Supervised Classification of Microtubule Ends: An Evaluation of Machine Learning Approaches

Master's Thesis by
Felix Herter

1. Reviewer:	Prof. Dr. Tim Conrad
2. Reviewer:	Prof. Dr. Knut Reinert
Supervisors:	Dr. Daniel Baum,
	Dr. Norbert Lindow

22nd February, 2018

carried out at
Zuse Institute Berlin (ZIB)

Department of Computer Science
Freie Universität Berlin
Takustraße 9
14195 Berlin, Germany

Danksagung

Ich möchte mich an dieser Stelle herzlich bei denen bedanken, die diese Arbeit ermöglicht haben. Durch sie hat es auch in anstrengenden Zeiten immer Spaß gemacht.

Zuerst möchte ich mich bei Daniel Baum und Norbert Lindow bedanken. Ohne Daniels geduldige Betreuung und seinen kompromisslosen Einsatz wäre diese Arbeit schlicht nicht zustande gekommen. Norbert danke ich für die ästhetische sowie konzeptionelle Inspiration und seine Hilfe speziell bei der Softwaregestaltung.

Tim Conrad und Knut Reinert danke ich für die Bereitschaft, die Arbeit zu begutachten. Tim Conrad danke ich darüber hinaus für die wertvollen Tipps und Hinweise zur Gestaltung der Arbeit.

Eine datenbasierte Arbeit kann ohne Daten nicht entstehen. Dies wurde durch unsere Projektpartner der Core Facility Cellular Imaging an der TU Dresden unter der Leitung von Thomas Müller Reichert ermöglicht. Insbesondere möchte ich mich bei Anna Schwarz, Gunar Fabig, Marcel Kirchner, Robert Kiewisz und Stefanie Redemann bedanken. Sie klassifizierten in mühseliger Handarbeit tausende von Mikrotubulienden.

Meinen Eltern danke ich für ihre bedingungslose Unterstützung und ihr anhaltendes Vertrauen in mich.

Anne danke ich dafür, dass sie während der Entstehung der Arbeit immer für mich da war.

Declaration of Authorship

I hereby confirm that I have written this thesis on my own and that I have not used any other materials than the ones referred to. This thesis has not been submitted, either in part or whole, for a degree at this or any other university.

Ich versichere hiermit an Eides statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben. Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Berlin, 22nd of February, 2018

Felix Herter

Abstract

Aim of this thesis was to evaluate the performance of three popular machine learning methods – decision trees, support vector machines, and neural networks – on a supervised image classification task from the domain of cell biology. Specifically, the task was to classify microtubule ends in electron tomography images as open or closed. Microtubules are filamentous macromolecules of the cytoskeleton. Distribution of their end types is of interest to cell biologists as it allows to analyze microtubule nucleation sites. Currently classification is done manually by domain experts, which is a difficult task due to the low signal-to-noise ratio and the abundance of microtubules in a single cell. Automating this tedious and error prone task would be beneficial to both efficiency and consistency.

Images of microtubule ends were obtained from electron tomography reconstructions of mitotic spindles. As ground truth data for training and testing four independent expert classifications for the same samples from different tomograms were used. Image information around microtubule ends was extracted in various formats for further processing.

For all classifiers we considered how the performance varies when different preprocessing techniques (per-feature and per-image standardization) are applied. For decision trees and support vector machines we also evaluated the effect of training on a) imbalanced versus under- and over-sampled data and b) image-based vs feature-based input for specifically designed features.

The results show that for decision trees and support vector machines classification on features outperforms classification on images. Both methods give most equalized per-class accuracies when the training data was undersampled and when preprocessed with per-image standardization prior to features extraction. Neural networks gave the best results when no preprocessing was applied.

The final decision tree, support vector machine, and neural network obtained accuracies on the test set for $(open, closed)$ samples of $(62\%, 72\%)$, $(66\%, 70\%)$, and $(61\%, 78\%)$ respectively, when considering all samples where at least one expert assigned a label. Restricting the test set to samples with at least three agreeing expert labels raised these to $(78\%, 84\%)$, $(74\%, 92\%)$, and $(82\%, 88\%)$. It can be observed that many samples misclassified by the algorithms were also difficult to classify for the experts.

Zusammenfassung

Ziel dieser Arbeit war es, die Leistungsfähigkeit aktueller Supervised-Learning-Methoden anhand einer Bildklassifikationsaufgabe aus der Zellbiologie zu evaluieren. Insbesondere sollten Entscheidungsbäume, Support-Vector-Maschinen und Neuronale Netze untersucht werden. Die Aufgabe bestand darin, Mikrotubulienden aufgrund von Elektronentomographieaufnahmen als offen oder geschlossen zu klassifizieren. Mikrotubuli sind filamentartige Makromoleküle des Zellskeletts. Zellbiologen sind unter anderem an der Verteilung ihrer Endtypen interessiert, da dies Rückschlüsse auf die Orte der Selbstassemblierung zulässt. Im Augenblick wird die Klassifikation der Enden manuell von Experten durchgeführt. Diese Aufgabe wird insbesondere durch die verrauschten Bilddaten und die hohe Anzahl an Mikrotubuli in einer einzelnen Zelle stark erschwert. Sie zu automatisieren, würde die Klassifizierung sowohl effizienter machen als auch zu konsistenteren Ergebnissen führen.

Die Bilddaten der Mikrotubulienden stammen von Zellen im Stadium der Mitose, aufgenommen mittels Elektronentomographie. Als Ground-Truth-Daten wurden Klassifizierungen von je vier Experten verwendet, die alle von denselben Daten erzeugt wurden, die aus verschiedenen Tomographieaufnahmen zusammengestellt wurden. Für die Weiterverarbeitung wurde die Bildinformation um die Mikrotubulienden in verschiedenen Formaten aus den Tomogrammen herausgeschnitten.

Bei allen Klassifizierern wurde der Einfluss der Vorverarbeitung (Standardisierung pro Komponente oder Standardisierung pro Bild) auf die Qualität der Ergebnisse getestet. Für Entscheidungsbäume und Support-Vector-Maschinen wurden ebenfalls die Einflüsse durch unbalancierte und balancierte Daten untersucht. Letzteres wurde einmal durch Verwerfen von Überschussdaten und einmal durch das Einfügen von Kopien erreicht. Für diese zwei Klassifizierer wurde ebenfalls ihre Leistungfähigkeit bei bildbasierten Eingaben mit der von merkmalbasierten Eingaben verglichen. Die verwendeten Bildmerkmale wurden dabei eigens für diese Aufgabe entwickelt.

Die Ergebnisse zeigen, dass die Qualität der Klassifizierung von Entscheidungsbäumen und Support-Vector-Maschinen auf merkmalsbasierten Eingaben besser ist als auf bildbasierten Eingaben. Beide Klassifizierer behandeln die Endtypen am ausgewogensten und lieferten die besten Ergebnisse, wenn die Trainingsdaten durch Verwerfen der Überschussdaten ausbalanciert und vor der Merkmalsextraktion bildweise standardisiert wurden. Die verwendete Architektur der Neuronalen Netzes lieferte die besten Ergebnisse, wenn die Bilddaten nicht vorverarbeitet wurden.

Die finalen Ergebnisse der trainierten Entscheidungsbäume, Support-Vector-Maschinen und Neuronalen Netze erreichten auf den (offenen, geschlossenen) Endtypen die folgenden Genauigkeiten, wenn sämtliche Enden zum Testen verwendet wurden bei denen wenigstens ein Experte den Endtypen zuordnen konnte: $(62\%, 72\%)$, $(66\%, 70\%)$ und $(61\%, 78\%)$. Beschränken wir uns auf Enden, bei denen wenigstens drei Experten die gleichen Endtypen zuordnen konnten, so steigen die Genauigkeiten auf $(78\%, 84\%)$, $(74\%, 92\%)$ und $(82\%, 88\%)$. Bei Betrachtung der falsch klassifizierten Enden können wir feststellen, dass viele von den Algorithmen falsch klassifizierten Enden auch für die Experten schwer zu klassifizieren waren.

Conventions

We use the term 'we' to refer to the author and the reader. This pronoun is kept throughout the thesis including sections were 'I' would be more appropriate.
When referring to the training set, it is possible that duplicate samples are included. More formally we should refer to it as the training hyperset. Treating the set as a sequence also alleviates this problem but introduces an order to the elements. We use both training set and training sequence and allow duplicate elements when using the first and assume an arbitrary ordering when using the second. The word *significant* is used in its qualitative, informal sense and does not refer to *statistical significance* as in hypothesis testing. We write vectors in bold font and refer to its components by a lower index. For example, x_j is the j-th component of $\mathbf{x}$. The term $[n]$ stands for the first n nonnegative integers, $\{1, 2, \ldots, n\}$.

Contents

1 Introduction

In this thesis, we investigate a supervised image classification task from the domain of cell biology. Our main reference for cell biology is the seminal work by Alberts et al. [55]. Eukaryotic cells, that is cells having a nucleus, contain a multitude of various molecules and organelles, which are made up of various different molecules. The target of cell biology is to understand how local interactions between individual molecules, happening at the scale of a few nanometres, can cause behavioral and structural changes of the whole cell at the scale of several tens of micrometers [16]. Such local interactions are, for example, what enables white blood cells to move towards hostile microorganisms, muscles to contract, and cells to perform the complex cycle of cell division.

A major component that enables and organizes these interactions is the cyto- or cell-skeleton. This structural component serves a variety of purposes. It forms the shape of the cell, builds an internal network that serves as infrastructure for associated motor proteins, and enables the application of mechanical intra- and inter-cellular forces. During cell division, it forms the spindle apparatus which is responsible of pulling sister chromatids to opposite ends of the cell.

Opposed to its name, the cytoskeleton is not a fixed structure. It can be highly dynamic and is capable of reassembling within minutes. It is able to answer to stimuli like external forces or a rise in concentration of certain proteins. Three different types of filaments constitute this structure:

- microfilaments consisting of actin polymers,
- intermediate filaments with cell specific building blocks,
- microtubules consisting of tubulin dimers that assemble into a stiff cylindrical tube.

A cell structure that shows different behavior at opposing ends is said the be polarized. Microfilaments and microtubules are polarized. Both have a preferred end for assembly and disassembly. The end that grows and shrinks faster is called (+)-end, the other is called (−)-end. Goal of this thesis is the automatic classification of microtubule ends. In the following we give a short description of these filaments.

Tubulin dimers, the building blocks of microtubules, are already polarized. They consist of an α-tubulin protein bound to a β-tubulin. Several tubulin dimers can bind to one another, placing α-tubulin always next to β-tubulin forming a chain that is called *protofilament*. The regular pattern and the dimer's polarization cause these chains to be polarized as well. Microtubules consist of 13 laterally attached protofilaments that form a tube-like macromolecule. All protofilaments are arranged with (−)-ends at one side and (+)-ends at the other. Therefore, the whole microtubule again is polarized and has one end exposing only α-tubulin (the (−)-end) and the other only β-tubulin (the (+)-end). Current research in cell biology analyzes the distribution of microtubule nucleation sites [45]. This can be done by looking at the distribution of microtubules with a closed end morphology. These ends are capped by a protein to hinder (diss-)assembly, thus, create a stable end. While open ends can indicate both (+)-ends and (−)-ends, closed ends are believed to indicate (−)-ends. Electron tomography reconstructions have just

enough resolution to show the end morphologies. But the low signal-to-noise ratio due to the reconstruction process and the abundance of other proteins in the cell render the classification on these images a hard task, even for experts in the field. Figure 1.1 illustrates the wide quality range that can occur. The top row shows ends with clear end morphology. In the bottom row we see examples that are harder or impossible to classify. Some specimen show indicators of both classes, others suffer from too much noise to be recognized as ends at all.

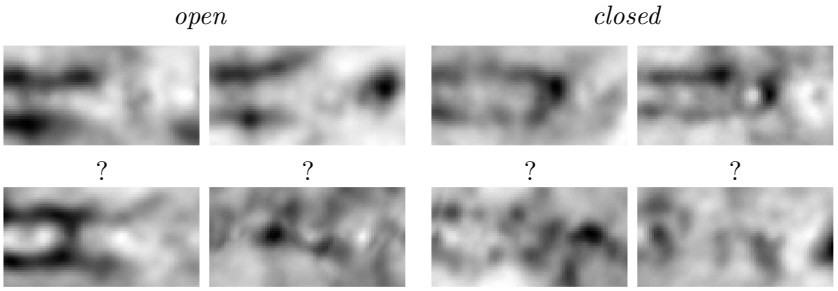


Figure 1.1: Images of microtubule ends obtained by electron tomograpy.

Currently, classification is done manually. Considering that there can be several tens of thousands microtubules in a single cell, this is a lengthy and error prone task and the results vary greatly between different experts. Automating this task would be beneficial to both efficiency and consistency. Therefore, in this thesis we explore the performance of three popular supervised machine learning methods, decision trees, support vector machines, and neural networks, on the task of microtubule end classification.

In this context, we design tools that ease the manual classification process and allow to gather and extract image information from 3D electron tomography data for further processing. Training and test data is collected by having multiple experts classify the same datasets comprising data from different tomograms. The dissimilar frequencies of distributions of end types lead to imbalanced training data. We account for this by comparing the performance on under- and oversampled datasets to the imbalanced case. Decision trees and support vector machines were not particularly designed to classify in the image space, as they rely on input components to hold the same semantic. We give arguments that, in the present case, it still might be interesting to test classification on images. Additionally, we design features that were extracted from the images and compare the performance of feature- and image-based learning. For the feature-based input we also check whether supplying projections onto the training set's most dominant principle components improves performance. For these classifiers we also inspect the results from training on smaller, resampled images. When exploring the parameters for each classifier, we systematically search for the most promising training set-up. While doing so, we address performance fluctuations by executing multiple runs for every configuration and considering the average values as well as the spread. We conclude with a comparison of the performance of all three final classifiers on the test set.

In summary, we made the following contributions:

1. To obtain ground truth data, we designed and implemented a tool to ease the manual classification process.
2. To obtain training data, we designed and implemented a tool to gather and extract image information around microtubule ends from tomography reconstructions.
3. We prepared training and test data that was classified by four domain experts.
4. We thoroughly reviewed the theoretical basis for the three machine learning methods decision tree, support vector machine, and neural networks.
5. We applied the three methods to the task of image classification. During the search for well performing parameter configurations, we did the following.
   - We accounted for the statistical variation by performing several runs for each parameter configuration.
   - We systematically explored the effect for each tested parameter, such as the class imbalance, preprocessing technique, input format, or those specific to the algorithms.
6. We designed three image based on the difference of per class averages.
7. Additionally to resampled images, decision trees and support vector machines were also trained on this feature-based input. Neural networks were only applied to images.
8. We thoroughly analyzed the results of all three types of classifiers and compared them with respect to the image classification task investigated in this thesis.

This thesis is structured as follows. In Section 2, we introduce the preliminary necessities. These include the classification and extraction tools, ground truth data, preprocessing and extracted features. Afterwards we start with the actual classification. Decision trees are handled in Section 3, support vector machines in Section 4, and neural networks in Section 5. Each section for a classifier follows the same pattern. We start with an introduction in which we review the theoretical foundations. This is followed by the methods section in which we explain and evaluate the search for a well performing set-up. Lastly, we train the final classifier and report the results. We finish this thesis with a comparison of the performances of all classifiers in Section 6 and a discussion of the findings in Section 7.

2 Data Acquisition, Preprocessing, and Preliminaries

In this section we describe the process of data acquisition and preprocessing. We describe the start set-up of this thesis in Section 2.1. In Section 2.2, we introduce the two tools that were developed to support the manual classification and allow the extraction of image information. Afterwards, in Section 2.3, we look at the expert classification that constitutes our ground truth data. In 2.4 we explain the applied preprocessing techniques and the features we came up with for the decision tree and the support vector machine. Finally, we introduce the performance measure and sampling strategies to handle the class imbalance in Section 2.5.

2.1 Initial Datasets

Starting point of this thesis is a collection of datasets. Each collection contains two elements. One is a 3D scalar field T containing the image data obtained by electron tomography the other is a graph embedded into 3D space tracing the microtubules detected in T. We will give a short description of both.

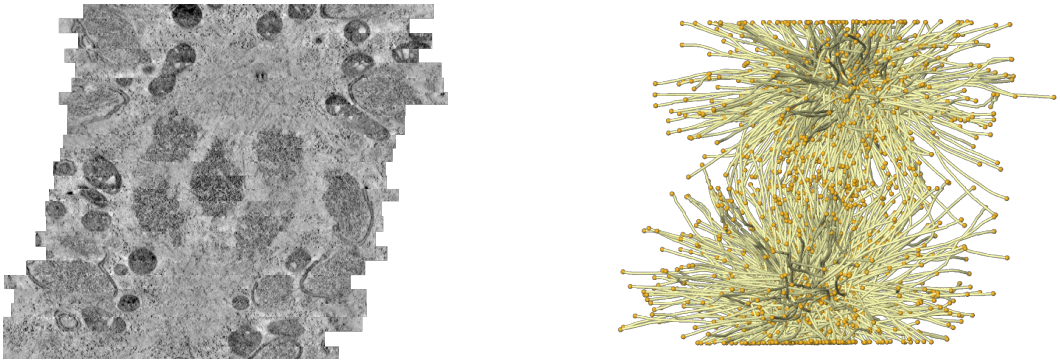


Figure 2.1: Left: Side view a stack of individually reconstructed tomography sections that were stitched together. Right: Embedded microtubule graph that traces the detected structures.

Tomography Data: The tomography data is given by a 3D scalar field and shows sections of an embryonic cell of the worm C. elegans during cell division. Some records show a complete spindle apparatus. The datasets were generated by the Müller-Reichert Lab from the Core Facility Cellular Imaging at Technische Universität Dresden.

In serial section electron tomography, the object of interest is cut into several thin slices each of which is reconstructed individually. The reconstruction is computed from a set of projection images obtained by transmission electron microscopy (see Fig. 2.2). The images are taken from different directions which is achieved by

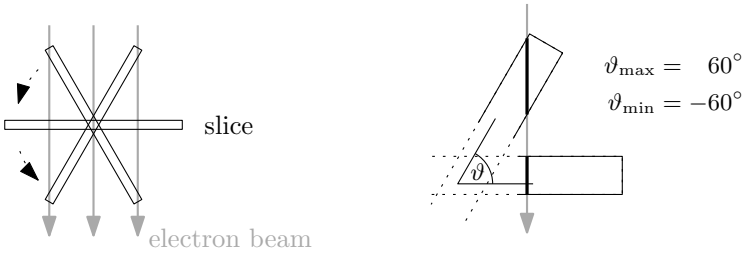
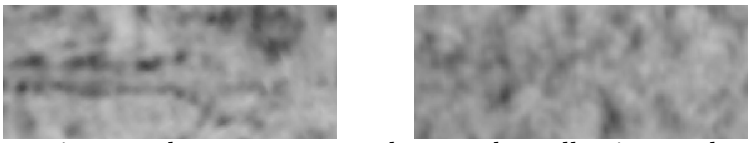


Figure 2.2: Left: In (tilted) electron tomography, a slice of the specimen is reconstructed from several images taken while tilting the object under consideration around a fixed axis. Thus, the images show projections from different perspectives. Right: Usually, the magnitude of the tilting angle ϑ will not surpass 60° deg as the increasing path length to fully penetrate the slice requires electron beams with destructive energy.

tilting the slice around a fixed axis relative to the direction of the electron beam. Tilting angles in which the angles between slice normal and the electron beam surpass a given threshold (usually 60°) are avoided since the necessary penetration depth increases and eventually requires electron beams with enough energy to destroy the specimen. The missing orientations cause a loss of information in the reconstruction. It can be shown that this loss has the form of missing frequency information in the Fourier transform of the reconstruction. The area without frequency information has a wedge like shape and is known as *missing wedge effect*. This effect results in non-isotropic effects (missing information) in the reconstruction. We denote the direction which suffers the most under this as the z-direction of the tomography data. As an illustration, the left image below shows the cross-section perpendicular to the z-axis of an endpoint; the right image shows the same endpoint from a cross-section parallel to the z-axis.



For more information on electron tomography see the collective work edited by Frank [18]; in particular chapter 10 by Penczek and Frank, for the missing wedge effect.

The datasets under consideration consist of a stack of reconstructed slices that were stitched together (see Fig. 2.1, left). Further information on how the slices were joined can be found in the work of Weber et al. [59].

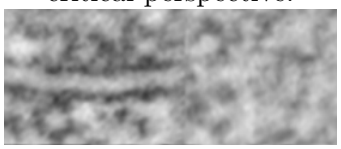
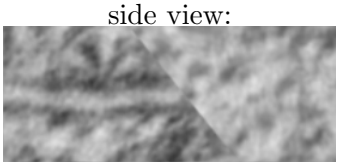
Microtubule Graph: The microtubule graph is a graph embedded into 3D space. Each vertex is mapped to a point in 3D space, and each edge is mapped to a polygonal chain (see Fig. 2.1, right). The endpoints of the polygonal chain coincide with the images of the corresponding edge vertices. When we overlay the tomography data with the microtubule graph, the latter traces the center-lines of all detected microtubules. Due to the structure of an isolated microtubule, the abstract graph is a union of paths, and the embedding of the vertex set locates the microtubule endpoints that we are interested in.
For more information on the tracing, see the work of Weber et al. [58].

Prior to the start of this thesis, there were about 2000 endpoints that had been labelled

as being *open* (label *open*) or *closed* (label *closed*), by experts in the field. This set was intended to form the labelled data for the supervised learning approach taken in this work. Further analysis of the data revealed some issues:

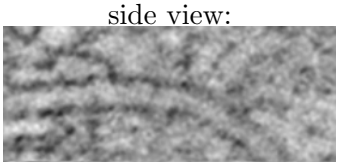
- Some detected and labelled endpoints revealed to be artificially created microtubules ends that were caused by the physical slicing and virtual reassembly of the specimen. From a different perspective the different tomography slices and the seam between them can easily be recognized.

  critical perspective: side view:

- Some endpoints turned out to be unusually bent microtubules that left the visualized plane. Again, an different perspective reveals this phenomenon.

  critical perspective: side view:

- Some endpoints appeared to be in the wrong class.

These problems indicated that it was necessary to improve the manual classification process. The old procedure restricted the expert to a single perspective. A mental 3D image of the endpoint morphology had to be created in mind by sliding through a stack of 2D images

2.2 Two New Tools

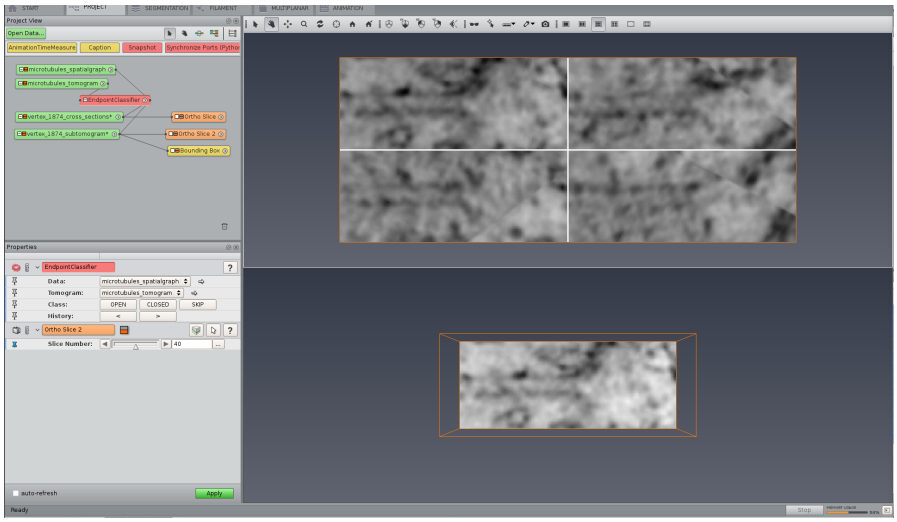


Figure 2.3: Snapshot of the classification tool implemented in the Amira software. The top viewer window shows four different cross-sections of an endpoint that is eligible for classification. The bottom window shows a 3D volume around the endpoint.

Supervised learning requires labeled ground truth data that can be fed to the learning algorithm during the training phase. To allow and efficient manual classification of microtubule ends, we developed a new classification tool. It was implemented as a compute module, called `EndpointClassifier`, in the visualization software *Amira*[1] [54]. Figure 2.3 shows a snapshot. The module is attached to a microtubule graph and the corresponding tomography data. The tool automatically filters microtubule endpoints that are too close to the border of the scalar field or to seams of adjacent tomography slices (see Fig. 2.4).

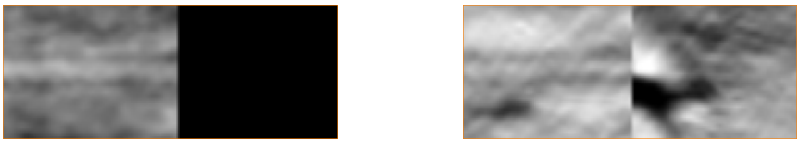


Figure 2.4: Examples of microtubule endpoints that are filtered out automatically. Left: Microtubule end that is caused by reaching the border of the tomography data. Right: Microtubule end that is caused by a seam between adjacent slices.

The remaining endpoints are traversed in order of their enumeration. For each, the tool extracts two datasets from the tomography data – an image that shows four different cross-sections of the endpoint and a 3D sub-volume centered at it. Both datasets are aligned with the end part of the microtubule. Further explanation is given in Figure 2.5.

Figure 2.5: The two datasets that are extracted and displayed for every endpoint that is eligible for classification. The exact generation is described in the section for the second tool. Left: One half of the sub-volume centered at the microtubule end. The color planes indicate the orientation of the cross-section that form the second dataset. Right: The second dataset consists of four different cross-sections. The color indicates their orientation compared to the sub-volume. The cross-section at the bottom right usually suffers the most from the missing wedge effect.

A user interface allows to assign one of the two labels *open* or *closed* to the endpoint. Alternatively the endpoint can be skipped in which case the label *undefined* is set implicitly. Further, the user is allowed to navigate through endpoints that were classified in the current session and to assign a new labels. When navigating through already classified endpoints, the currently assigned labels are not shown, and any new assignment

[1] Amira is a registered trademark owned by Thermo Fisher Scientific. It was originally developed at Zuse Institue Berlin, where an internal research version is still collaboratively developed and allows the implementation of new packages.

has to be done based on the image information alone.

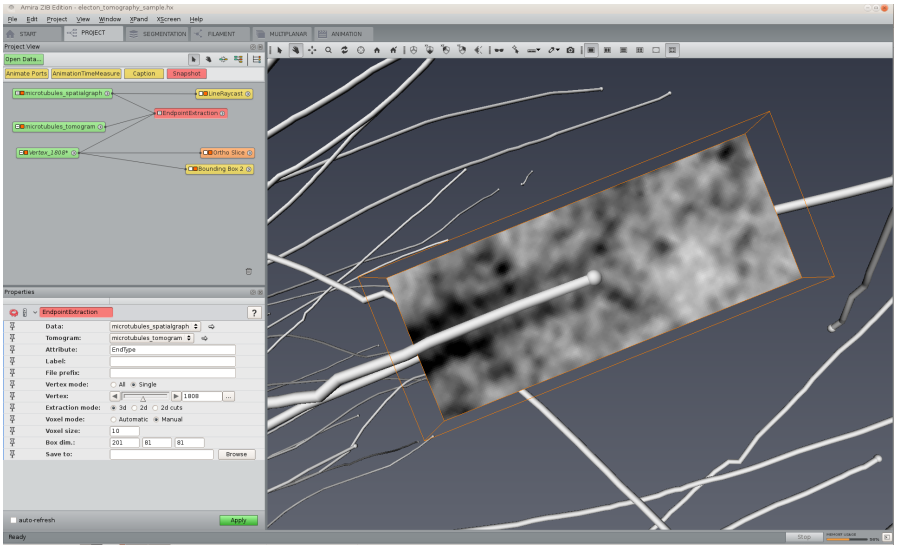


Figure 2.6: Snapshot of the extraction module applied in the Amira software. The viewer window shows parts of a microtubule graph and an aligned 3D box around a microtubule endpoint. One mode of the tool allows the extraction of these aligned boxes.

The second tool, `EndpointExtraction`, was also implemented in Amira. For a snapshot, see Figure 2.6. Similar to the classification tool, it is attached to a microtubule graph and the corresponding tomography data. It allows the automatic extraction of image information from the tomography data around microtubule ends. The specific extraction format can be chosen from the three following modes.

3D data: In this mode a box shaped 3D sub-volume will be extracted that is centered at the microtubule endpoint. The number of voxels in the box can be set by hand. The orientation is found as follows (see Fig. 2.7 for an illustration). The three axes of the box follow the right-hand-rule. The first axis points into the same direction as the end part of the microtubule. This direction is defined by the difference vector $(\mathbf{y} - \mathbf{x})$ of two points of the embedded microtubule graph. The target point $\mathbf{y}$ is the endpoint of the corresponding microtubule and the starting point $\mathbf{x}$ is at the intersection of the polygonal chain that represents the microtubule and a ball around $\mathbf{y}$ with radius equal to half the length of the first side of the box. The second axis is perpendicular to the first and to the z-axis of the tomography data. Remember that the z-axis suffers the most from the missing wedge effect, hence, slices from the sub-volume that are parallel to the first and second axis will show the least noise. The third axis is fully defined by the first two and the right-hand-rule.

The output voxel will have cubic shape with a side length that can be either selected by hand or computed automatically. In the latter case, cube shaped voxel with the same volume as the voxel in the original dataset will be used. The tomography data is sampled once for each output voxel, at its center location.

Single 2D image: In this mode a single image is extracted. It is identical to a slice of

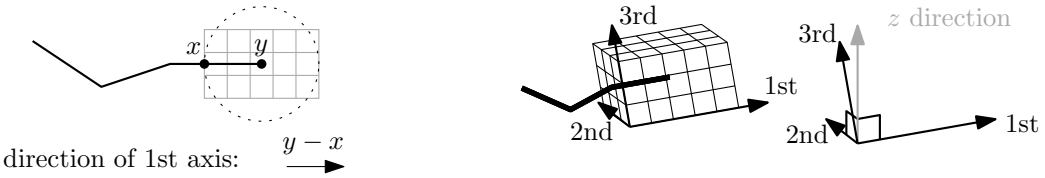


Figure 2.7: Orientation of the extracted 3D box. Left: The direction of the first axis is defined by the location of the endpoint and the intersection of the microtubule with a ball of radius equal to the radius of the box along the first dimension. Right: the second axis is chosen to be perpendicular to the z-axis of the tomography data and the first axis. The third axis is perpendicular to the first and follows the right-hand-rule.

the box in the previous mode, that is perpendicular to the third axis of the box and intersects the center (see Fig. 2.8).

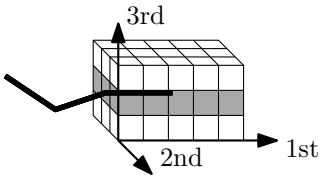


Figure 2.8: Extraction of a single 2D image corresponds to extraction of the grey slice.

Multiple 2D images: This mode allows the user to specify a power of two as the number of 2D images that will be extracted. If a single image is requested, it will be arranged as in the mode above. The arrangement for 2^i images can be obtained by taking the arrangement of 2^{i-1} images, copying it, and rotating the copies by an angle $\pi/2^{i-1}$ around an axis A that has the same direction as the microtubule end and runs through it. In other words, all images share one intersection axis, and when we look along this axis into the microtubule, images and microtubule end are arranged as ⊖, ⊕, ✳, . . . , for $i = 0, 1, 2, \ldots$.

Opposed to the previous extraction modes, each output pixel shows the average value of a set of sample values taken from the tomography data. The number n of sample values can be specified manually. For each output pixel v, the n samples will be taken from an arc of the circle c centered at A, perpendicular to it, that runs through the center of v (see Fig. 2.9). The circle will intersect centers of 2^{i+1} pixels simultaneously, and the corresponding $n2^{i+1}$ sample values are arranged evenly spaced on it. For odd n, one sample value will lay at the center of the pixel, and for $n = 1$, this mode returns cross-sections of the tomography data. In fact, the four different cross-sections of the classification tool can be obtained by the setting $i = 2$ and $n = 1$.

The tool allows the extraction of either all endpoints, only those that share a specific label (set by the user), or a single endpoint.

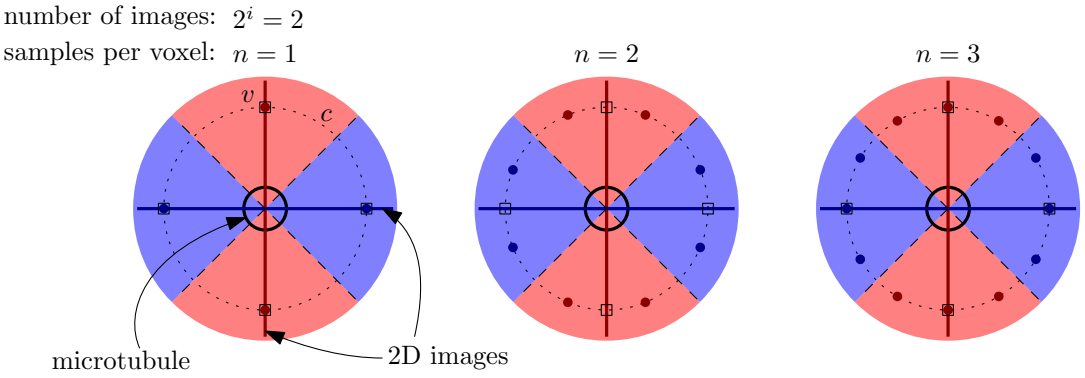


Figure 2.9: Computation of a pixel value in the last extraction mode. Sketched is the top view along axis A, into a microtubule. This view is parallel to the image planes which appear as line segments. An output pixel shows the average of n sample values arranges on a circular arc, that is, the red (blue) image will give an average of the red (blue) region. Illustrated are the settings $i = 1$ (2 output images) and $n = 1, 2, 3$.

2.3 Getting Labeled Data

The labeled data was obtained with the classification tool described in the previous section. To gain insights on the deviation between different experts, we asked four cell biologists to classify identical datasets. We shuffled the endpoint enumeration prior to the classification since this defines the order in which the classification tool selects the endpoints. The intention was to prevent any effects that could be caused by a systematic traversal of areas in the microtubule graph. One example scenario is that closed ends might appear clustered in the original microtubule graph. The occurrence of a closed end might than increase the willingness of a human classifier to assign the same label to the next endpoint e, while, if e had occurred in a series of open endpoints, the tendency might have been for the *open* label. To prevent the use of such context information and enforce that everything that led to a decision can be found in the images alone.

The final label of an endpoint. We have multiple labels for every endpoint, one by each expert. We treated the labelling as a vote on the final class. An endpoint with at least one vote for *open* and one vote for *closed* is labelled *contradictory* and excluded. For the remaining endpoints the final label is obtained by majority vote.

