Efficient learning machines

-

from kernel methods to deep learning

vorgelegt von
M.Sc.
Maximilian Alber
ORCID: 0000-0002-4614-6297

von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften
– Dr. rer. nat. –

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender	Prof. Dr. Benjamin Blankertz
Gutachter	Prof. Dr. Klaus-Robert Müller
Gutachter	Prof. Dr. Fei Sha
Gutachter	Prof. Dr. Volker Markl

Tag der wissenschaftlichen Aussprache: 10. Mai 2019

Berlin 2019

Abstract

Science is in a constant state of evolution. There is a permanent quest for advancing knowledge in the light of changing capabilities and matters. The field of Machine Learning itself is shaped by the ever-increasing amount of data and computing power, creating new challenges as well as paving the way for new opportunities. This thesis is on adapting learning-based machines to these emerging prospects. In particular, subject of this work are three distinct research topics with the underlying drivers: the wish to reliably predict (a) given a large number of classes, (b) given a large number of samples, and (c) to understand complex algorithms and data models. Each reflects a unique need for an efficient proposition and we contribute by creating approaches located in the intersection of algorithm and software development in order to tackle the following problem statements.

The first contribution researches multi-class classification with large label spaces and the effective prediction method support vector machines. Recent work suggests so-called all-in-one support vector machine formulations outperform one-vs.-rest formulations, but it is a challenge to leverage their potential for settings with a large number of classes. We approach this problem by proposing for two all-in-one machines exact optimization algorithms that distribute computation and model parameters evenly on computing instances. This allows us to perform an analysis on text data with a large label spaces and to confirm the favorable performance of all-in-one formulations.

Other cornerstones of Machine Learning are kernel methods and neural networks. The ever-growing data collections expose a scaling issue of kernel methods with respect to a large number of data points. The predominant approach to alleviate this is an approximation based on random features. In our second contribution we argue that this randomness renders the method inefficient and dissect the effect of these data- and task-agnostic learning bases by means of an empirical study. Viewing approximated kernel machines as neural networks and a novel, efficient optimization approach enable us to shed light onto the interplay of these two important learning paradigms.

Our last contribution aims to facilitate a better understanding of the predictions of deep neural networks. These data models have shown impressive results in a wide range of applications and are an invaluable tool. Yet, compared to many other Machine Learning techniques, their functioning is hard to understand and to retrace. Among many proposed methods, propagation-based prediction analysis has shown convincing results and is a promising candidate to address this shortcoming. A drawback is the lack of efficient software for many methods and emerging network structures. We contribute to this by developing the software library *iNNvestigate*, whose features are an intuitive interface and a modular design — enabling non-expert users access to these methods as well as accelerating research on complex neural networks.

Zusammenfassung

Die Wissenschaften befinden sich in einer fortwährenden Evolution, welche von den sich permanent ändernden Fähigkeiten und Gegebenheiten getrieben ist. Das Feld des maschinellen Lernens selbst ist durch ständig wachsende Datenmengen und Rechenleistungen geprägt, was sowohl neue Herausforderungen schafft als auch den Weg für neue Lösungen ebnet. Diese Arbeit beschäftigt sich mit der Anpassung von lernbasierten Methoden an diese neuen Perspektiven. Gegenstand dieser Arbeit sind insbesondere drei unterschiedliche Forschungsbereiche mit den folgenden, zugrundeliegenden Treibern: dem Wunsch eine zuverlässige Vorhersage zu treffen (a) gegeben einer großen Anzahl von Klassen oder (b) gegeben einer großen Anzahl von Datenpunkten, und (c) komplexe Datenmodelle besser verstehen zu können. Jeder dieser Punkte bedarf einer effizienten Lösung und um die folgenden Problemstellungen zu bearbeiten, erforschen wir Ansätze im Schnittpunkt von Algorithmen- und Software-Entwicklung.

Der erste Beitrag behandelt die Klassifizierung mit einer großen Anzahl von Klassen sowie die effektiven Vorsagemethoden Stützvektormaschinen. Vielversprechende Formulierungen dieser Methode optimieren alle Klassen gleichzeitig, jedoch stellt sich die Nutzung ihres Potentials in der genannten Anwendung als Herausforderung dar. Für zwei solche Formulierungen schlagen wir exakte Optimierungsverfahren vor, deren Berechnungen und Modellparameter sich auf verschiedene Recheninstanzen verteilen lassen. Dies ermöglicht es uns eine Analyse mit großen Textdaten durchzuführen und die Überlegenheit von Klassen-übergreifenden Ansätzen zu bestätigen.

Weitere Eckpfeiler des maschinellen Lernens sind Kernmethoden und neuronale Netze. Das Aufkommen von Datensätzen mit einer großen Anzahl von Datenpunken offenbart ein Skalierungsproblem von Kernmethoden und der vorherrschende Lösungsansatz beruht auf einer Approximation mit Zufallszahlen. In unserem zweiten Beitrag argumentieren wir die Ineffizienz der Nutzung von Zufallszahlen und zerlegen den Effekt dieser Daten- und Problem-unabhängigen Methode in einer empirischen Studie. Dazu betrachten wir approximierte Kernmaschinen als neuronale Netze und entwickeln einen neuartigen, effizienten Optimierungsansatz um den Übergang zwischen den beiden Lernparadigmen zu untersuchen.

Das Ziel unseres letzten Beitrags ist ein besseres Verständnis für die komplexen Arbeitsweisen tiefer, neuronaler Netze zu ermöglichen. Dieses wertvolle Lernwerkzeug hat in einer Vielzahl von Anwendungen beeindruckende Ergebnisse erzielt. Seine Funktionsweise ist aber im Vergleich zu anderen Techniken schwierig nachzuvollziehen. Eine Reihe von Methoden wurde als Lösung vorgeschlagen und darunter sind "propagation"-basierte Analysen überzeugende Kandidaten — jedoch fehlt für viele dieser Algorithmen effiziente Software. Mit der Entwicklung der Softwarebibliothek *iNNvestigate*, welche sich durch eine intuitive Schnittstelle und einen modularen Aufbau auszeichnet, eröffnen wir Laien den Zugang zu solchen Methoden und beschleunigen deren Forschung mit komplexen, neuronalen Netzen.

Acknowledgements

Words seldom cover the whole story and nor does or should this thesis reflect the entirety of my PhD time. Yet, it marks the conclusion of this phase in my life and I would like to use this opportunity to thank the dear people involved.

Danke Mama fir die bedingungslose Unterstützung, 'es Vertrauen in mi, und dein Einsotz mir a guats Leben zu ermeglichen — une di war'i net dor Mensch, der'i bin!

Kära Emilie, vi föjdes åt under större delen av min avhandlingstid, i vilken, att citera Kilpi, "du rubbar hela min existens". Jag vill tacka dig för allt nytt du förde in i mitt liv, för ditt stöd och din kärlek.

In this period I had a lot of fun and joy with my friends and colleagues and received a lot of support and advice from them. Alex, Beccy, Nina, and Philipp thank you — so much — for the good times, all the talks about life and for being around when needed and when not! Alma, Annika, Daniel, Jack, Linus, Lisa, Magda, Mattias, Nico, Stephi, Yori and many more: I am very happy and grateful for the nice times spend with you guys! Special thanks I owe you, Sven and Pieter-Jan, for the guidance and support throughout the — sometimes hard — PhD time and beyond! While I stayed in California, you welcomed me to your beautiful home: I thank you wholeheartedly Christine, John, and Tom!

In four years a lot of life happens and I want to express my deepest gratitude to you, Klaus, for the unconditional trust in difficult times and for facilitating my research efforts with freedom and support. Your guidance and patience is the basis for the great work environment in our lab!

My research was always a product of synergies with awesome people and I thank Ann-Kathrin, David, Grégoire, Irwan, Julian, Klaus, Kristof, Miriam, Pan, Philipp, Pieter-Jan, Prajit, Quoc, Sara, Sebastian, Stephi, Sven, Urun, Wojciech, and many more for insightful discussions and fruitful collaborations. Especially, I would like to thank Fei for sharing his experience and views on Machine Learning — I learned a lot during our project, thank you! — and Marius for supporting me in my first project and beyond.

No man is an island — I am very happy to be part of such a nice lab — thank you peeps! During my PhD time I worked in the BBDC project and thank Mikio for introducing me to it and Shin for leading it so carefully. While writing this text I was glad to receive advice from Alex and reviews from Andreas, Kristof, Marina, Pieter-Jan, and Stephi. Thank you! I also thank Andrea and Dominik for their patient assistance in administrative and technical matters. Finally, I would like to thank Irwan, Sara, Gabriel and Quoc for the good time at Google.

Abbreviations and notation

Abbreviations

Abbreviation	Description
CPU	Central processing unit
CS	Crammer and Singer (SVM formulation)
DAB	Discriminatively adapted basis
DAG	Directed acyclic graph
DBCA	Dual block coordinate ascent
densenet121	Name of a neural network from Huang et al. (2017)
DMOZ	Directory Mozilla
DTD	Deep Taylor Decomposition (prediction analysis method)
GB	Guided Backprop (prediction analysis method)
GPU	Graphics processing unit
IG	Integrated Gradients (prediction analysis method)
inception_v3	Name of a neural network from Szegedy et al. (2016)
I*G	Input * gradient (prediction analysis method)
KAE	Kernel approximation error
LIME	"Local interpretable model-agnostic explanations" (prediction analysis method)
LLW	Lee, Lin, and Wahba (SVM formulation)
LRP	Layer-wise Relevance Propagation (prediction analysis method)
LSHTC	A benchmark for large-scale text classification
MC	Multi-core
MC-SVM	Multi-class SVM
ML	Machine Learning
MPI	Message passing interface
NASNetA/nasnet_large	Name of a neural network from Zoph et al. (2018)
OVR	One-vs.-rest (SVM)
OVO	One-vs.-one (SVM)
PCA	Principal component analysis
RB	Random basis
ReLU	Rectified linear unit
resnet50	Name of a neural network from He et al. (2016)
SAB	Supervised adapted basis
SG	SmoothGrad (prediction analysis method)
SGD	Stochastic gradient descent

Abbreviations

Abbreviation	**Description**
SpecINT	Benchmark for CPUs from the Standard Performance Evaluation Corporation
SVM	Support Vector Machine
SW	Software
TF/IDF	Term frequency-inverse document frequency (pre-processing)
UAB	Unsupervised adapted basis
VGG16/vgg16	Name of a neural network from Simonyan and Zisserman (2014)
WW	Weston and Watkins (SVM formulation)

Mathematical notation

Symbol	**Description**
$\mathbb{1}_{predicate}$	Indicator function: 1 if predicate is true, 0 otherwise.
$\mathcal{C}$	Number of classes
C	SVM regularization parameter
d	Number of input dimensions
D	Number of (random) features or size of embedding
$\bar{d}$	Average number of non-zero input features per sample
$k(x, x') : \mathbb{R}^d \times \mathbb{R}^d \mapsto \mathbb{R}$	Kernel function
$l(x) = \max\{0, 1 - x\}$	Hinge loss
n	Number of training samples
$\bar{n}$	Average number of input samples per class
n_{max}	Maximum number of input samples per class
$\phi : \mathbb{R}^d \mapsto \mathbb{R}^D$	Feature function
σ	Bandwidth of Gaussian kernel
x_i	Features of sample i
y_i	Label of sample i

Specific notations are presented in the corresponding contexts.

Contents

Chapter I

Introduction

1 Challenges in Machine Learning

Science is in a constant state of evolution. As opportunities emerge new research questions arise, other areas call for adaptation to changing conditions, and again others become obsolete. The field of Machine Learning itself has steady drivers for such change, and the two most important ones are data and computing power. The inherent and close bond with both factors stimulated and facilitated many breakthroughs, including deep learning.

Figure I.1 exemplarily depicts the rapid change and shows the increase of computing capabilities and data availability in the last decades. On one hand, this influence is characterized by the progress of computational frameworks — from the increase of single-core speedup (E.g., Schaller, 1997) and the advent of parallel and distributed systems (E.g., Gropp et al., 1996; Dagum and Enon, 1998) to dedicated Machine Learning hardware (E.g., Jouppi et al., 2017) — calling for and enabling advancements in Machine Learning methodologies (E.g., Rumelhart et al., 1986; Cortes and Vapnik, 1995; Breiman, 1996; Freund and Schapire, 1996; Breiman, 2001; Müller et al., 2001; Schölkopf and Smola, 2002; Montavon et al., 2012; LeCun et al., 2015). On the other hand, this is backed by data collection and management technologies — from relational databases (E.g., Codd, 1970) over to distributed, NoSQL storages (E.g., DeCandia et al., 2007; Chang et al., 2008; Lakshman and Malik, 2010; Vavilapalli et al., 2013) and adaptive, stream-oriented systems (E.g., Alexandrov et al., 2014; Toshniwal et al., 2014; Carbone et al., 2015) — fostering a data basis to express the upcoming modeling capabilities (E.g., Schölkopf et al., 1998; Simonyan and Zisserman, 2014; Szegedy et al., 2016; Schütt et al., 2017a; Mikolov et al., 2013; Sutskever et al., 2014; Vaswani et al., 2017).

The dependence on mathematical and computational frameworks as well as on the availability of data characterizes the unique and inter-disciplinary position of Machine Learning. To unveil its abilities, this field is reliant on a fruitful synergy between mathematical/methodological and engineering domains. An example is the success (Krizhevsky et al., 2012) of deep, convolutional neural network at the ImageNet competition (Deng et al., 2009) which was caused by algorithmic advances (Nair and Hinton, 2010; Srivastava et al., 2014; Bottou, 2010) hand in hand with a tailored implementation to exploit the available hardware capabilities maximally. One reason for the need of such synergy is the complexity of learning algorithms. Far from being embarrassingly parallel, many exhibit worse than linear asymptotic complexity, which causes their needs to outgrow the available resources easily. Accordingly, a large body of Machine Learning research is about making learning faster and more efficient, e.g.,

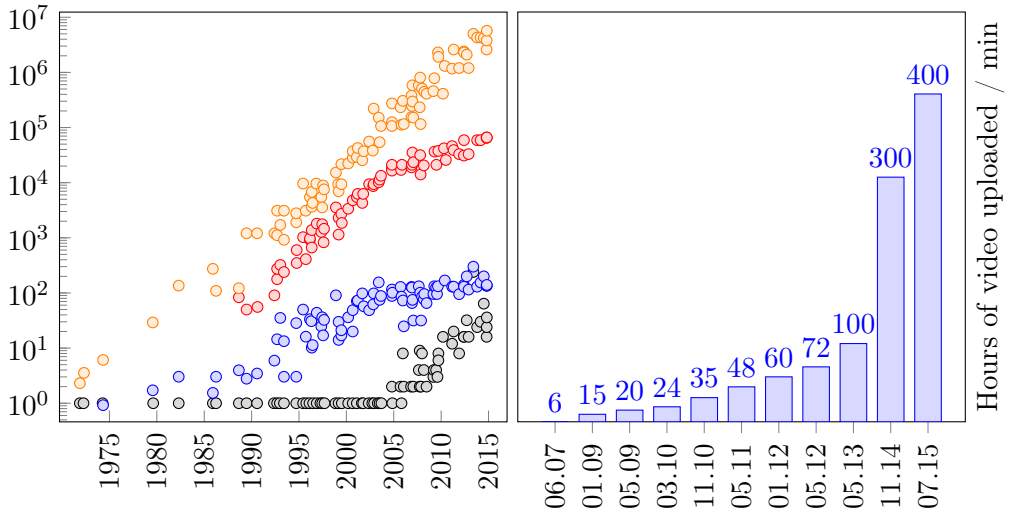

Fig. I.1: Machine Learning drivers. *Inherent drivers of Machine Learning are the increase in computing power and data availability. This* exponentional *development is exemplarily sketched by the presented plots. The plot on the left side shows the characteristics of CPUs over four decades*[1]*. The depicted properties are: Transistors* $/10^3$*, SpecINT* $\times 10^3$*, Typical power consumption in Watt, Number of logical cores. The second plot depicts the average amount of video hours uploaded to the platform YouTube in the indicated months*[2]*.*

Platt (1999), Müller et al. (2001), LeCun et al. (2012), Bottou (2010), Rahimi and Recht (2008), Ioffe and Szegedy (2015), and Nair and Hinton (2010).

Above all, next to this perpetual strive for tackling larger problems new and distinct challenges arise. For instance, the application of learning machines in critical decision processes sparked interest in research on attack scenarios (E.g., Dalvi et al., 2004; Goodfellow et al., 2014; Kurakin et al., 2016; Papernot et al., 2017) or privacy (E.g, Shokri and Shmatikov, 2015; Abadi et al., 2016a) in Machine Learning. Special among these new topics, and driven by models becoming increasingly complex due to the potential of leveraging knowledge from more and more data, is the revived desire to understand and explain the workings of data models (E.g., Baehrens et al., 2010; Zeiler and Fergus, 2014; Springenberg et al., 2015; Lundberg and Lee, 2017; Ancona et al., 2018; Montavon et al., 2018).

These challenges and the described interplay of the mathematical and engineering disciplines in Machine Learning is what this thesis is dedicated to: the development of new algorithms as well as the implementation of new software to tackle the research problems posed by the increase in computing power, available data, and modeling capabilities.

[1] Data up to the year 2010 was collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten. Data was collected for 2010-2015 by K. Rupp. Source: https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/

[2] Source: https://www.statista.com/statistics/259477/hours-of-video-uploaded-to-youtube-every-minute/

2 Contribution of this thesis

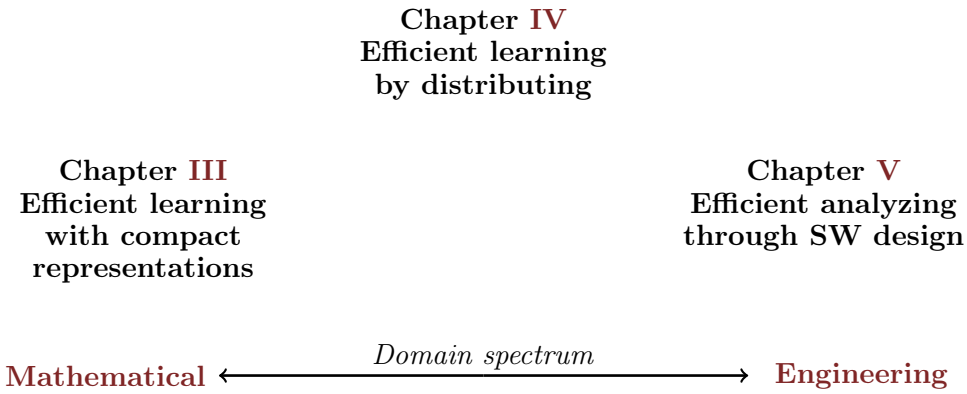

Fig. I.2: Machine Learning domain spectrum.

The overall objective of this work is to find new ways to deal with the challenges given by the rising data availability, computational capabilities, and model capacities. Specifically, this thesis contributes to the described evolution by choosing and treating three distinct research questions and aims. Together they form an exemplary representation of the inter-disciplinary nature of Machine Learning and Figure I.2 depicts this spectrum figuratively by showing where the three contributions of this thesis reside. Each of the following propositions is characterized by a different setting and solution approach:

Chapter III: Distributed optimization of multi-class SVMs

- Setting: Support vector machines (SVMs) are an important tool for classification problems, yet promising all-in-one SVM formulations hardly scale to problems with a large number of categories.
- *Research question: Can all-in-one SVMs training be distributed and scaled to problems with a large amount of classes and, if so, do all-in-one SVMs still perform favorably compared to one-vs.-rest SVMs?*
- Contribution: The development of scalable, distributed optimization algorithms for two promising all-in-one formulations and subsequently a comparison with one-vs.-rest SVMs on large text data.

Chapter IV: Efficient learning of kernel approximations

- Setting: Kernel machines are a well-researched and reliable method for non-linear prediction problems and kernel approximations with agnostic, random basis functions are used to scale them to problems with a large number of samples.

- *Research question: Can the performance of such kernel machines be increased by adapting the basis' parameters either to the kernel function or to the task at hand, instead of using agnostic, random sampling?*
- Contribution: The development of an end-to-end optimization scheme and the design and execution of an empirical evaluation to examine the space between approximated kernel machines and neural networks.

Chapter V: Efficient software for prediction analysis

- Setting: Due to growing capabilities data models are becoming increasingly complex, thereby harder to understand and to retrace. Propagation-based prediction analysis is a promising approach to remedy this, yet has a lack of efficient software for many methods and for emerging network structures.
- *Research aim: To reveal, develop and implement approaches for efficient implementations of propagation-based prediction analysis with an emphasis on the usability for the non-expert user and facilitating research on emerging network architectures.*
- Contribution: We develop the software library *iNNvestigate* featuring an intuitive, accessible interface as well as a modular design to allow focused research and show the first successful applications.

The remaining part of this thesis is organized as follows: Chapter II treats the fundamentals such as basic Machine Learning concepts, optimization, prediction analysis algorithms and relevant engineering topics and Chapter VI completes this work with a summary and a conclusion.

2.1 Included publications

The work presented in this thesis has been or is about to be published in peer-reviewed journals, books and conferences. Indeed, the main chapters of this thesis follow these very closely. The following list enumerates the publications in chronological order and a note at the end of every entry indicates the status of the publication.

1. M Alber, J Zimmert, U Dogan, and M Kloft (2017b). "Distributed optimization of multi-class SVMs". In: *PLOS ONE* 12.6, pp. 1–18
2. M Alber, P-J Kindermans, KT Schütt, K-R Müller, and F Sha (2017a). "An Empirical Study on The Properties of Random Bases for Kernel Methods". In: *Advances in Neural Information Processing Systems 30*, pp. 2763–2774
3. M Alber, S Lapuschkin, P Seegerer, M Hägele, KT Schütt, G Montavon, W Samek, K-R Müller, S Dähne, and P-J Kindermans (2019). "iNNvestigate neural networks!" To appear in: *Journal of Machine Learning Research* (Accepted)
4. M Alber (2019). "Software and application patterns for explanation methods". To appear in: *Interpretable AI: Interpreting, Explaining and Visualizing Deep Learning*. Springer (Accepted)

2.2 All publications

The subsection contains all the publications I have (co-)authored during my work for this thesis. Again, the list enumerates the publications in chronological order and a note at the end of every entry indicates the status of the publication.

Journal articles

1. M Alber, J Zimmert, U Dogan, and M Kloft (2017b). "Distributed optimization of multi-class SVMs". In: *PLOS ONE* 12.6, pp. 1–18

2. M Alber, S Lapuschkin, P Seegerer, M Hägele, KT Schütt, G Montavon, W Samek, K-R Müller, S Dähne, and P-J Kindermans (2019). "iNNvestigate neural networks!" To appear in: *Journal of Machine Learning Research* (Accepted)

Book chapters

3. M Alber (2019). "Software and application patterns for explanation methods". To appear in: *Interpretable AI: Interpreting, Explaining and Visualizing Deep Learning*. Springer (Accepted)

4. P-J Kindermans, S Hooker, J Adebayo, M Alber, KT Schütt, S Dähne, D Erhan, and B Kim (2019). "The (Un)reliability of saliency methods". To appear in: *Interpretable AI: Interpreting, Explaining and Visualizing Deep Learning*. Springer (Accepted)

Conference articles

5. M Alber, P-J Kindermans, KT Schütt, K-R Müller, and F Sha (2017a). "An Empirical Study on The Properties of Random Bases for Kernel Methods". In: *Advances in Neural Information Processing Systems 30*, pp. 2763–2774

6. P-J Kindermans, KT Schütt, M Alber, K-R Müller, D Erhan, B Kim, and S Dähne (2018). "Learning how to explain neural networks: PatternNet and PatternAttribution". In: *International Conference on Learning Representations*

7. A-K Dombrowski, M Alber, CJ Anders, M Ackermann, K-R Müller, and P Kessel (2019). "Could not get lock /var/lib/dpkg/lock-frontend". Submitted to: *Advances in Neural Information Processing Systems 33* (Submitted)

8. S Brandl, D Lassner, and M Alber (2019). "Balancing the composition of word embeddings". Submitted to: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing* (Submitted)

Conference workshop contributions

9. M Alber, J Zimmert, U Dogan, and M Kloft (2016). "Distributed optimization of multi-class SVMs". In: *Neural Information Processing Systems 2016 - Extreme Classification workshop* (Received best paper award)

10. P-J Kindermans, S Hooker, J Adebayo, M Alber, KT Schütt, S Dähne, D Erhan, and B Kim (2017). "The (Un)reliability of saliency methods". In: *Neural Information Processing Systems 2017 - Interpreting, Explaining and Visualizing Deep Learning - Now what? workshop*

11. M Alber, I Bello, B Zoph, P-J Kindermans, P Ramachandran, and Q Le (2018a). "Backprop Evolution". In: *International Conference on Machine Learning 2018 - AutoML workshop*

12. M Alber, S Lapuschkin, P Seegerer, M Hägele, KT Schütt, G Montavon, W Samek, K-R Müller, S Dähne, and P-J Kindermans (2018b). "How to iNNvestigate neural networks' predictions!" In: *Neural Information Processing Systems 2018 - Machine Learning Open Source Software workshop*

Chapter II

Fundamentals

Based on the spectrum of Machine Learning presented in the introduction we split the fundamentals for this thesis in two parts. The first one will cover mathematical and methodological topics regarding theory and algorithms on learning machines. The second one will introduce engineering tools and approaches for Machine Learning applications.
While it is hard to draw a clear line between the domains, we note for the readers awareness that — according to the overview in Figure I.2 — the Machine Learning fundamentals on learning are mostly applied in Chapter III and IV and on analyzing predictions in Chapter V. The engineering fundamentals are most practical in Chapter III and V.

1 Machine Learning

Along with Big Data technologies, Machine Learning is the technical engine of Artificial Intelligence (Russell and Norvig, 2016) and is the practice of designing algorithms that, given data, are able to create and use (mathematical) models to make predictions and decisions — without being *explicitly* programmed to perform the task at hand (Bishop, 2006). Subsequently we will introduce how machines can perform learning and make predictions as well as how one can analyze the resulting data models. We will confine this description to the areas and algorithms concerning this thesis, namely supervised classification using support vector machines, kernel methods and neural networks and the analysis of the according predictions. A further treatment of the topics is out of the scope of this thesis and we refer the interested reader to the dedicated literature (E.g., Bishop, 2006; Duda et al., 2012; Montavon et al., 2012; Goodfellow et al., 2016).
This section is organized in three parts. The first will introduce a theoretical framework for classification tasks and how it can be used for gradient based optimization of models. The second will cover specific algorithms for tackling such problems and the last will present techniques to interpret the learned prediction processes.

1.1 Learning models

Learning can be decomposed in three factors: representation, evaluation, and optimization (Domingos, 2012). The representation specifies how data and model parameters are encoded, the optimization describes how model parameters are adapted or "learned" to fit the posed task, and, finally, the evaluation states the measure for evaluating

the resulting model. Examples are support vectors, trees; evolutionary algorithms, continuous optimization; and accuracy, squared error respectively (Domingos, 2012). In this section we treat optimization and evaluation of data models. The next section gives examples how these are combined with chosen representations in specific learning algorithms.

1.1.1 Risk minimization for classification tasks

In Machine Learning three major types of learning are distinguished depending on the learning signal during training (Bishop, 2006): (1) Supervised learning, where for each sample the correct solution is given. (2) Reinforcement learning, where during training an oracle only reveals, if a proposed solution is correct or not. (3) Unsupervised learning, where no learning signal is given.

In this work we are concerned with supervised learning and more specifically with classification. In more detail, we assume to be given identical and independently distributed (iid) training data $(x_1, y_1), \ldots, (x_n, y_n)$ which is drawn from a (unknown) data distribution $P(x, y)$, where the features are $x_i \in \mathbb{R}^d$ and the labels are $y_i \in \{-1, +1\}$ for binary classification and $y_i \in \{1, ..., \mathcal{C}\}$ for multi-class classification. The fact that we are given the true labels for each data point makes this a supervised learning problem. In the following we use the binary classification setup to present the theoretical foundations. The results can be extended to the multi-classification case.

Our goal is to find some function f that is able to classify *unseen* data x correctly. The first step to approach this objective is to define an evaluation metric that estimates the classification ability and the field of statistical learning theory (Vapnik, 1995) defines tools for such purpose. The expected error for a function f is defined as:

$$R[f] = \int l(f(x), y) \;\; dP(x, y) \tag{II.1}$$

where l is a loss function suited to the task at hand. Predominant for classification is the Hamming or 0/1 loss $l(y, y') = \mathbb{1}_{y=y'}$.

The function f that solves our objective best, is the one that minimizes the expected error. Unfortunately, the data distribution $P(x, y)$ is typically unknown and we cannot compute or minimize the expected error directly. We are only given a sample of the data distribution, namely the training data, and approximate the expected error with the *empirical* risk — the sum of the losses on the training set:

$$R_{emp}[f] = \frac{1}{n} \sum_{i=1}^{n} l(f(x_i), y_i) \tag{II.2}$$

One can always obtain a empirical error of zero — by simply memorizing the feature/label pairs — and the art of Machine Learning is to find a solution that nonetheless *generalizes* to unseen data, i.e., minimizes the expected error. The discrepancy between expected error and empirical risk is known as generalization error and Vapnik

and Chervonenkis (1974) give a bound for this true risk:

$$R[f] \leq R_{emp}[f] + \sqrt{\frac{h(ln(\frac{2n}{h}) + 1) - ln(\frac{\delta}{4})}{n}} \quad \forall \delta \geq 0 \tag{II.3}$$

with a probability of at least $1-\delta$ when $n > h$ and where h is the Vapnik-Chervonenkis (VC) dimension. The VC dimension is a measurement for the separability of data points of different classes.
Given predictions for a dataset there are several ways to evaluate it. The Hamming loss or accuracy as evaluation is predominant for classification problems, but, depending on the given tasks, alternative evaluation functions might be more suitable. For instance consider binary classification where we have the classes "false" and "true", then precision and recall are defined as follows (Fawcett, 2006):

$$precision(Y, \hat{Y}) = \sum_i \mathbb{1}_{Y=1 \text{ and } \hat{Y}=1} / \sum_i \mathbb{1}_{\hat{Y}=1} \tag{II.4}$$

$$recall(Y, \hat{Y}) = \sum_i \mathbb{1}_{Y=1 \text{ and } \hat{Y}=1} / \sum_i \mathbb{1}_{Y=1} \tag{II.5}$$

and are measures of how many samples classified as true are actually true and of how many of the actually true samples where classified as true. Both measures are combined by the harmonic mean in the so-called F1 score, which defined as follows (Rijsbergen, 1979):

$$f1\_score(Y, \hat{Y}) = \frac{2\ precision(Y, \hat{Y})\ recall(Y, \hat{Y})}{precision(Y, \hat{Y}) + recall(Y, \hat{Y})} \tag{II.6}$$

The F1-score can be extended to multi-class settings by, e.g., ignoring the class distribution (micro) or by calculating the F1-scores for each class individually and taking the mean (macro) [1]. The presented measures are useful, e.g., for classification problems with a skewed class distribution.

1.1.2 Optimization

Given a loss for evaluation our goal is to find a solution that minimizes the empirical risk (Vapnik, 1995). In practice this is done by choosing a function family, e.g., a function f with parameters Θ, and minimizing:

$$\min_{\Theta} \quad \frac{1}{n} \sum_i l(f(x_i; \Theta), y) \tag{II.7}$$

While the empirical risk is theoretically bounded, in practice the (estimated) generalization error might be not good enough and Machine Learning proposes many ways to improve learning. One way to improve the true risk is to collect more data or use data augmentation to lower the bound, if possible, and another way is limit

[1] Scikit-learn F1-measures: https://scikit-learn.org/stable/modules/model_evaluation.html#precision-recall-f-measure-metrics

the function complexity to prevent overfitting to the training data — so-called regularizing. This last approach is theoretically analyzed under the term structural risk optimization (Vapnik and Chervonenkis, 1974).
The exact way to perform the optimization depends on the chosen function family. In this work we will use functions that cannot be optimized (efficiently) with closed form solution, but that are differentiable and we will use their derivatives for the learning process. In the following two optimization schemes are presented. They are used in Chapter IV and III respectively.
We note that Machine Learning setups typically contain so-called hyper-parameter that are not directly optimized using the presented approaches, but are set by using experience, heuristics or schemes like cross-validation (Bishop, 2006).

Gradient descent Given a derivative with respect to the parameters of a model:

$$\frac{\partial l(f(x;\Theta),y)}{\partial \Theta} \tag{II.8}$$

gradient-based optimization builds upon the fact that the gradient points into a direction in parameter space that increases the objective function. The informal idea is to take one step into the opposite direction to find better parameters — and result in a smaller objective value:

$$\Theta_{new} = \Theta - \gamma \frac{1}{n} \sum_{i=0}^{n} \frac{\partial l(f(x_i;\Theta),y_i)}{\partial \Theta} \tag{II.9}$$

with the hyper-parameter learning rate $\gamma \in \mathbb{R}_+$ controlling the relative step size of the update. Ideally this is done repeatedly until a minimum is reached, in which case the gradient is equal to zero. In contrast to gradient descent which relies on the gradient over the whole training set (Equation II.9), stochastic gradient descent (SGD) relies on a stochastic sample to compute a noisy estimate of the gradient:

$$\Theta_{new} = \Theta - \gamma \frac{\partial l(f(x_i;\Theta),y_i)}{\partial \Theta} \tag{II.10}$$

where the sample (x_i, y_i) is drawn randomly from the training set. Given a limited computation or time budget this allows for a increased number of updates of the model parameters and can speed up the training significantly. It is proven that given a set of conditions — such as lowering the learning rate during training — SGD converges to the same solution as gradient descent (Bottou, 2010, 2012).
In practice a so-called mini-batch of samples is used instead of a single one to reduce the influence of the stochastic noise. A complementary technique to counter the noise is to use a running average of the gradient direction, called SGD with momentum (Polyak, 1964):

$$\begin{aligned} M_{new} &= \alpha M - \gamma \frac{\partial l(f(x_i;\Theta),y_i)}{\partial \Theta} \\ \Theta_{new} &= \Theta + M_{new} \end{aligned} \tag{II.11}$$

In Chapter IV we use an advanced version of this scheme named ADAM (Kingma and Adam, 2015) that further adapts the learning rate to the noise during the training. Unfortunately, the optimization for complex functions can lead to challenging error landscapes, e.g., they can have discontinuities and many local minima. It is matter of active research how neural networks can be optimized most efficiently (E.g., Reddi et al., 2018).

Lagrangian optimization Some optimization problems like support vector machines can be formulated by means of (in-)equality constraints. For instance a constraint could be that a data sample should not be miss-classified. A well-known solution to these constrained optimization problems is the method of Lagrange multipliers (Borwein and Lewis, 2010) and its extension for inequalities by Karush (1939) and Kuhn and Tucker (1951). In more detail, we assume an optimization problem in the so-called primal form:

$$\min_x f(x) \quad \text{subject to } g(x) \leq 0 \tag{II.12}$$

where we maximize the function f that is subject to a (number of inequality) constraints g. Then the basic idea is to introduce a Lagrangian multiplier λ:

$$L(x, \alpha) = f(x) + \lambda g(x) \tag{II.13}$$

and to formulate the optimization problem in this way:

$$\min_x \max_\lambda L(x, \alpha) \tag{II.14}$$

where the influence of the Lagrangian multiplier only cancels if the constraint holds. There are a number of conditions that need to hold that both problems lead to the same solution (Borwein and Lewis, 2010). We assume they hold and reformulate Equation II.14 to:

$$\max_\lambda \min_x L(x, \alpha) \tag{II.15}$$

and — assuming we can solve x in terms of λ — we can formulate the so-called dual of the problem:

$$\max_\lambda L_d(\alpha) \tag{II.16}$$

which can be solved using optimization approaches like gradient descent. In a convex setting the difference between the minimum of the primal problem and the maximum of the dual problem is called duality gap. If strong duality holds both solutions are the same and the duality gap is 0. For more details we refer to Borwein and Lewis (2010).

1.2 Algorithms

Support Vector Machines, kernel methods and neural networks are three central algorithms of Machine Learning. They describe ways to represent data and model

parameters, and how to optimize them: Support Vector Machines are a margin-based approach to create classification machines, Kernel methods are feature transformations — based on a similarity metric — that can be used in a subsequent classifier, and neural networks are both — feature transformers and classifiers.

1.2.1 Support Vector Machines

Recall our binary classification setup from Section 1.1.1 and assume that the data points of the two classes can be separated linearly, then the idea of Support Vector Machines (Cortes and Vapnik, 1995, SVMs) is to find a hyper-plane that separates both classes. There might exist many hyper-planes that are able to do so and SVM search for the one with largest margin to the closest data points of the respective classes. The intuition is that future, unseen points are likely to be close to existing ones and that a hype-plane most distant to them will therefore generalize best.
In more detail, given the function $f(x) = \langle x, w\rangle + b$ with the parameters $w \in \mathbb{R}^d$ and $b \in \mathbb{R}$ the hyper-plane is given by $f(x) = 0$ and its so-called canonical form can be ensured by following conditions:

$$y_i(\langle x_i, w\rangle + b) \geq 1 \quad \forall i = 1, \ldots, n. \tag{II.17}$$

In this case the margin is given by $\frac{2}{\|w\|}$ and is the smallest distance between any data point and the hyper-plane, i.e., no data point lies inside this margin of the hyper-plane. Thus it can be regarded as confidence of the solution and SVMs try to maximize it:

$$\begin{aligned} \min_{w,b} \quad & \frac{1}{2}\|w\|^2 \\ \text{s.t.} \quad & y_i(\langle x_i, w\rangle + b) \geq 1 \quad \forall i = 1, \ldots, n \end{aligned} \tag{II.18}$$

In practice many classification problems are not strictly, linearly separable — e.g., due to noise or outliers — and Cortes and Vapnik (1995) propose to relax the hard margin constraint with slack variables ξ_i:

$$y_i(\langle x_i, w\rangle + b) \geq 1 - \xi_i \quad \forall i = 1, \ldots, n, \quad \xi_i \geq 0 \tag{II.19}$$

leading to following optimization problem:

$$\begin{aligned} \min_{w,b,\xi} \quad & \frac{1}{2}\|w\|^2 + C\sum_{i=0}^{n} \xi_i \\ \text{s.t.} \quad & y_i(\langle x_i, w\rangle + b) \geq 1 - \xi_i \qquad \forall i = 1, \ldots, n \\ & \xi_i \geq 0 \qquad \forall i = 1, \ldots, n \end{aligned} \tag{II.20}$$

where C is a regularization parameter controlling the trust in the data and thus the capacity of the function.

This constrained optimization problem can be solved using Lagrange multipliers (see Section 1.1.2) and the resulting dual formulation reads as follows:

$$\begin{aligned} \max_{\alpha} \quad & L_d(\alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \langle x_i, x_j \rangle \\ \text{s.t.} \quad & \sum_{i=1}^{n} \alpha_i y_i = 0 \\ & 0 \leq \alpha_i \leq C \quad \forall i = 1, \ldots, n \end{aligned} \tag{II.21}$$

where the initial parameters w are represented as linear combinations of the so-called support vectors — the data points with $\alpha_i \neq 0$:

$$w = \sum_{i=1}^{n} \alpha_i y_i x_i \tag{II.22}$$

For the derivation of the dual problem, how to recover the bias and further details we refer to Schölkopf and Smola (2002).

1.2.2 Kernel machines

While the presented SVM and other linear models are an effective tool to perform classification, their successful application depends on the linear separability of the input data. Linear separability is often not given and one approach to tackle such problems is to define a feature function $\phi(x)$ that projects the data into a space in which classes are linearly separable. Many such mappings project into high dimensional feature space where it is easier to find discriminative features, but the large number of dimensions can pose a burdensome computational complexity. Kernel methods are a well-known technique to avoid an expansion in such a space by using the so-called kernel trick (Müller et al., 1997). This is done by defining a similarity metric called kernel that is based on a feature function and can be used in methods that solely rely on the similarity of features — such as SVMs that can be expressed with the dot product of two data points $\langle x_i, x_j \rangle$.
A kernel is defined as follows (Vert et al., 2004):

Definition 1 ***(Kernel)*** *A function $k : \mathcal{X} \times \mathcal{X} \mapsto \mathbb{R}$ is called a positive definite kernel iff it is symmetric, that is, $k(x, x') = k(x', x)$ for any two objects $x, x' \in \mathcal{X}$, and positive definite, that is, $\sum_{i=1}^{n} \sum_{j=1}^{n} c_i c_j k(x_i, x_j) \geq 0$ for any $n > 0$, any choice of n objects $x_1, \ldots, x_n \in \mathcal{X}$, and any choice of real numbers $c_1, \ldots, c_n \in \mathbb{R}$.*

then relationship between kernel and feature function is given as

Theorem 2 *For any kernel k on an space $\mathcal{X}$, there exists a Hilbert space $\mathcal{F}$ and a mapping $\phi : \mathcal{X} \mapsto \mathbb{F}$ such that*

$$k(x, x') = \langle \phi(x), \phi(x') \rangle$$

for any $x, x' \in \mathcal{X}$ where $\langle u, b \rangle$ represents the dot product in the Hilbert space between any two points $u, b \in \mathcal{F}$.

Well-known kernels are the linear kernel $k(x, x') = \langle x, x' \rangle$, the polynomial kernel $k(x, x') = (\langle x, x' \rangle + c)^d$, and the Gaussian or radial basis function kernel $k(x, x') = e^{-\frac{\|w\|^2}{2\sigma^2}}$.
A given kernel can be used to replace the dot product in the dual formulation of a SVM (Equation II.20) to classify in the induced feature space. For more details on kernel methods we refer to Schölkopf and Smola (2002).

