

samples from a generative model. The generative classification model, introduced in Sect. 4.2, is a complete representation of all probabilities of input features and class labels. Hence, feature vectors of new samples may be drawn from this distribution. In a convolutional neural network, input features are the pixels themselves. Hence, a generative network model would allow to draw arbitrary sample images from this distribution.

This augmentation has to be taken with a grain of salt. We will see that the distribution function of a generative model is estimated from labeled samples in the training database. The model generalizes this distribution but this relates to appearance features only (remember our initial discussion in Sect. 1.1 on class membership that depends on use rather than appearance). Augmentation by a generative model will work best if the training data represents the unknown distribution in features space well and the abovementioned smoothness assumption holds.

Two strategies exist to derive a generative network model from training data. Variational autoencoders (VAE) reduce and expand the input data (see, e.g., Doersch (2016) for a tutorial on VAE). The concept is based on autoencoders that encode the input by a reduced representation that can later be expanded again by the decoder. A variational autoencoder represents the reduced data by parameters of a multivariate Gaussian density function (see Fig. 11.7). New images will be generated by drawing samples from this distribution and expanding them to produce the images. Hence, the decoder acts as generator.

Generative adversarial networks (GAN) are an alternative to VAE. They are preferred for generative models applied to data augmentation because they seem to produce better results than VAE. We will sketch the concept of a GAN here but not go into detail. It is a wide field with a number of excellent tutorials (e.g., Goodfellow (2016) and Creswell et al. (2018)) and surveys (e.g., Navidan et al. (2021)) which show the many applications of different types of GAN. They also explain conceptual and practical limitations of training and representational power of a GAN.

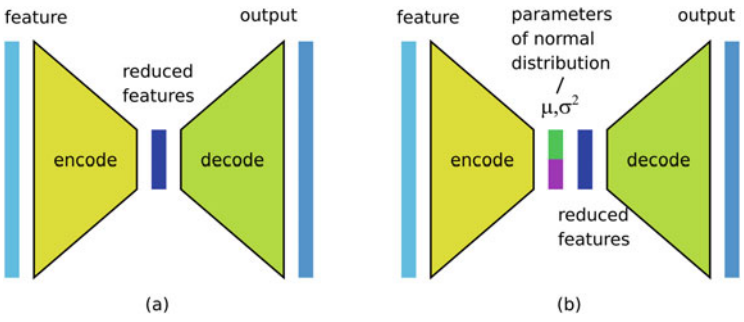



**Fig. 11.7** (**a**) An autoencoder reduces the input information (in our case, an image) to a low-dimensional representation that is then expanded to the original size. Encoder and decoder are trained such that the expanded output is as similar as possible to the input. (**b**) A variational autoencoder trains to find suitable parameters of a normal distribution instead from which the reduced features are drawn

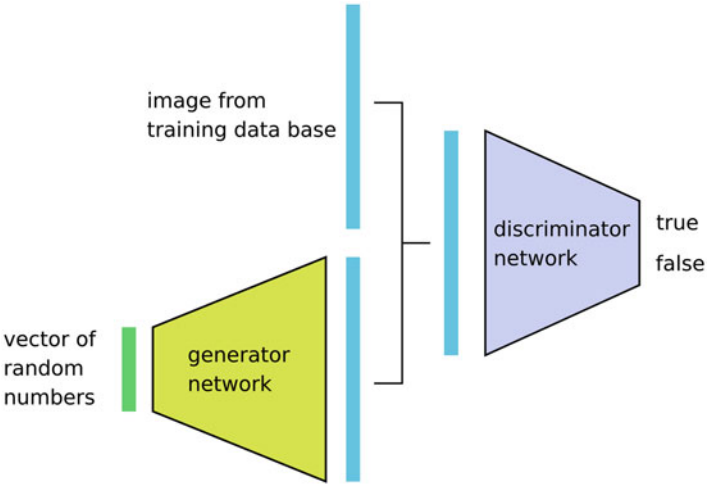



**Fig. 11.8** A simple generative adversarial model has two different networks. The generator network creates images from random input. The discriminator networks get an image of the database or one that was created by the generator as input and have to learn to differentiate between the two sources

A GAN is a combination of two opposing networks (see Fig. 11.8). A generator network takes samples from a multidimensional random number generator (often just a uniform random distribution) and maps them to images. A discriminator network receives either an image from the database or one that was created from the generator. The choice is random and unknown to the discriminator. It has to decide whether the submitted image is from the database or not (the usual analogy is that of a forger of banknotes, the role of the generator, and that of the bank which has to decide whether a banknote is genuine or not, the role of the discriminator).

The two networks have conflicting goals since each wants to beat the other network. The generator wants to produce images that are indistinguishable from image in the database and the discriminator wants to find even the smallest distinguishing feature between the two classes. Hence, a good discriminator will force the generator to produce excellent "forgeries." The ideal training of a GAN will result in what is called a Nash equilibrium, where neither of the two networks can gain through further improvement. This is far from what a gradient descent does. Hence, training of a GAN consists of keeping always one of the two networks fixed while optimizing the other. One can imagine that this kind of training is costly and not very robust.

A well-trained GAN on images for an image class or a GAN that not only predicts the image but also the label is then used to predict further training samples. The generator network alone creates new samples from random number vectors. An example for using a GAN for data augmentation is the network presented by Antoniou et al. (2017), others can be found as well.

**Classes and Functions in Python**
The discriminator network of a GAN carries out a binary classification with topology similar to classification networks already presented. Compared to this, the generator has a kind of reverse topology as it maps an input vector to an image. Hence, it consists of dense layers, followed by a reshaping from 1d to 2d, followed by a sequence of deconvolution layers. A deconvolution reverses a convolution and, for reverse strides $< 1$, interpolates to produce a feature map with higher resolution. Keras has in its submodule `layers` a class `Conv2DTranspose` that does just this.

Training iteratively adapts weights in the generator and the discriminator using two different loss functions. The discriminator minimizes the binary cross entropy, whereas the generator attempts to achieve the opposite (different loss functions for the generator have been suggested, see Goodfellow (2016)). However, training of a GAN is notoriously difficult. It is thus strongly recommended to study underlying concepts and practicalities of GANs first before attempting to create your own. Otherwise, it may be difficult to scale a GAN trained on a simple problem (for which you can find implementations on the internet) to a more complex problem. Introductory texts such as Goodfellow et al. (2014) or Creswell et al. (2018) are a good start for understanding concepts, limitations, and implementation details of GANs.

### *11.1.6 Semi-supervised Learning with Unlabeled Samples*

So far, most of the presented methods mainly increase the smoothness of sample distribution under some assumptions about the kind of smoothness. Another means to increase the training database is to include unlabeled samples. As we will see, this also increases the smoothness of the sample distribution but it takes real samples to do so. Semi-supervised learning is still a hot topic for image classification, see Jing and Tian (2020) for a recent review. Unlabeled samples are often abundant since image acquisition is just cheaper than manual labeling of images.

For learning with labeled and unlabeled samples, we assume that all images of a class, labeled or not, are distributed such that their features occupy a smooth, low-dimensional manifold of feature space. The collection of labeled and unlabeled images represents the manifold much better than the labeled images alone. If labeled and unlabeled images of a class are identically distributed, a model trained with labeled samples can predict labels of unlabeled samples. This is the basic idea for including unlabeled samples in the training.

Learning repeatedly carries out the following steps on the training data set (initially, it is the set of labeled images), see Fig. 11.9:

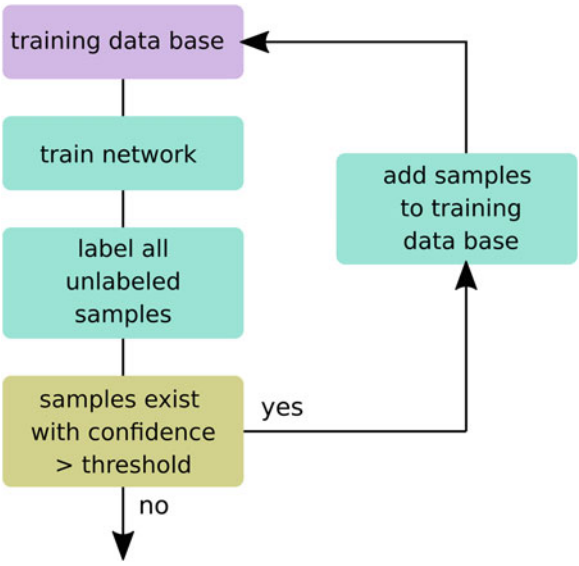



**Fig. 11.9** Semi-supervised training with unlabeled samples creates an initial model from labeled samples and repeatedly extends the training database by unlabeled samples, if their classification confidence exceeds some threshold

- Train network on labeled images.
- Apply trained network on unlabeled images.
- If the confidence of the predicted label of an unlabeled image exceeds some threshold
  - Assign the label to this image.
  - Add the now labeled image to the training database.
  - Remove the image from the set of unlabeled images.

The process terminates when no new samples have been added to the training data set in a run. Confidence is defined by the softmax activation in the last layer (e.g., assign a label, if the output is $>0.8$).

The advantage over the methods from the previous sections is a weaker constraint about the nature of the smoothness of the manifold. If the original training data characterize it well enough, the training will lead to a better model. Semi-supervised training does not increase the diversity of the training data. A sample that diverges from the currently established generalization will be an outlier that either results in an incorrect label or in one that is assigned with low confidence. The former will hurt rather than help and the latter will not change the training data.

This has been just a short introduction into semi-supervised learning with unlabeled samples. In-depth information on requirements and different techniques are found in van Engelen and Hoos (2020) or Ouali et al. (2020).

## 11.2 Improving Training

Unsatisfactory training may have a number of reasons:

- Training may converge too slowly.
- It may be subject to strong random fluctuations that makes determination of a stopping point difficult.
- Overfitting is happening long before a sufficient quality of the classification model has been reached.

We will discuss several strategies in this section that deal with this problem in order to achieve efficient and effective training.

### *11.2.1 Transfer Learning*

The simplest training is no training. If an already well-trained model exists for a classification problem, then using this model applies this strategy. Even though it is not as unlikely as it seems—after all, many models have been trained on complex classification tasks—it is not necessarily effective to re-use a model unchanged even if the classification task at hand comprises classes that are contained in one of those challenges.

The reason is that the problem at hand may be one that requires labeling of much fewer classes than those that networks have solved when taking part in these challenges. A top-1 accuracy of a state-of-the-art network of more than 90% on the 1000-class ImageNet challenge is impressive but there is no guarantee that it turns into a 99% accuracy if, e.g., just a subset of 10 classes from this challenge are to be distinguished. It is usually a better idea to take one of these networks and fine-tune it to the current problem. This is called *transfer learning*. It is a well-known concept from machine learning (see Zhuang et al. (2020) for a recent survey, the paper of Ribani and Marengoni (2019) surveys applications to deep networks). The concept profits from the knowledge and experience that is represented by a good classification network (such as the already mentioned VGG16 network or any of the newer networks that were applied to the ImageNet data).

Transfer learning takes a part of a trained model and uses it in a new trainable model. This kind of knowledge transfer is known from human cognitive psychology. Learning rarely starts from scratch but re-uses already learned concepts. Applied to machine learning, it requires to determine parts of the trained network that represent knowledge about the general problem—in our case the classification of pictures—and re-use only these. For images, this knowledge is most likely to be found in feature extraction. Features that have been trained from millions of images are likely to be relevant as well for our new classification problem (see Fig. 11.10).