The datasets were obtained in two sessions. Results from the first session were used as training data. Results from the second session were used as test data. In the second session, the focus was on variation in quality of the tomography data.

Training data. In the first session, three datasets ds1, ds2, and ds3, with approximately 4100, 2300, and 1000 endpoints (prior to the filtering of the classification tool), respectively, were selected. The experts were asked to classify the first half of every dataset, and, if time permits, continue with the second halves (from ds3 down to ds1). This resulted in 982, 230, and 128 endpoints in the respective datasets that were labelled by each of the experts as either *open*, *closed*, or *undefined*.

Table 2.1 shows the number of endpoints in each category and dataset. Table 2.2 lists the number of endpoints in the union of ds1, ds2, and ds3 for all possible assignments of four votes in $\{open, closed, undefined\}$. The left chart in Figure 2.10 shows statistics on the labelling behaviour for every expert and dataset.

dataset	#*open*	#*closed*	#*undefined*	#contradictions
ds1	982	230	308	143
ds2	148	39	79	52
ds3	89	12	27	22
total	1219	281	414	217

Table 2.1: Number of endpoints per label and dataset obtained by majority vote of four experts.

	#votes for *closed*				
#votes for *open*	0	1	2	3	4
0	197	110	72	59	40
1	255	130	31	7	
2	409	39	4		
3	320	6			
4	235				

Table 2.2: The number of endpoints in the union of ds1, ds2, and ds3 for every possible combination of votes. The entry at $(0, 0)$ gives the number of undefined endpoints. The entries at $(0, 4)$ and $(4, 0)$ are the unanimously voted endpoints. Entries at (i, j) with $i, j > 1$ are contradictory.

Restriction to endpoints that were unanimously voted open or closed would have reduced the number of usable samples to 235 and 40, respectively. Considering all endpoints that were not labelled contradictory increases these numbers to 1219 and 281 (see Table 2.1 or, equivalently, the sum of all entries $(i, 0)$ and $(0, i)$ in Table 2.2, for $i = 1, 2, 3, 4$). Since this is still little data for some algorithms, we further relaxed the restrictions by considering all endpoints that were labelled by at least one expert. If only a subset of the experts participated in the labelling of a specific endpoint, the label *undefined* was assumed for the remaining experts. The results can be seen in Tables 2.3 and 2.4. As is to be expected, the number of unanimously labelled endpoints remains the same, but considering all non-contradictory labels further increases usable samples to 1650 (#*open*) and 424 (#*closed*).

dataset	#*open*	#*closed*	#contradictions
ds1	1310	340	145
ds2	217	64	52
ds3	123	20	27
total	1650	424	224

Table 2.3: Number of endpoints per label and dataset obtained by majority vote of at least one expert. The number of undefined endpoints are not listed we artificially increased them (see text).

Test data. In the second session, four datasets qs1, ..., qs4 (for tomography *q*uality

	#votes for *closed*				
#votes for *open*	0	1	2	3	4
0	5158	236	88	60	40
1	593	136	32	7	
2	495	39	4		
3	327	6			
4	235				

Table 2.4: The number of endpoints in the union of ds1, ds2, and ds3 for every possible combination of votes. Here, all endpoints are considered that were seen by at least one expert.

*s*ample) were prepared analogously to the training data. The dataset selection was intended to cover a broad range of tomogram qualities based on the self assessment of the person that performed the tomography and a test classification of the author. Again, four experts were asked to classify ∼500 endpoints in every dataset. In this section every endpoint under consideration was labelled by each of the four experts, thus, has exactly four votes. The results can be seen in Tables 2.5 and 2.6. It turned out that an ambitious experts classified slightly more than was necessary. We indicated these additional endpoints by with a plus sign to prevent inconsistencies when we later evaluate the classifier performances. The right chart in Figure 2.10 gives summarizing statistics on the labelling behaviour for every expert and dataset for the second session, similar to the chart on the left.

dataset	#*open*	#*closed*	#*undefined*	#contradictions
qs1	302	77	121	53
qs2	280	98	122	31
qs3	176	83	241	10
qs4	252	70	156	26
total	1010 +15	328+8	640	120

Table 2.5: Number of endpoints per label and dataset obtained by majority vote of four experts. In brackets, we added the number of endpoints that were obtained by a single expert surpassing the agreed number of total endpoints to classify. We accompany these numbers to prevent inconsistencies during the test phase.

2.3.1 Extracting Endpoint Data

The input data for the learning algorithms was generated with the second tool. We extracted image information around every endpoint in five different formats, resulting in five different input versions. Later, the most promising input format is chosen for every algorithm. The `3d` format was dropped quickly due to the high dimensionality of the input vectors ($81 \cdot 41 \cdot 41 = 136161$).

	#votes for *closed*				
#votes for *open*	0	1	2	3	4
0	520	205+8	74	29	20
1	528+13	77	20	4	
2	249+1	15			
3	138+1	4			
4	95				

Table 2.6: The number of endpoints in qs1,..., qs4 combined for every possible combination of votes. In brackets, we added the number of endpoints that were obtained by a single expert surpassing the agreed number of total endpoints to classify. We accompany these numbers to prevent inconsistencies during the test phase.

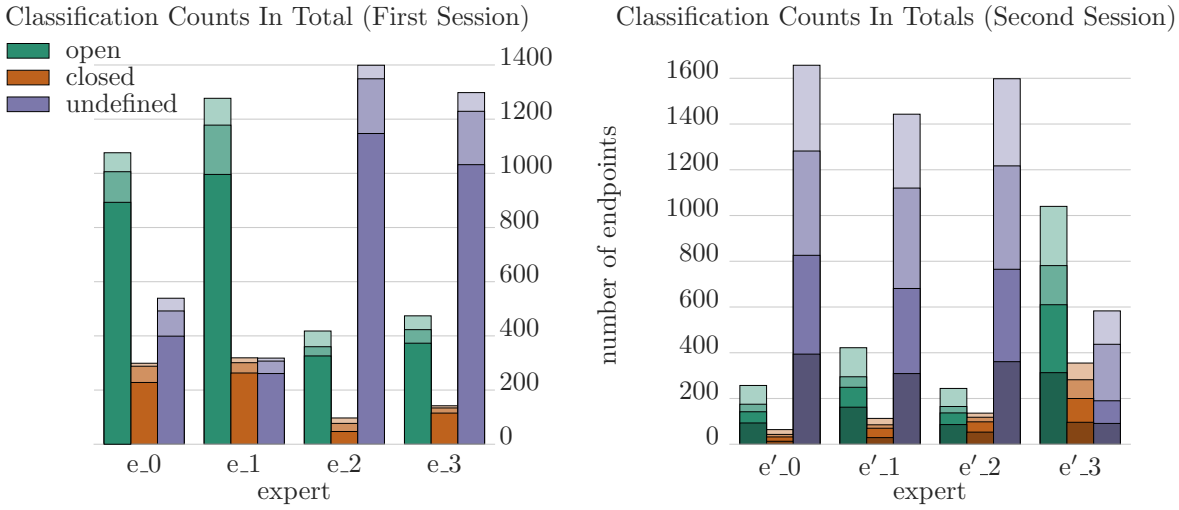


Figure 2.10: For each expert the number of votes for *open*, *closed*, or *undefined* per dataset (datasets ds1, ds2, and ds3 on the left, qs1, qs2, qs3, and qs4 on the right). Both charts only consider those endpoints that were labelled by four experts. Brighter versions of the same color indicate the next dataset. The strong deviations in the labelling counts could be indicators of the difficulty of the labelling process. For example, the first two experts in the left chart skipped significantly fewer endpoints, and in the right chart the fourth expert labelled more than twice as many endpoints than each of the others.

identifier	description
`3d`	A sub-volume with dimension 81×41×41.
`2d_slice1_sample_1`	Single 2D cross-section image with dimension 81×41.
`2d_slice4_sample_1`	Four 2D cross-section images each with dimension 81×41.
`2d_slice1_sample_64`	Single 2D image with dimension 81×41 that shows the average of a cylindrical volume around the end.
`2d_slice4_sample_16`	Four 2D images each with dimension 81×41 that show averages of segments a cylindrical volume around the end.

Table 2.7: The five different input formats.

2.4 Data Preprocessing

Before we feed data to a learning algorithm, it is advisable to perform some kind of normalization to circumvent effects that might result from variations in the data generation process. For example, the average brightness might differ from tomogram to tomogram or between different areas of the same tomogram. Other normalization techniques might scale feature ranges so that each feature displays similar distributions. Here, the intention is to equalize the importance of distinct features. Without such preprocessing, it might occur that the range of one feature is several orders of magnitude larger than the range of other features. For many distance-based learning algorithms this single feature would exclusively steer the learners behaviour, as the distance of two samples would only insignificantly depend on the remaining feature values. In this thesis, we compare two normalization techniques.

First, we arrange the input samples in a matrix X, and store the individual samples as rows. If we have n input samples, each comprising m features, then X will have shape $n \times m$ with entry $x_{i,j}$ denoting the j-th feature value of the i-th sample. The first technique normalizes the features, we call it *per-feature standardization*; the second normalizes the samples, hence, we call it *per-image standardization*. As the name suggests, both perform standardizations on the corresponding values, that is, we interpret the values in a row (or column) as instantiations of a random variable that we transform such that it has zero mean and unit standard deviation.

Formally, the per-feature standardization can be described by m transformations $(\varphi_j)_{j=1..m}$, defined as

$$\varphi_j(x) = \frac{x - \mu_j}{\sigma_j},$$

$$\text{where} \quad \mu_j = \frac{1}{n}\sum_{i=1}^{n} x_{i,j} \quad \text{and} \quad \sigma_j = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_{i,j} - \mu_j)^2}.$$

We obtain the per-feature standardization of X by applying φ_j to the j-th column of X. The effect of this technique is illustrated in Figure 2.11. A technicality we need to keep in mind is that the m transformations φ_j will have to be computed on the training set exclusively, that is, above, n referred to the number of training samples. These φ_j will then be applied to both the validation and test set.

The per-image standardization is obtained by exchanging the columns for rows in the definition above. An advantage of the second technique is that it can be readily applied to any input sample regardless whether it belongs to the training, validation or test set.

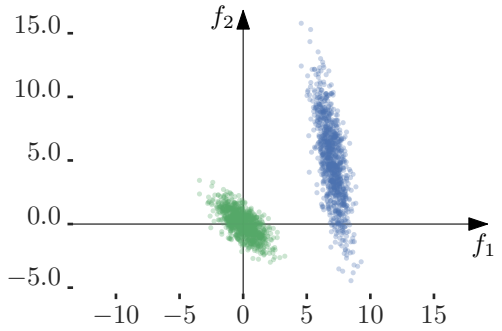


Figure 2.11: The effect of applying per-feature standardization. Both features f_1, f_2 of a set of 2-dimensional samples (blue) are transformed. In the resulting set (green), both features have zero mean and a standard deviation of one.

Opposed to this, performing per-feature standardization on the whole set of labelled data would introduce validation and test information to the algorithm at training time. For an illustration of the effects of per-image standardization consider the two endpoint images $\mathbf{x}_1$ (left) and $\mathbf{x}_2$ (right) from different tomograms.

Figure 2.12 shows the histograms for both images prior to the transformation and after. We see that the mean of both images was shifted to the origin. Moreover, the standard deviations of both images are now close to one resulting in a change of the overall shape of the histogram.

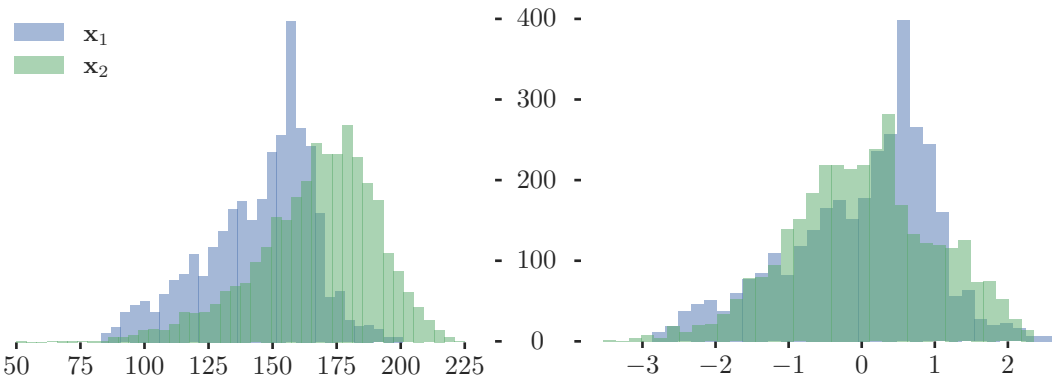


Figure 2.12: The effect of per-image standardization. Left: Histograms of two input samples $\mathbf{x}_1$, $\mathbf{x}_2$ prior to the preprocessing. Both samples have individual mean and standard deviation. Right: Histograms after preprocessing. Both samples share the same mean and standard deviation.

Independent of these two transformations we evaluate the performance of some algorithms on downsampled versions of the original image data. A motivation is given in the next section. As suggested in chapter 4 of the book by Forsyth and Ponce [17] we

smooth the image prior to the resampling with a Gaussian filter[2]

2.4.1 Feature Extraction

Most machine learning algorithms were not specifically designed for image data as input. Instead, they operate on feature vectors and rely on foxed feature dimensions to hold consistent information across all samples. If for one input vector the j-th feature describes, for example, the weight of the corresponding sample, then it should not describe the height for another. The learning algorithm would interpret it as having the same semantic (unless there exists another feature that indicates how the j-th component should be interpreted). In general, we do not have this kind of *semantic consistency of input components* for images. A pixel could describe the foreground in one image and the background in the next. Even if we can safely assume that a pixel shows part of the foreground, as is often the case for pixels in the center of an image, we still could not say which part of the object of interest is captured.

The usual approach for images, thus, is to extract features from them, combine the features in a fixed order to a vector, and feed this feature vector to a learning algorithm. Natural images usually contain recognizable gradient structures like edges or corners, and popular feature descriptors (for example the *histogram of oriented gradients* or *shift invariant feature transform*) exploit this. The tomography data lacks the necessary resolution for this kind of structures. The images usually have a 'washed out' look and most images show no recognizable edge or corner structures. Additionally, noise inherent in the tomography reconstruction process and from the unwanted recording of ubiquitous cell structures further complicate the design of reliable image features. We came up with two (similar) features whose performance will be tested for learning algorithms that are not natively designed for images. A description follows further below.

Notwithstanding the exposition above, the specific images we are dealing with here might be suited as input even for feature vector based algorithms. The way we extract the endpoint information from the tomogram leads to registered images. Depending on the quality of the microtubule tracing, the endpoint is in close proximity of the image center, and the extraction tool aligns the first image axis with the microtubule direction. Moreover, the structural similarity of all microtubules causes a fixed diameter of the filament wall. This can be interpreted as a weak version of semantic consistency of input components. The meaning (for example, interior of the filament) of a pixel p in one image must not correspond to the meaning of the exact same pixel in another image, but it is likely that it corresponds to a pixel in the neighbourhood of p. This motivates downsampling as a preprocessing step. We meld local neighbourhoods to a single representative value.

Feature description

We now describe the two features that we designed. Both are based on the *average image* of a class. Let X be defined as above and $\mathbf{y} \in \{0,1\}^n$ be the corresponding class vector, with $y_i = 1$ iff the i-th row vector $\mathbf{x}_i$ of X belongs to the class *open*. (We implicitly assumed that the images, usually given in matrix or tensor form, are flattened into an m-dimensional vector.) The average open image $\overline{\mathbf{o}}$ and the average closed image $\overline{\mathbf{c}}$ are

[2] In Ref. [17] an image is resampled to produce a result with half the side length of the original image.

then defined as

$$\overline{\mathbf{o}} = \frac{\sum_{i=1}^{n} y_i \mathbf{x}_i}{\sum_{i=1}^{n} y_i} \quad \text{and} \quad \overline{\mathbf{c}} = \frac{\sum_{i=1}^{n} (1 - y_i) \mathbf{x}_i}{\sum_{i=1}^{n} (1 - y_i)}. \tag{2.1}$$

Again, $\mathbf{x}_i$ refers to the i-th row vector in X. Equation (2.1) allows us to define the first feature $f_1(\mathbf{x})$ of a sample image $\mathbf{x}$ as the similarity to $\overline{\mathbf{o}}$ or $\overline{\mathbf{c}}$, expressed by the scalar product. We can condense the two similarities into a single number by introducing the difference of the averages $\overline{\mathbf{d}} = \overline{\mathbf{o}} - \overline{\mathbf{c}}$. The scalar product of $\mathbf{x}$ and $\overline{\mathbf{d}}$ indicates whether $\mathbf{x}$ closer resembles $\overline{\mathbf{o}}$ or $\overline{\mathbf{c}}$:

$$f_1(\mathbf{x}) = \mathbf{x}\overline{\mathbf{d}} = \mathbf{x}(\overline{\mathbf{o}} - \overline{\mathbf{c}}) = \mathbf{x}\overline{\mathbf{o}} - \mathbf{x}\overline{\mathbf{c}}. \tag{2.2}$$

That is, the more $\mathbf{x}$ resembles $\overline{\mathbf{o}}$, the larger $f_1(\mathbf{x})$ becomes, and the more it resembles $\overline{\mathbf{c}}$, the smaller $f_1(\mathbf{x})$ becomes. When this is performed on images that were per-image-standardized, then we can interpret f_1 as a comparison between the *Pearson correlations* of the sample image and the two class averages. Figure 2.13 shows examples for $\overline{\mathbf{o}}$, $\overline{\mathbf{c}}$, and $\overline{\mathbf{d}}$ generated from a subset of dataset ds1.

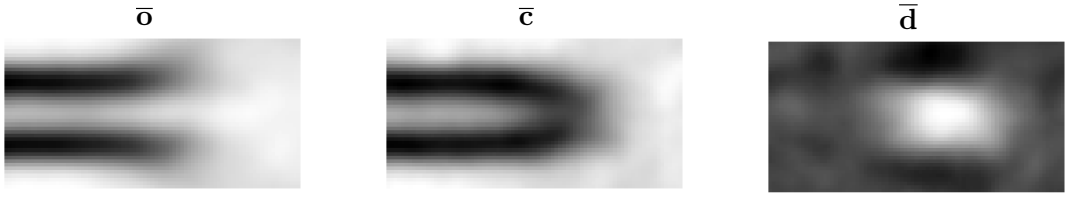


Figure 2.13: Average images $\overline{\mathbf{o}}$ and $\overline{\mathbf{c}}$ of each class for a training subset of dataset ds1 and the difference image $\overline{\mathbf{d}} = \overline{\mathbf{o}} - \overline{\mathbf{c}}$. The image $\overline{\mathbf{o}}$ was generated from 1210 samples, $\overline{\mathbf{c}}$ was generated from 240 samples.

The plot in Figure 2.13 visualizes the effect described in Equation (2.2). It shows the distribution of samples over the value of feature f_1 for an evaluation subset of ds1 comprising 100 samples per class. It also visualizes that the naive threshold of $f_1(\mathbf{x}) = 0$ that could be derived from Equation (2.2) does not perform optimally since the two classes do not behave symmetrically with respect to the feature. The mean of the closed evaluation samples is further away from the origin than the mean of the open evaluation samples (-207 vs. 81). Further, the open samples seem to have a slightly larger standard deviation than the closed samples (165 vs 160). Therefore, even for a single feature, it might be beneficial to learn the threshold, at which to separate the classes.

The second feature is computed similar to the first, but after projecting the input along its first axis onto its second (for 2-dimensional images) or onto the plane spanned by second and third axis. This can be thought of as looking from the top into the microtubule and merging the values from all pixels (voxels) that lie on top of each other, with a merging function φ. There are several possibilities to chose φ. We chose the minimum function and the mean. A 3-dimensional image $\mathbf{x} = (x_{ijk})$ with $i \in [n_1]$, $j \in [n_2]$, and $k \in [n_3]$, hence, will be mapped to a 2-dimensional image $\mathbf{x}'$ with shape $n_2 \times n_3$, by

$$x'_{j,k} = \varphi(x_{1jk}, x_{2jk}, \ldots, x_{n_1 jk}), \quad \text{for} \quad j \in [n_2],\ k \in [n_3], \tag{2.3}$$

$$\text{with} \quad \varphi(\cdot) = \min(\cdot) \quad \text{or} \quad \varphi(\cdot) = \text{mean}(\cdot). \tag{2.4}$$

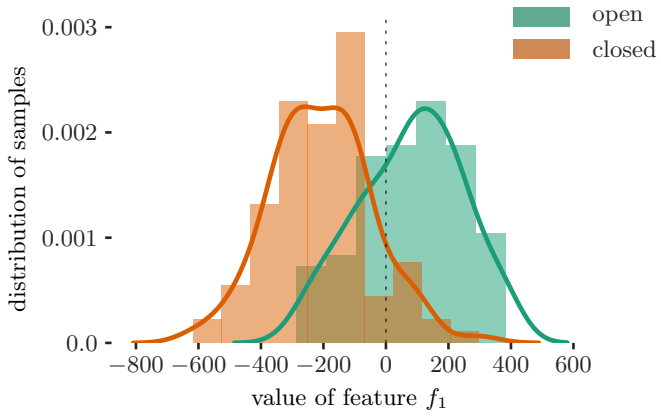


Figure 2.14: Density of open and closed samples for values of the first feature f_1, as estimated on an evaluation subset of ds1.

Every image will be processed this way, and, subsequently, we continue as for feature f_1. Figure 2.15 shows the resulting average images for projections, for both choices of φ. The images were generated on a training subset of ds1 (analogously to the previous average images). We call the resulting features f_2 (for $\varphi = \min$) and f_3 (for $\varphi = \text{mean}$).

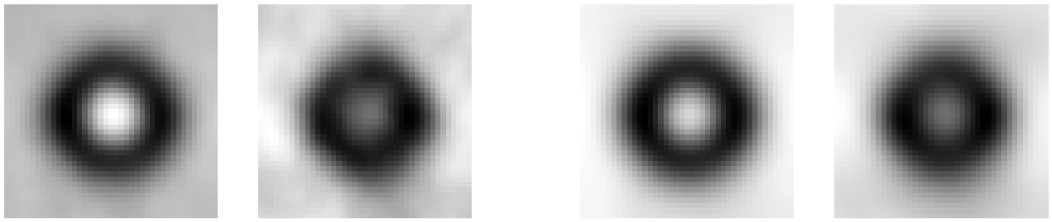


Figure 2.15: Average images after projecting the input samples along the first axis. From left to right, we have the average open and closed images for $\varphi = \min$, and the average open and closed images for $\varphi = \text{mean}$.

2.5 General Preliminaries

Before we start with the actual classification we need to introduce a few more general things.

2.5.1 Performance Measure

We have to consider the accuracies in two classes, but often it is handy to reduce them to a single number. Our interest is in classifiers with strong performances in both classes, and we want to avoid any inherent preference of one class over the other. Therefore, we simply stick to the smaller of the per-class accuracies as our measure, whenever we want a single number. We call this the *min-accuracy*.

2.5.2 Class Imbalance

As described in Table 2.3, the ratio of *open* to *closed* elements is roughly 4 : 1 for the training set. A classifier trained on such imbalanced datasets is likely to favour the

majority class. In the given problem setting, we try to achieve a balanced performance on both classes. Therefore, the initial phase of the exploration for each classification will consist of examining the behaviour with respect to the three following strategies.

(imbalance) Ignore the imbalance and use all the data at hand.

(undersampling) Replace the majority class with a sample of it, the same size as minority class, and drawn uniformly at random without replacement.

(oversampling) Increase the size of the minority class by repeatedly duplicating elements in it, until the size of the majority class is matched. Assuming that the number of open and closed elements is given by o and c, respectively, we copy the whole set of closed elements $\lfloor o/c \rfloor$ times and sample the remaining $o - \lfloor o/c \rfloor \cdot c$ elements uniformly at random without replacement.

3 Decision Trees

3.1 Basic Principles

Decision trees are classifiers that allow for a high level of interpretability. In their basic form they try to learn a set of binary decision rules in order to categorize all elements from the feature space. Depending on the final category into which a new sample falls, the decision tree outputs its prediction. Here, we only consider simple decision rules which boil down to projecting all data elements to a single feature space axis and checking to which side of some learned threshold a data element lies.

In this description, we follow the book by Friedman, Hastie and Tibshirani [19] which base their description on the seminal work of Breiman, Friedman, Stone, and Olshen [11]. Earlier work can be dated back to at least 1963 by Morgan and Sonquist [37]. For a historical overview, see Loh's survey [35].

Given is a training sequence $T = (\mathbf{x}_i, y_i)_{i=1..n}$ of elements from $\mathcal{X} \times \mathcal{Y}$, where $\mathcal{X}$ denotes the feature space and $\mathcal{Y}$ the set of possible labels, that were sampled according to some distribution $\mathcal{D}$ over $\mathcal{X} \times \mathcal{Y}$. We assume that $\mathcal{X}$ is a p dimensional feature space and $\mathcal{Y} = \{0, 1\}$. Let f_B be the *Bayes hypothesis* of the joint distribution $\mathcal{D}$, that is, out of all functions in $\{\mathcal{X} \to \mathcal{Y}\}$ the one that best approximates[1] $\mathcal{D}$.

The decision tree learning algorithm tries to find a piecewise constant function to approximate f_B. The returned estimator $DT\colon \mathcal{X} \to \mathcal{Y}$ partitions the feature space into axis-aligned rectangular regions R_i and returns a constant c_i for each x in R_i. See Figure 3.1 for an example of a possible feature space partition.

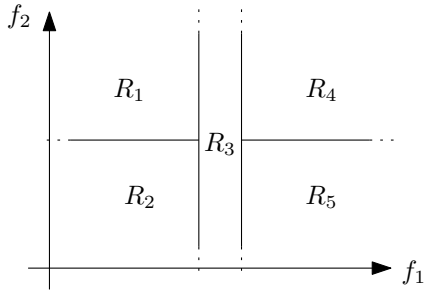


Figure 3.1: Illustration of a possible feature space partition due to decision tree. For each sample in region R_i, the tree will predict the value c_i.

This partition can be implemented in form of a binary tree, hence, we call the estimator the *decision tree*. We will use DT to refer to both, the estimator and the implementing tree structure. If we traverse this tree from the root to the leaves, each node further subdivides $\mathcal{X}$ until we have reached the final partition class. Specifically, each node v is associated with an axis-aligned rectangular region $R_v \subseteq \mathcal{X}$ and an axis-aligned hyperplane $S_v \subseteq \mathcal{X}$ that divides R_v into two nonempty subregions $R_{v_\leq}$, $R_{v_>}$; these will be the regions associated with the children $v_\leq$, $v_>$ of v. The root node is associated with

[1] Formally, f_B minimizes $\mathbf{E}_{(x,y)\sim\mathcal{D}}\left[err(f(x), y)\right]$ for $err\colon \mathcal{Y} \times \mathcal{Y} \to \{0, 1\}$, with $(y, y') \mapsto 1$, iff $y \neq y'$.

the whole feature space. We introduce two parameters f and t to specify the hyperplane for v. Parameter f selects the feature dimension and t the value along dimension f:

$$S_v(f,t) = \{\mathbf{x} \in \mathcal{X} \mid x_f = t\}\,. \tag{3.1}$$

We call $S_v(j,t)$ the *decision* or *splitting rule*, or just *split* of v. A hyperplane separates the space into two half-spaces and we will refer to them with

$$\mathcal{X}_{\leq}\big(S_v(f,t)\big) = \{\mathbf{x} \in \mathcal{X} \mid x_f \leq t\}\,,$$
$$\mathcal{X}_{>}\big(S_v(f,t)\big) = \{\mathbf{x} \in \mathcal{X} \mid x_f > t\}\,.$$

We omit the parameters to S_v if they are not of interest. The two subregions of R_v can then be written as

$$R_{v_{\leq}} = R_v \cap \mathcal{X}_{\leq}(S_v),$$
$$R_{v_{>}} = R_v \cap \mathcal{X}_{>}(S_v).$$

Each region is further divided, until some stopping criterion is fulfilled; this creates a leaf in DT. If $\ell_1, \ell_2, \ldots, \ell_m$ denote the leaves of DT, than the regions R_ℓ with $\ell = \ell_1, \ell_2, \ldots, \ell_m$ represent the final partition classes. For each $\mathbf{x} \in R_\ell$, the decision tree returns the same value c_ℓ. Figure 3.2 illustrates one possible decision tree for a given training sequence and the corresponding feature space partition.

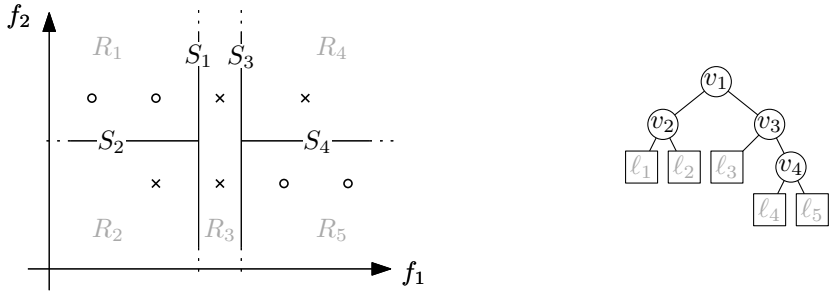


Figure 3.2: Left: A training set and a partition of the feature space into rectangular axis-aligned regions R_i. Also indicated are the slitting rules S_j of the corresponding decision tree. Right: A decision tree that implements the partition to the left.

Given a new element $\mathbf{x}$, it is now straightforward to compute $DT(\mathbf{x})$. We start at the root v and check to which side of S_v the element $\mathbf{x}$ lies. This determines the child node v' of v at which the next check is to be performed, namely, the one whose associated region $R_{v'}$ contains $\mathbf{x}$. We iterate this process until we reach a leaf node ℓ and return c_ℓ.

We have yet to explain how the best tree topology and decision rules are found, and which values c_ℓ to return. It turns out that finding the optimal decision tree is computationally hard in several variants (see the work of Hancock, Jiang, Li, and Tromp [22] or Hyafil and Rivest [28]). No version of the algorithm we describe here would have found the optimal tree for the partition as depicted in Figure 3.3 of the same training data as in Figure 3.2.

In practice, finding a well performing decision tree is performed by a greedy approach. Below we describe the CART (from Classification And Regression Trees) approach from Breiman et al. [11] as described in Ref. [19]. Alternative algorithms were developed, for

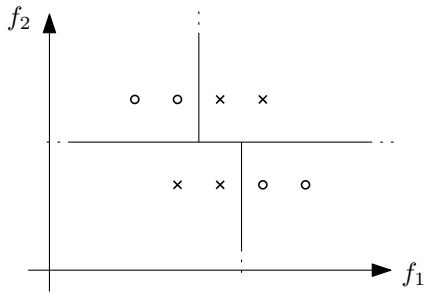


Figure 3.3: This partition has the least number of axis aligned splits and regions but requires an initial horizontal split that cannot be found with the naive greedy approach described here.

example, by Quinlan [43][44].

The Learning Process

During the learning phase we try to grow a tree that performs well on the training sequence T. Every vertex v will only consider the elements $(\mathbf{x}_i, y_i)$ in T with $\mathbf{x}_i \in R_v$; we will call this subsequence T_v. The root will have to consider all elements. We will iteratively select a leaf ℓ in the current tree and assign a splitting rule to it that best divides T_ℓ. This will create two new child nodes attached to ℓ. There is an infinite number of possible splits, even if we fix the feature space dimension f. Luckily, since the training set is finite, the actual number of splits that need to be considered can be reduced to a number polynomial in the size of the training sequence. This is due to the fact that all axis aligned splits in-between the same two neighboring training samples will perform identically. Therefore, for each feature space dimension we only need to consider the $n+1$ splits; $(n-1)$ in-between neighboring samples plus the two to the very left and right. Doing this creates $p(n+1)$ splits in total.

To select the best split, we first need to define an impurity criterion $\mathit{impurity}(\widetilde{T})$ for a given sequence $\widetilde{T} = (\mathbf{x}_i, y_i)_{i=1..\widetilde{n}}$. The impurity should be smallest for sequences in which all elements belong to the same class, and the more the class ratio approaches $1/2$ the larger the impurity should grow. We will give three popular criteria. For $y \in \mathcal{Y}$ let $\Pr(y)$ be the probability of choosing an element with class y when sampling the elements in $\widetilde{T}$ uniformly at random, that is, $\Pr(y) = \big|\{i \in [\widetilde{n}] \mid y_i = y\}\big|/\widetilde{n}$.

$$\text{Training/Missclassification error:} \qquad \mathit{impurity}(\widetilde{T}) = \min_{y \in \mathcal{Y}} \Pr(y) \tag{3.2}$$

$$\text{Cross-entropy:} \qquad \mathit{impurity}(\widetilde{T}) = -\sum_{y \in \mathcal{Y}} \Pr(y) \log \Pr(y) \tag{3.3}$$

$$\text{Gini index:} \qquad \mathit{impurity}(\widetilde{T}) = 2 \min_{y \in \mathcal{Y}} \Pr(y)(1 - \Pr(y)) \tag{3.4}$$

Figure 3.4 shows plots of the three criteria (scaled to the same maximum value). As can be seen, they are very similar and all show the desired behavior.

Given $\mathit{impurity}(\cdot)$, we can define the quality of a split S by the purity gain $\mathit{gain}(S)$ it produces. This will simply be the reduction of impurity if we were to perform S. To define it formally, let $\widetilde{T}_\leq$ be the subsequence of all elements $(\mathbf{x}_i, y_i)$ in $\widetilde{T}$ with $\mathbf{x}_i$ in $\mathcal{X}_\leq(S)$; let $\widetilde{T}_>$ be defined analogously. With $\widetilde{n}_\leq$ and $\widetilde{n}_>$ as the number of elements in

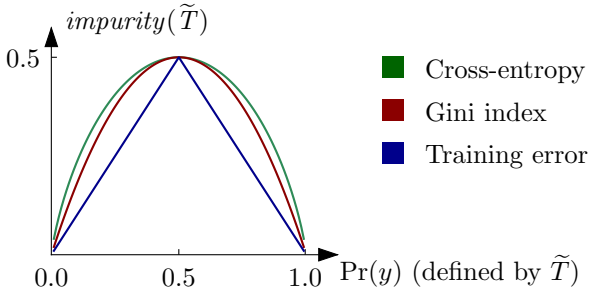


Figure 3.4: Plots of the three impurity criteria (scaled to the same maximum value).