1.2.3 Neural networks

We will introduce first the basic building block of neural networks and then make the leap to modern computer vision models. Inspired by the human brain structure, neural networks consist of connected neurons (McCulloch and Pitts, 1943). A neuron is composed of weighted, incoming connections and an activation mechanism to signal its activity. Mathematically this is expressed as a dot product of incoming connections $x \in \mathbb{R}^d$ and their weights $w \in \mathbb{R}^d$, a bias $b \in \mathbb{R}$, and a scalar, activation function $a : \mathbb{R} \mapsto \mathbb{R}$:

$$h(x) = a\left(\sum_{i=1}^{d} x_i w_i + b\right) \tag{II.23}$$

An example for an activation function is the sigmoid function $a(x) = 1/(1 + e^{-x})$ which transforms the input to a "on/off" response.
The prevalent structuring of these connections is by grouping neurons in so-called layers. In the simplest form each neuron is connected to all neurons from the previous layer — no incoming connections within or to the upper layer exist. Using matrix algebra this can be expressed as follows and is known as a fully-connected layer with a weight matrix $W_i \in \mathbb{R}^{d_i \times d_{i+1}}$ and a bias $b_i \in \mathbb{R}^{d_{i+1}}$:

$$h_i(x) = a\left(xW_i + b_i\right) \tag{II.24}$$

Exemplary a 2-layer neural network can be expressed as follows with $d_0 = d$, $d_3 = 1$ and d_1 as well as d_2 as hyper-parameters:

$$f(x) = h_2(h_1(h_0(x))) \tag{II.25}$$

Relating this to Section 1.2.2, a neural network can be seen as complex feature function $\phi(x)$:

$$f(x) = \langle w_2, \phi(x) \rangle + b_2 x \quad \text{with} \quad \phi(x) = h_1(h_0(x)) \tag{II.26}$$

The optimization of the multi-layered neural networks is typically done with gradient descent and relies on partial derivatives and gradient back propagation (Rumelhart et al., 1986). This is based on the idea of the chain rule:

$$\frac{\partial l(f(x), y)}{\partial x} = \frac{\partial l(f(x), y)}{\partial h_2(x)} \frac{\partial h_2(x)}{\partial h_1(x)} \frac{\partial h_1(x)}{\partial h_0(x)} \frac{\partial h_0(x)}{\partial x} \tag{II.27}$$

and, building upon, the gradients for the respective parameters are as follows:

$$\frac{\partial l(f(x), y)}{\partial W_i} = \frac{\partial l(f(x), y)}{\partial h_i(x)} \frac{\partial h_i(x)}{\partial W_i} \tag{II.28}$$

The gradients of the bias b_i are given accordingly. This structure allows for an efficient computation by reusing the results iteratively. We note that it can be challenging to train deep neural networks and there are many conventions and tricks used in practice (E.g., LeCun et al., 2012).

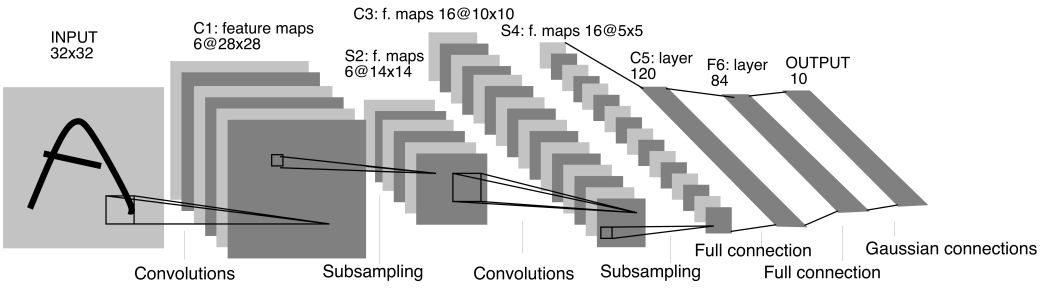

Fig. II.1: Convolutional neural network. *The structure of a convolutional neural network as originally proposed by LeCun et al. (1998a). It consists of convolutional layers with small filters that are applied in a grid-like fashion to the image. Max-pooling layers are applied to sub-sample the image representation and the final part is designed to classify given the extracted features. This figure is from LeCun et al. (1998a).*

Starting with this basic setup, the field of neural network has evolved into a complex domain and there are many different ingredients that are used to power deep learning — a synonym for deep neural networks. In this work we use convolutional neural networks for computer vision applications. For these networks the input image is represented as 2-dimensional map and this structure is retained until the last (classification) part of the network. The most important building blocks of convolutional neural networks are convolutional and max-pooling layers to extract shift and translation invariant features, rectified linear units (ReLUs) as fast and expressive activation functions, Batch Normalization and residual connections to condition the neural network, and finally a softmax layers to create a probability distribution over the class activations. In more detail:

Convolutional layers are — as the basic fully-connected layer — inspired by the biological processes, namely by receptive fields (LeCun et al., 1998a). A convolutional filter takes as input a fraction of the input image and the same filter is applied in a grid-like fashion on the whole image representation. Furthermore, in each layer a number of different filters are applied. Figure II.1 shows this exemplarily. The weight-sharing in convolutional layers makes them shift-invariant, i.e., the same filter gets applied on to many positions.

Pooling layers are used to reduce the "resolution" of the image representation. This is usually done by taking the maximum of a fraction of the input image — similar to the receptive field of a convolution filter. In Figure II.1 pooling is denoted as subsampling. Pooling makes the network to some extent translation invariant.

ReLU-activations were introduced by Nair and Hinton (2010) and are a piece-wise linear function that is $a(x) = x$ for $x > 0$ and 0 otherwise. This function is computationally

very efficient and has been widely adopted.
Batch normalization similar how LeCun et al. (2012) pointed out that neural networks need to be initialized in the right way, Ioffe and Szegedy (2015) propose to introduce layers that normalize the forward propagated values to 0 mean and standard deviation *during* the training. Furthermore, the layer learns a shift and multiplicative factor for the data, after normalizing it. The authors show that this enables more robust and faster training.
Residual connections were proposed to overcome a practical depth limitation of deep neural networks. He et al. (2016) showed that despite the fact that a deeper neural network can represent a shallower one and should be more expressive, in practice the training of neural networks was not able to converge to better solutions — from a certain depth onwards the results actually got worse. As solution He et al. (2016) proposed residual layers by using so-called short cuts $h(x) = x + g(x)$, where the first component is an identity mapping and makes the retention of information over layers more likely. This allows to train much deeper networks.
Softmax is multi-class prediction function to normalize the outputs of a network to a probability distribution. The formula reads as follows: $f(x) = \frac{e^z}{\sum_i e_i^z}$ with $z \in \mathbb{R}^{\downarrow}$ the activation for the respective classes and e^z is applied element-wise. As surrogate multi-class loss, typically $log(f(x))$ is used.
Further details are beyond the scope of this thesis and we refer to the dedicated literature (Bishop, 1995; Montavon et al., 2012; Goodfellow et al., 2016).

1.3 Analyzing predictions

Neural networks can be very complex models and there exists a natural desire to understand their workings. Several approaches are proposed in literature and in this work we focus on the direction called prediction analysis. Prediction analysis tries to "explain" a single prediction of a classifier — in our specific case of a neural network. In more detail, we will describe algorithms that, given a neural network $f(x) : \mathbb{R}^d \mapsto \mathbb{R}$, create an analysis $e : \mathbb{R}^d \mapsto \mathbb{R}^d$ with the same dimensionality as the initial input.
How such an analysis can be interpreted depends on the applied algorithm. To give a first intuition on possible interpretations we will review the simplest neural network, namely a linear model, first. This allows us to establish a number of concepts, before we will present the relevant prediction analysis algorithms for deep neural networks.

1.3.1 Linear model

To understand deep neural networks it is helpful to examine their basic building blocks — linear models — first. The toy example from Kindermans et al. (2018) shown in Figure II.2 is based on the data x:

$$\begin{aligned} x = s + d \qquad & s = a_s y, && \text{with } a_s = (1,0)^T, && y \in [-1,1] \\ & d = a_d \epsilon, && \text{with } a_d = (1,1)^T, && \epsilon \in \mathcal{N}(\mu, \sigma^2) \end{aligned} \tag{II.29}$$

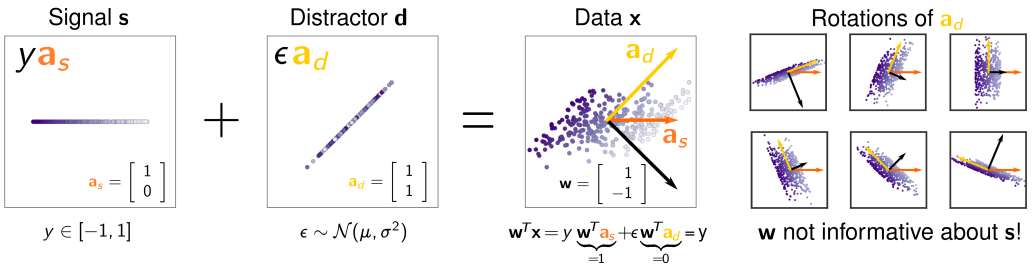

Fig. II.2: Interpretation of a linear model. *This figure shows an interpretation for the prediction of a linear model (Haufe et al., 2014). The data is generated by $x = a_s y + a_d \epsilon$ and is color coded w.r.t. to the output of the learned model $\hat{y} = w^T x$. The influence of the distractor a_d is shown on the right hand side: the weight vector tries to filter the "noise" and accordingly adapts to it — as a result it is not informative about the signal direction. This figure is from Kindermans et al. (2018). Best viewed in digital and color.*

A linear regression model w is trained to extract $\hat{y} = w^T x$ from the data. By construction s is the *signal* of the data — and contains the information about y. The *distractor* d (Haufe et al., 2014) obfuscates the signal and makes the regression task more difficult. In order to extract y, our model needs to filter d from the data — therefore weight vectors are also called *filters*. In our example $w = (1, -1)^T$ fulfills this task.
With this example in mind, we note:

- In general the optimal weight w does *not* align with the signal direction a_s.
  - It rather tries to cancel the contribution of the distractor — which is given when the weight vector w is orthogonal to the distractor direction $w^T d = 0$.
  - Accordingly, when the direction of the distractor changes the weight vector changes. See right hand side in Figure II.2.
- A change of the signal direction a_s can be compensated by the weight vector w by a change of sign and magnitude such that $w^T a_s = 1$ — but its direction stays constant.

To summarize: (1) The weight vector's direction is determined by the distractor's direction and (2) in general it is not aligned with the signal's direction in the data. Thus the weight vector reveals how the noise from the data is filtered, whereas the signal direction reveals how the signal is generated. For the given example it is possible to recover the signal direction a_s. We refer to Haufe et al. (2014) and Kindermans et al. (2018) for more details.

1.3.2 Neural networks

Based on the interpretation of a linear model Kindermans et al. (2018) propose to categorize prediction analysis algorithms in three categories:

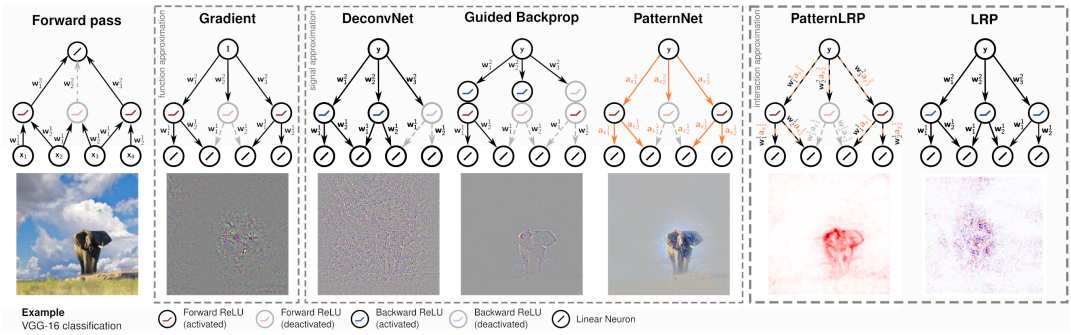

Fig. II.3: Analyzing by back-propagating. *This figure depicts schematically the analysis categorization (function, signal, and interaction approximation) and schematically how a selection of algorithms works. The prediction is done with a VGG16 network (Simonyan and Zisserman, 2014). For a detailed description of the algorithms we refer to the main text. This figure is adapted from Kindermans et al. (2018). Best viewed in digital and color.*

Function approximation: Algorithms that reveal how the function filters the signal from the data.

Signal approximation: Algorithms that show the signal directions in the data — based on the model's filtering.

Interaction approximation: Algorithms that combine both principles to create a focused analysis (see Montavon et al. (2017)).

Exemplarily Figure II.3 shows the three categories.
Compared to a linear model deep neural networks are more complex and a duality with a generative process — as described for a linear model — is not given. This makes the prediction analysis for deep neural networks harder and indeed many different approaches have been proposed. Due to the absence of a generative model or any other ground truth it is unclear which method performs "best". We will now describe different attempts and use the categorization to interpret the algorithms.
The considered prediction analysis algorithms can be split into two major groups. The first group regards the given model as a black box — by only using function and gradient evaluations for the analysis. This group can be applied to arbitrary prediction functions. The second group is linked to neural networks and uses the respective structure of a model to create a tailored analysis procedure. Typically this is done with a backward propagation through the model, similar to the computation of the gradient w.r.t. to the input.

Prediction- and gradient-based algorithms This algorithm family only uses the prediction $f(x)$ or the gradient $\frac{\partial f(x)}{\partial x}$ to create an analysis (Kindermans et al., 2016; Shrikumar et al., 2017; Smilkov et al., 2017; Sundararajan et al., 2017; Zintgraf et al., 2017; Ribeiro et al., 2016; Lundberg and Lee, 2017). We will now present the algorithms used in this thesis.

Gradient or *Saliency map* (Baehrens et al., 2010) shows the sensitivity of the function w.r.t. to the input $e(x) = \frac{\partial f(x)}{\partial x}$ and is the weight vector w for a linear model — thus can be seen as function approximation algorithm.
*Input * gradient* (Shrikumar et al., 2017; Kindermans et al., 2016) is formulated as $e(x) = \frac{\partial f(x)}{\partial x} \odot x$ and can be regarded as interaction approximation. This algorithm is the same as the basic LRP method (Bach et al., 2015) for a number of neural networks (Kindermans et al., 2016).
SmoothGrad (Smilkov et al., 2017) tries to "smooth" the gradient by averaging it over a number m distorted inputs $e(x) = \sum_{i=0}^{m} \frac{\partial f(x+\epsilon)}{\partial x+\epsilon}$ with $\epsilon \in \mathcal{N}(0, \sigma^2)$. It can be seen as a function approximation.
Integrated Gradients (Sundararajan et al., 2017) analyzes a prediction with respect to a reference image. The reference image should not contain any information correlated with the prediction, e.g., a black image. This idea is closely related to Montavon et al. (2017), but is based on the whole function instead of a single neuron. Given a reference image x_{ref} the function reads as follows:

$$e(x) = (x - x_{ref}) \odot \int_{\alpha=0}^{1} \frac{\partial f(x)}{\partial x}\Big|_{x=x_{ref}+\alpha(x-x_{ref})} d\alpha. \tag{II.30}$$

To capture the non-linear structure of a function Integrated Gradients uses an average of the gradients along a path and can be seen as an interaction approximation.
Occlusion (Zeiler and Fergus, 2014) creates an analysis by combining predictions where a part of the image was occluded. In its simplest form the an image is divided into non-overlapping regions and each regions gets assigned the predicted value produced for the perturbed input. Finally, all the values get normalized by subtracting the prediction for the original images. We would regard this as an interaction approximation.
Local interpretable model-agnostic explanations (Ribeiro et al., 2016, LIME) uses a different approach by creating a dataset to learn a linear classifier mimicking the network in a local neighborhood. The resulting weights are indicating the importance of the respective features. For images a common way is to use connected regions with the same color as features which are occluded or not. Again, we would regard this as an interaction approximation.

Propagation-based algorithms The next function group adapts to the structure of the network by performing a back-propagation from the output back to the network's input.
DeconvNet (Zeiler and Fergus, 2014) is depicted in Figure II.3 and tries to approximate the signal. The idea is that in a ReLU-network the signal cannot be negative and thus negative values get blocked in the backward pass by applying a ReLU.
Guided Backprop (Springenberg et al., 2015) is closely related to DeconvNet and in fact can be seen as its extension. Basically, the idea that a negative signal should be blocked on the backward pass is retained and extended by also blocking signals for which the corresponding value in the forward pass was negative — as it was no signal in the first place. This algorithm is also shown in Figure II.3.
Layer-wise relevance propagation (Bach et al., 2015, LRP) defines output value of the prediction as "relevance" R. The relevance gets then back-propagated by using a "rule"

for a each layer. The relevance R_i for the input neuron i for a layer is computed as follows, given the relevance R_j of a number of connected neurons j:

$$R_i = \sum_j \frac{x_i w_{ij}}{\sum_i x_i w_{ij} + b_i} R_j \tag{II.31}$$

and can be rewritten for a linear layer $h(x) = y = xW + b$ as:

$$\begin{aligned} bw\_mapping(x, y, r = bp_y) = x \odot (W^t z) \\ \text{with } z = r \oslash (xW + b) \end{aligned} \tag{II.32}$$

The final analysis is created by using the rule to back-propagate until the input is reached. In literature this rule was denoted as "LRP-Z" and the authors have propose at least two more rules. The first one is called "LRP-epsilon" and introduces a parameter for numerical stability or to filer signals. It reads as follows for a single neuron:

$$R_i = \sum_j \frac{x_i w_{ij}}{\sum_i x_i w_{ij} + b_i + sign(x_i w_{ij})\epsilon} R_j \tag{II.33}$$

Furthermore, the "LRP-alpha-beta" rule weighs positive and negative contributions differently:

$$R_i = \sum_j \left(\alpha \frac{(x_i w_{ij})^+}{\sum_i (x_i w_{ij} + b_i)^+} - \beta \frac{(x_i w_{ij})^-}{\sum_i (x_i w_{ij} + b_i)^-} \right) R_j \tag{II.34}$$

The initial LRP-Z algorithm is (only) equal to input * gradient for networks only consisting of convolutional, fully-connected, max-pooling layers (Kindermans et al., 2016) — therefore it can be seen as interaction approximation and is depicted in Figure II.3.

Deep Taylor Decomposition (Montavon et al., 2017) is, similar to LRP, based on definitions for single neurons. It is theoretically more founded and some LRP rules can be expressed in this framework — giving them a theoretical meaning. The Deep Taylor Decomposition is based on a linear Taylor expansion and follows the same idea as Integrated Gradients, namely finding a so-called rootpoint for which the function evaluates to zero and using the second element of the expansion — the gradient — as analysis. In Deep Taylor this is done for each neuron individually and, given some assumptions and depending on the input value range, different rules are applied. If a neuron has an unbounded input the W^2-rule should be used and reads for a single neuron as follows:

$$R_i = \sum_j \frac{w_{ij}^2}{\sum_i w_{ij}^2} R_j \tag{II.35}$$

If a neuron has only positive input the $Z+$-rule should be used:

$$R_i = \sum_j \frac{x_i w_{ij}^+}{\sum_i x_i w_{ij}^+} R_j \tag{II.36}$$

If the neuron's input is bounded in the range $[a, b]$ the Z_B-rule should be used:

$$R_i = \sum_j \frac{x_i w_{ij}^+ - a w_{ij}^+ - b w_{ij}^-}{\sum_i x_i w_{ij}^+ - a w_{ij}^+ - b w_{ij}^-} R_j \tag{II.37}$$

We note that the Deep Taylor has a range of assumptions, e.g., the output must be positive, and refer to Montavon et al. (2017) for more details. Deep Taylor can be seen as interaction approximation.

PatternNet & PatternAttribution (Kindermans et al., 2018) tries to extend the pattern theory for a linear model (Haufe et al., 2014) to deep neural networks. It is limited to networks similar to VGG16 (Simonyan and Zisserman, 2014). PatternNet attempts to approximate the signal by learning the individual pattern direction of single neurons. These directions are then used to propagate the output signal back to the input. PatternAttribution approximates the interaction by means of the Deep Taylor framework. While the Deep Taylor algorithms search for an adequate rootpoint in the direction of weight vector, PatternAttribution uses the signal direction to do so. The algorithms are showcased in Figure II.3. For more details we refer to Kindermans et al. (2018).

2 Software tools for Machine Learning

In this section we first highlight some challenges to consider when creating efficient Machine Learning software and subsequently describe the programming languages, software libraries and tools for the implementation of the proposed algorithms. We group the approaches in three complimentary parts: The first is dedicated to local implementations, i.e., on a single machine, and the second focuses on means to distribute Machine Learning algorithms — both parts are used to implement a distributed SVM for sparse data in Chapter III. Chapter IV and Chapter V focus on neural networks (related) topics and for the development we employ dedicated deep learning software which is treated in the last part.

The application in Chapter III posed three challenges that are representative for Machine Learning workloads. Compared to other sciences Machine Learning can be very data intensive: the more data an algorithm can process for training the better the resulting models (E.g., Bottou, 2012). This is one of the reasons why graphical processing units (GPUs) are very popular in Machine Learning. They are tailored to perform similar operations on an array of data — using the single instruction - multiple data principle (Flynn, 1972). In general this requires efficient data loading and processing.

Many data science tasks like, e.g., data pre-processing, are embarrassingly parallel, Machine Learning is typically not. The training of data models can be inherently complicated and is characterized by complex data dependencies and iterative processes. In order to still take advantage of parallel computation and data pipelines, communication with low latency is a key to scale these algorithms. This is is one requirement for tools to distribute such computations.

Finally, we note that Machine Learning applications can be divided into programs processing sparse or dense data. In this context sparse data and, typically, sparse models are given when a large fraction of their values is exactly zero. While software for dense data can be used to process sparse data, this is usually rather inefficient and therefore dedicated software for sparse data is required. One big implication of sparse data is a irregular memory access pattern that — recalling the Machine Learning is rather memory bound — has a major influence on the performance. The SVMs in Chapter III work on sparse data and the algorithms in Chapter IV and Chapter V use dense data.

2.1 Local computing

We define as local computing the computation on a single machine, which can entail the usage of one or several CPUs or of one or several GPUs. Programs handling dense data are typically expressed in linear algebra routines like matrix-vector-multiplications and the usage of standard software like Numpy (Van Der Walt et al., 2011) or deep learning frameworks (E.g., Abadi et al., 2016b; Paszke et al., 2017) allows already for implementations with good performance. These software tools in turn rely on highly efficient backends such as BLAS (Blackford et al., 2002) for CPUs or CUDA (Nickolls et al., 2008) for GPUs. We highlight that this is possible, because linear algebra and tensor operations have proven to be useful interfaces and the algebra routines typically form the largest fraction of the program.

The case with sparse data requires more involvement. Many algorithms with sparse data are specially tailored for this type of representation. They consist of many small operations on data and usually require many loops. In this scenario high level languages such as Python are not well suited due to the interpretation overhead and low level and compiled programming languages like C are needed in order to obtain efficient code.

There is no general recipe on how to implement sparse algorithms best, but one important aspect is to be aware of the memory and cache structures of CPUs. There is a chain of memory structures that starts with the main memory as largest, but slowest, and several caches that are becoming smaller‘, but faster, the closer they are located to the processing core. Memory locations that are accessed frequently will be kept in caches close to the CPU core and can be retrieved more quickly. Furthermore, when accessing a single memory location a whole “memory page” — a continuous larger region around the location — will be loaded into the caches. Accordingly it is important to design the memory access pattern of a program to access similar regions together and reuse variables as long as they are in the cache. For applications with (irregular) sparse data this is a key for making an implementation efficient. For more information on access patterns we refer to, e.g., Kreutzer et al. (2014).

An alternative to implement an algorithm in a low level language is to program non-performance critical section in, e.g., Python, and using C routines for performance critical sections. In this gap, Cython (Behnel et al., 2011) allows to program with low level routines using Python syntax and many other benefits. It is able to run as fast as a conventional C program, but needs to be compiled before the execution.

In Chapter III we distribute the computation over several processes where each one works on a dedicated core. There are several ways to do so and a well-known one is OpenMP (Dagum and Enon, 1998). This library offers a slim and portable interface, which helps to avoid low level interactions with the respective operating system. It works as follows. The same program is called for a number of times and each parallel process gets a unique id. Based on this id the processes can perform different tasks and communicate with each other. Special group communication routines are available such as broadcasting messages and — for Machine Learning purposes useful — reducing operations like performing a sum of values over all processes. A big advantage of multi-process computation is that the processes or alternatively threads can share the same memory, thus there is no need to communicate large memory blocks like , e.g., vectors, as one can exchange pointers or locks instead. The drawback of shared memory is the need for active memory management between the processes to ensure the consistency of the values. The implementation of parallel or distributed programs requires attention on a number of further issues like, e.g., deadlocks. We refer to the according literature for more details (Andrews, 2000).
In Chapter III we use Python, Cython, and OpenMP to realize a distributed implementation with efficient memory access patterns. Leveraging the power of GPUs is very hard for applications with sparse data due to the irregular memory pattern. Additionally, many algorithms, which are tailored to sparse data, are sequential in nature and not well suited for the highly parallel nature of such processors. For these reasons we did not pursue this direction.

2.2 Distributed computing

Distributed computing and parallel computing are similar in the sense that several processes run in parallel. Their key distinction is the ability to share memory. While in parallel computing (on one machine) processes *can* share the same memory, distributed (on different machines) processes cannot share the same memory[2]. This changes the programming model and makes the communication in distributed systems more costly. In Machine Learning, where many, rapid iterations and large model parameters are common, it can be a challenge to successfully use distributed computations (E.g., Boden et al., 2018). The main reasons are the higher latency and the cost to communicate many or big messages.
There a several options for distribute computation and we discuss three types. The first is MPI (Gropp et al., 1996), the second are parameter server (E.g., Ho et al., 2013; Li et al., 2014), and the last are data flow systems (E.g. Dean and Ghemawat, 2008; Carbone et al., 2015; Zaharia et al., 2016).
MPI (Gropp et al., 1996) is closely related to OpenMP (Dagum and Enon, 1998). They share similar interfaces and have the same programming setup, namely that all processes are based on the same program, but are executed with different identifiers. Due to this common interface and its low overhead it was our choice in Chapter III.
For (stochastic) gradient descent based training it is common to use distributed parameter server setups (E.g., Ho et al., 2013; Li et al., 2014). The idea of this

[2] An exception are emerging systems that enable remote direct memory access (RDMA).

approach is that a set of servers saves and manages the model parameters, and a set of workers retrieves the parameters to train them in a predefined setting. One important factor for these systems to perform well is the stochastic component of SGD allowing for a relaxed parameter update scheme (E.g., Ho et al., 2013) to for instance to ease the "last reducer" problem. This is not given for the SVM algorithms in Chapter III and therefore we do not use this approach.
In data management data flow systems such as MapReduce (Dean and Ghemawat, 2008), Flink (Carbone et al., 2015) and Spark (Zaharia et al., 2016) are key enabler for big data applications. They are designed for use cases with (1) large amounts of data, (2) unknown data flow patterns, e.g., (the amount of) data that flows in join operation etc. depends on the processed data and is not known a priori, (3) few to no iterations of the directed acyclic computational graph, and (4) moderate latency requirements, with the notable exception of stream processing engines like, e.g., Flink (Carbone et al., 2015). This stands in contrast to the training workload of Machine Learning models with the properties: (1) often moderate amounts of data that can be stored on a single machine, (2) known data flow patterns, e.g., the batch-size typically determines (the amount of) data to be communicated and induces fixed communication patterns, (3) acyclic directed computational graphs that are *repeated* for a large number of times, and (4) low latency requirements. For instance Boden et al. (2018) examines this issue. Our application in Chapter III is exemplary for a Machine Learning workload and accordingly we decided against data flow systems to distribute our computation. We note that data flow systems are useful for many other tasks in the a data pipeline like pre-processing or for deploying data models at the inference time.

2.3 Deep learning frameworks

The success of deep learning has triggered the development for a number of dedicated systems for neural networks (E.g., Bergstra et al., 2010; Abadi et al., 2016b; Paszke et al., 2017). Neural networks can be described as computational graphs relying on tensor data types and operations. This uniform pattern is the basis for deep learning frameworks. As these software libraries are specialized for the training and application of (deep) neural networks, algorithms for sparse data can so far not be realized efficiently with these systems.
Modern deep learning systems have a number of features: (1) high level descriptions of operations using tensor algebra, (2) the ability to run the computational graph on heterogeneous hardware, (3) specialized backends to take advantage of GPUs as accelerators, (4) integrated parameter servers to distribute computations. For a more detailed description we refer to, e.g., Abadi et al. (2016b)
There are a number of libraries that try to standardize deep learning workflows and one of them is Keras (Chollet, 2015) with the aim to make deep learning accessible to a larger audience. This has the advantage that the user can focus on task like the development of a new neural network architecture. On the other hand, it can require some workarounds to implement tasks with different usage patterns.
The applications in Chapter IV and Chapter V are related to deep learning and we use

Keras and TensorFlow (Abadi et al., 2016b) for our implementations. In Chapter V we use Keras as basis to make our code accessible, yet it required us to implement several workaround to make our advanced graph processing possible and efficient.

Chapter III

Distributed optimization of multi-class SVMs

Classification tasks with a large number of categories are among the most common and important problem settings in Machine Learning. For tackling such problems as for instance the classification of text into categories, the recommendation of items to users, or matching words to audio in natural language processing, support vector machines (SVMs) have proven to be very effective. These margin-based approaches can be divided into two major groups, one-vs.-rest and all-in-one SVMs, where the former prevails due to its simpler nature. Yet recent research showed for tasks with few classes all-in-one SVMs consistently perform better and leads us to the main question of this chapter: *Can all-in-one SVMs training be distributed and scaled to problems with a large amount of classes and, if so, do all-in-one SVMs still perform favorably compared to one-vs.-rest SVMs?* To approach this question we develop decomposable optimization routines for two well-known, all-in-one SVMs formulations. The decomposition allows us to distribute the computation and memory efforts evenly among different computing instances. As a result we are able to conduct an empirical evaluation with sparse and high dimensional text-data. The comparison confirms that all-in-one SVMs can indeed outperform one-vs.-rest SVMs and, furthermore, lead to significantly sparser models. Most of the work in this chapter was previously published in Alber et al. (2016, 2017b).

1 Introduction

Modern data analysis often requires computation with a large number of classes. Consider the following real world examples: (1) A crawler continuously monitors the internet for new webpages, which should be categorized. (2) Given data from an online biomedical, bibliographic database, the task is to index it for quick access for

clinicians. (3) A company collects data from an online feed of photographs and would like to classify image categories. (4) For newly added articles to an online encyclopedia, the respective article categories should be predicted. (5) Given a huge collection of ads, one desires to built a classifier from this data. The mentioned problems — taken from varying application domains ranging from the sciences to technology — involve a large number of classes, typically at least in the thousands.
Classification problems are one of the key applications of Machine Learning and one of the most powerful and most used methods to tackle them are multi-class support vector machines (MC-SVMs, Vapnik, 1995). This family of algorithms is well-studied (Schölkopf and Smola, 2002) and can be divided into two major groups:

1. One-vs.-one (OVO) and one-vs.-rest (OVR) MC-SVMs decompose the problem into multiple binary subproblems that are subsequently aggregated (Vapnik, 1995; Rifkin and Klautau, 2004). Training can be parallelized in a straight forward way.

2. *All-in-one* MC-SVMs extend the concept of the margin to multiple classes. Because there is no unique extension of the margin concept, multiple all-in-one MC-SVMs have been proposed, including the ones by Weston and Watkins (WW, 1999), Lee, Lin, and Wahba (LLW, 2004), and Crammer and Singer (CS, 2002). See Rifkin and Klautau (2004), Allwein et al. (2001), Hsu and Lin (2002), Hill and Doucet (2007), Liu (2007), Guermeur (2007), and Doğan et al. (2016) for conceptual and empirical comparisons.

Recently, Doğan et al. (2016) compared the various all-in-one MC-SVM variants on rather moderately sized datasets and showed advantages of all-in-one MC-SVMs over OVR MC-SVM, but — so far — slow training time has prohibited further comparisons on data involving a large number of classes. Naturally, this leads to the following two questions: (1) Is it possible to scale exact, all-in-one MC-SVMs to problems with a large number of classes, and (2) do they still perform favorably compared to OVR MC-SVMs?
We approach these questions by developing distributed algorithms where up to $\mathcal{O}(\mathcal{C})$ nodes solve Weston and Watkins (WW, 1999) and Lee, Lin, and Wahba (LLW, 2004) in parallel, dividing model and computation evenly on different computing instances. The resulting solvers are compared to a state-of-the-art OVR solution. Our choice of MC-SVMs is motivated by the comparison of Doğan et al. (2016).
The algorithm proposed for WW draws inspiration from graph theory, namely the solution to the 1-factorization problem of a graph (Bondy and Murty, 1976). On the other hand, we parallelize LLW training by introducing an auxiliary variable to the dual problem that decouples the objective into a sum over $\mathcal{C}$ many independent subproblems.
We provide both multi-core and distributed implementations of the proposed algorithms. We report an empirical comparison of the proposed solvers with the one-vs.-rest implementation by LIBLINEAR (Fan et al., 2008) on text classification data taken from the LSHTC corpus (Partalas et al., 2015). The datasets pose a challenge for training of linear models such as SVMs due to their high-dimensional input *and* out-

put domain, which results in potentially very large models. Our approach counteracts this by distributing the model evenly among different nodes.
The main contributions of this work are the following:

- We propose the first distributed, exact solver for WW and LLW.
- We provide both, multi-core and truly distributed, implementations of the solvers.
- We give the first comparison of WW, LLW, and OVR on the DMOZ data from the LSHTC '10–'12 corpora using the full feature resolution.

The chapter is structured as follows. In the next two Sections, 2 and 3, we discuss related work as well as the problem setting and preliminaries, respectively. Building upon, we present in Section 4 the proposed distributed algorithms for LLW and WW and analyze their convergence properties empirically in Section 5. The last part consists of a critical discussion in Section 6 and concluding words in Section 7.

2 Related work

First we introduce related work that motivates our research question, before confining our approach from other related work.

On one-vs.-rest and all-in-one SVMs First and foremost, Doğan et al. (2016) show that all-in-one SVM formulations outperform the one-vs.-rest formulation. Additionally, they suggest WW as the best choice of all-in-one SVMs and also indicate that LLW is a good fit for very high-dimensional feature spaces. This analysis is conducted on small scale datasets and rises the question, if this result translates to setups of larger scale. Furthermore, recent work (Babbar and Schölkopf, 2017, 2018) suggests that *approximated* all-in-one SVMs do not match the performance of OVR methods, which motivates our choice to work with *exact* optimization procedures. We will contribute more to this discussion in Section 6.

Distributed algorithms for multi-class SVMs Most approaches to parallelization of MC-SVM training are based on OVO or OVR (E.g., Fan et al., 2008; Babbar et al., 2016; Babbar and Schölkopf, 2018), including a number of approaches that attempt to learn a hierarchy of labels (E.g., Bengio et al., 2010; Deng et al., 2011; Gao and Koller, 2011; Choromanska and Langford, 2015; Zhou et al., 2011; Gopal and Yang, 2013b; Madzarov et al., 2009) or train ensembles of SVMs on individual subsets of the data (E.g., Govada et al., 2015b,a; Lodi et al., 2010). We benchmark our results against LIBLINEAR's (Fan et al., 2008) efficient one-vs.-rest solver.
There is a line of research on parallelizing stochastic gradient (SGD) based training of MC-SVMs over multiple computers (Gupta et al., 2014; Do, 2014). SGD builds on iteratively approximating the loss term by one that is based on a subset of the data, e.g., a single sample or a mini-batch. In contrast, batch solvers — such as the ones proposed in the present chapter — are based on the full sample. While there is a long

ongoing discussion whether the batch or the SGD approach is superior, the common opinion is that SGD has its advantages in the early phase of the optimization, while classical batch solvers shine in the later phase due to less noisy gradients (Bottou, 2012). In this sense, the two approaches are complementary and could also be combined.
Other work on distributed algorithms distinguishes from our approach that it requires each computation node to hold the whole model in memory as well as to *communicate* it to each node after an update. In our problem setting, high-dimensional sparse input and output data, the model size can be very large and prohibits this approach from scaling to problems with very large label spaces. In contrast our algorithms store the model in distributed manner on different nodes and communicate only a global class mean (LLW) or a fraction of the model between node-pairs after updates (WW).
The related work closest to the present one is by Han and Berg (2012). They build on the alternating direction method of multipliers (ADMM, Boyd et al., 2011) to break the Crammer and Singer optimization problem (Crammer and Singer, 2002) into smaller parts, which can be solved individually on different computers. In contrast to our technique, the optimization problem is parallelized over the samples, not the variables and thus classes, and requires, as pointed out, each computation instance to hold the whole model in memory as well as to communicate it to all nodes after an update. Note also, it is unclear at this point whether the approach of Han and Berg (2012) could be adapted to LLW and WW, which are the objects of study in the present chapter.
Other work on binary linear SVMs is a distributed box-constrained quadratic optimization algorithm by Lee and Roth (2015). It works by distributing the kernel matrix over all available computational nodes, hence again distributing the data and exhibiting the same scaling issue as noted above. Some other competitive families of distributed algorithms solve the primal objective directly using distributed coordinate ascent (Mahajan et al., 2018). This approach is similar to Lee and Roth (2015), however requires a continuously differentiable loss function.
Traditional, but no longer competitive, approaches on distributed binary SVMs include Zhu et al. (2008), Forero et al. (2010), and Pechyony et al. (2011). We cite these here for completeness. Beyond SVMs there is a large body of work on distributed multi-class, e.g., Agarwal et al. (2014) and Gopal and Yang (2013a), and multi-label learning algorithms, e.g., Prabhu and Varma (2014), which will be discussed in more detail in Section 6.

Single-core solvers for multi-class SVMs Even though LLW and WW exhibit advantageous theoretical properties (Lee et al., 2004; Weston and Watkins, 1999; Doğan et al., 2016), to our knowledge no extensive empirical studies on large datasets exist in the literature. A library that includes almost all proposed MC-SVM formulations is the Shark ML library (Igel et al., 2008). Shark features single-core solvers for LLW, WW and CS based on sequential, dual block coordinate ascent, which updates all dual variables of the same datapoints at the same time. While in practice Shark is not suitable for our large scale problems, we include the classification results of this solver in our comparision in order to show that our implementations are exact.
The two most popular MC-SVM libraries, SVM$^{\text{multiclass}}$ (Joachims et al., 2009) and

LIBLINEAR (Fan et al., 2008), implement only the Crammer and Singer (2002) formulation. Out of the two, we restrict our comparison to LIBLINEAR, which we found to outperform SVM$^{\text{multiclass}}$.
Another interesting idea is pursued by Jenssen et al. (2012). They developed a scatter-based MC-SVM using class prototypes. Unfortunately, their algorithm and implementation is based on kernel matrices which is prohibitive given our data with many samples and classes. Therefore we decided to not compare to this work.
Last but not least, it needs to be mentioned that large-scale multi-class classification problems are in practice often solved using Vowpal Wabbit (Agarwal et al., 2014), which is an increasingly popular on-line classification tool. Vowpal Wabbit is, however, a fundamentally inexact solver and is thus excluded from our comparison.

3 All-in-one SVMs

We consider the following problem from Section 1.1.1 where we are given some data $(x_1, y_1), \ldots, (x_n, y_n)$ with $x_i \in \mathbb{R}^d$ and $y_i \in \{1, ..., \mathcal{C}\}$. Each class has in average $\bar{n}$ samples and the largest number of samples for a single class is n_{max}. Based on this, we are predicting a class by using the model

$$\hat{y}(x) := \underset{c}{\operatorname{argmax}}\, w_c^T x, \tag{III.1}$$

where $W = (w_1, .., w_{\mathcal{C}}) \in \mathbb{R}^{d \times \mathcal{C}}$ are the parameters. The aim is to efficiently find good parameters in order to predict well on unseen data using Equation III.1.
To address this problem setting, a number of generalizations of the binary SVM (Cortes and Vapnik, 1995) have been proposed. We are specifically studying the two formulations proposed by Lee, Lin, and Wahba (LLW, 2004) and Weston and Watkins (WW, 1999) dropping the bias terms in both:

Lee, Lin, and Wahba (LLW) - Primal:

$$\begin{aligned} &\min_W \sum_{c=1}^{\mathcal{C}} \left[\frac{1}{2} ||w_c||^2 + C \sum_{i:y_i \neq c} l(-w_c^T x_i) \right] \\ &\text{s.t.} \quad \sum_c w_c = 0 \end{aligned} \tag{III.2}$$

Weston and Watkins (WW) - Primal:

$$\min_W \sum_{c=1}^{\mathcal{C}} \left[\frac{1}{2} ||w_c||^2 + C \sum_{i:y_i \neq c} l(w_{y_i}^T x_i - w_c^T x_i) \right] \tag{III.3}$$

Throughout this chapter, $l(x) = \max\{0, 1 - x\}$ will denote the hinge-loss.
Both formulations lead to the following, very similar dual problems:

Lee, Lin, and Wahba (LLW) - Dual:

$$\begin{aligned} \max_{\alpha\in\mathbb{R}^{n\times\mathcal{C}},\bar{w}\in\mathbb{R}^d} & \sum_{c=1}^{\mathcal{C}}\left[-\frac{1}{2}||-X\alpha_c+\bar{w}||^2+\sum_{i:y_i\neq c}\alpha_{i,c}\right] \\ \text{s.t.} \quad & \forall i:\ \alpha_{i,y_i}=0, \forall c\neq y_i: 0\leq\alpha_{i,c}\leq C \end{aligned} \tag{III.4}$$

Weston and Watkins (WW) - Dual:

$$\begin{aligned} \max_{\alpha\in\mathbb{R}^{n\times\mathcal{C}}} & \sum_{c=1}^{\mathcal{C}}\left[-\frac{1}{2}||-X\alpha_c||^2+\sum_{i:y_i\neq c}\alpha_{i,c}\right] \\ \text{s.t.} \quad & \forall i:\ \alpha_{i,y_i}=-\sum_{c:c\neq y_i}\alpha_{i,c}, \\ & \forall c\neq y_i:\ 0\leq\alpha_{i,c}\leq C \end{aligned} \tag{III.5}$$

3.1 Derivation of the Lagrangian dual problems

Now we derive the dual formulation for LLW and WW. For LLW we introduce an auxiliary variable $\bar{w}$ that is exploited by our distributed algorithm, as explained in the next section.