In its simplest form, the complete feature computation, i.e., all of its convolutional building blocks, of a trained image classifier network such as the VGG16 network is

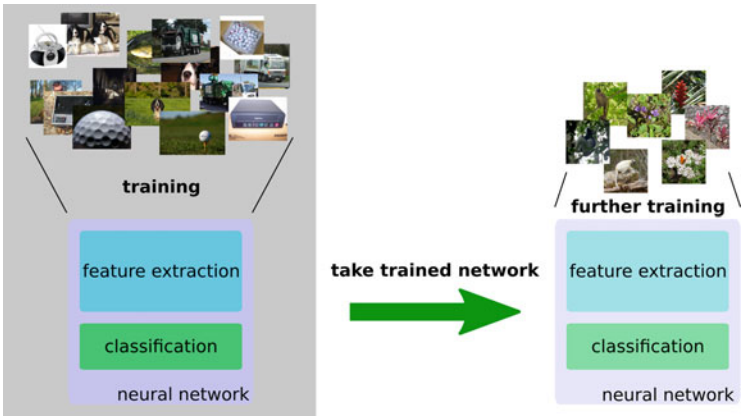



**Fig. 11.10** Transfer learning takes a trained network and re-uses it for solving a new classification problem. The extent to which the network has to be trained further depends on the difference between the problem already solved and the new problem for which the network shall be adapted

taken and then combined with a new multi-layer perceptron. If the new problem is simpler, e.g., the task to classify a subset of the ImageNet classes from fewer images per class, then the MLP may be simpler as well. For instance, instead of the two hidden layers in the VGG16 network with their 4096 nodes just a single layer with much fewer nodes is trained. It may even suffice to replace the MLP by a single-layer perceptron.

Training freezes all weights in the convolutional building blocks and optimizes weights only in the MLP. In this example, transfer learning replaces the operator-guided selection of image features in traditional image classification by training on the ImageNet data. Features found to be appropriate to distinguish 1000 very diverse classes trained from millions of images may be adequate as well for another image classification problem.

Even if true, it may be ineffective. Some features from a 1000-class-problem may be irrelevant for a more restricted problem. Furthermore, the further the domain of an image classification problem differs from that of the ImageNet data the less likely does the assumption hold. If, for instance, medical images with their much lower contrast and higher noise level and their entirely different semantics of pixel values are to be classified, ImageNet features may not be the most appropriate.

However, at least some common attributes between the image domains can be expected. After all, images are created for analysis by a human observer. Discriminating low-level features should be similar for various classification tasks. Hence, freezing weights only in the first couple of CBBs allows the later layers of the network to rearrange impact of such low-level features. Weights of the trained network may be a good starting point for the unfrozen layers.

Other kinds of knowledge transfer may be explored as well, see Tan et al. (2018) for a survey. Some examples are (see Fig. 11.11):

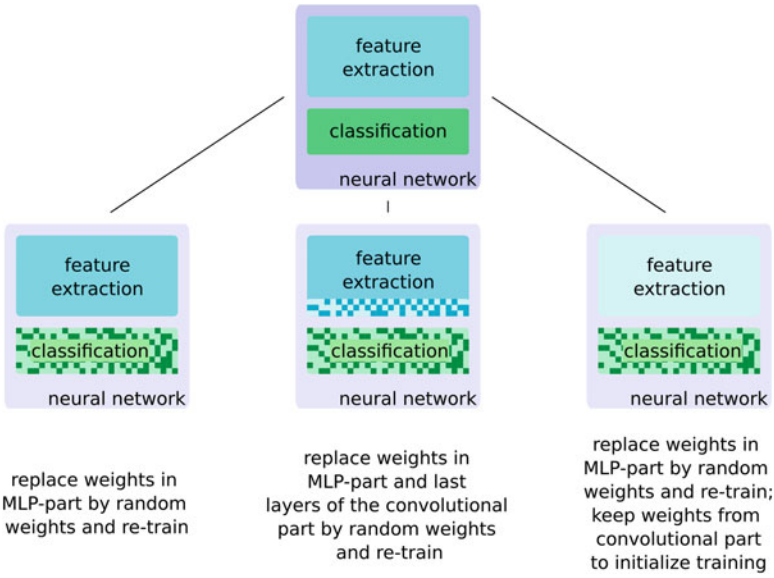



**Fig. 11.11** Different kinds of transfer learning. Some or all of the weights are frozen after copying the trained networks, while the remaining, trainable weights are initialized with random weights or the weights from the source network

- If very few or just one of the last layers of the convolutional part of the network shall be retrained, it makes sense to start with random weights instead of using previous training results. The motivation is that features on earlier levels are assumed to be relevant for the new problem but their combination as input for the MLP is not. In this case, starting with random weights is more appropriate.
- Another option is to start with results from previously trained weights but to train all weights of the convolutional part. The underlying assumption here is that the knowledge represented by the pretrained network needs just a bit of modification for being efficient and effective for the current problem.

What kind of transfer learning to use depends on the problem and on the knowledge incorporated in the previously trained network. As underlying assumptions for different kinds of transfer learning cannot be proven, it is usually a matter of experimentation to find the optimal information transfer.

**Classes and Functions in Python**

A simple way of transfer learning that replaces the classifier part of a trained network is to use two separate models. The first uses the convolutional part of a trained network and submits the output to a trainable new multi-layer

(continued)

perceptron for classification. An example to use transfer learning on two iterators `train` and `val` that provide training and validation samples is

```
from tensorflow.keras.applications import VGG16
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense, Dropout

# take VGG16 with weights trained on Imagenet without ´MLP part
pretrained_model = VGG16(include_top=False,
weights='imagenet')

print('pre-trained model summary:')
pretrained_model.summary()

# extract train and val features
vgg_features_train = pretrained_model.predict(train)
vgg_features_val  = pretrained_model.predict(val)

# one-hot encoding of labels
train_target = to_categorical(train.labels)
val_target  = to_categorical(val.labels)

# add new classification layers
model_new = Sequential()
model_new(Flatten(input_shape=(7,7,512)))
model_new.add(Dense(100, activation='relu'))
model_new.add(Dropout(0.5))
model_new.add(Dense(10, activation='softmax'))

# compile the model
model_new.compile(optimizer='adam', metrics=['accuracy'],
         loss='categorical_crossentropy')

print('new classifier on top of VGG-features:')
model_new.summary()

# train model using features generated from VGG16 model
model_new.fit(vgg_features_train, train_target, epochs=50,
       batch_size=128, validation_data=(vgg_features_val,
       val_target))
```

To freeze and unfreeze individual layers, a Boolean attribute of all layers and models called `trainable` may be set to **`True`** or **`False`**. If a new layer is instantiated, e.g., a dense layer, by

```
layer = Dense(...)
```

(continued)

The object has an attribute `layer.trainable`. In a dense layer, all weights are trainable by default, i.e.,

`layer.trainable` = **`True`**.

If weights shall be frozen in this layer, set

```
layer.trainable=False
```

If this layer is added to a model, e.g., by using the function `add()` for a sequential model `model` by

```
model.add(layer)
```

Weights in this layer will not be updated. The attribute trainable can be switched any time. Hence, many kinds of transfer learning can be realized. To access the layer of a pre-trained model and freeze it, use

```
layer=model.get_layer(<name>) # <name> is the name of the layer,
                              # use model.summary() to find names
layer.trainable= False
```

Now, you can load a pre-trained model, decide for each layer whether weights should be frozen, and—for unfrozen pre-trained layers—whether to initialize training with pre-trained weights or with random weights.

### *11.2.2 Weight Regularization*

We already encountered regularizers when discussing the optimization of a support vector machine in Sect. 6.2. Their purpose was to restrict the decision boundary to one with the best generalization properties. It is also the motivation for using regularizers in neural networks if overfitting indicates unsatisfactory generalization. The two types of regularizers (l1- and l2-regularization) from Sect. 6.2 can be used as a subtractive component to the loss function $l(\mathbf{x},\mathbf{w})$ for input $\mathbf{x}$ and weight vector $\mathbf{w}$, i.e., for l1 regularization

$$l_{L1}(\mathbf{x},\mathbf{w}) = l(\mathbf{x},\mathbf{w}) - \lambda|\mathbf{w}| \tag{11.6}$$

and for l2 regularization

$$l_{L2}(\mathbf{x}, \mathbf{w}) = l(\mathbf{x}, \mathbf{w}) - \lambda \|\mathbf{w}\|^2. \tag{11.7}$$

The parameter $\lambda$ controls the influence of the regularizer on the loss function.

The two regularizers behave differently during gradient descent. Since the derivative of the absolute values of the l1-regularizer is a constant, the same amount is subtracted from each weight. Hence, smaller weights will be reduced to zero. Since zero weights remove the influence of corresponding nodes in the network on the output result, l1-regularization will enforce a sparse model. Hence, it will be most useful for an over-complex network. This is different for the l2-regularizer. Its derivative scales with the weights. Hence, larger weights will decrease faster than smaller weights. In the result, l2-regularization will lead to an even distribution of influence of the different nodes on the mapping between input and output.

**Classes and Functions in Python**
A submodule `regularizers` exists in `tensorflow.keras` that contains regularizers. A regularizer may be added to any layer. It may be applied to the bias weights, the kernel weights, or the output activity of this layer. Regularizing the bias seems to have the least influence on the output result, so that kernel- or activity-regularization is more often used. L1- and L2-regularization may also be applied together, called L1L2. An example that exemplifies the use of the three different regularizers and the different weights in a dense (fully connected layer) is

```
from tensorflow.keras.layers import Dense
from tensorflow.keras import regularizers

# create model 'model'
...
# add layer with regularizers (parameters are the lambda values)
model.add(Dense(64,
     kernel_regularizer=regularizers.L1L2(l1=1e-5, l2=1e-4),
     bias_regularizer=regularizers.L2(1e-4),
     activity_regularizer=regularizers.L1(1e-5)
          )     )
```

### *11.2.3 Batch Normalization and Weight Normalization*

Training of the network becomes easier if the stochastic influence on the gradient descent is reduced and the gradient descent itself does not suffer from vanishing or exploding gradients. This is partially addressed by normalization of input and output values (the latter is always normalized if classification output is represented by a one-hot-vector) and by the selection of large minibatches.

The latter, however, may be impossible for a large network or large images sizes. A large size of an input image means a high resolution or an image dimension larger than 2-d (e.g., 3-d or 2d+time). Both require a more detailed feature extraction which in turn requires a greater depth of the network. In such cases, a batch consists of just a few samples in order to fit it into memory.

Small minibatches may amplify unwanted fluctuation of the optimization procedure. The reason is easily explained if a linear transformation is assumed to happen at every layer. We argued earlier why this is a reasonable approximation for our non-linear networks. Learning parameters of a linear transformation involves determination of a bias term and weights for the input. The bias term shifts the means and the weights change the variance of the features.

The shift of mean and variance is called *internal covariate shift*. It will be different for different minibatches (see Fig. 11.12). Internal covariate shift introduces a fluctuation in the weight adaptation that may lead to an instable behavior or slow convergence of the training process.

*Batch normalization* deals with this (see Fig. 11.13). For a minibatch with $m$ samples and output values $x_i$ in some layer, expected value $\widehat{\mu}$ and standard deviation $\widehat{\sigma}$ are estimated

$$\widehat{\mu} = \frac{1}{m} \sum_{i=1}^{m} x_i \tag{11.8}$$

$$\widehat{\sigma} = \sqrt{\frac{1}{m-1} \sum_{i=1}^{m} (x_i - \widehat{\mu})^2} \tag{11.9}$$

and then used to correct output values to

$$\widehat{x}_i = \frac{x_i - \mu}{\sigma + \varepsilon}. \tag{11.10}$$

The value of $\varepsilon$ is a small quantity to avoid division by zero if the standard deviation is zero.

Since shift of mean and variance may be a necessary attribute of the learning process, two trainable parameters $\gamma$ and $\beta$ are added resulting in

$$\check{x}_i = \gamma \cdot \widehat{x}_i + \beta. \tag{11.11}$$

Different to the internal shift represented by $(\widehat{\mu}, \widehat{\sigma})$ it is a trainable parameter across all samples. It should be free of variations due to the differently composed minibatches.