$\widetilde{T}_{\leq}$ and $\widetilde{T}_{>}$, respectively, we set

$$gain(S) = impurity(\widetilde{T}) - \frac{\widetilde{n}_{\leq}}{\widetilde{n}} impurity(\widetilde{T}_{\leq}) - \frac{\widetilde{n}_{>}}{\widetilde{n}} impurity(\widetilde{T}_{>}).$$

We could continue to split every leaf that considers a training sequence with nonzero impurity, that is, has a split with nonzero purity gain. This way we grow a large tree with a potentially complex decision region that eventually will classify all training samples correctly. We would believe that such a tree is very likely to overfit the training data. This motivates a stopping criterion that terminates the growing phase, or parts of it, although there are still splits remaining that would decrease the training error. Some possible stopping criteria are

- split a leaf only if the resulting purity gain exceeds some threshold,
- only consider leaves that exceed some impurity threshold,
- only consider leaves that contain a minimum number of training samples,
- specify the maximal allowed depth of the tree,
- any combination of the above.

Once the tree is finished, the final set of leaves $\ell_1, \ell_2, \ldots, \ell_m$ and the corresponding regions R_ℓ, with $\ell = \ell_1, \ell_2, \ldots, \ell_m$ are determined. For each ℓ we define

$$c_\ell = \arg\max_{y \in \mathcal{Y}} \left| \{(\mathbf{x}_i, y_i) \mid \mathbf{x}_i \in R_\ell,\ y_i = y\} \right|$$

as the value to return if a new sample $\mathbf{x}$ falls into the region R_ℓ. The full training algorithm is described by the following pseudocode.

function make_tree($T = [(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)]$)
  if stop_criterion_fulfilled(T) **then**
    $c \leftarrow$compute_estimate(T)
    return new_leaf(c)
  else
    $(f, t) \leftarrow$ best_axis-aligned_split(T) ▷ j is the feature, t the threshold
    $T_{\leq} \leftarrow [(\mathbf{x}_i, y_i)$ in T with $x_f \leq t]$
    $T_{>} \leftarrow [(\mathbf{x}_i, y_i)$ in T with $x_f > t]$
    left_child $\leftarrow$ make_tree($T_{\leq}$)
    right_child $\leftarrow$ make_tree($T_{>}$)
    return new_node((f, t), *left_child*, *right_child*)

end if
end function

function stop_criterion_fulfilled($T = [(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)]$)
Returns true if T fulfills the chosen criterion,
else returns false.
end function

function compute_estimate($T = [(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)]$)
Returns the majority class in T.
end function

function new_leaf(c)
Returns a new leaf with c stored as estimate for the corresponding region.
end function

function best_axis-aligned_split($T = [(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)]$)
Returns index of feature dimension f and threshold t for the best split on T.
end function

function new_node((f, t), *left_child*, *right_child*)
Returns a tree where the left and right children of the root are given by
left_child and *right_child*, and the split at the root is defined by
feature dimension f and threshold t.
end function

An extension that can yield superior results to the early stopping approach above is to grow a large, probably overfitting tree and prune it afterwards. Several pruning variants have been proposed, see Mingers [36] for a survey. We mention this only for completeness, as, at the time of this writing, it was not implemented in the chosen library.

Random Forests

This section itroduces random forests and serves as an outlook on how to improve the decision tree method.

Random forests were introduced by Ho [26] and can be seen as a special case of *ensemble methods*. In these, several classifiers are gathered to form a single predictor whose output is determined by some form of vote of the constituent classifiers. The hope is that in regions where a few classifiers have overfitted and adjusted to noise in the data, the majority has not and, hence, will vote for the true label. Especially for classifiers with high variance, like decision trees, this might improve the generalization performance.

In random forests, all classifiers in the ensemble are decision trees. To improve upon the performance of a single tree classifier that was trained on the whole training data, randomization is introduced. Each decision tree instance is trained on its own random sample of the training data. The sample is drawn uniformly at random with replacement, which is known as *bootstrapping*. The idea of unifying several classifiers trained on bootstrap samples was introduced as *bagging* (short for bootstrap aggregating) by Breiman [9]. An additional source of randomness is introduced when growing the trees, as introduced by Ho [27]. When considering a leaf for a split, a random subset of the

feature dimensions is selected. The split is then performed with respect to this subset only. For each split, a new subset can be chosen. For other options on how to create the decision trees, see the seminal work by Breiman [10].

3.1.1 Software

We have used the Scikit-learn package [41] of the SciPy library [29]. For decision trees the DecisionTreeClassifier implementation was used. The description states that it implements an optimized version of the CART algorithm.

3.2 Parameter Exploration and Insights

We need to consider various parameters when trying to find a well performing decision tree. We have to decide on

- how to tackle the class imbalance,
- what kind of preprocessing to apply,
- what kind of format to choose
  (single or multiple images, averaged or not, original size or resampled),
- whether images or extracted features perform better,
- and how deep the tree should be.

We will address some of the issues in isolation and others combined, depending on how universal we assume the effects to be. The performance of images and features will be explored in parallel.

3.2.1 Addressing the class imbalance

We will first examine the behaviour of the decision tree learning algorithm to the three strategies to tackle the class balance, introduced in section 2.5.2: train on imbalanced classes, undersampling, and oversampling. We conjecture that the behaviour of the decision tree learning algorithm with respect to these strategies is rather general, that is, it is independent of the other parameters listed above. Therefore, we fix a setting of the remaining parameters.

We restrict the maximal tree depth to 5. For each strategy, we train 100 trees and evaluate their performance on a validation set that was split-off from the training set and contains 100 elements per class. Training and evaluation sets are re-sampled for each of the 100 trees (*Monte Carlo cross-validation*). Moreover, in the under- and over-sampling cases, the total set of elements under consideration is sampled new for each tree. To keep it simple, we use only ds1. The chosen format is `2d_slice1_sample_64`, and the only preprocessing is performed by resampling the images to a third of their size in both dimensions. The number of elements for every strategy is listed in Table 3.1.

Results of this experiment are plotted in Figure 3.5.
Table 3.2 shows the median validation accuracies in numbers. It turns out that undersampling seems to be the most promising strategy to continue.

strategy	size of training set	size of validation set
undersampling	240 per class	200 per class
imbalanced	1210 *open*, 240 *closed*	200 per class
oversampling	1210 per class	200 per class

Table 3.1: Training and validation set composition per strategy.

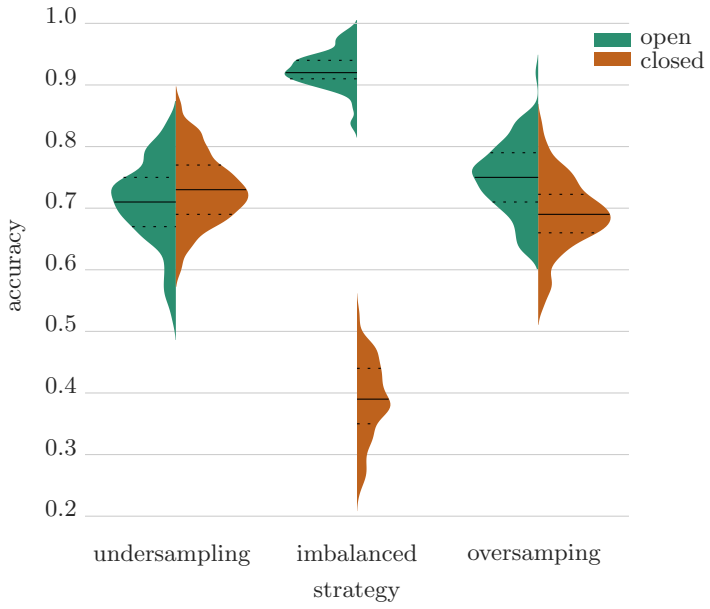


Figure 3.5: Kernel density estimations (KDEs) of the classification accuracy on the validation set for each class and strategy, generated from 100 runs. The solid lines indicate the median accuracy, dotted lines the quartiles. The plot shows that the decision tree reflects the imbalanced classes in the evaluation performance.

strategy	class	acc.
undersampling	open	**0.71**
	closed	**0.73**
imbalanced	open	0.92
	closed	0.39
oversampling	open	0.75
	closed	0.69

Table 3.2: For each strategy and class, the median validation accuracies. Emphasized is the pair that maximizes the median min-accuracy.

3.2.2 Preprocessing

Next, we analyze the impact of different preprocessing techniques. In particular, we compare the performance of per-feature-standardization to per-image-standardization. As baseline, we show the performance on the raw data without any preprocessing. A subsequent PCA transformation might prove beneficial if the distances between instances of different classes are large compared to the inter-class distances after the preprocessing. Therefore, we also compare each preprocessing version to itself with the additional step of a PCA transformation.

As before, we conjecture that the effect of preprocessing is rather general, at least inside the two input formats 'image' and 'feature', and can be explored in isolation while fixing the other parameters. Just to be sure, we perform the analysis on two different formats with opposite characteristics and check whether the results are consistent. The two formats are:

1. `2d_slice1_sample_64_small` Representing the most condensed information. An image in this format is obtained by averaging all voxel intensities in a cylindrical region in the tomography data. Subsequently it was resampled to half its size along both dimensions, which involves a Gaussian smoothing prior to it.

2. `2d_slice4_sample_1` represents the input with most information, but also most noise. It consists of 4 different cross-sections of the endpoint.

We also use these two formats to obtain a feature representation for each. The whole process of preprocessing, extracting features, and PCA transformation is ordered as

$$\mathbf{x} \xrightarrow{\text{preprocessing}} \mathbf{x}^* \xrightarrow{\text{extract features}} \big(f_1(\mathbf{x}^*), f_2(\mathbf{x}^*), f_3(\mathbf{x}^*)\big) \xrightarrow{\text{PCA}} (z_1, z_2, z_3).$$

Again, we train 100 trees for each setting and restrict the tree depth to five. Following the results of the first section, we undersample the labelled data form ds1, and split-off a validation set. Both, undersampling and separation of the validation set is redone for each tree. In total we run four experiments (two formats, each format either as image or as features). The results are plotted in Figure 3.6. First of all, we note that the plots in the first column are consistent with the plots in the second column. This strengthens our assumption about the independence of the preprocessing technique on the format, and we will not perform any tests for the remaining formats.

Secondly, if we focus our attention on the image-based input, we see that the performance deteriorated for all techniques when they were followed by PCA transformation. Opposed to this, performance on the feature-based input mostly seems to improve or remain constant.

On average, the best performance is observed when the input is per-image-standardized. This holds for both image- and feature-based inputs. For features, a subsequent PCA transformation should be considered.

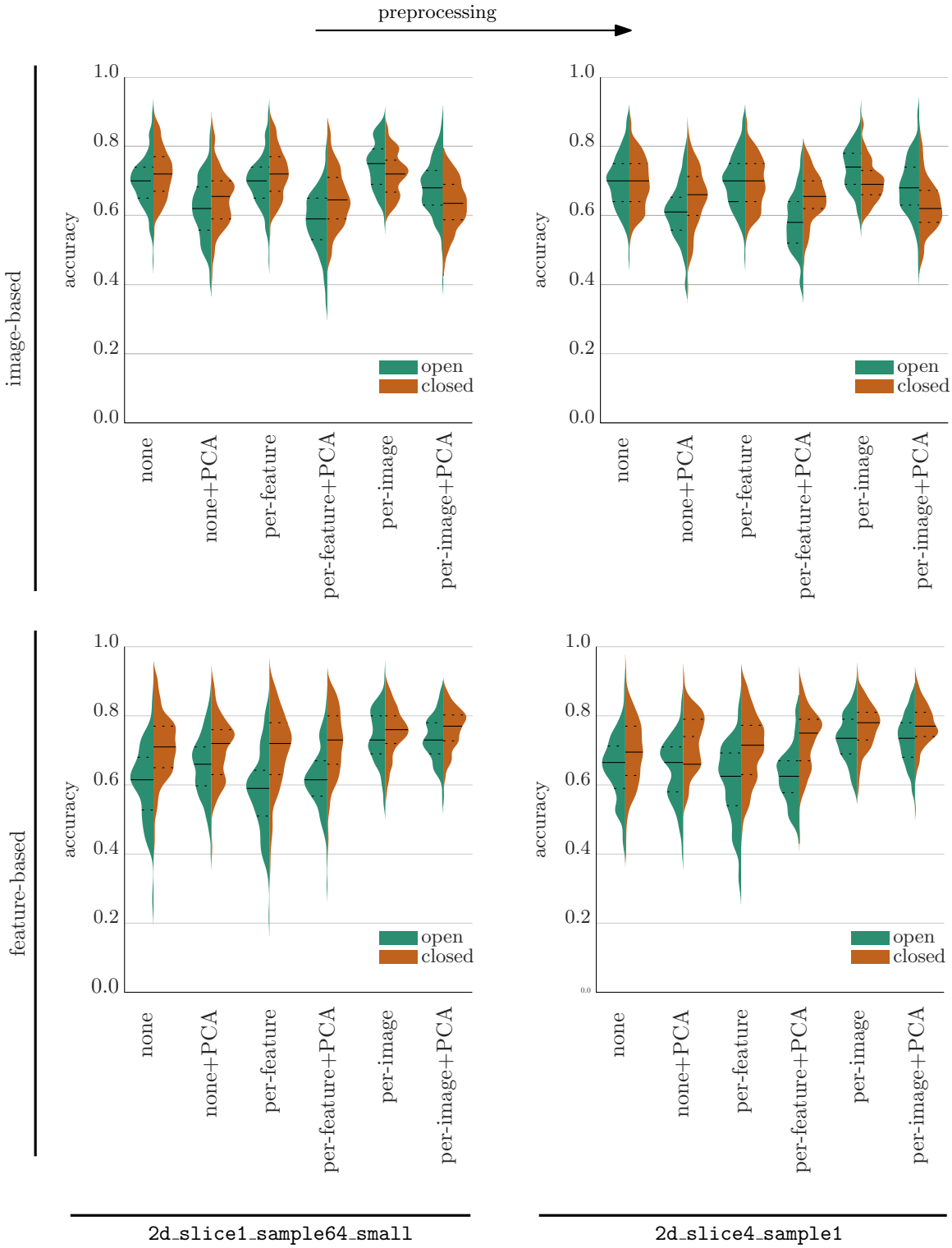


Figure 3.6: KDEs of the classification accuracy on the validation set for each class and preprocessing technique, generated from 100 runs per pair of open-closed KDEs. The first row shows plots for decision trees trained on images, the second shows plots for trees that were trained on features extracted form images. Plots in the same column were trained on input from the same format. Solid lines indicate the median accuracy, dotted lines the quartiles.

image-based:

		2d_slice1_sample_64_small		2d_slice4_sample_1	
preprocessing	class	acc.	acc. w/ PCA	acc.	acc. w/ PCA
none	open	0.700	0.620	0.700	0.610
	closed	0.720	0.655	0.700	0.660
per-feature	open	0.700	0.590	0.700	0.580
	closed	0.720	0.645	0.700	0.655
per-image	open	**0.750**	0.680	0.740	0.680
	closed	**0.720**	0.635	0.690	0.620

feature-based:

		2d_slice1_sample_64_small		2d_slice4_sample_1	
preprocessing	class	acc.	acc. w/ PCA	acc.	acc. w/ PCA
none	open	0.615	0.660	0.665	0.665
	closed	0.710	0.720	0.695	0.740
per-feature	open	0.590	0.615	0.625	0.625
	closed	0.720	0.730	0.715	0.750
per-image	open	0.730	0.730	**0.735**	0.735
	closed	0.760	0.770	**0.780**	0.770

Table 3.3: Median accuracies for each preprocessing technique with and without subsequent PCA transformation. Empathized are the pairs that maximize the median min-accuracy. PCA does not seem have a beneficial effect for trees that learned on images. Trees that learned on feature representations seemed to benefit from PCA transformation.

3.2.3 Remaining Parameters

The previous sections suggest that the training data should be balanced by undersampling and preprocessed with per-image-standardization. For feature-based input, a PCA transformation after feature extraction might benefit the performance.

In this section we decide on which format to use. Including the images that were resampled to half the size along each dimension, this leaves eight formats. It is possible that trees trained with different input format perform best at different depths. For example, a tree that is trained with small images might perform better when looking at fewer pixels than a tree that has to consider 16 times that many input features. Additionally, the pixels in some images show averages of tomogram regions which results in less high-frequent information when compared to pixels in images that show actual cross-sections. Therefore, we cannot test formats independently of the tree depth, and the number of cases to be considered has to be multiplied with the number of possible maximal decision tree depths, which we allow to be any number from 1 to 10. A decision tree with depth 10 could theoretically partition the feature space into 1024 regions (more than there are training samples), but this number implies a perfect binary tree which is not guaranteed by the algorithm. We still expect the best validation performance for trees with significantly smaller depth.

For feature-based input, we consider the performance behaviour with and without PCA transformation after feature extraction, which gives another factor of two to the number of possible configurations.

In contrast to the previous sections, we have to ensure that training and validation sets are not too similar if we reliably want to detect when overfitting sets in. We do this be taking training and validation samples from distinct datasets.

Training will be done on a balanced subset of ds1 comprising 340 endpoints per class; validation will be performed on the union of balanced subsets of ds2 and ds3 comprising 64 and 20 endpoints per class, respectively. We train 20 trees for every combination of input format and depth for both, image-based and feature-based input. For the latter, we also consider the application of PCA transformation prior to the learning. For each tree, the training and validation subsets are newly sampled, which introduces randomness to the choice of open endpoints (we take all closed endpoints). The accuracy measure that we consider is the minimum of the two per class accuracies (min-accuracy). Figure 3.7 shows the median of the min-accuracies for 20 trees per depth. For an easier visual reception, the plot shows lines interpolating between the discrete values. Upon visual inspection, both of the feature-based approaches seem to surpass the image-based by roughly 5% on average. At the same time, the training curves have less inclination. At depth 10, no feature-based approach was able to perfectly learn the training set, while the trees learned on images perfectly separated the training set, some already at depth 8.

We see that the rich feature set of the image input contains too much too specific information, which allows for a fast learning of the training set, but does not generalize well to new samples. This is supported by the difference in depth at which overfitting sets in, and the algorithm starts to adapt to training set specificities and noise. For images this starts at depth 2-3 while most feature-based results start to overfit at depth 4, theoretically, allowing twice to four times as many regions in sample space.

Inside each approach, the performance of different formats is rather coherent, without any one format standing out significantly, when plotting the full accuracy range. We see that there is only little improvement prior to depth 4 for the feature-based approach,

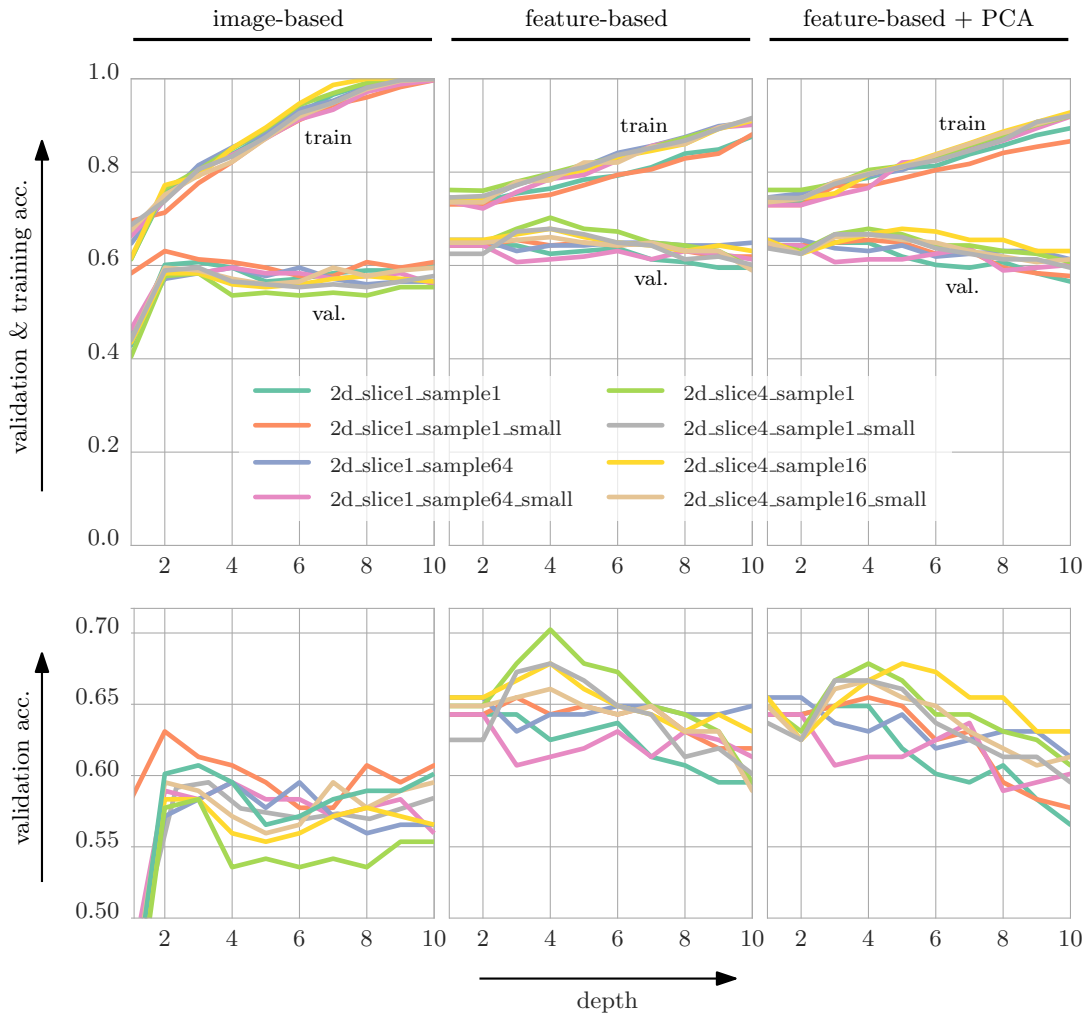


Figure 3.7: Median validation and training min-accuracies for 100 classifiers per input format, and tree depth. The discrete values are linear interplated.

which indicates that a single hyperplane and a single feature already achieve similar performance to a tree of depth 4. This is no surprise, considering the similarity of the three features. The bottom row in Figure 3.7 shows the magnification of the validation accuracies. Some formats seem to be more promising than others, and the difference of best medians between input formats is in the 5% range. The best results for each approach are listed in Table 3.4. These values correspond to the topmost peaks in the three magnified plots.

There is recognizable difference in the performance between images-based and feature-based input. We try to gain more insights from looking at the two approaches separately. The left column in Figure 3.8 shows isolated min-accuracies for each original format and its resampled version in the image-based approach. It could be argued the best performance is consistently achieved for smaller image formats, but the differences seem to be rather insignificant. The center and right columns show the performance of each format with its PCA transformed counterpart. More than before, it appears that there is no systematic connection to the performance.

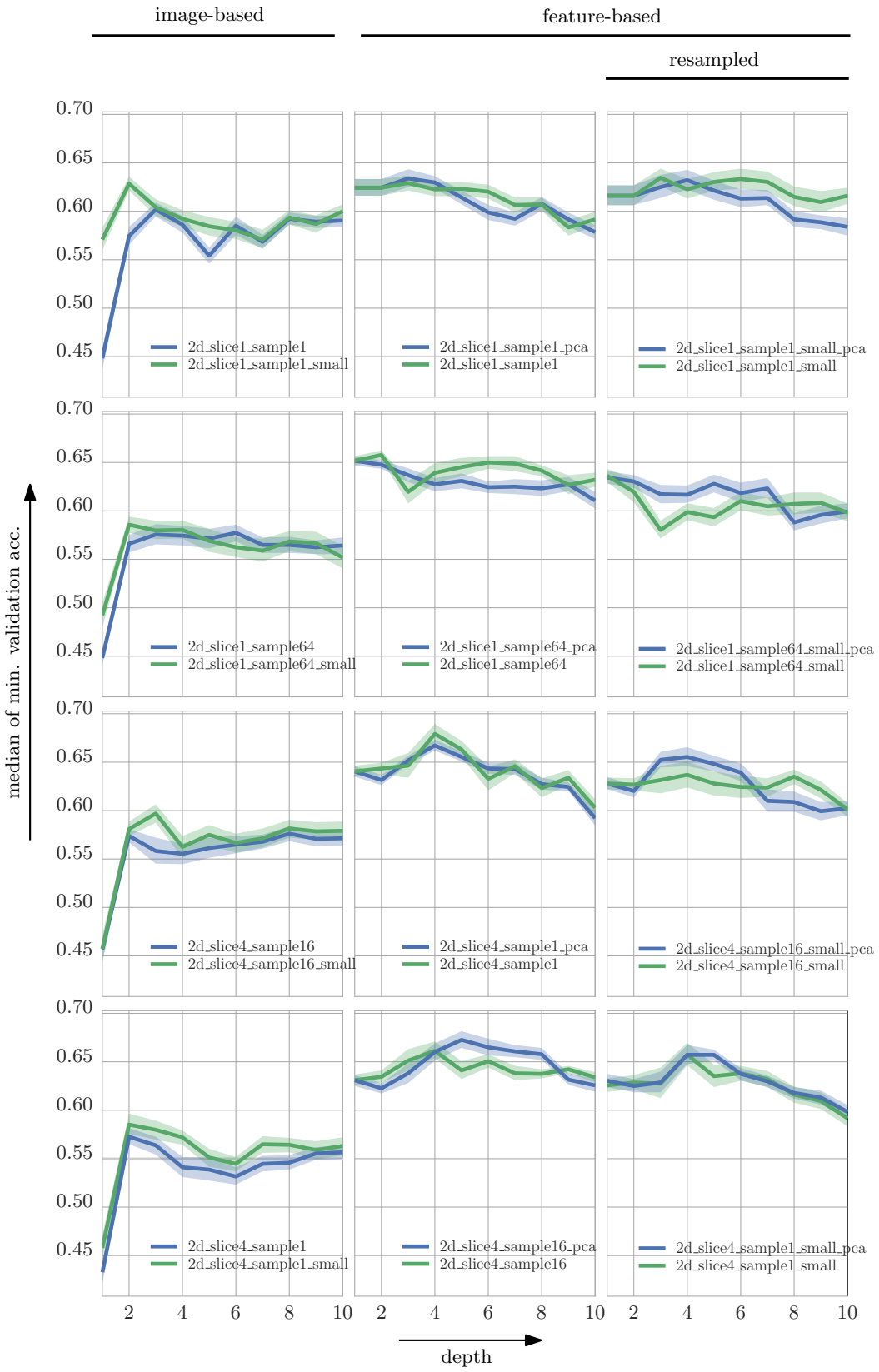


Figure 3.8: Isolated plots of median validation min-accuracies. Transparent bands indicate the 50% confidence intervals.

approach	format	depth	acc.
image-based	`2d_slice1_sample1_small`	2	0.630952
feature-based	`2d_slice4_sample1`	4	**0.702381**
feature-based PCA	`2d_slice4_sample1`	4	0.678571

Table 3.4: Best maximum median min-accuracy values over (in this order) input format, depth, run, and class. That is, the acc. value is obtained by first computing the min-accuracy over all trees, taking the median for the 20 trees for each depth, chosing the best depth for each input format, and, finally, choosing the best input format for each approach. Highlighted is the best performance.

3.2.4 Adding principal components

Realizing that the feature based approach outperforms the image-based one, it is natural to ask which features can be added to further increase the accuracy. It is obvious that the three extracted features are highly redundant and any new features should be designed to complement the existing ones. This would require further analysis of the misclassified samples.

Here, we take a more general approach and consider what happens, when we add the projections along the c most dominant principle components of the training set. This gives us c new features, and we can write the resulting feature map of an image $\mathbf{x}$ as

$$\mathbf{x} \mapsto \big(f_1(\mathbf{x}), f_2(\mathbf{x}), f_3(\mathbf{x}), p_1(\mathbf{x}), \dots, p_c(\mathbf{x})\big),$$

where $p_i(\mathbf{x})$ is computed by taking the scalar product of $\mathbf{x}$ shifted by the negative mean training image (centralization) and the direction of the i-th most dominant principle component of the training set. Reducing an image to its projection along the c most dominant principle components can be regarded as lossy information compression, and, if it turns out that the lost information is non-informative for the classification task, as a denoising strategy.

For an effective denoising, we have to set c small enough. At the same time we want enough components to reconstruct the endpoint morphology. Figure 3.9 shows reconstructed endpoints for various choices of c; we settle for $c = 25$. The results of experiments with the same setting as in the tree selection section and format `2d_slice4_sample_1` are shown in the plot in Figure 3.9. Adding principle components actually has a negative effect on the validation performance. We see similar behaviour as when we compared image-based and feature-based input. The steeper training curve indicates that the additional information does help in the discrimination of the training set where the principle components were computed on but does not generalize well to data from new tomograms. It seems that they are too specific to a single tomogram.

3.3 Final Training and Results

In conclusion, we choose to train the final decision tree on feature-based inputs without PCA transformation, obtained from per-image-standardized images with format `2d_slice4_sample_1`, on a balanced subset of the available training data.

The training set is given by the union of three sets, one from each of ds1, ds2, and ds3. We take all available closed endpoints, and a subset of the available open endpoints (randomly chosen without replacement) with matching size. This leads to 424 endpoints

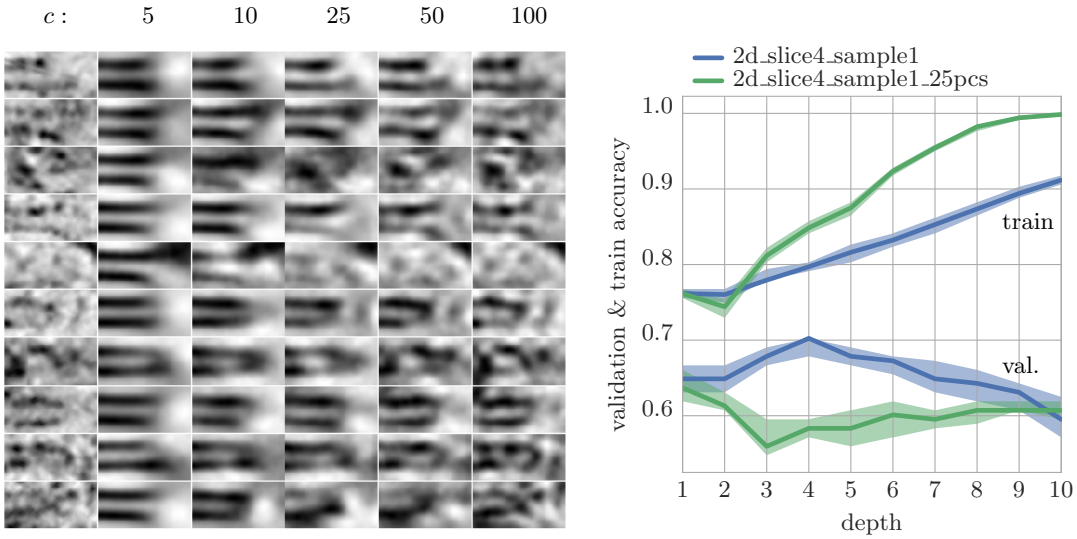


Figure 3.9: Left: Reconstructions for ten microtubule ends from projections onto the c most dominant principle components. Right: Median training and validation min-accuracies for feature-based inputs with additional principal components (green) and without (blue).

per class. On the training set, it achieves per-class accuracies of

0.80 (*open*)
0.82 (*closed*).

After training, the final tree was tested on the test set comprising all non-contradictory labeled endpoints from qs1,. . . ,qs4. This set contains 1025 *open* and 336 *closed* samples. The final tree is shown in Figure 3.10. It achieves per-class accuracies of

0.62 (*open*)
0.72 (*closed*).

Figure 3.11 lists the first 10 elements of each class, where the decision tree failed.

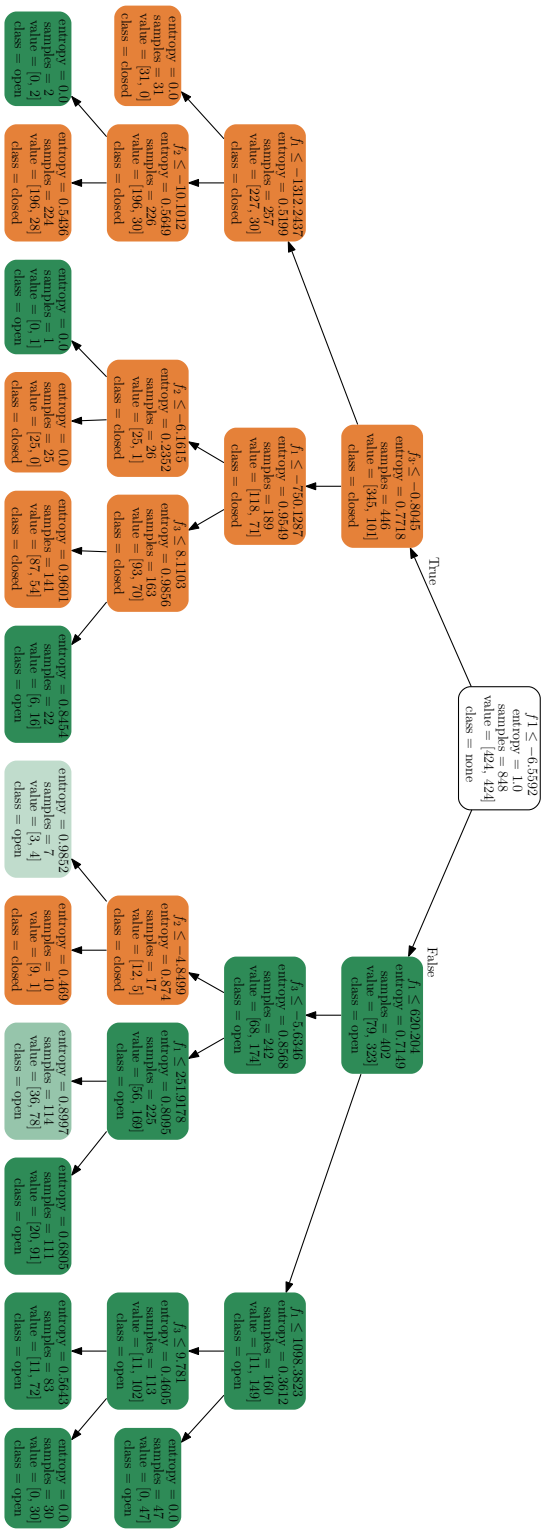


Figure 3.10: The final classification tree trained to depth 4. The first and third leaves contain rather few open elements and would probably be removed in an implementation that allows an subsequent trimming of the tree. We see that both subtrees of the root are already quite pure.