Lin, Lee, and Wahba Using slack variables, the primal LLW problem is stated as follows:

$$\begin{aligned} \min_{W} \quad & \sum_{c=1}^{\mathcal{C}}\left[\frac{1}{2}||w_c||^2+C\sum_{i:y_i\neq c}\xi_{i,c}\right] \\ \text{s.t.} \quad & \sum_c w_c=0 \\ & \forall i:\ \begin{array}{l}\xi_{i,c}\geq 1+w_c^T x_i \\ \forall c\neq y_i:\ \ \xi_{i,c}\geq 0.\end{array} \end{aligned} \tag{III.6}$$

We introduce the Lagrangian multipliers $\alpha\in\mathbb{R}^{n\times\mathcal{C}}$, $\beta\in\mathbb{R}^n$, and $\bar{w}\in\mathbb{R}^d$ with $\alpha_{i,c},\beta_i\geq 0$.

$$\begin{aligned} L(W,\xi,\alpha,\beta,\bar{w}) = & \sum_{c=1}^{\mathcal{C}}\left[\frac{1}{2}||w_c||^2+\sum_{i:y_i\neq c}\left(C\xi_{i,c}+\alpha_{i,c}(1+w_c^T x_i-\xi_{i,c})-\beta_{i,c}\xi_{i,c}\right)\right] \\ & -\bar{w}^T(\sum_c w_c) \end{aligned} \tag{III.7}$$

Due to the fact that Slater's condition (Borwein and Lewis, 2010) holds, we have strong duality and can use the dual formulation

$$\begin{aligned} \max_{\alpha,\beta,\bar{w}} \min_{W,\xi} \quad & L(W,\xi,\alpha,\beta,\bar{w}) \\ \text{s.t.} \quad & \forall\, i\ \forall\, c:\ \alpha_{i,c},\beta_{i,c} \geq 0. \end{aligned}$$

Subsequently, the partial derivatives are given by

$$\begin{aligned} \frac{\partial}{\partial \xi_{i,c}} L(W,\xi,\alpha,\beta,\bar{w}) &= C - \alpha_{i,c} - \beta_{i,c} \\ \frac{\partial}{\partial w_c} L(W,\xi,\alpha,\beta,\bar{w}) &= w_c + \sum_{i:y_i \neq c} \alpha_{i,c} x_i + \bar{w}. \end{aligned}$$

Setting the derivatives equal to zero leads to the following conditions:

$$\begin{aligned} \forall\, i\, \forall\, c:\quad & 0 \leq \alpha_{i,c} \leq C \\ & w_c = -\sum_{i:y_i \neq c} \alpha_{i,c} x_i + \bar{w} \\ & \quad = -X\alpha_c + \bar{w}. \end{aligned}$$

Finally, after updating the Lagrangian, the dual formulation is as follows

$$\max_{\alpha \in \mathbb{R}^{n\times \mathcal{C}}, \bar{w} \in \mathbb{R}^d} \quad \sum_{c=1}^{\mathcal{C}} \left[-\frac{1}{2} || - X\alpha_c + \bar{w}||^2 + \sum_{i:y_i \neq c} \alpha_{i,c} \right] \tag{III.8}$$

$$\begin{aligned} \forall i:\quad & \alpha_{i,y_i} = 0 \\ & \forall c \neq y_i:\ 0 \leq \alpha_{i,c} \leq C. \end{aligned}$$

Weston and Watkins Similarly, we also derive the WW dual. This dualization can also be found in Keerthi et al. (2008). With slack variables the primal WW problem reads as follows:

$$\begin{aligned} \min_{W} \quad & \sum_{c=1}^{\mathcal{C}} \left[\frac{1}{2} ||w_c||^2 + C \sum_{i:y_i \neq c} \xi_{i,c} \right] \\ \text{s.t.} \quad & \forall i:\ \begin{aligned} & \xi_{i,c} \geq 1 + w_c^T x_i - w_{y_i}^T x_i \\ & \forall c \neq y_i:\ \ \xi_{i,c} \geq 0. \end{aligned} \end{aligned} \tag{III.9}$$

Once more we introduce the Lagrangian multipliers $\alpha \in \mathbb{R}^{n\times\mathcal{C}}$, $\beta \in \mathbb{R}^n$ with $\alpha_{i,c}, \beta_i \geq 0$.

$$L(W,\xi,\alpha,\beta) = \sum_{c=1}^{\mathcal{C}} \left[\frac{1}{2} ||w_c||^2 + \sum_{i:y_i \neq c} \left(C\xi_{i,c} + \alpha_{i,c}(1 + w_c^T x_i - w_{y_i}^T x_i - \xi_{i,c}) - \beta_{i,c}\xi_{i,c} \right) \right] \tag{III.10}$$

Again, we have strong duality and can use the dual formulation, due to the fact that Slater's condition (Borwein and Lewis, 2010) holds:

$$\begin{aligned} &\max_{\alpha,\beta} \min_{W,\xi} \quad L(W,\xi,\alpha,\beta) \\ &\text{s.t.} \quad \forall\, i\ \forall\, c:\ \alpha_{i,c}, \beta_{i,c} \geq 0. \end{aligned}$$

Then the partial derivatives are:

$$\begin{aligned} \frac{\partial}{\partial \xi_{i,c}} L(W,\xi,\alpha,\beta) &= C - \alpha_{i,c} - \beta_{i,c} \\ \frac{\partial}{\partial w_c} L(W,\xi,\alpha,\beta) &= w_c + \sum_{i:y_i \neq c} \alpha_{i,c} x_i. \end{aligned}$$

We set the derivatives equal zero and get to the following conditions:

$$\begin{aligned} \forall\, i\, \forall\, c:\quad & 0 \leq \alpha_{i,c} \leq C \\ & w_c = -\sum_{i:y_i \neq c} \alpha_{i,c} x_i = -X\alpha_c. \end{aligned}$$

Then the Lagrangian dual formulation is as follows

$$\begin{aligned} \max_{\alpha \in \mathbb{R}^{n \times \mathcal{C}}} \quad & \sum_{c=1}^{\mathcal{C}} \left[-\frac{1}{2} || - X\alpha_c||^2 + \sum_{i:y_i \neq c} \alpha_{i,c} \right] \\ \text{s.t.} \quad & \forall i:\ \alpha_{i,y_i} = -\sum_{c:c \neq y_i} \alpha_{i,c}, \\ & \forall c \neq y_i:\ 0 \leq \alpha_{i,c} \leq C \end{aligned} \tag{III.11}$$

4 Distributed SVM-algorithms

In this section, we derive algorithms that solve LLW and WW in a distributed manner. We start with addressing LLW.

4.1 Algorithm for Lee, Lin, and Wahba

Please note the optimality condition for LLW is:

$$\bar{w} = \frac{1}{\mathcal{C}} \sum_{c=1}^{\mathcal{C}} X\alpha_c. \tag{III.12}$$

Which can be exploited by solvers to remove the variable $\bar{w}$ from the optimization. In contrast, the core idea of our LLW solver is to actually keep the auxiliary variable, as it decouples the objective function into the following sum:

$$\text{obj}(\alpha) = \sum_{c=1}^{\mathcal{C}} \text{subobj}(\alpha_c, \bar{w}). \tag{III.13}$$

Algorithm 1 Lee, Lin, and Wahba

```
1: function SOLVE-LLW(C, X, Y)
2:     for c = 1..𝒞 do in parallel                ▷ For each class c in parallel.
3:         w_c ← 0
4:         α_c ← 0
5:         for i ∈ I do
6:             k_i ← x_i^T x_i
7:         while not optimal do
8:             optimal ← True
9:             shuffleData()
10:            for i ∈ I \ I_c do                 ▷ Optimize each w_c in parallel.
11:                solve1DimLLW(i, c)
12:            w̄ ← Reduce(Σ_c w_c/𝒞)              ▷ Compute and communicate w̄ across nodes.
13:            w_c ← w_c − w̄
```

Algorithm 2 Solving 1-dim sub-problem of LLW

```
1: function SOLVE1DIMLLW(i,c)
2:     global C, X, k, α_c, w_c, optimal
3:     g ← w_c^T x_i − 1                          ▷ Compute gradient.
4:     if g < −ϵ and α_{i,c} < C then
5:         δ ← min{C − α_{i,c}, −g/k_i}
6:         optimal ← False
7:     if g > ϵ and α_{i,c} > 0 then
8:         δ ← max{−α_{i,c}, −g/k_i}
9:         optimal ← False
10:    w_c ← w_c + δx_i                           ▷ Update primal parameters.
11:    α_{i,c} ← α_{i,c} + δ                      ▷ Update dual parameters.
```

As optimization algorithm we use the dual block coordinate ascent (DBCA, Keerthi et al., 2008, Algorithm 3.1) with a specifically tailored *block structure*, considering as blocks $\bar{w}$ as well as each single coordinate $\alpha_{i,c}$. As we observe from Equation III.13, the optimization of the columns $\alpha_{:,c}$ is mutually independent of each other, given fixed $\bar{w}$. Hence, it can be distributed evenly over $\mathcal{C}$ nodes. On the cth node, we run coordinate ascend on the subobjective subobj$(\alpha_c, \bar{w})$ over $\alpha_{i,c}, i = 1, \ldots, n$, as described in the next paragraph. After one epoch of α computation, the variable $\bar{w}$ is updated via Equation III.12. The final algorithm is shown in Algorithm 1. Following the strategy of Keerthi et al. (2008) we work with primal and dual variables at the same time to enhance the computational efficiency.

Solving the one-dimensional problem It is mandatory to update every single $\alpha_{i,c}$ within Algorithm 1. Let the objective in Equation III.4 be $D(\alpha, \bar{w})$, then using

block ascent (Keerthi et al., 2008) $\alpha_{i,c}$ is optimized by solving the problem

$$\begin{aligned} &\underset{\delta}{\operatorname{argmax}}\, D(\alpha + \delta e_{i,c}, \bar{w}) \\ &\text{s.t. } 0 \leq \alpha_{i,c} + \delta \leq C, \end{aligned} \tag{III.14}$$

where $e_{i,c} \in \mathbb{R}^{n \times \mathcal{C}}$ is one at the (i,c)th coordinate and zero elsewise. Set $w_c := -X\alpha_c + \bar{w}$; then the gradient for δ is $\frac{\partial}{\partial \delta}\left[D(\alpha + \delta e_{i,c})\right] = x_i^T w_c - x_i^T x_i \delta + 1$. Hence, the optimal solution of Equation III.14 is given by $\delta = \min\{C - \alpha_{i,c}, \max\{-\alpha_{i,c}, -\frac{x_i^T w_c - 1}{x_i^T x_i}\}\}$. The corresponding pseudo-code can be found in Algorithm 2.

4.1.1 Convergence

It was shown that the block coordinate ascent method converges under suitable regularity conditions (Tseng, 2001; Bertsekas et al., 1995). Our objective is continuously differentiable and strictly convex. The constraints are solely box constraints, hence the feasible set decomposes as a Cartesian product over the blocks. Algorithm 1 traverses the two blocks in cyclic order. Under these conditions, the DBCA method provably converges (Bertsekas et al., 1995, Prop. 2.7.1).
Note that in practice, we observed speedups by updating $\bar{w}$ in Algorithm 1 after each tenth of an epoch, breaking the cyclic order. The blocks of coordinates are then traversed in so-called *essentially cyclic order*, e.g., Section 2 in Tseng (2001), meaning that there exists $T \in \mathbb{N}$ such that each block is traversed at least once after T iterations. Closer inspection of the proof in Prop. 2.7.1 in Bertsekas et al. (1995) reveals that the result holds also under this slightly more general assumption.
Further, we drop variables $\alpha_{i,c}$ in the optimization if they are not updated in 3 subsequent epochs. This is called shrinking and, e.g., also used in Keerthi et al. (2008) for MC-SVMs. Once the stopping condition holds, we run another epoch of optimization over all variables, including the ones that were dropped. If the stopping criterion is then met, we terminate the algorithm, otherwise we continue the optimization.

4.1.2 Implementation details

Our implementation uses OpenMPI (MPI, Gropp et al., 1996) for inter-machine and OpenMP (multi-core or MC, Dagum and Enon, 1998) for intra-machine communication. Note that Algorithm 1 has very mild communication requirements: only the sum of all weight vectors $\bar{w} = \sum_c w_c$ needs to be communicated. Hence, MPI suffers very little from communication overhead between the various machines. In practice, we may not be able to fully parallelize to the maximum of $\mathcal{C}$ cores; therefore our algorithm will divide the set of classes into a number of chunks and optimize each one sequentially. Given c cores, it is advisable to use c chunks with $\mathcal{C}/c$ classes.
Recall, $\bar{d}$ is the average number of non-zero entries per sample and $\bar{n}$ is the average number of samples per class. Given c cores and I iteration steps, the optimization has an asymptotic runtime estimate of $\mathcal{O}(I \cdot (\frac{\mathcal{C}}{c} \cdot \bar{n} \cdot \bar{d} + d \log_2 c))$, where the first part of the sum amounts for the gradient updates and the second for the model

communication and update. Given the maximal average dual sparsity during training as $1 - \bar{a}$ at each node, the algorithm exhibits an asymptotic space complexity of $\mathcal{O}(\frac{\mathcal{C}}{c} \cdot (\bar{d} + \bar{n} \cdot \bar{a}) + d)$ to store the weight matrix, the dual coefficients and the weight vector used for averaging in sparse format. The current implementation relies on a dense parameter representation, but can be adjusted accordingly.

4.2 Algorithm for Weston and Watkins

In this section, we propose a distributed algorithm for WW, which draws inspiration from the 1-factorization problem of a graph.

4.2.1 Preliminaries

Also this approach is based on running dual coordinate ascend, e.g., algorithm 3.1 in Keerthi et al. (2008), this time over $\alpha_{i,c}$ on the WW objective function as follows. Denote the objective in Equation III.5 by $D(\alpha)$ and we recall that black ascent (Keerthi et al., 2008) optimizes $\alpha_{i,c}$ by solving the following problem

$$\begin{aligned} &\underset{\delta}{\operatorname{argmax}}\, D(\alpha + \delta \mathbf{e}_{i,c}) \\ &\text{s.t. } 0 \leq \alpha_{i,c} + \delta \leq C. \end{aligned} \tag{III.15}$$

Setting $w_c = \sum_{i:y_i \neq c}(-x_i \alpha_{i,c} + \sum_{c:c \neq y_i} x_i \alpha_{i,c})$, the gradient for δ is given by $\frac{\partial}{\partial \delta}\left[D(\alpha + \delta \mathbf{e}_{i,c})\right] = -x_i^T(w_{y_i} - w_c) - x_i^T x_i \delta + 1$. And is optimal at:

$$\delta = \min\left(C - \alpha_{i,c}, \max\left(-\alpha_{i,c}, \frac{x_i^T(w_{y_i} - w_c) - 1}{2x_i^T x_i}\right)\right) \tag{III.16}$$

This computation is summarized in Algorithm 3.

Algorithm 3 Solving 1-dim sub-problem of WW

```
1:  function SOLVE1DIMWW(i,c)
2:      global C, X, w_{y_i}, w_c, α_c, optimal
3:      g ← (w_{y_i}^T − w_c^T) x_i − 1                ▷ Compute gradient.
4:      if g < −ε and α_{i,c} < C then
5:          δ ← min{C − α_{i,c}, −g/2k_i}
6:          optimal ← False
7:      if g > ε and α_{i,c} > 0 then
8:          δ ← max{−α_{i,c}, −g/2k_i}
9:          optimal ← False
10:     w_{y_i} ← w_{y_i} + δ x_i                      ▷ Update primal parameters.
11:     w_c ← w_c − δ x_i
12:     α_{i,c} ← α_{i,c} + δ                          ▷ Update dual parameters.
```

4.2.2 Core observation

We observe from above that optimizing with regard to $\alpha_{i,c}$ will require only the weight vectors w_{y_i} and w_c. In other words, given four different classes c_1, c_2, c_3, c_4, the optimization of the block of variables $(\alpha_i, c_1)_{i:y_i=c_2}$ — according to Equation III.16 — is independent of the optimization of the block $(\alpha_i, c_3)_{i:y_i=c_4}$. Hence it can be parallelized. In the next section we describe how we exploit this structure to derive a distributed optimization algorithm.

4.2.3 Excursus: 1-factorization of a graph

Assume that $\mathcal{C}$ is even. The core idea now is to form $\frac{\mathcal{C}}{2}$ many disjoint blocks $(\alpha_i, c_1)_{i:y_i=c_2}, \ldots, (\alpha_i, c_{\mathcal{C}-1})_{i:y_i=\mathcal{C}}$ of variables. Each of these blocks can be optimized in parallel. The challenge is to derive a maximally distributed optimization schedule where each block $(\alpha_i, c_j)_{i:y_i=c_k}$ for any $j \neq k$ is optimized.

To better understand the problem, we consider the following analogy. We are given a football league with $\mathcal{C}$ teams. Before the season, we have to decide on a schedule such that each team plays with every other team exactly once. Furthermore, all teams shall play on every matchday so that in total we need only $\mathcal{C} - 1$ matchdays. This problem is the *1-factorization problem in graph theory*, e.g., Bondy and Murty (1976). The solution to this problem, illustrated in Figure III.1, is as follows.

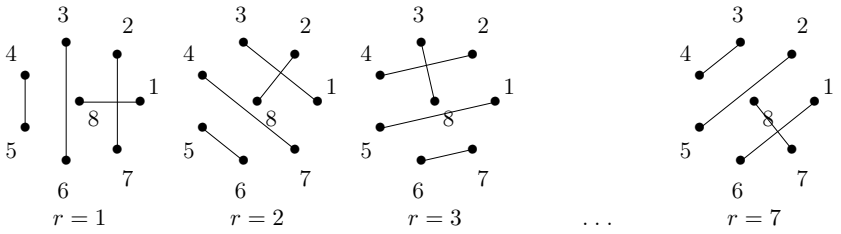

Fig. III.1: 1-factorization. *Illustration of the solution of the 1-factorization problem of a graph with $\mathcal{C} = 8$ many nodes. The goal is to match each node with any other node once. The solution idea is to arrange node 8 centrally and at each step rotate the pattern by one.*

We arrange one node centrally and all other nodes in a regular polygon around the center node. At round r, we connect the centered node with node r and connect all other nodes orthogonal to this line. The pseudocode to compute the partner of a given node c at a certain round r is given in Algorithm 4. Note that in case of an uneven number of classes, we introduce a dummy class $\mathcal{C} + 1$, making the number of classes even. When running the algorithm we skip all computations involving the dummy class.

Algorithm 4 Solving the graph 1-factorization problem. Indices start with one.

```
1: function MatchClass(C,c,r)
2:     if C is even and c = C then
3:         return r
4:     if c = r then
5:         if C is even then
6:             return C
7:         else
8:             return c
9:     return mod(2r − c, C − 1)
```

4.2.4 Algorithm

Finally the algorithm shown in Algorithm 5 optimizes the WW formulation. It performs DBCA over the variables $\alpha_{i,c}$ using the schedule derived in Section 4.2.3 and the coordinate updates derived in Section 4.2.1.

Algorithm 5 Watkins-Weston

```
1: function Solve-WW(c,X,Y)
2:     for c = 1..C do in parallel                ▷ Initialize in parallel.
3:         w_c ← 0
4:         α_c ← 0
5:     for i ∈ I do
6:         k_i ← x_i^T x_i
7:     while not optimal do
8:         optimal ← True
9:         shuffleData()
10:        for r = 1..C − 1 do                    ▷ For each matching round:
11:            for c = 1..C do in parallel        ▷ Update class pairs in parallel.
12:                c̃ ← matchClass(C, c, r)
13:                if c̃ > c then
14:                    Gather w_c̃
15:                    for i ∈ I_c do
16:                        solve1DimWW(i, c̃)
17:                    for i ∈ I_c̃ do
18:                        solve1DimWW(i, c)
19:                    Return w_c̃
```

4.2.5 Convergence and implementation details

Note that our algorithm performs again the same coordinate updates as Algorithm 3.1 in Keerthi et al. (2008). Hence, they share the same favorable convergence behavior. Formally, convergence is guaranteed for exactly the same reasons discussed

in Section 4.1.1. We also employ the same speedup tricks, i.e., shrinking and updating every tenth of an epoch.

In practice, because of limitations of computational resources, we might not be able to fully parallelize to the maximum of $\mathcal{C}/2$ cores. In that case, our algorithm divides the set of classes into number-of-cores many chunks and solves each bundle sequentially. For optimal speedup, it is advisable to arrange the classes into chunks of equal number of classes and data points.

Given the average number of non-zero entries per sample $\bar{d}$ and the average number of samples per class $\bar{n}$, c cores and I iteration steps, the optimization has an asymptotic runtime estimate of $\mathcal{O}(I \cdot \mathcal{C} \cdot (\frac{\mathcal{C}}{c} \cdot \bar{n} \cdot \bar{d} + d))$, where the first part of the sum amounts for the gradient updates and the second for the model communication. Given the maximal average dual sparsity during the training as $1 - \bar{a}$, at each node the algorithm exhibits an asymptotic space complexity of $\mathcal{O}(\frac{\mathcal{C}}{c} \cdot (\bar{d} + \bar{n} \cdot \bar{a}))$ to store the weight matrix and the dual coefficients in sparse format. The current implementation uses a dense parameter representation, but can be adjusted accordingly.

As with LLW, we implemented a mixed MPI-OpenMP solver for WW. However, note that, while LLW has mild communication needs, WW needs to pair the weight vectors of the matched classes c and $\tilde{c}$ in each epoch, for which $\mathcal{C}/2$ weight vectors needs to communicated among computers. Therefore it is crucial to communicate efficiently.

We tackled the problem as follows. First of all, we use OpenMP for computations on a single machine to efficiently parallelize among cores. Here, due to the shared memory, no weight vectors need to be moved. The more challenging task is to handle inter-machine communication efficiently. Our approach is based on two key observations.

If the data is high-dimensional data, yet sparse, we keep the full weight matrix in memory for fast access, and communicate only the non-zero entries between computers. Regardless of the increased computational effort, this takes only a fraction of time compared to sending dense data.

Furthermore, we relax the WW matching scheme to reduce the communication effort. Coming back to the football analogy, recall that we match classes like football teams are matched during a season (see Section 4.2.3). Considering that each football team belongs to a country (like each class belongs to a machine), we would like to make every team play against every other team, while reducing the amount of "travel" performed. To do so we add an overlay matching scheme — in addition to the matching of teams on a single country, which is performed as follows. In the overlay scheme the countries are arranged with the same matching scheme (see Section 4.2.3) and every time two countries are matched, the teams of country travel *together* to the matched country and play against each team of that country. Returning to classes and machines, this means we transfer bundles of classes (countries) together between computers, e.g., employ batching, and reduce the network communication drastically compared to a scheme where single classes are matched across machines. Every epoch we first execute the scheme within a country and then the one across countries.

5 Experiments

This section is structured as follows. After introducing the experimental setup we empirically verify the soundness of the proposed algorithms by comparing them to an existing, slower solver on small scale datasets. Then we introduce the employed datasets, on which we investigate the convergence and runtime behavior of the proposed algorithms as well as the induced classification performance.

5.1 Setup

We use two different types of machines for our experiments. Type A has 20 physical CPU cores, 128 GB of memory and a 10 GigaBit ethernet network. Type B has 24 physical CPU cores and 386 GB of memory. We use up to 4 type A machines for our distributed solvers. On type B we ran the experiments involving the not distributed solver for Crammer and Singer (2002) due to the memory requirements.
Training repetitions were run on training sets with a random order of the data. Note that the training set is the same in each run; only the order of points is shuffled, which can impact the DBCA algorithm. For each result we report the average over the indicated number of repetitions. Furthermore, we report the model density, or sparsity, which is defined as the fraction of non-zero model parameters.
On one hand, the results of our solvers always converge to same prediction result and, on the other hand, we found the timing results to be consistent and reliable across datasets and regularization parameters. Therefore we omit further statistical hypotheses tests.
For LIBLINEAR solvers we use the newest available version as of April 2016 with the default settings. We implemented our solveres using OpenMP, OpenMPI, and the Python-ecosystem. In more detail, we used Van Der Walt et al. (2011), Behnel et al. (2011), and Dalcin et al. (2011) in our software. The source code is provided under `https://github.com/albermax/xcsvm`.

5.2 Validation of solvers

In our first experiment, we validate the correctness of the proposed solvers. We downloaded data from the LIBLINEAR (Fan et al., 2008)[1] and UCI (Asuncion and Newman, 2007)[2] dataset repositories. Where training and test splits are unavailable, we split the data once into 90% train and 10% test sets. For each dataset, the optimal feature scaling was selected, in order to maximize the average accuracy on the test sets. Datapoints in the datasets iris and news were thus normalized to unit norm, and in the datasets letter and satimage were normalized to unit variance. All other data was considered without normalization.
Then we compared our LLW and WW solvers with the state-of-the-art implementation contained in the ML library Shark (Igel et al., 2008). To do so we implemented the same stopping criteria as Igel et al. (2008). The results, averaged over 10 runs, are

[1] `https://www.csie.ntu.edu.tw/~cjlin/liblinear/`
[2] `https://archive.ics.uci.edu/ml/datasets.html`

	S-LLW		D-LLW		S-WW		D-WW	
Dataset:	Err.	Den.	ΔErr.	ΔDen.	Err.	Den.	ΔErr.	ΔDen.
SensIT								
$\log(C)$: *-1*	21.34	100.0	0.00	0.00	19.88	100.0	0.00	0.00
0	20.95	100.0	0.00	0.00	19.51	100.0	0.00	0.00
1	20.78	100.0	0.00	0.00	19.38	100.0	0.00	0.00
glass								
$\log(C)$: *-1*	66.67	100.0	0.00	0.00	38.10	100.0	0.00	0.00
0	61.90	100.0	0.00	0.00	19.05	100.0	0.00	0.00
1	33.33	100.0	0.00	0.00	19.05	100.0	0.00	0.00
iris								
$\log(C)$: *-1*	13.33	100.0	0.00	0.00	6.67	100.0	0.00	0.00
0	26.67	100.0	0.00	0.00	13.33	100.0	0.00	0.00
1	26.67	100.0	0.00	0.00	13.33	100.0	0.00	0.00
letter								
$\log(C)$: *-1*	87.04	100.0	0.00	0.00	28.26	100.0	**-0.01**	0.00
0	87.24	100.0	0.00	0.00	29.03	100.0	+**0.01**	0.00
1	61.91	100.0	0.00	0.00	28.93	100.0	**-0.01**	0.00
news20								
$\log(C)$: *-1*	29.23	97.24	0.00	0.00	15.30	49.72	+**0.02**	+**0.44**
0	22.97	97.24	0.00	0.00	14.80	42.70	0.00	+**2.04**
1	16.15	97.04	0.00	+**0.13**	15.98	43.47	0.00	+**2.50**
rcv1								
$\log(C)$: *-1*	47.96	78.00	0.00	0.00	11.31	23.45	0.00	+**2.97**
0	33.41	77.98	**-0.14**	+**0.02**	11.52	20.12	0.00	+**2.81**
1	12.03	77.98	0.00	+**0.02**	12.03	20.06	0.00	+**2.99**
satimage								
$\log(C)$: *-1*	26.73	100.0	+**0.02**	0.00	15.80	100.0	0.00	0.00
0	26.80	100.0	0.00	0.00	15.53	100.0	**-0.06**	0.00
1	26.90	100.0	0.00	0.00	16:00	100.0	**-0.04**	0.00
splice								
$\log(C)$: *-1*	16.37	100.0	**-0.08**	0.00	16.16	100.0	0.00	0.00
0	16.15	100.0	**-0.06**	0.00	16.28	100.0	+**0.09**	0.00
1	16.28	100.0	+**0.06**	0.00	16.24	100.0	+**0.08**	0.00
usps								
$\log(C)$: *-1*	31.84	100.0	0.00	0.00	8.17	100.0	0.00	0.00
0	30.04	100.0	+**0.05**	0.00	9.37	100.0	0.00	0.00
1	28.00	100.0	0.00	0.00	10.51	100.0	0.00	0.00

Tab. III.1: Comparison to existing solver. *Error on the test set and model density in % of the Shark solver (denoted S) and the respective difference achieved by the proposed solver (denoted D), averaged over* 10 *repetitions. The results across solver implementations show very good accordance.*

shown in Table III.1. We observe good accordance of the results and model sparsity of the proposed solvers and the reference implementation from the Shark toolbox, thus confirming that our respective solvers are indeed exact solvers of LLW and WW.
Furthermore, at random we tested whether the duality-gap of our solvers closes. We did this for both solvers with different C values and datasets. In any case the duality gap closed, i.e., decreased by an order of two magnitudes. Based on this we chose our stopping criteria ϵ equal to 0.1 for the LSHTC datasets in the next section.

5.3 Datasets

We experiment on large classification datasets, where the number of classes ranges between 451 and 27,875. The relevant statistics of the datasets are shown in Table III.2. The LSHTC-* datasets are high-dimensional text datasets taken from the LSHTC corpus (Partalas et al., 2015). The datasets belong to the released competition rounds 1 to 3, i.e., '10-'12. LSHTC-2011 and LSHTC-2012 originate from the DMOZ corpus. The most challenging dataset is given by LSHTC-2011. It contains the most samples, classes and dimensions. The according features were extracted using the TF/IDF representation and we use the full feature resolution for training.

Dataset	**n train**	**n test**	$\mathcal{C}$	**d**
LSHTC-small	4,463	1,858	1,139	51,033
LSHTC-large	128,710	34,880	12,294	381,581
LSHTC-2012	383,408	103,435	11,947	575,555
LSHTC-2011	394,754	104,263	27,875	594,158

Tab. III.2: Dataset properties. *The table shows the used datasets from the LSHTC-corpus and their properties. n train and n test denote the number of samples in the training and test set respectively, $\mathcal{C}$ the number of classes and d the number of dimensions. The most challenging dataset is given by LSHTC-2011. It contains the most samples, classes and dimensions.*

5.4 Speedup

In order to measure the speedup provided by increasing the number of machines/cores, we run a fixed amount of iterations for a fixed amount of class bundles over the whole LSHTC-large dataset. We use 10 runs over 10 iterations with a fixed parameter C equal 1 without shrinking. While the MC execution works on one machine, the MPI executes on 2 or 4 machines by spreading the used cores evenly on each node. This analysis covers merely the technical aspects of our implementations and later in the section we will have a closer look into the interplay of the mathematical, hence convergence, and distribution properties of our proposed solutions.
The results are shown in Figure III.2. Both solvers exhibit linear speedup, regardless if distributed or not, due to the small communication cost. Yet the speedup of WW is bounded by a larger constant compared to LLW.

In more detail, we note that WW implementation does not show linear speedup when the number of used cores per machine gets increased from 1 to 2. Afterwards again a linear increase is observed, but now with a higher constant factor. This holds for each variant, e.g., MC, MPI-2, and MPI-4. We suspect that the cache throughput limit is hit and causes this effect. The implementation for WW was designed to take advantage of memory alignment, in contrast to the LLW implementation for which this stalling cannot be seen. Thus, the scaling of our application is memory-bound rather than CPU-bound and suggests systems with more cache throughput would allow for a higher speedup. For instance, GPUs are optimized for memory throughput, but in general they do not perform well with applications with irregular memory access such as our sparse solvers and it is unclear if our application would benefit from using them.
The scaling across machines, i.e., from MC to MPI-2 to MPI-4, seems also to perform favorable, but due to the limited number of machines used one cannot generalize this further.

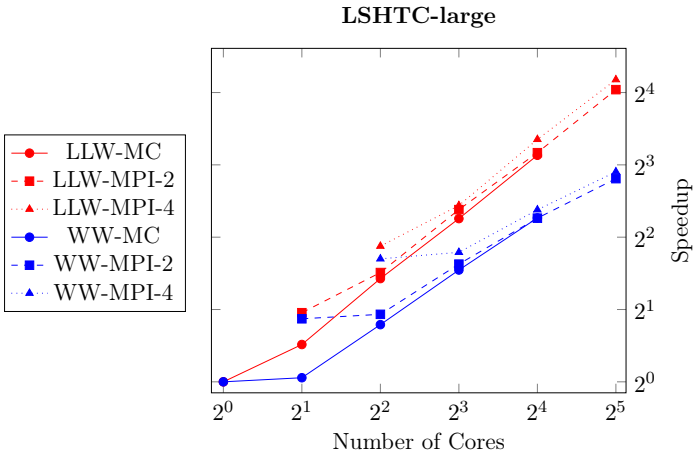

Fig. III.2: Speedup. *Speedup of our solver respectively in the number of cores. For *-MPI-2 and *-MPI-4 the number of cores is split evenly on 2 and 4 machines. We observe a linear speedup in the number of cores for both solvers.*

5.5 Classification and timing results

Now we evaluate and compare the proposed algorithms on the LSHTC datasets for a range of C values, i.e., we perform no cross-validation. For comparison we use a solver from the well-known LIBLINEAR package, namely the one-vs.-rest multi-core implementation with L2L1-loss (OVR, Chiang et al., 2016). For completeness we also include the single-core Crammer and Singer implementation (CS, Fan et al., 2008). Each training algorithm was run 3 times, using randomly shuffled data, and the results were averaged. Note that the training set is the same in each run, but the different order of data points can impact the convergence and runtime of the algorithms. Due to the lack of performance we do not compare to the LLW and WW implementation in the Shark ML library.

Dataset: LSHTC	*Error*				*Model-Density*			
	OVR	**CS**	**WW**	**LLW**	**OVR**	**CS**	**WW**	**LLW**
-small								
$\log(C)$: *-3*	93.00	59.74	72.82	**93.00**	92.74	11.11	69.73	**92.74**
-2	85.36	59.74	65.34	93.00	81.54	11.13	16.44	92.74
-1	74.54	59.74	57.59	93.00	46.76	11.12	6.06	92.74
0	64.37	55.49	54.57	93.00	38.20	11.76	5.74	92.74
1	**57.75**	**54.57**	**54.41**	93.00	**38.63**	**11.69**	**5.73**	92.74
-large								
$\log(C)$: *-3*	88.12	58.57	66.47	**95.86**	75.26	2.53	18.50	**100.0**
-2	85.21	58.57	60.58	95.86	45.14	2.53	4.45	100.0
-1	77.96	57.82	55.28	95.86	25.28	2.55	1.71	100.0
0	63.11	**53.61**	**53.98**	95.86	18.33	**2.69**	**1.61**	100.0
1	**57.18**	54.18	54.41	*	**18.55**	2.67	1.66	*
-2012								
$\log(C)$: *-3*	83.66	49.81	58.02	**92.63**	72.60	1.73	16.97	**99.52**
-2	75.15	49.65	50.20	92.63	46.20	1.71	4.06	99.52
-1	60.38	46.14	44.94	92.63	25.87	1.76	1.52	99.52
0	47.33	**42.67**	**44.01**	*	18.20	**2.06**	**1.42**	*
1	**46.83**	45.60	46.15	*	**18.46**	2.09	1.47	*
-2011								
$\log(C)$: *-3*	87.95	59.09	68.19	**96.18**	72.38	1.57	13.49	**100.0**
-2	85.85	59.09	62.14	96.18	45.97	1.57	3.16	100.0
-1	76.78	58.18	57.31	96.18	25.97	1.55	1.19	100.0
0	63.11	**55.58**	**56.94**	*	18.24	**1.69**	**1.11**	*
1	**60.01**	57.78	58.32	*	**18.46**	1.70	1.14	*

Tab. III.3: Test error and model density. *Test set error and model density in % as achieved by the OVR, WW, and LLW, and CS solvers on the LSHTC datasets. Lower is better. For each solver the result with the best error is in bold font. For LLW entries with a '*' did not converge within a day of runtime. The all-in-one solvers WW and CS outperform consistently OVR.*

Classification results Table III.3 shows the error and the model sparsity, i.e., fraction of zeros in the model parameters, for the compared solutions. We further provide the Micro-F1 and Macro-F1 score in Table III.4.
Unfortunately, we note that the solution for LLW shows for all datasets a high error and also results in very dense models. We examine the behavior of this solver in more detail below, and omit it from the further analysis here.
For all datasets the other all-in-one multi-class formulations, i.e., WW and CS, perform significantly better than OVR. On one hand the error is smaller and the F1-scores are better. On the other hand, the learned models are much sparser, i.e., up to a magnitude. The results confirm the results from Doğan et al. (2016) and also justify the increased solution complexity of these formulations.
Comparing WW and CS, CS performs as well or slightly better at classifying. Though

Dataset:	*Micro-F1*				*Macro-F1*			
LSHTC	**OVR**	**CS**	**WW**	**LLW**	**OVR**	**CS**	**WW**	**LLW**
-small								
$\log(C)$: *-3*	7.00	40.26	27.18	**7.00**	0.61	22.08	10.73	**0.61**
-2	14.42	40.26	34.66	7.00	2.70	22.08	16.15	0.61
-1	25.46	40.26	42.41	7.00	8.72	22.08	24.71	0.61
0	35.47	44.46	45.43	7.00	16.42	26.70	28.75	0.61
1	**42.41**	**45.48**	**45.59**	7.00	**25.09**	**28.73**	**29.15**	0.61
-large								
$\log(C)$: *-3*	11.77	41.35	33.53	**4.14**	0.88	25.43	15.05	**0.09**
-2	14.80	41.52	39.42	4.14	1.51	25.41	20.83	0.09
-1	22.02	42.19	44.72	4.14	3.35	25.83	27.90	0.09
0	36.86	**46.41**	**46.02**	*	14.76	30.99	**31.29**	*
1	**42.80**	45.83	45.59	*	**25.87**	**31.13**	31.12	*
-2012								
$\log(C)$: *-3*	16.34	50.19	41.98	**7.37**	0.28	20.55	8.08	**0.01**
-2	24.85	50.35	49.80	7.37	0.69	20.72	16.17	0.01
-1	39.62	53.86	55.06	7.37	2.64	23.76	25.94	0.01
0	52.67	**57.33**	**55.99**	*	12.46	**32.57**	**32.06**	*
1	**53.17**	54.40	53.85	*	**24.41**	31.84	30.95	*
-2011								
$\log(C)$: *-3*	12.05	40.91	31.81	**3.82**	0.46	22.44	10.47	**0.05**
-2	14.15	40.91	37.86	3.82	0.62	22.46	16.48	0.05
-1	23.22	41.82	42.69	3.82	1.89	23.37	23.17	0.05
0	36.89	**44.42**	**43.06**	*	10.60	**26.97**	**27.25**	*
1	**39.99**	42.22	41.86	*	**21.30**	26.31	26.97	*

Tab. III.4: F1-scores. *Micro-F1 and Macro-F1 scores in % as achieved by the OVR, WW, LLW, and CS solvers on the LSHTC datasets. Higher is better. For each solver and each metric the best result across C values is in bold font. For LLW entries with a '*' did not converge within a day of runtime. The all-in-one solvers WW and CS outperform consistently OVR.*

WW leads to sparser models, which are up to half of the size of the respective CS models. To the best of our knowledge this is the first comparison of these well-known multi-class SVMs on the studied LSHTC data.

One of the key challenges when learning a linear model on such datasets is the model size. Due to the high-dimensional input and high-dimensional output domains the model has a potential size of $d \times \mathcal{C}$. Given the right regularization the actual model size can be significantly smaller due to sparsification, i.e., pushing a large fraction of the parameters towards 0 (see Table III.3). Yet there is no guarantee that the model will be sparse as we see from the results — especially during the optimization, thus the model size can be and is a prohibitive factor for the scaling of such methods and distributing the model across computing instances can be a viable tool to alleviate this.

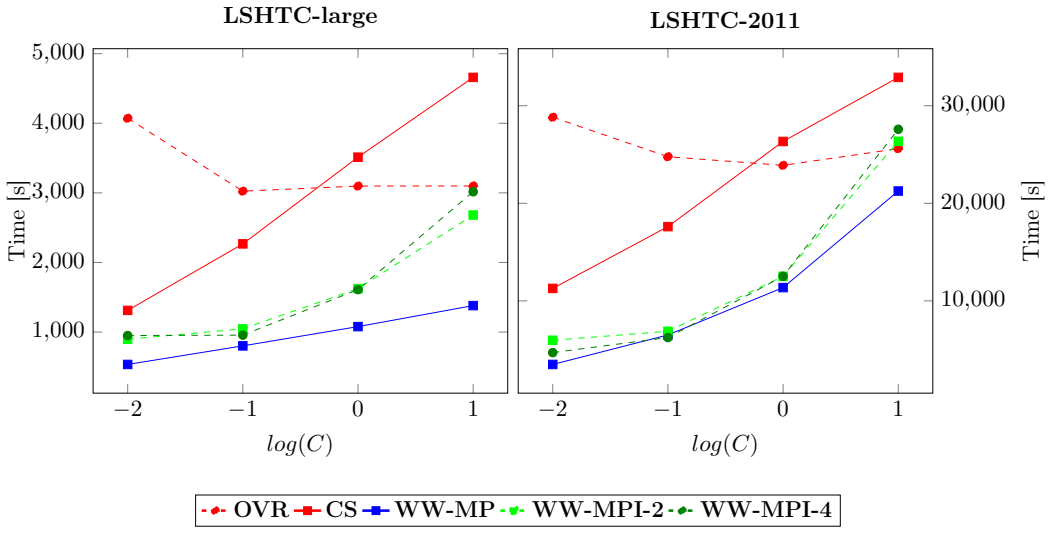

Fig. III.3: Training times. *Training time for different regularization parameters C for the various solvers. We observe that the parameter C has a significant influence on the runtime of the all-in-one solvers WW and CS, while it is modest for the OVR solution. All parallel solvers use the same amount of cores. LLW was omitted due to the slow convergence.*

Timing results Another important aspect of our analysis is to examine how long our solvers take to *converge* to a solution. Here it is important to note that different formulations and optimization schemes lead to different convergence rates. This is of interest as the runtime of many Machine Learning solution is affected on one hand by the mathematical optimization properties and the implementation details on the other hand.