Batch normalization can be applied to all layers but seems to be most useful for the convolutional part. Applying batch normalization speeds up convergence and allows for larger initial learning rates without danger of exploding gradients or oscillation. However, the exact functionality of batch normalization is not well understood. It was shown by experiment, see Huber (2020), that batch normalization

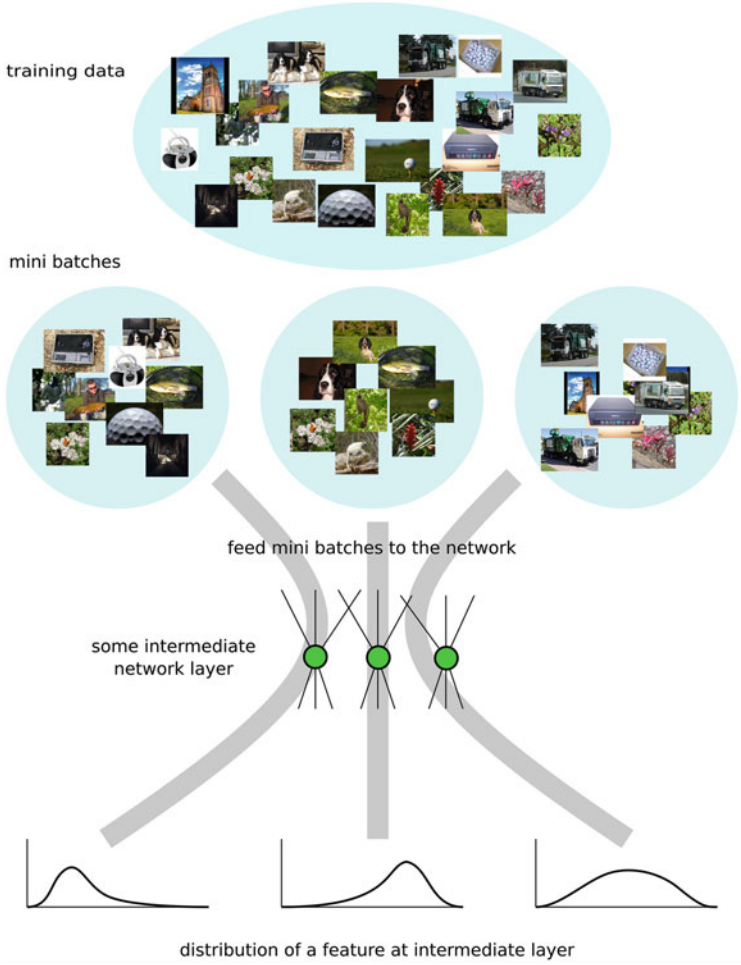



**Fig. 11.12** The internal covariate shift describes discrepancies between distributions of input values at an internal layer of a network from different minibatches

followed by adding a noisy covariate shift in the same range did not noticeably change the convergence speed. It seemed, however, that batch normalization smoothed the surface of the loss function. It would explain that higher learning rates can be used without introducing oscillation.

This smoothness of the loss function was shown in an experiment of Huber (2020) where the course of the optimization in feature space was traced. Local variability measures for each weight vector were computed during optimization. Variability measures were:

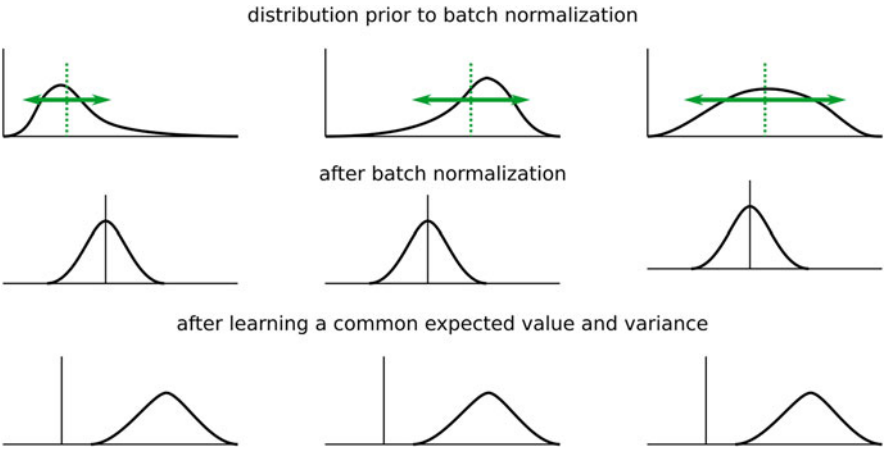



**Fig. 11.13** Batch normalization first normalizes for each minibatch before training a common expected value and variance for each minibatch

- Change of the loss
- Change of the direction of the weight vector
- Change of the length of the weight vector

The experiment showed a much higher variation for each of these measures if batch normalization was omitted. Without batch normalization, the surface of the loss function along the path traced during optimization was more convoluted.

Batch normalization requires computations across minibatches. It may be costly for a lengthy optimization procedure since optimization of a large network requires many epochs. For each epoch, a large number of minibatches have been fed through the network. It has been found that *weight normalization* can achieve similar effects than batch normalization, see Salimans and Kingma (2016). Weight normalization is based on the observation that the direction of the weight vector is more relevant than its length.

Since the length is multiplied with the learning rate, it just modifies the step size when computing the next iteration during optimization. This modification of step size can even be counterproductive as it may lead to overshoot and oscillation in regions of steep gradients. It has been suggested that, instead of normalizing the activation that gives rise to the gradient, the gradient itself shall be normalized. Gradient normalization is faster than batch normalization and makes optimization independent of the local curvature of the loss surface. It has been shown to speed up convergence just as batch normalization although it does not address variation from gradient estimates by different minibatches.

**Classes and Functions in Python**
Batch normalization is a layer that can be added to the network. The following sequence, for instance

```
from tensorflow.keras.layers import (Conv2D,MaxPool2D,
                                     BatchNormalization)
model.add(Conv2D(50, kernel_size=(3,3), strides=(1,1),
                 padding='same',
                 activation='relu', name='conv01')
model.add(Conv2D(50, kernel_size=(3,3), strides=(1,1),
                 padding='same',
                 activation='relu', name='conv02')
model.add(MaxPool2D(pool_size=(2,2)))
model.add(BatchNormalization())
```

adds two convolutional and one pooling layer to a network `model` before adding a batch normalization layer that carries out the batch normalization on the output after pooling.

### *11.2.4 Ensemble Learning and Dropout*

Instead of improving training in a complex network, another strategy to improve network learning is to use network ensembles. It is motivated by results from traditional image classification where a combination of several simple classifiers has been shown to be as effective as a single, more powerful classifier. Its additional advantage is that overfitting is less likely to happen because of the simpler structure of the base classifiers.

It works for neural networks as well. Instead of a single network, an ensemble of several base networks classifies the samples (see Fig. 11.14). Each base network is simpler in terms of number of layers, number of nodes, or number of node connections. Bagging, i.e., the voting from several independent classifiers, and boosting, i.e., a sequential training of networks where each new classifier improves deficiencies of earlier classifiers, may both be used.

The base classifiers mostly differ in the fully connected part responsible for classification. Given a deep network topology, the convolutional part is combined with a simple MLP with a single or no hidden layer. Training this network with different subsets of the data results in different classifiers just as in the bagging and boosting methods discussed in Sect. 6.4. More diversity among the classifier can also be created by selecting base classifiers with different topology in the fully connected part.

Ensemble learning is time-consuming since several networks need to be trained. Bagging can be done in parallel if sufficient memory and processing power are

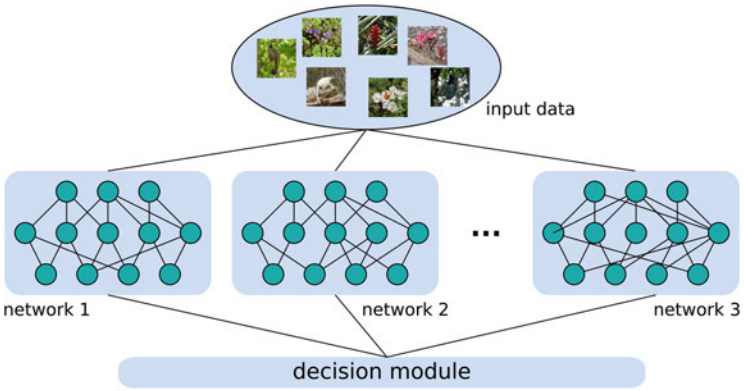



**Fig. 11.14** Ensemble learning with networks implements a bagging strategy. It classifies a sample by an ensemble of networks. The decision module gathers the different decisions and creates an aggregate decision (e.g., by letting each network vote for a decision)

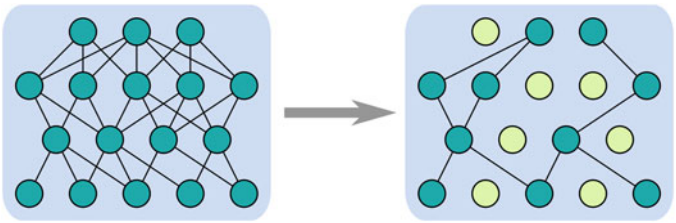


**Fig. 11.15** Learning with dropout excludes a certain number of network nodes from weight correction. Edges connected to these nodes are not updated. Information for gradient computation is not flowing along these edges

available. Boosting, where a new classifier builds on characteristics of previously trained classifiers, inherently requires sequential training.

A technique that is similar to bagging avoids all this. Instead of building several networks, a single network is used but just a part of it is trained at every epoch. The strategy is called *dropout technique*. For every gradient descent step, a certain number of nodes do not take part in the optimization, hence they drop out of the process (see Fig. 11.15). The ratio is called the *dropout rate*. It is another hyperparameter of the network and may be defined separately for every layer.

Training with dropout randomly selects nodes at the specified rate for every optimization step. These nodes will neither contribute to the result in the feedforward step nor will they influence gradient computation and weight correction in backpropagation. Hence, for every optimization step, training enforces a sparse model to map images to labels. Since fewer nodes are trained, it will be less susceptible to overfitting. A tendency to underfitting is counteracted by selecting a different subset of dropout nodes for every optimization step.

Training with dropout trains many different sparse networks at the same time. It has been shown that dropout indeed reduces overfitting so that better loss values in the validation data can be achieved, see Srivastava et al. (2014). Under certain conditions the introduction of dropout into network training is equivalent to the training of several independent networks followed by letting these networks vote on the final result (see Chap. 7 in Goodfellow et al. (2016). Hence, it is an efficient implementation of the bagging strategy.

Dropout may be used for convolutional and for dense layers. The dropout rate in convolution layers is usually lower (around 20%) than in fully connected layers (where 50% is a commonly chosen value). Some authors suggest to use dropout only for the dense layers and use gradient smoothing techniques such as batch normalization for the convolutional part. It would be consistent to the use of bagging for the representation of an ensemble of classifiers.

Dropout reduces the network capability only for training. Inference during validation and test applies the complete network to the unseen input samples. As it reduces expressiveness of the network, it has been suggested to increase the number of nodes and layers of network compared to one that does not use dropout. However, the network designers have usually just a vague idea about necessary expressiveness and tend to overestimate necessary expressiveness. Hence, the usual strategy is to create a network and then add dropout to deal with premature overfitting.

**Classes and Functions in Python**
Dropout is a layer in `tensorflow.keras.layers` that can be added after any convolution or fully connected layer. The following sequence, for instance,

```
from tensorflow.keras.layers import Dense, Dropout
model.add(Dense(layers[i], activation='relu', name='fc01'))
model.add(Dropout(dropout_rate))
```

adds a dropout layer with dropout rate `dropout_rate` to the dense layer with the name `fc01`.

### *11.2.5 Residual Neural Networks*

The introduction of convolutional building blocks led to deeper networks. Greater depth is a desired property as such network may represent more complex relationships than a shallow network. While space and number of trainable weights do not really limit the number of layers in a network, greater depth has a deteriorating effect on the gradient. Even activation functions with derivatives close to one over a wide range and carefully chosen batch normalization do not remove this effect completely.