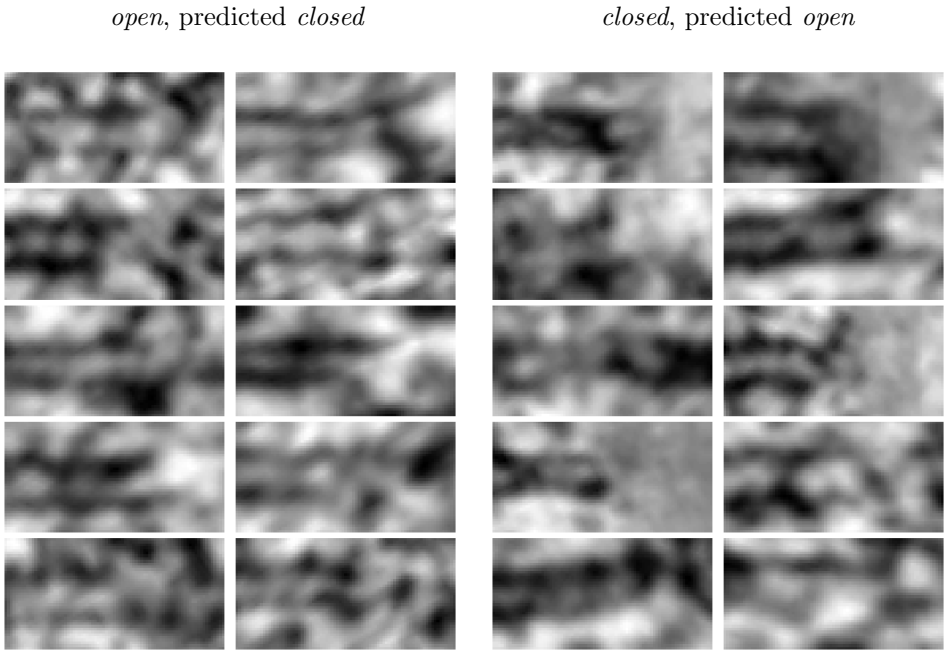


Figure 3.11: Examples of misclassified elements. Left: Samples were experts labeled *open* and the tree predicted *closed*. Right: Samples were experts labeled *closed* and the tree predicted *open*.

4 Support Vector Machines

4.1 Basic Principles

Support Vector Machines (*SVM*s)[6][14] are, in their basic form, linear separators. They try to find a hyperplane that well separates a given training sequence. What distinguishes them from other linear separators is that, of all possible separating hyperplanes, they try to find the one that maximizes the minimal distance to any training sample. The left image in Figure 4.1 shows training samples with possible separating hyperplanes. Although each of these perfectly separates the training samples, they run very close to them and we would expect that newly generated samples are likely to appear on the wrong side of the plane. In the image to the right we see the hyperplane chosen by the SVM. This choice will probably better generalize to new samples. The area around the hyperplane that does not contain any training samples is called the *margin*, hence, we call the SVM classifier a *maximum margin classifier*.

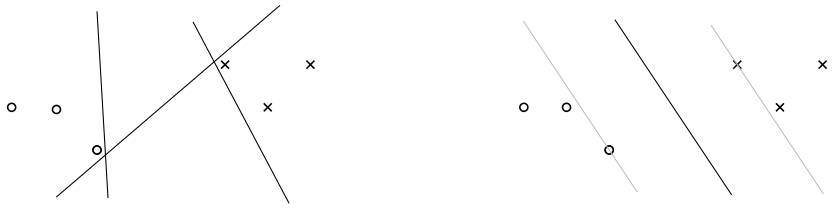


Figure 4.1: Left: Possible separating hyperplanes, each of which has optimal training error although they approach the training samples rather closely. Right: The maximum margin hyperplane that will be chosen by the SVM.

Often there is no guarantee that the problem under consideration will produce samples that are linearly separable. In such a case the approach above can be relaxed to allow for samples to violate the margin and enter the wrong side of the hyperplane, hence, introducing some non-zero training error (see Figure 4.3). We call this the soft-margin approach while referring to the previous case as the hard-margin approach.

Finally, SVMs allow for a generalization – so called *kernel methods* – to nonlinear separation boundaries in feature space $\mathcal{X}$. This is based on the general idea of applying a non-linear map $\varphi\colon \mathcal{X} \to \mathcal{X}'$ to the training sequence and performing the SVM technique in $\mathcal{X}'$ instead of $\mathcal{X}$. Exploiting some special properties of SVMs will allow for the *kernel trick*, an efficient implementation of this idea.

In the subsequent parts, we follow the expositions in the *Support Vector Machine* and *Kernel Methods* part of Abu-Mostafa's lectures [4] and the *Support Vector Machines* part of Ng's lecture notes [38], with some insertions from books by Shalev-Shwartz and Ben-David [52] and Friedman, Hastie, and Tibshirani [19]. For an in-depth discussion see the work of Vapnik [56], one of the originators of this technique, or the tutorial by Burges [13]. An exposition of Kernel Methods can be found in the book by Schölkopf and Smola [51].

4.1.1 Hard Margin – The Separable Case

We are given a training sequence $T = (\mathbf{x}_i, y_i)_{i=1..n}$ with $\mathbf{x}_i \in \mathcal{X} = \mathbb{R}^p$ and $y_i \in \mathcal{Y} = \{+1, -1\}$. The specific choice of $\mathcal{Y}$ will be clear in a moment. In this first part, we will assume that T is linearly separable, that is, there exists a unit vector $\hat{\mathbf{w}}$ and distance d that describe a hyperplane g, such that for all i:

$$\langle \mathbf{x}_i, \hat{\mathbf{w}} \rangle + d = \begin{cases} > 0 & \text{if } y_i = +1, \\ < 0 & \text{if } y_i = -1. \end{cases}$$

The choice of $\mathcal{Y}$ allows us to multiply with y_i and express both cases by the single condition

$$y_i(\langle \mathbf{x}_i, \hat{\mathbf{w}} \rangle + d) > 0.$$

Due to the strict inequality we can set ε as the minimal distance of any point $\mathbf{x}_i$ to the separating plane and write

$$y_i(\langle \mathbf{x}_i, \hat{\mathbf{w}} \rangle + d) \geq \varepsilon, \quad \text{where} \quad \varepsilon = \min_{i \in [n]} |\langle \mathbf{x}_i, \hat{\mathbf{w}} \rangle + d| \,. \tag{4.1}$$

That is, 2ε is the width of the largest *margin* we could place around g such that no point $\mathbf{x}_i$ lies inside of it. We call such a separating hyperplane a *hard-margin* classifier, as any penetration of the margin is prohibited. In the left of Figure 4.2 we give an example. Note how shifting g to the right would allow for a larger margin. A different orientation like g' in the right image would lead to further improvements. We can divide Equation

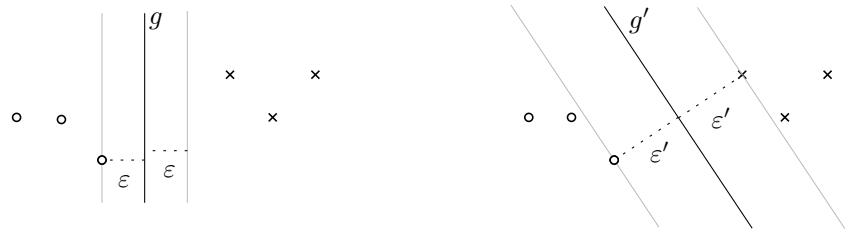


Figure 4.2: Illustration of a training sequence that is linearly separable and separating hyperplanes g and g'. The gray lines indicate the boundary of the largest margins (with width ε and ε', respectively) that are permitted.

(4.1) by ε to obtain the equivalent description

$$y_i(\langle \mathbf{x}_i, \mathbf{w} \rangle + b) \geq 1, \quad \text{where } \mathbf{w} = \frac{\hat{\mathbf{w}}}{\varepsilon} \text{ and } b = \frac{d}{\varepsilon} \tag{4.2}$$

We are looking for a hyperplane that allows the widest margin without violating these restrictions. In Equation (4.2) we see that the width of the margin $2\varepsilon = 2/\|\mathbf{w}\|$ is inversely proportional to the size of $\mathbf{w}$. Thus, we can formulate our goal in form of a constraint optimization problem, where we put the constraints in their zero-form:

$$\max_{\mathbf{w}, b} \frac{1}{\|\mathbf{w}\|}, \quad \text{such that} \quad \forall i \colon y_i(\langle \mathbf{x}_i, \mathbf{w} \rangle + b) - 1 \geq 0.$$

Instead of maximizing the inverse, we might as well minimize $\|\mathbf{w}\|$ which is equivalent to

minimizing $\|\mathbf{w}\|^2/2$. This leads to

$$\min_{\mathbf{w},b} \frac{1}{2}\|\mathbf{w}\|^2, \quad \text{such that} \quad \forall i\colon y_i(\langle \mathbf{x}_i, \mathbf{w}\rangle + b) - 1 \geq 0. \tag{4.3}$$

This is a quadratic minimization problem with affine inequality constraints and can be solved efficiently with optimization techniques. The usual approach here is to form the *Lagrangian* of Problem (4.3) and solve its *dual*. We will not go into much detail here but present enough information to explain the origin of the *support vector* part of the name.

First, we reformulate Problem (4.3) while introducing some notational simplifications. We want to find the argument of the solution to the minimization problem

$$\min_{\mathbf{w},b} \frac{1}{2}\langle \mathbf{w}, \mathbf{w}\rangle, \quad \text{such that} \quad \forall i\colon g_i(\mathbf{w}, b) \geq 0, \tag{4.4}$$

where $g_i(\mathbf{w}, b) := y_i(\langle \mathbf{x}_i, \mathbf{w}\rangle + b) - 1$. We build the corresponding Lagrangian L by introducing a multiplier α_i for every inequality constraint g_i and adding the product to the main objective:

$$L(\mathbf{w}, b, \boldsymbol{\alpha}) = \frac{1}{2}\langle \mathbf{w}, \mathbf{w}\rangle - \sum_{i=1}^{n} \alpha_i g_i(\mathbf{w}, b). \tag{4.5}$$

The negative sign is due to the fact that the Lagrange multiplier formalism requires inequality constraints of the form $g(\mathbf{w}, b) \leq 0$, hence, we multiplied the g_i by -1. We find the solution to our initial problem by the *primal* or *dual* formulation,

$$\text{primal:} \quad \min_{\mathbf{w},b} \max_{\boldsymbol{\alpha}} L(\mathbf{w}, b, \boldsymbol{\alpha}) \tag{4.6}$$

$$\text{dual:} \quad \max_{\boldsymbol{\alpha}} \min_{\mathbf{w},b} L(\mathbf{w}, b, \boldsymbol{\alpha}). \tag{4.7}$$

We have a convex objective function $\langle \mathbf{w}, \mathbf{w}\rangle/2$ and affine constraints g_i. This guarantees that both formulations yield the same result, as long as our training set is linearly separable, see for example Ref. [8, p. 226f.].

Let us consider the dual problem and try to find a stationary point with respect to the inner minimization, that is , we search for a point with $\nabla_{\mathbf{w}} L(\mathbf{w}, b, \boldsymbol{\alpha}) = \partial/\partial b\, L(\mathbf{w}, b, \boldsymbol{\alpha}) = 0$. With $\nabla_{\mathbf{w}} g_i(\mathbf{w}, b) = y_i \mathbf{x}_i$, we get

$$\begin{aligned}
\nabla_{\mathbf{w}} L(\mathbf{w}, b, \boldsymbol{\alpha}) &= \nabla_{\mathbf{w}} \frac{1}{2}\langle \mathbf{w}, \mathbf{w}\rangle - \nabla_{\mathbf{w}} \sum_{i+1}^{n} \alpha_i g_i(\mathbf{w}, b) \\
&= \mathbf{w} - \sum_{i+1}^{n} \alpha_i \nabla_{\mathbf{w}} g_i(\mathbf{w}, b) \\
&= \mathbf{w} - \sum_{i+1}^{n} \alpha_i y_i \mathbf{x}_i \\
&\stackrel{!}{=} 0. \\
\implies \mathbf{w} &= \sum_{i+1}^{n} \alpha_i y_i \mathbf{x}_i
\end{aligned} \tag{4.8}$$

That is, the orientation of the optimal hyperplane and the width of the corresponding margin are given by a linear sum of the position vectors of the training samples. The α_i give us the influence of the corresponding training sample $\mathbf{x}_i$. Further, with $\partial/\partial b\, g_i(\mathbf{w}, b) = y_i$ and omitting all terms that do not depend on b, we get

$$\begin{aligned}\frac{\partial}{\partial b} L(\mathbf{w}, b, \boldsymbol{\alpha}) &= \frac{\partial}{\partial b}\Big(- \sum_{i=1}^{n} \alpha_i g_i(\mathbf{w}, b)\Big) \\ &= -\sum_{i=1}^{n} \alpha_i \frac{\partial}{\partial b} g_i(\mathbf{w}, b) \\ &= -\sum_{i=1}^{n} \alpha_i y_i \\ &\stackrel{!}{=} 0.\end{aligned}$$

Therefore,

$$\sum_{i=1}^{n} \alpha_i y_i = 0. \tag{4.9}$$

$$\iff \sum_{i \in \{j | y_j = +1\}} \alpha_i = \sum_{i \in \{j | y_j = -1\}} \alpha_i \tag{4.10}$$

Together, Equation (4.10) and (4.8) tell us that there will be a balance between positive and negative samples in the choice of $\mathbf{w}$. Plugging expression (4.8) for $\mathbf{w}$ back into the Lagrangian (4.5) and exploiting Equation (4.10) will lead to a quadratic optimization problem which only depends on $\boldsymbol{\alpha}$:

$$\begin{aligned}&\max_{\boldsymbol{\alpha}} \Bigg(\sum_{i+1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \langle \mathbf{x}_i, \mathbf{x}_j \rangle \Bigg), \\ &\text{with } 0 \leq \alpha_i, \quad \text{for } i = 1, \ldots, n, \\ &\text{and } \sum_{i=1}^{n} \alpha_i y_i = 0.\end{aligned} \tag{4.11}$$

We will derive this further below in the section dealing with the soft-margin case. Equation (4.11) can be solved with standard numerical optimization techniques, as proposed in [7]. In practice, we are likely to encounter problems comprising many thousands or more training samples. In these cases, the quadratic size of the problem in the number of input samples might render this approach too inefficient. This has motivated the development of more specialized methods [40] [42].

Another consequence of only having affine inequality constraints is that the argument of the minimum $(\mathbf{w}^*, b^*, \boldsymbol{\alpha}^*)$ of Problem (4.7) satisfies the *Karush-Kuhn-Tucker* (KKT) conditions [33]. One of these – the *dual complementary condition* – is of particular interest. It states that

$$\forall i\colon \ \alpha_i^* g_i(\mathbf{w}^*, b^*) = 0. \tag{4.12}$$

This encodes the origin of the name *Support Vector Machine*. The condition says that for each point $\mathbf{x}_i$ of the training sequence, either $g_i(\mathbf{w}^*, b^*) = y_i(\langle \mathbf{x}_i, \mathbf{w} \rangle + b) - 1 = 0$,

that is, the point $\mathbf{x}_i$ is precisely on the boundary of the margin, or $\alpha_i = 0$ meaning that $\mathbf{x}_i$ will not play a role in the choice of $\mathbf{w}$, as indicated by Equation (4.8). We call all $\mathbf{x}$ in $\{\mathbf{x}_i \mid \alpha_i > 0\}$ the *support vectors*, since they support the decision boundary. All other points do not play a role in its selection.

Knowing the support vectors allows us to compute the optimal b^*, once we have found $\boldsymbol{\alpha}^*$. We compute $\mathbf{w}^*$ with Equation (4.8) and b^* by solving $y_i(\langle \mathbf{x}, \mathbf{w} \rangle + b) - 1 = 0$ for b, for any support vector x.

Then we can classify a new sample $\mathbf{x}$ with

$$\text{class}(\mathbf{x}) = \text{sign}\left(\langle \mathbf{x}, \mathbf{w}^* \rangle + b^*\right). \tag{4.13}$$

4.1.2 Soft Margin – The Non-Separable Case

In general, we cannot assume that a given training sequence is linearly separable (see Figure 4.3, left) and even if this is the case there might be outliers that force the hard margin SVM to an unfortunate decision boundary (see Figure 4.3, right). We can resolve

Figure 4.3: Left: An arrangement that is not linearly separable. Right: A linearly separable arrangement with solid lines indicating the hard margin classifier. Allowing the outlier (red) to violate the margin – even up to misclassification – would yield a more promising separation and margin (dashed lines).

this by introducing a *slack variable* $\xi_i \geq 0$ for every training sample $\mathbf{x}_i$. Its purpose is to relax the margin condition in Equation (4.2):

$$y_i(\langle \mathbf{x}_i, \mathbf{w} \rangle + b) \geq 1 - \xi_i.$$

Each variable is now allowed to violate the margin by a value proportional to ξ_i. To keep these violations low, we introduce a small addition to the optimization problem (4.3):

$$\min_{\mathbf{w},b} \ \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{l} \xi_i, \quad \text{such that} \quad \begin{array}{l} \forall i\colon g_i(\mathbf{w}, b, \xi_i) \geq 0, \\ \xi_i \geq 0, \end{array} \tag{4.14}$$

where $g_i(\mathbf{w}, b, \xi_i) := y_i(\langle \mathbf{x}_i, \mathbf{w} \rangle + b) - 1 + \xi_i$. So we are still trying to maximize the width of the margin but at the same time we try to keep the sum of all margin violations low. The trade-off parameter C will be set independently and adjusts how much we value one goal over the other. With growing C we increase the penalty of a margin violation and eventually approach the hard margin SVM. In the scope of the optimization problem it simply acts as a constant.

Similarly to the hard-margin case, we can set up the Lagrangian and formulate the primal and dual tasks. We only need to slightly modify the objective and introduce new Lagrange multipliers β_i for the additional inequality constraints $\xi_i \geq 0$.

$$L(\mathbf{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = \frac{1}{2}\langle \mathbf{w}, \mathbf{w} \rangle + C\sum_{i=1}^{n} \xi_i - \sum_{i=1}^{n} \alpha_i g_i(\mathbf{w}, b, \xi_i) - \sum_{i=1}^{n} \beta_i \xi_i \tag{4.15}$$

$$\text{primal:} \quad \min_{\mathbf{w},b,\boldsymbol{\xi}} \max_{\boldsymbol{\alpha},\boldsymbol{\beta}} L(\mathbf{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\beta}) \tag{4.16}$$

$$\text{dual:} \quad \max_{\boldsymbol{\alpha},\boldsymbol{\beta}} \min_{\mathbf{w},b,\boldsymbol{\xi}} L(\mathbf{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\beta}) \tag{4.17}$$

Again, we want to find stationary points with respect to the inner minimization in the dual problem (4.17). The gradient $\nabla_{\mathbf{w}} L(\mathbf{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\beta})$ and the partial derivative $\frac{\partial}{\partial b} L(\mathbf{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\beta})$ are identical to the hard-margin case. Therefore, we still have the same implications as in Equations (4.8) and (4.9):

$$\nabla_{\mathbf{w}} L(\mathbf{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\beta}) \stackrel{!}{=} 0 \qquad \Longrightarrow \qquad \mathbf{w} = \sum_{i+1}^{n} \alpha_i y_i \mathbf{x}_i \tag{4.18}$$

$$\frac{\partial}{\partial b} L(\mathbf{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\beta}) \stackrel{!}{=} 0 \qquad \Longrightarrow \qquad \sum_{i=1}^{n} \alpha_i y_i = 0. \tag{4.19}$$

The minimization with respect to ξ_i is new. With the partial derivative $\partial / \partial \xi_i \; g_i(\mathbf{w}, b, \xi_i) = 1$, we get:

$$\begin{aligned} \frac{\partial}{\partial \xi_i} L(\mathbf{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\beta}) &= \frac{\partial}{\partial \xi_i} \left(C \sum_{i=1}^{n} \xi_i \right) - \frac{\partial}{\partial \xi_i} \left(\sum_{i=1}^{n} \alpha_i g_i(\mathbf{w}, b, \xi_i) \right) - \frac{\partial}{\partial \xi_i} \left(\sum_{i=1}^{n} \beta_i \xi_i \right) \\ &= C - \alpha_i - \beta_i \\ &\stackrel{!}{=} 0 \\ \Longrightarrow \; \beta_i &= C - \alpha_i \end{aligned} \tag{4.20}$$

This gives us a new constraint.

We will now substitute $\mathbf{w}$ by Equation (4.18) and β_i by Equation (4.20) in the Lagrangian (4.15) and derive a minimization problem that only depends on α_i. We will do it one term at a time and start by plugging $\mathbf{w}$ into the first term of (4.15). In the following part we will make use of the distributivity and linearity of the scalar product.

$$\begin{aligned} \frac{1}{2} \langle \mathbf{w}, \mathbf{w} \rangle &= \frac{1}{2} \left\langle \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i, \sum_{j=1}^{n} \alpha_j y_j \mathbf{x}_j \right\rangle \\ &= \frac{1}{2} \sum_{i=1}^{n} \left\langle \alpha_i y_i \mathbf{x}_i, \sum_{j=1}^{n} \alpha_j y_j \mathbf{x}_j \right\rangle \\ &= \frac{1}{2} \sum_{i=1}^{n} \alpha_i y_i \left\langle \mathbf{x}_i, \sum_{j=1}^{n} \alpha_j y_j \mathbf{x}_j \right\rangle \\ &= \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \langle \mathbf{x}_i, \mathbf{x}_j \rangle \end{aligned} \tag{4.21}$$

Next, we fully expand the third term:

$$\begin{aligned} \sum_{i=1}^{n} \alpha_i \big(y_i (\langle \mathbf{x}_i, \mathbf{w} \rangle + b) - 1 + \xi_i \big) &= \sum_{i=1}^{n} \alpha_i y_i (\langle \mathbf{x}_i, \mathbf{w} \rangle + b) - \sum_{i=1}^{n} \alpha_i + \sum_{i=1}^{n} \alpha_i \xi_j \\ &= \sum_{i=1}^{n} \alpha_i y_i \langle \mathbf{x}_i, \mathbf{w} \rangle + \sum_{i=1}^{n} \alpha_i y_i b - \sum_{i=1}^{n} \alpha_i + \sum_{i=1}^{n} \alpha_i \xi_j \end{aligned}$$

and insert $\mathbf{w}$:

$$\begin{aligned}
&= \sum_{i=1}^{n} \alpha_i y_i \left\langle \mathbf{x}_i, \sum_{j=1}^{n} \alpha_j y_j \mathbf{x}_j \right\rangle + \sum_{i=1}^{n} \alpha_i y_i b - \sum_{i=1}^{n} \alpha_i + \sum_{i=1}^{n} \alpha_i \xi_j \\
&= \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \langle \mathbf{x}_i, \mathbf{x}_j \rangle + b \sum_{i=1}^{n} \alpha_i y_i - \sum_{i=1}^{n} \alpha_i + \sum_{i=1}^{n} \alpha_i \xi_i. \qquad (4.22)
\end{aligned}$$

There is nothing to do in the third term and plugging the expression for β_i into the fourth term gives

$$\begin{aligned}
\sum_{i=1}^{n} \beta_i \xi_i &= \sum_{i=1}^{n} (C - \alpha_i) \xi_i \\
&= C \sum_{i=1}^{n} \xi_i - \sum_{i=1}^{n} \alpha_i \xi_i. \qquad (4.23)
\end{aligned}$$

All that is left to do is to replace the left hand sides of Equations (4.21), (4.22), and (4.23) in the Lagrangian with the right hand sides. After erasing complementary terms and exploiting Equation (4.31), the formula will emerge in its final form.

$$\begin{aligned}
L(\mathbf{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\beta}) =& \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \langle \mathbf{x}_i, \mathbf{x}_j \rangle \\
&+ C \sum_{i=1}^{n} \xi_i \\
&- \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j (\langle \mathbf{x}_i, \mathbf{x}_j \rangle) - b \sum_{i=1}^{n} \alpha_i y_i + \sum_{i=1}^{n} \alpha_i - \sum_{i=1}^{n} \alpha_i \xi_i \\
&- C \sum_{i=1}^{n} \xi_i + \sum_{i=1}^{n} \alpha_i \xi_i. \\
=& \sum_{i+1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \langle \mathbf{x}_i, \mathbf{x}_j \rangle \qquad (4.24)
\end{aligned}$$

This allows us to rewrite the dual problem (4.17) equivalently as

$$\max_{\boldsymbol{\alpha}} \left(\sum_{i+1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \langle \mathbf{x}_i, \mathbf{x}_j \rangle \right), \qquad (4.25)$$

$$\text{with } 0 \leq \alpha_i \leq C, \quad \text{for } i = 1, \ldots, n, \qquad (4.26)$$

$$\text{and } \sum_{i=1}^{n} \alpha_i y_i = 0. \qquad (4.27)$$

This is remarkably close to the hard-margin Problem (4.5). The only difference is the upper bound C on α_i which is due to Equation (4.20) and the restriction $\beta_i \leq 0$, for $i = 1, \ldots, n$, on Lagrange multipliers for inequality constraints.

Analogously to the hard-margin case, this poses a quadratic optimization problem in $\boldsymbol{\alpha}$ with linear inequality constraints (Constraint (4.27) can be rewritten appropriately; we can simply express an equality $a = b$ by the two inequalities $a \leq b$ and $a \geq b$) which

can be solved efficiently. One algorithm designed specifically for this problem is the Sequential Minimal Optimization algorithm [42] (SMO). The SMO algorithm iteratively selects two components α_i, α_j of $\boldsymbol{\alpha}$ (heuristically) and maximizes (4.25) with respect to them. (The necessity to pick at least two components results from constraint (4.27).) The maximization with respect to two components can be done analytically, which reduces the numerical error in the computation. Fortunately, the algorithm also computes the value of b, as the slack variables make is unclear how the hard-margin approach should be translated to the soft-margin case.

Having found the optimal values $(\boldsymbol{\alpha}^*, b^*)$, we can compute $\mathbf{w}^*$ as in the hard-margin case and classify a new sample $\mathbf{x}$, also as in the hard-margin case, with

$$\text{class}(\mathbf{x}) = \text{sign}\left(\langle \mathbf{x}, \mathbf{w}^* \rangle + b^*\right). \tag{4.28}$$

4.1.3 Kernel Methods

This section introduces a generalization of support vector machines and only serves as an outlook.

Kernel methods are a generalization of the SVM principle. They allow nonlinear classification boundaries in the feature space with minute alteration of the already introduced SVM principle. Here, we will only consider the (more general) problem of soft margin SVMs. Remember that the optimization problem to find the parameters $\mathbf{w}$ and b of the best separating hyperplane could be stated by the dual Lagrangian (repetition of Formula (4.17)),

$$\max_{\boldsymbol{\alpha},\boldsymbol{\beta}} \min_{\mathbf{w},b,\boldsymbol{\xi}} L(\mathbf{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\beta}). \tag{4.29}$$

When looking for stationary points with respect to the inner minimization, we found

$$\nabla_{\mathbf{w}} L(\mathbf{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = 0 \quad \Longrightarrow \quad \mathbf{w} = \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i \tag{4.30}$$

$$\frac{\partial}{\partial b} L(\mathbf{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = 0 \quad \Longrightarrow \quad \sum_{i=1}^{n} \alpha_i y_i = 0 \tag{4.31}$$

$$\forall i\colon \ \frac{\partial}{\partial \xi_i} L(\mathbf{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = 0 \quad \Longrightarrow \quad \beta_i = C - \alpha_i \tag{4.32}$$

Plugging this back into the Lagrangian led to the optimization problem

$$\begin{aligned} &\max_{\boldsymbol{\alpha}} \left(\sum_{i+1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \langle \mathbf{x}_i, \mathbf{x}_j \rangle \right), \\ &\text{with } 0 \le \alpha_i \le C, \quad \text{for } i = 1, \ldots, n, \\ &\text{and } \sum_{i=1}^{n} \alpha_i y_i = 0, \end{aligned} \tag{4.33}$$

a quadratic optimization problem which can be solved efficiently, for example with the Sequential Minimal Optimization algorithm which also takes care of computing b.

Once we have the α_i and b, we can compute $\mathbf{w}$ by Equation (4.30) and classify a new

sample $\mathbf{x} \in \mathcal{X}$ by

$$\begin{aligned}\text{class}(\mathbf{x}) &= \text{sign}\left(\langle \mathbf{x}, \mathbf{w}\rangle + b\right)\\ &= \text{sign}\left(\left\langle \mathbf{x}, \sum_{i=0}^{n} \alpha_i y_i \mathbf{x}_i \right\rangle + b\right)\\ &= \text{sign}\left(\sum_{i=0}^{n} \alpha_i y_i \langle \mathbf{x}, \mathbf{x}_i\rangle + b\right).\end{aligned} \tag{4.34}$$

The interesting realization is that in the optimization problem (4.33) only depends on the training samples $\mathbf{x}_i$ in form of inner products $\langle \mathbf{x}_i, \mathbf{x}_j\rangle$. When we apply Equation (4.34) to classify a new previously unseen element $\mathbf{x}$ then, again, all we need are inner products $\langle \mathbf{x}, \mathbf{x}_i\rangle$ of elements from feature space $\mathcal{X}$. This is the key property behind kernel methods.

A general strategy, which can be applied by any classification method that is not restricted to one fixed dimensional input, is to use *embeddings into feature spaces*. The idea is to find a map $\varphi\colon \mathcal{X} \to \mathcal{X}'$ with some usually higher-dimensional feature space $\mathcal{X}'$ that conveniently spreads the training data. We call the target space $\mathcal{X}'$ the *intermediate space*. If the training sequence $T = ((x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n))$ does not allow for a clean separation by the chosen algorithm, then maybe its embedding $T' = ((\varphi(x_1), y_1), (\varphi(x_2), y_2), \ldots, (\varphi(x_n), y_n))$ does. Assume that $\text{classify}_{\varphi}(\cdot)$ is a classification algorithm that was successfully trained on T'. How would we use $\text{classify}_{\varphi}(\cdot)$ to label a new element $\mathbf{x}$? Since the classifier lives in the intermediate space, we would call it with $\varphi(\mathbf{x})$ and assign the label $\text{classify}_{\varphi}(\varphi(\mathbf{x}))$ to $\mathbf{x}$.

One problem of this approach is that if the intermediate space has very high dimensionality, computation of $\varphi(\mathbf{x})$ might become inefficient, even infeasible. Kernel methods avoid this problem by the realization that the SVM method never actually needs any isolated samples. Only inner products of pairs of samples and these are just single numbers. If we know how to compute $\langle \varphi(\mathbf{x}), \varphi(\mathbf{x}')\rangle$ directly from $\mathbf{x}$ and $\mathbf{x}'$, that is, if we know a function $\kappa\colon \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ such that for all $\mathbf{x}$, $\mathbf{x}'$ in $\mathcal{X}$ we have $\kappa(\mathbf{x}, \mathbf{x}') = \langle \varphi(\mathbf{x}), \varphi(\mathbf{x}')\rangle$, then we never need to actually apply φ, and we never need to actually handle any element in $\mathcal{X}'$. We can go one step further. There is no need to know how $\mathcal{X}'$ looks like. As long as we know that there exists some intermediate (Hilbert) space such that the chosen function κ corresponds to the inner product in this space, we can apply κ. This is called the *kernel trick* and κ is called a *kernel*. All we need to adjust is the minimization problem (4.33):

$$\begin{aligned}&\max_{\boldsymbol{\alpha}} \left(\sum_{i+1}^{n} \alpha_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \kappa(\mathbf{x}_i, \mathbf{x}_j)\right),\\ &\text{with } 0 \le \alpha_i \le C, \quad \text{for } i = 1, \ldots, n,\\ &\text{and } \sum_{i=1}^{n} \alpha_i y_i = 0,\end{aligned} \tag{4.35}$$

and the classification (4.34):

$$\text{class}(\mathbf{x}) = \text{sign}\left(\sum_{i=0}^{n} \alpha_i y_i \kappa(\mathbf{x}_i, \mathbf{x}_j) + b\right). \tag{4.36}$$

But effectively, we have now created a classifier in some, possibly infinite-dimensional, intermediate space. One example for this is the *Radial Basis Function* (RBF) or *Gaussian* kernel,

$$\kappa_{RBF}(\mathbf{x}, \mathbf{x}') = e^{-\gamma(\mathbf{x}-\mathbf{x}')^2}. \tag{4.37}$$

For small difference vectors $\mathbf{x} - \mathbf{x}'$, the RBF kernel approaches 1 and is identical to 1 iff $\mathbf{x} = \mathbf{x}'$. For more and more dissimilar vectors, $\|\mathbf{x} - \mathbf{x}'\|$ grows and the RBF kernel approaches zero. Therefore, it can be interpreted as an isotropic similarity measure. A tedious but necessary task is to confirm that there exists a map φ to an intermediate space such that $\kappa(\mathbf{x}, \mathbf{x}') = \langle \varphi(\mathbf{x}), \varphi(\mathbf{x}') \rangle$. Here, we will simply describe the map φ, following an exercise in Ref. [52, p. 221]. In the following part, we will make use of the identity $e^x = \sum_{n=0}^{\infty} x^n/n!$ and the fact that for every $\mathbf{x}$, $\mathbf{x}' \in \mathbb{R}^p$:

$$\langle \mathbf{x}, \mathbf{x}' \rangle^n = \sum_{J \in \{1,2,\ldots,p\}^n} \prod_{i=0}^{n} x_{J_i} x'_{J_i}. \tag{4.38}$$

Assume that φ is a map such that for every $n \in \mathbb{N}$ and every $J \in \{1, 2, \ldots, p\}^n$ there exists a component, say the $c(n, J)$-th component, such that

$$\varphi(\mathbf{x})_{c(n,J)} = \frac{1}{\sqrt{n!}} e^{-\frac{\|\mathbf{x}\|^2}{2}} \prod_{i=0}^{n} \mathbf{x}_{J_i}. \tag{4.39}$$

Then,

$$\begin{aligned}
\underline{\langle \varphi(\mathbf{x}), \varphi(\mathbf{x}') \rangle} &= \sum_{\substack{n \in \mathbb{N} \\ J \in \{1,..,p\}^n}} \varphi(\mathbf{x})_{c(n,J)} \varphi(\mathbf{x}')_{c(n,J)} \\
&= \sum_{\substack{n \in \mathbb{N} \\ J \in \{1,..,p\}^n}} \left(\frac{1}{\sqrt{n!}} e^{-\frac{\|\mathbf{x}\|^2}{2}} \prod_{i=0}^{n} \mathbf{x}_{J_i} \right) \left(\frac{1}{\sqrt{n!}} e^{-\frac{\|\mathbf{x}'\|^2}{2}} \prod_{i=0}^{n} \mathbf{x}'_{J_i} \right) \\
&= \sum_{n \in \mathbb{N}} \sum_{J \in \{1,..,p\}^n} \left(\frac{1}{\sqrt{n!}} e^{-\frac{\|\mathbf{x}\|^2}{2}} \prod_{i=0}^{n} \mathbf{x}_{J_i} \right) \left(\frac{1}{\sqrt{n!}} e^{-\frac{\|\mathbf{x}'\|^2}{2}} \prod_{i=0}^{n} \mathbf{x}'_{J_i} \right) \\
&= e^{-\frac{\|\mathbf{x}\|^2 + \|\mathbf{x}'\|^2}{2}} \sum_{n \in \mathbb{N}} \frac{1}{n!} \sum_{J \in \{1,..,p\}^n} \left(\prod_{i=0}^{n} \mathbf{x}_{J_i} \right) \left(\prod_{i=0}^{n} \mathbf{x}'_{J_i} \right) \\
&= e^{-\frac{\|\mathbf{x}\|^2 + \|\mathbf{x}'\|^2}{2}} \sum_{n \in \mathbb{N}} \frac{1}{n!} \underbrace{\sum_{J \in \{1,..,p\}^n} \prod_{i=0}^{n} \mathbf{x}_{J_i} \mathbf{x}'_{J_i}}_{\overset{\text{Eq.(4.38)}}{=} \langle \mathbf{x}, \mathbf{x}' \rangle^n} \\
&= e^{-\frac{\|\mathbf{x}\|^2 + \|\mathbf{x}'\|^2}{2}} \sum_{n \in \mathbb{N}} \frac{\langle \mathbf{x}, \mathbf{x}' \rangle^n}{n!} \\
&= e^{-\frac{\|\mathbf{x}\|^2 + \|\mathbf{x}'\|^2}{2}} e^{\langle \mathbf{x}, \mathbf{x}' \rangle} \\
&= e^{-\frac{1}{2}(\|\mathbf{x}\|^2 + \|\mathbf{x}'\|^2 - 2\langle \mathbf{x}, \mathbf{x}' \rangle)} \\
&= \underline{e^{-\frac{(\mathbf{x}-\mathbf{x}')^2}{2}}}.
\end{aligned} \tag{4.40}$$

If we want to find the map φ for $e^{-\gamma(\mathbf{x}-\mathbf{x}')^2}$ we adjust the component in Equation (4.39) to

$$\varphi(\mathbf{x})_{c(n,J)} = \frac{1}{\sqrt{n!}} e^{-\frac{\|\mathbf{x}\|^2\gamma}{2}} \prod_{i=0}^{n} \mathbf{x}_{J_i} \sqrt{2\gamma}, \tag{4.41}$$

and carry out the same computation as above. Thus, we have shown that the RBF kernel is a valid kernel function.