The first property to inquire is the regularization parameter C, which has a major impact on the convergence and thus the runtime of SVMs algorithms. The larger C the less constraint the optimization space and the more possible solutions exist. The second is the interplay between shrinking and distributing. Shrinking removes variables that are unlikely to change from the optimization process and thereby reduces the size of the problem as well as the runtime. But this also renders the positive effect of distributing the computation smaller, because the parallelizable fraction of the computation shrinks and this puts more stress on the communication overhead.

The main focus is the comparison of the parallel and distributed solvers which run on type A machines, of which we had 4 to our disposition. For completeness we also report the runtime of the single core solver CS, which runs on one machine of type B to meet the large memory requirements of this approach. The multi-core solutions OVR and WW-MC use 16 cores, while the MPI solution WW-MPI spreads over 2 or 4 machines using 8 and 4 cores respectively at each node, *thus trains the model distributed*. We omit LLW from this comparison, because the classification and convergence performance did not meet the expectations and refer to the next section.

From Figure III.3, we observe that the runtime of all-in-one methods WW and CS

varies with varying C and they perform fastest in a more regularized domain, i.e., for small C values. On the other hand OVR seems more resilient to a changing regularization parameter C. We also observe that the communication overhead for our solution of WW changes depending on the regularization strength. The figure indicates that overhead is modest, yet increases significantly in the unregularized domain.
Comparing the parallel and distributed solvers one can find that multi-core solution WW-MC always performs faster than multi-core solution OVR, and that the distributed solutions WW-MPI-* despite increased communication effort still perform better than the multi-core solution OVR in all except one case.
Comparing the all-in-one solvers, the sequential solver CS converges surprisingly fast compared to the parallelized WW. The single core performance of WW, which approximately takes the time of WW-MC times the ideal speedup 4, as derived from Figure III.2, would run somewhat slower than the one of CS. Which is reasonable as the objective of CS is far less complex than the one of WW and is very suitable for shrinking approaches. We note that CS will likely not scale to much larger problem settings due to its sequential nature.

5.5.1 Lin, Lee, & Wahba

Knowing that LLW converges to the correct solution, as the duality-gap closes (see Section 5.2), the results indicate that the chosen C range is not suitable. For LSHTC-small we conducted experiments with much larger C values. And indeed, as shown in Table III.5, LLW performs best in a nearly unconstrained setting. In our experiments we observed that the model learned by LLW is never sparse, neither in the weight matrix W, nor in dual factors α. Resource limitations and slow convergence properties hindered us to conduct experiments with even larger C values. It is left to future work to explore this space or even develop a unconstrained version of LLW.

$\log(C)$:	**2**	**3**	**4**
Error:	87.73	66.74	59.31
Micro F1:	2.08	15.07	40.69
Macro F1:	12.27	33.26	24.58
W**-Density:**	92.74	92.74	92.74
α**-Density:**	99.88	99.87	99.90

Tab. III.5: Further results for the LLW-solver. *Error, Micro-F1, and Macro-F1 on the test set and model density in % of the LLW solver on the LSHTC-small dataset. One can observe that LLW performs the better the less regularized the optimization is, i.e., the larger C.*

6 Discussion

Our first discussion point is the limitation of our solvers to multi-class classification, which hinders us from exploring even larger datasets in this context as they typically

have multiple labels per data point (multi-label classification). Our work as well as Doğan et al. (2016) suggests that all-in-one solvers classify better than OVR in multi-class classification settings. On the other hand, the work of Babbar and Schölkopf (2017, 2018) shows that approximated all-in-one solvers for CS do not match the performance of plain OVR in the case of multi-label classification, even though on larger scale, and thereby points to the most promising future direction of this contribution: the extension of the proposed WW formulation to multi-label losses. In order to examine the hypothesis that exact all-in-one solvers perform better for the discussed (larger) multi-label datasets, it requires an adaptation of the WW formulation and it might require an adaptation of the presented scaling scheme. A possible extension of Equation III.3 could be realized in the following way where y_i denotes the set of labels for a data point i:

$$\min_W \sum_{c=1}^{\mathcal{C}} \left[\frac{1}{2}||w_c||^2 + C \sum_{i:c \notin y_i} l(w_{y_i}^T x_i - w_c^T x_i) \right] \tag{III.17}$$

Another hindrance to scale to the larger multi-label datasets for the current implementation is handling the potential huge model sizes. This contribution tackles this by distributing the model across different computing instances, yet this measure might be not enough for model sizes of larger datasets[3]. We note that the discussed large multi-label datasets are very sparse, thus it is unlikely that the actual model will be very dense, yet still possible as we observed in our experiments. One way to alleviate this issue would be to store the model parameter in sparse format in a hash table. It is unclear how this would effect the overall scaling behavior of the implementation. An alternative might be to only store model parameters which can have a non-zero gradient — as for some it will always be zero, because there is no corresponding connection between an output label and input feature in the training set. The PD-Sparse algorithms (Yen et al., 2016, 2017) approximates the CS loss by imposing additional sparsity constraints which provably lead to a final sparse solution and empirically also keep the model during training sparse. Other methods like tree- (Prabhu and Varma, 2014; Jain et al., 2016; Jasinska et al., 2016; Prabhu et al., 2018), embedding- (Yu et al., 2014; Bhatia et al., 2015; Tagami, 2017) or deep learning-based (Vincent et al., 2015; Liu et al., 2017; Grave et al., 2017) techniques circumvent this issue by construction. These methods impose a limit to the model capacity and this leads us to the next discussion point.

Another interesting question for datasets with large output label spaces revolves around so call head and tail labels (Jain et al., 2016). The key idea is that few head labels occur very often in the dataset, in contrast to many tail labels that occur seldom and form the tail of the label distribution. Interestingly, it is to note that in many applications it is more important or informative to predict rarely occurring labels with higher accuracy than head labels. E.g., consider a data point with the following labels: "book", "novel", "Miss Marple", "Agatha Christie". Labels like "book" and "novel" would

[3] E.g., considering the Amazon-3M dataset in the Extreme Classification repository (http://manikvarma.org/downloads/XC/XMLRepository.html) the size of a (dense) linear model would be in the order of Petabytes.

be likely head label, i.e., occur often, in contrast to tail labels like "Miss Marple" and "Agatha Christie" that occur seldom, but are much more informative and might even imply more frequent labels. In our experiments especially the F1-macro score puts more stress on tail-labels and the result suggest that all-in-one methods perform favorable to OVR. In literature there is a discussion which methods captures the structure of tail labels best (Jain et al., 2016; Babbar and Schölkopf, 2018). Generally, tree-, embedding- or deep-learning-based approaches, as mentioned above, tend to capture global structures better, linear classifiers such as SVMs tend to capture tail distributions more accurately (Babbar and Schölkopf, 2017, 2018). The work of Cheng et al. (2016) tries to combine the best of both worlds by combining a deep neural network with a shallow classifier. This, again, suggests to extend the WW solver to multi-label losses.

For the domain of classification with large label spaces a few desiderata can be named. Most important are a small asymptotic training or prediction time — ideally it would be logarithmic in the number of classes $log(\mathcal{C})$. In the case of prediction time this is achieved, e.g., by tree-based techniques (Prabhu and Varma, 2014; Jain et al., 2016; Jasinska et al., 2016; Prabhu et al., 2018). For our SVM solvers the prediction and training time is at least linear and thus not ideal. The prediction time itself could possibly be reduced by using methods like maximum inner product search, e.g., Shrivastava and Li (2014). Likely one needs to trade-off between matching these criteria and the prediction performance of shallow models like SVMs.

In this context we would like to discuss the interplay of mathematical/methodological and engineering domains in Machine Learning. We did not mention the adaptation of the LLW formulation as the performance was surprisingly weak in our experiments. While our solution and implementation for LLW shows very good scaling properties and, furthermore, it provably and empirically converges to the right solutions; but eventually it did not converge in a *reasonable* amount of time of one day on the posed problems. This exemplifies that, while distributed and parallel solutions are often needed for scaling Machine Learning applications, a solely technical approach like an efficient distribution scheme is not necessarily enough — underlining the need for synergy of both domains. Nevertheless the scalability of this algorithm is very good, due to its low communication overhead, and a version specifically designed to converge fast in nearly unconstrained settings might result in a promising solution.

Also to mention is the effect of shrinking on the runtime and distribution efficiency. The algorithm of WW scales quadratically in the number classes, which is very large in our case, but due to the combination of shrinking and distribution techniques our solution yields reasonably good runtime performance. The drawback is that the interplay of both techniques is complex as shrinking effectively reduces the problem during training and thereby it reduces the advantage of distributing the optimization procedure. This interaction had significant impact on our solution and is another example for the complexity of applied, distributed Machine Learning.

7 Conclusion

We proposed distributed algorithms for solving the multi-class SVM formulations by Lee, Lin, and Wahba (LLW, 2004) and Weston and Watkins (WW, 1999). The algorithm addressing LLW takes advantage of an auxiliary variable, while our approach for optimizing WW in parallel is based on the 1-factorization problem from graph theory.

The experiments confirmed the correctness of the solvers, in the sense of an exact solver, and show linear speedup when the number of cores is increased. This speedup allows us to train LLW and WW on LSHTC datasets, and our analysis contributed by comparing MC-SVM formulations these rather large data sets, where such analysis was still lacking.

In comparison to OVR we showed that WW can achieve competitive classification results in less time, while still leading to a much sparser model. An interesting path for future work is to extend the formulation for multi-label problems and subsequently to inquiry if the algorithm can convince with good performance on even larger problem settings.

Unexpectedly, LLW shows clear disadvantages over the other MC-SVMs. Yet the favorable scaling properties might make further research interesting, for instance regarding the development of an unconstrained algorithm. We ease further research by publishing the source code under `https://github.com/albermax/xcsvm`.

Overcoming the limitations of a single machine, i.e. distribution, is a key problem and a key enabler in large scale learning. To best of our knowledge, we are the *first to train an exact, all-in-one MC-SVMs in a distributed manner*. We hope this first step inspires further research in this context.

Chapter IV

Efficient learning of kernel approximations

Kernel methods have strong theoretical foundations and proven their usefulness in numerous applications. With the increasing availability of data and the emerging of deep learning techniques a weakness of kernel methods got exposed, namely the quadratic dependence on the number of input samples. To alleviate this handicap random approximations of kernel projections have been proposed and constitute a middle ground between neural networks and kernel methods: they can be cast as a one-layer neural network where the basis layer's parameters are drawn *at random*. In this chapter we pose two critical questions: *Can the performance of such kernel machines be increased by adapting the basis' parameters either to the kernel function or to the task at hand, instead of using agnostic random sampling?* In order to answer this question we consider random feature kernel approximations as neural networks and subsequently show how they can be trained *end-to-end*. This enables us to, first, adapt the basis layer to specific kernel objectives and, second, directly to the task at hand. Our empirical evaluation suggests that the random parameterization is inefficient and that our more direct optimization approach can reduce the number of parameters by orders of magnitude, while still capturing general data properties. Most of this chapter's work was previously published in Alber et al. (2017a).

1 Introduction

Kernel methods are a major contribution to the field of Machine Learning. They exhibit strong theoretical foundations (Schölkopf et al., 1999; Müller et al., 2001; Schölkopf and Smola, 2002) and have been successfully embedded in many methods (E.g., Cortes and Vapnik, 1995; Mika et al., 1999; Baudat and Anouar, 2000; Schölkopf et al., 1998). The core idea of kernel functions is to rely on a similarity metric between two

input points and especially for tasks with a small number of training samples n this shows advantageous behavior. On the downside it hinders kernel methods to scale to problem settings with a large sample count n, because building on a metric between two points scales by design quadratically in n. To alleviate this issue many techniques have been proposed (E.g., Schölkopf et al., 1999; Williams and Seeger, 2000; Drineas and Mahoney, 2005) including random feature approximations (Rahimi and Recht, 2008, 2009). This last method approximates the similarity metric by using randomized basis functions and allows for new usage patterns by enabling kernel-based methods to be (re-)formulated efficiently in primal space.

Recent work on scaling kernel methods using random basis functions has shown that their performance on challenging tasks such as speech recognition can match closely those of deep neural networks (Lu et al., 2014; Dai et al., 2014; Wilson et al., 2016). However, research also highlighted two disadvantages of random basis functions. First, a large number of basis functions, i.e., features, is needed to obtain useful representations of the data. In a recent empirical study (Lu et al., 2014), a kernel machine matching the performance of a deep neural network required a much larger number of parameters. Second, a finite number of random basis functions leads to an inferior kernel approximation error that is *data-specific* (Rahimi and Recht, 2008; Sutherland and Schneider, 2015; Yang et al., 2012).

Deep neural networks learn representations that are adapted to the data using end-to-end training. Kernel methods on the other hand can only achieve this by selecting the optimal kernels to represent the data — a challenge that persistently remains. Furthermore, there are interesting cases in which learning with deep architectures is advantageous, as they require exponentially fewer examples (Montufar et al., 2014). Yet arguably both paradigms have the same modeling power as the number of training examples goes to infinity and empirical studies suggest that for real-world applications the advantage of one method over the other is somewhat limited (Lu et al., 2014; Dai et al., 2014; Wilson et al., 2016; Yang et al., 2015b).

Understanding the differences between approximated kernel methods and neural networks is crucial to use them optimally in practice. In particular, there are two aspects that require investigation: (1) How much performance is lost due to the kernel approximation error of the random basis? (2) What is the possible gain of adapting the features to the task at hand? Since these effects are expected to be data-dependent, we argue that an empirical study is needed to complement the existing theoretical contributions (Rahimi and Recht, 2008; Yang et al., 2012; Le et al., 2013; Sutherland and Schneider, 2015; Yu et al., 2016).

In this work, we investigate these issues by making use of the fact that approximated kernel methods can be cast as shallow neural networks with one hidden layer. The bottom layers of these networks are random basis functions that are generated in a *data-agnostic* manner and are *not adapted* during training (Rahimi and Recht, 2008, 2009; Le et al., 2013; Yu et al., 2016).

This stands in stark contrast to — even the conventional single hidden layer — neural network where the bottom-layer parameters are optimized with respect to the data distribution and the loss function. This so-called end-to-end learning has not been applied to random feature based networks before, because the optimization

process with trigonometric activation functions can be very unstable. In this work we contribute by adding an additional constant in the lower layer that stabilizes the optimization routine. It does not reduce the attainable function space and enables our approach to make very efficient use of the parameter space.
Then to pursue the stated research aims we constructed experiments that allow us to isolate the effect of the randomness of the basis and contrast it to data- and task-dependent adaptations. Specifically, we designed our experiments to distinguish four cases:

- **Random Basis (RB):** we use the (approximated) kernel machine in its traditional formulation (Rahimi and Recht, 2008; Yu et al., 2016).
- **Unsupervised Adapted Basis (UAB):** we adapt the basis functions to better approximate the true kernel function.
- **Supervised Adapted Basis (SAB):** we adapt the basis functions using kernel target alignment (Cristianini et al., 2001) to incorporate label information.
- **Discriminatively Adapted Basis (DAB):** we adapt the basis functions with a discriminative loss function, i.e., optimize jointly over basis and classifier parameters. This corresponds to conventional *neural network* optimization.

In particular, we want to gain insights in whether data-dependent basis functions (UAB and SAB) improve over data-agnostic basis functions (RB), and how well task-adapted basis functions (SAB) can perform in contrast to bases that result from end-to-end training (DAB).
The remainder is structured as follows. After a presentation of related work we explain approximated kernel machines in context of neural networks in Section 3 and describe our propositions in Section 4. In Section 5 we quantify the benefit of adapted basis function in contrast to their random counterparts empirically. Finally, we give a critical discussion in Section 6 and conclude in Section 7.

2 Related work

Kernel approximations and their applications To overcome the limitations of kernel learning, several approximation methods have been proposed. In addition to Nyström methods (Williams and Seeger, 2000; Drineas and Mahoney, 2005), random Fourier features (Rahimi and Recht, 2008, 2009) have gained a lot of attention. Random features or (faster) enhancements (Le et al., 2013; Feng et al., 2015; Yu et al., 2015, 2016) were successfully applied in many applications (Dai et al., 2014; Lu et al., 2014; Huang et al., 2014; Wilson et al., 2016), and were theoretically analyzed (Yang et al., 2012; Sutherland and Schneider, 2015). They inspired scalable approaches to learn kernels with Gaussian processes (Wilson et al., 2016; Yang et al., 2015a; Làzaro-Gredilla et al., 2010). Similarly to our approach Yu et al. (2015) explores ways to optimize the random features basis. This work presents an alternating optimization scheme, which is in contrast to our end-to-end adaptation. Other recent work (Rudi and Rosasco, 2017; Carratino et al., 2018) focuses on optimizing random features

without adapting the basis itself. Another way to use the kernel framework in primal space are (sub-sampled) empirical kernel maps (Schölkopf et al., 1999).

On the intersection between kernel methods and neural networks Notably, Montavon et al. (2011) explores the connection between kernel methods and neural networks by modeling the internal representations in a kernel framework and Cho and Saul (2009) examines the connection by creating kernels with neural network inspired feature functions. In this chapter we build upon the theory in Cho and Saul (2009). The field of RBF-networks explores basis functions with an Gaussian activation function, e.g., Moody and Darken (1989) and Müller et al. (1999) show how to efficiently adapt them to data. Next to random features approximations there exists some exploratory research body (E.g., Sakhnini et al., 1999; Zuo et al., 2009; Gashler and Ashmore, 2014; Chan et al., 2018) on trigonometric activation functions.

Distinction Our work contributes in several ways: we view kernel machines from a neural network perspective and delineate the influence of different adaptation schemes. None of the above does this. The related work of Yang et al. (2012) compares the data-dependent Nyström approximation to random features. While our approach generalizes to structured matrices, i.e., fast kernel machines, Nyström does not. Most similar to our work is Yang et al. (2015b). They interpret the Fastfood kernel approximation as a neural network. Their aim is to reduce the number of parameters in a convolutional neural network, while we try to shed light on the connection between approximated kernel machines and kernels. They also use dropout inside the kernel approximation, which further limits the comparability to this analysis.

3 Casting kernel approximations as shallow, random neural networks

Recall from Section 1.2.2 that kernels are pairwise similarity functions $k(x, x') : \mathbb{R}^d \times \mathbb{R}^d \mapsto \mathbb{R}$ between two data points $x, x' \in \mathbb{R}^d$. They are equivalent to the inner-products in an intermediate, potentially infinite-dimensional feature space produced by a function $\phi : \mathbb{R}^d \mapsto \mathbb{R}^D$

$$k(x, x') = \phi(x)^T \phi(x') \tag{IV.1}$$

Non-linear kernel machines typically avoid using ϕ explicitly by applying the kernel trick. They work in the dual space with the (Gram) kernel matrix. This imposes a quadratic dependence on the number of samples n and prevents its application in large scale settings. Several methods have been proposed to overcome this limitation by approximating a kernel machine with the following functional form

$$f(x) = W^T \hat{\phi}(x) + b, \tag{IV.2}$$

where $\hat{\phi}(x)$ is the approximated kernel feature map. Now, we will explain how to obtain this approximation for the Gaussian and the ArcCos kernel (Cho and Saul,

2009). We chose the Gaussian kernel because it is the default choice for many tasks. On the other hand, the ArcCos kernel yields an approximation consisting of rectified, piece-wise linear units (ReLU) as used in deep learning (Nair and Hinton, 2010; Glorot et al., 2011; Krizhevsky et al., 2012).

Gaussian kernel To obtain the approximation of the Gaussian kernel, we use the following property (Rahimi and Recht, 2008). Given a smooth, shift-invariant kernel $k(x - x') = k(z)$ with Fourier transform $p(w)$, then:

$$k(z) = \int_{R^d} p(w) e^{jw^T z} dw. \tag{IV.3}$$

Using the Gaussian distribution $p(w) = \mathcal{N}(0, \sigma^{-1})$, we obtain the Gaussian kernel

$$k(z) = \exp^{-\frac{\|z\|_2^2}{2\sigma^2}}.$$

Thus, the kernel value $k(x, x')$ can be approximated by the inner product between $\hat{\phi}(x)$ and $\hat{\phi}(x')$, where $\hat{\phi}$ is defined as

$$\hat{\phi}(x) = \sqrt{\frac{1}{D}}[\sin(W_B^T x), \cos(W_B^T x)] \tag{IV.4}$$

and $W_B \in \mathbb{R}^{d \times D/2}$ as a random matrix with its entries drawn from $\mathcal{N}(0, \sigma^{-1})$. The resulting features are then used to approximate the kernel machine with the implicitly infinite dimensional feature space,

$$k(x, x') \approx \hat{\phi}(x)^T \hat{\phi}(x'). \tag{IV.5}$$

ArcCos kernel To yield a better connection to state-of-the-art neural networks we use the ArcCos kernel (Cho and Saul, 2009)

$$k(x, x') = \frac{1}{\pi} \|x\| \left\|x'\right\| J(\theta)$$

with $J(\theta) = (\sin\ \theta + (\pi - \theta)\cos\ \theta)$ and $\theta = \cos^{-1}(\frac{x \cdot x'}{\|x\|\|x'\|})$, the angle between x and x'. The approximation is not based on a Fourier transform, but is given by

$$\hat{\phi}(x) = \sqrt{\frac{1}{D}} \max(0, W_B^T x) \tag{IV.6}$$

with $W_B \in \mathbb{R}^{d \times D}$ being a random Gaussian matrix. This makes the approximated feature map of the ArcCos kernel closely related to ReLUs in deep neural networks.

Neural network interpretation The approximated kernel features $\hat{\phi}(x)$ can be interpreted as the output of the hidden layer in a shallow neural network. To obtain the neural network interpretation, we rewrite Equation IV.2 as the following

$$f(x) = W^T h(W_B^T x) + b, \tag{IV.7}$$

with $W \in \mathbb{R}^{D \times \mathcal{C}}$ with $\mathcal{C}$ number of classes, and $b \in \mathbb{R}^{\mathcal{C}}$. Here, the non-linearity h corresponds to the obtained kernel approximation map. Now, we substitute $z = W_B^T x$ in Equations. IV.4 and IV.6 yielding $h(z) = \sqrt{1/D}[\sin(z), \cos(z)]^T$ for the Gaussian kernel and $h(z) = \sqrt{1/D}\max(0, z)$ for the ArcCos kernel.

4 Adaptation of random kernel approximations

Having introduced the neural network interpretation of random features, the key difference between both methods is which parameters are trained. For the neural network, one optimizes the parameters in the bottom-layer and those in the upper layers jointly. For kernel machines, however, W_B is fixed, i.e., the features are not adapted to the data. Hyper-parameters (such as σ defining the bandwidth of the Gaussian kernel) are selected with cross-validation or heuristics (Gretton et al., 2005; Dai et al., 2014; Yu et al., 2016). Consequently, the basis is not directly adapted to the data, loss, and task at hand.
In our experiments, we consider the classification setting (see Section 1.1.1) where for the given data $X \in \mathbb{R}^{n \times d}$ containing n samples with d input dimensions one seeks to predict the target labels $Y \in [0, 1]^{n \times \mathcal{C}}$ with a one-hot encoding for $\mathcal{C}$ classes. We use accuracy as the performance measure and the multinomial-logistic loss as its surrogate. All our models have the same, generic form shown in Equation IV.7. However, we use different types of basis functions to analyze varying degrees of adaptation. In particular, we study whether data-dependent basis functions improve over data-agnostic basis functions. On top of that, we examine how well label-informative, thus task-adapted basis functions can perform in contrast to the data-agnostic basis. Finally, we use end-to-end learning of all parameters to connect to neural networks.

Random Basis - RB: For data-agnostic kernel approximation, we use the current state-of-the-art of random features. Orthogonal random features (Yu et al., 2016, ORF) improve the convergence properties of the Gaussian kernel approximation over random Fourier features (Rahimi and Recht, 2008, 2009). The ArcCos kernel is applied as described above.
We also use these features as initialization of the following adaptive approaches.

Unsupervised Adapted Basis - UAB: While the introduced random bases converge towards the true kernel with an increasing number of features, it is to be expected that an optimized approximation will yield a more compact representation. We address this by optimizing the sampled parameters W_B w.r.t. the kernel approximation

error (KAE):

$$\hat{L}(x, x') = \frac{1}{2}(k(x, x') - \hat{\phi}(x)^T \hat{\phi}(x'))^2 \tag{IV.8}$$

This objective is kernel- and data-dependent, but agnostic to the classification task.

Supervised Adapted Basis - SAB: As an intermediate step between task-agnostic kernel approximations and end-to-end learning, we propose to use kernel target alignment (Cristianini et al., 2001) to inject label information. This is achieved by a target kernel function k_Y with $k_Y(x, x') = +1$ if x and x' belong to the same class and $k_Y(x, x') = 0$ otherwise. We maximize the alignment between the approximated kernel k and the target kernel k_Y for a given data set X:

$$\hat{A}(X, k, k_Y) = \frac{\langle K, K_Y \rangle}{\sqrt{\langle K, K \rangle \langle K_Y, K_Y \rangle}} \tag{IV.9}$$

with $\langle K_a, K_b \rangle = \sum_{i,j}^{n} k_a(x_i, x_j) k_b(x_i, x_j)$.

Discriminatively Adapted Basis - DAB: The previous approach uses label information, but is oblivious to the final classifier. On the other hand, a discriminatively adapted basis is trained jointly with the classifier to minimize the classification objective, i.e., W_B, W, b are optimized at the same time. This is the end-to-end optimization performed in neural networks.

Optimizing the basis The periodic structure of trigonometric functions makes their optimization a challenge. Our proposition to overcome this hurdle is to model the bandwidth scale $1/\sigma$ explicitly as a constant. Practically, we substitute W_B with $1/\sigma\ G_B$ in Equation IV.4, sample $G_B \in \mathbb{R}^{d \times D/2}$ from $\mathcal{N}(0, 1)$ and orthogonalize the matrix as given in Yu et al. (2016) to approximate the Gaussian kernel. For adapting schemes UAB, SAB, and DAB we only update G_B and keep $1/\sigma$ fixed. The hyper-parameter σ can be reliably set using the heuristics from Yu et al. (2015, 2016). The resulting mapping looks as follows:

$$\hat{\phi}(x) = \sqrt{\frac{1}{D}}[\sin(1/\sigma\ G_B^T x), \cos(1/\sigma\ G_B^T x)] \tag{IV.10}$$

Note, as we do not directly regularize the weights, the model can still assume the same parameter space as without this re-parameterization, but this change allows for a stable (end-to-end) optimization process.

5 Experiments

In the following, we present the empirical results of our study, starting with a description the experimental setup. Then, we proceed to present the results of using data-dependent and task-dependent basis approximations. In the end, we bridge our analysis to deep learning and fast kernel machines.

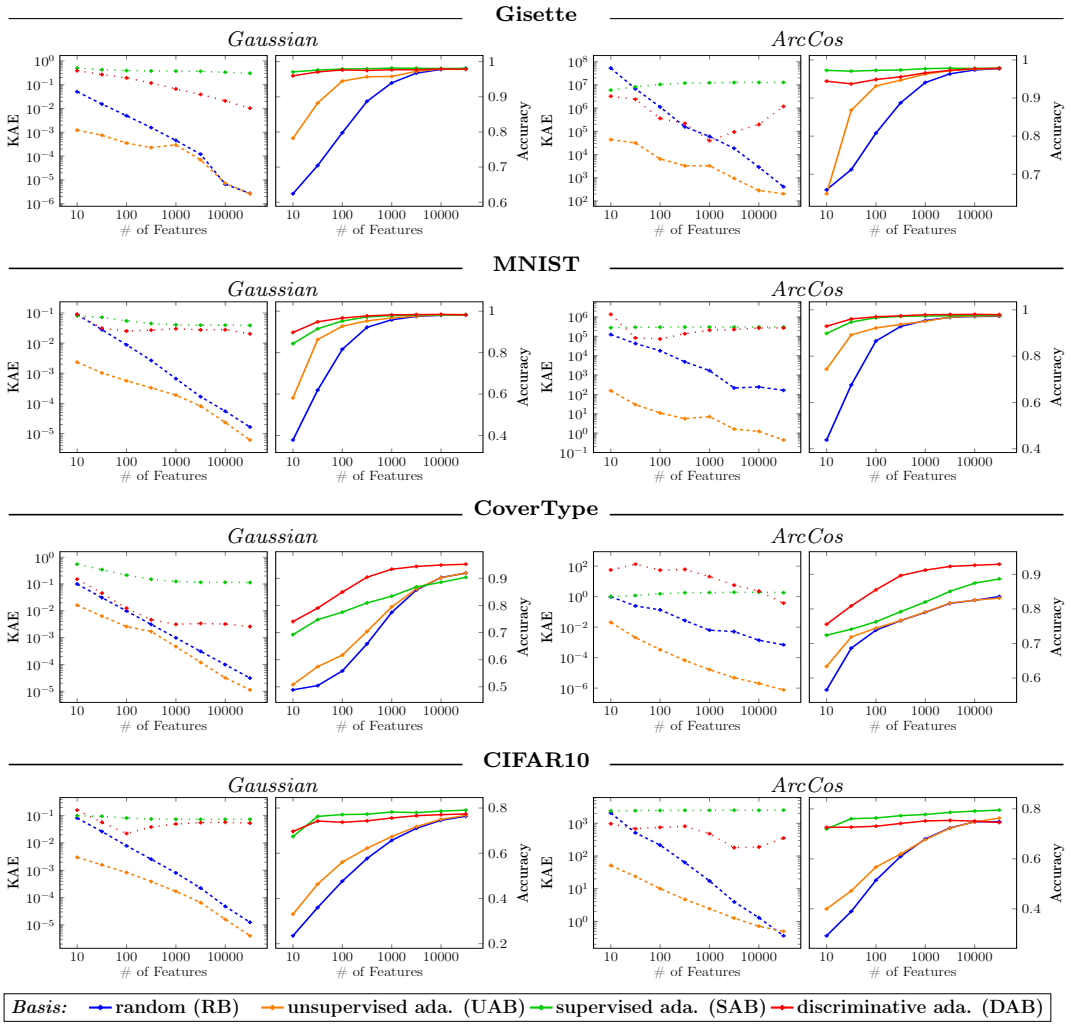

Fig. IV.1: Adapting bases. *The plots show the relationship between the number of features (x-axis), the KAE in logarithmic spacing (**left, dashed lines**) and the classification error (**right, solid lines**). Typically, the KAE decreases with a higher number of features, while the accuracy increases. The KAE for SAB and DAB (orange and red dotted line) hints how much the adaptation deviates from its initialization (blue dashed line). Best viewed in digital and color.*

5.1 Experimental setup

We used the following seven data sets for our study: Gisette (Guyon et al., 2004), MNIST (LeCun et al., 1998b), CoverType (Blackard and Dean, 2000), CIFAR10 features from (Coates et al., 2011), Adult (Kohavi, 1996), Letter (Frey and Slate, 1991), USPS (Hull, 1994). The results for the last three can be found in the supplement. We center the data sets and scale them feature-wise into the range $[-1, +1]$. We use validation sets of size $1,000$ for Gisette, $10,000$ for MNIST, $50,000$ for CoverType, $5,000$ for CIFAR10, $3,560$ for Adult, $4,500$ for Letter, and $1,290$ for USPS. We repeat

every test three times and report the mean over these trials and we found the results to be consistent and reliable across datasets, methods and number of features. Therefore we refrain from further statistical hypotheses tests.

Optimization We train all models with mini-batch stochastic gradient descent. The batch size is 64 and as update rule we use ADAM (Kingma and Adam, 2015). We use early-stopping where we stop when the respective loss on the validation set does not decrease for ten epochs. We use Keras (Chollet, 2015), Scikit-learn (Pedregosa et al., 2011a), NumPy (Van Der Walt et al., 2011) and SciPy (Jones et al., 2014) for our implementation. We set the hyper-parameter σ for the Gaussian kernel heuristically according to Yu et al. (2015, 2016).
The UAB and SAB learning problems scale quadratically in the number of samples n. Therefore, to reduce memory requirements we optimize by sampling mini-batches from the kernel matrix. A batch for UAB consists of 64 sample pairs x and x' as input and the respective value of the kernel function $k(x, x')$ as target value. Similarly for SAB, we sample 64 data points as input and generate the target kernel matrix as target value. For each training epoch we randomly generate $10,000$ training and $1,000$ validation batches, and evaluate the performance on $1,000$ unseen, random batches.

5.2 Analysis

Table IV.1 gives an overview of the best performances achieved by each basis on each data set.

	Gaussian				*ArcCos*			
Dataset	**RB**	**UAB**	**SAB**	**DAB**	**RB**	**UAB**	**SAB**	**DAB**
Gisette	98.1	97.9	98.1	97.9	97.7	97.8	97.8	97.8
MNIST	98.2	98.2	98.3	98.3	97.2	97.4	97.7	97.9
CoverType	91.9	91.9	90.4	95.2	83.6	83.1	88.7	92.9
CIFAR10	76.4	76.8	79.0	77.3	74.9	76.3	79.4	75.3

Tab. IV.1: Classification performance. *Best accuracy in % for different bases.*

Data-adapted kernel approximations First, we evaluate the effect of choosing a data-dependent basis (UAB) over a random basis (RB). In Figure IV.1, we show the kernel approximation error (KAE) and the classification accuracy for a range from 10 to 30,000 features (in logarithmic scale). The first striking observation is that a data-dependent basis can approximate the kernel equally well with up to two orders of magnitude fewer features compared to the random baseline. This holds for both the Gaussian and the ArcCos kernel. However, the advantage diminishes as the number of features increases. When we relate the kernel approximation error to the accuracy, we observe that initially a decrease in KAE correlates well with an increase in accuracy. However, once the kernel is approximated sufficiently well, using more feature does not impact accuracy anymore. Taking the difficulty of the classification problem

into account — the predictors classify Gisette and MNIST better than CoverType and CIFAR10 — the trend suggests that UAB leads to more compact and accurate representations when tasks are simpler, yet more datasets would need to be examined to make a conclusive statement.
We conclude that the choice between a random or data-dependent basis strongly depends on the application. When a short training procedure is required, optimizing the basis could be too costly. On the other hand, if the focus lies on fast inference, we argue to optimize the basis to obtain a compact representation. In settings with restricted resources, e.g., mobile devices, this can be a key advantage.

Task-adapted kernels A key difference between kernel methods and neural networks originates from the training procedure. In kernel methods the feature representation is fixed while the classifier is optimized. In contrast, deep learning relies on end-to-end training such that the feature representation is tightly coupled to the classifier. Intuitively, this allows the representation to be tailor-made for the task at hand. Therefore, one would expect that this allows for an even more compact representation than the previously examined data-adapted basis.
In Section 4 we proposed a task-adapted kernel (SAB). Figure IV.1 shows that the approach is comparable in terms of classification accuracy to discriminatively trained basis (DAB). Only for CoverType data set SAB performs significantly worse due to the limited model capabilities, which we will discuss below. Both task-adapted features improve significantly in accuracy compared to the random and data-adaptive kernel approximations.

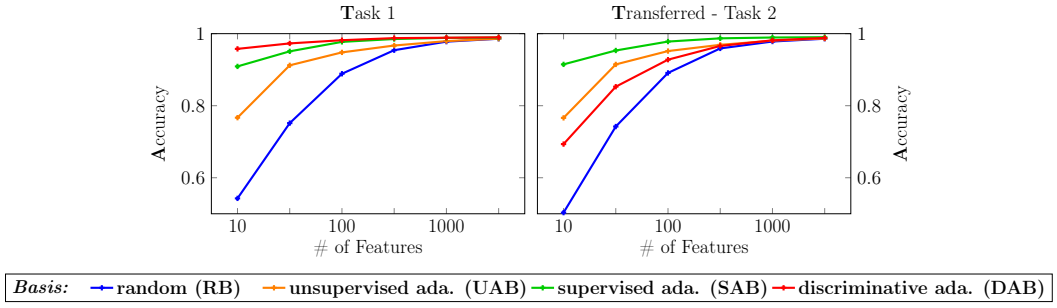

Fig. IV.2: Transfer learning. *We train to discriminate a random subset of* 5 *classes on the MNIST data set (**left**) and then transfer the basis function to a new task (**right**), i.e., train with the fixed basis from task 1 to classify between the remaining classes.*

Transfer learning The beauty of kernel methods is, however, that a kernel function can be used across a wide range of tasks and consistently result in good performance. Therefore, in the next experiment, we investigate whether the resulting kernel retains this generalization capability when it is task-adapted. To investigate the influence of task-dependent information, we randomly separate the classes MNIST into two distinct subsets. The first task is to classify five randomly samples classes and their

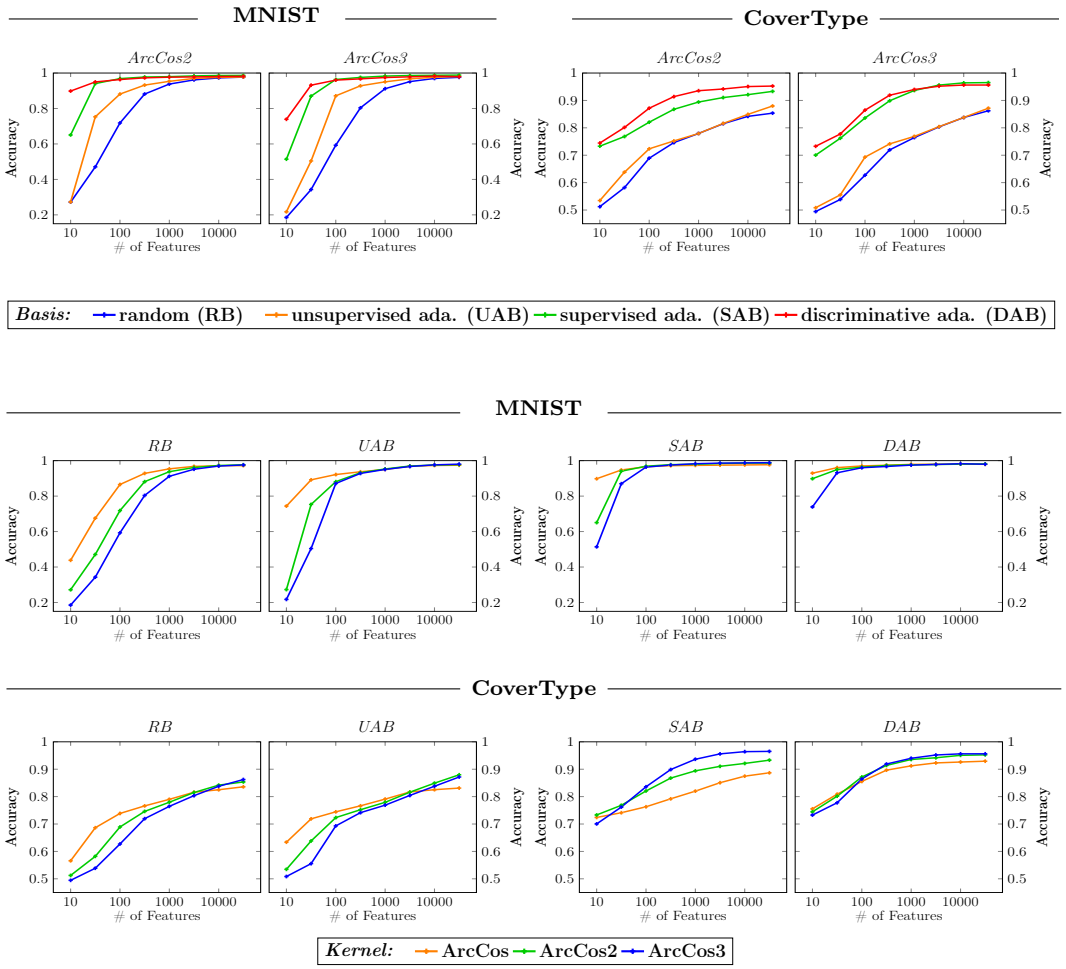

Fig. IV.3: Deep kernel machines. *The plots show the classification performance of the ArcCos-kernels with respect to the kernel (**first part**) and with respect to the number of layers (**second part**). Best viewed in digital and color.*

respective data points, while the second task is to do the same with the remaining classes. We train the previously presented model variants on task 1 and transfer their bases to task 2 where we only learn the classifier. The experiment is repeated with five different splits and the mean accuracy is reported.

Figure IV.2 shows that on the transfer task, the random and the data-adapted bases RB and UAB approximately retain the accuracy achieved on task 1. The performance of the end-to-end trained basis DAB drops significantly, however, yields still a better performance than the default random basis. Surprisingly, the supervised basis SAB using kernel-target alignment retains its performance and achieves the highest accuracy on task 2. This shows that using label information can indeed be exploited in order to improve the efficiency and performance of kernel approximations without having to sacrifice generalization. I.e., a target-driven kernel (SAB) can be an efficient and still general alternative to the universal Gaussian kernel.