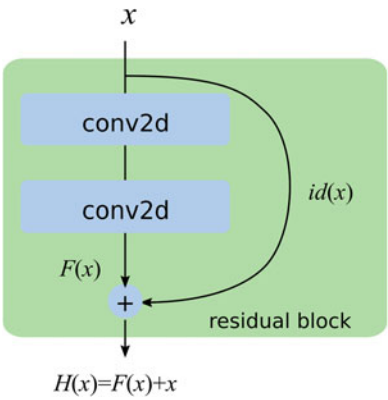



**Fig. 11.16** Information in a residual block flows along two paths. Along the skip connection, the information flows unaltered through the block, while the residual flows through the convolution layers

An activation function such as ReLU avoids the vanishing gradient effect at a node but every layer redistributes the loss over several nodes in the previous layer. This redistribution happens for all nodes in a layer. The redistributed loss more and more resembles white noise. This effect is sometimes called *shattered gradient*. Furthermore, normalization of input, output, and in-between keeps weights mostly below one. Hence, every new layer reduces the gradient length. Shattered and low gradients will cause weights to change slowly if at all.

The solution to this problem is to give up the sequential organization and introduce connections between nodes that skip layers, see He et al. (2016). The skipped layers along with the bypassing connection form what is called a *residual block*. The signal is passed along the original path through the layers of a residual block in the usual way and then combined with the signal received through the bypassing skip connection. If the input signal to a residual block were a tensor $x$ and the combination of weighted sums and application functions applied to it is $F(x)$, then the output of the residual block is

$$H(x) = F(x) + id(x), \tag{11.12}$$

where $id(x)$ is just the identity function applied to $x$ and "+" means a pointwise addition of the two tensors $F(x)$ and $id(x)$ (see Fig. 11.16). It requires that the resolution of $x$ and that of $F(x)$ are equal which is the case for connections within building blocks if the number of channels does not change in the block.

If resolutions are different, e.g., when pooling of convolutions with stride $>1$ happens along the original path, the function $id(x)$ is replaced by an interpolating function $ip(x)$ that changes the dimension of $x$ in order to match that of $F(x)$. If $F(x)$ is downscaled compared to $x$, then $ip(x)$ could just be a linear transformation that maps the $N_1$ elements of $x$ to the $N_2$ elements of $F(x)$. Interpolation is not appropriate if the number of channels is different for $x$ and $F(x)$ as there is no natural order of channels. Instead, $1 \times 1$ convolutions upscale or downscale the number of channels.

The values $F(x)$ and $x$ are the output from a previous layer with activation functions being applied to both terms. Sometimes, the result $H(x)$ includes a further activation function as part of the residual block.

The general topology of a residual network is similar to sequential networks discussed so far. In particular, a network with skip connections does not contain cycles so that the backpropagation algorithm can still be used for gradient computation. The feedforward step in a residual network forks at every residual block into the original path and the bypath through the skip connection.

If the network consists of a sequence of residual blocks, the input will partially be transported directly to the last residual block by using all the bypaths. In fact, every residual block doubles the number of paths through which the input signal reaches the last layer of the convolutional part of the network. It greatly amplifies the capabilities for model representation compared to a network without residual links.

The same happens when backpropagating the loss to the input layer for computing the gradient. A part of the information is directly propagated through all the skip connections to the input layer. Further paths for backpropagation use at least some of the skip connections. Hence, when computing the partial derivatives at earlier layers, gradient shattering or a vanishing partial derivative does not happen since just the residual is propagated through all layers.

Residual networks do not only increase the representational power of a given network but also enable much deeper networks while still producing meaningful partial derivatives at all layers. Hence, many modern classification networks use residual blocks as the performance of such network is superior to strictly sequential networks.

In order to understand what happens in a residual block it helps to treat the flow through the layers as residual flow $F(x)$ of a mapping $H(x) = F(x) + id(x)$. A sequence of such mappings $H(x)$ is searched that is capable to learn an optimal model from the training data. Similarly to fully connected networks it can be shown that convolutional neural networks are universal estimators provided that the number of channels or the number of layers can be made arbitrarily large (the combined functions act as base functions with local support similarly to wavelets or the spatial base). Hence, if a sufficiently large number of residual blocks are stacked, they could learn a mapping from arbitrary input features to linear separable features at the output.

The use of residual blocks enables such deep networks. Each residual block maps the input data $id(x)$ to the output and $F(x)$ just represents the difference, i.e., the residual, between the input and the mapping $H$ to be trained (hence the name residual network). Every new block will add to the power to map the input to the desired linearly separable features.

Choosing too many residual blocks will not hurt the training. If a residual block is superfluous, information would be transferred along the skip connections and weights of the residual block would go to zero.

Two examples for residual networks with slightly different designs that were used for image classification are ResNet34 and ResNet50. The number refers to the number of layers in the network. ResNet34 consists of four convolutional building

blocks. Each building block comprises between 5 and 11 convolution layers with 3 × 3 convolutional kernels followed by a convolution with stride 2 for down-sampling. The first block has 64 channels and every following block doubles the number of channels. Skip connections skip one layer at a time (including layers with stride-2 convolutions). The final classifier is a single-layer perceptron. Such simple classifier is sufficient since the convolutional part apparently maps features to a linearly separable output.

The classifier is simple compared to classifiers in earlier sequential networks, e.g., the already mentioned VGG16 network with two hidden layers, see Fig. 11.17. The number of trainable weights is substantially lower. As the network does not suffer from gradient shattering or a vanishing gradient, it can be trained more efficiently since fewer weights are to be adapted. ResNet34 outperforms VGG19 (an improved version of the VGG16 network) while being faster to train.

ResNet50 is deeper than ResNet34. The main difference besides being deeper is the so-*called bottleneck design* (see Fig. 11.18). Each 3 × 3 convolution layer is preceded by a 1 × 1 convolution to reduce the number of channels to that from the previous residual block. The 3 × 3 convolution layer is then applied to the reduced set of channels. A second 1 × 1 convolution extends the output to the number of channels that are transmitted via the skip connection from the previous residual block. The bottleneck design has been used in very deep networks since the usual doubling of channels per CBB (as in ResNet34) would grow the number of trainable weights too quickly leading, among others, to unwanted overfitting. Reducing the number of channels for each CBB allows the network to remove redundant information while avoiding too many channels.

As mentioned above there is no limit to the number of layers in a residual network. Very deep networks with several hundreds of layers, some even with more than 1000 layers, have been presented and have been found to be capable to represent quite complex models.

Residual blocks do not necessarily have to be organized sequentially. It is well possible to have a residual block within another residual block. It does not change algorithms for forward and backpropagation. An extreme example is DenseNet where every layer is connected to the last layer of the convolutional part of the network, see Iandola et al. (2014).

**Classes and Functions in Python**
Residual blocks are no longer sequential. Hence, instead of `Sequential` the more general class `Model`, that we introduced in Sect. 9.1.5, needs to be used to implement a residual block with Keras. The class `Model` has functions that combine input tensors to produce an output tensor. For modeling a residual block such as the one depicted in Fig. 11.13 two different flows of information are defined and added using the class `Add()` from `layers`:

(continued)

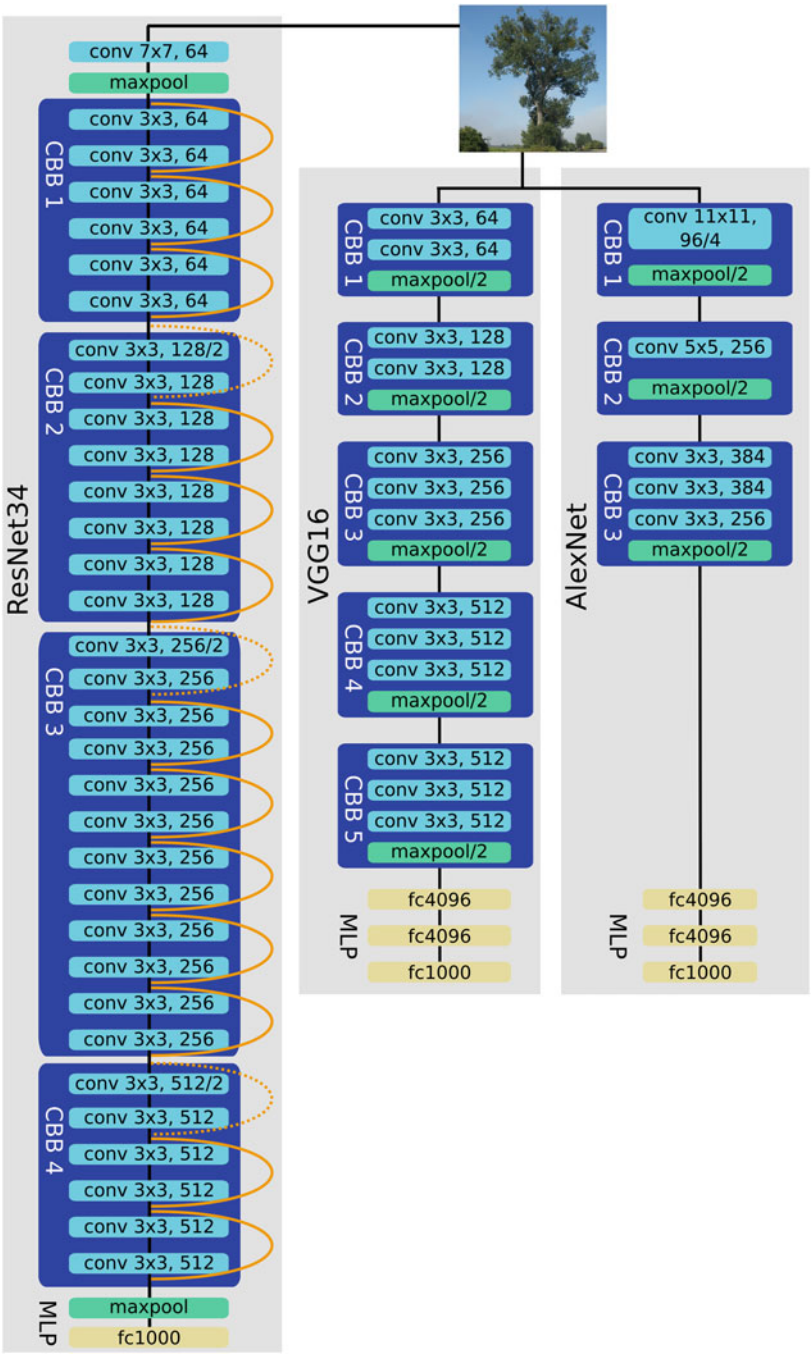



**Fig. 11.17** Comparison between AlexNet and VGG16, two older, sequential network structures, and ResNet34. The larger number of layers of ResNet34 is able to create advanced features that can be classified by a single-layer perceptron. The performance is better, nonetheless, compared to the older sequential networks

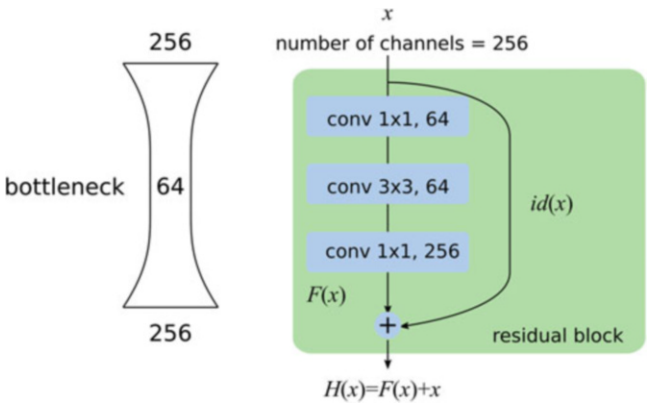



**Fig. 11.18** The residual block of ResNet50 (and other even deeper networks) contains a bottleneck that reduces the number of channels (filters) along the residual path

```
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Conv2D, Add
...
# function creates a residual block consisting of two convolutional
# layers with n_filters filters each
def add_res_block(inputs, n_filters):
    outputs= Conv2D(kernel_size=3, filters= n_filters,
                strides=(1,1), padding='same',
                activation='relu') (inputs)
    outputs= Conv2D(kernel_size=3, filters= n_filters,
                strides=(1,1), padding='same',
                activation='relu') (outputs)
    res_out= Add()([inputs,outputs])

    return res_out
```

For a residual block where the resolution is reduced and the number of filters is increased the input data has to be changed accordingly before adding it to the convolution results:

```
def add_res_block_d(inputs, n_filters):
    outputs= Conv2D(kernel_size=3,filters= n_filters,
                    strides=(2,2), padding='same',
                    activation='relu') (inputs)
    outputs= Conv2D(kernel_size=3, filters= n_filters,
                    strides=(1,1), padding='same',
                    activation='relu') (outputs)
    input_red= Conv2D(kernel_size=1, filters= n_filters,
                      strides=(2,2), padding='same',
```

(continued)

```
                    activation='relu') (inputs)
    res_out= Add()([input_red,outputs])

    return res_out
```

after importing all the necessary objects, a small residual neural network can be created by adding the final pooling and dense layers, for instance, for

```
inputs= Input(shape=(32, 32, 3))
res0 = Conv2D(kernel_size=3, filters=20, strides=(1,1),
              padding='same', activation='relu') (inputs)
res1= add_res_block(res0, 20)

# reduce feature maps, increase filters
res2= add_res_block_d(res1, 40)
res3= add_res_block(res2,40)

# reduce feature maps, increase filters
res4= add_res_block_d(res3,80)
res5= add_res_block(res4,80)

# Pooling, flattening and final MLP
conv_output= MaxPool2d(pool_size(2,2))(res5)
features= Flatten()(conv_output)
outputs = Dense(10,activation='softmax')(features)

# make network model
model= Model(inputs,outputs)
```

## 11.3 Exercises

### *11.3.1 Programming Project P11.1: Transfer Learning*

Use pre-trained networks and train them to label the Imagenette and Imagewoof data. Imagewoof is found at the same location than Imagenette. It shows different breeds of dogs and is more difficult to classify compared to the very distinctive Imagenette classes.

