Ensemble Learning with Discrete Classifiers on Small Devices

Dissertation

zur Erlangung des Grades eines

Doktors der Naturwissenschaften

der Technischen Universität Dortmund
an der Fakultät für Informatik

von

Sebastian Buschjäger

Dortmund

2022

Tag der mündlichen Prüfung: 10.10.2022
Dekan: Prof. Dr.-Ing. Gernot A. Fink
GutachterIn: Prof. Dr. Katharina Morik
Prof. Dr. Johannes Fürnkranz

Abstract

Machine learning has become an integral part of everyday life ranging from applications in AI-powered search queries to (partial) autonomous driving. Many of the advances in machine learning and its application have been possible due to increases in computation power, i.e., by reducing manufacturing sizes while maintaining or even increasing energy consumption. However, 2-3 nm manufacturing is within reach, making further miniaturization increasingly difficult while thermal design power limits are simultaneously reached, rendering entire parts of the chip useless for certain computational loads. In this thesis, we investigate discrete classifier ensembles as a resource-efficient alternative that can be deployed to small devices that only require small amounts of energy. Discrete classifiers are classifiers that can be applied – and oftentimes also trained – without the need for costly floating-point operations. Hence, they are ideally suited for deployment to small devices with limited resources. The disadvantage of discrete classifiers is that their predictive performance often lacks behind their floating-point siblings. Here, the combination of multiple discrete classifiers into an ensemble can help to improve the predictive performance while still having a manageable resource consumption. This thesis studies discrete classifier ensembles from a theoretical point of view, an algorithmic point of view, and a practical point of view. In the theoretical investigation, the bias-variance decomposition and the double-descent phenomenon are examined. The bias-variance decomposition of the mean-squared error is re-visited and generalized to an arbitrary twice-differentiable loss function, which serves as a guiding tool throughout the thesis. Similarly, the double-descent phenomenon is – for the first time – studied comprehensively in the context of tree ensembles and specifically random forests. Contrary to established literature, the experiments in this thesis indicate that there is no double-descent in random forests. While the training of ensembles is well-studied in literature, the deployment to small devices is often neglected. Additionally, the training of ensembles on small devices has not been considered much so far. Hence, the algorithmic part of this thesis focuses on the deployment of discrete classifiers and the training of ensembles on small devices. First, a novel combination of ensemble pruning (i.e., removing classifiers from the ensemble) and ensemble refinement (i.e., re-training of classifiers in the ensemble) is presented, which uses a novel proximal gradient descent algorithm to minimize a combined loss function. The resulting algorithm removes unnecessary classifiers from an already trained ensemble while improving the performance of the remaining classifiers at the same time. Second, this algorithm is extended to the more challenging setting of online learning in which the algorithm receives training examples one by one. The resulting shrub ensembles algorithm allows the training of ensembles in an online fashion while maintaining a strictly bounded memory consumption. It outperforms existing state-of-the-art algorithms under resource constraints and offers competitive performance in the general case. Last, this thesis studies the deployment of decision tree ensembles to small devices by optimizing their memory layout. The key insight here is that decision trees have a probabilistic inference time because different observations can take different paths from the root to a leaf. By estimating the probability of visiting a particular node in the tree, one can place it favorably in the memory to maximize the caching behavior and, thus, increase its performance without changing the model. Last, several real-world applications of tree ensembles and Binarized Neural Networks are presented.

Acknowledgements

This thesis would have never existed if it weren't for all the people that I shared this journey with – let it be in good or in bad times, and this is something I am extremely grateful for.
I was fortunate enough to – somewhat by accident – stumble upon the Artificial Intelligence Group in Dortmund, which affected my journey more than I could have predicted in the beginning. My supervisor Katharina Morik never failed to challenge me while providing an environment in which failure is a learning experience and not the end of the world. She constantly pushed me to new endeavors while giving me the freedom to pursue them at my own pacing and with my own thoughts. While I started out seeking advice on research papers, equations, and experiments I soon realized that I can ask her advice for any situation in life beyond research which I am grateful for.
I also want to thank my second supervisor, Johannes Fürnkranz, who made the final steps of writing, defending and publishing my dissertation go as smoothly as possible. Yet, at the same time, he provided one of the most thorough and insightful reviews I ever received for one of my works. I also thank Jian-Jia Chen, with whom I was fortunate enough to be on the same research project SFB876-A1. Jian-Jia not only shared insight and his expertise whenever he had to offer it, but he also has the ability to quickly understand and follow the thoughts of other people while giving them enough room to share their ideas.
Arguably, the largest impact on my journey was due to my colleagues at the Artificial Intelligence Group, that never failed to support me in any way necessary – let it be by helping me with hardware, software or by simply having someone to talk to over a coffee or beer. To not turn these acknowledgments into a long list of names, I will not name everybody from LS 8, although they all deserve praise. From this long list of people, special Thanks go to Lukas Pfahler. Lukas and I started to work at the Chair around the same time, and we (or at least I) had the pleasure of sharing the office. I believe this thesis would have looked very different without his support. The number of times he pointed out errors in my code, in my math or in my writing are countably infinite. At the same time, he never failed to ask for a coffee or lunch break at the right time.
Finally, words cannot express the gratitude I feel for the support from my friends and my family, and hence I will keep it short here. My mother always supported me by pushing me to see beyond numbers and equations and to think about the people in the real world. My father always supported me with an open ear that would listen to my problems at any time of day or night without judging me. My friends offered me support when reviewer 2 would reject a paper and offered me drinks whenever reviewer 1 convinced reviewer 2 to accept the paper. I am happy that I can call Basti, Janis, and Tobias my dearest friends. My cat Nemo never failed to crash a Zoom meeting that took too long or to discuss a research paper with me, although his main argument always seems to be that he needs more food. Last, I fell deeply in love with Katja throughout this journey. So, before these acknowledgments really turn into a list of names, I just want to say: Thank you all for being there for me.

Contents

Abstract iii

Acknowledgements v

I Fundamentals 1

1 Introduction 3
- 1.1 Contributions and Outline . . . 6
- 1.2 Publications Covered by this Thesis . . . 8
- 1.3 Publications Not Covered by this Thesis . . . 10

2 Background 13
- 2.1 Small Devices . . . 13
  - 2.1.1 Von Neumann architecture . . . 13
  - 2.1.2 Beyond the von Neumann Architecture . . . 18
- 2.2 Mathematical Optimization . . . 21
- 2.3 Machine Learning . . . 25
- 2.4 Discrete Classifiers . . . 29
  - 2.4.1 Decision Trees . . . 30
  - 2.4.2 Binarized Neural Networks . . . 41
  - 2.4.3 Naive Bayes . . . 51

II Additive Ensembles 57

3 The Bias-Variance Decomposition for Additive Ensembles 59

4 Training Additive Ensembles 67
- 4.1 Generalized Negative Correlation Learning . . . 67
- 4.2 Bagging and Related Algorithms . . . 72
- 4.3 Boosting and Related Algorithms . . . 74
- 4.4 Dropout and Pseudo-Ensembles . . . 78
- 4.5 Stochastic Multiple Choice Learning . . . 79

5 On the Double-Descent Phenomenon in Random Forests 81
- 5.1 The Complexity of Tree Ensembles . . . 82
- 5.2 There is no Double-Descent in RF . . . 84
- 5.3 The Complexity of RF is bounded by the Data . . . 87
- 5.4 Complexity does not predict a RF’s Error . . . 87
- 5.5 Negative Correlation Forests . . . 91

III Additive Ensembles and Small Devices 95

6 Training Ensembles for Small Devices 97
- 6.1 Ensemble Pruning . . . 98
- 6.2 Leaf-Refinement . . . 101
- 6.3 Combining Leaf-Refinement and Ensemble Pruning . . . 102
- 6.4 Experiments . . . 104
  - 6.4.1 (Q1) What Method has the Best Predictive Performance? . . . 105
  - 6.4.2 (Q2) What Method has the Best Predictive Performance under Memory Constraints? . . . 109

7 Training Ensembles on Small Devices 115
- 7.1 Online Learning of Tree Ensembles . . . 115
  - 7.1.1 Online DT Learning . . . 116
  - 7.1.2 Online BNNs . . . 120
  - 7.1.3 Online Naive Bayes . . . 120
  - 7.1.4 Online Ensembling . . . 120
- 7.2 Shrub Ensembles . . . 121
  - 7.2.1 Optimizing the Weights when $\mathcal{H}$ is known . . . 122
  - 7.2.2 Optimizing $\mathcal{H}$ simultaneously with the Weights . . . 123
  - 7.2.3 Theoretical Performance of Shrub Ensembles . . . 124
  - 7.2.4 Runtime and Memory Consumption of Shrub Ensembles . . . 127
- 7.3 Experiments . . . 127
  - 7.3.1 Quantitative Analysis . . . 129
  - 7.3.2 Qualitative Analysis . . . 132

8 Implementing Ensembles on Small Devices 135
- 8.1 Implementing Decision Trees . . . 135
- 8.2 A probabilistic View of DT Execution . . . 138
- 8.3 Optimizing the Memory Layout of Trees . . . 139
  - 8.3.1 Optimization of If-Else Trees . . . 140
  - 8.3.2 Optimization of Native Tree . . . 143
  - 8.3.3 Experiments . . . 145
- 8.4 A Theoretical Execution Model of DT ensembles . . . 151
  - 8.4.1 A Theoretical von Neumann Architecture . . . 151
  - 8.4.2 A Theoretical Model of FPGAs . . . 152
  - 8.4.3 Implementing DT ensembles on von Neumann Architectures . . . 154
  - 8.4.4 Implementing DT Ensembles on FPGAs . . . 159

IV Discrete Classifiers and Small Devices in Practice 165

9 Software and Libraries 167
- 9.1 Overview of existing Software . . . 167
- 9.2 PyPruning . . . 168
- 9.3 Fastinference . . . 170

10 Case Studies with Discrete Classifiers 179
- 10.1 Discrete Classifiers and the PhyNetLab . . . 179
- 10.2 Discrete Classifiers and the FACT Telescope . . . 181
  - 10.2.1 The FACT Telescope . . . 181
  - 10.2.2 Optimal Implementations of Random Forests . . . 182

10.2.3 On-Site Gamma-Hadron Separation with BNNs 185
10.3 Discrete Classifiers and Approximate Memory 191
10.3.1 Margin-Maximization for Error Tolerant BNNs 193
10.3.2 Experiments 194

11 Conclusion and Future Work 197

A Bibliographical Remarks 201

B Additional Results for the Experiments in Chapter 6 203
B.1 Accuracy under Memory Constraints 203
B.2 F_1 score under Memory Constraints 206

C Additional Results for the Experiments in Chapter 7 209

List of Figures

1.1 The power consumption and execution time given different processing power 4

2.1 Sketch of the von Neumann architecture with its pipeline 15
2.2 Sketch of a memory hierarchy. 16
2.3 Sketch of a hardware accelerator. 19
2.4 Sketch of a GP-GPU. 20
2.5 Sketch of an FPGA. 21
2.6 Hoeffdings Bound. 28
2.7 An example of an axis-aligned binary decision tree. 33
2.8 Comparison between different split scores. 35
2.9 Example architecture for BNNs based on the VGG architecture [SZ15]. 49

5.1 The Bias-Variance trade-off. 82
5.2 Test error, training error, and complexity of RF and DT. 86
5.3 Test error, training error and complexity of DA-RF and DA-DT. 90
5.4 Mean-squared error and test error of NCForest. 93

6.1 Critical Difference Diagram for the accuracy and F_1 score of ensemble pruning and leaf-refinement. 109
6.2 Average number of estimators across all datasets and configuration of L1+LR for different λ values. 112
6.3 Two-dimensional critical difference diagram for the accuracy of pruning and leaf-refinement under memory constraint 113
6.4 Two-dimensional critical difference diagram for the F_1 score of pruning and leaf-refinement under memory constraint 113

7.1 Example of a hyperparameter configuration for the experiments for Shrub Ensembles in Section 7 128
7.2 Critical Difference Diagram for Shrub Ensembles. 132
7.3 Accuracy and memory consumption of online learning algorithms over time on the gas-sensor dataset. 133
7.4 Accuracy and memory consumption of online learning algorithms over time on the led_a dataset. 134

8.1 Sketch of the probabilistic execution of decision trees. 138
8.2 Example implementation of a decision tree using the native implementation in `C++`. 139
8.3 Example implementation of a decision tree using the if-else implementation in `C++`. 140

8.4 Example implementation of a decision tree using the goto implementation in C++. . . . 142
8.5 Average speed-up of tree framing on Intel. . . . 147
8.6 Average speed-up of tree framing on PPC. . . . 148
8.7 Average speed-up of tree framing on ARM. . . . 149
8.8 Implementation of single decision tree node for the native implementation in the theoretical von Neumann architecture. . . . 155
8.9 Implementation of a decision tree using the native approach in the theoretical von Neumann architecture. . . . 155
8.10 Implementation of a decision tree using the if-else approach in the theoretical von Neumann architecture. . . . 157
8.11 Implementation of single decision tree node for the SIMD implementation in the theoretical von Neumann architecture. . . . 157
8.12 Implementation of a decision tree using the SIMD approach in the theoretical von Neumann architecture. . . . 158
8.13 Implementation of a majority vote in the theoretical von Neumann architecture. . . . 159
8.14 Implementation of a decision tree using the DNF approach in the theoretical FPGA. . . . 162

9.1 Example of a custom metric for pruning in PyPruning. . . . 169
9.2 Example of a custom pruner in PyPruning. . . . 170
9.3 Workflow of FastInference. . . . 174
9.4 Base class for every model in FastInference. . . . 175
9.5 The swap optimization implemented in FastInference. . . . 176
9.6 Template code for the if-else implementation used in FastInference. . . . 177
9.7 Realization of the if-else implementation in FastInference. . . . 178

10.1 Sketch of an air shower hitting the FACT telescope. . . . 182
10.2 Processing pipeline of FACT extracting high-level features. . . . 183
10.3 Processing pipeline of FACT extracting low-level features and applying a model learned from raw observations. . . . 186
10.4 Histogram of the frequencies as a function of the squared angular distance between the trajectory of any incoming ray and a position in the sky. . . . 190
10.5 Test accuracy for models trained via bit-flip injection and MHL under different bit error rates. . . . 195
10.6 Test accuracy for models trained via a combination of bit-flip injection and MHL under different bit error rates. . . . 195

C.1 Test-then-train accuracy, memory consumption and pareto front of each method on the agrawal_a dataset 209
C.2 Test-then-train accuracy, memory consumption and pareto front of each method on the agrawal_g dataset 209
C.3 Test-then-train accuracy, memory consumption and pareto front of each method on the airlines dataset 210
C.4 Test-then-train accuracy, memory consumption and pareto front of each method on the covtype dataset 210
C.5 Test-then-train accuracy, memory consumption and pareto front of each method on the elec dataset 210

C.6 Test-then-train accuracy, memory consumption and pareto front of each method on the gas-sensor dataset 211
C.7 Test-then-train accuracy, memory consumption and pareto front of each method on the led_a dataset 211
C.8 Test-then-train accuracy, memory consumption and pareto front of each method on the led_g dataset 211
C.9 Test-then-train accuracy, memory consumption and pareto front of each method on the nomao dataset 212
C.10 Test-then-train accuracy, memory consumption and pareto front of each method on the rbf_f dataset 212
C.11 Test-then-train accuracy, memory consumption and pareto front of each method on the rbf_m dataset 212
C.12 Test-then-train accuracy, memory consumption and pareto front of each method on the weather dataset 213

List of Tables

1.1 Energy consumption of instructions and memory access based on word size in CMOS 45nm structures [Hor14]. For comparison, the relative costs depict the costs relative to a single 32-bit integer addition. . . . 6

2.1 Typical microcontroller units (MCUs) found in edge and IoT devices. . 14
2.2 Illustration of a dataset as a table with corresponding label vector. . . 26