Deep kernel machines We extend our analysis and draw a link to deep learning by adding two deep kernels (Cho and Saul, 2009). As outlined in the aforementioned paper, stacking a Gaussian kernel is not useful, instead we use ArcCos kernels that are related to deep learning as described below. Recall the ArcCos kernel from Eq. 3 as $k_1(x, x')$. Then the kernels ArcCos2 and ArcCos3 are defined by the inductive step $k_{i+1}(x, x') = \frac{1}{\pi}[k_i(x, x)k_i(x', x')]^{-1/2}J(\theta_i)$ with $\theta_i = \cos^{-1}(k_i(x, x')[k_i(x, x)k_i(x', x')]^{-1/2})$. Similarly, the feature map of the ArcCos kernel is approximated by a one-layer neural network with the ReLU-activation function and a random weight matrix W_B

$$\hat{\phi}_{ArcCos}(x) = \hat{\phi}_B(x) = \sqrt{\frac{1}{D}} \max(0, W_B^T x), \tag{IV.11}$$

and the feature maps of the ArcCos2 and ArcCos3 kernels are then given by a 2- or 3-layer neural network with the ReLU-activations, i.e., $\hat{\phi}_{ArcCos2}(x) = \hat{\phi}_{B_1}(\hat{\phi}_{B_0}(x))$ and $\hat{\phi}_{ArcCos3}(x) = \hat{\phi}_{B_2}(\hat{\phi}_{B_1}(\hat{\phi}_{B_0}(x)))$. The training procedure for the ArcCos2 and ArcCos3 kernels remains identical to the training of the ArcCos kernel, i.e., the random matrices W_{B_i} are simultaneously adapted. Only, now the basis consists of more than one layer, and, to remain comparable for a given number of features, we split these features evenly over two layers for a 2-layer kernel and over three layers for a 3-layer kernel.

In the following we describe our results on the MNIST and CoverType data sets. We observed that the so far described relationship between the cases RB, UAB, SAB, DAB also generalizes to deep models (see Fig. IV.3, first part, and Fig. A.3 in the supplement). I.e., UAB approximates the true kernel function up to several magnitudes better than RB and leads to a better resulting classification performance. Furthermore, SAB and DAB perform similarly well and clearly outperform the task-agnostic bases RB and UAB.

We now compare the results across the ArcCos-kernels. Consider the third row of Fig. IV.3, which depicts the performance of RB and UAB on the CoverType data set. For up to $3,000$ features, the deeper kernels perform worse than the shallow ones. Only given enough capacity the deep kernels are able to perform as good as or better than the single-layer bases. On the other hand, for the CoverType data set, task related bases, i.e., SAB and DAB, benefit significantly from a deeper structure and are thus more efficient. Comparing SAB with DAB, for the ArcCos kernel with only one layer SAB leads to worse results than DAB. Given two layers the gap diminishes and vanishes with three layers (see Fig. IV.3). This suggests that for this data set the evaluated shallow models are not expressive enough to extract the task-related kernel information.

Fast kernel machines By using structured matrices one can speed up approximated kernel machines (Le et al., 2013; Yu et al., 2016). We will now investigate how this technique influences the presented basis schemes. The approximation is achieved by replacing random Gaussian matrices with an approximation composed of diagonal and structured Hadamard matrices. The advantage of these matrix types is that they allow for low storage costs as fast multiplications. Recall that the input dimension

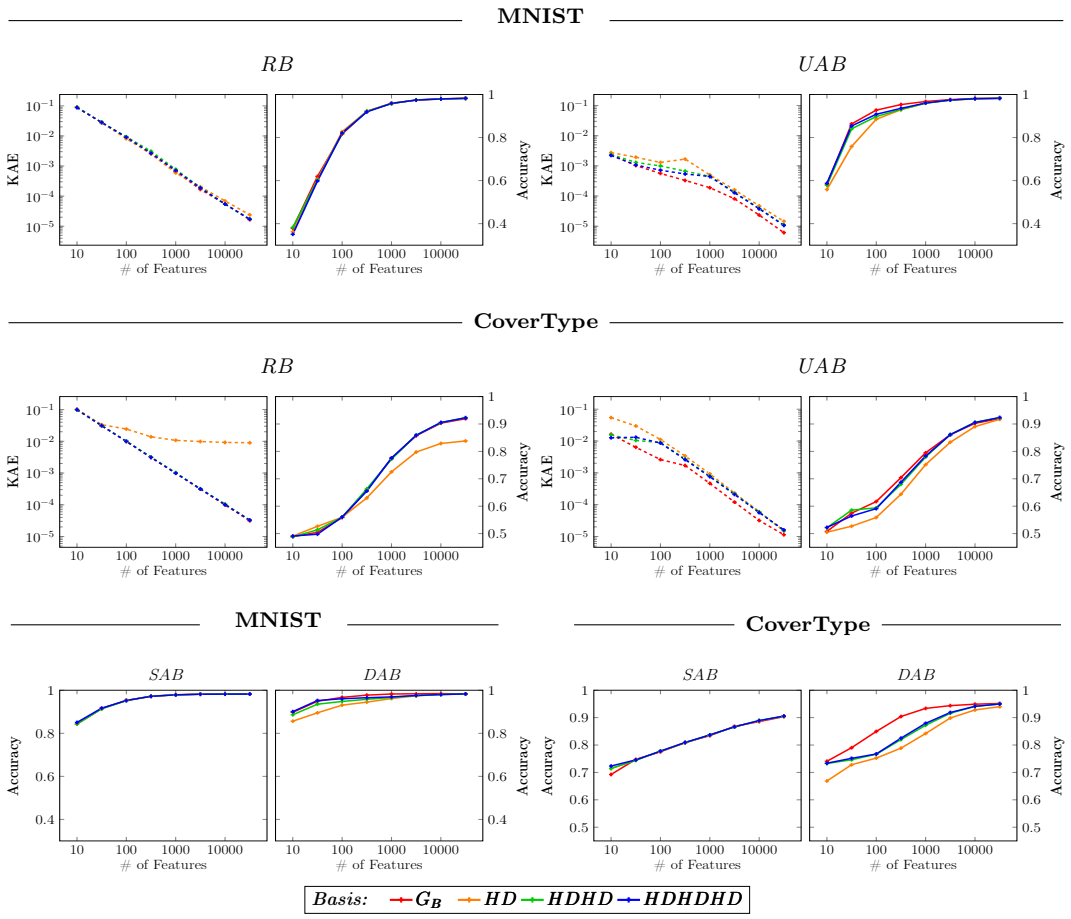

Fig. IV.4: Fast kernel machines. *The plots show how replacing the basis G_B with an fast approximation influences the performance of a Gaussian kernel. I.e., G_B is replaced by 1, 2, or 3 structured blocks HD_i. Fast approximations with 2 and 3 blocks might overlap with G_B. Best viewed in digital and color.*

is d and the number of features is D. By using the fast Hadamard-transform these algorithms only need to store $O(D)$ instead of $O(dD)$ parameters and the kernel approximation can be computed in $O(D \log d)$ rather than $O(Dd)$.

We use the approximation from Yu et al. (2016) and replace the random Gaussian matrix $W_B = 1/\sigma\ G_B$ in Eq. IV.4 with a chain of random, structured blocks $W_B \approx 1/\sigma\ HD_1\ \ldots\ HD_i$. Each block HD_i consists of a diagonal matrix D_i with entries sampled from the Rademacher distribution and a Hadamard matrix H. More blocks lead to a better approximation, but consequently require more computation. We found that the optimization is slightly more unstable and therefore stop early only after 20 epochs without improvement. When adapting a basis we will only modify the diagonal matrices.

We re-conducted our previous experiments for the Gaussian kernel on the MNIST and CoverType data sets (Fig. IV.4). In the first place one can notice that in most cases the approximation exhibits no decline in performance and that it is a viable alternative

for all basis adaption schemes. There are two major exceptions: Consider first the left part of the second row which depicts a approximated, random kernel machine (RB). The convergence of the kernel approximation stalls when using a random basis with only one block. As a result the classification performance drops drastically. This is not the case when the basis is adapted unsupervised, which is given in the right part of the second row. Here one cannot notice a major difference between one or more blocks. This means that for fast kernel machines an unsupervised adaption can lead to a more effective model utilization, which is crucial for resource aware settings. Furthermore, a discriminatively trained basis such as a neural network, can be effected similarly from this re-parameterization (see Fig. IV.4, bottom row). Here an order of magnitude more features is needed to achieve the same accuracy compared to an exact representation, regardless how many blocks are used. In contrast, when adapting the kernel in a supervised fashion no decline in performance is noticeable. This shows that this procedure uses parameters very efficiently.

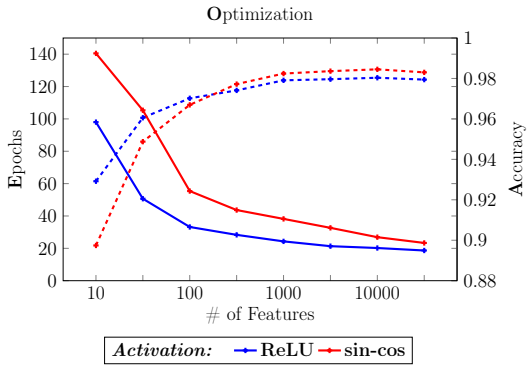

Fig. IV.5: Optimization. *Comparison of the optimization duration (**solid**) in epochs of the cos-sin and the ReLu non-linearity given a varying number of features on the MNIST benchmark. For reference the obtained accuracies are plotted as **dashed** lines.*

Optimization In general, the periodic nature of the sine and cosine function imposes limitations to applicability in neural networks. When scaled and initialized properly we found that the activation function in Equation IV.4 can be optimized and reaches similar performances as a ReLU-powered neural network. From Figure IV.5 one can see that the *sin-cos* activation function needs only a reasonable amount of epochs more to converge than with the ReLU activation.

6 Discussion

Compared to neural networks (Gaussian) kernel methods have the drawback that their application is limited to classification problems with a suitable feature space. Random feature approximations and subsequently our method inherit this disadvantage. Whereas neural network are known for their capabilities to learn such spaces,

kernel methods do not. In principle these two methods could be combined as, e.g., Yang et al. (2015b) uses neural network as feature extractor and as classifier based on random features. A limitation when jointly optimizing the feature extractor and the classifier can be that the input distribution to the classifier varies and makes choosing suitable hyper-parameter, such as σ, challenging. In general, our approach to set the hyper-parameter σ to be a constant can be a limitation for future applications.
Furthermore, the variants UAB and SAB adapt to a kernel function and consequently scale quadratically with the number of samples n, because there are n^2 target values to learn. Yet UAB and SAB are optimized with a mini-batch scheme and training times can be adjusted to the actual complexity of the target kernel function. Therefore the learning process can be adapted to a time or resource budget.
Our work suggests that drawing parameters at random is wasteful, yet knowing that a Gaussian kernel can only be approximated with an infinite number of features (Rahimi and Recht, 2008; Sutherland and Schneider, 2015; Yang et al., 2012), one might wonder how our approaches can yield good performance with a small number of features. The first reason is that the Gaussian kernel bases (RB and UAB) represent a much more complex feature function than is needed for the ultimate task at hand. Compared to them, SAB and DAB are tailored to the classification problem and the complexity of the SAB is implicitly and of the DAB is explicitly bounded by the task at hand — thus they can be much more efficient. This in concordance with the findings of Braun et al. (2008), who show that information needed to fulfill the task at hand is typically contained in a low number of the leading PCA dimensions of the kernel feature space. We note that traditional kernel methods circumvent this expansion of the (infinite) feature space by taking advantage of the kernel trick (Müller et al., 2001; Schölkopf and Smola, 2002), while random features and other approximation schemes do not by design.
Especially the SAB variant shows a good compromise between generalization and classification performance and has the potential to fuse the benefits of kernel methods and end-to-end training of neural networks. E.g., in domains with limited number of data samples this can be a promising way to leverage the data-efficiency of kernel methods and the feature-learning capabilities of neural networks.

7 Conclusion

Our analysis shows how random and adaptive bases affect the quality of learning. For random features this comes with the need for a large number of features and suggests that two issues severely limit approximated kernel machines: the basis being (1) agnostic to the data distribution and (2) agnostic to the task. To inquiry this we proposed an efficient optimization scheme for approximated kernel machines that allows to train them like neural networks end-to-end.
In our analysis we have found that data-dependent optimization of the kernel approximation consistently results in a more compact representation for a given kernel approximation error. The size of the basis could be reduced by up to two orders of magnitude. Moreover, task-adapted features could further improve upon this. Even with fast, structured matrices, the adaptive features allow to further reduce the num-

ber of required parameters. This presents a promising strategy when a fast and computationally cheap inference is required, e.g., on mobile devices.
Beyond that, we have evaluated the generalization capabilities of the adapted variants on a transfer learning task. Remarkably, all adapted bases outperform the random feature baseline here. We have found that the kernel-target alignment works particularly well in this setting, having almost the same performance on the transfer task as the target task. At the junction of kernel methods and deep learning, this shows that incorporating label information can indeed be beneficial for performance without having to sacrifice generalization capability. Investigating this in more detail appears to be highly promising and suggests the path for future work.

Chapter V

Efficient software for prediction analysis

The ability to adapt to elaborate data distributions makes deep neural networks powerful learning machines. However, design, training, and implementation of them can pose serious challenges as research is still far from a thorough understanding of the inner workings of such models. The community proposed prediction analysis as one way to remedy this. The idea is to highlight "important" features in the input domain, and among these algorithms propagation-based techniques pose a promising direction. They allow for fast execution time and have the ability to take advantage of high-level features in their analysis, but efficient software is lacking for many methods and for emerging network structures. The research aim of this chapter is therefore: *To reveal, develop and implement approaches for efficient implementations of propagation-based prediction analysis with an emphasis on the usability for the non-expert user and facilitating research on emerging network architectures.* Based on an review of the state-of-the-art, we will motivate our aim and then introduce the software package *iNNvestigate* which is — in a broader sense — the essence of this chapter's work. It is designed to provide an intuitive interface to users, which facilitates the application and comparability of the discussed methods. The modular approach of the library allowed us to efficiently implement a wide range of methods and subsequently compare them on different network architectures. Moreover, we give examples how our work stimulates research efforts and questions. Most of the work in this chapter is based on Alber et al. (2018b, 2019) and Alber (2019).

1 Introduction

Recent developments showed that neural networks can be applied successfully in many technical applications like computer vision (E.g., Krizhevsky et al., 2012; He et al., 2016; LeCun et al., 2015), speech synthesis (E.g., Van Den Oord et al., 2016) and translation (E.g., Vaswani et al., 2017; Sutskever et al., 2014; Bahdanau et al., 2015). Inspired by such successes many more domains use Machine Learning and specifically deep neural networks for, e.g., material science and quantum physics (E.g., Montavon et al., 2013; Schütt et al., 2017b,a; Chmiela et al., 2017, 2018), cancer research (E.g., Binder et al., 2018; Korbar et al., 2017), strategic games (E.g., Silver et al., 2016, 2017), knowledge embeddings (E.g., Mikolov et al., 2013; Pennington et al., 2014; Alber et al., 2017a), and even for automatic Machine Learning (E.g., Zoph et al., 2018; Alber et al., 2018a). With this broader application focus the requirements beyond predictive power alone rise. One key requirement in this context is the ability to understand and interpret predictions made by a neural network or generally by a learning machine. In at least two areas this ability plays an important role: domains that require an understanding because they are intrinsically critical or because it is mandatory by law, and domains that strive to extract knowledge beyond the predictions of learned models. As exemplary domains can be named: health care (E.g., Binder et al., 2018; Korbar et al., 2017; Gondal et al., 2017), applications affected by laws like the GDPR (Voigt and Bussche, 2017), and natural sciences (E.g., Montavon et al., 2013; Schütt et al., 2017b,a; Chmiela et al., 2017, 2018).

The advancement of deep neural networks is based on their potential to leverage complex and structured data by learning complicated inference processes. This makes a better understanding of such models challenging, yet a rewarding target. Various approaches to tackle this problem have been developed, e.g., Baehrens et al. (2010), Bach et al. (2015), Montavon et al. (2017), Springenberg et al. (2015), Smilkov et al. (2017), Sundararajan et al. (2017), Shrikumar et al. (2017), Kindermans et al. (2018), Zeiler and Fergus (2014), Zintgraf et al. (2017), Ribeiro et al. (2016), Lundberg and Lee (2017), and Selvaraju et al. (2017). While the nature and objectives of explanation algorithms can be ambiguous (Lipton, 2016), in practice gaining specific insights can already enable practitioners and researchers to create knowledge as first promising results show, e.g., Lapuschkin et al. (2016a, 2017, 2019), Binder et al. (2018), Zintgraf et al. (2017), and Sundararajan et al. (2017).

In particular, we focus on techniques to analyze the predictions of neural networks that aim to highlight "important" features in the input space of a targeted neural network (Montavon et al., 2017). For the rest of this chapter we will use the terms "explaining" and "explanation" to refer to the process or the result of such methods.

Given the potential of such approaches, the existence and an understanding of standard usage patterns and, especially, standard software is of particular importance to facilitate the transition of explanation methods from research into widespread application domains. This, on one hand, lowers the application barrier and effort for non-expert users and, on the other hand, it allows experts to focus on algorithm customization and research.

Our work supports this development by extracting, explaining, and implementing patterns for efficient implementations of propagation-based explanation algorithms — and ultimately contributing the software package *iNNvestigate* to the research community. It features a clear interface for users and many implemented explanation methods as well as modular backend for developers of new algorithms. The software is efficient due the chosen abstractions and a seamless integration into deep learning frameworks. In this chapter we will present a selection of application patterns and first results on how our contribution stimulated research for explanation methods. Moreover, we will outline challenges and research questions for future explanation algorithms and software.
The content of this chapter is organized as follows. In the next section we discuss related work and specify the aim of our effort in more detail. In Section 3 we will present the software *iNNvestigate*, describe how it is designed and how it is used to implement explanation algorithms. In Section 4 we showcase applications of the software and in Section 5 we will provide a discussion on limitations as well as further challenges. Eventually, we give a conclusion in Section 6.

2 Related work and aims

With the rise of deep neural networks Machine Learning models are getting more and more evolved — and accordingly such learning machines become less comprehensible. Research proposed many approaches to tackle this shortcoming, e.g., Haufe et al. (2014), Zeiler and Fergus (2014), Montavon et al. (2018), Nguyen et al. (2016), Mordvintsev et al. (2015), Ribeiro et al. (2016), and Ancona et al. (2018). Among these attempts, prediction analysis received particular attention (E.g., Baehrens et al., 2010; Bach et al., 2015; Montavon et al., 2017; Springenberg et al., 2015; Smilkov et al., 2017; Sundararajan et al., 2017; Shrikumar et al., 2017; Kindermans et al., 2018; Zeiler and Fergus, 2014; Selvaraju et al., 2017). This algorithm family differs in that it tries to "explain" a single prediction of a neural network. Typically this entails emphasizing or highlighting presumably important features in the input space (Montavon et al., 2017).
More specifically we focus in this work on propagation-based algorithms (E.g., Bach et al., 2015; Montavon et al., 2017; Springenberg et al., 2015; Sundararajan et al., 2017; Shrikumar et al., 2017; Kindermans et al., 2018; Zeiler and Fergus, 2014). They adapt to the architecture of a neural network by, as the name already suggests, performing a back-propagation along the network. This requires knowledge about the networks structure and stands in contrast to methods that treat the model as a black box and only use function or gradient evaluation for creating an analysis (Kindermans et al., 2016; Shrikumar et al., 2017; Smilkov et al., 2017; Sundararajan et al., 2017; Zintgraf et al., 2017; Ribeiro et al., 2016; Lundberg and Lee, 2017). Such algorithms typically use repeated function or gradient calls to create an explanation — therefore their runtime is often a multiple of a single call. On the other hand, propagation-based methods perform only one backprop-pass and can be very fast. For more details we refer to the Section 1.3 in Chapter II.

Based on their characteristics different methods define different ways to propagate information and, subsequently, deliver different explanations. Yet ho is it possible that for a single prediction different explanations exist? One reason is that each method emphasizes distinct features, e.g., signals vs. filters of the network (Haufe et al., 2014; Kindermans et al., 2018). Another, more complicated, reason is the absence of a ground truth or evaluation criteria for explanation methods. Samek et al. (2017) tries to solve this problem, yet in general it persists and is part of the issues of the discussed methods, e.g., (Kindermans et al., 2017; Dombrowski et al., 2019). Nonetheless, despite their short existence explanation methods have shown promising results (E.g., Lapuschkin et al., 2016a; Binder et al., 2018; Lapuschkin et al., 2017; Zintgraf et al., 2017; Sundararajan et al., 2017) and already their objective — facilitating understanding — makes them a worthwhile research topic.

Aim In this work we contribute to the need for a better understanding of neural networks (predictions) by pursuing the following objectives:

- (1) Lowering the application barrier and effort of explanation methods for non-experts and,
- (2) providing an API that allows experts to focus on algorithm customization and research.

Additional reasons underline the need for this software to be efficient and well designed:

- (1) There is no clear evaluation criteria for explanation algorithms and the suitability of a method must be determined for each task at hand. Therefore it is important that software facilitates the comparison of different approaches in order to find the best match.
- (2) The biggest need for understanding is created by large and complicated neural networks, yet for many methods (E.g., Bach et al., 2015; Montavon et al., 2017) there are no implementations that can adapt or generalize to them.
- (3) New network architectures are emerging and challenge current explanation approaches. Ideally, an existing software library would facilitate research efforts to tackle this.
- (4) Novel research directions apply the mentioned *instance-based* methods to whole datasets and use the results to learn about the prediction process, e.g., (Lapuschkin et al., 2016a, 2019). This requires software with good performance.

2.1 Related software packages

None of the subsequently discussed implementations addresses these objectives and requirements (fully). The most common reason is that the respective implementation accompanies the publication of an algorithm and is product of a research process. This set of implementations does not adhere to a common interface or platform, and

such a fragmentation makes it hard to compare algorithms. The most similar related work are the two software packages, DeepExplain (Ancona et al., 2018) and keras-vis (Kotikalapudi and contributors, 2017). Compared to our library both support a considerably smaller number of methods, especially they don't support the complex, but promising methods LRP, Deep Taylor Decomposition, PatternNet, and PatternAttribution. Furthermore, none offers an interface to support the implementation of propagation-based methods.
In more detail, the following software implementations of explanation techniques are available: For the LRP-algorithm a toolbox was published (Lapuschkin et al., 2016b) that contains explanatory code in Python and MatLab as well as a faster Caffee-implementation for production purposes. For the algorithms DeepLIFT (Shrikumar et al., 2017), DeepSHAPE (Lundberg and Lee, 2017), "prediction difference analysis" (Zintgraf et al., 2017), and LIME (Ribeiro et al., 2016) the authors also published source code that is based on Keras/Tensorflow, Tensorflow, Tensorflow and scikit-learn respectively. For the algorithm GradCam (Selvaraju et al., 2017) the authors published a Caffee-based implementation. There exist more GradCam implementations for other frameworks, e.g., Kotikalapudi and contributors (2017).
Software packages that contain more than one algorithm family are the following. The software to the paper DeepExplain (Ancona et al., 2018) contains implementations for the gradient-based algorithms saliency map, gradient * input, Integrated Gradients, one variant of DeepLIFT and LRP-Epsilon as well as for the occlusion algorithm. The implementation is based on Tensorflow. The Keras-based software keras-vis (Kotikalapudi and contributors, 2017) offers code to perform activation maximization, saliency algorithms Deconvnet and GuidedBackprop as well as GradCam. For comparison, the library presented in this chapter, *iNNvestigate* (Alber et al., 2019), is also Keras-based and contains implementations for the algorithms saliency map, gradient * input, Integrated Gradients, Smoothgrad, DeconvNet, GuidedBackprop, Deep Taylor Decomposition, different LRP algorithms as well as PatternNet and PatternAttribution. It also offers an interface to facilitate the implementation of propagation-based explanation methods.

3 The iNNvestigate library

The name hints it all: Investigating the predictions of neural networks — *iNNvestigate*. The software package was developed to meet the outlined objectives, in a nutshell, creating an accessible software for applying, comparing and developing explanation methods.
The roots of this project originate from the work on PatternNet and PatternAttribution (Kindermans et al., 2018). The authors encountered many issues that inspired us to create this software. Since then the code was entirely redesigned and rewritten, and, eventually, released as an open-source project.
In this section we will discuss the design and implementation patterns and showcase them by implementing the most important explanation methods. In more detail, Section 3.1 gives an overview on the specific characteristics of this software package, Section 3.2 describes the design and concrete implementation of propagation-based

methods and Section 3.4 completes this by introducing the user interface and other important functionality for applying explanation algorithms. In Section 3.3 we will benchmark the runtime performance of *iNNvestigate*.

3.1 Characterization

To give an overview of this software package we outline first the methodological part and then the technical details. The distinctive features of *iNNvestigate* are as follows:

- A clear and simple user interface (See Section 3.4.1).
- A modular backend for propagation-based methods (See Section 3.2).
- A wide range of implemented methods:
  - The reference implementation for PatternNet and PatternAttribution (Kindermans et al., 2018).
  - The new reference implementation for the LRP methods (Bach et al., 2015).
  - The only implementation for Deep Taylor Decomposition for Keras and TensorFlow (Montavon et al., 2017).
  - Implementations for: saliency maps (Baehrens et al., 2010), SmoothGrad (Smilkov et al., 2017), input*gradient (Kindermans et al., 2016; Shrikumar et al., 2017), Deconvnet (Zeiler and Fergus, 2014), GuidedBackprop (Springenberg et al., 2015), and IntegratedGradients (Sundararajan et al., 2017).
- An implementation of the evaluation algorithm "perturbation analysis" (Samek et al., 2017).

The software library is released as open-source package under the BSD-2 license and the repository is available at `https://github.com/albermax/innvestigate`. The development is based on the Python ecosystem and the deep learning framework Keras (Chollet, 2015). Keras is designed to make deep learning accessible for a broad range of users and *iNNvestigate* users benefit from that as they can take advantage of features like reference implementations for many neural networks, data preprocessing pipelines, and a big community. Currently only TensorFlow (Abadi et al., 2016b) is supported as Keras backend and the computation of all mentioned methods can be scheduled on the CPU or GPU devices. There are no dependencies on proprietary software.

3.2 Propagation-based prediction analysis

Implementing a neural network efficiently can be a complicated and error-prone process and additionally implementing an explanation algorithm makes things even trickier. We will now introduce the key patterns of explanation algorithms that allow for an efficient and structured implementation. Subsequently we complete the section by

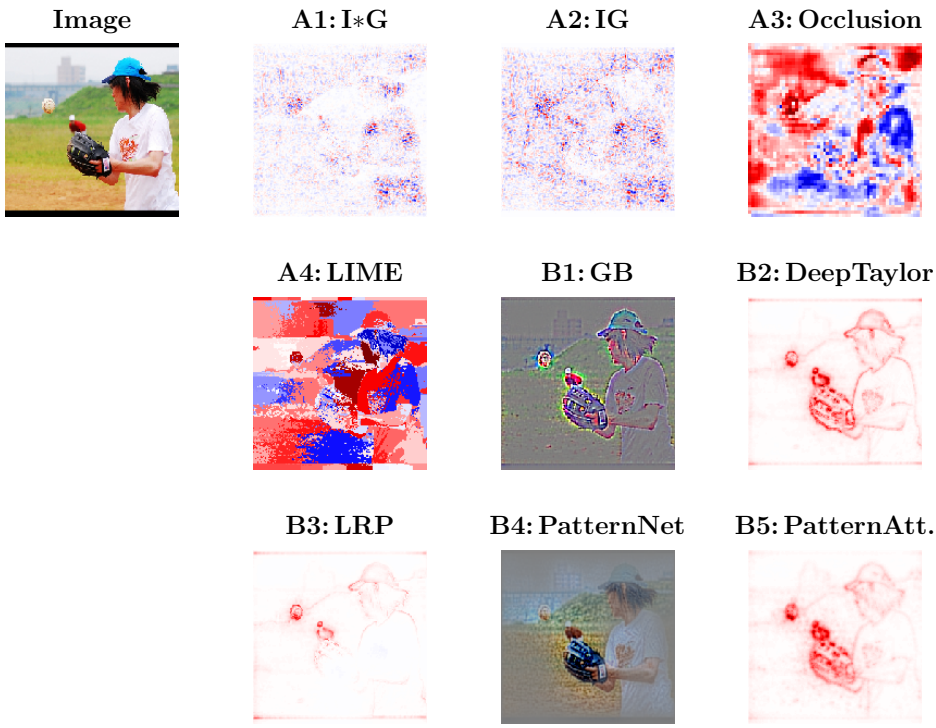

Fig. V.1: Exemplary application of the implemented algorithms. *This figure shows the results of the implemented explanation methods applied on the image in the upper-left corner using the VGG16 network (Simonyan and Zisserman, 2014). The prediction- or gradient-based methods (group A, see Appendix 2.1) are Input * Gradient (Kindermans et al., 2016; Shrikumar et al., 2017, A1), Integrated Gradients (Sundararajan et al., 2017, A2), Occlusion (Zeiler and Fergus, 2014, A3), and LIME (Ribeiro et al., 2016, A4). The propagation-based methods (group B) are Guided Backprop (Springenberg et al., 2015, B1), Deep Taylor (Montavon et al., 2017, B2), LRP (Lapuschkin et al., 2017, B3), PatternNet & PatternAttribution (Kindermans et al., 2018, B4 and B5). On how the explanations are visualized we refer to Section 3.4.3. Best viewed in digital and color.*

explaining how to approach interface design, parameter tuning, and visualization of the results.

To make the code examples as useful as possible we will not rely on pseudo-code, but rather use Keras (Chollet, 2015), TensorFlow (Abadi et al., 2016b) and *iNNvestigate* (Alber et al., 2019) to implement our examples for the example network VGG16 (Simonyan and Zisserman, 2014). The results are illustrated in Figure V.1 and will be created step-by-step. The code listings contain the most important code fragments and we refer for the corresponding executable code to Alber (2019).

Let us recall that the algorithms we explore have a common functional form, namely they map from the input to a equal-dimensional saliency map, e.g., the output saliency map has the same tensor shape as the input tensor. More formal: given a neural network model that maps some input to a single output neuron $f : \mathbb{R}^n \mapsto \mathbb{R}$, the considered algorithms have the following form $e : \mathbb{R}^n \mapsto \mathbb{R}^n$. We will select as output neuron the neuron with the largest activation in the final layer. Any other neuron could also be used. We assume that the target neural network is given as Keras model and the corresponding input and output tensor are given as follows:

```
# Create model without trailing softmax
model = make_a_keras_model()

# Get TF tensors
input, output = model.inputs[0], model.outputs[0]
# Reduce output to response of neuron with largest activation
max_output = tf.reduce_max(output, axis=1)

# Select a sample image
x_not_pp = select_a_sample_image()
# and preprocess it for the network
x = preprocess(x_not_pp)
```

The explanation algorithms of interest can be divided into two major groups depending on how they treat the given model. The first group of algorithms treats the model as a black-box with only access to function and gradient evaluations and the second group considers the model as a white-box, i.e., requires the ability to introspect the model and adapt to its composition. The former extracts information on the prediction process by repetitive function or gradient evaluations with altered inputs, the latter is typically more complex to implement, but aims to gain insights more efficiently and/or of different quality.

As mentioned propagation-based algorithms are the main focus of this work — albeit *iNNvestigate* contains also gradient-based techniques. For completeness and for comparison we describe and implement also a number of prediction- and gradient-based methods. The according section can be found in the Appendix 2.1. We will now describe the implementation of propagation-based methods.

Algorithms using a custom back-propagation routine to create an explanation are in stark contrast to prediction- or gradient-based explanation algorithms: they rely on knowledge about the model's internal functioning to create more efficient or diverse explanations.

Consider gradient back-propagation that works by first decomposing a function and then performing an iterative backward mapping. For instance, the function $f(x) = u(v(x)) = (u \circ v)(x)$ is first split into the parts u and v — of which it is composed of in the first place — and then the gradient $\frac{\delta f}{\delta x}$ is computed iteratively $\frac{\delta f}{\delta x} = \frac{\delta u \circ v}{\delta v} \frac{\delta v}{\delta x}$ by backward mapping each component using the partial derivatives $\frac{\delta u \circ v}{\delta v}$ and $\frac{\delta v}{\delta x}$. Similar to the computation of the gradient, all propagation-based explanations have this approach in common: (1) each algorithm defines, explicitly or implicitly, how a network should be decomposed into different parts and (2) how for each component

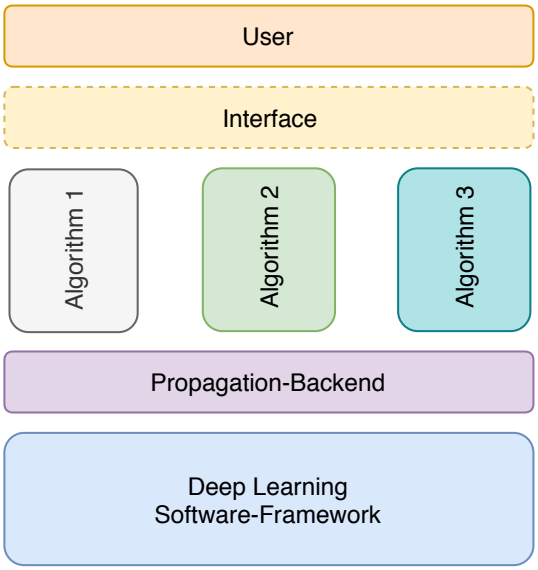

Fig. V.2: Software-stack. *The diagram depicts exemplarily the software stack of iNNvestigate (Alber et al., 2019). It shows how different propagation-based methods are build on top of a common graph-backend and expose their functionality through a common interface to the user.*

the backward mapping should be performed. When implementing an algorithm for an arbitrary network it is important to consider that different methods target different components of a network, that different decompositions for the same method can lead to different results and that certain algorithms cannot be applied to certain network structures.
For instance consider GuidedBackprop (Springenberg et al., 2015) and Deep Taylor Decomposition (Montavon et al., 2017, DTD). The first targets ReLU-activations in a network and describes a backward mapping for such non-linearities, while partial derivatives are used for the remaining parts of the network. On the other hand, DTD and many other algorithms expect the network to be decomposed into linear(izable) parts — which can be done in several ways and may result in different results.
When developing such algorithms the emphasis is typically on how a backward mapping can lead to meaningful explanations, because the remaining functionality is very similar and shared across methods. Knowing that, it is useful to split the implementation of propagation-based methods in the following two parts. The first part contains the algorithm details — thus defines how a network should be decomposed and how the respective mappings should be performed. It builds upon the next part which takes care of the common functionality, namely decomposing the network as previously specified and iteratively applying the mappings. Both are denoted as "Algorithm" and "Propagation-backend" in *iNNvestigate*'s software stack in Figure V.2.
This abstraction has the big advantage that the complex and algorithm independent graph-processing code is shared among explanation routines and allows the developer to focus on the implementation of the explanation algorithm itself.

We will now describe how each component can be implement, starting with the backend. Eventually it should allow the developer to realize a method in the following schematic way — using the interface to be presented in Section 3.4.1:

```
# A backward mapping function, e.g., for convolutional layers
def backward_mapping(Xs, Ys, bp_Ys, bp_state):
  return compute_backward_mapping_magic()

# A class bundling all algorithm functionality
class ExplanationAlgorithm(Analyzer):
  ...
  # Defining how to perform the algorithm
  def _create_analysis(self):
    # Tell the backend that this mapping
    # should be applied, e.g., to all convolutional layers.
    register_backward_mapping(
      condition=lambda x: is_convolutional_layer(x),
      backward_mapping)
    ...

# Create and build algorithm for a model
analyzer = ExplanationAlgorithm(model)
# Perform the analysis
analyze = analyzer.analyze(x)
```

3.2.1 Creating a propagation backend

Let us reiterate the aim, which is to create routines that capture common functionality to all propagation-based algorithms and thereby facilitate their efficient implementation. Given the information which graph-parts shall be mapped and how, the backend should decompose the network accordingly and then process the back-propagation as specified. It would be further desirable that the backend is able to identify if a given neural network is not compatible with an algorithm, e.g., because the algorithm does not cover certain network properties.
In this respect we see as major challenges for creating an efficient backend the following:

Interface: How shall an algorithm specify the way a network should be decomposed and how should each backward mapping be performed?

Graph matching: Decomposing the neural network according to the algorithm's specifications and, ideally, detecting possible incompatibilities. Note that the specifications can describe the structure of the targeted components as well as their location in the network, e.g., DTD treats layer differently depending where they are located in the network.

Back-propagation: Once determined which backward mapping is used for which part of the network graph, the respective mappings should be applied in the right order until the final explanation is produced.

The first two challenges are solved by choosing appropriate abstractions. The abstractions should be fine-grained enough to enable the implementation of a wide range of

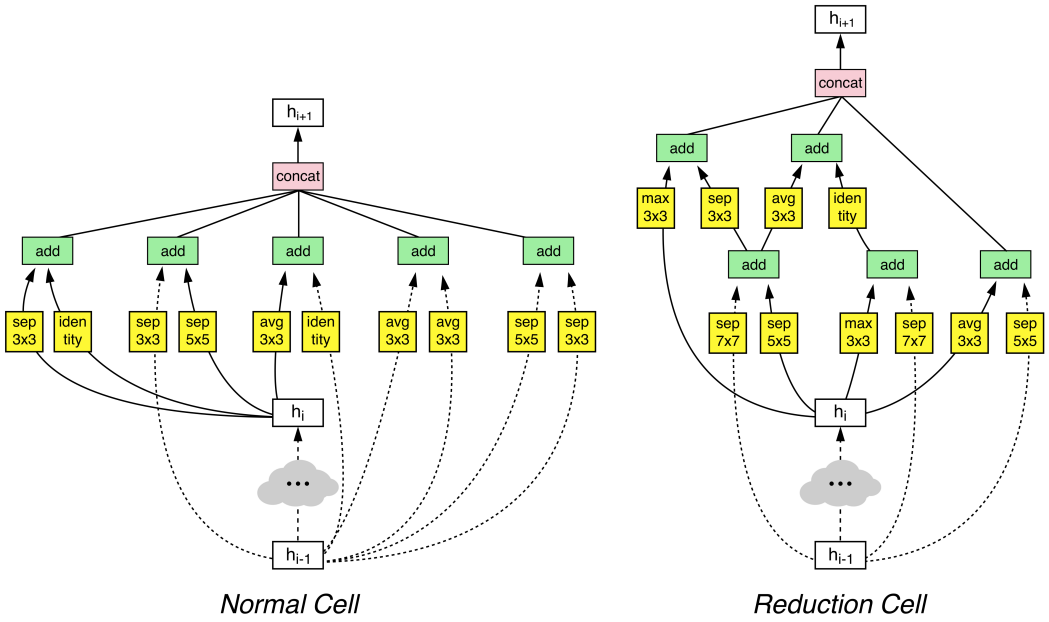

Fig. V.3: NASNetA cells. *The computer vision network NASNetA (Zoph et al., 2018) was created with automatic Machine Learning, i.e., the architecture of the two depicted building blocks was found with an automated algorithm. The normal cell and the reduction cell have the same purpose as convolutional or max-pooling layers in other networks, but are far more complex. Figure is from Zoph et al. (2018).*

algorithms, while being coarse-grained enough to allow for an efficient implementation. The last challenge is in the first place an engineering task.

Interface & Matching Before we discuss how such a backend can be implemented, we would like to consider potential architectures of neural networks. Neural networks are often characterized by their layer-oriented structure and the simplest of them are sequential neural networks where each layer is stacked on another layer. In contrast many state-of-the-art neural network have much more complicated and non-sequential architectures. For instance, NASNetA (Zoph et al., 2018) has a similar high level structure as VGG16 (Simonyan and Zisserman, 2014), namely they are both composed of normal and reduction cells. In VGG16 these cells are implemented as a single convolutional (or dense) layer and as a max-pooling layer respectively. The normal and reduction cells of NASNetA are shown in Figure V.3. They are composed of many different layers which are connected in a non-sequential manner. This increases the overall network complexity considerably and to correctly implement an explanation software that is applicable to such and other network structures it is necessary to choose the right abstractions.
Accordingly, the first step towards a clear interface is to regard a neural network as a directed-acyclic-graph (DAG) of layers — instead of a stack of layers. The notion of a graph of "layers" might not seem intuitive in the first place and comes from the

time when neural networks were typically sequential, thus one layer was stacked onto another. Modern networks, e.g., as NASNetA in Figure V.3, can be more complex and in such architectures each layer is rather a node in a graph than a layer in a stack. Regardless of that, nodes in such a DAG are still commonly called layers and we will keep this notation.
A second step is to be aware of DAGs granularity. Different deep learning frameworks represent neural networks in different ways and layers can be composed of more basic operations like additions and dot products, which in turn can be decomposed further. The most intuitive and useful level for implementing explanation methods is to view each layer as node. A more fine-grained view is in many cases not needed and would only complicate the implementation. On the other hand, we note that it might be desired or necessary to fuse layers of networks into one node, e.g., adjacent convolutional and batch normalization layers can be expressed as a single convolutional layer.
Building on this network representation, there are two interfaces to define. One to define where a mapping shall be applied and one how it should be performed.
There are two ways to realize the matching interface and they can be sketched as follows. The first binds a custom backward mapping before or during network building to a method of a layer class — statically by extending a layer class or by overloading its gradient operator. The second receives the already build model and matches the mappings dynamically to the respective layer nodes. This can be done by evaluating a programmable condition for each layer instance or node in order to assign a mapping. Except for the matching conditions, both techniques expose the same interface and in contrast to the first approach the later is more challenging to implement, but has several advantages: (1) It exposes a clear interface by separation of concerns: the model building happens independently of the explanation algorithm. (2) The forward DAG can be modified before the explanation is applied, e.g., batch normalization layer can be fused with convolutional layers. (3) When several explanation algorithms are build for one network, they can share the forward pass. (4) The matching conditions can introspect the whole model, because the model was already build at that point in time. (5) One can build efficiently the gradient w.r.t. explanations by using forward-gradient computation — in the background and for all explanation algorithms by using automatic differentiation.