Since the two are subsets of ImageNet, a network that is pre-trained on ImageNet (e.g., VGG-style or ResNet-style networks) should be appropriate to transfer trained knowledge. Decide on the type of transfer learning and implement it by copying and combining trained and untrained models, or by freezing layers.

### 11.3.2 Programming Project P11.2: Label the Imagenette Data III

Extend your network from P10.1 for Imagenette classification by adding batch normalization and dropout layers. Experiment with noise augmentation and cutout augmentation. If, for cutout augmentation, you do not have access to the keras-cv library, write your own cutout function that cuts out randomly positioned and sized regions from the image (it is not difficult, use the random functions of NumPy to determine cutout boundaries and delete everything within the cutout region).

The goal is to have a model that is powerful enough for a good classification (again, take VGG16 for inspiration on the number of layers, building blocks, and channels but downscale it) but does not overfit easily. Augmentation, dropout, and batch normalization all have different effects on the training process (see the respective sections in this chapter) so that you have to find out for yourself what combination is the most useful.

Try your final model on the Imagewoof data as well.

### 11.3.3 Programming Project P11.3: Residual Networks

Implement a residual model by taking a sequential model as blueprint for building blocks and layers in a block and adding residual links that skip blocks. Start with your best model on the CIFAR10 data (it will be small) and compare network performances with and without residual links. Do then the same for your best sequential network that labels the Imagenette data.

### 11.3.4 Exercise Questions

- Why should the noise level not be too large when using added noise for data augmentation?
- Does adversarial training increase robustness with regard to a change of scope of the input data (e.g., classifying pictures with the same labels but taken in a different region of the world)? Justify your answer.
- Why does none of the augmentation techniques presented in Sect. 11.1 increase the diversity with respect to the training data set?
- Explain what would happen if a GAN has a very powerful generator and a weak discriminator? Think of the two conflicting goals that have to be optimized.
- Does GAN augmentation reduce bias in the training data? Justify your answer.
- What is the underlying assumption for semi-supervised learning that includes unlabeled samples? Does this technique increase the robustness of the classifier with regard to outliers? Justify your answer.

- What is the goal of batch normalization? Why is it necessary?
- What is the benefit of batch normalization for the training process? What is one (suspected) reason for it?
- Why are dropout layers usually used more in the fully connected part of a classification network?
- What does the use of dropout layers have to do with ensemble learning? What is the advantage of ensemble learning?
- Why does the use of skip connections allow for deeper networks?

## References

Akhtar, N., & Mian, A. (2018). Threat of adversarial attacks on deep learning in computer vision: A survey. *IEEE Access, 6*, 14410–14430.

Antoniou, A., Storkey, A., & Edwards, H. (2017). *Data augmentation generative adversarial networks*. arXiv preprint arXiv:1711.04340.

Creswell, A., White, T., Dumoulin, V., Arulkumaran, K., Sengupta, B., & Bharath, A. A. (2018). Generative adversarial networks: An overview. *IEEE Signal Processing Magazine, 35*(1), 53–65.

DeVries, T., & Taylor, G. W. (2017). *Improved regularization of convolutional neural networks with cutout*. arXiv preprint arXiv:1708.04552.

Doersch, C. (2016). *Tutorial on variational autoencoders*. arXiv preprint arXiv:1606.05908.

Goodfellow, I. (2016). *Nips 2016 tutorial: Generative adversarial networks*. arXiv preprint arXiv:1701.00160.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT Press.

Goodfellow, I. J., Shlens, J., & Szegedy, C. (2014). *Explaining and harnessing adversarial examples*. arXiv preprint arXiv:1412.6572.

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *IEEE conference on computer vision and pattern recognition* (pp. 770–778). IEEE.

Huber, J. (2020). Batch normalization in 3 levels of understanding. *Towards Data Science, 6*, 1.

Iandola, F., Moskewicz, M., Karayev, S., Girshick, R., Darrell, T., & Keutzer, K. (2014). *Densenet: Implementing efficient convnet descriptor pyramids*. arXiv preprint arXiv:1404.1869.

Jing, L., & Tian, Y. (2020). Self-supervised visual feature learning with deep neural networks: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 43*(11), 4037–4058.

Navidan, H., Moshiri, P. F., Nabati, M., Shahbazian, R., Ghorashi, S. A., Shah-Mansour, V., & Windridge, D. (2021). Generative adversarial networks (GANs) in networking: A comprehensive survey & evaluation. *Computer Networks, 194*, 108149.

Ouali, Y., Hudelot, C., & Tami, M. (2020). *An overview of deep semi-supervised learning*. arXiv preprint arXiv:2006.05278.

Ribani, R., & Marengoni, M. (2019). A survey of transfer learning for convolutional neural networks. In *32nd IEEE SIBGRAPI conference on graphics, patterns and images tutorials (SIBGRAPI-T)* (pp. 47–57). IEEE.

Salimans, T., & Kingma, D. P. (2016). Weight normalization: A simple reparameterization to accelerate training of deep neural networks. *Advances in Neural Information Processing Systems, 29*, 1–9.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research, 15*(1), 1929–1958.

Tan, C., Sun, F., Kong, T., Zhang, W., Yang, C., & Liu, C. (2018). A survey on deep transfer learning. In *Artificial neural networks and machine learning–ICANN 2018: 27th international conference on artificial neural networks, proceedings, part III* (pp. 270–279). Springer International Publishing.

Van Engelen, J. E., & Hoos, H. H. (2020). A survey on semi-supervised learning. *Machine Learning, 109*(2), 373–440.

Zhuang, F., Qi, Z., Duan, K., Xi, D., Zhu, Y., Zhu, H., Xiong, H., & He, Q. (2020). A comprehensive survey on transfer learning. *Proceedings of the IEEE, 109*(1), 43–76.

# Chapter 12
# Learning Effects and Network Decisions


**Abstract** Methods will be presented that explain what a network has learned. The inspection of trained filters continues first attempts in Chap. 8 to investigate what kind of knowledge a filter represents. While filters of the first layers may be interpreted as templates for relevant image textures, filter effects from later layers have to be backpropagated to the input layer first. A deconvolution network is described that does this based on an input image. A more general way to look at trained filters is to search for optimal input of a given filter. An image is computed that maximizes the activation for some filter.

A different kind of analysis is to investigate what contributes to a specific decision. Two strategies will be presented. Occlusion analysis investigates the impact of removal of image parts on the classification result. Class activation maps are an alternative to compute what image regions may have been relevant for a labeling decision. They compute the impact of the different feature maps of the last convolution layer on the classification decision and use this as weights to generate a class activation map from weighted averaging over all feature maps.

Neural networks are often termed black boxes as it seems intractable how they come to a decision. This is of course not so. We learned in the previous chapters that a sequential neural network consisting of convolutional building blocks and a multi-layer perceptron has a clear division of tasks. The convolutional part extracts features based on a non-linear multiresolution concept that are then classified by logistic regression in the fully connected part. Hence, effectiveness and function of feature generation as well as the model behind the MLP classifier can be analyzed.

The tendency to choose rather simple MLPs actually supports this division of tasks. Since the complete network is trained simultaneously, a simple MLP forces the convolutional part to extract advanced features that are easily separated by class in feature space. In order to understand how the two parts of the network contribute to this, the impact of image features on a decision in a specific case can be investigated as well as the way trained filters represent such features in the image. It is the topic of this chapter.

© The Author(s), under exclusive license to Springer Nature Singapore Pte Ltd. 2024

K. D. Toennies, *An Introduction to Image Classification*,
https://doi.org/10.1007/978-981-99-7882-3_12

## 12.1 Inspection of Trained Filters

The convolutional part of a deep classification network should extract meaningful features. Unfortunately, it is difficult to define and to assess what exactly is meant by "meaningful." From traditional image classification, we know that features are deemed to be appropriate, if

- they do not contain redundant information.
- the information is necessary to distinguish between classes (relevancy).

From considerations about the interpretation of classifier output we know that features would be ideal, if their distribution is log-linear. In this case, output can be mapped to a posteriori class membership probabilities. Separation between classes is then possible by linear decision boundaries. In such case, a neural network requires a number of single-layer perceptrons, one for each class, in the MLP part.

Non-redundant feature representation cannot be guaranteed by network design. However, a number of experiments may help to see whether redundancy is sufficiently removed. Redundancy is mainly due to the number of channels in each convolutional building block. We have learned earlier that part of it is intentional.

Independent training of channels may accidentally produce several channels that represent the same or very similar characteristics of the input. If the number of channels is higher than the potential number of different input characteristics it makes it more likely that these differences are represented by the trained model.

### *12.1.1 Display of Trained Filter Values*

Redundancy is not easy to detect. Similar characteristics of two filters may not be redundant but necessary to distinguish between classes. Display of features that have been extracted by the various channels of a given layer can be used to analyze the behavior of different network designs in this respect.

Extracted features of the first convolution layer are given by the kernel values of the filters (channels) themselves. Since a convolution is related to a cross-correlation with this kernel,[1] the filters represent templates that are matched with the image (see Fig. 12.1). They should represent relevant image detail within the perceptive field of the filter. Features are particularly expressive when the perceptive field of the first layer is large (such as the $11 \times 11$ kernels of the first layer of AlexNet).

Observing trained filters from this first layer tells us three things:

[1] A convolution with a function is equivalent to computing the covariance with the mirrored function. Cross-correlation is just the covariance normalized by the variances of the two functions. Hence, convolution represents the non-normalized cross-correlation.