5.1 Datasets used for the experiments on double-descent in Section 5. . . . 82

6.1 Hyperparameters of each method used in the experiments on pruning and leaf-refinement in section 6. 106
6.2 Datasets used for the experiments on pruning and leaf-refinement in Section 6. 106
6.3 Accuracy of pruning methods and leaf-refinement without memory constraints. 107
6.4 The F_1 score of pruning methods and leaf-refinement without memory constraints. 108
6.5 Accuracy of pruning methods and leaf-refinement for models under 128 KB memory. 110
6.6 Accuracy of pruning methods and leaf-refinement for models under 512 KB memory. 110
6.7 Accuracy of pruning methods and leaf-refinement for models under 2048 KB memory. 111

7.1 Datasets used for experiments on Shrub Ensembles in Section 7 128
7.2 Best test-then-train accuracy for each method on the given datasets without any memory constraints. 130
7.3 Best test-then-train accuracy for each method on the given datasets with model size below 10 MB. 130
7.4 Best test-then-train accuracy for each method on the given datasets with model size below 1 MB. 131
7.5 Normalized area under the Pareto front (APF) for each method and each dataset with models smaller than 100 MB 131

8.1 The expected size of instructions for a split node and a leaf node in a decision tree on ARM (`Raspberry PI 2`), PPC (`NXP T4240 processors`) and Intel (`Intel Core i7-6700`) processors. 143
8.2 Datasets used for experiments on Tree Framing in Section 8. 146
8.3 Number of instructions for an unoptimized if-else decision tree implementation on different architectures. 149

9.1 Supported languages in FastInference. 177

10.1 Accuracy and F_1 score for different models on the PhyNode. 181
10.2 Throughput comparison for different implementations of random forests and decision trees on the FACT data. 184
10.3 Resource comparison for different implementations of random forests and decision trees on the FACT data. 185
10.4 Power consumption for different random forests and decision implementations on the FACT data. 185
10.5 Accuracy on simulation data FACT data. 188
10.6 Significance of detection. 189
10.7 Latency of different neural net configurations using different inference engines for the FACT data. 192
10.8 BNN architectures used for the bit-error tolerance experiments. 194

B.1 Accuracy of pruning methods and leaf-refinement for models under 128 KB memory. 203
B.2 Accuracy of pruning methods and leaf-refinement for models under 256 KB memory. 204
B.3 Accuracy of pruning methods and leaf-refinement for models under 512 KB memory. 204
B.4 Accuracy of pruning methods and leaf-refinement for models under 1024 KB memory. 205
B.5 Accuracy of pruning methods and leaf-refinement for models under 2048 KB memory. 205
B.6 The F_1 score of pruning methods and leaf-refinement for models under 128 KB memory. 206
B.7 The F_1 score of pruning methods and leaf-refinement for models under 128 KB memory. 206
B.8 The F_1 score of pruning methods and leaf-refinement for models under 128 KB memory. 207
B.9 The F_1 score of pruning methods and leaf-refinement for models under 128 KB memory. 207
B.10 The F_1 score of pruning methods and leaf-refinement for models under 2048 KB memory. 208

List of Algorithms

1 Coordinate descent with Gauß-Southwell update (CD-GS). 24
2 Training of a decision tree. 32
3 Application of a decision tree. 32
4 Training of an axis-aligned binary decision tree. 36
5 Application of a BNN. 50
6 Training of a BNN. 51
7 Training of Naive Bayes. 54
8 Application of Naive Bayes. 55
9 Bagging. 73
10 Boosting as Coordinate Descent with Gauß-Southwell rule. 75
11 Training of a random forest. 83
12 Training with Data Augmentation. 89
13 Training of a Negative Correlation Forest (NCForest). 92
14 Ordering-based pruning. 101
15 Leaf-Refinement (LR) 102
16 Leaf-Refinement with L_1 regularization (L1+LR) 104
17 Online Training of a decision tree via racing. 117
18 Training of a Shrub Ensemble. 124
19 Optimization of the if-else tree. 142
20 Optimization of the native tree. 144

Part I

Fundamentals

1 | Introduction

Machine learning (ML) has become an integral part of our everyday life ranging from AI-powered search queries on the web to (partially) autonomous driving. As machine learning practitioners, we are more often than not concerned with the best predictive power re-training models over and over until we find the best combination of data processing, ML method, and hyperparameters for a specific task. This short feedback loop of training and re-evaluating the model makes machine learning fundamentally different from high-performance computing which usually runs large batch jobs on clusters over days, if not weeks. Hence, it is no surprise that machine learning has become one of the main driving forces for new software, hardware and programming approaches alike in a search for faster model training. For example, the MapReduce programming paradigm has been largely introduced with machine learning problems [CKL+06, DG08], whereas deep learning is nearly indistinguishable from advances in processing power [HB].

Every ML problem faces two fundamental resource constraints: First, the amount and quality of the data are vital for training a good predictor. Second, the energy that is available to explore and compare different ML pipelines determines the quality of the final product. While finding and extracting good data is an integral part of machine learning it is tightly interconnected with the specific use case and hence it is difficult to make general recommendations from a technical point of view. This thesis focuses on the second resource constraint: Energy. The total amount of energy used for an ML pipeline is given by

$$W = P \cdot t \tag{1.1}$$

where P is the power required to run the hardware and t is the amount of time this hardware needs to execute the pipeline. Thus, in order to reduce energy consumption we may use smaller hardware, faster methods, or both. In practice there is often a trade-off between both quantities (c.f. Figure 1.1): Hardware that has a lot of processing power can execute a pipeline very quickly, but often also requires much more energy. Similarly, less powerful hardware may take longer to execute a given pipeline while the overall energy consumption is smaller since it requires less power.

Once an ML problem has been sufficiently explored, and the performance is satisfactory, the workload shifts from training the model to continuously applying it. While models in production are still retrained from time to time in order to account for new training data and possible concept drift, they are primarily applied continuously. This continuous application suddenly can become much more costly in terms of runtime, processing power, and energy than the original model training. To illustrate this point, consider the following two examples:

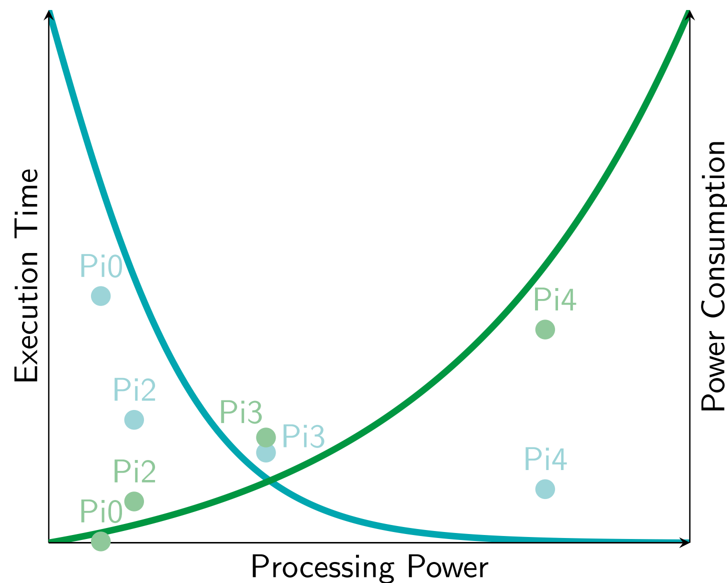


FIGURE 1.1: The power consumption and execution time for the Raspberry Pi 0, Raspberry Pi 2, Raspberry Pi 3, and Raspberry Pi 4. The processing power of each device is measured using the coremark benchmark (https://www.eembc.org/coremark/), and the execution time is its corresponding runtime. The measurements are taken from https://github.com/fm4dd/sbc-benchmarks. The energy consumption is measured running each device on full load and is taken from https://www.pidramble.com/wiki/benchmarks/power-consumption and https://raspberrypi.stackexchange.com/questions/63519/power-consumption-of-pi-zero-w. The Raspberry Pi 1 was excluded from this comparison because no reliable measurements of its power consumption could be found. Green indicates the power consumption, and blue indicates the execution time. Best viewed in color.

Online Search

One of the main reasons for the success of the internet are search engines that help the user to find relevant and interesting information using keyword searches. The second-largest search engine by market share is Microsoft's bing.com (As of 2017, see https://tinyurl.com/2evswsuv). This search engine largely utilizes machine learning models such as Gradient Boosted Trees (https://tinyurl.com/2nwyvv3m) to rank search results for their relevancy and present these to the user. Bing processed roughly 12 billion search queries a month worldwide in 2019 (https://tinyurl.com/2evswsuv), which are roughly $4\,480\,287$ queries per second. Assuming that a machine learning model is executed for each of these requests with a computational budget of 1 ms, then $4\,481$ CPUs are required to run these models and keep up with the queries. Now assuming that a regular Intel i7-7700K CPU (with a TDP of 91 W, https://tinyurl.com/4dzvmufa) is used to serve all requests, then $407\,771$ W of energy is required leading to a total energy consumption of approximately 3.57 TWh for the entire year. Differently put, the largest black coal plant in Germany (https://tinyurl.com/4p723pnv) would be required to power the model inference for all search queries on bing.com.

Self-Driving Cars

With advances in image classification and machine learning, self-driving cars are now within reach. Current prototypes, such as Tesla's autopilot, not only rely on deep-learning approaches for image recognition but also for fundamental steering (https://tinyurl.com/mz2y9abj). Thus, again, the ML model must run continuously to provide autopilot functionality. To power the deep learning model, Tesla introduced their own chip that requires 57 W to run (https://tinyurl.com/ycxvr2d5). It can be estimated that personal motorcars traveled a combined distance of 630 Billion kilometers in Germany alone in 2018 (https://tinyurl.com/yckpfrd9). Assuming the average speed is around 45 km/h [PFM+12] and that Tesla's autopilot is used in each of these trips then the autopilots are running for 14 Billion combined hours for all trips, which are equal to 0.79TWh of energy consumption per year. This is roughly the amount of energy produced by the largest run-of-the-river hydroelectricity plant in Germany (https://tinyurl.com/ky6sxv6j) to provide an autopilot feature in self-driving cars.

It is clear that the energy consumption of model application must be addressed at some point during the deployment process – for example, right at the beginning of the ML pipeline where new models are trained, as a post-processing step right before deployment, or somewhere in between these steps. Part of this process is to choose the hardware platform that ultimately trains and applies these models. Although it has become the norm to execute ML models on mid to high-end smartphones, 'smaller', more resource-constraint devices are rarely considered even though they have the potential to offer a better performance-energy trade-off.

Runtime and the efficient use of the hardware was always a major concern in machine learning. However, the direct connection between hardware and machine learning – especially in the context of small, resource-constraint systems – has only recently become an issue in the machine learning community. Deep learning has transformed the landscape of machine learning and hardware platforms alike, as it not only achieved super-human performance in some tasks but required excessive amounts of computation to do so. Hence, it not only sparked new interest in neural network research but also resulted in new advances in hardware accelerators as well as a whole new ecosystem of frameworks and software libraries. The miniaturization of deep learning closely followed these new trends in an attempt to reduce its resource consumption while preserving its performance. Over the course of this thesis, the term 'TinyML' became more and more popular to describe the combination of machine learning and small, resource-constrained devices. Somewhat confusing, this term is most often used to describe the miniaturization of deep learning and its application on small devices, but not the application of 'classical' methods in resource-constraint contexts. However, these classical methods often outperform deep networks in many applications while using fewer resources.

One of the major drawbacks of deep learning is that it requires the consecutive computation of a matrix-vector product with floating-point inputs. In order to compute a floating-point product, roughly 37 times more energy is required compared to an integer product (cf. Table 1.1). Moreover, `double` variables take two times more space than integers and thereby doubling the high energy cost of accessing caches and RAM. Hence, the miniaturization of deep learning – while certainly a fruitful endeavor – will always have to battle the inefficiency of floating-point computations. A natural question arises here: What happens if we remove floating-point computations altogether? Can we achieve a good classifier that does not use any floating-point

operations?

TABLE 1.1: Energy consumption of instructions and memory access based on word size in CMOS 45nm structures [Hor14]. For comparison, the relative costs depict the costs relative to a single 32-bit integer addition.

Operation	Energy [pJ]	Relative Cost
8 bit integer ADD	0.03	0.3
32 bit integer ADD	0.1	1
8 bit integer MULT	0.2	2
16 bit float ADD	0.4	4
32 bit float ADD	0.9	9
16 bit float MULT	1.1	11
32 bit integer MULT	3.1	31
32 bit float MULT	3.7	37
Cache access	5-50	50-500
DRAM access	320-640	3200-6400

1.1 Contributions and Outline

Discrete classifiers are machine learning models that can be executed *without* any floating-point operations. Moreover, many discrete classifiers can also be trained without floating-point operations, making them ideal candidates for small devices. Unfortunately, the predictive performance of discrete classifiers often lacks behind their continuous counterparts. To circumvent this problem, one can train multiple discrete models and combine them into an ensemble that offers better predictive performance while maintaining a low resource consumption. The training and application of an ensemble on a small device that might already be challenged by a single model seem counterintuitive at first but brings numerous advantages with it:

- Ensembles are well-known to improve the performance over single classifiers, especially when the single models are 'weak'.
- Ensembles of discrete classifiers offer fine-grained control over the resource consumption of the final model. One can either increase the complexity of the individual ensemble members, increase the size of the ensemble or balance both quantities to achieve the best performance.
- Ensembles are robust to changes in the hyperparameters: Changing a single hyperparameter (e.g., the number of classifiers in the ensemble) usually does not change the overall behavior of the ensemble, and the predictive performance remains similar across regions of hyperparameters.

Arguably the most well-known and one of the most widely used machine learning methods are random forests (RF), which are an ensemble of discrete classifiers. Interestingly, despite their wide usage, random forests have not been explicitly studied in the context of small devices in recent years. And indeed, RFs are often regarded as 'small' models compared to, e.g., deep learning models. In this thesis, we will revisit additive ensembles of discrete classifiers and, in particular, RFs in the context of

small devices. Due to their wide use, large parts of the thesis focus on decision tree ensembles, but more recent developments in the context of discrete classifiers, such as binarized neural networks, will also be discussed. This thesis is split into four parts and makes the following contributions:

Part I: Chapter 1 gives an introduction and offers an overview of the thesis. Chapter 2 introduces the necessary background for this thesis. It first surveys the landscape of small devices and introduces the basic principles of the von Neumann computer architecture. Then, three types of discrete classifiers, namely decision trees (DT), binarized neural networks (BNN), and naive Bayes (NB), are introduced. All methods are presented in a general framework that allows practitioners to fine-tune the algorithmic implementation for their specific task at hand. For each method, the properties of universal function approximation and consistency are discussed. While most of this chapter summarizes existing works, it contains some original work in the context of decision trees. In particular, the universal function approximation property of DTs is shown, to the best of my knowledge, for the first time using the Simple Function Approximation Theorem. This theorem simplifies the analysis while implying that any model tree with constants in the leaf nodes is a universal function approximator.

Part II: Chapter 3 formally introduces additive ensembles and the bias-(co-)variance decomposition. It introduces the well-known bias-variance decomposition for the mean-squared error and surveys existing decompositions that generalize it to other loss functions. Then, a novel decomposition for twice differentiable loss functions using a second-order Taylor approximation is presented, and examples are discussed at the end of the chapter. In particular, this novel decomposition shows that for some losses, a second-order approximation can have an unbounded remainder implying that such losses cannot be fully explained in terms of bias and variance, but higher moments are also required. Chapter 4 introduces the novel generalized negative correlation learning (GNCL) objective that is based on the new bias-variance decomposition. Existing ensembling algorithms are surveyed, showing how many of them can be recovered as a special case in the novel GNCL framework. Chapter 5 discusses the double-descent phenomena in the context of additive tree ensembles and, specifically, random forests. While this phenomenon has been studied in the context of neural networks and deep learning, it has not been studied explicitly for additive ensembles and random forests. This chapter offers the first comprehensive analysis of the complexity of additive tree ensembles in the PAC framework based on the currently available theory and shows experimentally that there is no double-descent in random forests.