A characterization of valid kernels is given in Ref. [52, p. 222], which is described as a simplification of *Mercers condition* (see for example Ref. [13]). The book by Schölkopf and Smola [51, 405ff.] gives practical tips on how to create valid kernels.

4.1.4 Software

We have used the Scikit-learn package [41] of the SciPy library [29]. For SVMs without kernel, the LinearSVC class was used which relies on liblinear [15] to solve the inner optimizations.

4.2 Parameter Exploration and Insights

The experiments in each section of this part are similar in setup and purpose to those in the decision tree part (Section 3.2). To reduce redundancy, we will not re-motivate each experiment but only those parts that differ from the previous ones. Any detail not listed, such as the training set used or the particular sample sizes, is the same as in the corresponding decision tree experiments. We will make sure to mention any deviation explicitly.

4.2.1 Addressing the class imbalance

First, we analyze the sensitivity of support vector machine classifiers to imbalanced inputs and compare the results to under- and oversampled training sets. Figure 4.4 shows the results of training 100 classifiers for each of the strategies to tackle the imbalance (see Section 2.5.2). The KDE plots to the left show significantly wider distributions than in the decision tree setting. We see that in case of imbalanced training sets the majority class is likely to be preferred. Under- and oversampling seem to perform similarly. The box plots to the right show median and quartiles of the classifiers' min-accuracies. We get the following median min-accuracy values:

strategy	min-acc.
oversampling	0.575
imbalanced	0.305
undersampling	0.570

The difference between over- and under-sampling seems to be insignificant if we consider the width of the confidence intervals in Figure 4.4. Thus, we will consider both strategies in the first experiment of the next section.

4.2.2 Preprocessing

This section compares the different preprocessing techniques. We consider no preprocessing, per-feature standardization, and per-image standardization. Each technique will

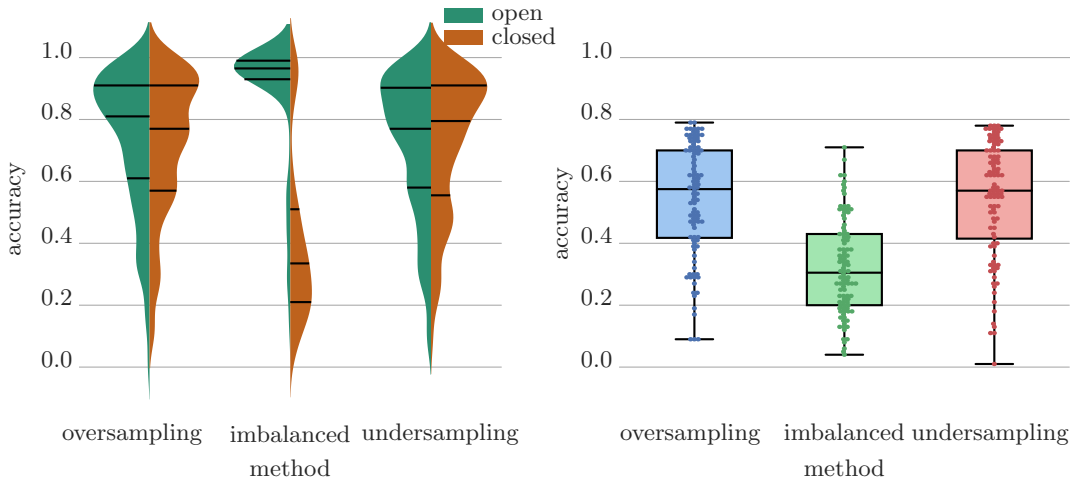


Figure 4.4: Accuracy values for 100 trained classifiers per input strategy. Left: KDEs for per-class accuracies. The plot make the amount of variation inside each strategy apparent. Right: Box plots of the min-accuracy values showing median and quartiles. The actual min-accuracy values are drawn as dots on top.

be tested with and without subsequent PCA transformation, thus we have six techniques to compare. We will do this for both image- and feature-based input. Again, we consider the two input-formats,

1. `2d_slice1_sample_64_small`, and
2. `2d_slice4_sample_1`.

For the experiments in this section, we fix the trade-off parameter at $C = 1$. The previous section indicated that imbalanced inputs lead to imbalanced performance in the per-class accuracy but was inconclusive regarding the question whether over- or under-sampling should be preferred. For further analysis we ran the first experiment with both sampling strategies. We trained 100 image-based classifiers for each of the six preprocessing techniques on format 1, listed above. Figure 4.5 shows the results. The application of preprocessing reveals substantial performance imbalances when training is performed oversampled input. Although oversampling significantly equalizes the per-class accuracies when compared with imbalanced inputs (see Fig. 4.4), the minority class is still under-represented.

This result matches the intuition: A slight amount of equalization can be explained by the increased penalty of falsely classifying an element $\mathbf{x}$ from the minority class. The penalty is proportional to the number of duplicates of $\mathbf{x}$ that exist in the oversampled set. This might prevent the classifier from letting the decision region invade the area claimed by minority samples, that is, its convex hull. But the actual area itself in feature space still remains the same. The majority class, on the other hand, might claim a larger area due to its independent samples which results, out of the two classes, in better generalization performance.

Therefore, we chose to balance the training sets for subsequent support vector machine classifiers by undersampling. Figure 4.6 shows the results of training 100 classifiers for each of the six preprocessing techniques and both input formats, trained on image-based input. We consistently find the poorest performance when we do not apply any preprocessing at all. PCA transformation alone is the second worst technique for format

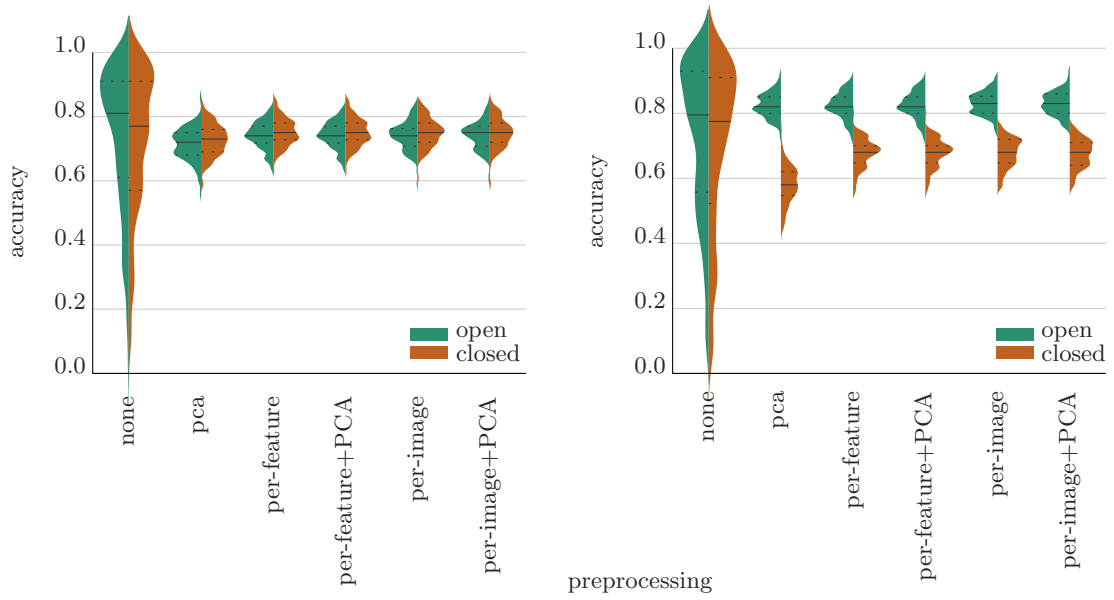


Figure 4.5: KDEs of per-class accuracies for the six different preprocessing techniques obtained from 100 classifiers trained on image-based input. Solid lines show the median, dotted lines the quartile accuracies. Left: Results for undersampling. Right: Results for oversampling.

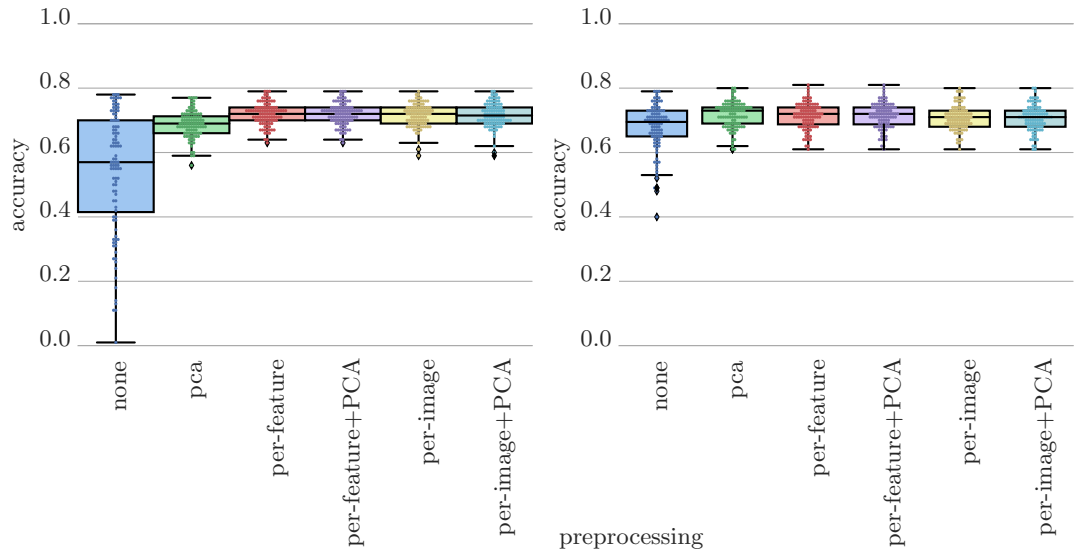


Figure 4.6: Min-accuracies for 100 support vector machine classifiers per preprocessing technique trained on image-based input. Left: Trained and validated on format`2d_slice1_sample_64_small`. Right: Trained on format `2d_slice4_sample_1`.

`2d_slice1_sample_64_small` and (slightly) the best for format `2d_slice4_sample_1`. In general, apart from not applying any preprocessing, the techniques seem to give similar results. The noticeably weaker performance of pure PCA transformation on format `2d_slice1_sample_64_small` excludes it from further consideration. Both, per-feature and per-image standardization seem to perform similar, regardless of whether it is fol-

lowed by PCA transformation or not. We choose to continue with per-feature standardization, as its median min-accuracy is slightly in the lead and the confidence intervals show slightly less variance, but the difference seems to be negligible and we might as well have decided for per-image standardization.

Results for the feature-based approach are shown in Figure 4.7, where we changed the plotting style to account for the increased variation. The plots show that there is no-

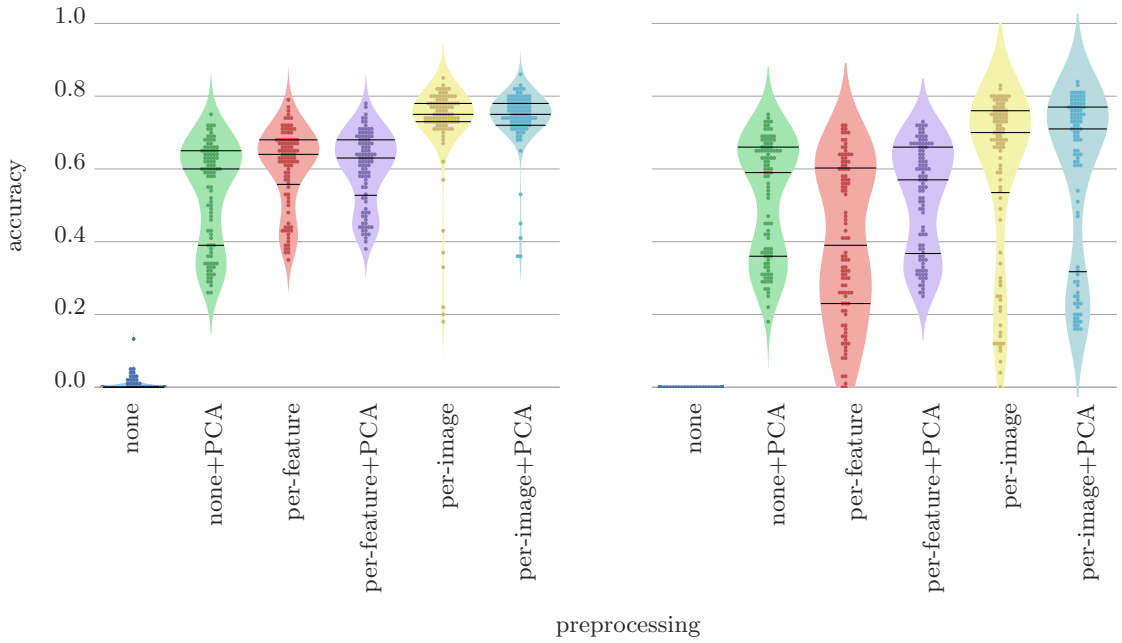


Figure 4.7: Min-accuracies for 100 support vector machine classifiers per preprocessing technique; classifiers were trained on extracted features. Left: Trained and validated on format`2d_slice1_sample_64_small`. Right: Trained on format `2d_slice4_sample_1`.

tably more performance variation when the classifiers are trained on the feature-based input compared with those trained on the image-based input. This holds for the median min-accuracy among different preprocessing techniques but, as indicated by the wide confidence intervals, also for min-accuracies inside one technique. For most configurations a second mode exist indicating imbalanced performance.

Not performing any preprocessing gives by far the worst results. All min-accuracies are zero or close to it, indicating a heavily imbalanced per-class performance. Here, PCA transformation alone already lifts the performance to a level competitive to per-feature standardization.

Features computed from per-image standardized images appear to give the best performance. They show some of the most extreme outliers but apart from that the main bulk of min-accuracies lies relatively close. Moreover, subsequent PCA transformation seems to further reduce the variance slightly.

4.2.3 Where does the performance variation come from?

The great amount of performance variation, especially in the feature-based experiments, poses the question of its origin. There are two sources of randomness.

1. Data selection: For each classifier, we select a random subset of labeled data to use, and randomly split off a validation set.

2. Optimization: The SMO-algorithm to solve the Lagrangian optimization in the support vector machine learning method randomly chooses pairs of Lagrange multipliers for joint optimization. This leads to different computation paths and, possibly, to decision boundaries with different generalization performances.

We assume that the variation is mainly a product of the first source. To gain more insights, we ran a new series of experiments with setup identical to the previous section but fixed the random seeds for a) none of the two sources, b) the support vector machine function call, hence, the internal optimization, and c) the data selection. For these experiments, we restrict to feature-based input computed from format `2d_slice1_sample_64_small`. Figure 4.8 shows the resulting min-accuracies. The plots labeled a) reproduce the results of the left plot in Figure 4.7. The plots labeled with b) show that fixing the internal optimization does not seem to reduce the performance variation at all if the subsets for training and validation are still chosen randomly. For comparison, the plots labeled c) show significant reduction in performance variation, except for when no preprocessing is applied. Interestingly there still remains a wide performance range.

Although we identify the main contributor to be the random data selection, we see that the randomness in the internal optimization has non-negligible effect on the quality of the resulting classifier. The left plot in Figure 4.9 shows the resulting min-accuracies when we fix the random seeds for both sources across different training sessions. There is no performance variation anymore and we conclude that the two mentioned sources are indeed the only sources.

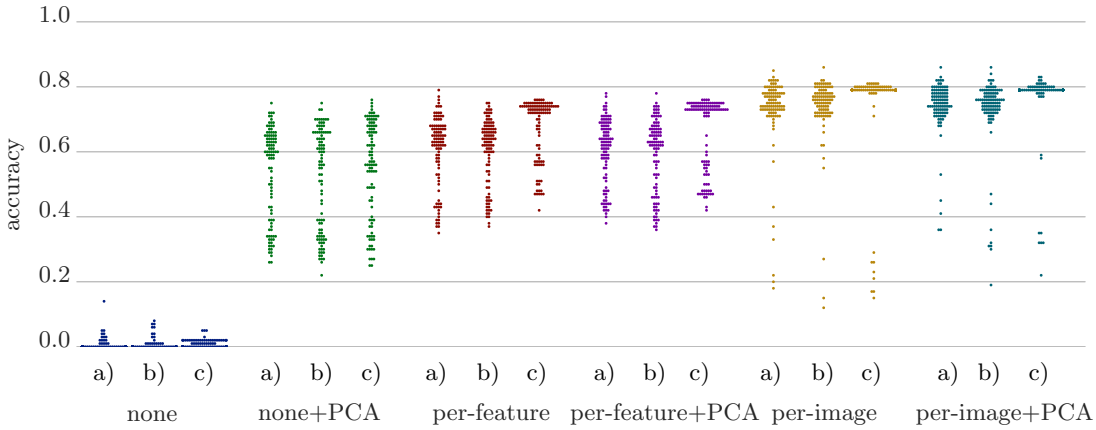


Figure 4.8: Min-accuracies for 100 classifiers per preprocessing technique. Fixed are random seeds to a) not fixed b) the support vector machine function call c) the training and validation set generation.

Having detected the data selection as main source of performance fluctuation, it would be interesting to see whether there are subsets that are generally non-representative performance regardless of the preprocessing technique or if every technique has its own weak subsets. In case of the still relevant optimization algorithm, it would be interesting if some computation paths generally perform better than others.

In the right part of Figure 4.9 we show the per-class accuracies of the classifiers whose min-accuracies were shown in Figure 4.8 b),c). We connected dots that got identical

random seeds for the source of randomness that we did not fix. For the top row we fixed the training and validation sets. Connected dots got the same random seeds in the support vector machine function call. For the bottom row it is the other was around.

Generally badly performing subsets or computational paths would result in an accumulation of lines, with their endpoints in the same relative performance level across different preprocessing techniques. It seems that each technique has its own weak subsets. For every technique we find points among the weakest performances that were generated by a data-optimization pair of random seeds that worked well for the neighbouring techniques or vice versa. The same statement holds for the plots in the top row.

Thus, deterministically selecting well performing data subsets and computational paths in the optimization to control the observed variation would require further investigation.

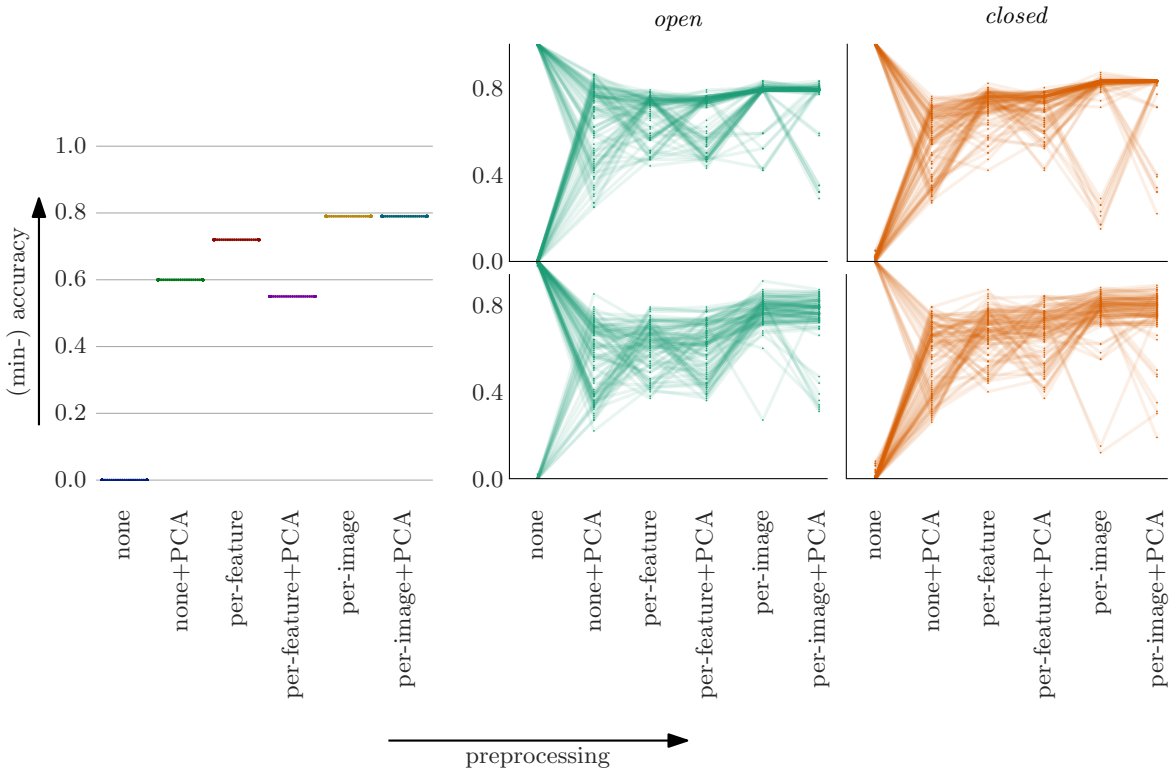


Figure 4.9: Left: Min-accuracies for 100 classifiers per preprocessing technique when random seeds for both sources are fixed. Right: Performance variation for 100 classifiers when random seeds for one source is fixed. Connected dots got the same seeds for the other sourcea. Top: Fixed datasets, variation due to optimization. Bottom: Fixed optimization, variation due to datasets.

4.2.4 Remaining Parameters

The first experiments revealed that we should balance the data by undersampling, preprocess image-based input by per-feature standardization, and compute the features for feature-based input from per-image standardized images, possibly followed by PCA transformation. In this chapter, we will fix the remaining parameters, that is, which input format to chose (eight possibilities), whether the input should be image-based,

feature-based or feature-based with PCA transformation, and what value the trade-off variable C should take. We do not have any bounds for the latter, hence, we scan a wide range with an exponentially increasing step width. For image-based inputs, we try $C = 1 \cdot 10^{-8}, 5 \cdot 10^{-8}, 1 \cdot 10^{-7}, 5 \cdot 10^{-7}, \ldots, 1, 5$; for feature-based inputs, we use the same pattern but start at $1 \cdot 10^{-5}$. These ranges were found to include the interesting C values.

Some of the following results show effects that directly correlate with the number of input dimensions they were obtained from. The following table orders the available input formats according to their dimensionality.

dim.	input format
861	`2d_slice1_sample_X_small`
3321	`2d_slice1_sample_X`
3444	`2d_slice4_sample_X_small`
13284	`2d_slice4_sample_X`

Figure 4.10 shows the median min-accuracies obtained from training 100 classifiers for both image- and feature-based inputs for all input formats as well as C values. The latter is shown with and without additional PCA transformation. Also, note that the x-axis is scaled logarithmically.
Similar to the decision tree case, the best feature-based approaches surpass the best image-based ones by roughly 5%.

First, we focus on the image-based results. As expected, increasing the C value, that is, the penalty for margin violations in the training data, leads to an increase in the training performance. For smaller values than $5 \cdot 10^{-7}$, a performance increase is not detected.

The training curves show four distinct clusters. Each cluster contains two formats with identical dimensionality. Some curves are steeper an lie above others, thus, show better training performance per fixed C. When arranged according to slope, we see that those with better training performance belong to classifiers trained on higher dimensional inputs. The training training performance take-off itself appears to start earlier, as well. For large enough C values – at least for classifiers trained on higher dimensional inputs – the training data is separated perfectly. These phenomenon agrees with the intuition of higher dimensional input being better separated, especially in our case considering the large amount of noise in the data. Unfortunately, separation due to noise does not generalize well. This is illustrated by the validation performance. After an initial increase, the earliest start to stagnate or drop-off at $C = 5 \cdot 10^{-6}$. For a fixed C value, the validation curves do not show the same clustering as the training curves.

Next, we focus on the feature-based results. Here, for the 3-dimensional inputs, training and validation performance is noticeably closer than in the high dimensional images-based setting. Even the best training curves do not surpass a min-accuracy of 0.8.

Comparing the two feature-based approaches, we see that the most prominent difference occurs at small C values, where subsequent PCA transformation seems to equalize both training and validation performance for all input formats. In general, the feature-based approaches show less performance dynamics paired with superior results. Interestingly, for C values larger than 0.1 without PCA transformation or 0.001 with PCA transformation the training curves start do decrease again. Since we are only looking at the worse of the two per-class accuracies, it is likely that this is compensated for by the accuracy on the other class.

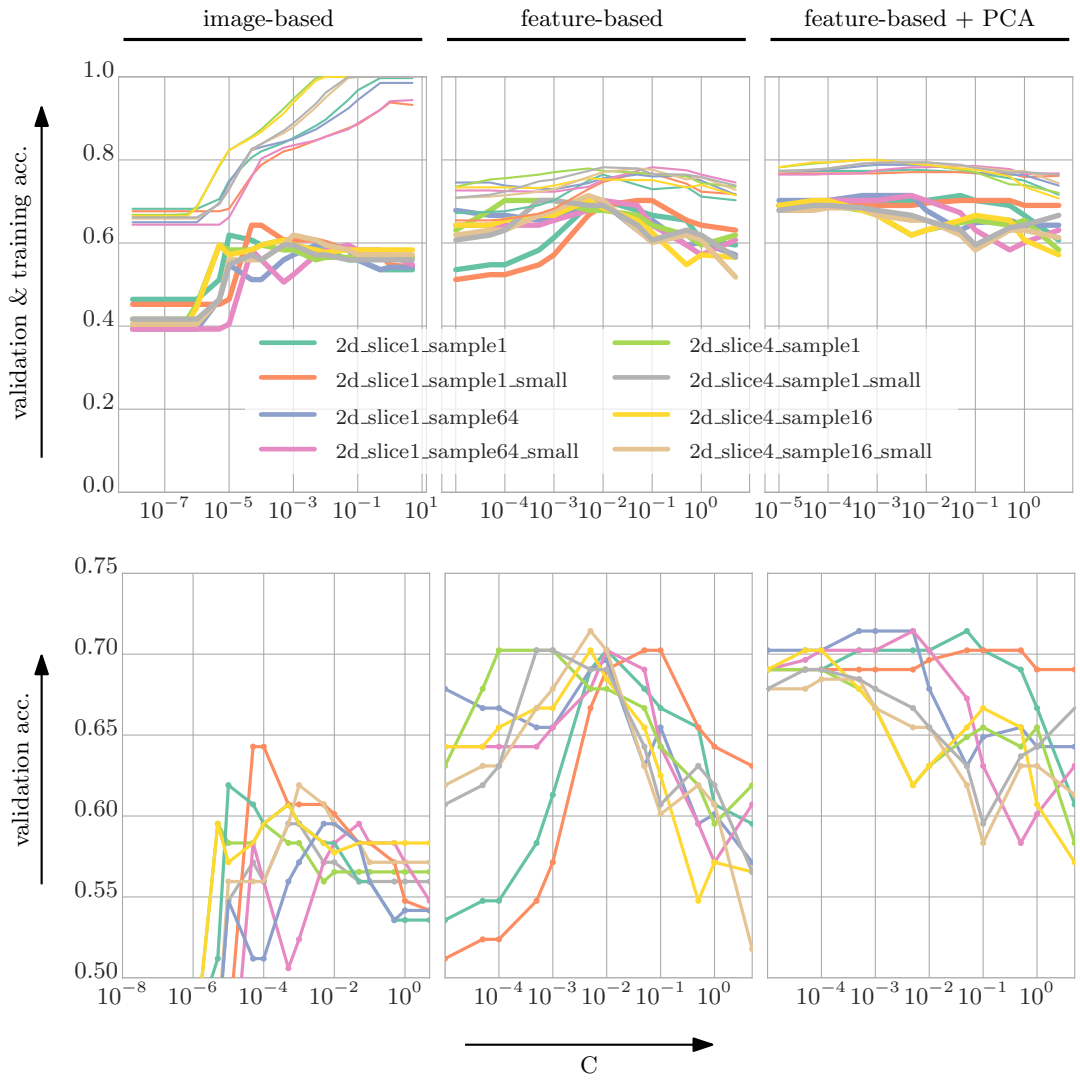


Figure 4.10: Median validation and training min-accuracies for 100 classifiers per input format, and C value. The x axis is scaled logarithmically. The discrete values are linearly interpolated for an easier visual comprehension.

Figure 4.11 gives a more detailed view on the validation min-accuracies. We see that the large performance variation we witnessed earlier, for feature-based input from formats `2d_slice1_sample_64small` and `2d_slice4_sample_1` with $C = 1$, only occurs in the feature-based approach and for large C values. All other scenarios show only little performance variation. PCA transformation after feature extraction appears to increase the performance for small C values in all tested cases.

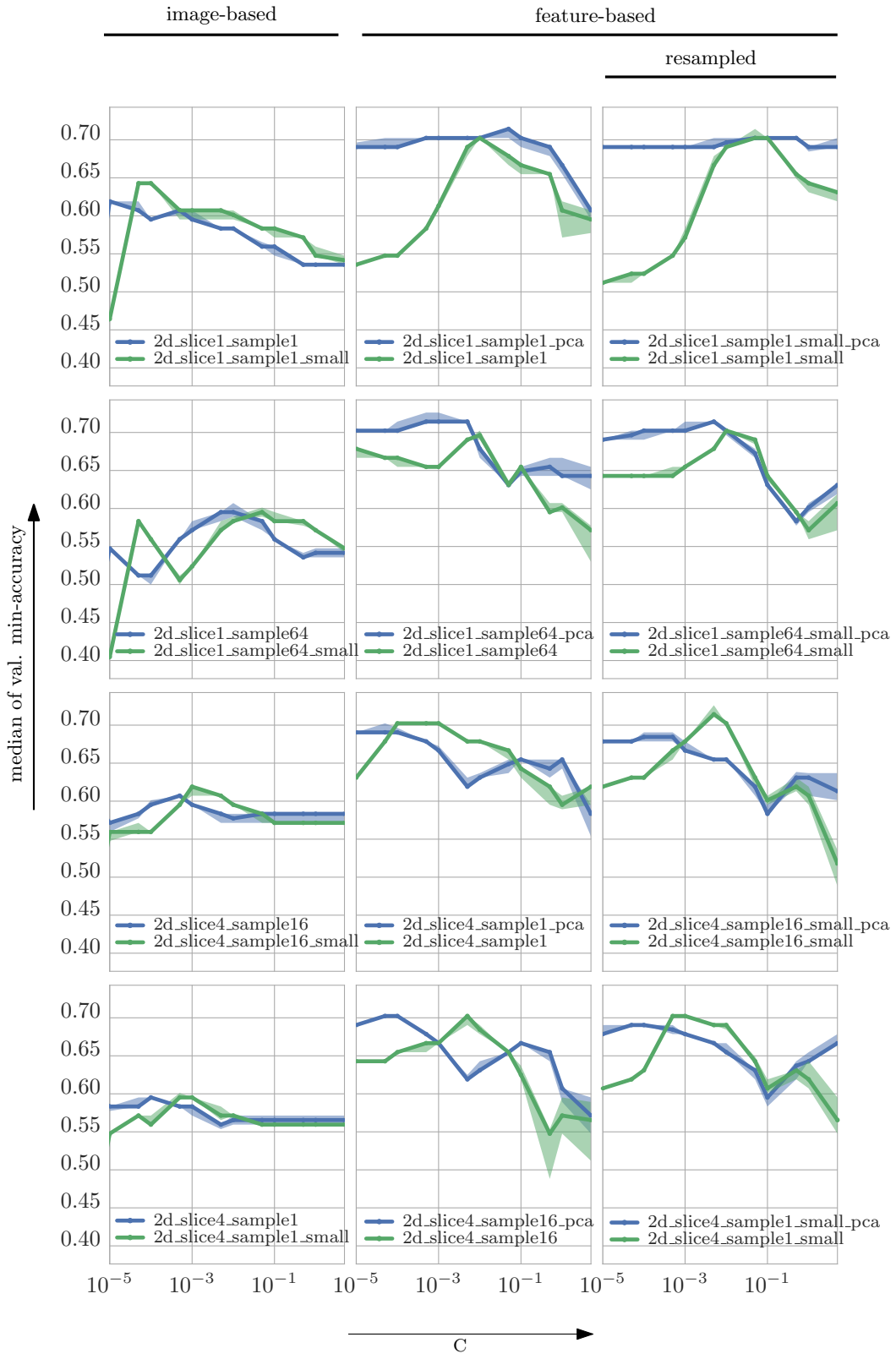


Figure 4.11: Isolated plots of median validation min-accuracies. Transparent bands indicate the 50% confidence intervals. The discrete points are linearly interpolated, and the axis is logarithmically scaled.