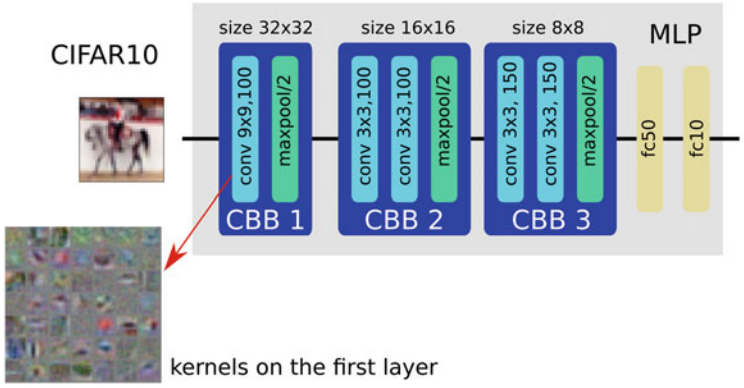



**Fig. 12.1** Example of filters of the first layer of a simple network that has been trained on the CIFAR10-data set. Because of the small size (32 × 32 pixels), features are somewhat difficult to discern. However, preference to differently oriented edges and different colors can be seen

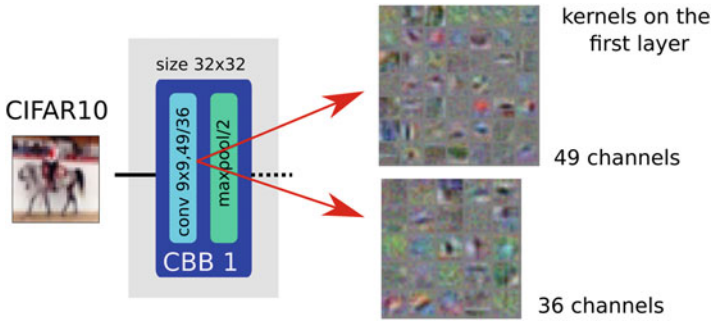



**Fig. 12.2** Adaptation of the network of Fig. 12.1. The first layer has now either 49 or 36 channels. The performance is similar (about 70% accuracy after convergence). It is noticeable that edges and lines with different orientations are extracted in both cases

- Filters represent edges, corners, simple textures, and various colors. This is expected since these kinds of features are already found to be relevant in traditional image classification.
- Filters represent differently rotated versions of a texture. It is unsurprising as well since rotational invariance is a relevant attribute of several high-level feature extractors in traditional image classification.
- Some characteristics are represented several times in slightly perturbed versions. This distinction may be relevant but it may also point at a redundant representation.

An interesting experiment with regard to the relevancy of features at this level is to change the number of channels and analyze change of trained filters and classification performance (see Fig. 12.2). If the classification performance decreases with a smaller number of channels, the observed similarity was either not redundant or the

redundancy was necessary in order to capture other image attributes by other channels. If classification performance does not change or even improve for this scenario, it is worthwhile to analyze the trained filters. Are similarly looking filters missing? Are distinctive filters missing while others appear? Both cases point at the removal of redundant information. A similar analysis with an increase in the number of filters tells whether this increase adds necessary capabilities to the model.

### *12.1.2 Deconvolution for Analyzing Filter Influence*

Analysis of patterns for which filters are sensitive is interesting for all layers in the convolutional part. However, a filter is a template for the input from the previous layer. If this is not the first layer it cannot easily be interpreted in terms of the input image. All the linear and non-linear transformations between this layer and the input layer need to be included in the analysis.

A method presented by Zeiler and Fergus (2014) inverts the convolutions on the way from a given image to some filter in a layer in order to find out just what this filter extracts from the image. The deconvolution network does unpooling and deconvolution of trained filters. Unpooling is approximate by interpolation since the information loss by the resolution reduction cannot be undone. Deconvolution for a linear convolution kernel can be done if the convolution matrix is invertible and may be approximated otherwise.

With the deconvolution ability included in the classifier, a sample image is fed through the network (see Fig. 12.3). The resulting estimate of the one-hot-vector is

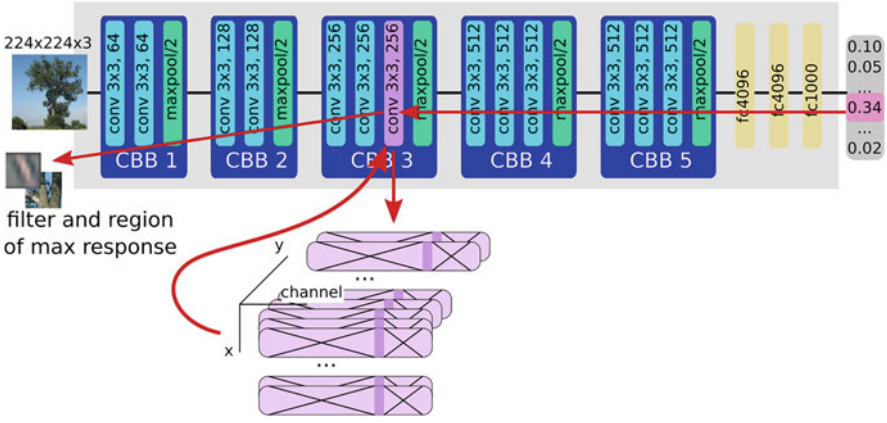



**Fig. 12.3** Principle of the deconvolution network. A sample is fed through the network and the output for a specific class is propagated back. At the layer for which a filter response shall be determined all but this filter is set to zero before continuing the backpropagation. The response at the output shows the structure to which this filter responded. The location where this response is maximum is computed as well

then erased except for one class. This class is usually selected from those with highest response. Now, this altered information is fed backward. When the layer with the filter in question is reached, all output of this layer except for this filter is set to zero. After that, the output from this layer is fed further backward toward the image. The result shows image features that led to the activation of the filter in question for the selected class. The procedure can be repeated for several other entries of the output vector to show different "templates" represented by this filter that gave rise to this specific output value.

In their publication, the authors showed responses for the top-9 classes in the 1000-class image net challenge applied to the VGG16 network. Image features in earlier layers were quite similar for all nine classes but the further the information progressed through the layers the more different were the extracted features.

Using a deconvolution network to detect patterns to which specific filters in a layer are sensitive gives similar insights than those gained from looking at filters in the first layer. However, it is just sensitivity with respect to a certain image and a certain class. It is less suitable for finding a good network design but rather points at an explanation why a network has made a certain decision.

One can go even a step further in this regard. Since convolution layers retain the spatial organization of an input image, a subregion in the image can be determined that delivered the highest response to the analyzed filter in a layer. Analysis of this aspect shows how later layers concentrate on regions of interest that belong to the foreground and are relevant to the classification. Comparing responses for different classes shows the variation of what is deemed to be part of the relevant foreground in an image. We will explore this aspect further in Sec. 12.3.

### *12.1.3 About Linearly Separable Features*

If samples satisfy conditions for a log-linear model, a logistic regression is the optimal classifier. Log-linearity could be tested using samples from the validation data set. Unfortunately, the dimensionality of features that are fed to the fully connected layers of the network is too high and the number of samples in the validation data set is too low for making this kind of estimate.

Furthermore, a moderately deep classification network may be too shallow with too few channels per layer to result in a log-linear feature distribution after the last layer of the convolutional part. Hence, even if an estimate of log-linearity were possible, the results would be unsatisfactory. The network would still require several layers in the subsequent fully connected layers to provide a log-linear feature distributions of the raw output after the last hidden layer. Since just the overall network performance is known, it would be impossible to compare suitability of feature extraction for different networks.

An indirect means, however, to get an idea about the quality of features produced in the convolutional part, is to experiment with different architectures for the fully connected layers. Good features will not require a complex MLP. It is a desired

property since MLPs produce the bulk of weights to be trained. Fewer weights mean more efficient training.

## 12.2 Optimal Input

Using the deconvolution as described above analyzes optimal filter input for specific images and for specific classes. It is difficult to conclude anything about the general behavior of the network. It is just as interesting to ask to what input a given filter is most sensitive. It bears some similarity to filter inspection in the first layer because it generates textures for which a filter in an arbitrary layer produces the highest response.

The idea has been presented in Chollet (2021). For not biasing the analysis, the process starts with a random noise image. Given a suitable loss function to maximize the filter response, it computes the output at this filter and changes image values by gradient ascent, while networks weights are frozen. Consequently, the random image is changed to one that maximizes the filter response.

The steps to analyze a specific filter $f$ on a layer $l$ are easily implemented via TensorFlow because they use functions that are also used to train a network via backpropagation. The following processes need to be carried out:

- Load a trained model for which filters are to be analyzed. Since we are not interested in the final output, the model may be loaded without the classification part to save memory and computation time.
- Define a suitable loss function. Since we want to maximize the filter response a suitable loss would be the average filter response in the layer. Taking the maximum in the layer could be even more suitable as we do not expect object characteristics to be represented everywhere in the image. However, taking the maximum is susceptible to artifacts and may produce instable results.
- Compute the gradient of the loss with respect to the image data by backpropagation from the filter layer to the input image and apply it with a given step size to the image.

The process is repeated for several epochs. Finally, the optimized input image should show a pattern that leads to the largest average response of the filter in question (see Fig. 12.4). Since average response has been chosen as loss function the result is usually a repetitive pattern. The size of the pattern is in the range of the perceptive field of the filter in the layer in terms of image pixels.

The process can be repeated for all filters in a layer and then displayed. As they contain substantial redundancy, some of the optimized input images will still contain noise. These filters will have zero weights at the incoming edges and do not contribute useful information to the output. All other filters will show patterns with high impact on the layer output. Since activation functions commonly used for classification networks increase monotonically with the input, this high impact

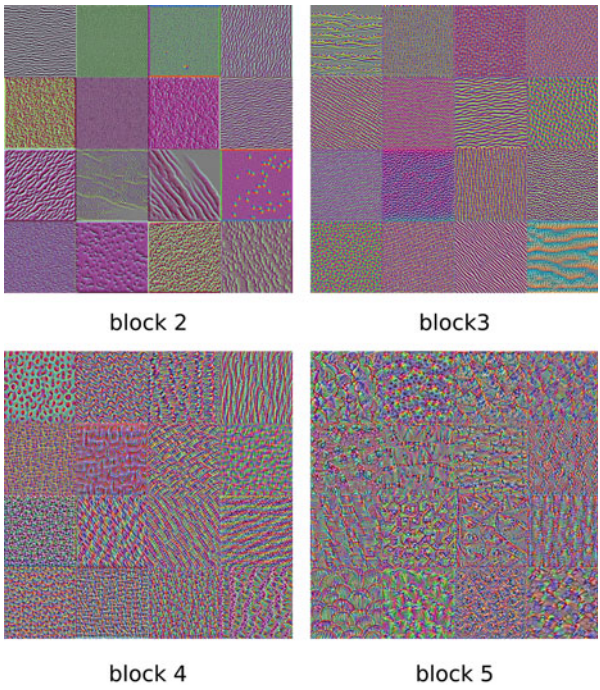



**Fig. 12.4** Optimal input for some of the filters in the first convolution layer of convolutional blocks of the VGG16 network

will transfer to the next layers and have substantial influence on features that will be delivered to the classifier part.

Inspection of filters on the layers provides insight into the diversity of features relevant for classification. Highly diverse features on lower levels point at late decisions about relevant features, a property which is usually wanted. Very similar patterns represented by different filters of a layer or many noise patterns in a layer point at a redundant representation of features. Distinctive patterns on the last layers that match object-specific image content point at an efficient extraction of object appearance characteristics.

If analyzing a well-trained model (e.g., the VGG16 network trained on millions of samples from ImageNet) it usually shows the expected behavior with well-defined, diverse low-level features and distinctive shape patterns at higher levels. It is less so, however, if simpler networks, trained on fewer samples, are investigated. For such networks, analysis and subsequent adaptation of the model architecture may lead to solutions where meaningful patterns emerge on the later layers despite the limited number of training samples. Of particular interest are also pre-trained networks where some of the later layers have been retrained to adapt to particularities of the new problem. If successful, these layers should result in filters that show specific attributes of objects depicted in these new images.

**Classes and Functions in Python**

There is no class or object in the TensorFlow distribution to compute optimal images. However, as it is just a gradient ascent on the image data, you can use a similar program snippet to access and use gradients from backpropagation than it was used for computing adversaries in Sect. 11.1.4. The differences are

- The process starts with a random image.
- The loss maximizes a filter response (instead of the classification loss).
- The gradient is used for carrying out an iterative ascent over several epochs (instead of just adding it once to the input image).