Part III: Chapter 6 focuses on the problem of training a small ensemble that can be deployed to small devices. To do so, the framework of ensemble pruning is introduced, which is based on the principle of 'overtrain and remove'. Similarly, the idea of leaf-refinement of DTs is introduced as a different approach to training small ensembles. Then, both approaches are combined into a novel objective that refines the leaves in the DTs of an ensemble while removing unnecessary trees using a L_1 regularization term. A novel pruning algorithm is introduced that minimizes this objective using proximal gradient descent, and the chapter is concluded with an extensive experimental analysis that shows the usefulness of the novel pruning algorithm. Chapter 7 expands the ideas on ensemble pruning and leaf-refinement to the more challenging setting of on-device training. To do so, online learning in which each observation is

presented exactly once to the algorithm is formally introduced. Then, existing ensembling algorithms for online learning are discussed. The novel leaf-refinement and pruning objective is adapted to the online-learning setting. In particular, the L_1 regularization is replaced with a L_0 regularization that guarantees a maximum memory consumption during training at all times. This leads to a more challenging optimization scenario which is solved by a novel shrub ensembles algorithm based on proximal gradient descent. After the behavior of shrub ensembles is formally established, the chapter finishes with an experimental analysis that highlights the usefulness of shrub ensembles in resource constraint environments. In chapter 8, the deployment and optimal implementation of (tree) ensembles are discussed. Based on a novel probabilistic view of DT inference, several optimizations to existing implementation schemes are presented that maximize cache efficiency. In a series of experiments, the benefits of the novel memory layout are shown. The second part of the chapter analyzes the optimal implementation of DTs from a more theoretical perspective. To do so, a theoretical von Neumann architecture, as well as a theoretical FPGA design, are introduced, which is then combined with the probabilistic view of DT inference to determine the optimal implementation for a given tree ensemble and a given target architecture.

Part IV: Chapter 9 gives a short overview of existing software and introduces two software projects that were implemented as part of this thesis. In particular, the PyPruning library is introduced, which includes ensemble pruning algorithms and allows the implementation of novel ensemble pruning algorithms. Second, the model compiler FastInference is presented that generates architecture- and model-specific code for pre-trained models. Chapter 10 gives an overview of three practical use cases of discrete-classifier ensembles that have been studied over the course of the research for this thesis. It starts with applying ensemble pruning and leaf refinement in the context of the PhyNetLab. The PhyNetLab is an IoT warehouse test environment that has been built as a part of the SFB876 project at the TU Dortmund University. The task in this application is to deploy ML models to ultra-low power devices so that these devices can predict the position of their respective storage boxes in a warehouse given environmental sensor measurements. The second example is due to the physics cooperation in the SFB 876. Here, astrophysicists are monitoring the sky with a Cherenkov Telescope and use random forests to distinguish interesting gamma-ray events from the hadronic background noise of the universe. This chapter details how to deploy random forests for such a task and discusses binarized neural networks that can process the raw data directly from the telescope using a reference FPGA implementation as an alternative. Third, this chapter showcases how BNNs could potentially be used on devices with approximate memory. Approximate memory requires much less energy than conventional memory architectures but also has a much higher bit-error rate during readout. Hence, error-resilient BNNs that maintain a high accuracy even under a large number of bit errors seem favorable. To do so, the chapter introduces a novel loss function based on the max-margin approach that leads to more resilient BNNs that can cope with large amounts of bit errors. Finally, chapter 11 concludes the thesis.

1.2 Publications Covered by this Thesis

The work on this thesis was partially funded by the Deutsche Forschungsgesellschaft (DFG) through their grant on the collaborative research center SFB 876, Providing

Information by Resource-Constrained Data Analysis. This thesis covers the following scientific publications that emerged as a part of this project:

Peer-Reviewed Publications

- Sebastian Buschjäger, Sibylle Hess, and Katharina Morik. Shrub ensembles for online classification. In *Proceedings of the Thirty-Sixth AAAI Conference on Artificial Intelligence (AAAI-22)*. AAAI Press, 2022. URL: https://arxiv.org/abs/2112.03723, doi:10.1609/aaai.v36i6.20560
- Sebastian Buschjäger, Jian-Jia Chen, Kuan-Hsun Chen, Mario Günzel, Christian Hakert, Katharina Morik, Rodion Novkin, Lukas Pfahler, and Mikail Yayla. Margin-maximization in binarized neural networks for optimizing bit error tolerance. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2021, Grenoble, France, February 1-5, 2021*, pages 673–678. IEEE, 2021. doi:10.23919/DATE51398.2021.9473918
- Sebastian Buschjäger, Philipp-jan Honysz, and Katharina Morik. Randomized outlier detection with trees. *International Journal of Data Science and Analytics*, 13(2):91–104, 2020. doi:10.1007/s41060-020-00238-w
- Sebastian Buschjäger, Lukas Pfahler, Jens Buß, Katharina Morik, and Wolfgang Rhode. On-site gamma-hadron separation with deep learning on fpgas. In Yuxiao Dong, Dunja Mladenic, and Craig Saunders, editors, *Machine Learning and Knowledge Discovery in Databases: Applied Data Science Track - European Conference, ECML PKDD 2020, Ghent, Belgium, September 14-18, 2020, Proceedings, Part IV*, volume 12460 of *Lecture Notes in Computer Science*, pages 478–493. Springer, 2020. doi:10.1007/978-3-030-67667-4_29
- Sebastian Buschjäger, Thomas Liebig, and Katharina Morik. Gaussian model trees for traffic imputation. In *Proceedings of the International Conference on Pattern Recognition Applications and Methods (ICPRAM)*, pages 243–254. SciTePress, 2019. URL: https://www.scitepress.org/PublicationsDetail.aspx?ID=g+tVIY+KNts=&t=1
- Sebastian Buschjäger, Kuan-Hsun Chen, Jian-Jia Chen, and Katharina Morik. Realization of random forest for real-time evaluation through tree framing. In *The IEEE International Conference on Data Mining series (ICDM)*, pages 19–28, November 2018. URL: https://ieeexplore.ieee.org/document/8594826, doi:10.1109/ICDM.2018.00017
- Sebastian Buschjäger and Katharina Morik. Decision tree and random forest implementations for fast filtering of sensor data. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 65-I(1):209–222, 2018. doi:10.1109/TCSI.2017.2710627

Non-Peer-Reviewed Publications

- Sebastian Buschjäger and Katharina Morik. There is no double-descent in random forests. *CoRR*, abs/2111.04409, 2021. URL: https://arxiv.org/abs/2111.04409, arXiv:2111.04409 (Currently under review)

- Sebastian Buschjäger and Katharina Morik. Joint leaf-refinement and ensemble pruning through l1 regularization. *CoRR*, abs/2110.10075, 2021. URL: https://arxiv.org/abs/2110.10075, arXiv:2110.10075 (Currently under review)
- Sebastian Buschjäger, Jian-Jia Chen, Kuan-Hsun Chen, Mario Günzel, Katharina Morik, Rodion Novkin, Lukas Pfahler, and Mikail Yayla. Bit error tolerance metrics for binarized neural networks. *CoRR*, abs/2102.01344, 2021. URL: https://arxiv.org/abs/2102.01344, arXiv:2102.01344
- Sebastian Buschjäger, Lukas Pfahler, and Katharina Morik. Generalized negative correlation learning for deep ensembling. *CoRR*, abs/2011.02952, 2020. URL: https://arxiv.org/abs/2011.02952, arXiv:2011.02952

1.3 Publications Not Covered by this Thesis

Over the course of this thesis I was fortunate enough to collaborate with my colleagues in the SFB 876 project A1 on numerous other topics leading to the following publications:

- Kuan-Hsun Chen, ChiaHui Su, Christian Hakert, Sebastian Buschjäger, Chao-Lin Lee, Jenq-Kuen Lee, Katharina Morik, and Jian-Jia Chen. Efficient realization of decision trees for real-time inference. *ACM Transactions on Embedded Computing Systems*, dec 2021. doi:10.1145/3508019
- Mikail Yayla, Sebastian Buschjäger, Aniket Gupta, Jian-Jia Chen, Jorg Henkel, Katharina Morik, Kuan-Hsun Chen, and Hussam Amrouch. Fefet-based binarized neural networks under temperature-dependent bit errors. *IEEE Transactions on Computers*, pages 1–1, 2021. URL: https://ieeexplore.ieee.org/document/9513530, doi:10.1109/TC.2021.3104736
- Christian Hakert, Mikail Yayla, Kuan-Hsun Chen, Georg von der Brüggen, Jian-Jia Chen, Sebastian Buschjäger, Katharina Morik, Paul R. Genssler, Lars Bauer, Hussam Amrouch, and Jörg Henkel. Stack usage analysis for efficient wear leveling in non-volatile main memory systems. In *1st ACM/IEEE Workshop on Machine Learning for CAD (MLCAD)*, 2019. URL: https://ieeexplore.ieee.org/abstract/document/9142113

Outside this project I also had the opportunity to collaborate with my colleagues at the Artificial Intelligence Unit in Dortmund and to pursue other research projects leading to the following publications:

- Sebastian Buschjäger, Philipp-Jan Honysz, Lukas Pfahler, and Katharina Morik. Very fast streaming submodular function maximization. In Nuria Oliver, Fernando Pérez-Cruz, Stefan Kramer, Jesse Read, and José Antonio Lozano, editors, *Machine Learning and Knowledge Discovery in Databases. Research Track - European Conference, ECML PKDD 2021, Bilbao, Spain, September 13-17, 2021, Proceedings, Part III*, volume 12977 of *Lecture Notes in Computer Science*, pages 151–166. Springer, 2021. doi:10.1007/978-3-030-86523-8_10
- Philipp-Jan Honysz, Alexander Schulze-Struchtrup, Sebastian Buschjäger, and Katharina Morik. Providing meaningful data summarizations using examplar-based clustering in industry 4.0. *CoRR*, abs/2105.12026, 2021. URL: https://arxiv.org/abs/2105.12026, arXiv:2105.12026

- Philipp-Jan Honysz, Sebastian Buschjäger, and Katharina Morik. Gpu-accelerated optimizer-aware evaluation of submodular exemplar clustering. *CoRR*, abs/2101.08763, 2021. URL: https://arxiv.org/abs/2101.08763, arXiv:2101.08763

- Sebastian Buschjäger, Katharina Morik, and Maik Schmidt. Summary extraction on data streams in embedded systems. In Moamar Sayed Mouchaweh, Albert Bifet, Hamid Bouchachia, João Gama, and Rita Paula Ribeiro, editors, *Proceedings of the Workshop on IoT Large Scale Learning from Data Streams co-located with the 2017 European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML-PKDD 2017), Skopje, Macedonia, September 18-22, 2017*, volume 1958 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2017. URL: http://ceur-ws.org/Vol-1958/IOTSTREAMING3.pdf

- Katharina Morik, Christian Bockermann, and Sebastian Buschjäger. Big data science. *Künstliche Intelligenz*, 32(1):27–36, 2018. doi:10.1007/s13218-017-0522-8

2 | Background

Discrete classifiers and ensembles of such can be powerful classifiers ideally suited for small, embedded systems. This chapter offers the necessary background to study them. The following section will give an overview of computer architectures in general and small devices specifically. After that, some general terminology in machine learning is discussed. The chapter finishes with three discrete classifiers that are considered throughout this thesis.

2.1 Small Devices

There is a wide range of different computing devices available, starting from entry points into large networks (e.g., edge devices or Internet Of Things (IoT) devices), tightly integrated computing devices with sensing and acting capabilities (e.g., embedded systems) as well as ‘regular’ desktop and server hardware. In the context of this thesis, we will focus on small devices: A computing device is a *small device* whenever the task we want to solve with it is much more complex than the resources offered by that device. In this sense, every computing device can be considered ‘small’ if the task we want to solve is big enough. Likewise, any device can be ‘big’ if it only performs a comparably simple task.

Since this definition does not exclude or include any specific types of hardware it can sometimes be difficult to derive general guidelines for designing resource-efficient ML algorithms. Hence, to ease the discussion and make it more tangible, we will look at embedded systems and IoT devices specifically to study resource-efficient ML algorithms theoretically as well as practically in experiments. However, virtually all algorithmic aspects discussed in this thesis can directly be translated to desktop or server hardware as well. Table 2.1 gives some examples of typical microcontroller units (MCUs) found in embedded systems and edge devices. For comparison, the Intel i7-770K as a typical desktop / server CPU is also added. Two main conclusions can be drawn from this table: First, the landscape of IoT and embedded devices is very heterogeneous. It ranges from very small devices clocked at 16 Mhz with only little memory and a limited set of features to feature-rich devices clocked in the Gigahertz range with an abundance of memory. Second, there is a clear trend in the energy consumption of devices. The ‘smaller’ a device is, the less power it requires.

2.1.1 Von Neumann architecture

The vast majority of CPUs are implemented using the von Neumann architecture, in which code and data reside in the same memory. A comprehensive overview of von Neumann CPUs can be found in [HP11]. A sketch of the von Neumann architecture is depicted in Figure 2.1a. Code and data are fetched using a common communication bus. The control logic of the CPU decodes instructions and loads data into registers,

TABLE 2.1: Typical microcontroller units (MCUs) found in edge and IoT devices. The top group shows bare-metal MCUs, which typically do not run an operating system. The bottom group shows MCU, which typically also runs an operating system. ✓ denotes the availability of a feature, ✗ marks its absence, and ✚ denotes optional/partial support. The original table is due to [BFC19] but the float, SIMD, and architecture columns have been added by taking the corresponding values from the referenced data sheets. For comparison, the Intel i7-7700K CPU has also been added as a typical desktop / server CPU. The data-sheet can be found under.

MCU	Clock	Float	Arch.	SIMD	Flash	(S)RAM	Power
Arduino Uno (ATMega128P)	16MHz	✗	8bit	✗	32KB	2KB	12mA
Arduino Mega (ATMega2560)	16MHz	✗	8bit	✗	256KB	8KB	6mA
Arduino Nano (ATMega2560)	16MHz	✗	8bit	✗	26-32KB	1-2KB	6mA
STM32L0 (Cortex-M0)	32MHz	✗	32bit	✗	192KB	20KB	7mA
Arduino MKR1000 (Cortex-M0)	48MHz	✗	32bit	✗	256KB	32KB	4mA
Arduino Due (Cortex-M3)	84MHz	✗	32bit	✗	512KB	96KB	50mA
STM32F2 (Cortex-M3)	120MHz	✗	32bit	✗	1MB	128KB	21mA
STM32F4 (Cortex-M4)	180MHz	✚	32bit	✚	2MB	384KB	50mA
Raspberry PI A+	700MHz	✓	32bit	✗	SD Card	256MB	80mA
Raspberry PI Zero	1GHz	✓	32bit	✗	SD Card	512MB	80mA
Raspberry PI 3B	4@1.2GHz	✓	64bit	✓	SD Card	1GB	260mA
Intel i7-7700K	4@4.5GHz	✓	64bit	✓	HDD/SSD	2 - 64GB	$\approx$ 80A

accordingly. Operations are performed on these registers, and results are written back into the main memory using the communication bus if needed.