The best performing configurations for the image- and feature-based approaches (the latter with and without PCA transformation) are listed in Table 4.1. Similar to the decision tree, we see that the feature-based approach gives the best results.

origin	format	C	accuracy	count
image-based	`2d_slice1_sample_1_small`	$5 \cdot 10^{-5}$	0.64	2
feature-based	`2d_slice4_sample_16_small`	0.005	0.71	1
feature-based+PCA	`2d_slice1_sample_1`	0.05	0.71	1
feature-based+PCA	`2d_slice1_sample_64`	0.0005-0.005	**0.71**	**3**
feature-based+PCA	`2d_slice1_sample_64_small`	0.005	0.71	1

Table 4.1: Best performing configurations for image- and feature-based approaches. Column 'accuray' shows the best median min-accuracy, column 'count' lists how often the best value was reached while varying the C value.

4.2.5 Adding Principle Components

Adding projections along the 25 most dominant principle components of the training data in an attempt to further separate the data showed similar results to the decision tree: An increase in training performance but a decrease in validation performance.

4.3 Final training and Results

The final support vector machine classifier was trained on the union of balanced undersampled subset of datasets ds1, ds2, and ds3, comprising all closed endpoints. The inputs took the form of features extracted from per-image standardized images of format `2d_slice1_sample_64` with subsequent PCA transformation. The final value for C was 0.0005. On the training set, the achieved per-class accuracies are

$$0.79 \text{ } (\textit{open})$$
$$0.79 \text{ } (\textit{closed}).$$

The resulting normalized weight vector $\mathbf{w}$ and threshold b are

$$\mathbf{w} \approx (-0.062, -0.991, 0.121)^T, \quad b \approx -5 \cdot 10^4.$$

For an easier comprehension, we normalized $\mathbf{w}$ and multiplied b by $|\mathbf{w}|$ to prevent shifting the decision boundary. We see that the decision boundary is almost perpendicular to the second feature axis, thus, relies almost exclusively on the second feature. The value for b is close to the theoretical threshold of 0.

On the test set comprising all non-contradictory samples from qs1, qs2, qs3, and qs4 (in total 1025 *open* and 336 *closed* elements), the per class accuracies obtained are

$$\mathbf{0.66} \text{ } (\textit{open})$$
$$\mathbf{0.70} \text{ } (\textit{closed}).$$

Figure 4.12 shows the first 10 samples per class that were misclassified by the support vector machine.

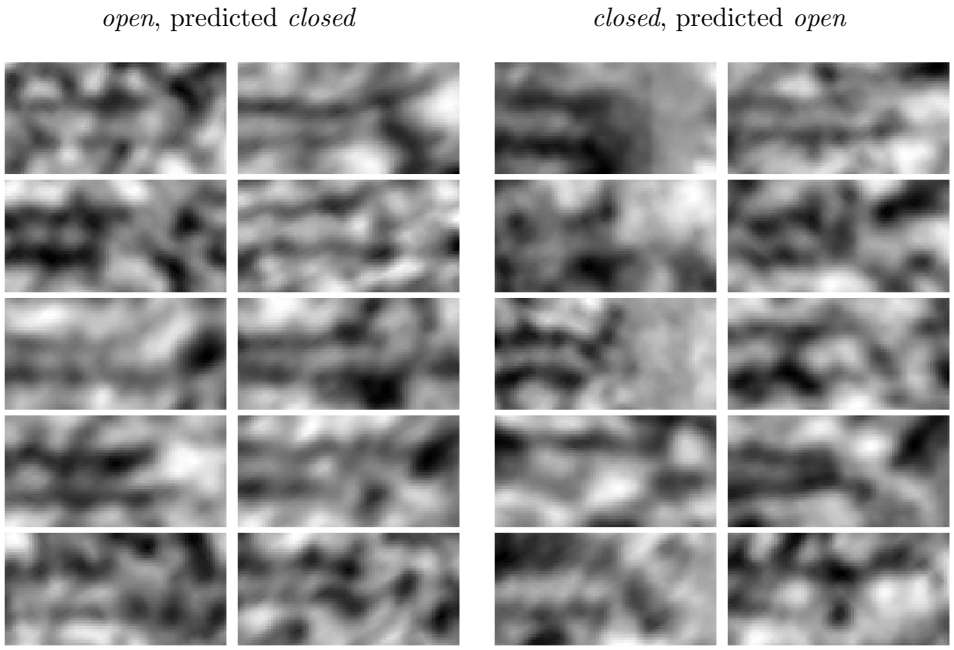


Figure 4.12: Examples of elements misclassified by the final support vector machine.

5 Neural Networks

5.1 Basic Principles

Neural networks, or *neural nets* for short, are implementations of nested functions. The input will be transformed in one or more steps to the final output. This transformation can be visualized by a *computation graph*, a directed graph in which every edge and every vertex stands for the application of a mathematical operation. We will say that the graph structure *stores* the corresponding operation. Input to the neural net starts at designated input vertices of the computational graph and moves along the edges from vertex to vertex until a designated output vertex is reached. The values at all output vertices form the output of the neural net. While moving through the graph, the input values are transformed according to the operations stored in the visited edges and vertices. These operations can be elementary, like an addition or multiplication with a predefined value, but they might as well encapsulate the application of more complicated functions. Sometimes it might be illustrative to replace a graph structure that corresponds to a more intricate mathematical operation by a sub-graph that makes the composite steps of the operation more apparent. In other cases, unifying a sub-graph to a single node that represents the same transformation might assist the clarity of the exposition.

While this visual interpretation greatly helps in the understanding of neural networks, the actual implementation can be done highly efficient and condensed in form of vector, matrix, and tensor operations. Main resource for the topic in this chapter is the book by Goodfellow, Bengio, and Courville [21]. For a historical overview of neural network research and (probably one of the most complete reference lists) we refer to the article by Schmidhuber [50].

5.1.1 Feedforward Neural Networks

In this first part we describe the basic form of a *fully-connected feedforward* neural network. Here, the vertices of the computational graph can be arranged in layers $L^1, L^2, \ldots, L^d$ such that all edges are directed from one layer to the next. This explains the feedforward part of the name. All input vertices form the first layer L^0, and there are as many input vertices as there are components in a single input vector. The last layer L^d comprises all output vertices of which there can be as many as needed. Each vertex of layer L^i, for $i = 0, 1, \ldots, d-1$, has an outgoing edge to every vertex of layer L^{i+1}, that is, if we set aside the edge orientation, layers L^i and L^{i+1} form a fully connected bipartite graph. This explains the fully-connected part of the name. Layers $L^1, \ldots, L^{d-1}$ are called the *hidden layers*. Opposed to the input and output layers, they would not need to be exposed in a black box model of the network. The *depth* of a network refers to either the number of total computational layers (which excludes the input layer) or the number of hidden layers; unfortunately there is no general consensus on this in the literature. In this thesis we will refer to d as the depth of a network, that is, the number of computational layers including the output layer. An example graph for a fully-connected feedforward neural net is given in Figure 5.1. The sub-graph drawn

in red is called a *neuron*, and was introduced under the name *perceptron*, a historically important predecessor of neural networks, by Rosenblatt in 1957 [48].

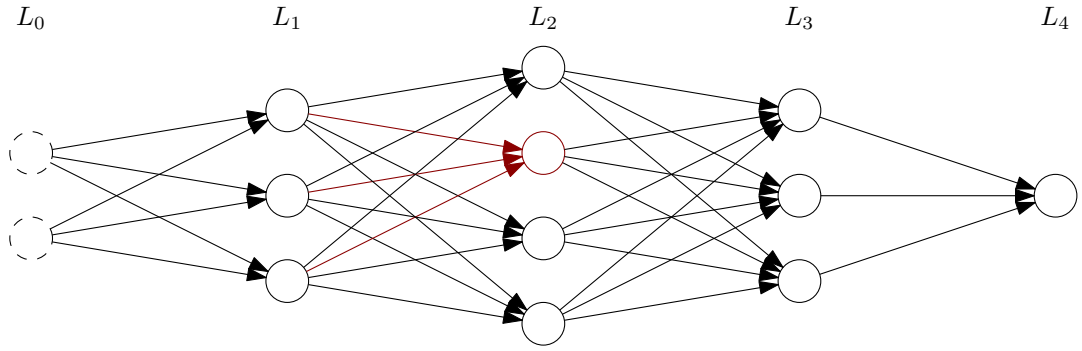


Figure 5.1: Example topology of a fully-connected feedforward neural network of depth $d = 4$.

We use the perceptron to introduce the different operations that will be encountered when feeding an input to the neural net. An example perceptron is given in Figure 5.2.

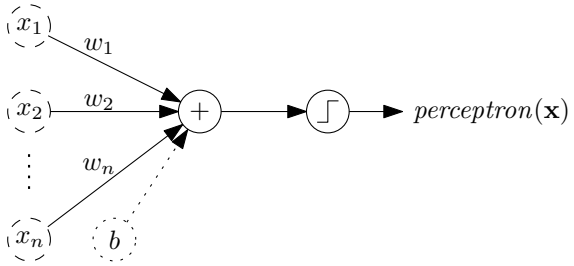


Figure 5.2: Computational graph of a perceptron. Each edge stores a weight that will be multiplied by the value that traverses it. The single computational node is the only output node and takes the sum over all incoming values and an implicit bias term b.

Again, there are as many input vertices as there are components in the input vector, but there will only be a single output vertex. Thus, for an n-dimensional input, the perceptron realizes a map $\mathbb{R}^n \to \mathbb{R}$. Each edge stores a weight w_i that will be multiplied by the value that traverses it. The first vertex takes the sum over all incoming values and one implicit additional value b. The next vertex applies a threshold function and produces the output. This defines the function *perceptron* as

$$perceptron(\mathbf{x}) = \begin{cases} 0 & \text{if } \langle \mathbf{x}, \mathbf{w} \rangle + b < 0, \\ 1 & \text{if } \langle \mathbf{x}, \mathbf{w} \rangle + b \geq 0, \end{cases} \qquad \text{where } \mathbf{w} = (w_1, w_2, \ldots, w_n). \tag{5.1}$$

Equation (5.1) describes a classifier that separates the two sides of a hyperplane defined by $\mathbf{w}$ and b.

In a *neuron*, we allow arbitrary nonlinear functions σ in place of the threshold function. We call it the *activation* function, as illustrated in Figure 5.3.
The function performed by a neuron is then

$$neuron(\mathbf{x}) = \sigma\big(\langle \mathbf{x}, \mathbf{w} \rangle + b\big), \quad \text{where } \mathbf{w} = (w_1, w_2, \ldots, w_n). \tag{5.2}$$

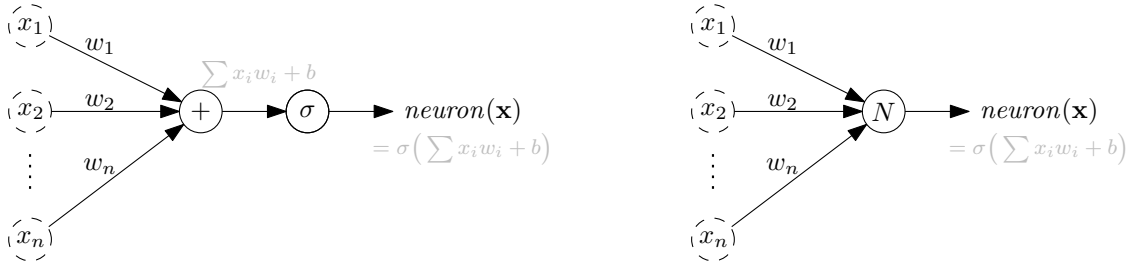


Figure 5.3: Computational graph of a neuron, the building block of a neural network. Left: The summation is followed by a nonlinear activation function $\sigma\colon \mathbb{R} \to \mathbb{R}$. Right: The summation and activation function are often condensed into a single vertex, here, labeled N.

A popular choice for the activation function is the *rectifying linear unit* (*ReLU*), a piecewise linear unit defined by

$$ReLU(x) = \max(0, x).$$

Glorot, Bordes, and Bengio [20] showed that it outperformed the previously most popular activations like the sigmoid and tanh functions and greatly improved the training speed. A practical consideration is that the ReLU function does not have a derivative at 0, as can be seen in Figure 5.4. In case the derivative at 0 is required, it is common to arbitrarily return the left or right derivative, that is, 0 or 1.

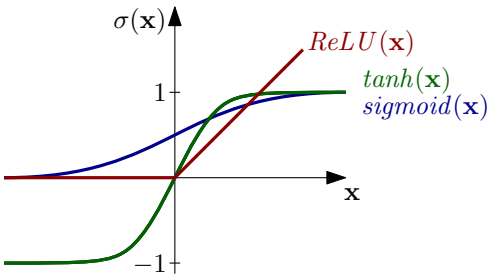


Figure 5.4: Sketches of different activation functions.

Each layer of a fully-connected feedforward neural network can be interpreted as a stack of neurons. Usually, they all have their individual weights and bias terms but perform the same kind of activation. It is notationally convenient to think of the input and output of layers in terms of vectors: a layer L^ℓ gets an input vector $\mathbf{o}^{\ell-1}$ from the previous layer and creates an output vector $\mathbf{o}^\ell$; each neuron creates one component of its layer's output vector. We can think of the order of the components as arbitrary but fixed and denote the j-th neuron in layer L^ℓ, that is, the neuron computing o_j^ℓ, as N_j^ℓ. For every pair of neurons $N_i^{\ell-1}, N_j^\ell$ of adjacent layers we have a connecting edge, and its weight will be w_{ij}^ℓ. This allows us to write the output value of N_j^ℓ as

$$o_j^\ell = \sigma\left(\sum_{i=1}^{n} o_i^{\ell-1} w_{ij}^\ell + b_j^\ell\right) = \sigma\left(\left\langle \mathbf{o}^{\ell-1}, \mathbf{w}_j^\ell \right\rangle + b_j^\ell\right), \tag{5.3}$$

where n is the number of neurons in layer $L^{\ell-1}$ and $\mathbf{w}_j^\ell = (w_{1j}^\ell, w_{2j}^\ell, \ldots, w_{nj}^\ell)^T$.

We want to describe the whole output vector of a layer in terms of the output of the previous layer. A convention that we will follow here is to assume that every vector is given as row vector unless stated otherwise. The way we index the weights allows us to define the weight matrix and bias vector

$$W^\ell = \left(w_{ij}^\ell\right)_{\substack{i=1..n \\ j=1..m}} \qquad\qquad \mathbf{b}^\ell = (b_1^\ell, b_2^\ell, \ldots, b_m^\ell),$$

where n and m are the number of neurons in layers $L^{\ell-1}$ and L^ℓ, respectively. The j-th column vector in W^ℓ is just $\mathbf{w}_j^\ell$, as defined is Equation (5.3). Let $\boldsymbol{\sigma}$ be the function that applies activation σ element wise to its input. The output of layer L^ℓ can then be written shortly as

$$\mathbf{L}^\ell(\mathbf{o}^{\ell-1}) := \boldsymbol{\sigma}(\mathbf{o}^{\ell-1}W^\ell + \mathbf{b}^\ell) = \mathbf{o}^\ell. \tag{5.4}$$

To further thin out the notation, we introduced the layer function $\mathbf{L}^\ell$. Figure 5.5 summarizes the notation. With this, the output of a fully-connected feedforward neural

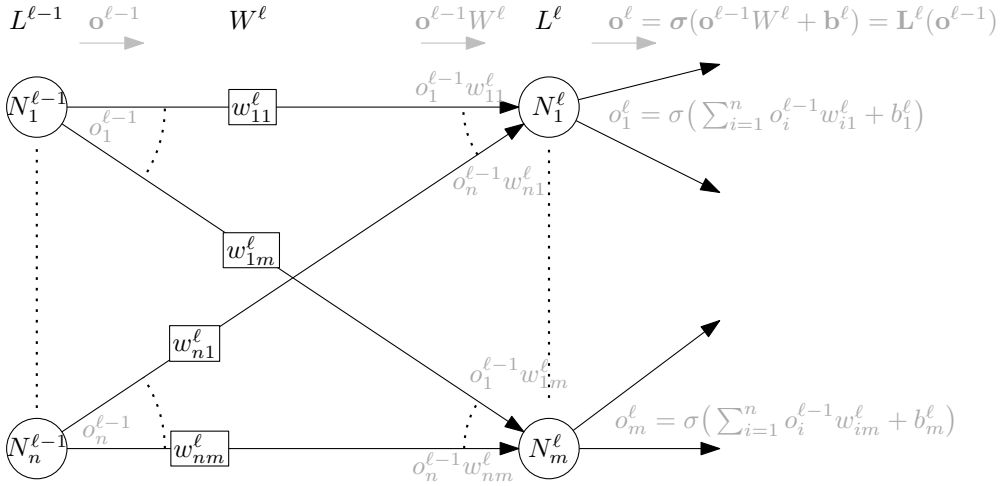


Figure 5.5: Illustration of the introduced notation.

network with layers $L_1, L_2, \ldots, L_d$ upon receiving input $\mathbf{x}$ is the composition of all layer functions

$$NN(\mathbf{x}) = \left(\mathbf{L}^d \circ \mathbf{L}^{d-1} \circ \cdots \circ \mathbf{L}^1\right)(\mathbf{x}). \tag{5.5}$$

5.1.2 Convolutional Neural Networks

Convolutional neural networks are a special type of neural nets that perform well when the input has the form of images. LeCun et al. [34] were among the first to successfully train a convolutional neural network for the task of handwritten digit recognition. The difference to fully-connected feedforward neural networks is the presence of at least one *convolutional layer* which, instead of the usual matrix multiplication as in Equation (5.4), performs a convolution operation between an input image and a kernel image.

We start by describing the discrete convolution operation of the 2-dimensional input image I and a kernel image K, also called filter mask. Both images are given as matrices; the kernel is restricted to have an odd number of pixels in both dimensions so that there

is a center pixel,

$$I = (i_{xy})_{\substack{x=1..n \\ y=1..m}}, \qquad K = (k_{xy})_{\substack{x=-n_k,..,0,..,n_k \\ y=-m_k,..,0,..,m_k}}.$$

Note that we indexed the kernel symmetrically around its center pixel at $(0, 0)$. The convolution of the two matrices creates a matrix where pixel p, q is given by[1]

$$(I * *K)_{pq} = \sum_{-n_k \le x \le n_k} \sum_{-m_k \le y \le m_k} i_{p+x,q+y}\, k_{xy}. \tag{5.6}$$

For pixels that are close to the boundary, for example if we want to compute $(I * *K)_{1,q}$ with $n_k \geq 1$, we need to extend the image region beyond its boundary. This can be done by padding it with zeros or other appropriate values (we could also pad with the mean value of the image or simply repeat the value of the closest pixel in the image region). Alternatively we could center the kernel exclusively at pixels that will not let it stick out of the image region, resulting in a smaller output image with dimension $(n - 2n_k) \times (m - 2m_k)$.
Figure 5.6 visualizes what happens in Equation (5.6). We get pixel (p, q) of $I * *K$ by centering K at pixel (p, q) of I and summing over the product of overlapping pairs of elements of K and I. This can be considered as an inner product of the flattened kernel and the flattened patch of I. We call $I * *K$ a *feature map*, as every pixel in it gives us a similarity information of the corresponding image patch to the kernel; we could consider the kernel as a feature that we search for in the image.

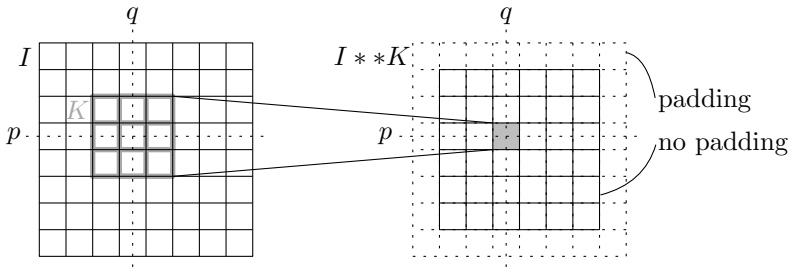


Figure 5.6: Visualization of the convolution operation. The kernel K is centered around pixel (p, q) of image I. The pairs of image and kernel pixels that lie in top of each other are multiplied and resulting products summed up. This gives the value of pixel (p, q) of $I * *K$. If we do not pad the image, the resulting image will be smaller and the corresponding pixel coordinates in $I * *K$ are $(p - n_k, q - m_k)$.

A convolutional layer applies this operation to an input image and optionally adds a bias term to the result. Similarly to the fully-connected layer the operation might be followed by application of a nonlinear activation function. The kernel can be learned, taken from another task or designed by hand. In case it should be learned it produces far fewer parameters than a fully-connected layer with the same number of outputs.

[1] We follow the convention of referring to the operation in Equation (5.6) as convolution. In the signal processing community this operation is known as correlation. The (discrete) convolution is given by

$$(I * *K)_{pq} = \sum_x \sum_y i_{p-x,q-y}\, k_{xy}.$$

While the fully-connected layer has parameters in the order of the product of input and output dimensions, the convolutional layer has a constant number of parameters for the kernel plus possibly a linear number (in the number of output pixels) of parameters for the biases. The left image in Figure 5.7 shows a visualization of a simple convolutional layer. In practice, however, usually both the input and output are in the form of tensors with depth > 1. The is the result of taking more than one kernel per layer. Each kernel is then assumed to cover the whole depth of the input tensor and creates one slice of the output tensor, as illustrated in the right image of Figure 5.7. To account for this, we could simply interpret the input tensors and kernels as 2D matrices of vectors and replace $i_{p+x,q+y}\, k_{xy}$ in Equation (5.6) by $\langle \mathbf{i}_{p+x,q+y}, \mathbf{k}_{xy} \rangle$.

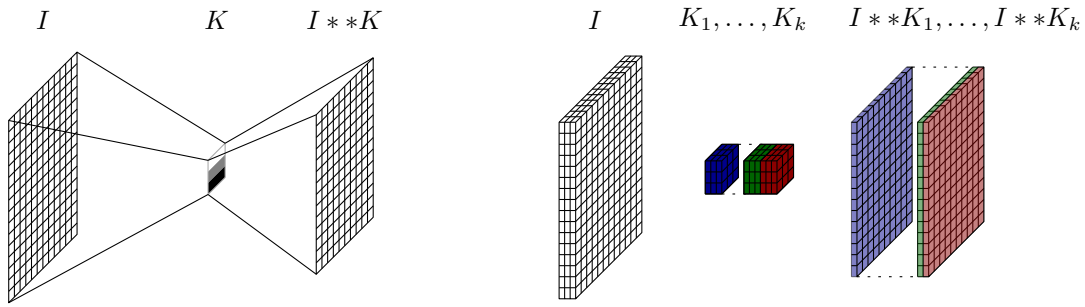


Figure 5.7: Left: Visualization of a convolutional layer. Right: In practical applications the input image is often a tensor with depth > 1, and more than one kernel is applied. When performing a 2D convolution, each kernel is assumed to cover the full depth of the input tensor and produces one slice of the output tensor.

A convolutional layer can also be modeled by a fully-connected layer as is shown in the left image of Figure 5.8. We arranged the convolution implementing fully-connected layer L^i in a rectangular shape and did the same for its input and output vectors $\mathbf{o}^{i-1}$, $\mathbf{o}^i$. This is only for a cleaner illustration; in the actual network all mentioned entities would be flattened out. Each neuron creates exactly one output, hence, we need as many neurons as there are pixels in the feature map, which is roughly the size of the input. Every neuron will focus on one pixel p in the input vector and use its weights to imitate the desired kernel around p. All weights of edges that are outside this kernel region around p will be set to zero. Therefore every neuron has the same weights, only the particular edges to which these weights are assigned differ to account for the different pixels that are focused on. The right image in Figure 5.8 sketches how the corresponding matrix W^i, that operates on the row-wise flattened image, would look like. The horizontal lines indicate where the next row in the input image starts.

Residual Networks

Residual networks were first introduced by He et al. [24] and gained attention by winning the 2015 ImageNet classification challenge (among other challenges). Their work was motivated by the phenomenon that at some point adding more layers to a neural network does not further improve the performance. Instead, they witnessed a degradation of validation and even training performance. Unintuitive about this effect is that, in theory, a deeper net should be able to simulate a shallower one by simply letting the excess layers perform identity maps, essentially doing nothing to the input. They conjectured that, by making it easier for the network to learn identity maps, it should be able to produce

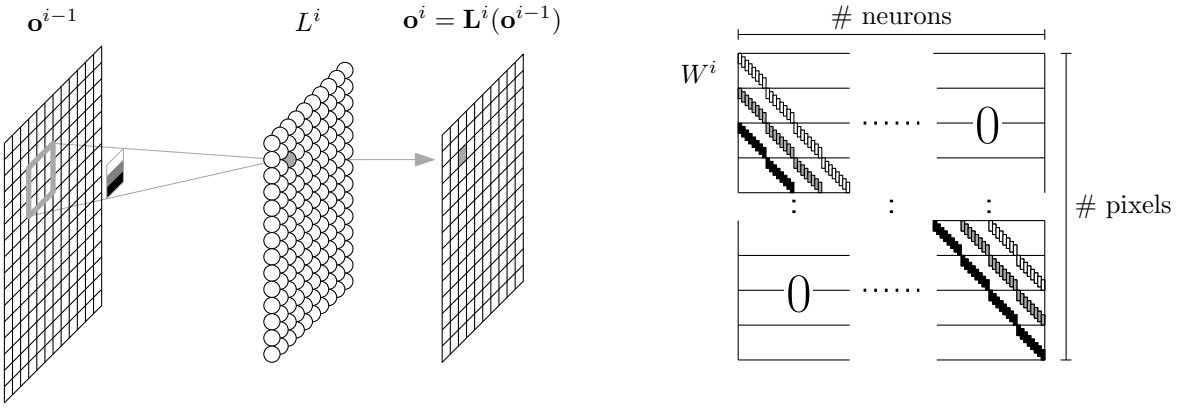


Figure 5.8: Left: Illustration of a fully-connected layer modeling a convolution layer. Right: The corresponding matrix.

results at least on par with those of shallower ones.

To accomplish this, they came up with the *residual block*. A network component, whose original design is illustrated in Figure 5.9 (they also introduced an alternative design for deeper networks in Ref. [24] and a refinement in Ref. [25]). The idea is to

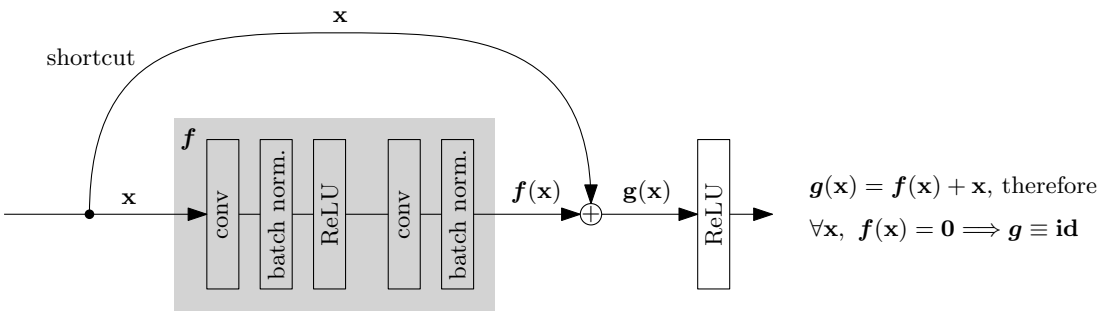


Figure 5.9: Original design of the residual block. The shortcut connection allows the block to perform an identity map by pushing all weights inside the gray area (realizing the function $\boldsymbol{f}$) to zero.

introduce a *shortcut connection*. We group a set of consecutive convolutional layers (including possible batch normalization and activation layers) to a unit and express the function they perform on an input $\mathbf{x}$ fed to the unit as $\boldsymbol{f}$. A shortcut connection simply adds $\mathbf{x}$ element wise to the output $\boldsymbol{f}(\mathbf{x})$. We denote the function this new unit performs as $\boldsymbol{g}$. Since $\boldsymbol{g}(\mathbf{x}) = \boldsymbol{f}(\mathbf{x}) + \mathbf{x}$, an identity map can easily be accomplished by pushing all weights in $\boldsymbol{f}$ to zero, thus, letting $\boldsymbol{f}$ approach the constant zero function.

The element wise addition operation requires $\boldsymbol{g}(\mathbf{x})$ to have the same dimensionality as $\mathbf{x}$. After a fixed number of residual block repetitions, the tensors will be downsampled to half the size along width and height and twice the depth. This is accomplished by doubling the number of kernels for the convolutional layers inside the residual block and performing strided convolution with step size two in one the convolutional layers. There are several proposals how the tensor $\mathbf{x}$ that is send along the shortcut connection could be modified to match the size. In the simplest case, width and height are downsampled by 2×2 average pooling (partition an images into 2×2 fields, and for each field keep only the average of its four contained values) and the missing depth is filled up with zero padding.

It turned out that a network build from residual blocks not only matched but surpassed the performance of shallower ones. The winning architecture had more than hundred layers.

5.1.3 Training Neural Networks

This description of the training process is loosely based on the book by Rojas[47]. For a beautiful, less technical explanation we refer to the online book by Nielsen [39].

To start explaining the training process, we first need to introduce a differentiable loss function $\mathcal{L}(NN, T)$ that measures the performance of a neural network $NN(\mathbf{x}; \mathbf{\Omega})$ on a training sequence $T = (\mathbf{x}_i, y_i)_{i=1..n}$. Here, the parameter vector $\mathbf{\Omega}$ comprises all parameters that adjust the behaviour of NN. In case of the fully-connected feedforward neural network, this includes the weights and biases of all layers. The loss function is designed to decrease with improving performance of NN on T. Further it is necessary to restrict loss functions to those that can be written as averages over per-sample losses $\ell(NN(\mathbf{x}), y)$:

$$\mathcal{L}(NN, T) = \frac{1}{n} \sum_{i=1}^{n} \ell(NN(\mathbf{x}_i), y_i). \tag{5.7}$$

In a regression task, a popular choice for the loss function is the is the quadratic error $\ell(NN(\mathbf{x}), y) = (NN(\mathbf{x}) - y)^2$ leading to the mean squared error loss

$$\mathcal{L}(NN, T) = \frac{1}{n} \sum_{i=1}^{n} (NN(\mathbf{x}_i) - y_i)^2.$$

For classification tasks the neural network is often designed to return a vector that can be interpreted as a (discrete) conditional probability distribution, that is $\boldsymbol{NN}(\mathbf{x})$ is a vector valued function and for the y-th component we have $NN(\mathbf{x})_y = \Pr(y \mid \mathbf{x})$. In this case, a popular choice for the loss function is the *information content* $\ell(NN(\mathbf{x}), y)) = -\log NN(\mathbf{x})_y$ leading to the *cross-entropy* loss

$$\mathcal{L}(NN, T) = -\frac{1}{n} \sum_{i=1}^{n} \log NN(\mathbf{x}_i)_{y_i}.$$

The particular choice of the loss function needs to be coordinated with the task that is to be performed by the network.

Once we have decided on a loss function $\mathcal{L}$, we can train the neural network by *gradient descent*. This technique strives to find a sequence of parameter vectors $\mathbf{\Omega}_0, \mathbf{\Omega}_1, \ldots$ that successively improves the performance of the neural network. The most basic idea is to start with a random parameter vector and improve by moving a step into the negative direction of the gradient of the loss function with respect to the current parameter vector:

$$\mathbf{\Omega}_{t+1} = \mathbf{\Omega}_t + \gamma \Delta \mathbf{\Omega}, \quad \text{with } \Delta \mathbf{\Omega} = -\nabla_{\mathbf{\Omega}} \mathcal{L}. \tag{5.8}$$

Note that $\mathcal{L}$ depends on $\mathbf{\Omega}$ through NN. The update rule in Equation (5.8) works since the vector $-\nabla_{\mathbf{\Omega}} \mathcal{L}$ points into the direction of the highest rate of decrease of $\mathcal{L}$. The new parameter γ is called the *learning rate*. In Equation (5.8) the magnitude by which $\mathbf{\Omega}_t$ and $\mathbf{\Omega}_{t+1}$ differ also depends on $|\nabla_{\mathbf{\Omega}} \mathcal{L}|$. If this is not desired, the gradient can be normalized before its application in the update rule. In this case, γ would be the total step width

of the descent step. There are several refinements on how to update the gradient. The most popular ones use a *momentum* term that is added to the gradient and reflects a decaying history of previous gradients. Gradient descent can then be summarized by iterating the two steps

1. estimate the gradient $\nabla_{\mathbf{\Omega}}\mathcal{L}$,
2. according to the gradient, update the parameter vector $\mathbf{\Omega} \to \mathbf{\Omega} + \Delta\mathbf{\Omega}$.

Computing the gradient with respect to parameters $\mathbf{\Omega}$ is done with an *automatic differentiation* technique called *backpropagation*. It has a long history, and Ref. [21] attributes its origin to work in the field of operations research in the early 1960s, particularly by Kelly [30] and Bryson [12]. The idea of applying it to train neural networks is accredited to Werbos [60] in 1974 and was rediscovered by Rumelhart, Hinton, and Williams in 1985 [49].

This technique exploits that the function a neural network computes can be expressed as a composition of layer functions $NN(\mathbf{x}) = (\mathbf{L}^d \circ \mathbf{L}^{d-1} \circ \cdots \circ \mathbf{L}^1)(\mathbf{x})$ and, hence, allows the application of the chain rule for multivariable (vector-valued) functions.

In the following description, we compute the gradien with respect to the loss caused by a single input. According to Equation (5.7), for several inputs we will need to sum up the element-wise gradients. In practical settings, usually only a few elements of the inputs are used to compute an *approximation* of the true gradient. This is called *batch gradient descent* and the batch size is another meta-parameter to tune.

We want to find the partial derivatives with respect to the parameters in $\mathbf{\Omega}$. Exemplarily we will derive this for a fully-connected feedforward neural network where we are interested in finding $\partial\mathcal{L}/\partial w_{ij}^{\ell}$ and $\partial\mathcal{L}/\partial\mathbf{b}^{\ell}$ for all ℓ, i, and j. We will focus on deriving the partial derivatives with respect to the weights; the partial derivatives with respect to the bias term can be obtained analogously. As introduced in Equation 5.4, we set $\mathbf{o}^{\ell} = \mathbf{L}^{\ell}(\mathbf{o}^{\ell-1})$, for $\ell = 1, \ldots, d$ and $\mathbf{o}^0$ equal to the input $\mathbf{x}$ fed to the network. The layer function was given by Equation (5.4). A weight w_{ij}^{ℓ} is only contributing in the computation of $\mathbf{o}^{\ell} = \mathbf{L}^{\ell}(\mathbf{o}^{\ell-1})$, and even there, it only takes part in the computation of the j-th component o_j^{ℓ} (see Fig. 5.10).