A program snippet to carry out this gradient ascent for a layer with name `layer_name` and a filter in this layer with index `filter_index` using the trained VGG16 network is the following:

```
import tensorflow as tf
import numpy as np

# Create a connection between the input and the target layer
model = tf.keras.applications.vgg16.VGG16(
                                        weights='imagenet')
submodel = tf.keras.models.Model([model.inputs[0]],
                  [model.get_layer(layer_name).output])

# random noise image, shifted/scaled to 128 +/-20
input_img = np.random.random((1, 224, 224, 3)) # random values
                                               # between 0 and 1
input_img = (input_img - 0.5) * 20 + 128.

# cast input_img to tensor object and declare it as variable
# for being observed by GradientTape
tf_output = tf.Variable(tf.cast(input_img, tf.float32))

# Iterate gradient ascent for 'epochs' steps
for _ in range(epochs):
    with tf.GradientTape() as tape:
        outputs  = submodel(tf_input)
        # loss value is average output at 'filter_index'
        loss_value = tf.reduce_mean(outputs[:, :, :, filter_index])
        # take gradiens of the loss value w.r.t. tf_output
        grads = tape.gradient(loss_value, tf_output)

    # normalize gradient (just the direction is of interest)
    normalized_grads = grads / (tf.sqrt(tf.reduce_mean(
                                tf.square(grads))) + 1e-5)
    # update output tensor
    tf_output.assign_add(normalized_grads * step_size)
```

(continued)

The resulting tensor `tf_output` needs to be rescaled to 0. . .255 and cast to a NumPy array of type "`uint8`" (use function `numpy()` of the tensor object) before it is displayed. If optimal images for several filters shall be produced, the procedure has to be repeated for a range of filter indices.

## 12.3 How Does a Network Decide?

We observed in Sect. 12.1.2 that the deconvolution network to infer templates and optimal input explains reactions to specific input rather than telling what the network has learned. This is an interesting question especially when the selected label for an unknown image is incorrect. However, looking up why a specific filter in some layer has produced the response that led to the final output explains only part of this. It tells us what parts of the image and what patterns have been responsible for delivering features to the fully connected layers to compute a decision. It does not tell us which of these features were responsible for the decision since the flow of information in the fully connected part has not been investigated.

Different strategies to trace the flow of information from the output back to a specific input image will be discussed in this section. The goal is to identify subregions in an image that are main contributors to the decision result. Together with image attributes and relevant subregions for a specific filter from the deconvolution network described above, the information can be used to determine attributes and regions being responsible for a correct or an incorrect result.

### *12.3.1 Occlusion Analysis*

A simple means to detect relevant image regions for some decision is to replace parts of an image by a patch of non-information and analyze the impact on the result. This is called *occlusion analysis* (see Fig. 12.5).

The method consists of the following steps:

- Compute output of a trained network for a certain image and keep class label $c$ and output probability $y(c)$ of the corresponding entry in the output vector.
- Specify a size of a patch and the values that replace the pixel values in the patch. The patch size should be large enough to cover substantial parts of the perceptive field for features of the last layer and small enough to allow spatial analysis. The replacement value is either uniform gray value (e.g., 0, 0.5 or 1 in a normalized image) or just noise.
- Slide the patch over the image and infer class membership with the patched image. Usually, the patch is shifted by a number of pixels corresponding to the length of the patch. Every inference produces its own output vector $y_{patch}(c)$ for class $c$. The relevance of occlusion is then the ratio $r_{patch} = y(c)/y_{patch}(c)$.


**Fig. 12.5** Two different results from occlusion analysis. Regions highlighted in red are the most important for the classifier decision. In the first case, the network concentrated on the horns and the texture of the fur. The misclassification of the second image seems to be caused by the repetitive nature of the patterns of shadows and trees

- The map of values $r_{patch}$ is upsampled to the original input size and overlayed to the image. It should show high values for patches where the occlusion led to a large decrease in label probability $y_{patch}$.

Occlusion analysis is a simple means to estimate the relevancy of subregions for the classification. It is, however, only indirectly related to the feature maps that are submitted from feature extraction to the classifier part. While results are just slightly dependent on the values with which the patch is filled, selection of the patch size has a higher influence on the result.

**Classes and Functions in Python**
There is no class for occlusion analysis in TensorFlow. However, such method is easily implemented:

- Take a trained network and an arbitrary image (use a model object from `tensorflow.keras.applications` such as `vgg16.VGG16` or `resnet50.ResNet50`).
- Make a label prediction for this image using the network by calling `model.predict()` on the trained model `model`. Keep class label `image_class` and probability `image_prob` for this class.
- Partition the image into NxN blocks.

(continued)

- Create a sequence of N·N images `block[i,j]`, where block (i,j) is erased.
- Call `model.predict()` on this sequence and extract probabilities `block_prob[i,j]` for class `image_class` for each `image block [i,j]`.
- Compute the ratio `relevancy[i,j]=image_prob/block_prob [i,j]`.
- Resize the relevance map to the original image size, overlay it on the original image, and display it.

### 12.3.2 Class Activation Maps

*Class activation maps* (CAMs) directly display the influence of filters of the last layer of convolutional blocks on the classification result, see Oquab et al. (2015). It bears some similarity to the simple output of raw activation that we used in Sect. 8.3.2 to identify regions that impact the classification. Different to this, it computes class-specific activation maps. CAM assumes a single-layer perceptron as classifier. Hence, features created by the network should be of sufficiently high quality. The different residual neural networks presented in Sect. 11.2.5 are examples for such networks.

A further restriction when creating class activation maps is that the feature vector submitted to the fully connected layer is generated by global average pooling. Let $A_i=\{x_{u,v,i}\}$ be the feature map of activations $x_{u,v,i}$ in channel $i$. The output to the MLP part is then averaged separately for each channel

$$f_i = \frac{1}{UV}\sum_{u=1}^{U}\sum_{v=1}^{V} x_{u,v,i}, \tag{12.1}$$

where $x_{u,v,i}$ is the filter output $i$ at location $(u,v)$ in the feature map with size $(U,V)$ of the last layer of the convolutional part. The feature value $f_i$ is part of the feature vector $\mathbf{f}$ that is submitted to the fully connected network.

If, for instance, we used the convolutional part of the VGG16 network on ImageNet data, the $7 \times 7$ spatial nodes with 512 channels per spatial node of the final layer of the convolutional part would be averaged over the $7 \times 7$ spatial nodes to form a feature vector $f$ of length $K = 512$.

For a given class $c_j$ the raw output $y_j$ before applying the softmax operation is a weighted sum of these features

$$y_j = \sum_{i=1}^{K} w_{i,j} f_i, \tag{12.2}$$

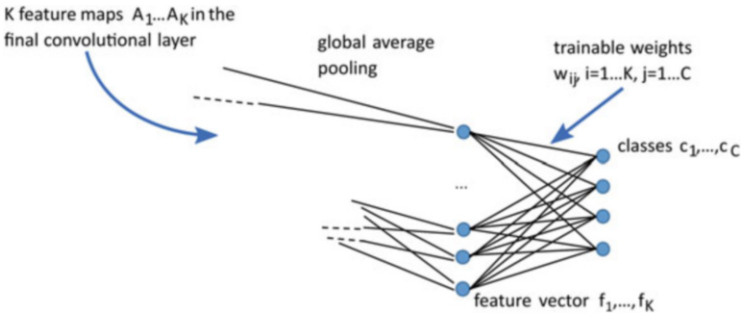



**Fig. 12.6** For computing class activation maps (CAMs), feature maps of the last convolution layer are averaged to form the feature vector $f$. If the feature vector is connected to the output layer without intermediate layers, the weight $w_{ij}$ of a feature $f_i$ connected to a specific class $c_j$ indicate the importance of the corresponding feature map $A_k$

where $w_{i,j}$ is the weight of the edge between feature $i$ and class $j$. Each feature is the activation average of a filter. The softmax function will assign higher probabilities for higher values of $y_j$. Hence, the weights $w_{i,j}$ indicate the importance of the activation map $A_i$. Since activation functions are usually monotonically increasing, the influence of an activation map depends on nodes with high input value $x$. Spatially varying importance $CAM_j$ for some class $j$ is determined by summing over all weighted feature maps (see Fig. 12.6):

$$CAM_j = \sum_{i=1}^{K} w_{ij} A_i. \tag{12.3}$$

For identifying corresponding image regions, the resulting activation map $CAM_j$ is upsampled to match the image size and overlayed to the image.

CAMs are relevance maps for a class decision that do not require specification of patch sizes and replacement values for pixels in the patch. However, requiring a single-layer perceptron with global average pooling after feature extraction are severe limitations.

### *12.3.3 Grad-CAM*

Grad-CAM, presented by Selvaraju et al. (2017), deals with the deficiencies of CAM by using the gradient of the output with respect to the activations instead of the weights themselves and by distinguishing between activations from different channels. In order to understand the concept of Grad-CAM, let us go back to CAM and analyze the relation between the weights $w_{i,j}$ and the gradient with regard to the activation. The partial derivative $\frac{\partial y_j}{\partial f_i}$ of the raw output $y_j = \sum_{i=1}^{K} w_{i,j} f_i$ (Eq. (12.2)) with respect to a feature $f_i$ is

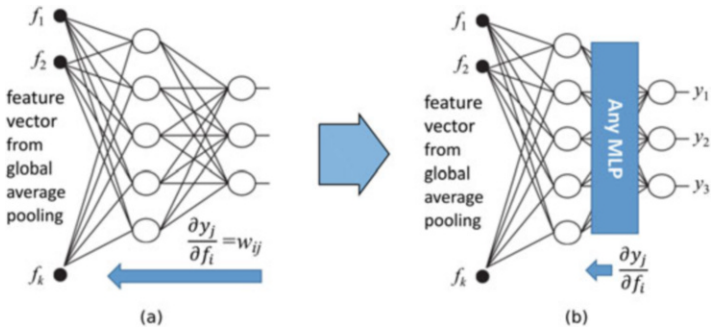



**Fig. 12.7** (**a**) The importance of a feature $f_i$ for some class $c_j$ in the perceptron is given by the derivative of the output $y_j$ for this class with respect to the feature $f_i$. It is the weight of the edge connecting class $j$ with feature $i$. (**b**) This does not change if the single layer perceptron replaced by an arbitrary multi-layer perceptron

$$\frac{\partial y_j}{\partial f_i} = \frac{\partial \sum_{k=1}^{K} w_{k,j} f_k}{\partial f_i} = w_{i,j}. \tag{12.4}$$

Hence, instead of the weights, the partial derivatives of the raw output with respect to a feature can be used to estimate the importance of a feature map.

This seems unnecessarily complicated but the argumentation applies as well, if the multiclass single-layer perceptron is replaced by an arbitrary multi-layer perceptron (see Fig. 12.7). The relevance of a feature map for a certain class decision can still be estimated by computing the partial derivative of the raw output for this class with respect to the feature $f_k$.

This simple change of perspective leads to a solution that applies to any kind of classifier as part of the deep convolutional network. For all classes $j$ class-specific activation importance values $\alpha_{i,j}$ are computed by backpropagation of the raw output $y_j$ to feature $f_i$:

$$\alpha_{i,j} = \frac{\partial y_j}{\partial f_i}. \tag{12.5}$$

What still remains is the requirement to average over the feature maps when generating the features from the output of the activation map. The concept of using derivatives to determine importance is applied again to remove this restriction. Instead of computing the global average the final output of each channel of the convolutional part of the network is flattened. Features from all channels are concatenated.

With $f^i_{u,v}$ being the feature generated at spatial location $(u,v)$ and channel $i$, the activation for a given channel is averaged from partial derivatives with respect to this feature at all locations $(u,v)$ (see Fig. 12.8):

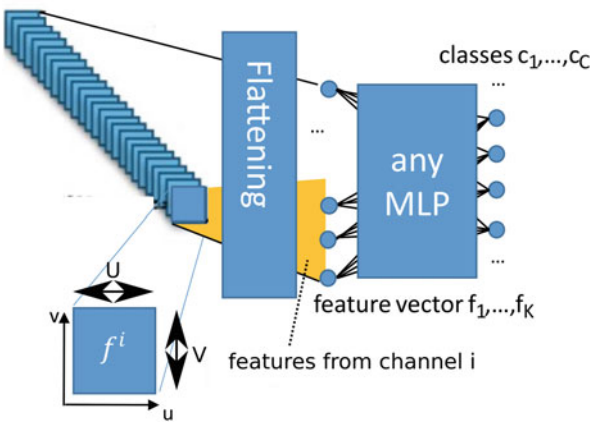



**Fig. 12.8** In order to remove the requirement for global average pooling, the partial derivatives with respect to all features of a feature map are averaged and used as a weight for aggregating feature maps to compute the Grad-CAM

$$\alpha_{i,j} = \frac{1}{UV} \sum_{u=1}^{U} \sum_{v=1}^{V} \frac{\partial y_j}{\partial f_{u,v}^i}. \tag{12.6}$$

Finally, class activation maps are computed as before, except that the authors decided not to include negative activations. Hence, ReLU is applied to create the activation map:

$$grad\_CAM_j = ReLU\left(\sum_{i=1}^{k} \alpha_{ij} A_i\right). \tag{12.7}$$

It is still an average importance that is computed, but now the weights are computed separately for every location $(u,v)$ before summing them up. It no longer requires global average pooling so that class activation maps can be computed for arbitrary classification networks.