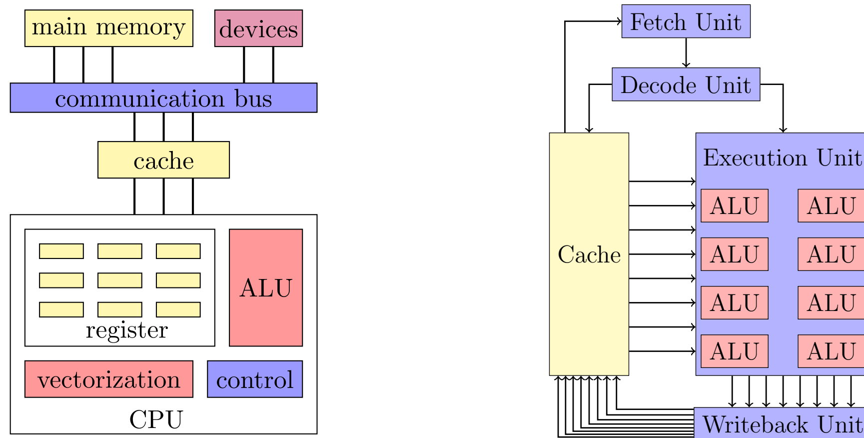
Since the common communication bus is used for data and instruction codes, it forms the bottleneck of the von Neumann architecture. Moreover, a *memory wall* amplifies this bottleneck: With increasing manufacturing capabilities, CPUs have become faster and faster, nearly doubling their processing power every 2 years. Memory access speed as well as memory transfer rates, however, could not keep up with this rapid speed-up, making access to main memory a magnitude slower than data processing inside the CPU [BC]. In a von Neumann architecture, instructions are clocked, i.e., the execution of each instruction is synchronized to a common clock. Von Neumann CPUs are inherently a single instruction, single data (SISD) system, in which one instruction performs one operation on one data item per clock. In order to cope with data and computation-intensive applications, several enhancements for von Neumann CPUs have been introduced:

Parallelized memory access

CPUs perform operations on packs of bits called words. The word size of a CPU thus denotes how fine-grained a CPU can access individual bits. In order to reduce address lookup, memory access is performed on packs of words in which each memory access loads neighboring words.

Pipelining

Executing an instruction involves multiple tasks, such as decoding the instruction, loading its operands, doing the desired calculation, and finally saving the result in some register or into main memory. However, once an instruction is decoded, the



(A) Sketch of the von Neumann architecture. The communication bus connects the main memory and cache, which in turn is connected to the CPU. The CPU performs arithmetic logical operations on register values using the arithmetic logical unit (ALU). Vectorization instructions are performed by the vectorization unit on special vector registers. A control logic administers register accesses and instruction execution.

(B) Sketch of a four-stage pipeline. The fetch unit fetches the next instruction from memory, the decode unit decodes its operands, whereas the execution units actually execute the operation. Finally, the writeback unit writes any results back into the memory.

FIGURE 2.1: Sketch of the von Neumann architecture with its pipeline.

functional unit used for decoding is in an idle state. Pipelining (c.f. Figure 2.1b) now explicitly breaks down every instruction into these smaller sub-operations and tries to maximize the utilization of every functional unit. To do so, the next instruction is already fetched and decoded, while the current instruction just loads its operands. Pipelines nearly utilize the given hardware to its fullest but sometimes need to be stopped or even flushed and restarted. Every time a program needs to branch, e.g. when an if-else instruction is executed, then the branch condition needs to be evaluated first before any further operations can be fetched. This introduces a halt of the pipeline, also known as `NOP` (no operation) into the pipeline reducing its overall efficiency.

Branch prediction

Since a pipeline needs to be stopped and sometimes even flushed when a branch occurs, CPUs may perform a simple branch prediction. The next possible branch is predicted based on the branches seen so far, and the next instruction is fetched based on this prediction. If the prediction was correct, nothing happens, and execution continues. However, if the prediction is wrong, the pipeline has to be flushed, and the state *before* the last branch needs to be reverted, adding an even larger performance penalty.

Vectorization

With more and more transistors available, more specialized hardware can be added to the same CPU circuit. Therefore, many CPUs offer additional digital-signal processing

(DSP) and single instruction multiple data operations (SIMD). These vectorization instructions execute the same operation on a vector of inputs and have the potential to greatly improve the performance of data-intensive operations.

Memory hierarchy

Memory is arranged in hierarchies so that instructions and data can be fetched from different hierarchy levels. The idea of a memory hierarchy is to hide memory latencies by providing different hierarchy levels with different memory speeds. On the lower hierarchy levels, small but fast memory, such as caches, can be found, whereas, on higher hierarchy levels, larger but slower memory, such as the main memory, is placed. While instructions and data reside in the same memory in the von Neumann architecture, they are often placed in different caches so that they do not interfere with each other. Data is placed in the (usually larger) data cache (D-cache), whereas instructions are placed in the (usually smaller) instruction cache (I-cache). The key assumption of the memory hierarchy is the *locality*:

- **Temporal locality:** Recently accessed items will be accessed in the near future, e.g. constants that are used during the execution of loops.
- **Spatial locality:** Items at addresses close to the addresses of recently accessed items will be accessed in the near future, e.g. sequential accesses to elements of an array.

When retrieving a data item, the CPU will first attempt to find it in the lowest cache. If the data item is found, this lookup results in a cache hit, and processing can continue. If the data item is not found in the cache, the lookup results in a cache miss, and the CPU will access the next higher cache in the memory hierarchy until the data item is fetched from the main memory. Figure 2.2 shows an example of a memory hierarchy with two CPUs. Note that due to the manufacturing, data items in the cache are arranged in cache lines, and the CPU receives the entire cache line upon requesting an item.

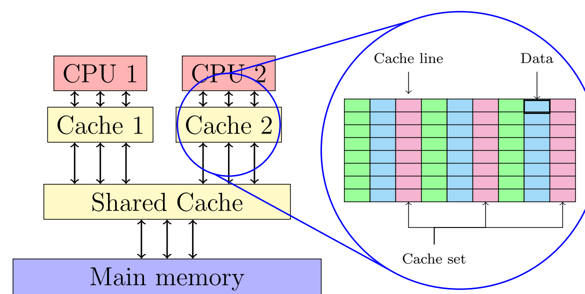


FIGURE 2.2: Sketch of a memory hierarchy. Each CPU has its own unique cache and a larger shared cache. If data cannot be found in either of these caches, the main memory is accessed.

By design, a cache can only hold limited information, and hence at some point, some information in the cache must be replaced. There are three main strategies to do so: A fully associative cache can replace any cache line with any content from the main memory, maximizing the utilization of the cache for higher operating costs. In a set-associative cache, each memory block of the main memory is mapped to several

cache lines. A memory block can only be placed into one of its corresponding cache lines. The operating costs for a set-associative cache are smaller because only the set of cache lines must be checked instead of the entire cache. However, the utilization of the cache also suffers because not all cache lines can be accessed freely. A direct-mapped cache is an extreme form of a set-associative cache in which each memory block can only be placed into a single cache line. This approach is by far the simplest but can severely underutilize the cache. During the operation of the cache, three types of cache misses can occur: compulsory, conflict, and capacity cache misses. The compulsory misses are due to the first access to a memory block, which by definition, is not in the cache. The capacity misses occur when some memory blocks are discarded from the cache due to the limited cache capacity, i.e., the program working set is much larger than the cache capacity. The conflict misses occur in set-associative or direct-mapped caches when several blocks are mapped to the same cache set.

Representing real numbers

One of the most fundamental challenges in the design of CPUs is how to represent and process numbers (i.e., add, subtract, multiply, divide, etc.) in a binary format. A comprehensive introduction to this topic justifies a thesis in its own right, so we will skip over the basics on how to represent numbers here. A deeper discussion on this topic can, e.g., be found in [Tan].

As depicted in Table 1.1 the energy cost of an operation differs significantly depending on the data type and data width. Especially float point numbers that require circuitry for dealing with exponents, mantissa, and normalization are 4 to 37 times more expensive than integer operations. Hence, many small devices do not natively support floating-point data types, as shown in Table 2.1, and compilers often resort to the (even more costly) emulation of floating-point data types through software. In these cases, the use of fixed-point data types can be beneficial. Fixed-point numbers are numbers with a pre-set number of bits before and after the decimal point:

$$X_{fx} = \overbrace{\underbrace{X_{(1)}X_{(0)}}_{N_l} \bullet \underbrace{X_{(-1)}X_{(-2)}X_{(-3)}X_{(-4)}}_{N_r}}^{N_t}$$

where N_t is the total number of bits, N_l is the number of bits left of the decimal point, and N_r is the number of bits to the right of the decimal point. Fixed and floating-point numbers are easily converted to one another:

$$\text{Fixed} \rightarrow \text{float: } X_{fl} = \sum_{i=0}^{N_l} X_{(i)} 2^i + \sum_{i=-1}^{-N_r} X_{(-i)} 2^{-i} \quad (2.1)$$

$$\text{Float} \rightarrow \text{fixed: } X_{fx} = \lfloor X_{fl} \cdot 2^{N_r} \rfloor \quad (2.2)$$

The addition (and subtraction) for fixed-point numbers mostly remain the same, whereas multiplication (and division) require the correction of the additional scaling by 2^{-N_r} (and 2^{N_r} respectively):

$$\text{Addition: } X''_{fx} = X_{fx} + X'_{fx} = \lfloor X_{fl} \cdot 2^{N_r} \rfloor + \lfloor X'_{fl} \cdot 2^{N_r} \rfloor = \lfloor (X_{fl} + X'_{fl}) \cdot 2^{N_r} \rfloor \quad (2.3)$$

$$\text{Multiplication: } \lfloor X_{fl} \cdot 2^{N_r} \rfloor \cdot \lfloor X'_{fl} \cdot 2^{N_r} \rfloor = \lfloor (X_{fl} \cdot X'_{fl}) \cdot (2^{N_r} \cdot 2^{N_r}) \rfloor \quad (2.4)$$

The advantage of fixed-point arithmetic is that they only require two integer operations plus the handling of overflows. Moreover, the correction during multiplication and division can be implemented via shift operations making fixed-point operations much cheaper than floating-point operations in most cases. On the other hand, fixed-point arithmetic introduces a fixed error into the computations with a much smaller range than floating-point: Its minimum and maximum value are given by the number of bits before the decimal point, i.e. $\pm 2^{N_r - 1}$. The number of bits after the decimal point determines the resolution, i.e. $\frac{1}{2^{N_l}}$. Hence, a 32 bit integer with $N_r = 15$ (using one sign bit) and $N_l = 16$ can represent numbers between $[-32\ 768, +32\ 768]$ with an resolution of roughly $1.52587890625 \cdot 10^{-5}$. In contrast, the IEEE-75 floating-point standards guarantees to represent numbers from roughly $1.18 \cdot 10^{-38}$ to $3.4 \cdot 10^{38}$ with a resolution of around 10^{-7}, hence offering a wider range of numbers *while* having a better resolution. Last, the IEEE-754 guarantees that floating-point numbers in this format can be compared using the same circuitry as integers (in addition to some handling of special values such as `NaN` or `infinity`) [IEE19] which makes it a very attractive data format if specialized circuitry for computations with it is available. The choice of using fixed-point or floating-point arithmetics often comes down to the specifics of the problem at hand and the hardware that is used. For example, sensor values are often reported in fixed-point numbers, and applying a decision tree directly to these values will likely not lead to any errors at all. On the other hand, the application of deep neural networks requires the consecutive computation of matrix-vector products that can suffer from fixed-point data types.

Instruction Set Architecture (ISA)

There are many more design choices that impact the performance of a CPU that can be summarized as the Instruction Set Architecture (ISA) of the CPU. The instruction set architecture of a CPU describes the instructions available to the programmer and compiler. There are two main design principles: A Reduced Instruction Set Architecture (RISC) only offers a limited set of instructions, and more complex instructions must be built manually by combining the available instructions. The benefit of a RISC architecture is that the pipeline is typically small, which reduces the effects of flushing, and the circuit can be optimized more aggressively for the few available instructions. A RISC architecture can typically be found in ARM CPUs. A Complex Instruction Set Architecture (CISC), on the other hand, offers a very rich set of instructions. The advantage of this approach is that the programmer or compiler does not need to combine instructions to form new ones but can often use what is already given. This results in small code sizes and often faster code. The downside of this approach is that the pipeline can become very large (e.g. up to 20 stages in Intel X86) so flushing it can have severe effects on the performance. Moreover, the circuits cannot be optimized as aggressively as in RISC since there are more instructions to be covered. A CISC architecture can typically be found in Intel X86 CPUs.

2.1.2 Beyond the von Neumann Architecture

While the von Neumann processor architecture is arguably the most widespread CPU architecture they are targeted towards the average use-case. Hardware accelerators, on the other hand, are application-specific circuits (ASICs) designed to interact with a CPU targeting very specific tasks. A hardware accelerator can potentially be much

more energy efficient and much quicker in performing its tasks compared to a general purpose CPU, but can only perform a limited set of functions.

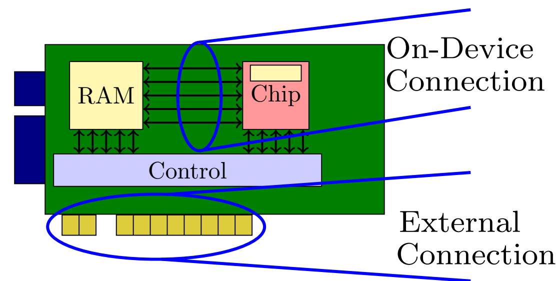


FIGURE 2.3: Sketch of a hardware accelerator. The chip contains the actual ASIC, whereas the accelerator itself usually has some additional memory. The internal on-device connections between ASIC and memory are often times much faster than the external memory connection used to interact with the accelerator itself.

Accelerators can come as an extension card (e.g., graphics processing unit) that uses a common communication bus such as PCIe to connect it to a CPU, or they can be directly integrated into the CPU chip itself. Figure 2.3 shows a sketch of a PCIe extension card. The chip contains the actual ASIC, whereas the accelerator itself often has some additional memory. The internal on-device connections between ASIC and memory are often times much faster than the external memory connection used to interact with the accelerator itself. In the context of machine learning applications, the three most common hardware accelerators are:

Graphics Processing Units (GPU) and Digital Signal Processing Units (DSP)

DSP and GPUs are special circuits for processing digital signals and computer graphics. They are characterized by a large number of parallel processing capabilities (e.g., SIMD) as well as built-in specialized features, e.g., a fast Fourier Transform (FFT). While DSPs usually offer a manufacture-dependent and comparably low-level interface, GPUs have evolved into general-purpose massively parallel accelerators that can be programmed in high-level programming languages such as OpenCL or CUDA. Figure 2.4 shows the sketch of a general-purpose GPU. It consists of a control logic that handles buffer allocations, data transfer as well as instructions, whereas a large fleet of processing cores performs the actual processing. In this way, GP-GPUs share many characteristics of SIMD instructions inside the CPU, although they are much more independent of the CPU. DSP and GPUs have long been used in the design of computing systems. Many IoT devices and embedded CPUs come with some form of built-in DSP instructions, whereas more powerful edge devices sometimes even have GPU support available.