The two approaches can be sketched in Python as follows:

```
# Approach A
# Use mapping Y for layer type X
register_mapping_for_layer_type(layer_type_X, mapping_Y)
build_model_with_custom_mapping()
execute_explanation()

# Approach B
model = build_model()
graph = extract_and_update_graph(model)
for node in graph:
  # Match node to mapping based on conditions
  # A node can be a layer or a sub-graph.
  # Condition can introspect whole model for decision.
  mapping = match_node_to_mapping(node, model.graph)
  assign_mapping_to_node(node, mapping)
```

One of the distinctive features of *iNNvestigate* is that it uses the second, more evolved variant.
The second interface addresses the backward mapping and is a function that takes as parameters the input and output tensors of the targeted layer, the respective back-propagated values for the output tensors and, optionally, some meta-information on the back-propagation process. The following code segment shows the interface of a backward mapping function in the *iNNvestigate* library. Due to same purpose other implementations have very similar interfaces.

```
# Xs = input tensors of a layer or sub-graph
# Ys = ouput tensors of a layer or sub-graph
# bp_Ys = back-propagated values for Ys
# bp_state = additional information on state
# return back-propagated values for Xs
def backward_mapping(Xs, Ys, bp_Ys, bp_state):
  # the backward mapped tensors correspond in shape
  # with respective the output tensors of the forward pass
  assert len(Ys) == len(bp_Ys)
  assert all(Y.shape == bp_Y.shape for Y, bp_Y in zip(Ys, bp_Ys))

  bp_Xs = compute_backward_mapping_magic()

  # the returned tensors correspond in shape
  # with the respective input tensors of the forward pass
  assert len(Ys) == len(bp_Ys)
  assert all(Y.shape == bp_Y.shape for Y, bp_Y in zip(Ys, bp_Ys))
  return bp_Xs
```

Note that this signature can not only be used for the backward mapping of layers, but for any connected sub-graph. In the remainder we will use a simplified interface where each layer has only one input and one output tensor.

Back-propagation Having matched backward mappings with network parts the backend still needs to create the actual backward propagation. Practically this can

be done explicitly, as we will show below, or by overloading the gradient operator in the deep learning framework of choice. While the latter is easier to implement it less flexible and has the dis-advantages mentioned above.
The implementations of neural networks is characterized by their layer-oriented structure and the simplest of them are sequential neural networks where each layer is stacked on another layer. To back-propagate through such a network one starts with the model's output value and propagates from top layer to the next lower one and so on. Given mapping functions that take a tensor and back-propagate along a layer, this can be sketched as follows:

```
current = output
for layer in model.layers[::-1]:
  current = back_propagate(layer.input, layer.output, current)
analysis = current
```

In general neural networks can be much more complex and are represented as directed, acyclic graphs. This allows for multiple input and output tensors for each "layer node". An efficient implementation is for instance the following. First the right propagation order is established using the depth-first search algorithm to create a topological ordering (Cormen et al., 2009). Then given this ordering, the propagation starts at the output tensors and proceeds in direction of the input tensors. At each step, the implementation collects the required inputs for each node, applies the respective mapping and keeps track of the back-propagated tensors after the mapping. Note, nodes that branch in the forward pass, i.e., have an output tensor that is used several times in the forward pass, receive several tensors as inputs in the backward pass. These need to be reduced to a single tensor before being fed to the backward mapping. This is typically like in the gradient computation, namely by summing the tensors:

```
intermediate_tensors = {output: output}
execution_order = calculate_execution_order()
for layer, inputs, outputs in execution_order[::-1]:
  # gather corresponding back-propagated tensors for each output tensor
  back_propagated_values = [
    # Reduce to single tensor if the forward passed branched!
    sum(intermediate_tensors[t])
    for t in outputs
  ]

  # backprop through layer
  tmp = back_propagate(inputs, outputs, back_propagated_values)

  # store intermediate tensors
  for input, intermediate in zip(inputs, tmp):
    if input in intermediate_tensors:
      intermediate_tensors[input] = [intermediate]
    else:
      # The corresponding forward tensor branched!
      intermediate_tensors[input].append(intermediate)

# get the last output
analysis = intermediate_tensors[model.input]
```

Despite its relative simplicity, implementing and debugging such an algorithm can be tedious. This among propagation-based methods common operation is part of the *iNNvestigate* library and as a result one only needs to specify how the back-propagation through specific layers should be performed. Even handier, as default backward mapping the gradient-propagation is used and one only needs to specify whenever the back-propagation should be performed differently.

3.2.2 Customizing the back-propagation

Based on the established interface we are now able to implement various propagation-based explanation methods in an efficient manner.

Guided Backprop As a first example we implement the algorithm Guided Backprop (Springenberg et al., 2015). The back-propagation of Guided Backprop is the same as for the gradient computation, except that whenever a ReLU is applied in the forward pass another ReLU is applied in the backward pass. Note that the default back-propagation mapping in *iNNvestigate* is the partial derivative, thus we only need to change the propagation for layers that contain a ReLU activation and apply an additional ReLU in the backward mapping. The corresponding code looks like follows and can already be applied to arbitrary networks (see B1 in Figure V.1):

```
# Guidded-Backprop-Mapping
# X = input tensor of layer
# Y = ouput tensor of layer
# bp_Y = backpropagated value for Y
# bp_state = additional information on state
def guided_backprop_mapping(X, Y, bp_Y, bp_state):
  # Apply ReLU to back-propagate values
  tmp = tf.nn.relu(bp_Y)
  # Propagate back along the gradient of the forward pass
  return tf.gradients(Y, X, grad_ys=tmp)

# Extending iNNvestigate base class with the Guideded Backprop code
class GuidedBackprop(ReverseAnalyzerBase):

  # Register the mapping for layers that contain a ReLU
  def _create_analysis(self, *args, **kwargs):

    self._add_conditional_reverse_mapping(
      # Apply to all layers that contain a relu activation
      lambda layer: kchecks.contains_activation(layer, 'relu'),
      # and use the guided_backprop_mapping to do the backrop step.
      tf_to_keras_mapping(guided_backprop_mapping),
      name='guided_backprop',
    )

  return super(GuidedBackprop, self)._create_analysis(*args, **kwargs)

# Creating an instance of that analyzer
analyzer = GuidedBackprop(model_wo_sm)
# and apply it.
B1 = analyzer.analyze(x)
```

Deep Taylor Typically propagation-based methods are more evolved. Propagations are often only described for fully connected layers and one key pattern that arises is extending this description seamlessly to convolutional and other layers. Examples for this case are the "Layerwise relevance propagation" (Bach et al., 2015, LRP), the "Deep Taylor Decomposition" (Montavon et al., 2017, DTD) and the "Excitation Backprop" (Zhang et al., 2018, EB) algorithms. Despite different motivation all algorithms yield similar propagation rules for neural networks with ReLU-activations. The first algorithm takes the prediction values at the output neuron and calls it relevance. Then this relevance is re-distributed at each neuron by mapping the back-propagated relevance proportionally to weights onto the inputs. We consider here the so-called Z+ rule. In contrast, Deep Taylor is motivated by a (linear) Taylor decomposition for each neuron and Excitation Backprop by a probabilistic "Winner-Take-All" scheme. Ultimately, for layers with positive input and positive output values — like the inner layers in VGG16 — they all have the following propagation formula:

$$\begin{aligned} bw\_mapping(x, y, r := bp\_y) &= x \odot (W_+^t z) \\ \text{with } z &= r \oslash (xW_+) \end{aligned} \tag{V.1}$$

given a fully connected layer with W_+ denoting the weight matrix where negative values are set to 0. Using the library *iNNvestigate* this can be coded in this way:

```
# Deep-Taylor/LRP/EB's Z+-Rule-Mapping for dense layers
# Call R=bp_Y, R for relevance
def z_rule_mapping_dense(X, Y, R, bp_state):
  # Get layer and the parameteres
  layer = bp_state['layer']
  W = tf.maximum(layer.kernel, 0)

  Z = tf.tensordot(X, W, 1) + b
  # normalize incoming relevance
  tmp = R / Z
  # map back
  tmp = tf.tensordot(tmp, tf.transpose(W), 1)
  # times input
  return tmp * X
```

Unfortunately, this mapping implementation only covers fully-connected layers, while another key layer family, namely convolutional layers, are not covered. By creating another mapping for two dimensional convolutions the code can be completed and applied to VGG16. For the unconstrained input layer we will use the bounded rule proposed by Montavon et al. (2017):

```
# Deep-Taylor/LRP/EB's Z-Rule-Mapping for conv layers
# Call R=bp_Y, R for relevance
def z_rule_mapping_conv(X, Y, R, bp_state):
  # Get layer and the parameters
  layer = bp_state['layer']
  W = tf.maximum(layer.kernel, 0)

  Z = tf.keras.backend.conv2d(X, W, layer.strides, layer.padding) + b
  # normalize incoming relevance
  tmp = R / Z
  # map back
  tmp = tf.keras.backend.conv2d_transpose(
    tmp, W, (1,)+keras.backend.int_shape(X)[1:],
    layer.strides, layer.padding)
  # times input
  return tmp * X

# Extending iNNvestigate base class with the Deep Taylor/LRP/EB's Z+-rule
class DeepTaylorZ1(ReverseAnalyzerBase):
  # Register mappings for dense and convolutional layers.
  # Add Bounded DeepTaylor rule for input layer.

analyzer = DeepTaylorZ1(model_wo_sm)
B2a = analyzer.analyze(x)
```

and the result is shown in Figure V.1 denoted as B2.
Still, this code does not cover one-dimensional, three-dimensional or any other special type of convolutions. Conveniently unnecessary code-replication can be avoided by using automatic differentiation. The core idea is that many methods can be expressed as pre-/post-processing of the gradient back-propagation. Using automatic differentiation our code example can be expressed as follows and works now with any type of convolutional layer:

```
# Deep-Taylor/LRP/EB's Z+-Rule-Mapping for all layers with a kernel
# Call R=bp_Y, R for relevance
def z_rule_mapping_all(X, Y, R, bp_state):
  # Get layer
  layer = bp_state['layer']
  # and create layer copy without activation part
  W = tf.maximum(layer.kernel, 0)
  layer_wo_act = kgraph.copy_layer_wo_activation(
    layer, weights=[W], keep_bias=False)

  Z = layer_wo_act(X)
  # normalize incoming relevance
  tmp = R / Z
  # map back
  tmp = tf.gradients(Z, X, grad_ys=tmp)[0]
  # times input
  return tmp * X
```

LRP For some applications or methods it can be necessary to use different propagation rules for different layers. E.g., Deep-Taylor requires different rules depending on the input data range (Montavon et al., 2017) or for LRP it was empirically demonstrated to be useful to apply different rules for different parts of a network (Lapuschkin et al., 2017). To exemplify this, we show how to use a different LRP rules for different layer types as presented in (Lapuschkin et al., 2017). In more detail, we will apply the epsilon rule for all dense layer and the alpha-beta rule for convolutional layers. This can be implemented in *iNNvestigate* by changing the matching conditions. Using provided LRP-rule mappings this looks as follows:

```
class LRPConvNet(ReverseAnalyzerBase):

  # Register the mappings for different layer types
  def _create_analysis(self, *args, **kwargs):

    # Use Epsilon rule for dense layers
    self._add_conditional_reverse_mapping(
      lambda layer: kchecks.is_dense_layer(layer),
      LRPRules.EpsilonRule,
      name='dense',
    )
    # Use Alpha1Beta0 rule for conv layers
      self._add_conditional_reverse_mapping(
      lambda layer: kchecks.is_conv_layer(layer),
      LRPRules.Alpha1Beta0Rule,
      name='conv',
    )

  return super(LRPConvNet, self)._create_analysis(*args, **kwargs)

analyzer = LRPConvNet(model_wo_sm)
B3 = analyzer.analyze(x)
```

The result can be examined in Figure V.1 marked with B3.

PatternNet & PatternAttribution PatternNet & PatternAttribution (Kindermans et al., 2018) are two algorithms that are inspired by the pattern-filter theory for linear models (Haufe et al., 2014). They learn for each neuron in the network a signal direction called pattern. In PatternNet the patterns are used to propagate the signal from the output neuron back to the input by iteratively using the pattern directions of the neurons and the method can be realized with a gradient backward-pass where the filter weights are exchanged with the pattern weights. PatternAttribution is based on the Deep Taylor Decomposition (Montavon et al., 2017). For each neuron it searches the rootpoint in the direction of its pattern. Given the pattern a the corresponding formula is:

$$bw\_mapping(x, y, r = bp\_y) = (w \odot a)^t r \tag{V.2}$$

and it can be implemented by doing a gradient backward pass where the filter weights are element-wise multiplied with the patterns.
So far we implemented the backward-mappings as functions and registered them inside an analyzer class for backpropagation. In the next example we will create a single class that takes a parameter, namely the patterns, and the mapping will be a class method that uses a different pattern for each layer mapping (B4 in Figure V.1). The following code sketches the implementations which can be found in Appendix 2.2:

```
# Extending iNNvestigate base class with the PatternNet algorithm
class PatternNet(ReverseAnalyzerBase):

  # Storing the patterns.
  def __init__(self, model, patterns, **kwargs):
    self._patterns = patterns[:]
    super(PatternNet, self).__init__(model, **kwargs)

  def _get_pattern_for_layer(self, layer):
    return self._patterns.pop(-1)

  # Peform the mapping
  def _patternnet_mapping(self, X, Y, bp_Y, bp_state):
    ...
    # Use patterns specific to bp_state['layer']
    ...

  # Register the mapping
  def _create_analysis(self, *args, **kwargs):
    ...

analyzer = PatternNet(model_wo_sm, net['patterns'])
B4 = analyzer.analyze(x)
```

Encapsulating the functionality in a single class allows us now to easily extend PatternNet to PatternAttribution by changing the parameters that are used to perform the backward pass (B5 in Figure V.1):

```
# Extending PatternNet to PatternAttribution
class PatternAttribution(PatternNet):

  def _get_pattern_for_layer(self, layer):
    filters = layer.get_weights()[0]
    patterns = self._patterns.pop(-1)
    return filters * patterns

analyzer = PatternAttribution(model_wo_sm, net['patterns'])
B5 = analyzer.analyze(x)
```

3.2.3 Generalizing to more complex networks

For our examples we relied on the VGG16 network (Simonyan and Zisserman, 2014) which is composed of linear and convolutional layers with ReLU-or Softmax-activations as well as max-pooling layers. Recent networks in computer vision like, e.g., InceptionV3 (Szegedy et al., 2016), ResNet50 (He et al., 2016), DenseNet (Huang et al., 2017), or NASNet (Zoph et al., 2018), are far more complex and contain a variety of new layers like batch normalization layer (Ioffe and Szegedy, 2015), new types of convolutional layers (E.g., Chollet, 2017) and merge layers that allow for residual connections (He et al., 2016).
The presented code examples either generalize to these new architectures or can be easily adapted to them. Exemplary, Figure V.10 in Section 4 shows a variety of algorithms applied to several state-of-the-art neural networks for computer vision. For each algorithm the *same* explanation code is used to analyze all different networks. The exact way to adapt algorithms to new network families depends on the respective algorithm and is beyond the scope of this chapter. Typically it consists of implementing new mappings for new layers, if required.

3.3 Benchmark

The the runtime efficiency of the presented code can be expressed with a benchmark. As a reference implementation we use the LRP-Caffe-Toolbox (Lapuschkin et al., 2016b), because it is in some sense the predecessor to *iNNvestigate* and it was designed to implement algorithms with a similar complexity, namely the LRP-variants — which are the most complex algorithms we reviewed.
We test three algorithms that are implemented in both libraries and run them with the VGG16 network (Simonyan and Zisserman, 2014). Both frameworks need some time to compile the computational graph and to execute it on a batch of images, accordingly we measure both, the setup time and the execution time, for analyzing 512 images.
The LRP-Toolbox has a sequential and a parallel implementation for CPUs. We show the time for the faster, parallel implementation. For *iNNvestigate* we evaluate the runtime on the CPU and on the GPU. The workstation for the benchmark is equipped with an Intel Xeon CPU E5-2690-v4 2.60GHz with 24 physical cores mapped to 56

virtual cores and 256GB of memory. Both implementation can use up to 32 cores. The GPU is a Nvidia P100 with 16G of memory. We repeat each test 10 times and report the average duration. We found the timing results to be consistent and reliable across methods and thus omit further statistical hypotheses tests.
Figure V.4 shows the measured duration on a logarithmic scale. The *iNNvestigate* library is up to 29 times faster when both implementations run on the CPU. This increases up to 510 times when using *iNNvestigate* with the GPU compared to the LRP-Toolbox implementation on the CPU. This is achieved while *iNNvestigate* also considerably reduces the amount and the complexity of code to implement the explanation algorithms compared to the LRP-Toolbox. On the other hand, when using *iNNvestigate* one needs to compile a function graph and accordingly the setup needs up to 3 times as long as for the LRP-Toolbox — yet amortizes already when analyzing a few images.
This runtime efficiency is an advantage for methods that require explanations for a whole dataset. For instance, Lapuschkin et al. (2016a, 2019) creates explanations for a whole dataset to detect if a model focuses on correlated background or to group similar predictions. For such applications *iNNvestigate* requires a fraction of the runtime compared of the LRP-Toolbox. When comparing to implementations of other (simpler) algorithms based on frameworks such as TensorFlow we don't expect the difference to be as significant.

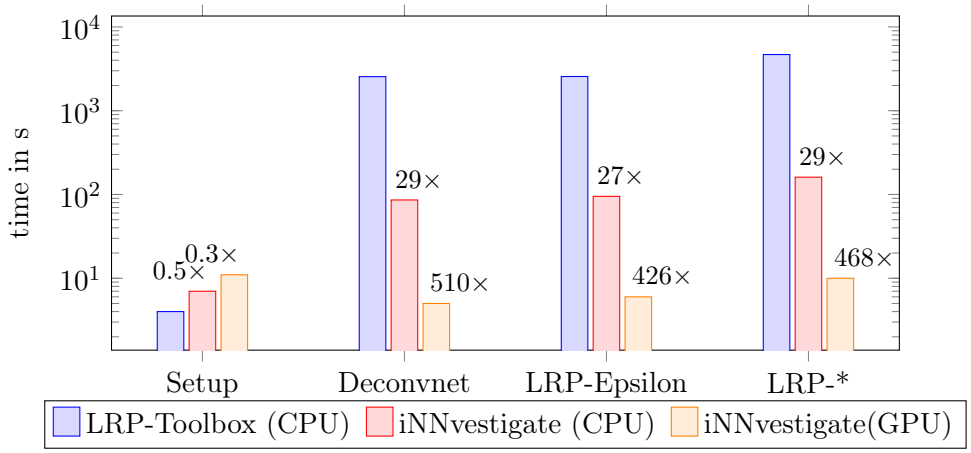

Fig. V.4: Runtime comparison. *The figure shows the setup- and run-times for 512 analyzed images in logarithmic range for the LRP-Toolbox and the iNNvestigate library. Each block contains the measured time for either the setup or one of the following algorithms: Deconvnet (Zeiler and Fergus, 2014), LRP-Epsilon (Bach et al., 2015), and the LRP configuration from Lapuschkin et al. (2017), denoted as LRP-*. The numbers in black indicate the respective speedup with regard to the LRP-Toolbox.*

3.4 Completing the implementation

More than the implementation of the methodological core is required to successfully apply and use explanation software. Depending on the hyper-parameter selection

and visualization approaches the explanation result may vary drastically. Therefore it is important that software is designed to help the users to easily select the most suitable setting for their task at hand. This can be achieved by exposing the algorithm software via an easy and intuitive interface, allowing the user to focus on the method application itself. Subsequently we will address these topics. First we will discuss interface design and then address hyper-parameter selection, post-processing and visualizations techniques implemented in *iNNvestigate*. We note that the input data pre-processing is by construction the same as for the given model.

3.4.1 Interface

Exposing clear and easy-to-use software interfaces and routines facilitates that a broad range of practitioners can benefit from a software package. For instance the popular scikit-learn (Pedregosa et al., 2011b) package offers a clear and unified interface for a wide range of Machine Learning methods, which can be flexibly adjusted to more specific use cases.
In our case one commonality of all explanation algorithms is that they operate on a neural network model and therefore an interface to receive a model description is required. There are two commonly used approaches. The first one is chosen by several software packages, e.g., DeepLIFT (Shrikumar et al., 2017) and the LRP-toolbox (Lapuschkin et al., 2016b), and consists of expecting the model in form of a configuration (file). A drawback of this approach is that the model needs to be serialized before the explanation can be executed.
An alternative way is to take the model represented as a memory object and operate directly with that, e.g., DeepExplain (Ancona et al., 2018) and *iNNvestigate* (Alber et al., 2019) work in this way. Typically this memory object was build with a deep learning framework. This approach has the advantage that an explanation can be created without additional overhead and it is easy to use several explanation methods in the same program setup — which is especially useful for comparisons and research purposes. Furthermore, a model, stored in form of a configuration, can still be loaded by using the respective deep learning framework's routines and then being passed to the explanation software.
The interface of the *iNNvestigate* package mimics the one of the popular software package scikit-learn and allows to create an explanation with a few lines of code. We note the this interface *separates concerns* — the model building is independent of the subsequent explanation — and is only possible due to our design decisions outlined in Section 3.2.1. An example application looks as follows:

```
# Build the explanation algorithm
# with the hyper-parameter pattern_type set to 'relu'
analyzer = PatternAttribution(model_wo_sm, pattern_type='relu')
# fit the analyzer to the training data (if an analyzer requires it)
analyzer.fit(X_train)
# and apply it to an input
e = analyzer.analyze(x)
```

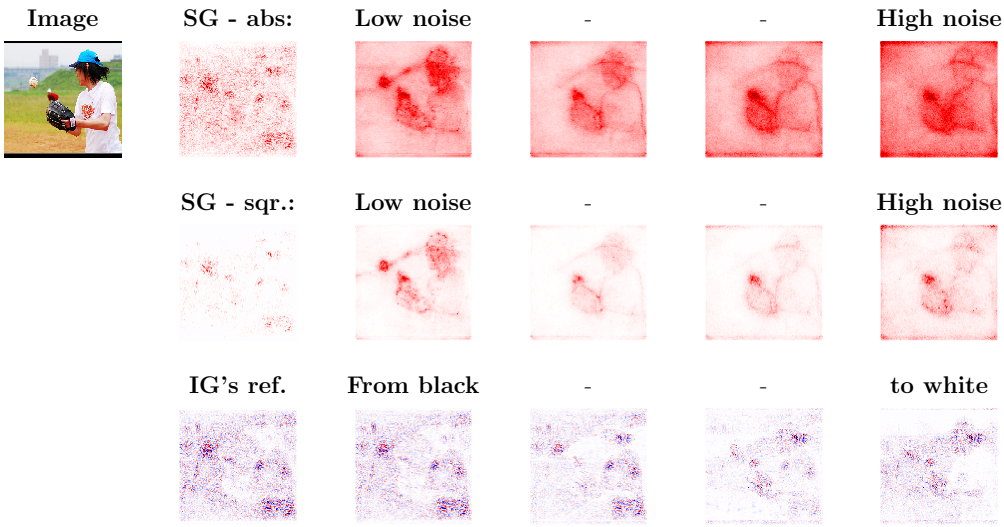

Fig. V.5: Influence of hyperparamters. *Row one to three show how different hyper-parameters change the output of explanation algorithms. Row 1 and 2 depict the Smoothgrad (**SG**) method where the gradient is transformed into a positive value by taking the absolute or the square value respectively. The columns show the influence of the noise scale parameter with low to high noise from left to right. In row 3 we show how the explanation of the Integrated Gradients (**IG**) method varies when selecting as reference an image that is completely black (left side) to completely gray (middle) to completely white (right). Best viewed in digital and color.*

3.4.2 Hyper-parameter selection

Like for many other tasks in Machine Learning explanation methods can have hyper-parameters, but unlike for other algorithms, for explanation methods no clear selection metric exists. Therefore selecting the right hyperparameter can be a tricky task. One way is a (visual) inspection of the explanation result by domain experts. This approach is suspected to be prone to the human confirmation bias. As an alternative in image classification settings Samek et al. (2017) proposed a method called "perturbation analysis". The algorithm divides an image into a set of regions and sorts them in decreasing order of the "importance" each regions gets attributed by an explanation method. Then the algorithm measures the decay of the neural networks prediction value when perturbing the blocks in the given order, i.e., "removing" the information of the most important image parts first. The key ideas is that if an explanation method highlights important regions better the performance will decay faster.

To visualize the sensitivity of explanation methods w.r.t. to their hyper-parameter Figure V.5 contains two example settings. The first example application shows the results for Integrated Gradients in row 3 where the image baseline varies from a black to a white image. While the black, nor the white, or the gray image as reference contains any valuable information, the explanation varies significantly — emphasizing the need to pay attention to hyper-parameters of explanation methods. More on the

Fig. V.6: Different visualizations. *Each column depicts a different visualization technique for the explanation of PatternAttribution or PatternNet (last column). The different visualization techniques for attribution methods are: graymaps (Smilkov et al., 2017) or single color maps to show only absolute values (column 1), heatmaps (Bach et al., 2015) to show positive and negative values (column 2), scaling the input by absolute values (Sundararajan et al., 2017, column 3), masking the least important parts of the input (Ribeiro et al., 2016, column 4), and blending the heatmap and the input (Selvaraju et al., 2017, column 5). The last technique is used to visualize signal extraction methods and is projecting the values back into the input value range (Kindermans et al., 2018, column 6). Best viewed in digital and color.*

sensitivity of explanation algorithms w.r.t. to this specific parameter can be found in Kindermans et al. (2017). The corresponding explanations can be generated with the code in Appendix 2.3.
Another example is the postprocessing of the saliency output. For instance for SmoothGrad the sign of the output is not considered to be informative and can be transformed to a positive value by using the absolute or the square value. This in turn has a significant impact on the result as depicted in Figure V.5 (row 1 vs. row 2). Furthermore, the second parameter of SmoothGrad is the scale of the noise used for smoothing the gradient. This hyper-parameter varies from small on the left hand side to large on the right hand side and, again, has a substantial impact on the result. Which setting to prefer depends on the application. The explanations were created with the code fragment in Appendix 2.3.

3.4.3 Visualization

The innate aim of explanation algorithms is to facilitate the understanding for humans. To do so the output of algorithms needs to be transformed into a human understandable format.
For this purpose different visualization techniques were proposed in the domain of computer vision. In Figure V.6 we depict different approaches and each one emphasizes or hides different properties of a method. The five approaches are using graymaps (Smilkov et al., 2017) or single color maps to show only absolute values (column 1), heatmaps (Bach et al., 2015) to show positive and negative values (column 2), scaling the input by absolute values (Sundararajan et al., 2017) (column 3), masking the least important parts of the input (Ribeiro et al., 2016) (colmun 4), blending the heatmap and the input (Selvaraju et al., 2017) (column 5), or projecting the values back into the input value range (Kindermans et al., 2018) (column 6). The last

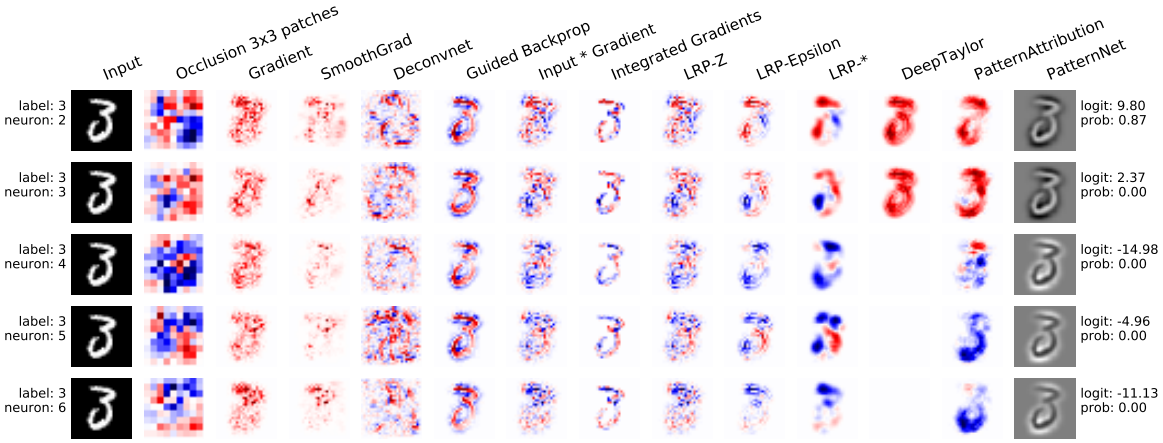

Fig. V.7: Analyzing a prediction. *The heatmaps show different analyses for a VGG-like network on MNIST. The network predicts the class* 2*, while the true label is* 3*. The heatmaps suggest that the network is not able detect the line discontinuity between the center and the lower, left stroke. Furthermore, while the presence of the right semicircle seems to be an indicator against a* 2*, this does not outweigh other factors. Each columns is dedicated to a different explanation algorithm. On the left hand side the true label and for each row the respective output neuron is indicated. Probabilities and pre-softmax activation are denoted on the right hand side of the plot. LRP-* denotes configuration from (Lapuschkin et al., 2017). We note that Deep Taylor is not defined when the output neuron is negative. Best viewed in digital and color.*

technique is used to visualize signal extraction techniques, while the other ones are used for attribution methods (Kindermans et al., 2018). To convert color images to a two-dimensional tensor, the color channels are typically reduced to a single value by the sum or a norm. Then the value gets projected into a suitable range and finally the according mapping is applied. This is done for all except for the last method, which projects each value independently. An implementation of the visualization techniques can be found in Appendix 2.4.
For other domains than image classification different visualization schemes are imaginable.

4 Applications

In this section we will use the implemented algorithms and examine common application patterns for explanation methods. We chose the following five patterns to reflect our initial objectives: (1) Analyzing single (miss-)prediction to gain insights on the model, and subsequently on the data. (2) Comparing algorithms to find a suitable explanation technique for the task at hand. (3) Researching and developing explanation methods. (4) Comparing prediction strategies of different network architectures. (5) Systematically evaluating the predictions of a network.

All except for the last application, which is semi-automatic, typically require a qualitative analysis to gain insights — and we will now see how explanation algorithms support this process. Furthermore, this section will give a limited overview and comparison of explanation techniques. A more detailed analysis is beyond the technical scope of this chapter.

We visualize the methods as presented in Section 3.4.3, i.e., use heatmaps for all methods except for PatternNet, which tries to produce a given signal and not an attribution. Accordingly we use a projection into the input space for it. Deconvnet and Guided Backprop are also regarded as signal extraction methods, but fail to reproduce color mappings and therefore we visualize them with heatmaps. This allows to identify the location of signals more easily. For more details we refer to Kindermans et al. (2018).

4.1 Analyzing a prediction

In our first example we focus on the explanation algorithms themselves and the expectations posed by the user. Therefore we chose a dataset without irrelevant features in the input space. In more detail we use a VGG-like network on the MNIST dataset (LeCun et al., 1998b) with an accuracy greater than 99% on the test set.

Figure V.7 shows the result for an input image of the class 3 that is incorrectly classified as 2. The different rows show the explanations for the output neurons for the classes 2, 3, 4, 5, 6 respectively, while each column contains the analyses of the different explanation algorithms.

The true label of the image is 3 and also intuitively it resembles a 3, yet it is classified as 2. Can we retrace why the network decided for a 2? Having a closer look, on the first row — which explains the class 2 — the explanation algorithms suggest that the network considers the top and the left stroke as very indicative for a 2, and does not recognize the discontinuity between the center and the lower, left part as contradicting. On the other hand, a look on the second row — which explains a 3 — suggests that according to the explanations the left stroke speaks against the digit being a 3. Potential takeaways from this are that the network does not recognize or does not give enough weight on the continuity of lines or that the dataset does not contain enough digit 3 with such a lower left stroke.

Taking this as an example of how such tools can help to understand a neural network, we would like to note that all the stated points are presumptions — based on the assumption that the explanations are meaningful. But given this leap of faith, our argumentation seems plausible and what a user would expect an explanation algorithm to deliver.

We would also like to note that there are common indicators across different methods, e.g., that the topmost stroke is very indicative for a 2 or that the leftmost stroke is not for a 3. This suggest that the methods base their analysis on similar signals in the network. Yet it is not clear which method performs “best” and this leads us to the next example.

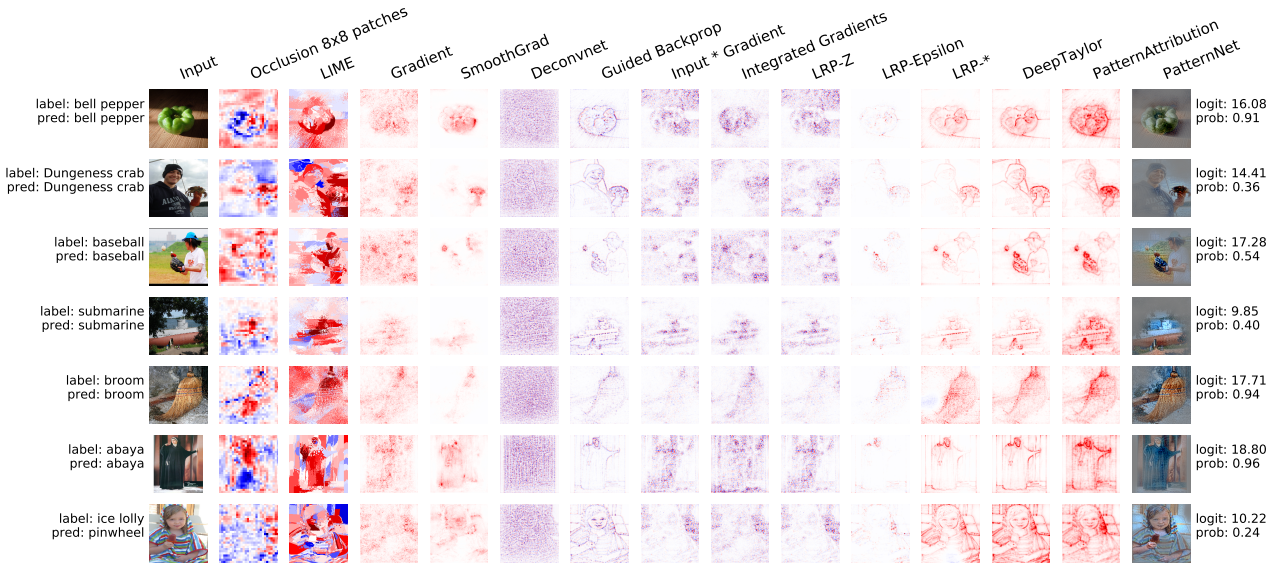

Fig. V.8: Comparing algorithms. *The figure depicts the prediction analysis of a variety of algorithms (columns) for a number of input images (rows) for the VGG16 network (Simonyan and Zisserman, 2014). The true and the predicted label are denoted on the left hand side and the softmax and pre-softmax outputs of the network are printed on the right hand side. LRP-* denotes the configuration from (Lapuschkin et al., 2017). Best viewed in digital and color.*

4.2 Comparing explanation algorithms

For explanation methods there exists no clear evaluation criteria and this makes it inherently hard to find a method that "works" or to choose hyper-parameters (see Section 3.4.2). Therefore we argue for the need of extensive comparisons to identify a suitable method for the task at hand.

Figure V.8 gives an example of such a qualitative comparison and shows the explanation results for a variety of methods (columns) for a set of pictures. We observe that compared to the previous example the analysis results are not as intuitive anymore and we also observe major qualitative differences between the methods. For instance, the algorithms Occlusion and LIME produce distinct heatmaps compared to the other gradient- and propagation-based results. Among this latter group, the results vary in sparseness, but also in which regions the attribution is located. Note that despite its "bad" results, for completeness we added the method DeconvNet (Zeiler and Fergus, 2014).

Consider the image in the last row, which is miss-classified as pinwheel. While one can interpret that some methods indicate the right part of the hood as significant for this decisions, this is merely a speculation and it is hard to make sense of the analyses — revealing the current dilemma of explanation methods and the need for more research. Nevertheless it is important to be clear about such problems and give the user tools to make up her own opinion.

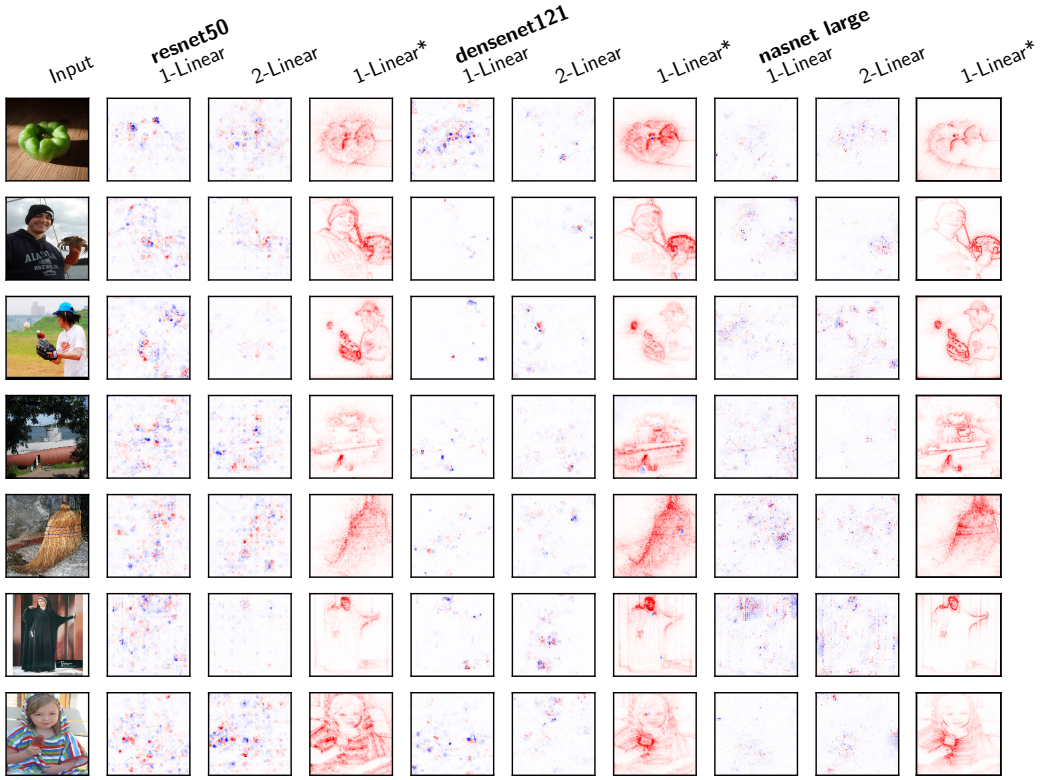

Fig. V.9: LRP batch normalization development. *The different columns show the suggested approaches to handle batch normalization layers for three different network architectures. "1-Linear", "1-Linear*", and "2-Linear" interpret the batch normalization as a one or two linear layers accordingly. We observe that default linearization approaches "1-Linear" and "2-Linear" lead to different results and also for different architectures their results vary qualitatively. The third variant "1-Linear*" was proposed by Hui and Binder (2018) and is described in the text. Best viewed in digital and color.*

4.3 Developing explanation algorithms

Before the development of this library the LRP and Deep Taylor methods were — to the best of our knowledge — never applied to emerging networks such as Inception V3 (Szegedy et al., 2016) or ResNet50 (He et al., 2016). With the new abstractions it became easier to adapt the algorithms to these networks, which are much more complex than VGG16 (Simonyan and Zisserman, 2014), not just due to new layers, but also due to structural differences such as skip-connections.

One of the upcoming issues when applying LRP to such networks was the handling of the batch normalization layer (Ioffe and Szegedy, 2015). As we mentioned in Section 3.2.1 this layer can be seen as one or two linear layers, or even be merged with a neighboring convolutional layer without modifying the prediction. This allows for different ways to linearize the network before applying the explanation algorithm. In Figure V.9 we show different approaches to tackle this problem for the LRP function family. We note that in this plot the heatmaps are scaled to highlight small values by taking the square root and preserving the sign. For all networks we use the LRP-PresetA (Lapuschkin et al., 2017) algorithm and the different columns denote the different batch normalization handling approaches for a given network. For "1-Linear" and "2-Linear" the batch normalization layers are interpreted as one or two linear layers and we use the default LRP epsilon rule (Bach et al., 2015) as back-propagation mapping. The approaches lead to very different results, both between "1-Linear" and "2-Linear" and between different network architectures. We highlight this as an example for the need to test algorithms on a wide range of architectures as well as the for the unsolved question how the linearization of neural networks influences propagation-based explanation algorithms.
The last approach "1-Linear*" is a proposition of Hui and Binder (2018) which uses a modified LRP alpha-beta rule (Bach et al., 2015) to suppress the back-propagated relevance (Bach et al., 2015) in batch normalization layers and is given in Appendix 2.5. This variant results in appealing heatmaps, but we stress that it is theoretically unclear what effect it has on the overall meaning of the explanation. It would be of interest to analyze this effect theoretically, e.g., within the Deep Taylor framework (Montavon et al., 2017).