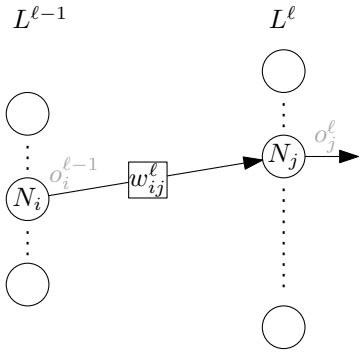


Figure 5.10: The weight w_{ij}^{ℓ} only contributes to the copmutation of the j-th component of $\mathbf{o}^{\ell}$.

Applying the chain rule for computing the derivatives gives

$$\frac{\partial\mathcal{L}}{\partial w_{ij}^{\ell}} = \frac{\partial\mathcal{L}}{\partial o_j^{\ell}} \frac{\partial o_j^{\ell}}{\partial w_{ij}^{\ell}}.$$

First we look at the second term of the right hand side. From Equation (5.4) we see that

$$o_j^\ell = \sigma\Big(\langle \mathbf{o}^{\ell-1}, \mathbf{w}_j^\ell \rangle + b_j^\ell\Big), \tag{5.9}$$

where we have written $\mathbf{w}_j^\ell$ for the j-th column vector of W^ℓ. Therefore, if we denote the derivative of σ with σ', we have

$$\frac{\partial o_j^\ell}{\partial w_{ij}^\ell} = \sigma'\Big(\langle \mathbf{o}^{\ell-1}, \mathbf{w}_j^\ell \rangle + b_j^\ell\Big)\, o_j^{\ell-1},$$

where, again, we made use of the chain rule.

Next, we look at the first component. The partial derivative $\partial\mathcal{L}/\partial o_j^\ell$ is just the j-th component of the gradient $\nabla_{\mathbf{o}^\ell}\mathcal{L}$. It follows that we can compute the gradient with respect to all weights if we can show how to compute the gradients with respect to all $\mathbf{o}^\ell$. This will be done by a dynamic programming approach. Usually, when we apply dynamic programming, the sub-problems serve the only purpose of finding a solution to the main problem. Here, the solutions to the sub-problems will be of interest on their own. We want to find the gradients $\nabla_{\mathbf{o}^\ell}\mathcal{L}$ for $\ell = 1, 2, \ldots, d$.

The sub-problem for finding $\nabla_{\mathbf{o}^{\ell-1}}\mathcal{L}$ is to find $\nabla_{\mathbf{o}^\ell}\mathcal{L}$ (that is, we work from d down to 1). The starting point $\nabla_{\mathbf{o}^d}\mathcal{L}$ needs to be computed explicitely. If we have $\nabla_{\mathbf{o}^\ell}\mathcal{L}$, we can compute $\nabla_{\mathbf{o}^{\ell-1}}\mathcal{L}$ by application of the chain rule

$$\nabla_{\mathbf{o}^{\ell-1}}\mathcal{L} = \nabla_{\mathbf{o}^\ell}\mathcal{L} \cdot J_{\mathbf{L}^\ell}, \quad \text{where } J_{\mathbf{L}^\ell} := \left(\frac{\partial \mathbf{L}^\ell}{\partial o_i^{\ell-1}}\right)_{ij} \text{ is the Jacobian matrix of } \mathbf{L}^\ell.$$

Note that we can write the entry $\partial\mathbf{L}^\ell/\partial o_i^{\ell-1}$ of the Jacobian as $\partial o_j^\ell/\partial o_i^{\ell-1}$. We already computed an expression similar to this in Equation (5.9). The only difference is that there we took the derivative with respect to weight w_{ij}^ℓ. But since w_{ij}^ℓ and $o_i^{\ell-1}$ only appear in form of the product $w_{ij}^\ell o_i^{\ell-1}$, that is, have symmetric roles, we can simply exchange them. Thus,

$$\frac{\partial \mathbf{L}^\ell}{\partial o_i^{\ell-1}} = \sigma'\Big(\langle \mathbf{o}^{\ell-1}, \mathbf{w}_j^\ell \rangle + b_j^\ell\Big)\, w_{ij}^\ell.$$

5.1.4 Software

The neural networks used were implemented in tflearn [1], a high-level API ontop on the deep learning framework Tensorflow [3].

5.2 Parameter Exploration and Insights

From the selection of classifiers in this theses, neural network are the most intricate to work with. Firstly, they have the largest set of parameters to deal with. Even with a fixed network architecture, we have to decide on the optimizer to use, its learning rate (and possibly other optimizer specific parameters), the batch size for stochastic gradient descent, and what kind of regularization to apply as well as its amount. These factors are joined by the already introduced questions on how the data imbalance should be handled, what input format to use, its size (originally sized images or resampled to half

the size along each dimension), and what preprocessing to apply. Secondly, they require the longest time to train, in our case, from seconds to hours depending the specific parameter configuration.

Due to time limitations, it is not possible to explore the parameter space as systematic and thorough as in previous sections. Instead, we focus on the effect of tweaking a subset of parameters and simply fix the remaining ones. In many situations, when several possible continuations arise, we decide in favor of the simplest, most efficient, or (subjectively) most interesting one.

Previous classifiers showed significant performance variation when trained multiple times with identical configurations. Neural networks do not appear to be an exception. Unfortunately, performing a multitude of runs with identical configurations gravely restricts the width of possible parameter configurations to explore. Performing only a single run per configuration without any idea of the underlying variation, on the other hand, hardly allow for any comparison between configurations at all. As a compromise, we perform five runs for every configuration that we wish to explore. For visualization, we introduce the *median line*. For n sequences $C^1, \ldots, C^n$ with $C^i = (c^i_1, \ldots, c^i_m)$, we denote with the median line the sequence

$$\overline{C} = (\overline{c}_1, \ldots, \overline{c}_m), \quad \text{where} \quad \overline{c}_j = \text{median}(\, \{c^i_j \mid i \in [n]\}\,).$$

The median line is unlikely to be identical to any C^i_j and can hide a lot of the variation of the underlying sequences. Therefore we will accompany it with the *maximal distance line* D. For the same setting as above, we set

$$\begin{aligned} &D = (d_1, \ldots, d_m), \quad \text{where} \\ &d_j = \max\big(\, \max(\, \{c^i_j \mid i \in [n]\}\,) - \overline{c}_j, \overline{c}_j - \min(\, \{c^i_j \mid i \in [n]\}\,)\,\big). \end{aligned}$$

5.2.1 Network architecture and training set-up

The following experiments were performed with a tflearn implementation [2] of a *residual network*. The tflearn implementation of a residual block (see Fig. 5.11) follows the refinement proposal in Ref. [25]. All convolutions are performed with 3×3 kernels, and,

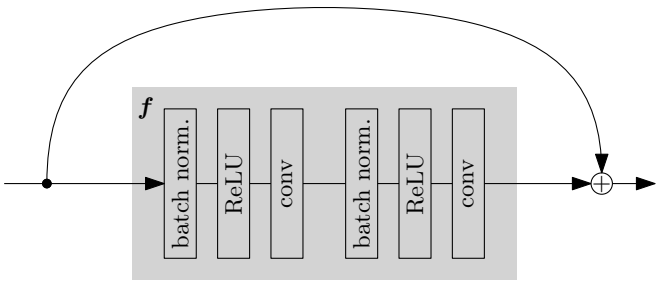


Figure 5.11: Residual block implementation in tflearn.

if downsampling is required, it is performed in the first convolutional layer. Figure 5.12 shows the whole network architecture. It starts with a single convolutional layer producing an output tensor with depth 16. It is followed by 3 groups of residual blocks, each

comprising 5 residual blocks. The initial residual block of the second and third group downsample the input. After a final batch normalization and the ReLU activation layer, global average pooling is performed. The resulting 64 values are fed to a fully connected layer with two output nodes. Finally, a softmax activation molds the output of the two nodes to a two-categorical distribution.

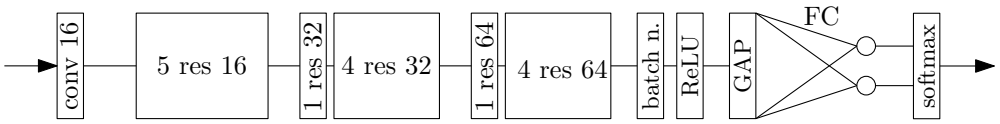


Figure 5.12: The network architecture used for the following experiments. All convolutional layers apply kernels with width and height 3; conv n stands for a convolutional layer with output depth n; m res n stands for m residual blocks, with output depth n. A residual block with more output- than input-depth performs downsampling along width and height. In total, this architecture has 31 convolutional and one fully connected layer (FC). A global average pooling layer is arranged prior to the fully connected layer.

We chose the cross-entropy loss function and used the Adam optimizer [31] for gradient descent. We The two moving window parameters were left at their default $\beta_1 = 0.9$, $\beta_2 = 0.999$. By manual searching, the learning rate was adjusted to $5 \cdot 10^{-5}$, a value that allowed for learning and showed the dynamics of the training curve. We fixed the batch size initially to 10.

The relatively small number of training samples suggests that regularization will play a fundamental role in increasing the network performance. Following Ref. [24], we apply weight decay to every convolutional layer.

For finding a well performing set of parameters, Bergstra and Bengio recommend [5] to search randomly. Beyond the mere number of possible parameter configurations, there are other obvious drawbacks to grid search. For example, it is likely that, out of the many possible parameters P, there will be a subset $U \subseteq P$ of non-informative parameters that only contribute negligibly to the observed performance. Changing these parameters will not affect the outcome in any detectable manner. If, for every $p \in P$, n_p is the number of values p can take, then the *factor* of non-informative runs is $\prod_{p \in U} n_p$. Even if there is just a single non-informative parameter among all tested ones, and it is only tested for two different values, half of the runs will be non-informative.

If, instead, we perform every run with a random parameter setting, chances are that at least one informative parameter changes.

But there is also a major drawback to random search. If not only the best performance of the resulting classifier, but insight into the effect (and its extent) of different parameters is desired, random search complicates the analysis significantly.

Therefore, we nevertheless decided to apply grid search, while considering only selected parameter subspaces.

5.2.2 Addressing the class imbalance

Preliminary experiments with a smaller convolutional architecture showed the same biased behavior when trained on imbalanced classes. We tried to adjust the cost function so that an error of a sample contributed proportional to the inverse of its class frequency, but results were inferior to under- and over-sampling. The best results were achieved on oversampled datasets, which is the applied balancing strategy for the remaining part.

5.2.3 Input format, size, weight decay, and preprocessing

In this first set of experiments, we explore the interplay of different input formats, image sizes, and the degree of regularization. To reduce the number of possibilities and computational cost, we stick to single-image input formats `2d_slice2_sample_1` and `2d_slice2_sample_64`. We check the performance on originally sized ($41\times$ 81 pixels) and resampled (21×41 pixels) images. As before, we consider the three preprocessing techniques no preprocessing, per-feature standardization, and per-image standardization. We test the weight decay parameter in an exponential pattern and let it take values in $\{0.0001, 0.001, 0.01, 0.1\}$.

Training and validation is performed on oversampled subsets of dataset ds1. First a validation set of 100 samples per class is separated. The remaining elements are oversampled, resulting in 1210 training samples per class.

We perform five training sessions for each of the 48 configurations. In order to read anything out of the resulting curves, we have to order them appropriately. We conjecture that the two image formats will behave similarly and gather them in the same plot. The five training and validation curves will be represented by their median lines. The resulting arrangement is shown in Figure 5.13.

As described, the chosen learning rate keeps the network from immediately overfitting the training data. We can draw the following conclusions.

- Best performances are surprisingly achieved when no preprocessing is applied. Per-feature and per-image standardization stay behind in top validation accuracy by roughly 10%.
- The two image formats indeed behave similarly.
- Increasing weight decay noticeably slowed down the training process. This is illustrated by the decreasing training curve slopes and the delay in overfitting. Although this is not accompanied by a significant validation increase, it appears that, in the case of no preprocessing, the range of iterations during which the best validation results are achieved is prolonged.
- For most of the validation curves, systematic validation performance increase during learning can only be observed for the highest weight decay settings. It is likely that for smaller settings all relevant learning already happened during the first epoch.
- Smaller image formats mimic the learning dynamics of their larger counterparts on a shorter time frame, in case of no preprocessing without decrease in top performance.

Based on these insights, we decide to continue with small images without preprocessing and of format `2d_slice1_sample_1`. We check whether an even larger weight decay setting further prolongs the top performance phase for small images and got the most promising curves at a weight decay parameter of 4 (see below). This large value results from the few available training images compared with the large hypothesis space of the neural network. For comparison, in Ref. [24] a weight decay parameter of 0.0001 is chosen. Later, when we apply methods to increase the number of input samples, we will readjust this value. We also need to consider whether the strong results without preprocessing can also be achieved on new tomograms, since in the previous experiments both training and validation sets came from the same tomogram.

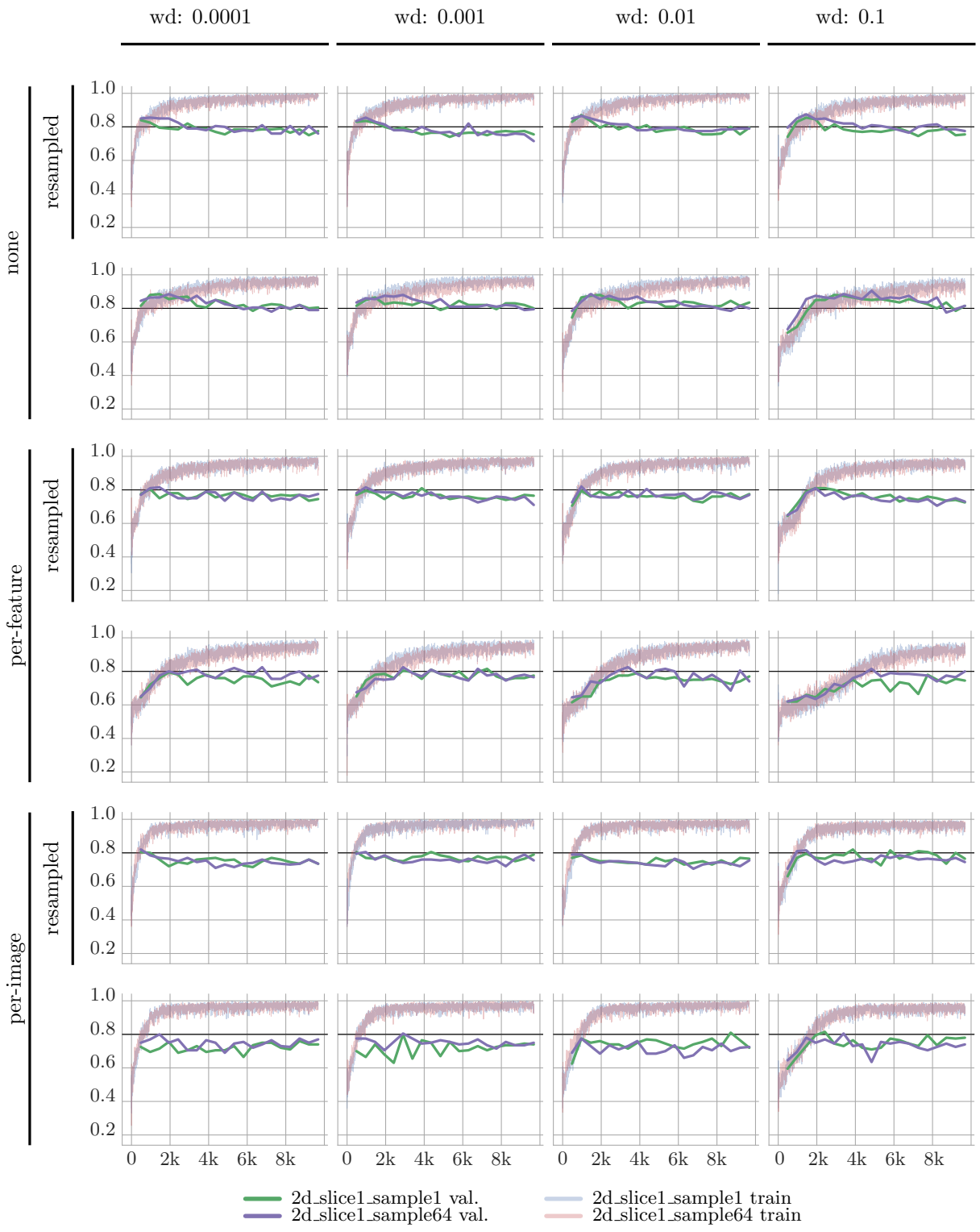


Figure 5.13: Median lines for validation and training accuracy from five training sessions, for three preprocessing techniques (none, per-feature standardization, per-image-standardization), two image formats (cross-sections, cylindrical average), two image sizes, and four weight decay values. Training and validation was performed on data from dataset ds1.

We refine the setup and train on all non-contradictory labeled samples from dataset ds1, balanced by oversampling, while validating on all non-contradictory labeled samples from dataset ds2, and ds3. Figure 5.14 shows the median lines from five runs of this setting with a weight decay parameter of 0.1 and 0.4. The left plot can be compared

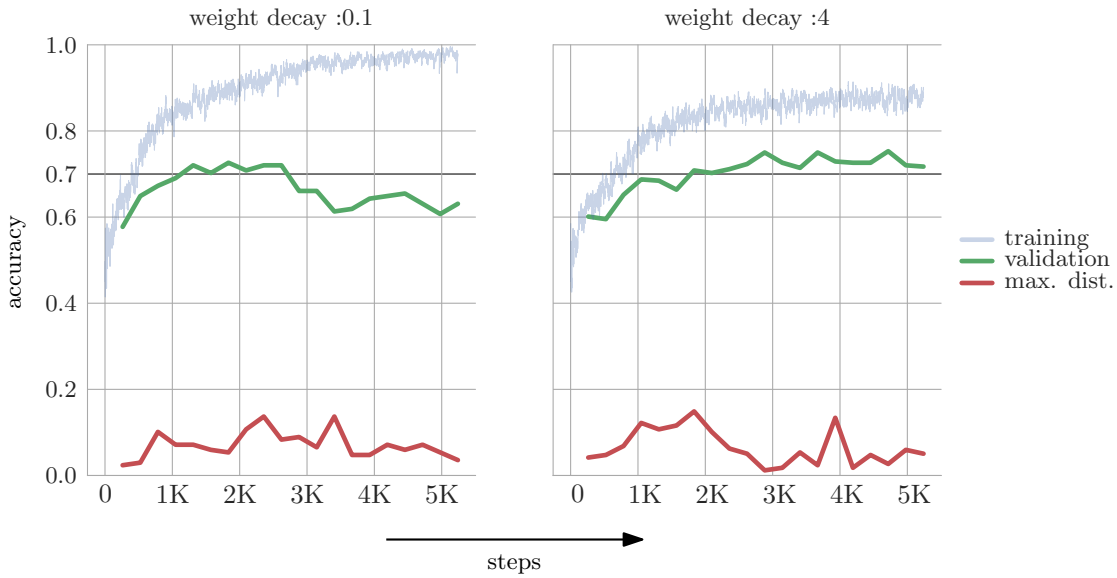


Figure 5.14: Median lines for validation and training curves and maximal distance lines for the validation performances from five training sessions. Training was performed on data from dataset ds1, validation on data from datasets ds2 and ds3.

to the top- and right-most plot in Figure 5.13. The only difference is in the training and validation data. While validation on the same tomogram achieved accuracies past 85%, the top performance on an unseen tomogram drops below 75%. The right plot shows that the large weight decay setting of 4 again prolongs the range in which the best performances are achieved and successfully reduces overfitting.

Next, we explore the effect of introducing more training data. We replace the previous training data by the separated single images from input formats `2d_slice4_sample_1` and `2d_slice4_sample_64`. This increases the number of input samples by a factor of 8. The validation set remains the same. Since validation was applied after each epoch, which would now take eight times as long and introduces the risk of overfitting during the first epoch, we increased the batch size also by a factor of eight to 80. We also introduced on-line data augmentation in which the training samples were randomly rotated by $\pm 30°$, flipped horizontally, translated by up to 5 pixels, and blurred with a Gaussian ($\sigma = 2$). To check whether this new adjustments would profit from a new learning rate, we also scanned it in the range $\{10^{-5}, 2 \cdot 10^{-5}, \ldots, 5 \cdot 10^{-5}\}$. Figure 5.15 shows the resulting curves for the new setting. The smoother training curves are an effect from the larger batch size. The bottommost plot on the left corresponds to the settings in the right plot of Figure 5.14. We see that the enlarged training set does not show significant validation improvements, neither does the additional data augmentation.

It appears as if the validation curves for the smaller learning rates have not reached their maximum, but experience and extrapolation from the larger learning rates showed that the ultimate upper bound is unlikely to be raised. More interestingly, we witness the large impact data augmentation has on overfitting. Distances between training an

validation curves frequently stay below 10% during the whole training phase. Without data augmentation these distances are over 20%, sometimes approaching 40%. This motivates readjusting of the weight decay parameter in hope that the recovering training performance is able to also increase validation performance. We re-scanned the weight decay parameter in the range $\{10^{-4}, 5 \cdot 10^{-3}, 10^{-3}, 5 \cdot 10^{-2}, \ldots, 5\}$ with the learning rate set to $4 \cdot 10^{-5}$.

We detected a minor performance increase for weight decay values as small as 0.1. Below that, the curves reproduced the behavior we saw before; for decreasing weight decay parameters, the phases during which the validation performance peaked became shorter. We never tested whether going without any preprocessing diminishes generalization performance on new tomograms. Ref. [24] applied per-feature centralization, which can be obtained from per-feature standardization by omitting the transformation to unit variance. We tested both per-feature standardization and per-feature centralization. The top performances decreased also for validation data from unseen tomograms. Exemplarily, we show plots for weight decay settings of 0.1 and 1 for no preprocessing and per-feature centralization in Figure 5.16.

5.3 Final Training and Results

The final training was performed on all non-contradictory labelled endpoints from datasets ds1, ds2, and ds3. We took all image orientations from input formats `2d_slice4_sample_1` and `2d_slice4_sample_64`. The minority class was oversampled which led to a total of 13200 images per class. We also applied on-line augmentation as described above. The batch size was kept at 80, the learning rate initially at $4 \cdot 10^{-5}$. We observed the training process in TensoBoard and trained without validation set until the training curve displayed roughly 80% accuracy. Afterwards the learning rate was reduced by a power of 10 and the training continued until a training performance of roughly 85% was displayed. When applied to the training set afterwards, the actual per-class accuracies were

0.8 (*open*)
0.9 (*closed*).

On the test set comprising all non-contradictory labeled samples from datasets qs1, . . . ,qs4 (1025 *open* and 336 *closed* samples) the network achieved per-class accuracies of

0.61 (*open*)
0.78 (*closed*).

The network was designed to output class-probabilities. The classes are obtained by applying a threshold to these probabilities. We can consider only labels that were obtained from a network probability ≥ 0.9. This can be interpreted as the network being *sure* of the assigned class. On 512 of the 1361 test samples (375 out of 1025 *open*, and 127 out of 336 *closed*) the network gave predictions above 0.9. For this subset, the obtained per-class accuracies are

0.83 (*open*)
0.8 (*closed*).

Figure 5.18 shows the first 10 samples where the network gave probabilities above 0.9

for the wrong label.

After testing the final network, we evaluated balanced subsets of datasets qs1, . . . ,qs4 on all intermediate checkpoints to see how the network decisions evolved through the learning phase. The left image in Figure 5.18 shows class probabilities output next to the true labels. The right images show the resulting classification decision after applying a threshold at 0.5. We see that most of the final labels are already found after the first half of the training process. Moreover the first test dataset appears to pose the biggest challenge. Notice in particular the thick black area at the top left of the images indicating that during this phase most of the samples in the dataset would be classified as closed.

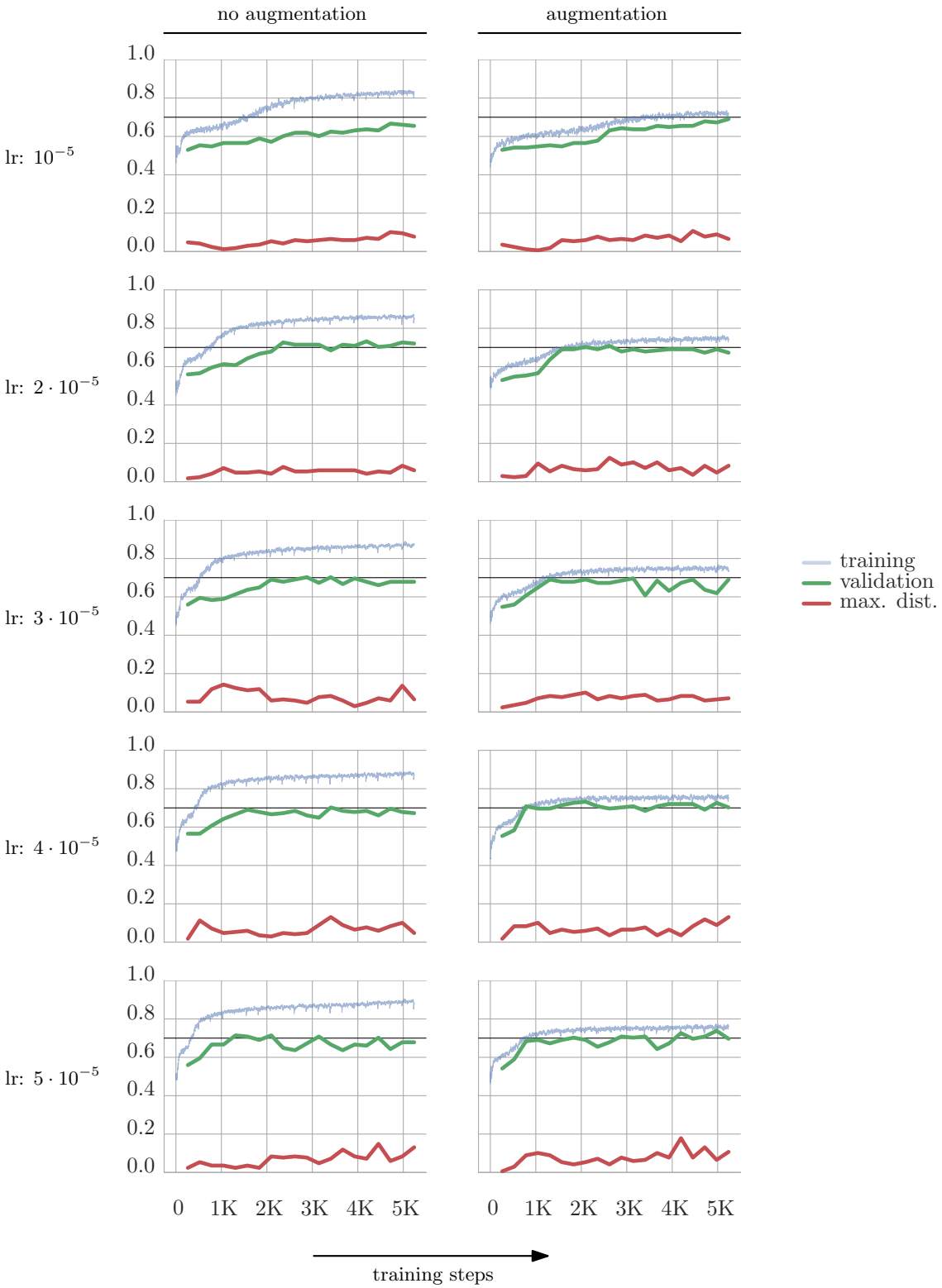


Figure 5.15: Median lines for validation and training curves and maximal distance lines for the validation performances from five training sessions. Training was performed on single images taken from input formats `2d_slice4_sample_1` and `2d_slice4_sample_64` of dataset ds1, validation on data from datasets ds2 and ds3.

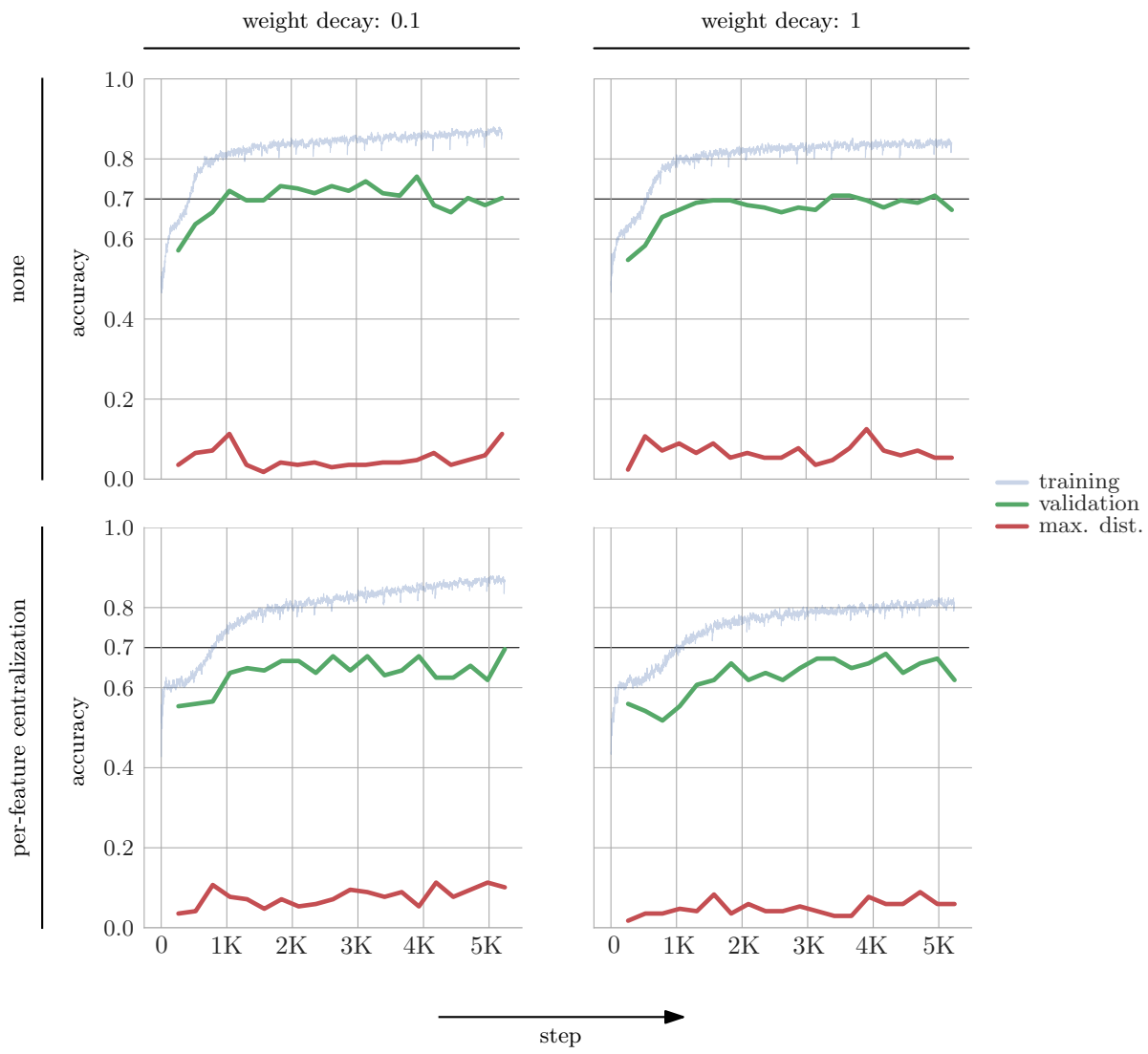


Figure 5.16: Median lines for validation and training curves and maximal distance lines for the validation performances from five training sessions. Training was performed on single images (on-line augmented) taken from input formats `2d_slice4_sample_1` and `2d_slice4_sample_64` of dataset ds1, validation data was taken from datasets ds2 and ds3. A slight performance degradation occurred when training and validation was performed on per-feature centralized images.

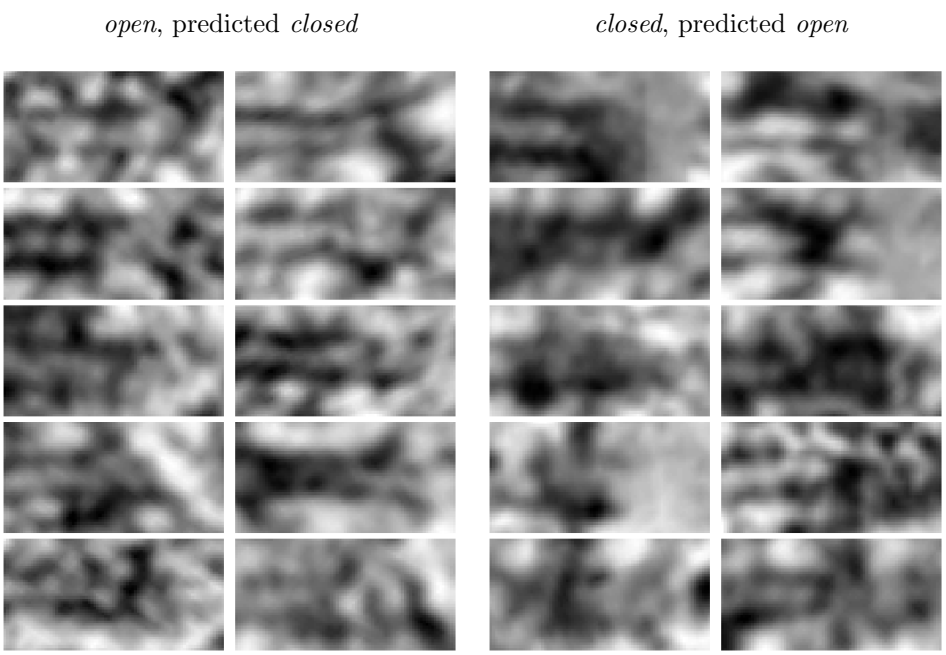


Figure 5.17: Samples that had network probabilities ≥ 0.9 but were predicted falsely.