Tensor Processing Units (TPU)

TPUs are a relativity recent type of accelerator specifically targeted toward deep learning applications. Due to the fluidity of the market, it is impossible to give a comprehensive overview here (a more detailed and updated list of AI accelerators

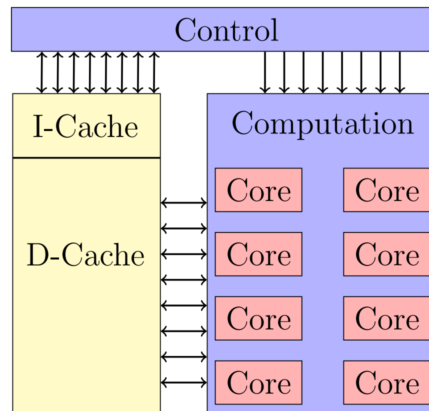


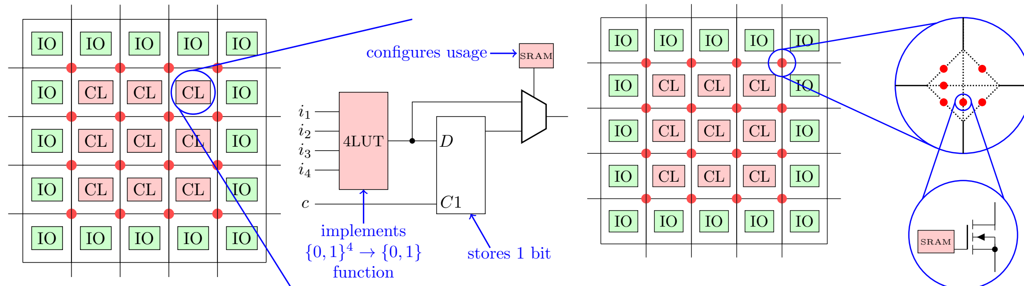
FIGURE 2.4: Sketch of a GP-GPU. It consists of a control logic that handles buffer allocations, data transfer as well as instructions, whereas a large fleet of processing cores performs the actual processing. Results are stored in the data cache (D-Cache), whereas instructions reside in the instruction cache (I-Cache).

can, e.g., be found at https://basicmi.github.io/AI-Chip/). Their general structure closely follows that of GP-GPUs depicted in Figure 2.4, but they offer specialized instructions for deep learning such as batched matrix-matrix-multiplications, inbuilt activation functions, and inbuilt loss functions. TPUs were first proposed by Google in [JYP+17] as a custom in-house data accelerator, which then also became available for edge devices and smartphones (see e.g.https://coral.ai/products/). NVIDIA closely followed this trend and includes specialized TensorCore circuits into their GP-GPU design [Nvi18]. Similarly, Intel offers a so-called Neural Compute Stick USB accelerator to process deep learning applications over USB [Int18]. Last, start-ups such as Sambanova, GraphCore, or Cerebras offer even more specialized solutions that map the entire computation graph of a neural network on a specialized circuit.

Field-Programmable Gate Arrays (FPGA)

FPGAs are reconfigurable hardware, i.e., their functionality is encoded in hardware that can be reprogrammed before and even during execution. Logic gates are combined with flip-flops into configurable logic cells (cf. Figure 2.5a). Each logic cell contains a truth table of size 2^t storing a boolean function $f\colon \{0,1\}^t \to \{0,1\}$. By programming the logic table, a logic block can image every boolean function of size t. Alternatively, a configurable logic block (CLB) can be configured to act as memory if necessary. In order to achieve more complex circuits, the CLBs are connected to each other by signal routes. Signal routing between logic cells is performed by flip-flops and transistors that statically enable or disable signal routes (see Figure 2.5b). The functional logic of the FPGA can be fully specified by programming the look-up tables and flip-flops for signal routing, effectively making the FPGA a reconfigurable circuit. FPGAs are functional complete [HD08].

FPGAs are essentially free to image every architecture desired for a given problem. This freedom, however, comes along with two major disadvantages. FPGAs only have limited resources and must mimic logic gates with truth tables. Therefore, even though FPGAs are functional complete in theory, they cannot express every function



(A) Sketch of Configurable logic blocks (CLB) of FPGAs. The logic block consists of a look-up table with 4 inputs and one flip-flop. Thus, it can represent any Boolean function with 4 inputs or save 1 bit.

(B) Sketch of Signal Routing in FPGAs. Each crossing has 6 transistor attached to it, which control each of the lanes. Transistors are programmed using a 1 bit SRAM cell.

FIGURE 2.5: Sketch of a Field Programmable Gate Array (FPGA).

due to resource constraints. Second, FPGAs perform operations at a much lower speed than CPUs do since they do not implement logic gates directly. In order to cope with these limitations, several extensions for FPGAs have been introduced, including ([BRS13, Xil]):

- **Block memory:** Configurable logic blocks are valuable processing resources that should not be used as memory, if possible. Therefore, FPGAs usually contain an additional block memory which can be used to store intermediate values. Access speed is similar to caches in CPUs.

- **DSP units:** Basic arithmetic, such as addition or multiplication, must be performed in most circuits. Therefore, FPGAs offer dedicated digital signal processing units (DSP) performing these kinds of tasks to save logic blocks.

Since FPGAs can be used to build any hardware architecture, they do not operate on fixed words. Data access and computation can be tailored specifically for the given task at hand. Additionally, it is not required that FPGAs are clocked. However, most practical implementations use block memory or the DSP units, which typically use a standardized clocked interface.

2.2 Mathematical Optimization

Mathematical optimization and machine learning share a strong connection and are often times indistinguishable from each other. Many ML methods are in their core optimization methods that solve a very specific optimization problem, and the resulting ML models are specific solutions to that problem. On the other hand, ML itself offers one of the richest application fields for mathematical optimization that challenge available optimization algorithms with new constraints, more data, new objectives, and new processing hardware at the same time. This section gives a short overview of numerical optimization in the context of machine learning and specifically focuses on gradient-based approaches. As discussed later, many ML methods either directly utilize these types of optimization algorithms or can be expressed as specialized instances of them.

Consider the problem of minimizing a smooth and differentiable scalar function $f\colon \mathbb{R}^K \to \mathbb{R}$ over its input space $\mathbb{R}^K$:

$$x^* = \arg\min_{x \in \mathbb{R}^K} f(x) \tag{2.5}$$

One possible approach to tackle this problem is to use an iterative algorithm that moves the current solution towards a smaller value in each step. More formally, let t be the iteration counter and x_t be the current solution. Further, let $\alpha_t \in \mathbb{R}$ be the step size and let $g\colon \mathbb{R}^K \to \mathbb{R}^K$ be a direction depending on the current solution. Starting from an initial solution at $t = 0$ a small update is performed in each iteration:

$$x_{t+1} \leftarrow x_t - \alpha_t g(x_t) \tag{2.6}$$

Three key questions arise here: (1) What is a good initial solution x_0? (2) How should the step size α_t look like? (3) How should the direction $g(x_t)$ look like?

A good initial solution is problem-dependent, and it is difficult to give a general recommendation here. Most approaches start with a (sensible) random solution, e.g., by drawing initial solutions from a Gaussian distribution, but more evolved initialization schemes can be used if there is more information about f available. The second and third questions are often tackled simultaneously: Some directions are more costly to compute but justify using a large step size thereby leading to a larger minimization of the function value in each iteration, whereas other directions are simpler to compute but require smaller steps and more iterations. Gradient-based optimization algorithms use the gradient

$$\nabla_x f(x) = \left(\frac{\partial f(x)}{\partial x_1}, \ldots, \frac{\partial f(x)}{\partial x_K} \right) \tag{2.7}$$

as a basis to compute the direction. Relevant to this thesis are two main variations of gradient-based optimization:

(Stochastic) Gradient Descent

Gradient descent (GD) and Stochastic Gradient descent (SGD) directly implement Eq. 2.6 into an algorithm. The main difference between both approaches is the computation of the direction. Gradient descent utilizes the entire gradient:

$$g(x_t) = \frac{1}{\|\nabla_{x_t} f(x_t)\|} \nabla_{x_t} f(x_t) \tag{2.8}$$

GD is simple to implement and offers linear convergence speed. More formally, let f be a convex function with the Lipschitz constant of L. Further, choose $\alpha_t \leq \frac{1}{L}$, then GD converges in $\mathcal{O}\left(\frac{1}{T}\right)$ where T is the total number of iterations (see [BV14] for a general introduction into the subject and [Gow19] for a more compact overview of proofs.). Similar results for e.g. non-convex functions are also available [LSJR16]. GD offers a simple framework with fast minimization but requires the computation of the full gradient in each iteration. As discussed later, we often face losses that are a mean of functions $f_1, \ldots, f_N$:

$$f(x_t) = \frac{1}{N} \sum_{i=1}^{N} f_i(x_t)$$

Here, the computation of the full gradient is costly for large N, and hence stochastic gradient descent (SGD) is a viable alternative. SGD utilizes an unbiased estimator of the gradient in each iteration instead of the full gradient. Mini-batch SGD is arguably the most widespread variant of SGD. It only considers a small batch of $B \ll N$ summands for the gradient computation:

$$g(x_t) = \frac{1}{B} \sum_{i \in \mathcal{B}} \nabla_{x_t} f_i(x_t) \tag{2.9}$$

where $\mathcal{B}$ is a random batch of summands. This way, SGD requires $\mathcal{O}(BK)$ instead of $\mathcal{O}(NK)$ to estimate the direction. The updates during SGD are noisier because an unlucky sample $\mathcal{B}$ can potentially increase the function value. However, the much faster runtime justifies more iterations with smaller step sizes so that some unlucky samples do not dominate the optimization. In the context of machine learning, SGD is usually run for a set of epochs. Here, the training data is shuffled at the beginning of each epoch and then split into $\lceil \frac{B}{N} \rceil$ consecutive batches that are used for the gradient estimation. One epoch denotes one linear scan over all batches so that the ML model receives the entire dataset once per epoch. There are numerous extensions of SGD available that adapt the computation of g and α, e.g., by incorporating a momentum or averaging [Pol64, Qia99]. In the context of deep learning, even more evolved variations have been presented, e.g., by tuning the step size for each coordinate x_i individually (see [Rud16] for an overview). Depending on the specific step-size policy the convergence speed of SGD is between $\mathcal{O}\left(\frac{1}{T}\right)$ and $\mathcal{O}\left(\frac{1}{\sqrt{T}}\right)$ for strongly-convex functions [NJLS09] which makes it generally slower than GD. However, due to the mini-batching, each iteration of SGD is much faster, requires less memory, and its implementation can leverage certain hardware features such as parallel processing in GPUs. Therefore, SGD and GD have similar convergence speeds in practice, and SGD is the de-facto standard algorithm in deep learning. A more evolved discussion on the trade-offs between GD and SGD can be found in [BCN16].

Coordinate Descent

SGD is ideally suited if the computation of the entire gradient is costly because it essentially only considers $B \ll N$ parts of the loss in each iteration. However, SGD still requires the computation of the entire gradient wrt. to all coordinates of the objective, which can become costly if the number of coordinates K is large. Coordinate descent (CD) specifically tackles this problem by considering only a single coordinate in each iteration while ignoring the others. More specifically, it uniformly samples a coordinate $i \sim \mathcal{U}(1, \ldots, K)$ in each iteration and then uses

$$g(x_t) = \vec{e}_i \frac{\partial f(x_t)}{\partial x_{t,i}} = \left(0, \ldots, 0, \frac{\partial f(x_t)}{\partial x_{t,i}}, 0, \ldots, 0\right) \tag{2.10}$$

where $\vec{e}_i = (0, \ldots, 0, 1, 0, \ldots, 0)$ is the unit vector with a '1' at coordinate i.

CD does not perform optimally compared to ordinary gradient descent because x is not moved in the optimal direction in each step. Thus, it requires more iterations compared to gradient descent but may gain overall speed due to the cheaper gradient computation. To increase convergence speed, note that CD chooses coordinates randomly without considering any information available about each coordinate. In some cases, it is possible to cheaply calculate the direction with the largest (absolute)

gradient value $\left|\frac{\partial f(x_t)}{\partial x_{t,i}}\right|$ that offers the steepest minimization. This selection rule is sometimes called Gauß-Southwell (GS) update[1]. Similarly, one may choose α_t in every iteration optimally, e.g., by a line search if this is cheaply available. Algorithm 1 depicts the resulting CD algorithm with GS rule (CD-GS). Note that for presentational purposes, this algorithm does not use the negative gradient[2].

Algorithm 1 Coordinate descent with Gauß-Southwell update (CD-GS).

```
1: function MINIMIZE_CD(f)
2:     x_0 ← init()                                        ▷ Random start solution
3:     for t = 1, ..., T do                                ▷ Run algorithm for T iterations
4:         i ← argmax_{i∈{1,...,K}} |∂f(x_t)/∂x_{t,i}|     ▷ Gauß-Southwell selection
5:         α_t ← argmin_{α∈ℝ} f(x_t + α e_i ∂f(x_t)/∂x_{t,i})   ▷ Linesearch
6:         x_(t+1) ← x_t + α_t e_i ∂f(x_t)/∂x_{t,i}        ▷ Apply gradient step in dimension i
       return x_T
```

In general, CD has an expected linear convergence rate $\mathcal{O}\left(\frac{K}{T}\right)$ which matches the convergence rate of GD given that each iteration of CD is K times cheaper (see e.g. [Wri15] for convex function and [PN15] for non-convex functions). However, it is important to note that due to the randomness involved in every iteration, the convergence analysis of CD is usually in expectation. Hence, it has an *expected* linear convergence rate. Interestingly, CD-GS has a deterministic linear convergence rate which is not necessarily faster than CD with a uniform sampling of the coordinates, but ultimately a stronger statement about the algorithm. Last, we note that Nutini et al. observed in [NSL+15] that the convergence speed of CD-GS can be substantially improved compared to CD if strong convexity is measured in terms of L_1 norm instead of L_2 norm.

Proximal (Stochastic) Gradient Descent

While CD and SGD are ideally suited for large-scale optimization problems, they both assume that some gradient information is available, which implies that the loss function f is continuously differentiable. This can become difficult for constraint optimization problems that frequently arise in machine learning. Constraint optimization problems introduce further constraints into the objective, e.g., in the form of a regularizer $R\colon \mathbb{R}^K \to \mathbb{R}$ that punishes undesired solutions. Formally, let $\lambda \in \mathbb{R}$ be the regularization strength, let R be the (potentially non-differentiable and non-smooth) regularize, and f be the differentiable objective, then the regularized objective is:

$$x^* = \arg\min_{x \in \mathbb{R}^K} f(x) + \lambda R(x) \tag{2.11}$$

A popular choice to enable the integration of constraints into gradient-based optimization is to use proximal gradient descent. Proximal gradient descent (PGD) is

[1] It is difficult to pinpoint the exact work which coined this term. Recent work often cites [FW60], but[For55] already states that this method was introduced first by Gauss in [Gau23] and then (re-)discovered by Southwell in [Sou46]. Unfortunately, we could not find any direct resemblance of the GS update rule or coordinate descent in [Gau23] due to its informal nature.

[2] The minimization in line(4) choose the appropriate sign for the overall minimization of f.

also an iterative algorithm that uses estimations of the gradient $g_f(x_t)$ of the differentiable part of the objective f to perform small updates on the current solution x_t. Contrary to SGD and CD, however, proximal gradient descent performs an additional projection step after the gradient update to account for the non-differentiable regularizer R. More formally, PGD first performs a regular gradient step that only considers the differentiable part of the objective and then corrects this update for the non-differentiable part of the objective using the so-called `prox`-operator $\mathcal{P}_R$:

$$x_{t+1} \leftarrow \mathcal{P}_R\left(x_t - \alpha_t g_f(x_t)\right) \tag{2.12}$$

where $\alpha_t \in \mathbb{R}_+$ is the step-size as usual. The `prox`-operator is defined as follows:

$$\mathcal{P}_R(x) = \underset{z \in \mathbb{R}^K}{\arg\min}\, R(z) + \frac{1}{2}\|z - x\|_2^2 \tag{2.13}$$

The intuition of the `prox`-operator is that it projects the current solution x to the nearest solution z (in terms of L_2 norm) that is the minimum of the regularizer, thereby retaining a good solution wrt. to f while also respecting the constraints R. The `prox`-operator introduces another optimization problem in itself which can be difficult to solve. Luckily, for many practically relevant constraints, a closed-form solution of Eq. 2.13 can be given. This makes PGD a very fast and practical algorithm. Similar to vanilla gradient descent, the 'regular' proximal gradient descent algorithm uses the entire gradient for the differentiable part of the objective as done in Eq. 2.8 and stochastic proximal gradient descent (PSGD) uses an estimation of the gradient, e.g., on mini-batches as done in Eq. 2.9. The convergence speed of PGD is $\mathcal{O}\left(\frac{1}{T}\right)$ if ∇f is Lipschitz continuous [CP11] and the convergence speed of PSGD is $\mathcal{O}\left(\frac{1}{T}\right)$ in expectation [RVV20]. A deeper discussion on proximal algorithms can be found in [PB14].