4.4 Comparing network architectures

Another possible comparative analysis is to examine the explanations for different architectures. This allows on one hand to assess the transferability of explanation methods and on the other hand to inspect the functioning of different networks.
Figure V.10 exemplarily depicts such a comparison for an image for the class "baseball". We observe that the quality of the results for the same algorithm can vary significantly between different architectures, e.g., for some algorithms the results are very sparse for deeper architectures. Moreover, the difference between different algorithms applied to the same network seems to increase with the complexity of the architecture (The complexity increases from the first to the last row).
Nevertheless, we note that explanation algorithms give an indication for the different prediction results of the networks and can be a valuable tool for understanding such networks. A similar approach can be used to monitor the learning of a network during the training.

4.5 Systematic network evaluation

Our last example uses a promising strategy to leverage explanation methods for analysis of networks beyond a single prediction. We evaluate explanations for a whole

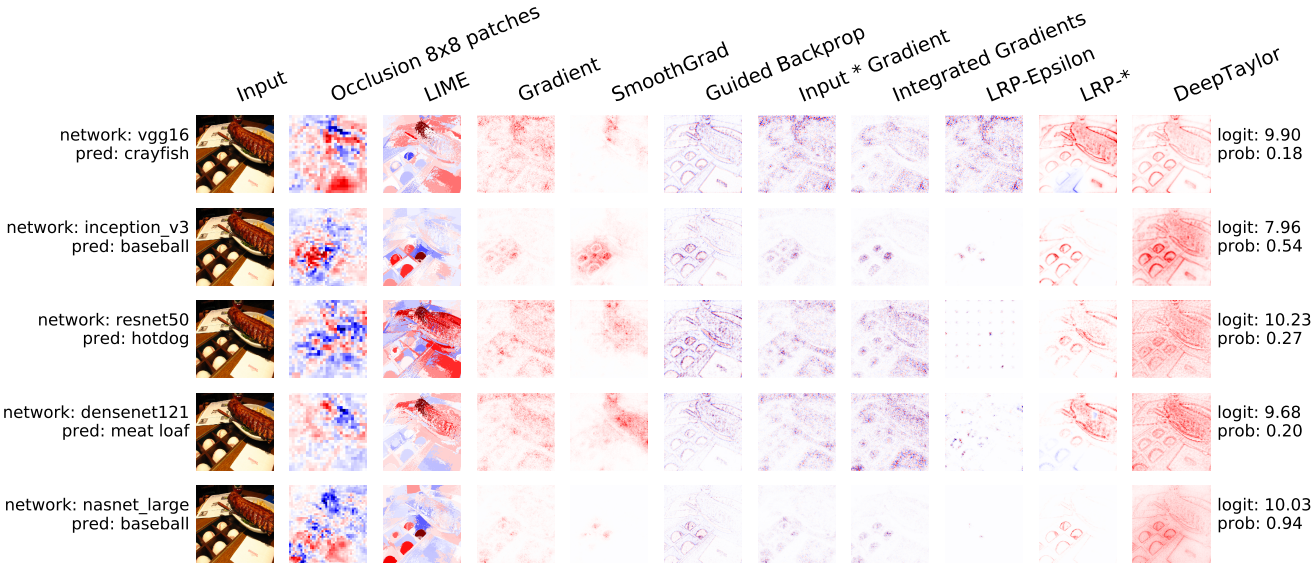

Fig. V.10: Comparing architectures. *The figure depicts the prediction analysis of a variety of algorithms (columns) for a number of neural networks (rows). The true label for this input image is "baseball" and the prediction of the respective network is given on the left hand side. The softmax and pre-softmax outputs of the network are printed on the right hand side. LRP-* denotes the configuration from (Lapuschkin et al., 2017). Best viewed in digital and color.*

dataset to search for classes where the neural network uses (correlated) background features to identify an object. Other examples for such systematic evaluations are, e.g., grouping predictions based on their frequencies (Lapuschkin et al., 2019). These approaches are distinctive in that they do not rely on the miss-classification as signal, i.e., one can detect undesired behavior for samples which are correctly classified by a network.

We use again a VGG16 network and create for each example of the ImageNet 2012 (Deng et al., 2009) validation set a heatmap using the LRP method with the configuration from (Lapuschkin et al., 2017). Then we compute the ratio of the attributions absolute values summed inside and outside of the bounding box, and pick the class with the lowest ration, namely "basketball". A selection of images and their heatmaps is given in Table V.1. The first four images are correctly classified, but one can observe from the heatmaps that the network does not focus on the actual basketball inside the bounding boxes. This suggests the suspicion that the network is not aware of the concept "basketball" as a ball, but rather as a scene. Similarly, in the next three images the basket ball is not identified — leading to wrong predictions. Finally, the last image contains a basketball without any sport scenery and gets miss-classified as ping-pong ball.

Tab. V.1: Bounding box analysis. *The result of our bounding box analysis suggests that the target network does not use features inside the bounding box to predict the class "basketball". The images have all the true label "basketball" and the label beneath an image indicates the predicted class. We note that for none of the images the network relies on the features of a basketball for the prediction, except for the prediction "ping-pong ball". The result suggest that concept "basketball" is a scenery rather than a ball object for the network. Best viewed in digital and color.*

One can argue that a sport scene is a strong indicator for the class "basketball", on the other the bounding boxes make clear that the class addresses a ball rather than a scene and the miss-classified images show that taking the scenery rather than a ball as indicator can be miss-leading. The use of explanation methods can support

developers to identify such flaws of the learning setup caused by, e.g., biased data or networks that rely on the "wrong" features (Lapuschkin et al., 2019).

5 Discussion

In the subsequent discussion we will first focus on general topics and eventually discussion *iNNvestigate* specific matters.

5.1 Challenges

Neural networks come in a large variety. They can be composed of many different layers and be of complex structure (E.g., Figure V.3 shows the sub-blocks of the NASNetA network). Many (propagation-based) explanation methods are designed to handle fully connected layers in the first place, yet to be universally applicable a method and its implementations must be able to scale beyond fully-connected networks and be able to generalize to new layer types. The advantage of methods that treat models as a blackbox is their applicability independent of a network's complexity. On the other hand, they are typically slower and cannot take advantage of high level features like white box methods (Lapuschkin et al., 2017; Selvaraju et al., 2017).
To promote research on white-box methods for complex neural networks it is necessary alleviate researchers from unnecessary implementation efforts. Therefore it is important that tools exist that allow for fast prototyping and let researchers focus on algorithmic developments. One example is the library *iNNvestigate*, which offers an API that allows to modify the backpropagation easily and implementations of many of state-of-the explanation methods ready for advanced neural networks. We showed in Section 3.2.3 how a library like *iNNvestigate* helps to generalize algorithms to various architectures. Such efforts are promising to facilitate research as they make it easier to compare and develop methods as well as facilitate faster adaption to (recent) developments in deep learning.
For instance, despite first attempts (Arras et al., 2017; Ancona et al., 2018) LSTMs (Hochreiter and Schmidhuber, 1997; Sutskever et al., 2014) and attention layers (Vaswani et al., 2017) are still a challenge for most propagation-based explanation methods. Another challenge are architectures discovered automatically with, e.g., neural architecture search (Zoph et al., 2018). They often outperform competitors that were created by human intuition, but are very complex. A successful application of and examination with explanation methods can be a promising way to shed led into their workings. The same reasoning applies to networks like SchNet (Schütt et al., 2017b), WaveNet (Van Den Oord et al., 2016), and AlphaGo (Silver et al., 2016) — which led to breakthroughs in their respective domains and a better understanding of their predictions would reveal valuable knowledge.
Another open research question regarding propagation-based methods concerns the decomposition of network into components. Methods like Deep Taylor Decomposition, LRP, DeepLIFT, DeepSHAPE decompose the network and create an explanation based on the linearization of the respective components. Yet networks can be decomposed in different ways: for instance the sequence of a convolutional and a batch normalization

layer can be treated as two components or be represented as one layer where both are fused. Another example is the treatment of a single batch normalization layer which can be seen as one or as two linear layers. Further examples can be found and it is not clear how the different approaches to decompose a network influence the result of the explanation algorithms. Future research should address this.

5.2 Limitations and outlook

Finally we would like to address limitations of the proposed software package and give an outlook.
First to mention is that not all explanation methods are covered yet. For instance DeepLIFT (Shrikumar et al., 2017) and DeepSHAPE (Lundberg and Lee, 2017) are two missing, propagation-based methods. Furthermore, it would be of value to extend the functionality to algorithms based on different concepts like LIME (Ribeiro et al., 2016) or prediction difference analysis (Zintgraf et al., 2017) to give the user a nearly complete selection of state-of-the-art algorithms for deep neural networks.
Like most research for explanation based methods *iNNvestigate* has a focus on computer vision — which allows for an intuitive validation. While in principle the library is applicable on arbitrary deep neural networks it would be helpful to extend examples and utility functionality to domains like text and natural language processing.
Recent work of Dombrowski et al. (2019) uses the gradient of explanation functions to create adversarial examples. Using solely current features of deep learning frameworks this is rather inefficient, because the libraries not are designed to compute gradients with respect to multiple neurons in one backward pass. The modular design of *iNNvestigate* allows for an integration of the so-called forward gradient computation, which would form a useful feature to make the computations of such gradients efficient. We plan to extend the library accordingly in the future.
Deep learning is a fast changing field and so is the popularity of software frameworks. While Keras and TensorFlow still form a big part of the spectrum, the uprising of PyTorch (Paszke et al., 2017) — especially among researchers — limits the reach of our software package. It remains a question for the future, if the software can be redesigned in order to support both major deep learning frameworks.
A closely connected matter is how a sustainable community beyond the authors around *iNNvestigate* can be created and maintained. For the long term view of the project it is rather important to foster a reasonably sized community of active supporters. An extension for PyTorch could help such growth.

6 Conclusion

With our development of the software *iNNvestigate* we added a missing piece to the (research) toolbox for deep learning. It aims to facilitate the access of non-expert users to a wide range of explanation methods and to support creating algorithms by bundling important functionality in a library. The two central pieces of the software are an intuitive and uniform interface towards the general user and a backend that supports the development of propagation-based algorithms.

Building on this we implemented a wide range of explanation algorithms – including the reference implementations for PatternNet, PatternAttribution (Kindermans et al., 2018) and the LRP methods (Bach et al., 2015). It allows practitioners to easily compare these algorithms and find the most suitable candidate for their task at hand. This is especially important in a field where no clear evaluation metric exists. Furthermore, our work enabled us to apply such algorithms to a number of complex computer vision networks and thereby pointing out issues of existing approaches as well as sparking research to alleviate this. We also report the first results in this direction.
We consider *iNNvestigate* as a first step into the direction of consolidating prediction analysis methods and are confident that its features will trigger more research in this area of Machine Learning. In this light we outlined future challenges for prediction analysis in general and *iNNvestigate* in specific, e.g., the ambiguous linearization of deep networks and the changing software landscape, to mention two. We also hope that library will continue to gain attention and that a sustainable community will grow to support it.

Chapter VI

Summary and conclusions

This thesis was initially motivated by the emerging challenges in Machine Learning that are posed by the increase of data availability, computing resources, and subsequent model complexity. In order to pursue this motivation we selected three distinct problem settings and presented approaches towards their solution. The proposed algorithms contribute to several fields within the broad spectrum of Machine Learning by making use of techniques from the mathematical/methodological and engineering domains.

Chapter III & IV The first two research questions were driven by the emergence of larger datasets, once regarding the number of classes and once regarding the number of samples. In both cases we adapted and examined methods in larger setups that showed successful results in smaller settings.
In the first case this concerned all-in-one SVMs and was approached by developing solutions for two such formulations that distribute computation and model parameter evenly on different computing instances. Based on this implementation we were able to show that one formulation is indeed able to perform better on large text problems than one-vs.-rest SVMs, while the analysis also revealed that the other formulation does not perform well on such problems. Drawbacks of the proposed solutions are their limitations to multi-class problems and the negative influence of shrinking on the distributed computation.
Outlook: Recalling that one-vs.-rest approaches outperform *approximated* all-in-one SVMs in large, multi-label tasks, the contradicting results of this work raise the question whether *exact* solvers can do better. One way to inquiry this is to extend our proposed algorithm to multi-label problems. A different research direction is given in the intersection of SVMs and neural networks, namely to explore if the expensive training in deep learning can benefit from shrinking techniques which are applied in the dual optimization of SVMs. Finally, another downside of SVMs is the linear prediction time and it is of interest to explore if the prediction with highly sparse matrices can be performed faster, yet still accurate enough, with maximum inner product search algorithms.
In the second case random feature approximations for kernel methods appeared to be wasteful regarding the potential parameter space. We developed an efficient optimization scheme to showcase our claim and, furthermore, shed light into the connection of approximated kernel machines and neural network by designing experiments that reveal the influence of data- and task-agnostic basis functions. Downsides of our proposed optimization methods are the limited applicability of (Gaussian) kernels and, moreover, the quadratic scaling factor of bases adapted to a kernel function.

Outlook: We suggest future work should inquiry in more detail how neural networks can take advantage of kernel methods by, e.g., using the supervised adaptive basis scheme to create (transferable) embeddings. For instance, in the setting of Chapter III a scalable adaptation of this scheme to large label spaces could facilitate the learning of embeddings that reflect the long tail of the class distribution better — based on the consideration that this kernel can put stress on the relation between rare or confusing classes.

Conclusion: In both cases we proposed efficient solutions and validated the initial hypothesis with an empirical evaluation. The novel results provide insights in two specific areas of Machine Learning: the effectiveness of all-in-one SVMs for extreme classification and the inefficiency of random features for approximated kernel machines. We are confident that they help to facilitate the understanding of the respective domains and inspire further research in those directions.

Chapter V In contrast, our last contribution is meant for direct application. For this work we approach another emerging issue in Machine Learning, namely increasingly complex models and prediction processes. To meet the wish and need for a better understanding and retracability many propagation-based methods were proposed in literature and exhibit promising results. A drawback is the lack of efficient software for many methods and the missing evidence of their effectiveness on recent and advanced neural networks. We extracted patterns for efficient software in this domain and created the software package *iNNvestigate*, whose purpose is to tackle this issue. Its key features are an intuitive interface lowering the access burden to such methods and a modular library design enabling efficient implementations. This allowed us to implement many analysis algorithms and, furthermore, to apply and extend them to new networks. A limitation of the library is the choice of the underlying framework Keras and it is of interest to research if the software can be extended to other deep learning frameworks.

Outlook: These first successful applications of the software library posed new research questions: How can algorithms be adapted best to new architectures and, closely related, what is the influence of the ambiguous decomposition of neural networks into presumably linear components — a technique many propagation-based methods rely on. In the future, it will be fruitful to extend the method selection of the library further and to research how (semi-)automated algorithms, e.g., for detecting if a network relies on irrelevant features, can be incorporated efficiently. Collecting feedback from the growing user base and including more developers in a subsequent re-factoring of the software can be a promising way to attract a larger number of contributors and to adapt the library to (future) needs.

Conclusion: In contrast to the previous contributions in this thesis *iNNvestigate* targets a more applied audience. The repository of the software has already gained significant attention in a short amount of time. We hope that in the future this will accelerate and more contributors will help to keep this software useful for many in the long term.

Conclusion In conclusion, this thesis tackled three distinct challenges in Machine Learning posed by the increase of data and computing power. Our contributions constitute of algorithm development, software design, implementations, and empirical analyses. The evaluations showed that our approaches yield improvements in terms of accuracy, model compactness, accessibility, and scalability in various dimensions. Our results give new insights in the respective domains that also lead to a number of new research questions. Lastly, with the presented library *iNNvestigate* we provide a novel and practical tool to the community.

Bibliography

Abadi, M, A Chu, I Goodfellow, HB McMahan, I Mironov, K Talwar, and L Zhang (2016a). "Deep learning with differential privacy". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 308–318.

Abadi, M, P Barham, J Chen, Z Chen, A Davis, J Dean, M Devin, S Ghemawat, G Irving, M Isard, et al. (2016b). "Tensorflow: a system for large-scale machine learning." In: *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation* 16, pp. 265–283.

Agarwal, A, O Chapelle, M Dudík, and J Langford (2014). "A Reliable Effective Terascale Linear Learning System". In: *Journal of Machine Learning Research* 15, pp. 1111–1133.

Alber, M (2019). "Software and application patterns for explanation methods". To appear in: *Interpretable AI: Interpreting, Explaining and Visualizing Deep Learning*. Springer.

Alber, M, J Zimmert, U Dogan, and M Kloft (2016). "Distributed optimization of multi-class SVMs". In: *Neural Information Processing Systems 2016 - Extreme Classification workshop*.

Alber, M, P-J Kindermans, KT Schütt, K-R Müller, and F Sha (2017a). "An Empirical Study on The Properties of Random Bases for Kernel Methods". In: *Advances in Neural Information Processing Systems 30*, pp. 2763–2774.

Alber, M, J Zimmert, U Dogan, and M Kloft (2017b). "Distributed optimization of multi-class SVMs". In: *PLOS ONE* 12.6, pp. 1–18.

Alber, M, I Bello, B Zoph, P-J Kindermans, P Ramachandran, and Q Le (2018a). "Backprop Evolution". In: *International Conference on Machine Learning 2018 - AutoML workshop*.

Alber, M, S Lapuschkin, P Seegerer, M Hägele, KT Schütt, G Montavon, W Samek, K-R Müller, S Dähne, and P-J Kindermans (2018b). "How to iNNvestigate neural networks' predictions!" In: *Neural Information Processing Systems 2018 - Machine Learning Open Source Software workshop*.

Alber, M, S Lapuschkin, P Seegerer, M Hägele, KT Schütt, G Montavon, W Samek, K-R Müller, S Dähne, and P-J Kindermans (2019). "iNNvestigate neural networks!" To appear in: *Journal of Machine Learning Research*.

Alexandrov, A, R Bergmann, S Ewen, J-C Freytag, F Hueske, A Heise, O Kao, M Leich, U Leser, V Markl, et al. (2014). "The stratosphere platform for big data analytics". In: *The International Journal on Very Large Data Bases* 23.6, pp. 939–964.

Allwein, EL, RE Schapire, and Y Singer (2001). "Reducing multiclass to binary: A unifying approach for margin classifiers". In: *Journal of Machine Learning Research* 1, pp. 113–141.

Ancona, M, E Ceolini, C Öztireli, and M Gross (2018). "Towards better understanding of gradient-based attribution methods for Deep Neural Networks". In: *International Conference on Learning Representations*.

Andrews, GR (2000). *Foundations of multithreaded, parallel, and distributed programming*. Addison-Wesley.

Arras, L, G Montavon, K-R Müller, and W Samek (2017). "Explaining Recurrent Neural Network Predictions in Sentiment Analysis". In: *Proceedings of the EMNLP'17 Workshop on Computational Approaches to Subjectivity, Sentiment & Social Media Analysis*, pp. 159–168.

Asuncion, A and D Newman (2007). *UCI machine learning repository*.

Babbar, R and B Schölkopf (2017). "DiSMEC: Distributed Sparse Machines for Extreme Multi-label Classification". In: *Proceedings of the 10th ACM International Conference on Web Search and Data Mining*, pp. 721–729.

Babbar, R and B Schölkopf (2018). "Adversarial Extreme Multi-label Classification". In: *arXiv preprint arXiv:1803.01570*.

Babbar, R, K Maundet, and B Schölkopf (2016). "TerseSVM: A Scalable Approach for Learning Compact Models in Large-scale Classification". In: *Proceedings of the 2016 SIAM International Conference on Data Mining*, pp. 234–242.

Bach, S, A Binder, G Montavon, F Klauschen, K-R Müller, and W Samek (2015). "On Pixel-wise Explanations for Non-Linear Classifier Decisions by Layer-wise Relevance Propagation". In: *PLOS ONE* 10.7, pp. 1–46.

Baehrens, D, T Schroeter, S Harmeling, M Kawanabe, K Hansen, and K-R Müller (2010). "How to explain individual classification decisions". In: *Journal of Machine Learning Research* 11, pp. 1803–1831.

Bahdanau, D, K Cho, and Y Bengio (2015). "Neural machine translation by jointly learning to align and translate". In: *International Conference on Learning Representations*.

Baudat, G and F Anouar (2000). "Generalized discriminant analysis using a kernel approach". In: *Neural Computation* 12.10, pp. 2385–2404.

Behnel, S, R Bradshaw, C Citro, L Dalcin, DS Seljebotn, and K Smith (2011). "Cython: The best of both worlds". In: *Computing in Science & Engineering* 13.2, pp. 31–39.

Bengio, S, J Weston, and D Grangier (2010). "Label embedding trees for large multi-class tasks". In: *Advances in Neural Information Processing Systems 23*, pp. 163–171.

Bergstra, J, O Breuleux, F Bastien, P Lamblin, R Pascanu, G Desjardins, J Turian, D Warde-Farley, and Y Bengio (2010). "Theano: A CPU and GPU math expression compiler". In: *Proceedings of the Python for scientific computing conference* 4.3, pp. 3–11.

Bertsekas, DP, ML Homer, DA Logan, and SD Patek (1995). *Nonlinear programming*. Athena scientific.

Bhatia, K, H Jain, P Kar, M Varma, and P Jain (2015). "Sparse local embeddings for extreme multi-label classification". In: *Advances in Neural Information Processing Systems 28*, pp. 730–738.

Binder, A et al. (2018). "Towards computational fluorescence microscopy: Machine learning-based integrated prediction of morphological and molecular tumor profiles". In: *arXiv preprint arXiv:1805.11178*.

Bishop, CM (2006). *Pattern recognition and machine learning*. Springer.

Bishop, CM et al. (1995). *Neural networks for pattern recognition*. Oxford university press.

Blackard, JA and DJ Dean (2000). "Comparative Accuracies of Artificial Neural Networks and Discriminant Analysis in Predicting Forest Cover Types from Cartographic Variables". In: *Computers and Electronics in Agriculture* 24.3, pp. 131–151.

Blackford, LS, A Petitet, R Pozo, K Remington, RC Whaley, J Demmel, J Dongarra, I Duff, S Hammarling, G Henry, et al. (2002). "An Updated Set of Basic Linear Algebra Subprograms (BLAS)". In: *ACM Transactions on Mathematical Software* 28.2, pp. 135–151.

Boden, C, T Rabl, and V Markl (2018). "Distributed Machine Learning-but at what COST". In: *Neural Information Processing Systems 2018 - Machine Learning Systems workshop*.

Bondy, JA and USR Murty (1976). *Graph theory with applications*. Elsevier Science.

Borwein, J and AS Lewis (2010). *Convex analysis and nonlinear optimization: theory and examples*. Springer.

Bottou, L (2010). "Large-scale machine learning with stochastic gradient descent". In: *Proceedings of COMPSTAT'2010*, pp. 177–186.

Bottou, L (2012). "Stochastic gradient descent tricks". In: *Neural networks: Tricks of the trade*, pp. 421–436.

Boyd, S, N Parikh, E Chu, B Peleato, and J Eckstein (2011). "Distributed optimization and statistical learning via the alternating direction method of multipliers". In: *Foundations and Trends in Machine Learning* 3.1, pp. 1–122.

Brandl, S, D Lassner, and M Alber (2019). "Balancing the composition of word embeddings". Submitted to: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*.

Braun, ML, JM Buhmann, and K-R Müller (2008). "On Relevant Dimensions in Kernel Feature Spaces". In: *Journal of Machine Learning Research* 9, pp. 1875–1908.

Breiman, L (1996). "Bagging predictors". In: *Machine Learning* 24.2, pp. 123–140.

Breiman, L (2001). "Random forests". In: *Machine Learning* 45.1, pp. 5–32.

Carbone, P, A Katsifodimos, S Ewen, V Markl, S Haridi, and K Tzoumas (2015). "Apache Flink™: Stream and Batch Processing in a Single Engine". In: *IEEE Data Engineering Bulletin* 38.4, pp. 28–38.

Carratino, L, A Rudi, and L Rosasco (2018). "Learning with SGD and Random Features". In: *Advances in Neural Information Processing Systems 31*, pp. 10213–10224.

Chan, K-H, S-K Im, W Ke, and N-L Lei (2018). "SinP [N]: A Fast Convergence Activation Function for Convolutional Neural Networks". In: *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion*, pp. 365–369.

Chang, F, J Dean, S Ghemawat, WC Hsieh, DA Wallach, M Burrows, T Chandra, A Fikes, and RE Gruber (2008). "Bigtable: A distributed storage system for structured data". In: *ACM Transactions on Computer Systems* 26.2, Article: 4, pp. 1–26.

Cheng, H-T, L Koc, J Harmsen, T Shaked, T Chandra, H Aradhye, G Anderson, G Corrado, W Chai, M Ispir, et al. (2016). "Wide & deep learning for recommender systems". In: *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems*, pp. 7–10.

Chiang, W-L, M-C Lee, and C-J Lin (2016). "Parallel Dual Coordinate Descent Method for Large-scale Linear Classification in Multi-core Environments". In: *Proceedings of the 22th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1485–1494.

Chmiela, S, A Tkatchenko, HE Sauceda, I Poltavsky, KT Schütt, and K-R Müller (2017). "Machine learning of accurate energy-conserving molecular force fields". In: *Science Advances* 3.5, ID: e1603015.

Chmiela, S, HE Sauceda, K-R Müller, and A Tkatchenko (2018). "Towards exact molecular dynamics simulations with machine-learned force fields". In: *Nature communications* 9.1, ID: 3887.

Cho, Y and LK Saul (2009). "Kernel methods for deep learning". In: *Advances in Neural Information Processing Systems 22*, pp. 342–350.

Chollet, F et al. (2015). *Keras*. https://github.com/fchollet/keras.

Chollet, F (2017). "Xception: Deep Learning with Depthwise Separable Convolutions". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1800–1807.

Choromanska, AE and J Langford (2015). "Logarithmic Time Online Multiclass prediction". In: *Advances in Neural Information Processing Systems 28*, pp. 55–63.

Coates, A, AY Ng, and H Lee (2011). "An Analysis of Single-Layer Networks in Unsupervised Feature Learning". In: *15th International Conference on Artificial Intelligence and Statistics*, pp. 215–223.

Codd, EF (1970). "A relational model of data for large shared data banks". In: *Communications of the ACM* 13.6, pp. 377–387.

Cormen, TH, CE Leiserson, RL Rivest, and C Stein (2009). *Introduction to Algorithms*. MIT press.

Cortes, C and VN Vapnik (1995). "Support-vector networks". In: *Machine Learning* 20.3, pp. 273–297.

Crammer, K and Y Singer (2002). "On the algorithmic implementation of multiclass kernel-based vector machines". In: *Journal of Machine Learning Research* 2, pp. 265–292.

Cristianini, N, A Elisseeff, J Shawe-Taylor, and J Kandola (2001). "On kernel-target alignment". In: *Advances in Neural Information Processing Systems 14*, pp. 367–373.

Dagum, L and R Enon (1998). "OpenMP: an industry standard API for shared-memory programming". In: *Computational Science & Engineering* 5.1, pp. 46–55.

Dai, B, B Xie, N He, Y Liang, A Raj, M-FF Balcan, and L Song (2014). "Scalable kernel methods via doubly stochastic gradients". In: *Advances in Neural Information Processing Systems 27*, pp. 3041–3049.

Dalcin, LD, RR Paz, PA Kler, and A Cosimo (2011). "Parallel distributed computing using python". In: *Advances in Water Resources* 34.9, pp. 1124–1139.

Dalvi, N, P Domingos, S Sanghai, D Verma, et al. (2004). "Adversarial classification". In: *Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 99–108.

Dean, J and S Ghemawat (2008). "MapReduce: Simplified data processing on large clusters". In: *Communications of the ACM* 51.1, pp. 107–113.

DeCandia, G, D Hastorun, M Jampani, G Kakulapati, A Lakshman, A Pilchin, S Sivasubramanian, P Vosshall, and W Vogels (2007). "Dynamo: Amazon's highly available key-value store". In: *ACM SIGOPS Operating Systems Review* 41.6, pp. 205–220.

Deng, J, W Dong, R Socher, L-J Li, K Li, and L Fei-Fei (2009). "Imagenet: A large-scale hierarchical image database". In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 248–255.

Deng, J, S Satheesh, AC Berg, and F Li (2011). "Fast and balanced: Efficient label tree learning for large scale object recognition". In: *Advances in Neural Information Processing Systems 24*, pp. 567–575.

Do, T-N (2014). "Parallel multiclass stochastic gradient descent algorithms for classifying million images with very-high-dimensional signatures into thousands classes". In: *Vietnam Journal of Computer Science* 1.2, pp. 107–115.

Dombrowski, A-K, M Alber, CJ Anders, M Ackermann, K-R Müller, and P Kessel (2019). "Could not get lock /var/lib/dpkg/lock-frontend". Submitted to: *Advances in Neural Information Processing Systems 33*.

Domingos, P (2012). "A Few Useful Things to Know About Machine Learning". In: *Communications of the ACM* 55.10, pp. 78–87.

Doğan, Ü, T Glasmachers, and C Igel (2016). "A Unified View on Multi-class Support Vector Classification". In: *Journal of Machine Learning Research* 17, pp. 1–32.

Drineas, P and MW Mahoney (2005). "On the Nyström method for approximating a Gram matrix for improved kernel-based learning". In: *Journal of Machine Learning Research* 6, pp. 2153–2175.

Duda, RO, PE Hart, and DG Stork (2012). *Pattern classification*. John Wiley & Sons.

Fan, R, K Chang, C Hsieh, X Wang, and C Lin (2008). "LIBLINEAR: A library for large linear classification". In: *Journal of Machine Learning Research* 9, pp. 1871–1874.

Fawcett, T (2006). "An introduction to ROC analysis". In: *Pattern recognition letters* 27.8, pp. 861–874.

Feng, C, Q Hu, and S Liao (2015). "Random Feature Mapping with Signed Circulant Matrix Projection." In: *2015 International Joint Conferences on Artificial Intelligence*, pp. 3490–3496.

Flynn, MJ (1972). "Some computer organizations and their effectiveness". In: *IEEE transactions on computers* 100.9, pp. 948–960.

Forero, PA, A Cano, and GB Giannakis (2010). "Consensus-Based Distributed Support Vector Machines". In: *Journal of Machine Learning Research* 11, pp. 1663–1707.

Freund, Y and RE Schapire (1996). "Experiments with a New Boosting Algorithm". In: *Proceedings of the 13th International Conference on Machine Learning*, pp. 148–156.

Frey, PW and DJ Slate (1991). "Letter recognition using Holland-style adaptive classifiers". In: *Machine Learning* 6.2, pp. 161–182.

Gao, T and D Koller (2011). "Discriminative learning of relaxed hierarchy for large-scale visual recognition". In: *Proceedings of the 2011 International Conference on Computer Vision*, pp. 2072–2079.

Gashler, MS and SC Ashmore (2014). "Training deep fourier neural networks to fit time-series data". In: *Proceedings of the 9th International Conference on Intelligent Computing*, pp. 48–55.

Glorot, X, A Bordes, and Y Bengio (2011). "Deep sparse rectifier neural networks". In: *15th International Conference on Artificial Intelligence and Statistics*, pp. 315–323.

Gondal, WM, JM Köhler, R Grzeszick, GA Fink, and M Hirsch (2017). "Weakly-supervised localization of diabetic retinopathy lesions in retinal fundus images". In: *2017 IEEE International Conference on Image Processing*, pp. 2069–2073.

Goodfellow, I, J Shlens, and C Szegedy (2014). "Explaining and Harnessing Adversarial Examples". In: *International Conference on Learning Representations*.

Goodfellow, I, Y Bengio, and A Courville (2016). *Deep learning*. MIT press.

Gopal, S and Y Yang (2013a). "Distributed training of Large-scale Logistic models." In: *Proceedings of the 30th International Conference on Machine Learning*, pp. 289–297.

Gopal, S and Y Yang (2013b). "Recursive regularization for large-scale classification with hierarchical and graphical dependencies". In: *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 257–265.

Govada, A, S Ranjani, A Viswanathan, and S Sahay (2015a). "A Novel Approach to Distributed Multi-Class SVM". In: *arXiv preprint arXiv:1512.01993*.

Govada, A, B Gauri, and SK Sahay (2015b). "Distributed Multi Class SVM for Large Data Sets". In: *Proceedings of the 3rd International Symposium on Women in Computing and Informatics*, pp. 54–58.

Grave, E, A Joulin, M Cissé, H Jégou, et al. (2017). "Efficient softmax approximation for GPUs". In: *Proceedings of the 34th International Conference on Machine Learning*, pp. 1302–1310.

Gretton, A, O Bousquet, AJ Smola, and B Schölkopf (2005). "Measuring statistical dependence with Hilbert-Schmidt norms". In: *Algorithmic Learning Theory*, pp. 63–77.

Gropp, W, E Lusk, N Doss, and A Skjellum (1996). "A high-performance, portable implementation of the MPI message passing interface standard". In: *Parallel Computing* 22.6, pp. 789–828.

Guermeur, Y (2007). "VC Theory for Large Margin Multi-Category Classifiers". In: *Journal of Machine Learning Research* 8, pp. 2551–2594.

Gupta, MR, S Bengio, and J Weston (2014). "Training highly multiclass classifiers." In: *Journal of Machine Learning Research* 15, pp. 1461–1492.

Guyon, I, SR Gunn, A Ben-Hur, and G Dror (2004). "Result Analysis of the NIPS 2003 Feature Selection Challenge." In: *Advances in Neural Information Processing Systems 17*, pp. 545–552.

Han, X and AC Berg (2012). "DCMSVM: Distributed parallel training for single-machine multiclass classifiers". In: *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3554–3561.

Haufe, S, F Meinecke, K Görgen, S Dähne, J-D Haynes, B Blankertz, and F Bießmann (2014). "On the interpretation of weight vectors of linear models in multivariate neuroimaging". In: *Neuroimage* 87, pp. 96–110.

He, K, X Zhang, S Ren, and J Sun (2016). "Deep residual learning for image recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770–778.

Hill, SI and A Doucet (2007). "A framework for kernel-based multi-category classification". In: *Journal of Artificial Intelligence Research* 30.1, pp. 525–564.

Ho, Q, J Cipar, H Cui, S Lee, JK Kim, PB Gibbons, GA Gibson, G Ganger, and EP Xing (2013). "More effective distributed ml via a stale synchronous parallel parameter server". In: *Advances in Neural Information Processing Systems 26*, pp. 1223–1231.

Hochreiter, S and J Schmidhuber (1997). "Long Short-Term Memory". In: *Neural Computation* 9.8, pp. 1735–1780.

Hsu, CW and CJ Lin (2002). "A comparison of methods for multiclass support vector machines". In: *IEEE Transactions on Neural Networks* 13.

Huang, G, Z Liu, L v. d. Maaten, and KQ Weinberger (2017). "Densely Connected Convolutional Networks". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2261–2269.

Huang, P-S, H Avron, TN Sainath, V Sindhwani, and B Ramabhadran (2014). "Kernel methods match deep neural networks on timit". In: *2014 IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 205–209.

Hui, YW and A Binder (2018). Personal communication.

Hull, JJ (1994). "A database for handwritten text recognition research". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 16.5, pp. 550–554.

Igel, C, T Glasmachers, and V Heidrich-Meisner (2008). "Shark". In: *Journal of Machine Learning Research* 9, pp. 993–996.

Ioffe, S and C Szegedy (2015). "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *Proceedings of the 32th International Conference on Machine Learning*, pp. 448–456.

Jain, H, Y Prabhu, and M Varma (2016). "Extreme multi-label loss functions for recommendation, tagging, ranking & other missing label applications". In: *Proceedings of the 22th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 935–944.

Jasinska, K, K Dembczynski, R Busa-Fekete, K Pfannschmidt, T Klerx, and E Hullermeier (2016). "Extreme f-measure maximization using sparse probability

estimates". In: *Proceedings of the 33th International Conference on Machine Learning*, pp. 1435–1444.

Jenssen, R, M Kloft, A Zien, S Sonnenburg, and K-R Müller (2012). "A scatter-based prototype framework and multi-class extension of support vector machines". In: *PLOS ONE* 7.10, pp. 1–16.

Joachims, T, T Finley, and C-NJ Yu (2009). "Cutting-plane Training of Structural SVMs". In: *Machine Learning* 77.1, pp. 27–59.

Jones, E, T Oliphant, and P Peterson (2014). *SciPy: Open source scientific tools for Python*. http://www.scipy.org/.

Jouppi, NP, C Young, N Patil, D Patterson, G Agrawal, R Bajwa, S Bates, S Bhatia, N Boden, A Borchers, et al. (2017). "In-datacenter performance analysis of a tensor processing unit". In: *ACM/IEEE 44th Annual International Symposium on Computer Architecture*, pp. 1–12.

Karush, W (1939). "Minima of functions of several variables with inequalities as side constraints". In: *M. Sc. Dissertation. Deptartement of Mathematics, Univeristy of Chicago*.

Keerthi, SS, S Sundararajan, K-W Chang, C-J Hsieh, and C-J Lin (2008). "A Sequential Dual Method for Large Scale Multi-class Linear Svms". In: *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 408–416.

Kindermans, P-J, KT Schütt, K-R Müller, and S Dähne (2016). "Investigating the influence of noise and distractors on the interpretation of neural networks". In: *Neural Information Processing Systems 2016 - Interpretable Machine Learning for Complex Systems workshop*.

Kindermans, P-J, S Hooker, J Adebayo, M Alber, KT Schütt, S Dähne, D Erhan, and B Kim (2017). "The (Un)reliability of saliency methods". In: *Neural Information Processing Systems 2017 - Interpreting, Explaining and Visualizing Deep Learning - Now what? workshop*.

Kindermans, P-J, KT Schütt, M Alber, K-R Müller, D Erhan, B Kim, and S Dähne (2018). "Learning how to explain neural networks: PatternNet and PatternAttribution". In: *International Conference on Learning Representations*.

Kindermans, P-J, S Hooker, J Adebayo, M Alber, KT Schütt, S Dähne, D Erhan, and B Kim (2019). "The (Un)reliability of saliency methods". To appear in: *Interpretable AI: Interpreting, Explaining and Visualizing Deep Learning*. Springer.

Kingma, D and JB Adam (2015). "A Method for Stochastic Optimisation". In: *International Conference on Learning Representations*.

Kohavi, R (1996). "Scaling Up the Accuracy of Naive-Bayes Classifiers: A Decision-Tree Hybrid." In: *Proceedings of the 2th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 202–207.

Korbar, B, AM Olofson, AP Miraflor, CM Nicka, MA Suriawinata, L Torresani, AA Suriawinata, and S Hassanpour (2017). "Looking Under the Hood: Deep Neural Network Visualization to Interpret Whole-Slide Image Analysis Outcomes for Colorectal Polyps". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 821–827.

Kotikalapudi, R and contributors (2017). *keras-vis*. https://github.com/raghakot/keras-vis.

Kreutzer, M, G Hager, G Wellein, H Fehske, and AR Bishop (2014). "A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units". In: *SIAM Journal on Scientific Computing* 36.5, pp. 401–423.

Krizhevsky, A, I Sutskever, and GE Hinton (2012). "Imagenet classification with deep convolutional neural networks". In: *Advances in Neural Information Processing Systems 25*, pp. 1097–1105.

Kuhn, HW and AW Tucker (1951). "Nonlinear Programming". In: *Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability*, pp. 481–492.

Kurakin, A, I Goodfellow, and S Bengio (2016). "Adversarial machine learning at scale". In: *International Conference on Learning Representations*.

Lakshman, A and P Malik (2010). "Cassandra: a decentralized structured storage system". In: *ACM SIGOPS Operating Systems Review* 44.2, pp. 35–40.

Lapuschkin, S, A Binder, G Montavon, K-R Müller, and W Samek (2016a). "Analyzing Classifiers: Fisher Vectors and Deep Neural Networks". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2912–2920.

Lapuschkin, S, A Binder, G Montavon, K-R Müller, and W Samek (2016b). "The Layer-wise Relevance Propagation Toolbox for Artificial Neural Networks". In: *Journal of Machine Learning Research* 17, pp. 3938–3942.

Lapuschkin, S, A Binder, K-R Müller, and W Samek (2017). "Understanding and Comparing Deep Neural Networks for Age and Gender Classification". In: *Proceedings of the ICCV'17 Workshop on Analysis and Modeling of Faces and Gestures*.

Lapuschkin, S, S Wäldchen, A Binder, G Montavon, W Samek, and K-R Müller (2019). "Unmasking Clever Hans predictors and assessing what machines really learn". In: *Nature Communcations* 10, ID: 1096.

Làzaro-Gredilla, M, J Quinonero-Candela, CE Rasmussen, and AR Figueiras-Vidal (2010). "Sparse Spectrum Gaussian Process Regression". In: *Journal of Machine Learning Research* 11, pp. 1865–1881.

Le, Q, T Sarlos, and AJ Smola (2013). "Fastfood – Computing Hilbert Space Expansions in loglinear time". In: *Journal of Machine Learning Research* 28, pp. 244–252.

LeCun, YA, L Bottou, Y Bengio, and P Haffner (1998a). "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11, pp. 2278–2324.

LeCun, YA, C Cortes, and CJ Burges (1998b). *The MNIST database of handwritten digits*. http://yann.lecun.com/exdb/mnist/.

LeCun, YA, L Bottou, GB Orr, and K-R Müller (2012). "Efficient backprop". In: *Neural networks: Tricks of the trade*, pp. 9–48.

LeCun, YA, Y Bengio, and GE Hinton (2015). "Deep Learning". In: *Nature* 521.7553, pp. 436–444.

Lee, C-p and D Roth (2015). "Distributed box-constrained quadratic optimization for dual linear SVM". In: *Proceedings of the 32th International Conference on Machine Learning*, pp. 987–996.

Lee, Y, Y Lin, and G Wahba (2004). "Multicategory Support Vector Machines: Theory and Application to the Classification of Microarray Data and Satellite Radiance Data". In: *Journal of the American Statistical Association* 99.465, pp. 67–82.