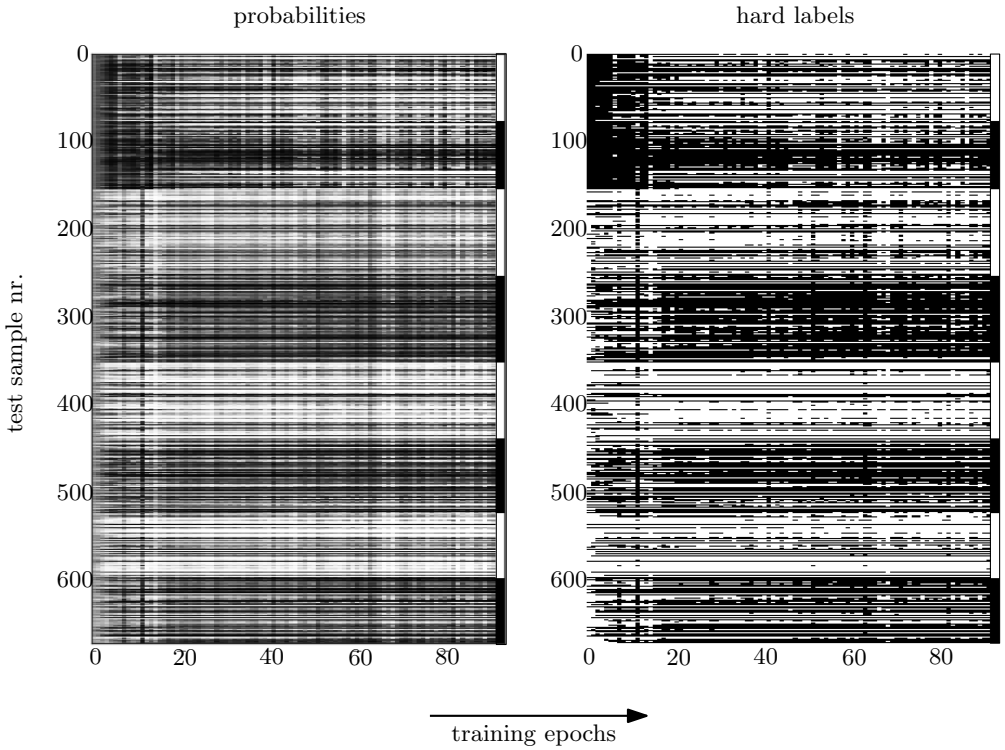


Figure 5.18: Evolution of classification decisions throughout the training phase on balanced subsets of datasets qs1, . . . ,qs4. The bars to the left indicate the true label, where white stands for open and black for closed. The four pairs of consecutive open and closed elements correspond to the four datasets. Left: Network probabilities. Right: Hard labels after applying a threshold at probability 0.5.

6 Comparison of the Results for the Different Classifiers

We introduce the abbreviations DT, SVM, RN to refer to the final decision tree, support vector machine, and neural network (**r**esidual **n**etwork) classifiers, respectively. Table 6.1 lists the confusion matrices for each classifier on the same balanced test set T_1. As a reminder, T_1 is obtained by taking all samples from datasets qs1, ..., qs4 for which at least one expert voted *open* or *closed* and no two experts voted contradictory. In total, T_1 contains 1025 *open* and 336 *closed* samples.

	DT		SVM		RN	
ground truth	*open*	*closed*	*open*	*closed*	*open*	*closed*
open	638 (62%)	387 (38%)	673 (66%)	352 (34%)	630 (61%)	395 (39%)
closed	94 (28%)	242 (72%)	101 (30%)	235 (70%)	74 (22%)	262 (78%)

Table 6.1: Confusion matrices for the final classifiers on test set T_1. Percentages are obtained by normalizing on the number of samples in each class of the ground truth ($\#open = 1025$, $\#closed = 336$) in the test set.

Compared with RN, classifiers DT and SVM achieve more balanced per class accuracies. But no classifier breaks the 70% accuracy on both classes.

For each classifier $c \in C = \{\mathrm{DT}, \mathrm{SVM}, \mathrm{RN}\}$ and set of labeled elements S, we introduce the *failure set* F_c^S of c on S as the set of all samples in S misclassified by c. We will omit the high index S when the set under consideration is given by context.

We want to study similarities between the failure sets on T_1. For all non-trivial subsets $S \subseteq C$, Table 6.2 lists the number of elements in the failure set intersection, $\bigcap_{c \in S} F_c$.

S	F_{DT}	F_{SVM}	F_{RN}	$F_{\mathrm{DT}} \cap F_{\mathrm{SVM}}$	$F_{\mathrm{DT}} \cap F_{\mathrm{RN}}$	$F_{\mathrm{SVM}} \cap F_{\mathrm{RN}}$	$F_{\mathrm{DT}} \cap F_{\mathrm{SVM}} \cap F_{\mathrm{RN}}$
$\lvert S \rvert$	481	453	469	400	265	257	238

Table 6.2: Sizes for intersections of failure sets on T_1.

The Venn diagram in Figure 6.1 gives finer detail on how many elements are in- and excluded in which failure sets. The numbers were derived from Table 6.2. Note that the areas in the Venn diagram do not reflect the actual size ratios. In a more representative illustration, the circles representing F_{DT} and F_{SVM} would almost completely overlap and F_{RN} could be represented by an hour glass shape with half its area enclosed by the intersection of the other two and the other half outside of their union.

There is a large number of elements that no classifier gets right. The similarity of F_{DT} and F_{SVM} can be explained by the similar input representations they were trained on and the fact that *DT* achieved the main separation in the first split; that is, DT is not far from being a linear separator like SVM. More surprising is the ratio of samples that also lie in F_{RN} considering the different input- and hypothesis-spaces of RN and, say,

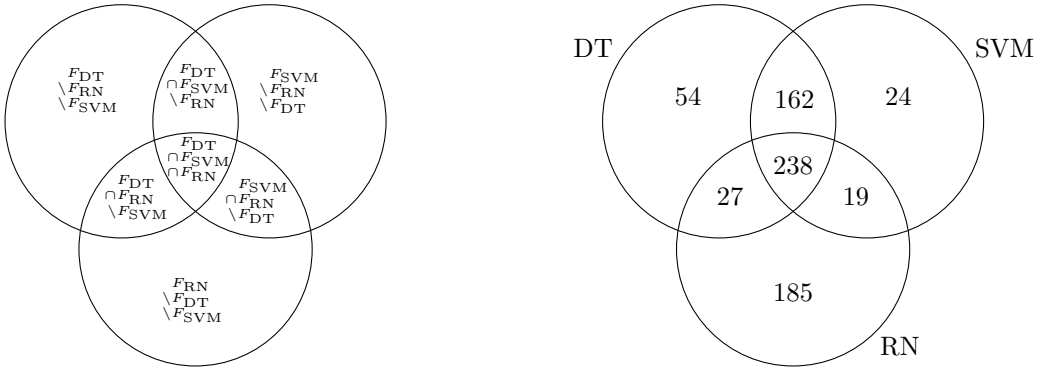


Figure 6.1: All possible failure set inclusion-exclusion configurations and the corresponding numbers of contained elements for failure sets on T_1.

SVM. For each classifier, almost half of the misclassified samples are also misclassified by the other two.

Inner test sets

The above observation motivates a closer look at the test samples. Figure 6.2 shows *open* and *closed* samples from $F_{SVM} \cap F_{SVM} \cap F_{RN}$. We see that for many samples the

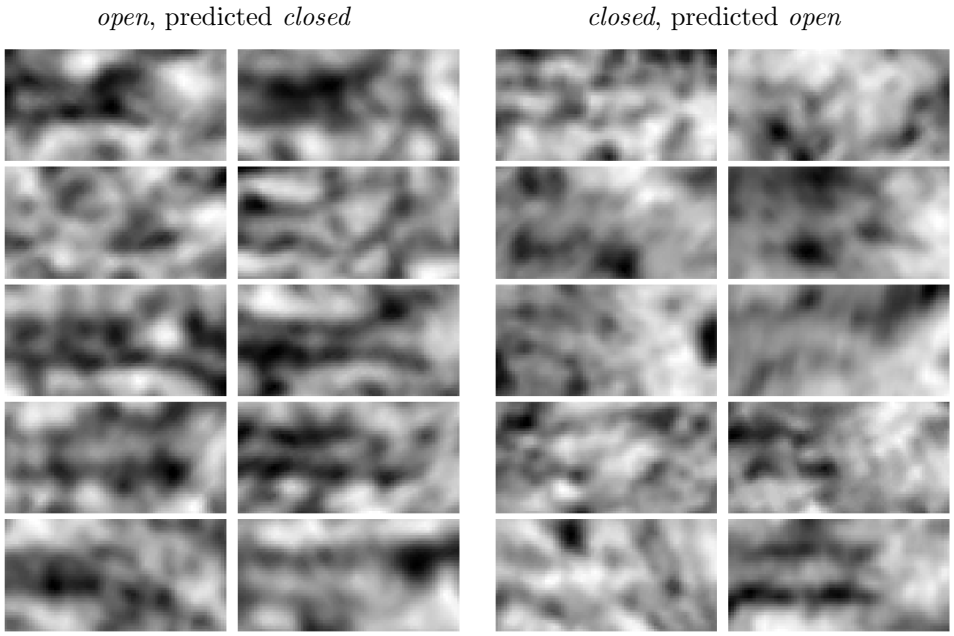


Figure 6.2: Samples in T_1 that no classifier got right.

actual class is hard to guess and it appears likely that even experts would argue on some cases. Therefore, we introduce an inner test set T_2 by taking all elements from T_1 that got at least two votes for *open* or *closed*. The test set melds down to 484 *open* and 123 *closed* samples. Table 6.3 lists the resulting confusion matrices. Apart from a 1% accuracy drop for DT on *closed* samples, all classifiers improved their performance on *both* classes. The min-accuracies improve for all classifiers while there is no dramatic exchange of sensitivity for specificity. Specifically, the min-accuracy improvements are DT: 9%, SVM: 3%, and RN: 15%. Gaps between the two per-class accuracies became

	DT		SVM		RN	
ground truth	*open*	*closed*	*open*	*closed*	*open*	*closed*
open	357 (74%)	127 (26%)	335 (69%)	149 (31%)	368 (76%)	116 (24%)
closed	36 (29%)	87 (71%)	15 (12%)	108 (88%)	19 (15%)	104 (85%)

Table 6.3: Confusion matrices for the final classifiers on the inner test set T_2. Percentages are obtained by normalizing on the number of samples per class ($\#open = 484$, $\#closed = 123$) in the inner test set.

inverted for DT, widened for SVM and became narrower for RN.

Again, we compute the failure set intersections on T_2. They are listed in Table 6.4.

S	F_{DT}	F_{SVM}	F_{RN}	$F_{\text{DT}} \cap F_{\text{SVM}}$	$F_{\text{DT}} \cap F_{\text{RN}}$	$F_{\text{SVM}} \cap F_{\text{RN}}$	$F_{\text{DT}} \cap F_{\text{SVM}} \cap F_{\text{RN}}$
$\lvert S \rvert$	163	164	135	119	64	74	53

Table 6.4: Sizes for intersections of failure sets on T_2.

The left image in Figure 6.3 shows the annotated Venn diagram derived from Table 6.4.

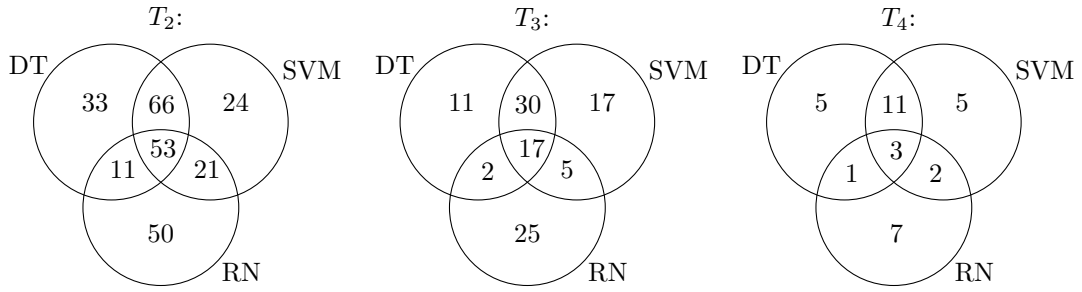


Figure 6.3: All possible failure set inclusion-exclusion configurations and corresponding number contained elements for failure sets on T_2, T_3, and T_4.

We also introduce the inner test sets T_3 and T_4, where we further reduce the considered endpoints to those seen by three and all four experts, respectively. Test set T_3 comprises 234 *open* and 49 *closed* samples and in test set T_4 there are 95 *open* and 20 *closed* elements left. Tables 6.5 and 6.6 show the confusion matrices and intersection set sizes. The center and right images in Figure 6.3 show the resulting annotated Venn diagrams obtained from the values in Table 6.6.

We see that the per-class accuracies for T_3 are significantly higher than for T_1. RN achieves accuracies past 80% in both classes. The small number of *closed* samples in T_4 diminishes the representative power of the *closed* accuracies, but the *open* accuracies on this subset reached at least 80% for all classifiers.

For two classifiers c and c', we can compute the ratios $|F_c \cap F_{c'}| \,/\, |F_c \cup F_{c'}|$ as a measure of how similar the two classifier predictions are. We call it the classifiers' resemblance. Computing the resemblance for each pair of classifiers and every test (sub-)set $T_1, ..., T_4$ gives Table 6.7. We see that while moving from T_1 to T_4, the resemblance decreases consistently for RN paired with each of the remaining classifiers. For DT and SVM we see an initial drop from T_1 to T_2 followed by a comparably high plateau.

	DT		SVM		RN	
ground truth	*open*	*closed*	*open*	*closed*	*open*	*closed*
T_3:						
open	182 (78%)	52 (22%)	172 (74%)	62 (26%)	191 (82%)	43 (18%)
closed	8 (16%)	41 (84%)	4 (8%)	45 (92%)	6 (12%)	43 (88%)
T_4:						
open	80 (84%)	15 (16%)	76 (80%)	19 (20%)	84 (88%)	11 (12%)
closed	5 (25%)	15 (75%)	2 (10%)	18 (90%)	2 (10%)	18 (90%)

Table 6.5: Confusion matrices for the final classifiers on inner test sets T_3 and T_4. Percentages are obtained by normalizing on the number of samples per class (T_3: $\#open = 234$, $\#closed = 49$, T_4: $\#open = 95$, $\#closed = 20$,) in the inner test sets.

S	F_{DT}	F_{SVM}	F_{RN}	$F_{\mathrm{DT}} \cap F_{\mathrm{SVM}}$	$F_{\mathrm{DT}} \cap F_{\mathrm{RN}}$	$F_{\mathrm{SVM}} \cap F_{\mathrm{RN}}$	$F_{\mathrm{DT}} \cap F_{\mathrm{SVM}} \cap F_{\mathrm{RN}}$
T_3:							
$\lvert S \rvert$	60	69	49	47	19	22	17
T_4:							
$\lvert S \rvert$	20	21	13	14	4	5	3

Table 6.6: Sizes for intersections of failure sets on T_3 and T_4.

c, c'	T_1	T_2	T_3	T_4
DT, SVM	0.75	0.57	0.59	0.52
DT, RN	0.39	0.27	0.2	0.14
SVM, RN	0.39	0.33	0.24	0.17

Table 6.7: Ratios $|F_c \cap F_{c'}| \,/\, |F_c \cup F_{c'}|$ for all pairs of classifiers and all test (sub-)sets.

7 Discussion and Conclusion

The aim of this thesis was to evaluate the performances of the three machine learning methods decision tree, support vector machine, and neural networks on a specific image classification task. The images under consideration were extracted from electron tomography reconstructions of cells and show ends of microtubules. Based on the morphology, these had to be classified as either *open* or *closed*.

Decision Tree

On test set T_1 comprising all non-contradictory labeled elements from datasets qs1, . . . ,qs4, the final decision tree classifier obtained per-class accuracies of

0.62 (*open*)
0.72 (*closed*).

The per-class accuracies on the training set of undersampled, balanced subsets of datasets ds1, ds2, and ds3 are

0.80 (*open*)
0.82 (*closed*).

This means that the generalization drop-offs are 18% and 10%, respectively, where *open* elements pose the bigger generalization challenge. This can be explained by the greater variation in morphology they exhibit. While closed ends are mostly similar, open ends can be blunt with an even or uneven cut-off, rolled to the sides, or anything in-between. The tree classifies on a 3-dimensional feature space, where features along each dimension are based on the element-wise difference of the per-class average images. For *open* ends, the single average image does not sufficiently represent all variations. Instead, the different end structures are averaged out and the resulting difference image boils down to a detector sensitive to pixels in the general center of the image. We see this phenomenon when we look at some of the misclassified samples (see Fig 3.11). Elements falsely labeled *closed* tend to be long and reach into the sensitive area while many of the elements falsely labeled *open* are rather short.

In Section 3.2.1 we saw that unbalanced input classes lead to prioritizing the majority class and that undersampling yields more balanced results than oversampling. On the downside, this reduces the number of usable training samples. It would be interesting to see whether more sophisticated oversampling techniques (see for example Ref. [23]) allow to use all available data while maintaining a balanced prediction performance.

Preprocessing also had a significant influence on prediction performance (Section 3.2.2). We saw that the right choice of preprocessing technique can benefit the performance. We applied decision trees with axis aligned splitting planes. In this setting, a natural preprocessing technique is PCA transformation of the input data. For images, this did not show systematic classification improvement (see Fig. 3.6). It can be concluded that, here, the main direction of sample variation does not correspond to the endpoint mor-

phology but is overshadowed by noise. For feature-based inputs, a slight improvement was detected, but the major benefit was obtained by per-image standardizing the images prior to feature extraction.

Main improvement for this classifier can be expected from the introduction of more independent features that better separate the inputs. Simply adding principle components from the training set did allow for a better classification on the training set but did not generalize well to new samples from different tomograms. An unsupervised approach could be to extract principle components of a dataset comprising microtubule ends from a variety of different tomograms in the hope that this allows for more dataset-independent features.

We did not apply an automatic feature extraction algorithm from the SIFT/HOG family. These rely on sharper defined image structures such as edges and corners that, while abundant in natural images, were not present consistently in the tomography data. A more promising approach would be to refine the applied features. Seeing that *open* ends are harder to classify with our features, a natural enhancement would be to cluster the training samples in each class based on a distance metric such as the Pearson correlation. If o and c denote the number of *open* and *closed* clusters, respectively, $o \cdot c$ difference images could be computed generating a set of features with finer resolution and improved separation.

Classifying in image space lead to early overfitting of the training set (see Fig. 3.7). This can be explained by the small ratio of samples to input dimensions. The images contain enough noise to allow perfect separation of the training samples while barely learning anything about the underlying distribution of end morphologies.

Random forests could improve the performance on this input space in two ways. Firstly, they allow to utilize more of the available training data; each tree in the forest could be trained on an independently selected, balanced subset of all available training data. Secondly, selecting a random subset of pixels for each decision tree in the random forest would reduce the available feature space for each classifier, reducing the chances that noisy pixels systematically dominate the learning process.

Support Vector Machine

The final support vector machine classifier obtained per-class accuracies of

$$\begin{aligned} &\mathbf{0.66}\ (\textit{open}) \\ &\mathbf{0.70}\ (\textit{closed}), \end{aligned}$$

on test set T_1, and

$$\begin{aligned} &0.79\ (\textit{open}) \\ &0.79\ (\textit{closed}) \end{aligned}$$

on the balanced, undersampled training set of elements from datasets ds1, ds2, and ds3. The generalization drop-offs, thus, are 13% and 9%, respectively. As for the decision tree, the support vector machine classifier performs weaker on the *open* class. This is not surprising since both classifiers were trained on similar feature spaces and the shortcoming mentioned above also hold in the current case. The final decision boundary is given in form of a linear plane in feature space with normal $\mathbf{w} \approx (-0.062, -0.991, 0.121)^T$ and offset $b \approx -5 \cdot 10^4$. We saw that the classification decision is based mainly on the second feature and that the offset is close to zero. Therefore, a similar performance can be

expected from a linear model that makes it prediction based on the sign of the second feature. We expect the major performance increase from the same ideas as in the section above regarding the increase of relevant features to allow for a better sample separation.

When we searched for a well-performing set-up (Section 4.2.1 and 4.2.2), we saw that support vector machines also perform imbalanced when trained on imbalanced classes. Here, again, the most balanced results were obtained from undersampling the dataset. Ref. [23] discusses oversampling techniques that particularly try to generate samples at the border to neighboring classes. These techniques would be of special interest for further improving the support vector machine results. A central prerequisite for applying these oversampling techniques is the existence of a reasonable distance metric between samples. A simple start could be given by the image correlation, but ultimately the measure should be able to reflect similarities between samples of the same class and differences to samples of other classes. Simple image correlations could be too sensitive to noise to achieve this. Applying PCA transformation after feature extraction did not show any systematic improvements.

In Section 4.2.3 we saw that large values for the trade-off parameter C, corresponding to little regularization, caused a great amount of performance variation. This phenomenon only occurred when training on features (compare Fig. 4.6 and 4.7) and it is likely to be connected to the dense clustering of samples from both classes. Significant variation remained after we had fixed everything but the random seeds for the support vector training (Section 4.2.3). We can conclude that different decision boundaries were found. This is surprising, as the support vector machine optimization problem is quadratic, thus, has a global minimum. It would be interesting to investigate the source of this variation.

Further resemblance to decision trees occurred when we tried to classify in image space. The learning algorithm overfitted the training data, while the best validation performances stopped to increase at least 5% below the best feature-based results. Here, support vector machines allow for a powerful generalization in form of kernel methods. Given an appropriate inner product between samples, they are capable of dealing efficiently with high-dimensional data. Choosing the scalar product between flattened, per-feature standardized images as inner product results in the Pearson correlation. Another alternative is given by choosing a Gaussian kernel. If such an approach does not hold the desired generalization performance, the inputs could be replaced by taking only representatives, such as the o and c cluster averages described above.

Neural Network

The final neural network classifier with residual network architecture reached per-class accuracies of

0.61 (*open*)
0.78 (*closed*),

on test set T_1 with training performances of

0.8 (*open*)
0.9 (*closed*),

on an oversampled, balanced training set of elements from in ds1, ds2, and ds3. The number of input images was increased by taking cross-sections and cylindrical average images from four different orientations. The generalization drop-offs are 19% and 12%, respectively.

Considering only those inputs on which the network was *sure* of the predicted class (probability > 0.9) reduced the test set to 375 *open* and 127 *closed* samples. Here, the obtained class-accuracies are

$$0.83\ (\textit{open})$$
$$0.8\ (\textit{closed}).$$

This increases the accuracy on *open* elements by 22%. We see that for many of the network's errors on *open* elements the accompanied prediction probabilities are low. Under the applied training set the network was capable to learn a reasonable class probability distribution. The relaxation of allowing the network to choose a third label, *undefined*, when the probability is too low noticeably improves the accuracy. This could be further harnessed by letting the network learn a class probability that reflects the number of experts that agreed on the label of a sample.

Looking at the samples that were misclassified despite the restriction to high prediction probability (see Fig 5.17) shows that many of the errors performed are reasonable. Many of the endpoints falsely classified as *closed* show features characteristic to both classes. Some images contain spreading microtubule walls indicating *open* ends as well as dark areas around the end as is typical for *closed* ends. For such images it would be interesting to analyze, how consistent the expert votes were. If the labeling is consistent and based on additional detail information contained in the images, fine-tuning the network could be attempted by augmenting more images to show such detail. Nonetheless, there were also ends falsely predicted as *open* that show clear characteristics of *closed* ends. It requires further investigation to see what led the network to such a prediction. We conjecture that it might be due to the thin lines that emanate from the end of *closed* microtubules that resemble the spreading walls of *open* ends.

During the search for a good training set-up, we saw that combining cross-sections and cylindrical averages from several orientations did not improve the training performance when compared with training on cross-sections from a single orientation (compare left-bottommost plot in Fig. 5.15 and right plot in Fig 5.14). It is possible that this step introduces misleading training samples as some closed microtubule ends can appear open when seen from an unfortunate angle. This requires further investigation, and possibly cleaning, of the increased training set. The best images are taken at an orientation close to being perpendicular to the tomogram z-direction. Tilting this plane increases the missing-wedge noise inherent in the tomography process, thus, the chance for misleading images might also increase.

Data augmentation showed the strongest effect on network performance when searching for means to improve network prediction (see Fig 5.15). Overfitting of the training set was significantly reduced; the training accuracy curves approached the validation accuracy curves by roughly 20% at the end of the training phases. Improvement of the validation curves was insignificant. We conclude that, while data augmentation has the potential to enhance network prediction, care has to be taken to find image manipulations that realistically reflect the sample variation. Simple geometric transformations and Gaussian blurring did not suffice in the present case. One approach to reproduce realistic noise could be to cut random image patches from different tomograms and

combine them with the training samples.

Increasing the weight decay parameter did not significantly improve generalization performance but prolonged the range in which the network showed its best validation performance (see the top rows in Fig. 5.13 and Fig. 5.14). The parameter value of 0.5 used to train the final network is unusually high which results from the small number of available training samples. It would be interesting to see whether additional regularization techniques, such as dropout [53], can improve validation performance.

The abundance of unlabeled microtubule ends also suggests trying an un-/semi-supervised approach to training neural networks by exploring the performance that can be obtained from training an autoencoder variant (see for example Ref. [57][46][32]).

Comparing the Results and Inner Test Sets

When we compared the results from the three classifiers (Section 6), we saw that there was a large set of test samples that were predicted falsely by all classifiers. This motivated the introduction of inner test sets T_2, T_3, T_4 of samples where at least $2, 3, 4$ experts agreed on the class. This can be interpreted as creating class probabilities from the expert votes with higher probability when more experts agreed on the label. The resulting confusion matrices for all test subsets are shown in Table 7.1. We mostly see a consistent rise of

	DT		SVM		RN	
ground truth	*open*	*closed*	*open*	*closed*	*open*	*closed*
T_1:						
open	638 (62%)	387 (38%)	673 (66%)	352 (34%)	630 (61%)	395 (39%)
closed	94 (28%)	242 (72%)	101 (30%)	235 (70%)	74 (22%)	262 (78%)
T_2:						
open	357 (74%)	127 (26%)	335 (69%)	149 (31%)	368 (76%)	116 (24%)
closed	36 (29%)	87 (71%)	15 (12%)	108 (88%)	19 (15%)	104 (85%)
T_3:						
open	182 (78%)	52 (22%)	172 (74%)	62 (26%)	191 (82%)	43 (18%)
closed	8 (16%)	41 (84%)	4 (8%)	45 (92%)	6 (12%)	43 (88%)
T_4:						
open	80 (84%)	15 (16%)	76 (80%)	19 (20%)	84 (88%)	11 (12%)
closed	5 (25%)	15 (75%)	2 (10%)	18 (90%)	2 (10%)	18 (90%)

Table 7.1: Confusion matrices for the final classifiers on test sets $T_1, \ldots, T_4$. Percentages are obtained by normalizing on the number of samples in each class of the ground truth in the test sets.

per-class accuracies while moving from T_1 to T_4 although the accuracy on *closed* samples in T_4 should be treated with care due to the small number of elements. Samples that were easier to classify for experts also were easier to classify by the algorithms.

When we looked at the ratios of failure set intersections to failure set unions for each test subset (Tab. 6.7), we saw that the decision tree and the support vector machine often made the same errors. This is reasonable since both were trained on similar input representations. Going from T_1 to T_4, the set of samples that were hard to classify for the neural network deviated more and more from the set of samples that were hard to classify for any of the other two classifiers. We see that the classifier start to make more individual errors in the inner subsets.

In conclusion, the neural network gave the most promising results. The performance of the other two classifiers relies substantially on the existence of an informative features.

The ability to successfully train on images without having to design features fundamentally eases the classification task. Furthermore, we saw that the neural network learned a reasonable class probability despite being trained on hard labels. We conjecture that treating the expert classification as a probability further increases the resemblance between the network output and the expert classification. Nevertheless, it is impressive how well the other two classifiers could keep up, considering their comparatively simple hypothesis space and the greatly condensed inputs they performed on.

Personal Conclusion

Upon the first glance at this task I thought that per-class accuracies of 80% should be within the realms of possibility. The actual results stay well behind. I believe that this is mainly due to the image noise and class imbalance which turned out to be a greater factor than anticipated, and so was the varying quality between tomograms. I had conjectured that neural networks would outpace the other methods by a considerable margin and I am surprised to see how well the other two performed. Especially, when considering the highly reduced feature space they were working on. Probably, all three classifiers can likely be improved, at least to some small extend: the feature-based methods most probably by enhancing the feature set and the network by further data collection or specific data augmentation. Especially the last technique is intriguing. Firstly, it increases the effective number of training samples without having to further conduct the tedious task of data collection. Secondly, it should allow for well-guided fine tuning of the network while treating it as a black box. If, for example, further analysis of the misclassified samples from one class shows that the network puts too much attention to irrelevant image features, samples from the other class could be augmented to also show more of these features.

In this thesis, the opportunity was given to acknowledge, explore, and tackle the many challenges that can occur when working on an interdisciplinary research problem. I am grateful that I was given the opportunity to work on such an interesting project and to gain practical insight into the field of machine learning applied to a very challenging real-world problem.

Bibliography

[1] tflearn. http://tflearn.org, 2017. [Online; accessed 11-February-2018].

[2] tflearn – resnet implementation. https://github.com/tflearn/tflearn/blob/master/examples/images/residual_network_cifar10.py, 2017. [Online; accessed 11-February-2018].

[3] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[4] Yaser S. Abu-Mostafa. Learning from data. https://work.caltech.edu/library/, 2012. [Online; accessed 15-December-2017].

[5] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[6] Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, COLT '92, pages 144–152, New York, NY, USA, 1992. ACM.

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

[8] Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[9] Leo Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

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

[11] Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. *Classification and regression trees*. CRC press, 1984.

[12] Arthur E Bryson. A gradient method for optimizing multi-stage allocation processes. In *Proc. Harvard Univ. Symposium on digital computers and their applications*, page 72, 1961.

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

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

[15] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. Liblinear: A library for large linear classification. *Journal of machine learning research*, 9(Aug):1871–1874, 2008.

[16] Daniel A Fletcher and R Dyche Mullins. Cell mechanics and the cytoskeleton. *Nature*, 463(7280):485, 2010.

[17] David A. Forsyth and Jean Ponce. *Computer Vision - A Modern Approach, Second Edition*. Pitman, 2012.

[18] Joachim Frank. *Electron tomography: methods for three-dimensional visualization of structures in the cell*. Springer, 2008.

[19] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning: data mining, inference and prediction*. Springer New York Inc., 2 edition, 2009.

[20] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 315–323, 2011.

[21] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[22] Thomas Hancock, Tao Jiang, Ming Li, and John Tromp. Lower bounds on learning decision lists and trees. *Information and Computation*, 126(2):114–122, 1996.

[23] Haibo He and Edwardo A Garcia. Learning from imbalanced data. *IEEE Transactions on knowledge and data engineering*, 21(9):1263–1284, 2009.

[24] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European Conference on Computer Vision*, pages 630–645. Springer, 2016.

[26] Tin Kam Ho. Random decision forest. In *Proc. of the 3rd Int'l Conf. on Document Analysis and Recognition, Montreal, Canada, August*, pages 14–18, 1995.

[27] Tin Kam Ho. The random subspace method for constructing decision forests. *IEEE transactions on pattern analysis and machine intelligence*, 20(8):832–844, 1998.

[28] Laurent Hyafil and Ronald L Rivest. Constructing optimal binary decision trees is np-complete. *Information processing letters*, 5(1):15–17, 1976.

[29] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. [Online; accessed 18-December-2017].

[30] Henry J Kelley. Gradient theory of optimal flight paths. *Ars Journal*, 30(10):947–954, 1960.

[31] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[32] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[33] H. W. Kuhn and A. W. Tucker. Nonlinear programming. In *Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability*, pages 481–492, Berkeley, Calif., 1951. University of California Press.

[34] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[35] Wei-Yin Loh. Fifty years of classification and regression trees. *International Statistical Review*, 82(3):329–348, 2014.

[36] John Mingers. An empirical comparison of pruning methods for decision tree induction. *Machine learning*, 4(2):227–243, 1989.

[37] James N Morgan and John A Sonquist. Problems in the analysis of survey data, and a proposal. *Journal of the American statistical association*, 58(302):415–434, 1963.

[38] Andrew Ng. Cs229 lecture notes, support vector machines. http://cs229.stanford.edu/notes/cs229-notes3.pdf, 2017. [Online; accessed 15-December-2017].

[39] Michael Nielsen. Neural networks and deep learning. http://neuralnetworksanddeeplearning.com/, 2017. [Online; accessed 3-January-2018].

[40] Edgar Osuna, Robert Freund, and Federico Girosi. An improved training algorithm for support vector machines. In *Neural Networks for Signal Processing [1997] VII. Proceedings of the 1997 IEEE Workshop*, pages 276–285. IEEE, 1997.

[41] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(Oct):2825–2830, 2011.

[42] John Platt. Sequential minimal optimization: A fast algorithm for training support vector machines. 1998.

[43] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[44] J Ross Quinlan. *C4. 5: programs for machine learning*. Elsevier, 2014.

[45] Stefanie Redemann, Johannes Baumgart, Norbert Lindow, Michael Shelley, Ehssan Nazockdast, Andrea Kratz, Steffen Prohaska, Jan Brugués, Sebastian Fürthauer, and Thomas Müller-Reichert. C. elegans chromosomes connect to centrosomes by anchoring into the spindle network. *Nature Communications*, 8(15288), 2017.

[46] Salah Rifai, Pascal Vincent, Xavier Muller, Xavier Glorot, and Yoshua Bengio. Contractive auto-encoders: Explicit invariance during feature extraction. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11, pages 833–840, USA, 2011. Omnipress.

[47] Raúl Rojas. *Neural networks: a systematic introduction*. Springer Science & Business Media, 2013.

[48] Frank Rosenblatt. *The perceptron, a perceiving and recognizing automaton Project Para*. Cornell Aeronautical Laboratory, 1957.

[49] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[50] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[51] Bernhard Schölkopf and Alexander J Smola. *Learning with kernels: support vector machines, regularization, optimization, and beyond*. MIT press, 2002.

[52] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

[53] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[54] Detlev Stalling, Malte Westerhoff, and Hans-Christian Hege. Amira: A highly interactive system for visual data analysis. In Charles Hansen and Christopher Johnson, editors, *The Visualization Handbook*, pages 749 – 767. 2005.

[55] Akif Uzman. Molecular biology of the cell (4th ed.): Alberts, b., johnson, a., lewis, j., raff, m., roberts, k., and walter, p. *Biochemistry and Molecular Biology Education*, 31(4):212–214, 2003.

[56] Vladimir Vapnik. *The nature of statistical learning theory*. Springer science & business media, 2 edition, 2013.

[57] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of Machine Learning Research*, 11(Dec):3371–3408, 2010.

[58] Britta Weber, Garrett Greenan, Steffen Prohaska, Daniel Baum, Hans-Christian Hege, Thomas Müller-Reichert, Anthony Hyman, and Jean-Marc Verbavatz. Automated tracing of microtubules in electron tomograms of plastic embedded samples of caenorhabditis elegans embryos. *Journal of Structural Biology*, 178(2):129 – 138, 2012.

[59] Britta Weber, Erin M. Tranfield, Johanna L. Höög, Daniel Baum, Claude Antony, Tony Hyman, Jean-Marc Verbavatz, and Steffen Prohaska. Automated stitching of microtubule centerlines across serial electron tomograms. *PLoS ONE*, page e113222, 2014.

[60] Paul John Werbos. Beyond regression: New tools for prediction and analysis in the behavioral sciences. *Doctoral Dissertation, Applied Mathematics, Harvard University, MA*, 1974.