2.3 Machine Learning

In computer science, one usually solves problems by the following approach: First, the problem is formalized mathematically. Second, an algorithm is invented that solves the abstract mathematical problem. Third, the abstract algorithm is implemented on specific hardware, and finally, the implementation is applied to the specific problem. Machine learning follows the same principle but deals with problems that a more difficult to formalize well. For example, routing queries can be formalized easily by representing a map via a graph and by defining some key characteristics of a 'good' route, such as the overall length and time traveled. On the other hand, the decision that there might be a cat or a dog in a given image is much more difficult to capture mathematically. However, it is fairly easy for humans to annotate images with the label 'cat' or 'dog'. The general idea of machine learning is to present such annotated datasets to the algorithm and let it figure out the mathematical connections between the presented data (e.g., images) and the provided labels (e.g., cats or dogs) without explicitly describing them. In this thesis, we consider a supervised learning problem defined in Definition 1 and specifically focus on classification problems.

Definition 1 (Supervised Learning)**.** *Let $\mathcal{D} = \mathcal{X} \times \mathcal{Y}$ be a distribution over the input space $\mathcal{X}$ and labels $\mathcal{Y}$. Let $\mathcal{S} = \{(x_i, y_i) | i = 1, \ldots, N\}$ be a labelled sample of i.i.d samples from $\mathcal{D}$ where $x_i \in \mathcal{X} \subseteq \mathbb{R}^d$ is a d-dimensional feature-vector and $y_i \in \mathcal{Y}$ is the corresponding target vector. The goal of supervised learning is to find a model $f\colon \mathcal{X} \to \mathcal{Y}$ that offers good predictions for any $x \in \mathcal{X}$ using the sample $\mathcal{S}$. For $\mathcal{Y} = \mathbb{R}$, this is called*

a regression problem. For $\mathcal{Y} = \{0, 1, \ldots, C-1\}$ with $C \in \mathbb{N}$ this is a (multiclass) classification problem with C classes.

Table 2.2 shows a representation of the data as a table. Depending on the context, we either write $\mathcal{S}$ to denote the training data or write $X \in \mathbb{R}^{N \times d}$ to denote the data matrix and write $y \in \mathcal{Y}^N$ to denote the corresponding target vector. In addition to the label $y_i \in \mathcal{Y}$ it is sometimes more convenient to use the one-hot encoding of each label $\vec{y}_i = (0, \ldots, 0, 1, 0, \ldots, 0)$ that contains a '1' at coordinate c for label $c \in \{0, \ldots, C-1\}$. To keep the notation light, the vector arrow is dropped for the rest of this thesis, and it should be clear from the context which writing is used. Last, depending on the context, we do not consider classes directly but focus on class scores or class probabilities. More formally, we consider the model $f \colon \mathcal{X} \to \mathbb{R}^C$ and use

$$\widehat{y} = \underset{c=0,\ldots,C-1}{\arg\max}\; f(x)_c \tag{2.14}$$

for predicting the specific class. Again, it should be clear from the context if classes are predicted by $f(x)$ directly or if $f(x)$ is a score vector.

TABLE 2.2: Illustration of a dataset as a table with corresponding label vector.

$\mathcal{S}$	Feature 1	Feature 2	...	Feature d	Label
Example 1	x_{11}	x_{12}	...	x_{1d}	y_1
Example 2	x_{21}	x_{22}	...	x_{2d}	y_2
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
Example N	x_{N1}	x_{N1}	...	x_{Nd}	y_N

Our goal is to learn a function mapping $f \colon \mathcal{X} \to \mathcal{Y}$ (or $f \colon \mathcal{X} \to \mathbb{R}^C$) which fits the training data well and generalizes to new, unseen data. Empirical Risk Minimization (ERM) proposes to minimize a loss function $\ell \colon \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$ (or $\ell \colon \mathbb{R}^C \times \mathcal{Y} \to \mathbb{R}$ depending on f) which compares the prediction $f(x)$ and the training label y. For clarity, let $\widehat{L}$ denote the empirical loss given the sample $\mathcal{S}$ and L denote the loss wrt. to the unknown distribution $\mathcal{D}$:

$$L_{\mathcal{D}}(f) = \mathbb{E}_{(x,y)\sim\mathcal{D}}\left[\ell(f(x), y)\right] \tag{2.15}$$

$$\widehat{L}(f) = L_{\mathcal{S}}(f) = \mathbb{E}_{(x,y)\sim\mathcal{S}}\left[\ell(f(x), y)\right] = \frac{1}{N} \sum_{(x,y)\in\mathcal{S}} \ell(f(x), y) \tag{2.16}$$

By the law of large numbers [DKLM05], it holds that $\widehat{L}$ approaches L for $N \to \infty$. Hence, minimizing $\widehat{L}$ will also likely minimize L given N is large enough. However, in practical applications, it can often be observed that there is a gap between the empirical loss during training and the empirical loss during the deployment of the model. There are two reasons for this: First, there might be a distributional mismatch between the training and the testing data, e.g., due to the inherent randomness of the application, a concept drift between the gathering of the data and the deployment of the model, or simply because not enough data was available during training that accurately describes the true distribution. Second, and more interesting from a model's perspective, the model might overfit the training data. Overfitting occurs when the model learns individual patterns in the training data which are not present in the true distribution of the data. In this case, the model performs extremely well

on the training data but usually generalizes poorly to unknown data. Arguably the most extreme form of overfitting occurs when the training data is simply memorized without extracting any meaningful patterns from the data.

To minimize the gap between L and $\widehat{L}$, the Empirical Risk Minimization (ERM) principle proposes to minimize the empirical loss in combination with a regularizer $R\colon \mathcal{F} \to \mathbb{R}$ where $\mathcal{F} = \{f : \mathcal{X} \to \mathcal{Y}\}$ is a set of functions and $\lambda \in \mathbb{R}$ is the regularization strength:

$$f^* = \arg\min_{f \in \mathcal{F}} \frac{1}{N} \sum_{(x,y)\in\mathcal{S}} \ell(f(x), y) + \lambda R(f) \tag{2.17}$$

Here, the regularizer can incorporate prior knowledge about the data into the learning process as well as punish models which simply memorize the training data without extracting meaningful patterns. Moreover, it can also help us to choose models which are more suitable for resource-constraint devices, e.g., by constraining the search space to hardware-friendly functions.

Once a model has been obtained, we are naturally interested in its performance. As ERM suggests, when $\mathcal{S}$ is large enough then the empirical performance will be close to its actual performance on the distribution $\mathcal{D}$. However, in most practical applications, $\mathcal{S}$ is too small to confidently make this conclusion. Hence, a different approach must be followed. Arguably the most direct way to assess the performance of a model is to test it on some test data. This test should be different from the training data – otherwise, we would be testing the memorization capabilities of the model which defies our goal to extract general patterns from the data. Hence, we split the sample $\mathcal{S}$ into a training and testing sample $\mathcal{S} = \mathcal{S}_{train} \cup \mathcal{S}_{test}$ so that $\mathcal{S}_{train} \cap \mathcal{S}_{test} = \emptyset$. We use the training sample $\mathcal{S}_{train}$ to train the model and $\mathcal{S}_{test}$ to test it. Again we are faced with the question of how large these samples should be. For model training, having more data is usually beneficial as it gives the algorithms a better sample of the original distribution $\mathcal{D}$. So the question becomes: How many samples are required for testing so that the empirical loss $L_{\mathcal{S}_{test}}$ is close to the true loss $L_{\mathcal{D}}$?

We use a tail-bound to quantify the distance between the empirical loss and the true loss. Since we do not have any assumptions on the true distribution $\mathcal{D}$, the Hoeffding Bound is an ideal candidate as it only assumes independent observations in $\mathcal{S}_{test}$.

Theorem 1 (Hoeffding's inequality [Hoe63])**.** *Let $X_1, \ldots, X_N$ be i.i.d random variables with $X_i \in [a_i, b_i]$. Let $\mu = \frac{1}{N}\sum_{i=1}^{N} X_i$ be the empirical mean and let $\varepsilon \in \mathbb{R}_+$, then the following holds for any N:*

$$P\left(|\mu - \mathbb{E}[X]| \geq \varepsilon\right) \leq 2\exp\left(-\frac{2N\varepsilon^2}{\sum_{i=1}^{N}(b_i - a_i)^2}\right)$$

This theorem can be useful to compute the number of data points required to reach a certain confidence in the computation of the empirical loss. To do so, let $\beta \in [0, 1]$ be the desired upper bound on the error probability:

$$P\left(|\mu - \mathbb{E}[X]| \geq \varepsilon\right) \leq 2\exp\left(-\frac{2N\varepsilon^2}{\sum_{i=1}^{N}(b_i - a_i)^2}\right) \leq \beta \tag{2.18}$$

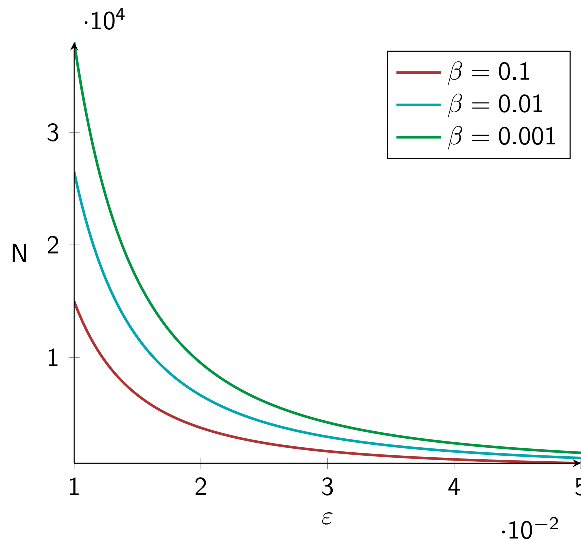


FIGURE 2.6: The number of data points N over the maximum absolute deviation ε according to Hoeffdings bound for different β values.

Now solving for N yields:

$$\begin{aligned} 2\exp\left(-\frac{2N\varepsilon^2}{\sum_{i=1}^N (b_i - a_i)^2}\right) &\leq \beta \\ -\frac{2N\varepsilon^2}{\sum_{i=1}^N (b_i - a_i)^2} &\leq \log\left(\frac{\beta}{2}\right) \\ \frac{2N\varepsilon^2}{\sum_{i=1}^N (b_i - a_i)^2} &\geq -\log\left(\frac{\beta}{2}\right) \\ N &\geq -\frac{1}{2}\log\left(\frac{\beta}{2}\right)\frac{\sum_{i=1}^N (b_i - a_i)^2}{\varepsilon^2} \end{aligned}$$

We can apply this theorem to the loss function. For simplicity assume that we attain losses $\ell(f(x), y) \in [0, 1]$ so that $b_i - a_i = 1$. It follows that we need at least

$$N \geq -\frac{1}{2\varepsilon^2}\log\left(\frac{\beta}{2}\right) \tag{2.19}$$

samples to guarantee that the probability that the empirical loss and true loss will deviate more than ε is bounded by β. Figure 2.6 gives an example of different β values for this bound. As one can see, the smaller ε, the higher N must be chosen. Similar, smaller β values also require larger N.

While the Hoeffding inequality helps us to determine how many samples we require to make an accurate assessment of the model's quality, it is comparably loose, and in many scenarios, there is not enough test data available to fully trust the bound. In addition, we might not be interested in the performance of a specific model trained on a specific dataset, but we are more concerned with the performance of the algorithm and the hyperparameters that produced this model. In this case, the following approaches are used to rate the performance of an algorithm on a specific data set:

- **Test/Train split:** If the entire dataset is large enough so that both the testing

and the training data are large enough, then a single test/train split may suffice to accurately gauge an algorithm's performance. While this is the quickest method to estimate the performance of an algorithm, a single split may lead to over- or underestimation of the true performance due to 'lucky' or 'unlucky' splits that favor specific algorithms.

- **Leave-One-Out:** Instead of performing a single test/train split, leave-one-out performs N splits. In each split, a single item is chosen for the test set, whereas the remaining $N-1$ observations are used for training. This estimation is by far the most computationally intensive estimation because N models have to be trained. However, it also allows using $N-1$ points for training and N points for testing, which makes it ideally suited when only limited data is available.

- **Cross-Validation:** A k-fold cross-validation splits the data into k similar-sized buckets. In each validation round, a model is trained on all but one bucket and tested on the remaining unused bucket. This process is repeated k times so that each bucket is used for testing exactly once. Usually, $k = 5$ or $k = 10$ is used, which makes cross-validation a computationally feasible alternative to the leave-one-out estimation without the risk of under- or overestimation of an algorithm's performance.

Caution has to be exercised when tuning the hyperparameters of an algorithm before deploying it. While cross-validation and leave-one-out minimize the effects of under- and overestimating an algorithm's performance, there is still the chance for overfitting, especially when hyperparameters are fine-tuned for the best performance. Ultimately, the training and testing of an algorithm and *all* its hyperparameters is based on the sample $\mathcal{S}$. Hence, when optimizing for the best hyperparameters, this process inevitably leads to overfitting on that sample. To circumvent this problem, a third, unrelated validation set $\mathcal{S}_{val}$ can be used to estimate the performance of the final model *after* the hyperparameter optimization is complete, and the final model has been trained[3]. However, once this validation set has been used to estimate the performance, there should not be another iteration to fine-tune hyperparameters unless the dataset or task changed, as this would again lead to overfitting.

2.4 Discrete Classifiers

Discrete classifiers are machine learning models that can be executed *without* any floating-point operations. Arguably the simplest approach to finding a discrete classifier is to train a regular model and then perform a post-training quantization that converts all of its floating-point parameters to a fixed-point representation. This approach often leads to a loss in performance that can sometimes be bounded theoretically (see e.g. [SKS17, SPZ+17]), but a more general theory is yet to be discovered. Fixed-point quantization in machine learning has become more popular due to deep learning (see e.g. [Guo18, GKD+21] for an overview), but is not limited to neural networks. For example, Papadonikolakis and Bouganis show how to implement the SVM using fixed-point quantization in [PB08a, PB08b] and Al-Zoubi et al. repeat a similar experiment for K-NN in [AZTK18].

The second approach to finding a discrete model that circumvents the performance loss of post-quantization is to directly compute a discrete solution. There is a

[3] Depending on the literature the terms 'validation set' and 'test set' are switched. In this thesis, we will use the terminology as presented.

considerable amount of research in the deep learning literature available that justifies a thesis in its own right. Therefore, we refer interested readers to the first two major papers on the topic [GAGN15, JKC+18] and recommend the survey by Gholami et al. for a more recent overview [GKD+21]. Going beyond deep learning there are quantization efforts for many other ML algorithms. For example, Anguita et al. carefully change the minimization problem of the SVM in [AGPR07] so that its minimum is guaranteed to be representable via fixed-point arithmetic's and Mariet and Sra give fixed-point algorithms for Determinantal Point Processes in [MS15].

The third branch of discrete classifiers – and the one we will pursue in this thesis – are classifiers that are discrete *by design*. For example, Mücke et al. present in [MPM19] an SVM implementation that guarantees to find a binary solution in any case. Similarly, Piatkowski et al. introduce integer undirected graphical models in [PLM16] that only require integer operations for training and sampling. The advantages of these models are clear: There is no need for post-training quantization and hence there is no performance loss. Similarly, there is no need to change the training algorithm as the models are already discrete making theoretical results readily available in practice. Moreover, most of the corresponding learning algorithms also only make limited use of floating-point operations leading to efficient training as well. Last and maybe most important for this thesis, they can be combined into ensembles to achieve state-of-the-art performance. In the following, we focus on three discrete classifiers: Decision tree, binarized neural networks, and naive Bayes.

2.4.1 Decision Trees

Decision trees (DT) structure knowledge into different hierarchical levels, which are connected by branches much like trees in the real world. They offer a very intuitive way of organizing knowledge that seems to transcend human history across all historical ages [Lim14]. Somewhat surprisingly, DTs in machine learning is a relatively new method that was first presented in the mid-1980s [BFSO, Qui86]. They quickly became one of the most used base-leaner for ensembling up to the point that they are intertwined and indistinguishable from some ensembling methods themselves.