Li, M, DG Andersen, JW Park, AJ Smola, A Ahmed, V Josifovski, J Long, EJ Shekita, and B-Y Su (2014). "Scaling distributed machine learning with the parameter server". In: *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, pp. 583–598.

Lipton, ZC (2016). "The mythos of model interpretability". In: *International Conference on Machine Learning 2016 - Human Interpretability in Machine Learning workshop*.

Liu, J, W-C Chang, Y Wu, and Y Yang (2017). "Deep learning for extreme multi-label text classification". In: *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 115–124.

Liu, Y (2007). "Fisher consistency of multicategory support vector machines". In: *11th International Conference on Artificial Intelligence and Statistics*, pp. 289–296.

Lodi, S, R Nanculef, and C Sartori (2010). "Single-pass distributed learning of multi-class svms using core-sets". In: *Proceedings of the 2010 SIAM International Conference on Data Mining*, pp. 257–268.

Lu, Z, A May, K Liu, AB Garakani, D Guo, A Bellet, L Fan, M Collins, B Kingsbury, M Picheny, and F Sha (2014). "How to scale up kernel methods to be as good as deepneural nets". In: *arXiv preprint arXiv:1411.4000*.

Lundberg, SM and S-I Lee (2017). "A unified approach to interpreting model predictions". In: *Advances in Neural Information Processing Systems 30*, pp. 4765–4774.

Madzarov, G, D Gjorgjevikj, and I Chorbev (2009). "A Multi-class SVM Classifier Utilizing Binary Decision Tree." In: *Informatica (Slovenia)* 33.2, pp. 225–233.

Mahajan, D, SS Keerthi, and S Sellamanickam (2018). "A distributed block coordinate descent method for training l1 regularized linear classifiers". In: *Journal of Machine Learning Research* 18, pp. 1–32.

McCulloch, WS and W Pitts (1943). "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5.4, pp. 115–133.

Mika, S, G Rätsch, J Weston, B Schölkopf, and K-R Müller (1999). "Fisher discriminant analysis with kernels". In: *Proceedings of the 1999 IEEE signal processing society workshop*, pp. 41–48.

Mikolov, T, I Sutskever, K Chen, GS Corrado, and J Dean (2013). "Distributed representations of words and phrases and their compositionality". In: *Advances in Neural Information Processing Systems 26*, pp. 3111–3119.

Montavon, G, ML Braun, and K-R Müller (2011). "Kernel analysis of deep networks". In: *Journal of Machine Learning Research* 12, pp. 2563–2581.

Montavon, G, GB Orr, and K-R Müller (2012). *Neural Networks: Tricks of the Trade*. Springer.

Montavon, G, M Rupp, V Gobre, A Vazquez-Mayagoitia, K Hansen, A Tkatchenko, K-R Müller, and OA Von Lilienfeld (2013). "Machine learning of molecular electronic properties in chemical compound space". In: *New Journal of Physics* 15.9, ID: 095003.

Montavon, G, S Bach, A Binder, W Samek, and K-R Müller (2017). "Explaining NonLinear Classification Decisions with Deep Taylor Decomposition". In: *Pattern Recognition* 65, pp. 211–222.

Montavon, G, W Samek, and K-R Müller (2018). "Methods for interpreting and understanding deep neural networks". In: *Digital Signal Processing* 73, pp. 1–15.

Montufar, GF, R Pascanu, K Cho, and Y Bengio (2014). "On the number of linear regions of deep neural networks". In: *Advances in Neural Information Processing Systems 27*, pp. 2924–2932.

Moody, J and CJ Darken (1989). "Fast learning in networks of locally-tuned processing units". In: *Neural Computation* 1.2, pp. 281–294.

Mordvintsev, A, C Olah, and M Tyka (2015). *Inceptionism: Going deeper into neural networks*. https://ai.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html.

Müller, K-R, AJ Smola, G Rätsch, B Schölkopf, J Kohlmorgen, and VN Vapnik (1997). "Predicting Time Series with Support Vector Machines". In: *Proceedings of the 7th International Conference on Artificial Neural Networks*, pp. 999–1004.

Müller, K-R, AJ Smola, G Rätsch, B Schölkopf, J Kohlmorgen, and VN Vapnik (1999). "Using support vector machines for time series prediction". In: *Advances in Kernel Methods: Support Vector Learning*, pp. 243–254.

Müller, K-R, S Mika, G Rätsch, K Tsuda, and B Schölkopf (2001). "An introduction to kernel-based learning algorithms". In: *IEEE Transactions on Neural Networks* 12.2, pp. 181–201.

Nair, V and GE Hinton (2010). "Rectified linear units improve restricted boltzmann machines". In: *Proceedings of the 27th International Conference on Machine Learning*, pp. 807–814.

Nguyen, A, A Dosovitskiy, J Yosinski, T Brox, and J Clune (2016). "Synthesizing the preferred inputs for neurons in neural networks via deep generator networks". In: *Advances in Neural Information Processing Systems 29*, pp. 3387–3395.

Nickolls, J, I Buck, M Garland, and K Skadron (2008). "Scalable Parallel Programming with CUDA". In: *Queue* 6.2, pp. 40–53.

Papernot, N, P McDaniel, I Goodfellow, S Jha, ZB Celik, and A Swami (2017). "Practical black-box attacks against machine learning". In: *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pp. 506–519.

Partalas, I, A Kosmopoulos, N Baskiotis, T Artières, G Paliouras, É Gaussier, I Androutsopoulos, M Amini, and P Gallinari (2015). "LSHTC: A Benchmark for Large-Scale Text Classification". In: *arXiv preprint arXiv:1503.08581*.

Paszke, A, S Gross, S Chintala, G Chanan, E Yang, Z DeVito, Z Lin, A Desmaison, L Antiga, and A Lerer (2017). "Automatic differentiation in pytorch". In: *Neural Information Processing Systems 2017 - Workshop Autodiff*.

Pechyony, D, L Shen, and R Jones (2011). "Solving large scale linear svm with distributed block minimization". In: *Neural Information Processing Systems 2011 - Big Learning: Algorithms, Systems, and Tools for Learning at Scale workshop*.

Pedregosa, F et al. (2011a). "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12, pp. 2825–2830.

Pedregosa, F, G Varoquaux, A Gramfort, V Michel, B Thirion, O Grisel, M Blondel, P Prettenhofer, R Weiss, V Dubourg, et al. (2011b). "Scikit-learn: Machine learning in Python". In: *Journal of Machine Learning Research* 12, pp. 2825–2830.

Pennington, J, R Socher, and C Manning (2014). "Glove: Global vectors for word representation". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, pp. 1532–1543.

Platt, JC (1999). "Fast training of support vector machines using sequential minimal optimization". In: *Advances in Kernel Methods*, pp. 185–208.

Polyak, BT (1964). "Some methods of speeding up the convergence of iteration methods". In: *USSR Computational Mathematics and Mathematical Physics* 4, pp. 1–17.

Prabhu, Y and M Varma (2014). "Fastxml: A fast, accurate and stable tree-classifier for extreme multi-label learning". In: *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 263–272.

Prabhu, Y, A Kag, S Gopinath, K Dahiya, S Harsola, R Agrawal, and M Varma (2018). "Extreme Multi-label Learning with Label Features for Warm-start Tagging, Ranking & Recommendation". In: *Proceedings of the 11th ACM International Conference on Web Search and Data Mining*, pp. 441–449.

Rahimi, A and B Recht (2008). "Random Features for Large-Scale Kernel Machines". In: *Advances in Neural Information Processing Systems 20*, pp. 1177–1184.

Rahimi, A and B Recht (2009). "Weighted Sums of Random Kitchen Sinks: Replacing minimization with randomization in learning". In: *Advances in Neural Information Processing Systems 21*, pp. 1313–1320.

Reddi, SJ, S Kale, and S Kumar (2018). "On the convergence of adam and beyond". In: *International Conference on Learning Representations*.

Ribeiro, MT, S Singh, and C Guestrin (2016). ""Why Should I Trust You?": Explaining the Predictions of Any Classifier". In: *Proceedings of the 22th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1135–1144.

Rifkin, R and A Klautau (2004). "In defense of one-vs-all classification". In: *Journal of Machine Learning Research* 5, pp. 101–141.

Rijsbergen, CJV (1979). *Information Retrieval*. 2nd. Butterworth-Heinemann.

Rudi, A and L Rosasco (2017). "Generalization Properties of Learning with Random Features". In: *Advances in Neural Information Processing Systems 30*, pp. 3215–3225.

Rumelhart, DE, GE Hinton, and RJ Williams (1986). "Learning internal representations by error propagation". In: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition* 1, pp. 318–362.

Russell, SJ and P Norvig (2016). *Artificial Intelligence: A modern approach*. Pearson Education.

Sakhnini, II, MT Manry, and H Chandrasekaran (1999). "Iterative improvement of trigonometric networks". In: *2015 International Joint Conferences on Artificial Intelligence*, pp. 1275–1280.

Samek, W, A Binder, G Montavon, S Lapuschkin, and K-R Müller (2017). "Evaluating the visualization of what a Deep Neural Network has learned". In: *IEEE Transactions on Neural Networks and Learning Systems* 28.11, pp. 2660–2673.

Schaller, RR (1997). "Moore's law: past, present and future". In: *IEEE Spectrum* 34.6, pp. 52–59.

Schölkopf, B and AJ Smola (2002). *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press.

Schölkopf, B, AJ Smola, and K-R Müller (1998). "Nonlinear component analysis as a kernel eigenvalue problem". In: *Neural Computation* 10.5, pp. 1299–1319.

Schölkopf, B, S Mika, CJ Burges, P Knirsch, K-R Müller, G Rätsch, and AJ Smola (1999). "Input space versus feature space in kernel-based methods". In: *IEEE Transactions on Neural Networks* 10.5, pp. 1000–1017.

Schütt, KT, F Arbabzadah, S Chmiela, K-R Müller, and A Tkatchenko (2017a). "Quantum-chemical insights from deep tensor neural networks". In: *Nature Communications* 8, ID: 13890.

Schütt, KT, P-J Kindermans, HES Felix, S Chmiela, A Tkatchenko, and K-R Müller (2017b). "SchNet: A continuous-filter convolutional neural network for modeling quantum interactions". In: *Advances in Neural Information Processing Systems 30*, pp. 991–1001.

Selvaraju, RR, M Cogswell, A Das, R Vedantam, D Parikh, and D Batra (2017). "Grad-cam: Visual explanations from deep networks via gradient-based localization". In: *Proceedings of the 2017 International Conference on Computer Vision*, pp. 618–626.

Shokri, R and V Shmatikov (2015). "Privacy-preserving deep learning". In: *Proceedings of the 2015 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1310–1321.

Shrikumar, A, P Greenside, and A Kundaje (2017). "Learning Important Features Through Propagating Activation Differences". In: *Proceedings of the 34th International Conference on Machine Learning*, pp. 3145–3153.

Shrivastava, A and P Li (2014). "Asymmetric LSH (ALSH) for sublinear time maximum inner product search (MIPS)". In: *Advances in Neural Information Processing Systems 27*, pp. 2321–2329.

Silver, D, A Huang, CJ Maddison, A Guez, L Sifre, G van den Driessche, et al. (2016). "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529.7587, pp. 484–489.

Silver, D, J Schrittwieser, K Simonyan, I Antonoglou, A Huang, A Guez, T Hubert, L Baker, M Lai, A Bolton, et al. (2017). "Mastering the game of Go without human knowledge". In: *Nature* 550.7676, pp. 354–359.

Simonyan, K and A Zisserman (2014). "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556*.

Smilkov, D, N Thorat, B Kim, F Viégas, and M Wattenberg (2017). "Smoothgrad: Removing noise by adding noise". In: *International Conference on Machine Learning 2017 - Workshop on Visualization for Deep Learning*.

Springenberg, JT, A Dosovitskiy, T Brox, and M Riedmiller (2015). "Striving for Simplicity: The All Convolutional Net". In: *International Conference on Learning Representations - Workshop track*.

Srivastava, N, GE Hinton, A Krizhevsky, I Sutskever, and R Salakhutdinov (2014). "Dropout: a simple way to prevent neural networks from overfitting". In: *Journal of Machine Learning Research* 15, pp. 1929–1958.

Sundararajan, M, A Taly, and Q Yan (2017). "Axiomatic Attribution for Deep Networks". In: *Proceedings of the 34th International Conference on Machine Learning*, pp. 3319–3328.

Sutherland, DJ and J Schneider (2015). "On the error of random Fourier features". In: *Proceedings of the 31st Conference on Uncertainty in Artificial Intelligence*, pp. 862–871.

Sutskever, I, O Vinyals, and QV Le (2014). "Sequence to Sequence Learning with Neural Networks". In: *Advances in Neural Information Processing Systems 27*, pp. 3104–3112.

Szegedy, C, V Vanhoucke, S Ioffe, J Shlens, and Z Wojna (2016). "Rethinking the inception architecture for computer vision". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2818–2826.

Tagami, Y (2017). "AnnexML: Approximate nearest neighbor search for extreme multi-label classification". In: *Proceedings of the 23th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 455–464.

Toshniwal, A, S Taneja, A Shukla, K Ramasamy, JM Patel, S Kulkarni, J Jackson, K Gade, M Fu, J Donham, et al. (2014). "Storm at twitter". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pp. 147–156.

Tseng, P (2001). "Convergence of a block coordinate descent method for nondifferentiable minimization". In: *Journal of Optimization Theory and Applications* 109.3, pp. 475–494.

Van Den Oord, A, S Dieleman, H Zen, K Simonyan, O Vinyals, A Graves, N Kalchbrenner, A Senior, and K Kavukcuoglu (2016). "Wavenet: A generative model for raw audio". In: *arXiv preprint arXiv:1609.03499*.

Van Der Walt, S, SC Colbert, and G Varoquaux (2011). "The NumPy array: A structure for efficient numerical computation". In: *Computing in Science & Engineering* 13.2, pp. 22–30.

Vapnik, VN (1995). *The nature of statistical learning theory*. Springer.

Vapnik, VN and AJ Chervonenkis (1974). *Theory of pattern recognition*. Nauka.

Vaswani, A, N Shazeer, N Parmar, J Uszkoreit, L Jones, AN Gomez, Ł Kaiser, and I Polosukhin (2017). "Attention is all you need". In: *Advances in Neural Information Processing Systems 30*, pp. 5998–6008.

Vavilapalli, VK, AC Murthy, C Douglas, S Agarwal, M Konar, R Evans, T Graves, J Lowe, H Shah, S Seth, et al. (2013). "Apache Hadoop Yarn: Yet another resource

negotiator". In: *Proceedings of the 4th annual Symposium on Cloud Computing*, Article: 5, pp. 1–16.

Vert, J, K Tsuda, and B Schölkopf (2004). "A Primer on Kernel Methods". In: *Kernel Methods in Computational Biology*, pp. 35–70.

Vincent, P, A de Brébisson, and X Bouthillier (2015). "Efficient exact gradient update for training deep networks with very large sparse targets". In: *Advances in Neural Information Processing Systems 28*, pp. 1108–1116.

Voigt, P and A Von dem Bussche (2017). *The EU General Data Protection Regulation (GDPR)*. Springer.

Weston, J and C Watkins (1999). "Support vector machines for multi-class pattern recognition". In: *Proceedings of the Seventh European Symposium On Artificial Neural Networks*, pp. 219–224.

Williams, CK and M Seeger (2000). "Using the Nyström method to speed up kernel machines". In: *Advances in Neural Information Processing Systems 13*, pp. 661–667.

Wilson, AG, Z Hu, R Salakhutdinov, and EP Xing (2016). "Deep kernel learning". In: *20th International Conference on Artificial Intelligence and Statistics*, pp. 370–378.

Yang, T, Y-F Li, M Mahdavi, R Jin, and Z-H Zhou (2012). "Nyström method vs random fourier features: A theoretical and empirical comparison". In: *Advances in Neural Information Processing Systems 25*, pp. 476–484.

Yang, Z, A Wilson, AJ Smola, and L Song (2015a). "A la Carte – Learning Fast Kernels". In: *Journal of Machine Learning Research* 38, pp. 1098–1106.

Yang, Z, M Moczulski, M Denil, N de Freitas, AJ Smola, L Song, and Z Wang (2015b). "Deep Fried Convnets". In: *Proceedings of the 2015 International Conference on Computer Vision*, pp. 1476–1483.

Yen, IE-H, X Huang, P Ravikumar, K Zhong, and I Dhillon (2016). "PD-Sparse: A primal and dual sparse approach to extreme multiclass and multilabel classification". In: *Proceedings of the 33th International Conference on Machine Learning*, pp. 3069–3077.

Yen, IE-H, X Huang, W Dai, P Ravikumar, I Dhillon, and E Xing (2017). "PPDsparse: A Parallel Primal-Dual Sparse Method for Extreme Classification". In: *Proceedings of the 23th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 545–553.

Yu, FX, S Kumar, H Rowley, and S-F Chang (2015). "Compact nonlinear maps and circulant extensions". In: *arXiv preprint arXiv:1503.03893*.

Yu, FX, AT Suresh, KM Choromanski, DN Holtmann-Rice, and S Kumar (2016). "Orthogonal random features". In: *Advances in Neural Information Processing Systems 29*, pp. 1975–1983.

Yu, H-F, P Jain, P Kar, and I Dhillon (2014). "Large-scale multi-label learning with missing labels". In: *Proceedings of the 31th International Conference on Machine Learning*, pp. 593–601.

Zaharia, M, RS Xin, P Wendell, T Das, M Armbrust, A Dave, X Meng, J Rosen, S Venkataraman, MJ Franklin, et al. (2016). "Apache spark: A unified engine for big data processing". In: *Communications of the ACM* 59.11, pp. 56–65.

Zeiler, MD and R Fergus (2014). "Visualizing and understanding convolutional networks". In: *Proceedings of the 2014 European Conference on Computer Vision*, pp. 818–833.

Zhang, J, SA Bargal, Z Lin, J Brandt, X Shen, and S Sclaroff (2018). "Top-down neural attention by excitation backprop". In: *International Journal of Computer Vision* 126.10, pp. 1084–1102.

Zhou, D, L Xiao, and M Wu (2011). "Hierarchical classification via orthogonal transfer". In: *Proceedings of the 28th International Conference on Machine Learning*, pp. 801–808.

Zhu, K, H Wang, H Bai, J Li, Z Qiu, H Cui, and EY Chang (2008). "Parallelizing Support Vector Machines on Distributed Computers". In: *Advances in Neural Information Processing Systems 20*, pp. 257–264.

Zintgraf, LM, TS Cohen, T Adel, and M Welling (2017). "Visualizing deep neural network decisions: Prediction difference analysis". In: *International Conference on Learning Representations*.

Zoph, B, V Vasudevan, J Shlens, and QV Le (2018). "Learning Transferable Architectures for Scalable Image Recognition". In: *2018 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 8697–8710.

Zuo, W, Y Zhu, and L Cai (2009). "Fourier-neural-network-based learning control for a class of nonlinear systems with flexible components". In: *IEEE Transactions on Neural Networks* 20.1, pp. 139–151.

List of figures

List of tables

Chapter A

Appendix

1 Efficient learning of kernel approximations

1.1 Additional empirical evidence

In Table A.1 and Figure A.1 we show the results of our main experiment for three additional data sets. Please find the analysis in the main text.

	Gaussian				*ArcCos*			
Dataset	**RB**	**UAB**	**SAB**	**DAB**	**RB**	**UAB**	**SAB**	**DAB**
Adult	85.1	85.0	84.8	85.1	85.0	85.1	84.9	85.0
Letter	96.1	96.1	97.1	97.6	95.3	95.3	90.0	98.7
USPS	95.1	95.0	94.5	95.3	94.3	94.4	92.0	95.1

Tab. A.1: Classification performance. *Best accuracy in % for different bases.*

1.2 Deep kernel machines

Figure A.2 and Figure A.3 depict in more detail how the kernels ArcCos2 and ArcCos3 perform on the MNIST and the CoverType data set. Please find the analysis in the main text.

1.3 Optimization

Overall, the training time of the different bases relate as follows. With respect to the classification performance SAB and DAB are considerably faster than RB and UAB. This holds mainly because one only needs to train a much smaller basis while reaching the same performance. With regard to the number of features all methods expose a linear increase in training time. This is caused due the chosen learning procedure. Given the same number of features, methods without kernel adaption, i.e., RB and DAB, are up to a magnitude faster than the others. Further, training using a RB can be up to a magnitude faster than DAB. Note that we did not tune nor implement the kernel adaption to be fast, but to give high accuracy.

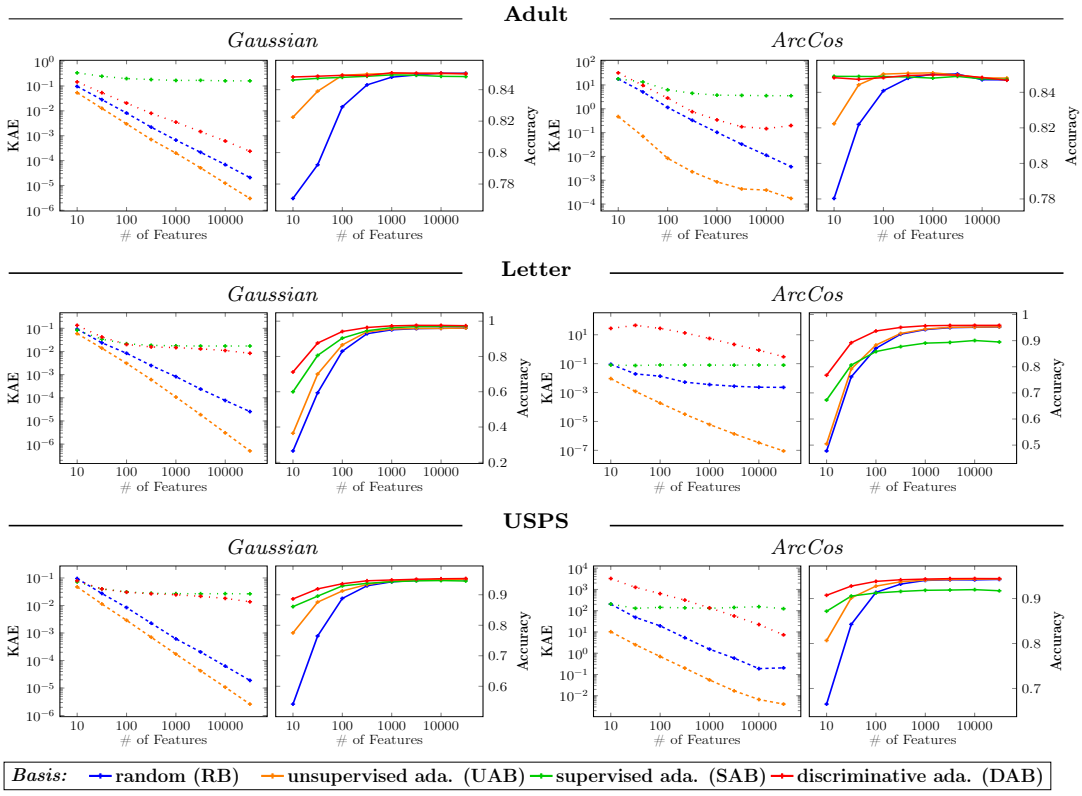

Fig. A.1: Adapting bases. *The plots show the relationship between the number of features (x-Axis), the KAE in logarithmic spacing (**left, dashed lines**) and the classification error (**right, solid lines**). Typically, the KAE decreases with a higher number of features, while the accuracy increases. The KAE for SAB and DAB (orange and red dotted line) hints how much the adaptation deviates from its initialization (blue dashed line). Best viewed in digital and color.*

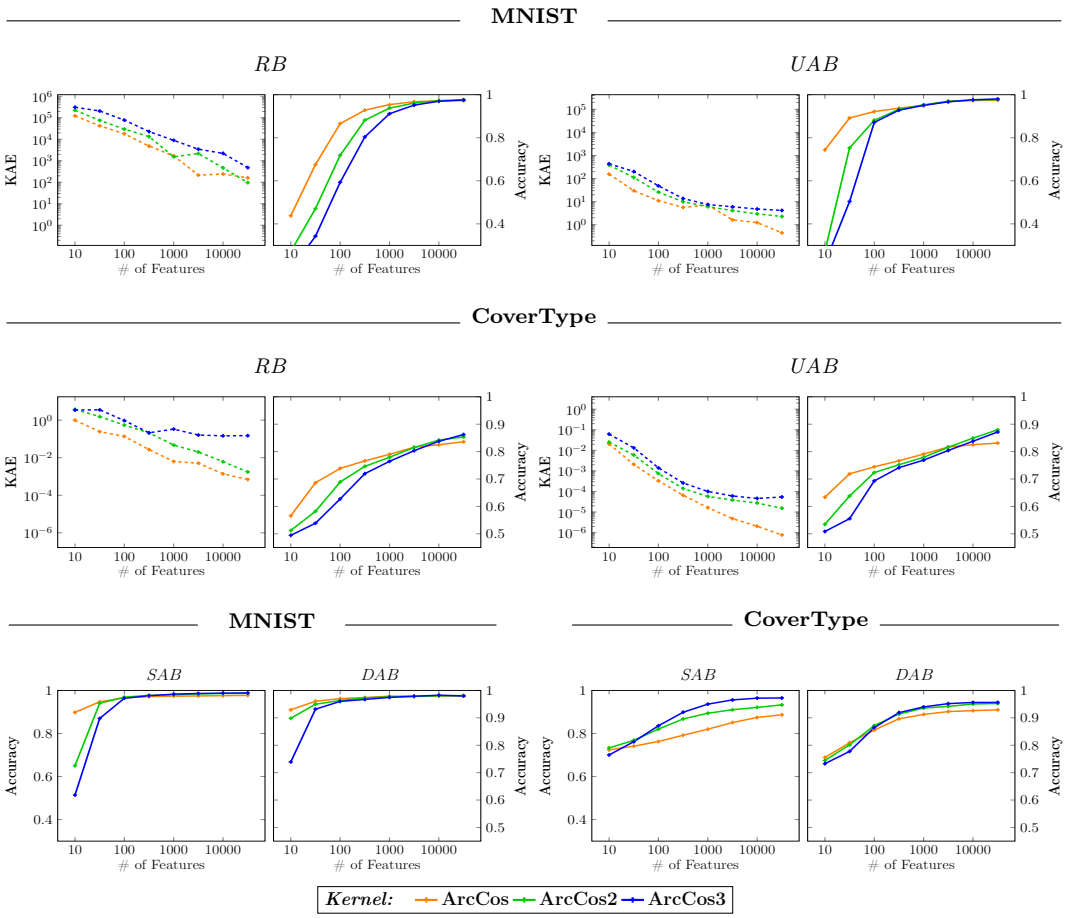

Fig. A.2: Deep kernel machines. *The performance of the ArcCos-kernels with 1-, 2-, and 3-layer models. The KAE is given in dashed lines and the accuracy in solid lines. Best viewed in digital and color.*

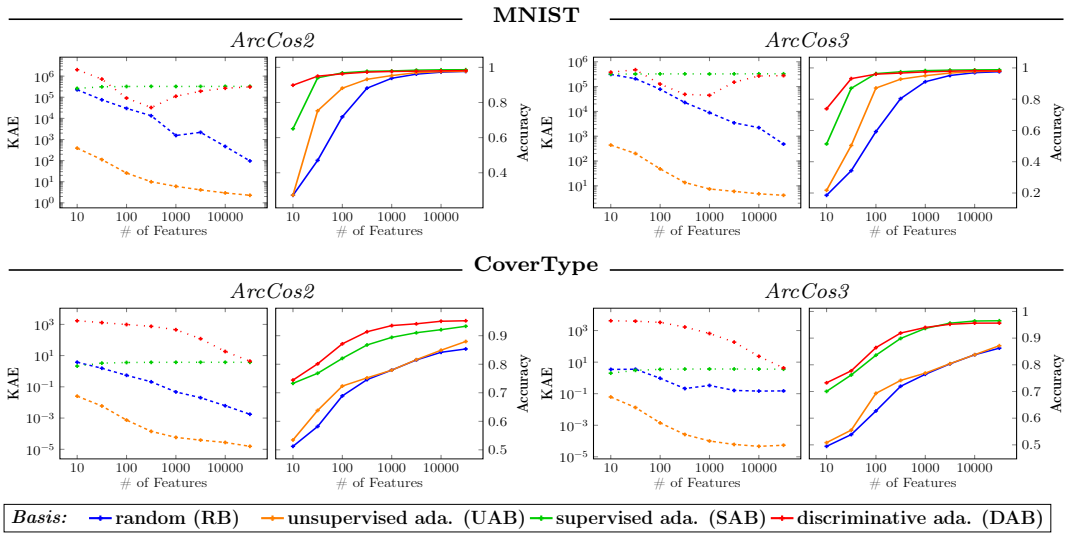

Fig. A.3: Deep kernel machines. *The plots show the relationship between the number of features (x-Axis), the KAE in logarithmic spacing (**left, dashed lines**) and the classification error (**right, solid lines**). Typically, the KAE decreases with a higher number of features, while the accuracy increases The KAE for SAB and DAB (orange and red dotted line) hints how much the adaptation deviates from its initialization (blue dashed line). Best viewed in digital and color.*

2 Efficient software for prediction analysis

This section contains additional content for Chapter V and the code snippets build up on the already presented ones.

2.1 Prediction- and gradient-based algorithms

Algorithms that only rely on function or on gradient evaluations can be of very simple, yet effective nature (Kindermans et al., 2016; Shrikumar et al., 2017; Smilkov et al., 2017; Sundararajan et al., 2017; Zintgraf et al., 2017; Ribeiro et al., 2016; Lundberg and Lee, 2017). A downside can be the their runtime, which is often a multiple of a single function call.

Input * gradient As a first example we consider input * gradient (Kindermans et al., 2016; Shrikumar et al., 2017). The name already says it: the algorithm consists of an element-wise multiplication of the input times the gradient. The corresponding formula is:

$$e(x) = x \odot \nabla_x f(x). \tag{A.1}$$

The method can be implemented as follows and the result is marked as A1 in Figure V.1:

```
# Take gradient of output neuron w.r.t. to the input
gradient = tf.gradients(max_output, input)[0]
# and multiply it with the input
input_t_gradient = input * gradient
# Run the code with TF
A1 = sess.run(input_t_gradient, {input: x})
```

Integrated Gradients A more evolved example is the method Integrated Gradients (Sundararajan et al., 2017) which tries to capture the effect of non-linearities better by computing the gradient along a line between an input image and a given reference image x'. The corresponding formula for i-th input dimension is:

$$e(x_i) = (x_i - x_i') \odot \int_{\alpha=0}^{1} \frac{\delta f(x)}{\delta x_i}\Big|_{x=x'+\alpha(x-x')} d\alpha. \tag{A.2}$$

To implement the method the integral is approximated with a finite sum and, building on the previous code snippet, the code looks as follows (result is tagged with A2 in Figure V.1):

```
# Nr. of steps along path
steps = 32
# Take as reference a black image,
# i.e., lowest number of the networks input value range.
x_ref = np.ones_like(x) * net['input_range'][0]
# Take gradient of output neuron w.r.t. to input
gradient = tf.gradients(max_output, input)[0]

# Sum gradients along the path from x to x_ref
gradient_sum = np.zeros_like(x)
for step in range(steps):
  # Create intermediate input
  x_step = x_ref + (x - x_ref) * step / steps
  # Compute and add the gradient for intermediate input
  gradient_sum += sess.run(gradient, {input: x_step})

# Integrated Gradients formula
A2 = gradient_sum * (x - x_ref)
```

Occlusion In contrast to the two presented methods occlusion-based methods rely on the function value instead of its gradient, e.g., Zeiler and Fergus (2014) and Zintgraf et al. (2017). The basic variant (Zeiler and Fergus, 2014) divides the input, typically an image, into a grid of non-overlapping patches. Then each patch gets the function value assigned that is obtained when the patch region in the original image is perturbed or replaced by a reference value. Eventually, all values are normalized with the default activation given when no patch is occluded. The algorithm can be implemented as follows and the result is denoted as A3 in Figure V.1:

```
diff = np.zeros_like(x)
# Choose a patch size
psize = 8

# Occlude patch by patch and calculate activation for each patch
for i in range(0, net['image_shape'][0], psize):
  for j in range(0, net['image_shape'][0], psize):

    # Create image with the patch occluded
    occluded_x = x.copy()
    occluded_x[:, i:i+psize, j:j+psize, :] = 0

    # Store activation of occluded image
    diff[:, i:i+psize, j:j+psize, :] = sess.run(
          max_output, {input: occluded_x})[0]

# Normalize with initial activation value
A3 = sess.run(max_output, {input: x})[0] - diff
```

LIME The last prediction-based explanation class, e.g., Ribeiro et al. (2016) and Lundberg and Lee (2017), decomposes the data into features. Subsequently, prediction results for inputs — composed of perturbed features — are collected, yet instead of

using the values directly for the explanation, they are used to learn an importance value for the respective features.

One representative algorithm is "Local interpretable model-agnostic explanations" (Ribeiro et al., 2016, LIME) that learns a local regressor for each explanation. It works as follows for images. First the image is divided into segments, e.g., continuous color regions. Then a dataset is sampled where the features are a randomly perturbed, e.g., filled with gray color. The target of the sample is determined by the prediction value for the accordingly altered input. Using this dataset a weighted, regression model is learned and the resulting weight vector's values indicate the importance of each segment in the neural network's initial prediction. The algorithm can be implemented as follows and the result is denoted as A4 in Figure V.1:

```
# Segment (not pre-processed) image
segments = skimage.segmentation.quickshift(
  x_not_pp[0], kernel_size=4, max_dist=200, ratio=0.2)
nr_segments = np.max(segments) + 1


# Create dataset
nr_samples = 1000
# Randomly switch segments on and off
features = np.random.randint(0, 2, size=(nr_samples, nr_segments))
# Make sure original image is present
features[0, :] = 1

# Get labels for features
labels = []
for sample in features:
  tmp = x.copy()
  # Switch segments on and off
  for segment_id, segment_on in enumerate(sample):
    if segment_on == 0:
      tmp[0][segments == segment_id] = (0, 0, 0)
  # Get predicted value for this sample
  labels.append(sess.run(max_output, {input: tmp})[0])


# Compute sample weights
distances = sklearn.metrics.pairwise_distances(
            features,
            features[0].reshape(1, -1),
            metric='cosine',
).ravel()
kernel_width = 0.25
sample_weights = np.sqrt(np.exp(-(distances ** 2) / kernel_width ** 2))

# Fit L1-regressor
regressor = sklearn.linear_model.Ridge(alpha=1, fit_intercept=True)
regressor.fit(features, labels, sample_weight=sample_weights)
weights = regressor.coef_


# Map weights onto segments
A4 = np.zeros_like(x)
for segment_id, w in enumerate(weights):
  A4[0][segments == segment_id] = (w, w, w)
```

As initially mentioned a drawback of prediction- and gradient-based methods can be slow runtime, which is often a multiple of a single function evaluation — as the loops in the code snippets already suggested. For instance Integrated Gradients used 32 evaluations, the occlusion algorithm $(224/4)^2 = 56^2 = 3136$ and LIME 1000 (same as in Ribeiro et al. (2016)). Especially for complex networks and for applications with time constraints this can be prohibitive.

2.2 PatternNet

The exemplary implementation for PatterNet discussed in Section 3.2:

```
# Extending iNNvestigate base class with the PatternNet algorithm
class PatternNet(ReverseAnalyzerBase):

  # Storing the patterns.
  def __init__(self, model, patterns, **kwargs):
    self._patterns = patterns[:]
    super(PatternNet, self).__init__(model, **kwargs)

  def _get_pattern_for_layer(self, layer):
    return self._patterns.pop(-1)

  def _patternnet_mapping(self, X, Y, bp_Y, bp_state):
    # Get layer,
    layer = bp_state['layer']
    # exchange kernel weights with patterns,
    weights = layer.get_weights()
    weights[0] = self._get_pattern_for_layer(layer)
    # and create layer copy without activation part and patterns as filters
    layer_wo_act = kgraph.copy_layer_wo_activation(layer, weights=weights)

    if kchecks.contains_activation(layer, 'relu'):
      # Gradient of activation layer
      tmp = tf.where(Y > 0, bp_Y, tf.zeros_like(bp_Y))
    else:
      # Gradient of linear layer
      tmp = bp_Y

    # map back along layer with patterns instead of weights
    pattern_Y = layer_wo_act(X)
    return tf.gradients(pattern_Y, X, grad_ys=tmp)[0]

  # Register the mappings
  def _create_analysis(self, *args, **kwargs):
    self._add_conditional_reverse_mapping(
      # Apply to all layers that contain a kernel
      lambda layer: kchecks.contains_kernel(layer),
      tf_to_keras_mapping(self._patternnet_mapping),
      name='pattern_mapping',
  )
  return super(PatternNet, self)._create_analysis(*args, **kwargs)

analyzer = PatternNet(model_wo_sm, net['patterns'])
B4 = analyzer.analyze(x)
```

2.3 Hyper-parameter selection

The code snippet for the hyper-parameter selection for Integrated Gradients:

```
IG = []
# Take 5 samples from network's input value range
for ri in np.linspace(net['input_range'][0], net['input_range'][1], num=5):
  # and analyze with each.
  analyzer = innvestigate.create_analyzer(
    'integrated_gradients',
    model_wo_sm,
    reference_inputs=ri,
    steps=32
  )
  IG.append(analyzer.analyze(x))
```

The code snippet for the hyper-parameter selection for SmoothGrad:

```
SG1, SG2 = [], []
# Take 5 scale samples for the noise scale of smoothgrad.
for scale in range(5):
  noise_scale = (net['input_range'][1]-net['input_range'][0]) * scale / 5
  # Smoothgrad with absolute gradients
  analyzer = innvestigate.create_analyzer(
    'smoothgrad',
    model_wo_sm,
    augment_by_n=32,
    noise_scale=noise_scale,
    postprocess='abs'
  )
  SG1.append(analyzer.analyze(x))

  # Smoothgrad with with squared gradients
  analyzer = innvestigate.create_analyzer(
    'smoothgrad',
    model_wo_sm,
    augment_by_n=32,
    noise_scale=noise_scale,
    postprocess='square'
  )
  SG2.append(analyzer.analyze(x))
```

2.4 Visualization

The exemplary implementation of visualization approaches discussed in Section 3.4.3:

```
def explanation_to_heatmap(e):
  # Reduce color axis
  tmp = np.sum(e, axis=color_channel_axis)
  # To range [0, 255]
  tmp = (tmp / np.max(np.abs(tmp))) * 127.5 + 127.5

  # Create and apply red-blue heatmap
  colormap = matplotlib.cm.get_cmap("seismic")
  tmp = colormap(tmp.flatten().astype(np.int64))[:, :3]
  tmp = tmp.reshape(e.shape)
  return tmp

def explanation_to_graymap(e):
  # Reduce color axis
  tmp = np.sum(np.abs(e), axis=color_channel_axis)
  # To range [0, 255]
  tmp = (tmp / np.max(np.abs(tmp))) * 255

  # Create and apply red-blue heatmap
  colormap = matplotlib.cm.get_cmap("gray")
  tmp = colormap(tmp.flatten().astype(np.int64))[:, :3]
  tmp = tmp.reshape(e.shape)
  return tmp

def explanation_to_scale_input(e):
  # Create scale
  e = np.sum(np.abs(e), axis=color_channel_axis, keepdims=True)
  scale = e / np.max(e)

  # Apply to image
  return (x_not_preprocessed / 255) * scale

def explanation_to_scale_input(e):
  # Get highest scored segments
  # Segments are reused from the LIME example.
  segments_scored = [(np.max(e[0][segments == sid]), sid)
    for sid in range(nr_segments)]
  highest_ones = sorted(segments_scored, reverse=True)[:50]

  # Compute mask
  mask = np.zeros_like(segments)
  for _, sid in highest_ones:
    mask[segments == sid] = 1

  # Apply mask
  ret = (x_not_pp.copy() / 255)
  ret[0][mask == 0] = 0
  return ret

def explanation_to_blend_w_input(e):
  e = np.sum(np.abs(e), axis=channel_axis, keepdims=True)
  # Add blur
  e = skimage.filters.gaussian(x[e], 3)[None]
  # Normalize
  e = (e - e.min())/(e.max()-e.min())
  # Get and apply colormap
  heatmap = plot.get_cmap('jet')(e[:, :,:,0])[:,:,:,:3]
  # Overlap
  ret = (1.0-e) * (x_not_pp / 255) + e * heatmap
```

```
60    return ret
61
62  def explanation_to_projection(e):
63    # To range [0, 1]
64    return (e / np.max(np.abs(e))) + 0.5
```

2.5 LRP proposition for batch normalization layers

Following the notation in Section 1.3.2 on LRP in Chapter II the rule proposed by Hui and Binder (2018) for batch normalization layers is defined as follows:

$$R_i = (\alpha_0 \frac{x_i^+ w_i^+}{x_i^+ w_i^+ + b_i^+} + \alpha_1 \frac{x_i^+ w_i^-}{x_i^+ w_i^- + b_i^-} + \alpha_2 \frac{x_i^- w_i^+}{x_i^- w_i^+ + b_i^+} + \alpha_3 \frac{x_i^- w_i^-}{x_i^- w_i^- + b_i^-}) R_j \tag{A.3}$$

where $w_i = \gamma_i / \sqrt{\sigma_i^2 + \epsilon}$ and $b_i = (-\gamma_i \mu_i / \sqrt{\sigma_i^2 + \epsilon}) + \beta_i$ with μ, σ, γ, β the parameters of the batch normalization layer (Ioffe and Szegedy, 2015) and $\alpha = (2, -1, 0, 0)$.