Grad-CAM may be used to understand an incorrect classification (see Fig. 12.9).

It may also show that regions of highest importance in a given image may not or may just partially include the class-determining characteristics because other objects dominate (see Fig. 12.10). Additional training images and/or data augmentation may help to overcome the problem. Grad-CAM may further show bias of a trained model. In this case, incorrect regions of importance are determined by the network in a set of images. An example would be an incorrect association of a class with certain types of background instead of relevant foreground detail. Again, further training that intentionally uses images with different backgrounds should help.

It is interesting to compare Grad-CAM results for different classes. Figure 12.11 shows an example for the incorrectly classified picture in Fig. 12.10. The overall activation map, i.e., the unweighted sum of activation maps shows that most of the activation is on the foreground object. However, the top-1 label is "traffic light" with most of the activations in the feature map stemming from a region that covers the two visible traffic lights. For labels that describe cars (e.g., the label "cab," which came in fourth) the foreground region indeed delivers most of the activation. Other, less

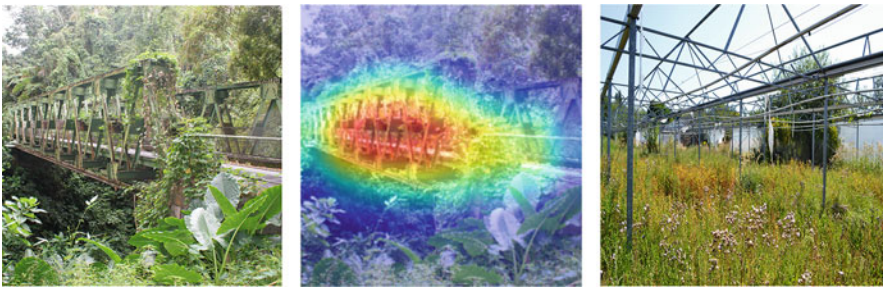


**Fig. 12.9** The object on the left is a class of ImageNet (bridges are contained but not this type). The label greenhouse is predicted. It is a nice example of the discrepancy between semantic defined by use and a decision based on appearance. A main attribute of the bridge is the repetitive pattern of the superstructure that resembles that of a greenhouse (example on the right)

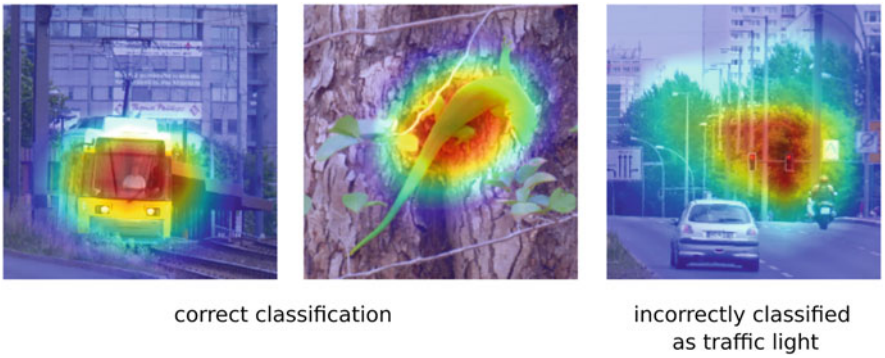



**Fig. 12.10** The first two images were correctly classified. Grad-CAM shows that the network concentrated on regions that contained class-specific attributes. The classification of the last image (traffic light) is correct for the region on which the network concentrated. However, the car in the foreground was not found. This is probably because its features were less discriminative than those of the traffic light (which is not a surprise as traffic lights are designed to stand out)

probable labels such as "moto-cycle" (9th) highlight regions that include this object. It tells us that the network was able to classify multiple objects. However, it assigned highest probability for a label that did not correspond to the region that caused the highest activation in the image. It seems that features that voted for traffic lights were highly discriminative so that individual feature maps that highlighted this region received higher weights in the final classification.

Grad-CAM may deliver incorrect results, if small but highly class-specific regions determine the label. As it averages over the feature map, such small regions may result in a lower average gradient than lower gradients that are present everywhere in the map. Small detail may often be relevant for image classification and is, for instance, the reason that max-pooling is the preferred means for pooling in classification networks.

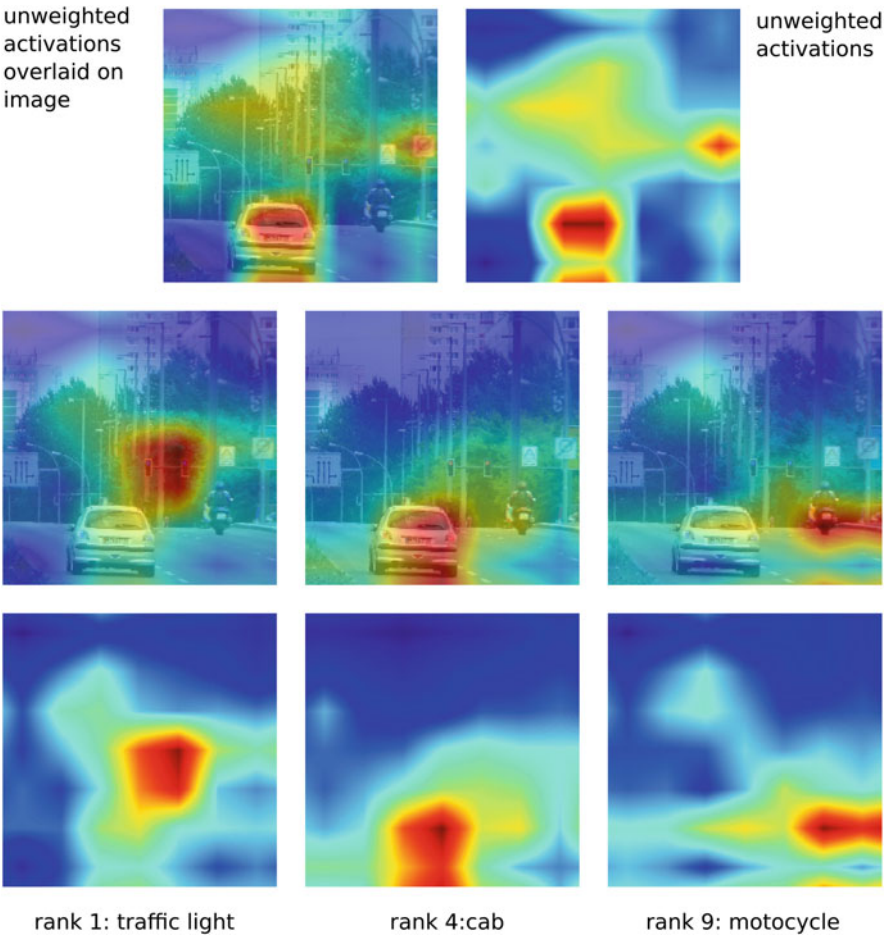



**Fig. 12.11** Regions for class activations vary for different classes. The highest class activation for the label that is ranked first not necessarily corresponds to the unweighted average activations for all classes

A variant of Grad-CAM that deals with this problem is HiResCAM, see Draelos and Carin (2020). The gradient is computed separately for every spatial node $(u,v)$. It results in a map $\alpha_{i,j,u,v}$ with different weights for every channel, spatial location, and class. The importance map is now computed by pointwise multiplication of $\alpha_{i,j,u,v}$ with every node $(u,v)$ and summing up over all channels. It has been shown to produce superior results in a number of cases, where class-specific characteristics in an image pertained to rather small regions.

Grad-CAM, HiResCAM, and various other adaptations of CAM for visual interpretation are useful tools to determine just what image content resulted in the output of the classifier. Combined with other methods that explain what input characteristics produced this output, they can help to understand specific network

decisions. It should enhance network performance, e.g., by improved training. This is to be taken with a grain of salt, however. With a poorly trained network or with a poorly designed network, results only give an indication of main influences for the model to deliver the observed result. It will probably not help much to determine just what of the many different network hyperparameters should be changed in order to achieve better results.

**Classes and Functions in Python**
There is no Grad-CAM class in TensorFlow. However, numerous implementations exist and are available on the internet. A version of Francois Chollet is part of the Keras documentation and can be found on GitHub (https://github.com/keras-team/keras-io/blob/master/examples/vision/grad_cam.py). Implementations of HiResCam can be found as well.

## 12.4 Exercises

### *12.4.1 Programming Project P12.1: Compute Optimal Input Images*

Take the program snippet from Sect. 12.2 and extend it to compute the optimal input for *N*x*N* filters. Write the *N*x*N* images in a common array, display it on the screen and write it to disk. Experiment with VGG16 and ResNet50 in `tensorflow.keras.applications`. Compare filters from layers in different building blocks and between the two different networks. What are the similarities between networks, what are the differences?

### *12.4.2 Programming Project P12.2: Occlusion Analysis*

Implement the sequence of steps listed in Sect. 12.3.1 and apply it to different images. Compare results for different trained networks from `tensorflow.keras.applications`. Then, use one of your own networks that you have implemented to label the Imagenette data. Submit images from the Imagenette database to a pre-trained large network and your own network. Analyze cases where both come to the same correct solution and cases where the networks produce different labels.

### 12.4.3 Exercise Questions

- What are possible strategies for finding out if a network has learned the "right" thing?
- Why is visualizing the filters of convolution layers most useful for the first layer?
- How can we figure out which image features are responsible for which filter response in any given convolution layer?
- What is the advantage of computing the optimal input for a given filter?
- What is the loss function for computing such an optimal input? What is the possible disadvantage of using this function?
- How do optimal images for filters differ within and between convolution layers when the network is well trained?
- What information can be drawn from the average of all activations of the last convolution layer?
- How can images with hidden subregions be used to find out how a network decides?
- Why can a class activation map only be generated if the classifier is linear?
- Why can Grad-CAM use the gradient of any network at the interface between CBBs and MLP?
- What strategy can be used to remove global average pooling for Grad-CAM?
- What is the motivation for HiResCAM? How is it different from Grad-CAM and why is this difference essential for better quality results?

## References

Chollet, F. (2021). *Deep learning with Python*. Simon and Schuster.

Draelos, R. L., Carin, L. (2020). *Use HiResCAM instead of Grad-CAM for faithful explanations of convolutional neural networks*. arXiv e-prints, arXiv-2011.

Oquab, M., Bottou, L., Laptev, I., & Sivic, J. (2015). Is object localization for free? Weakly-supervised learning with convolutional neural networks. In *IEEE conference on computer vision and pattern recognition* (pp. 685–694). IEEE.

Selvaraju, R. R., Cogswell, M., Das, A., Vedantam, R., Parikh, D., & Batra, D. (2017). Grad-cam: Visual explanations from deep networks via gradient-based localization. In *IEEE international conference on computer vision* (pp. 618–626). IEEE.

Zeiler, M. D., & Fergus, R. (2014). Visualizing and understanding convolutional networks. In *Computer vision–ECCV 2014: 13th European Conference, proceedings, part I* (pp. 818–833). Springer International Publishing.

# Index



© The Author(s), under exclusive license to Springer Nature Singapore Pte Ltd. 2024

K. D. Toennies, *An Introduction to Image Classification*,
https://doi.org/10.1007/978-981-99-7882-3