There are many DT variations available, and hence we will now present a common framework that describes DT classifiers in a very general form and offer a format with three parameters: The class of leaf nodes that is used for predictions, the class of split functions that is used for branching and the number of branches K. This framework encapsulates many tree-structured classifiers, such as regular axis-aligned decision trees [BFSO, Qui86], logistic model trees [LHF05] or Gaussian summary trees [BLM19] that are trained in a top-down greedy fashion, but also encapsulates more recent DT variations that are trained via gradient-based algorithms [KFCB15, ST15, AİA18, SGW+18, IA21] as well as racing-based online algorithms [DH00, HSD01, HKP05, PHK07, PHK08, BG09, RPDJ13, MHM21, MWS18].

Formally, a DT partitions the observation space $\mathcal{X}$ into increasingly smaller regions and uses independent predictions for each region. A tree is represented as a directed graph with a root node where each node has up to K child nodes. Each node in the tree belongs to a subregion $\mathcal{I} \subseteq \mathcal{X}$, and all children of each node recursively partition the region of their parent node into K non-overlapping smaller regions. Each node is associated with K split functions $s\colon \mathcal{X} \to \{0,1\}$ that are '1' if x belongs to the corresponding subregion of that node and '0' if not. Per construction, subregions of a node are non-overlapping so that exactly one split evaluates to '1' at a time for a given observation. The splitting is repeated recursively until a stop criterion, e.g. a maximum number of nodes, is reached. Then a prediction function $g\colon \mathcal{X} \to \mathcal{Y}$ is

trained on all the data points in the specific region of the leaf node. Let L be the total number of leaf nodes in the tree and let $L_i = (n_1, n_2, \dots)$ be the nodes visited on the path from the root node to leaf i, then the prediction of a tree is given by:

$$h(x) = \sum_{i=1}^{L} g_i(x) \prod_{l \in L_i} s_l(x) = \sum_{i=1}^{L} g_i(x)\pi_i(x) \tag{2.20}$$

Top-Down DT induction

Arguably the most common way to train DTs are top-down greedy algorithms that start with the entire dataset at the root node, select the split function by minimizing a loss function, and then repeat this process until a stopping criterion is met. In a sense, these algorithms are greedy because they choose the best split for the given subset of data, not taking subsequent splits into account. The earliest works on such algorithms for regression problems are due to Morgan and Sonquist in [MS63], which present the basic algorithm that is still used today. The general idea is as follows: We start at the root node and the entire dataset $\mathcal{S}$. Let

$$\mathcal{S}_i = \{(x, y) \in \mathcal{S} | s_i(x) = 1\}$$

be the set of training examples associated with split s_i. Then, to choose the optimal split, we minimize a loss (or sometimes called impurity) so that

$$s, g = \underset{\substack{g_1, \dots, g_K \in \mathcal{G} \\ \cup_{i=1}^{K} \mathcal{S}_i = \mathcal{S} \\ \cap_{i=1}^{K} \mathcal{S}_i = \emptyset}}{\arg\min} \sum_{j=1}^{K} \sum_{(x,y) \in \mathcal{S}_j} \ell(g_j(x), y) = \underset{j=1,\dots,K}{\arg\min} \sum_{j=1}^{K} e(j) \tag{2.21}$$

where $s = \{s_1, \dots, s_K\}$ are called the splits and $g = \{g_1, \dots, g_K\} \subseteq \mathcal{G}$ are called the leaves that are obtained from a set of possible predictors $\mathcal{G} = \{g \colon \mathcal{X} \to \mathcal{Y}\}$. Once the optimal splits are found, the data is divided into its subsets $\mathcal{S}_i$, and the splitting continues recursively on each new sub-set until a stopping criterion is reached. Algorithm 2 summarizes this approach. It has two main components: It selects the appropriate splits alongside the predictions for the newly introduced leaves. Then it sorts the data accordingly into K regions. This training method is recursively repeated and stops once a stop criterion is reached.

The specific runtime of algorithm 2 depends on the stopping criteria, the time required to select s, g, and the overall height of the tree. Consider the case in which the data is split until there is only one example left per leaf node. Let C be the cost of choosing the splits s and prediction functions g; let there be N training points at the current node. In the worst case, we will assign $N - K + 1$ points to a single node and distribute the remaining $K - 1$ examples evenly. In this case, the tree degenerates to a decision list with height $\mathcal{O}\left(\frac{N}{K}\right)$. At each level, we train K models and sort the entire data set into the corresponding leaf nodes leading to a total runtime of $\mathcal{O}\left(K\frac{N}{K}C + N\right) = \mathcal{O}\left((N+1)C\right)$. Similarly, if we consider the average case in which training data is evenly distributed among the K regions, the expected tree height is $\mathcal{O}\left(\log_K N\right)$ giving a total runtime of $\mathcal{O}\left(K \log_K(N)C + N\right)$. Last, note that the recursive `TRAIN_TREE` call (line 13) can be parallelized easily without any locking. However, such parallelization has to be implemented with caution. First, simply spawning new threads for each recursive call leads to up to K new threads per child node, quickly leading to an uncontrollable amount of threads competing for CPUs and

Algorithm 2 Training of a decision tree.

```
 1: function TRAIN_TREE(S)
 2:     node ← init()                                  ▷ Start a new node
 3:     s, g ← arg min_{j=1,...,K} Σ_{j=1}^{K} e(j)    ▷ Solve Eq. 2.21
 4:     node.s ← s
 5:     node.g ← g
 6:     if STOP then                                   ▷ Check stopping criterion
 7:         node.children ← null                       ▷ Ignore potential splits
 8:         node.leaf ← true                           ▷ Make this node a leaf
 9:     else
10:         for i = 1, ..., K do                       ▷ Split the data into sub-regions
11:             S_i ← {(x, y) ∈ S | s_i(x) = 1}
12:         for i = 1, ..., K do                       ▷ Train children on the sub-regions
13:             node.children[i] ← TRAIN_TREE(S_i)
        return node
```

degrading performance. Second, individual branches of the tree can become quite large, and thus recursive function calls may require a large amount of stack memory. Hence, a thread pool with a fixed number of threads seems to be appropriate. Instead of training a new DT node directly, we schedule its training using a job queue. The parent node thus can finish execution immediately and free its stack memory directly after all nodes have been submitted into this queue. Once one thread inside the pool finishes execution, it will fetch the next job in the queue and begin its execution, utilizing the system as much as possible without overloading it[4].

Once a DT has been trained it can be applied to new data. To do so, one starts at the root node and visits that sub-region to which the current observation x belongs to by checking if a split $s_i(x)$ equals one. This process is repeated until there are no more splits (i.e. a leaf node is found) of which then the prediction $g(x)$ is returned. Algorithm 3 summarizes this approach.

Algorithm 3 Application of a decision tree.

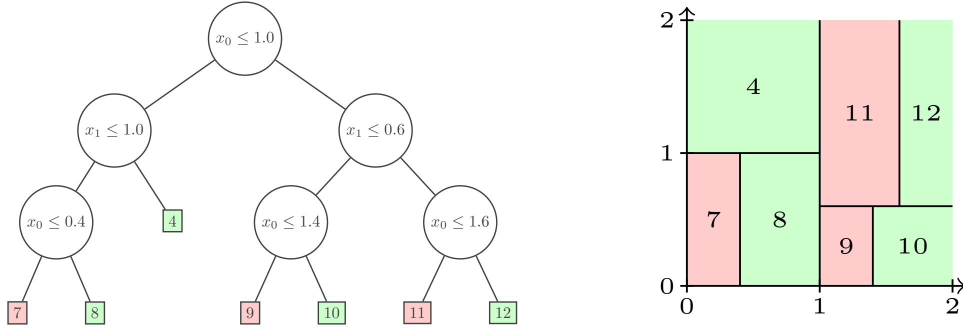
```
1: function APPLY_TREE(x, root)
2:     node ← root                          ▷ Start at the root node
3:     while not node.leaf do               ▷ Repeat until leaf node is found
4:         for i = 1, ..., K do             ▷ Check all splits
5:             if node.s_i(x) = 1 then
6:                 node ← node.children[i]  ▷ Visit child if split is 1
7:                 break
       return node.g(x)                     ▷ Return the prediction of the leaf
```

Axis-Aligned Binary Decision Trees

The original work of Morgan and Sonquist in [MS63] uses a binary DT with $K = 2$ child nodes, and axis-aligned splits $1\{x_k \leq t\}$ where $k \in \mathbb{N}$ is a feature index and $t \in \mathbb{R}$ is a threshold. For convenience, we denote the first child node with index 0 as the 'left' and the second child node with index 1 as the right child. An example of such a tree can be found in Figure 2.7a. It shows a DT that was trained on two

[4] For example, `OpenMP` implements this behavior with the `schedule(dynamic, 1)` pragma.

features, each having values from $[0, 2]$. The inner nodes (denoted by circles) split the feature space into two sub-regions, left and right, whereas the leaf nodes (denoted by rectangles) offer the predictions. The resulting split pattern in the observation space of this tree is shown in Figure 2.7b.



(A) An axis-aligned binary decision tree trained on two features $X_1, X_2 \in [0, 2]$. Round nodes depict the inner decision nodes for the tree, whereas rectangles depict the leaf nodes. Green shows a positive prediction of '+1', and red indicates a negative prediction of '-1'. Numbering is in breadth-first order to distinguish the regions in the figure on the right. Best viewed in color.

(B) The axis-aligned splits induced by the binary decision tree on the left. Each inner node splits the observation space into two sub-regions resulting in a rectangular 'split pattern'. A green rectangle indicates a positive prediction of '+1', and red indicates a negative prediction of '-1'. Numbering is in breadth-first order to distinguish the leaves in the figure on the left. Best viewed in color.

FIGURE 2.7: An example of an axis-aligned binary decision tree.

Somewhat surprisingly, the vast majority of DTs that are used are still of this form, and often times the term 'decision tree' is used interchangeably to denote axis-aligned DTs of this type. However, many improvements have been introduced into this framework over the years that go well beyond the scope of this thesis. In the following, we will summarize some of the most important aspects of training axis-aligned decision trees and refer interested readers to [Kot13, Loh14, BdCF15] for a more detailed overview of individual developments. As already mentioned, axis-aligned DTs utilize the following split function:

$$s(x) = \mathbb{1}\{x_k \leq t\} = \begin{cases} 1 \text{ if } x_k \leq t \\ 0 \text{ else} \end{cases} \tag{2.22}$$

where k denotes a specific feature of the input vector x and t is a precomputed threshold. These splits generate rectangular-shaped regions and thus offer sharp boundaries. For prediction, axis-aligned DTs commonly use the empirical class probabilities in each node:

$$g_j(x) = \left[\frac{1}{|\mathcal{S}_j|} \sum_{(x,y) \in \mathcal{S}_j} y_i \right]_{i=0,\dots,C-1} = p_j \tag{2.23}$$

These predictions are constant wrt. the current sample x and hence we adopt the more commonly used notation $(g_j(x))_i = p_{j,i}$ to denote the probability of class i in node j.

For selecting the best split, the Gini-Score (e.g., used by the CART algorithm [BFSO]) or the Entropy-Score (e.g. implemented in the ID3 algorithm [Qui86]) are

used most commonly. The Gini-Score of the j-th child is defined as

$$\Gamma(p_j) = 1 - \sum_{i=0}^{C} p_{j,i}^2 \tag{2.24}$$

whereas its Entropy (or sometimes called Information Gain) is given by:

$$\Gamma(p_j) = -\sum_{i=0}^{C} p_{j,i} \log_2(p_{j,i}) \tag{2.25}$$

Last, some works use the classification error if only the majority class would be predicted:

$$\Gamma(p_j) = \sum_{\substack{i=1 \\ i \neq \arg\max_k p_{j,k}}}^{C} p_{j,i} \tag{2.26}$$

To encourage the computation of balanced trees in which each node receives roughly the same amount of training samples, the impurities are weighted by their respective sample size $\frac{|S_j|}{|S|}$. Putting this all together leads to the following optimization problem:

$$k, t = \arg\min_{k \in \{0,\dots,d-1\}, t \in \mathbb{R}} \frac{|\mathcal{S}_0|}{|S|}\Gamma(p_0) + \frac{|\mathcal{S}_1|}{|S|}\Gamma(p_1) \tag{2.27}$$

where $\mathcal{S}_0, \mathcal{S}_1$ and $\Gamma(p_0), \Gamma(p_1)$ depend on k, t as shown in Eq. 2.22 and Eq. 2.23. Before discussing how to efficiently select s we want to discuss the choice of the impurity measure briefly. In the light of error minimization, Eq. (2.26) offers a natural interpretation. However, from the earlier works on DT algorithms, authors started to argue against direct error minimization in the light of worse empirical results compared to the Gini-Score or the Entropy-Score. For example, Breiman et al. construct a counter-example in [BFSO] where the classification error does not yield good splits during tree construction, but Entropy-Score does. In more general, Breiman et al. give in [BFSO] a list of desirable attributes which Γ should fulfill. Later, Kearns and Mansour introduced the definition of permissible splitting criterion in [KM96] to reflect these attributes.

Definition 2 (Permissible Split criterion). *A function $\Gamma : [0,1] \to [0,1]$ is called permissible, if the following properties hold*

- Γ *is symmetric about* $1/2$*, that is* $\Gamma(p) = \Gamma(1-p)$
- Γ *is concave*
- Γ *has its maximum at* $\Gamma(1/2)$ *with* $\Gamma(1/2) \geq 1/2$
- Γ *has its minima at* 0 *and* 1*, that is* $\Gamma(0) = \Gamma(1) = 0$

The Gini-Score, the Entropy-Score, and the classification error are permissible split criteria as shown in Figure 2.8. Additionally, note, that every permissible function is an upper bound for the classification error. Thus, by minimizing a permissible function we also minimize the classification error.

Figure 2.8 helps to explain a common argument in favor of the Gini- or Entropy-Score and against direct error minimization. Small changes in the classification error only result in small, linear changes in Γ, which makes progress difficult. On the

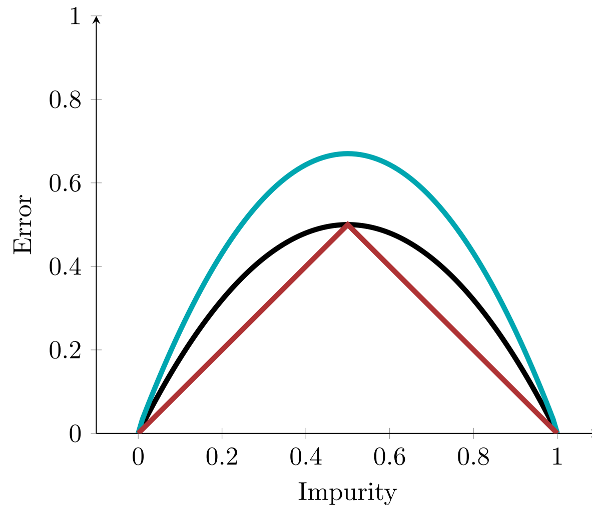


FIGURE 2.8: Comparison between different impurity measures: Gini-Score in black, Entropy in turquoise, and the classification error in red.

other hand, the Gini-Score and Entropy-Score offer steeper, quadratic curves, so that small changes in the error result in greater changes in Γ, which helps minimization. And indeed, Kearns and Mansour were able to prove in [KM96], that the number of nodes required to achieve 0 training error directly depends on the choice of Γ and is minimal if Γ is non-linear. Note, however, that to the best of our knowledge, there does not exist rigor mathematical analysis, nor consistent practical results which link the split criterion to the generalization abilities of a tree. It is merely the case, that the Entropy-Score and the Gini-Score are both the de-facto standard in today's implementations for historical reasons and thus used by most people.

So far, we did not discuss how to find the split threshold t as well as its corresponding feature index k that minimizes Eq. 2.27. Arguably, the simplest approach is to iterate over every possible feature value, record its impurity and later pick that feature/threshold pair that has the smallest impurity. For categorical features with only a fixed number of categories, this approach is straightforward. For real-valued features that can take any value from $\mathbb{R}$, this is not feasible. Looking at the training data, however, we can see that there are at most N different values per feature inside the sample. Assuming that the data is sorted according to feature i and all duplicate values are removed so that $x_{1,i} < x_{2,i} < \dots < x_{j,i} < x_{j+1,i} < \dots < x_{N',i}$, then there are in total $N' - 1$ intervals $[x_{j,i}, x_{j+1,i}]$. Note that all thresholds $t \in (x_{j,i}, x_{j+1,i}]$ from an interval result have the same score since there is no training data in-between. Hence, all splits from the same interval are equally good and *any* threshold from this interval can be chosen. In most implementations, the midpoint of the interval is used

$$t = \frac{x_{j,i} + x_{j+1,i}}{2} \tag{2.28}$$

This choice seems to have a minor positive impact on the performance of the tree, but as far as we know there is no comprehensive study on the effect of choosing this midpoint.

Putting this all together, Figure 4 shows the resulting algorithm. It recursively builds a tree until a STOP criterion is reached. In this case, the recursion is stopped and the class probabilities in the leaf nodes are assigned. If the recursion is not done yet, the algorithm iterates over all features. While doing so, it sorts the observations

in ascending order and simulates the splitting of the training data into the two subsets $\mathcal{S}_0$ and $\mathcal{S}_1$. Recall that for the computation of Γ the class frequencies are required, which can be easily updated instead of re-computing them for every new threshold. Last, it is checked if the current threshold is better than the previous one leading to the selection of the best threshold in line 22. After that, the recursion continues to build the tree.

Algorithm 4 Training of an axis-aligned binary decision tree.

1: **function** TRAIN_DT($\mathcal{S}$)
2: node $\leftarrow$ init()
3: **if** STOP **then** ▷ Check stopping criterion
4: node.g = $\left[\frac{1}{|\mathcal{S}|}\sum_{(x,y)\in\mathcal{S}} y_i\right]_{i=0,\ldots,C-1}$ ▷ Estimate class probabilities
5: node.leaf $\leftarrow$true ▷ Make this node a leaf
6: **else**
7: $k, t, e \leftarrow 0, 0, \infty$ ▷ Init variables
8: **for** $i = 1, \ldots, d$ **do**
9: $\mathcal{S} \leftarrow$ `sort_ascending`(i) ▷ Sort data wrt. to feature i
10: $S_0 \leftarrow \emptyset$ ▷ Left side is empty
11: $S_1 \leftarrow \mathcal{S}$ ▷ Right side has all data points
12: $c_0 \leftarrow (0, \ldots, 0)$ ▷ Class frequencies for left side
13: $c_1 \leftarrow \left[\sum_{(x,y)\in\mathcal{S}} y_i\right]_{i=0,\ldots,C-1}$ ▷ Class frequencies for right side
14: **for** $i = 1, \ldots, N-1$ **do**
15: $\mathcal{S}_0 \leftarrow \mathcal{S}_0 \cup \{x_i\}, \mathcal{S}_1 \leftarrow \mathcal{S}_1 \setminus \{x_i\}$ ▷ Update split
16: $c_{0,y_i} \leftarrow c_{0,y_i} + 1, c_{1,y_i} \leftarrow c_{1,y_i} - 1$ ▷ Update class frequencies
17: **if** $x_{j,i} = x_{j+1,i}$ **then continue**
18: $p_0 \leftarrow \frac{c_0}{\sum_{i=1}^{C} c_{0,i}}, p_1 \leftarrow \frac{c_1}{\sum_{i=1}^{C} c_{1,i}}$ ▷ Compute probabilities
19: $\gamma \leftarrow \frac{|\mathcal{S}_0|}{|\mathcal{S}|}\Gamma(p_0) + \frac{|\mathcal{S}_1|}{|\mathcal{S}|}\Gamma(p_1)$ ▷ Compute impurity
20: **if** $\gamma < e$ **then** ▷ Check for improvement
21: $\tau \leftarrow \frac{x_{j,i}+x_{j+1,i}}{2}$ ▷ Compute midepoint
22: $k, t, e \leftarrow i, \tau, \gamma$ ▷ Update feature / threshold
23: node.$k = k$ ▷ Set feature index
24: node.$t = t$ ▷ Set threshold
25: $\mathcal{S}_0 = \{(x, y) \in \mathcal{S} | x_k \leq t\}$ ▷ Re-compute left split
26: $\mathcal{S}_1 = \{(x, y) \in \mathcal{S} | x_k > t\}$ ▷ Re-compute right split
27: node.children[0] = train($\mathcal{S}_0$) ▷ Train left child
28: node.children[1] = train($\mathcal{S}_1$) ▷ Train right child
return node

The sorting of each feature costs $\mathcal{O}(N \log N)$, and the computation of γ can be performed in $\mathcal{O}(1)$ if the class frequencies are updated. The computation of g in line 4 also takes at most $\mathcal{O}(N)$ so that the cost of choosing a split function and computing g are at most $\mathcal{O}(dN \log N)$. Combining this with our previous discussion leads to the worst-case runtime of $\mathcal{O}(dN^2 \log N)$ and an average-case runtime of $\mathcal{O}(\log_2(N)dN \log N)$ for algorithm 4. The runtime of this approach can be further reduced if sub-optimal splits are considered. A series of similar works [CG16, PGV+18, KMF+17, DMD19] introduce different approaches to estimate the optimal split, thereby improving the overall computational efficiency while also enabling for efficient parallelism, e.g. in the form of GPUs. Maybe the most extreme variant of such an approach is to simply choose a set of splits randomly and then

select the best one from this set. Such an approach often shows competitive performance compared to other approaches when trees are combined in an ensemble [GEW06] and hence is an attractive training method for trees we will re-visit at the end of this chapter. Last, we note, that both, the application and the training of DTs do not require any floating-point operations in some cases: The application of a DT in algorithm 3 simply compares the input features against pre-computed thresholds and returns the corresponding leaf-values. Hence, no floating-point operations are required. The training of DTs requires the computation of the class probabilities and their corresponding scores in lines 18 and 19. Depending on the specific error function, these computations can be performed with integer operations only. Note that for choosing the optimal split, the weighting of each split by $|S|$ is constant and thus does not impact the choice. Therefore, we may omit $|S|$ in the score computation and only scale each impurity by $|S_0|$ and $|S_1|$, respectively, which are integers. Moreover, if Γ is the accuracy score, then we basically need to count the number of miss-classification that is also an integer.

So far we did not discuss when to stop the induction of a DT. In general, stopping the induction process has two main goals. First, it speeds up the tree induction process and second it combats overfitting. Among the most popular stopping criteria are:

1. **Class homogeneity:** All instances in a node have the same class, and thus further splitting is not necessary.
2. **Attribute homogeneity:** All instances in a node have the same attribute values (but not necessarily the same class) so future splitting is impossible.
3. **Maximum tree depth / maximum number of nodes:** A pre-defined depth or a pre-defined maximum number of nodes is reached.
4. **A minimum number of instances:** Each node must contain a minimum number of instances to justify further splitting. If this is not the case, do not split further.
5. **A minimum gain for splitting:** A pre-defined improvement in the split criteria must be achieved when splitting. Otherwise, do not split.

Criteria 1) and 2) are self-explanatory and should be implemented in any case, whereas criteria 3) - 5) require the choice of appropriate thresholds. In many applications, the choice of optimal hyperparameters is difficult and involves more fine-tuning. Hence, the second line of research studies how to prune a trained tree *after* its construction. These approaches first train the entire tree using 1) and 2) as stopping criteria and then prune away (i.e. delete) sub-trees in a bottom-up fashion. There is a vast array of different pruning strategies available (see e.g. [BdCF15] for an overview) of which we quickly survey the three most commonly used pruning methods

1. **Reduced-Error Pruning:** Reduced-Error-Pruning splits the training data into a pruning set and a training set. Once the tree has been trained, the error of each sub-tree is estimated on the pruning set. For each sub-tree, the following constraint is enforced: A sub-tree cannot be pruned if any of its sub-trees yield a lower error on the pruning set, otherwise delete the sub-tree. While this approach is very intuitive, it requires the splitting of the training data into two sets which can be unfavorable for small datasets.

2. **Sample Complexity Pruning:** Similar to Reduced-Error pruning, Sample Complexity Pruning also removes sub-trees that do not yield a lower error. However, instead of using a dedicated pruning set, it estimates the generalization error of a sub-tree via PAC learning theory which will be discussed in more detail in section 5. It states that the generalization error $L_{\mathcal{D}}$ of a tree h that was trained on N samples and that has n nodes in total is, with probability $1-\delta$ [SSBD14]:

$$L_{\mathcal{D}}(h) \leq L_{\mathcal{S}}(h) + \sqrt{\frac{(n+1)\log_2(d \cdot N + 3) + \log(2/\delta)}{2N}}$$

where L_S is the empirical error of the tree. Thus, when adding a new split node (t,k), the decrease in the error $L_{\mathcal{S}}(f \cup \{(t,k)\})$ must justify adding two new leaves $n \to n+2$. Hence, every sub-tree for which

$$L_{\mathcal{S}}(h \cup \{(t,k)\}) + \sqrt{\frac{(n+3)\log_2(d \cdot N+3)+\log(2/\delta)}{2N}} \leq L_{\mathcal{S}}(h) + \sqrt{\frac{(n+1)\log_2(d \cdot N+3)+\log(2/\delta)}{2N}} \quad (2.29)$$

does not hold is removed.

3. **Minimal Cost Complexity Pruning:** Minimal Cost Complexity Pruning is similar to Sample Complexity Pruning but introduces another parameter α to control the trade-off between the empirical error of sub-tree and its complexity. Let $L_{\mathcal{S}}(h)$ be the empirical error of the tree h and let L be the number of leaves in h. The complexity of a sub-tree h' is defined as

$$C(h') = \frac{L_{\mathcal{S}}(h) - L_{\mathcal{S}}(h')}{L-1}. \quad (2.30)$$

Cost complexity pruning prunes away all sub-trees h' with $C(h') \leq C$ where C is a user-defined hyperparameter. Hence, it is similar to sample complexity pruning but offers a more fine-grained control between the empirical error and the complexity through the parameters α and C.

Greedy DTs and Universal Function Approximation

We close our discussion on DTs by pointing out one remarkable fact about the top-down induction of DTs: Intuitively, we can continue splitting the tree until every leaf node contains exactly one observation. In this case, the DT practically memorizes the data while building an efficient retrieval structure during training. In the limit of $N \to \infty$ when enough training data is available this implies that DTs of sufficient size are universal function approximators that can represent *any* (measurable) function.

More formally, in real analysis, it is well-known that one can approximate any measurable function with a series of so-called simple functions. Simple functions are sums of step functions with different scaling constants. Theorem 2 shows how they can approximate any measurable function.

Theorem 2 (The Simple Function Approximation Theorem [RF10, Kle13])**.** *Let $(\Omega, \mathcal{A})$ be a measurable space and let f be a measurable function defined on Ω. Further, let $A_1, \dots, A_n \in \mathcal{A}$ be measurable, pairwise disjunct sets and let $\alpha_1, \dots, \alpha_n \in \mathbb{R}$. Then there exists a series of simple functions with $n \in \mathbb{N}$*

$$\vartheta_n(x) = \sum_{i=1}^{n} \alpha_i \mathbb{1}\{x \in A_i\}$$

such that

- $\vartheta_n(x)$ *converges pointwise to* $f(x)$ *on* Ω
- $|\vartheta_n(x)| \leq |f(x)|$ *for all* $n \in \mathbb{N}$

Consider a DT with constant predictions $g_i(x) = g_i \in \mathbb{R}$ in the leaf nodes, then the prediction of a DT is given by (c.f. Eq. 2.20)

$$h(x) = \sum_{i=1}^{L} g_i \pi_i(x)$$

Hence, DTs with constant predictions in the leaf nodes are by definition simple functions meaning that for $L \to \infty$ they can approximate any measurable function f. Theorem 2 makes it clear, that we can represent any measurable function with a decision tree, but it does not explain how we arrive at such a tree. Moreover, it is unclear if we can consistently arrive at the correct tree given our sample $\mathcal{S}$. The following theorem formally establishes the consistency of DTs.

Theorem 3 (Consistency of DTs). *Let* $f \colon \mathcal{X} \to \mathcal{Y}$ *be a measurable function and let* $\mathcal{S} = \{(x_0, y_0), \ldots, (x_N, y_N)\}$ *be a sample with* $y_i = f(x_i) + \varepsilon_i$ *where* ε_i *are errors with* $\mathbb{E}[\varepsilon] = 0$ *and* $f(x)$ *is the true function value. Let*

$$h_N(x) = \sum_{i=1}^{N} y_i \pi_i(x)$$

be the fully-grown DT that perfectly represents the sample $\mathcal{S}$*. Then it holds that*

$$P\left(\lim_{N\to\infty} \int_{\mathcal{X}} (h_N(x) - f(x))^2 \, dx = 0\right) = 1$$

Proof. First, we note there can be a subset of M points in the sample that are the same (i.e. $x_1 = x_2 = \cdots = x_M$), but do not have the same label, i.e. $y_1 \neq y_2 \neq \cdots \neq y_M$ due to the additive error. For simplicity, let there be $\widetilde{N}$ unique points in $\mathcal{X}$ and let M_i be the number of duplicate points for the i-th sample x_i and let $y_{i,j}$ be the corresponding labels with $j = 1 \ldots, M_i$. In this case, the prediction becomes

$$h_N(x) = \sum_{i=1}^{\widetilde{N}} \sum_{j=1}^{M_i} \frac{1}{M_i} y_{i,j} \pi_i(x) = \sum_{i=1}^{\widetilde{N}} \sum_{j=1}^{M_i} \frac{1}{M_i} (f(x_i) + \varepsilon_{i,j}) \pi_i(x)$$

Now we look at the case in which $N \to \infty$. Since $\widetilde{N}$ is constant this implies that $M_i \to \infty$ for all unique points in $\mathcal{X}$. The prediction for a single point x then becomes:

$$\begin{aligned}
\lim_{N\to\infty} h_N(x) &= \lim_{N\to\infty} \sum_{i=1}^{\widetilde{N}} \sum_{j=1}^{M_i} frac1M_i y_{i,j} \pi_i(x) = \lim_{M_i\to\infty} \sum_{j=1}^{M_i} \frac{1}{M_i} (f(x_i) + \varepsilon_{i,j}) \\
&= \lim_{M_i\to\infty} \sum_{j=1}^{M_i} \frac{1}{M_i} f(x_i) + \sum_{j=1}^{M_i} \frac{1}{M_i} \varepsilon_{i,j} = f(x_i) + \mathbb{E}[\varepsilon] = f(x_i)
\end{aligned}$$

By the definition of the Riemann-Sum, it holds:

$$\lim_{N\to\infty} \int_{\mathcal{X}} (h_N(x) - f(x))^2 \, dx = \lim_{N\to\infty} \lim_{K\to\infty} \sum_{i=1}^{K} \frac{1}{K} (h_N(x_i) - f(x_i))^2$$

The above discussion shows that $h_N(x) = f(x)$ under additive errors and Theorem 2 states the pointwise convergence of $h_N(x_i) \to f(x_i)$ for $N \to \infty$ so that

$$\lim_{N\to\infty} \frac{1}{K} (h_N(x_i) - f(x_i))^2 = 0 \; \forall i = 1, \ldots, K$$

implying that

$$P\left(\lim_{K\to\infty} \sum_{i=1}^{K} \lim_{N\to\infty} \frac{1}{K} (h_N(x_i) - f(x_i))^2 = 0\right) = 1$$

□

The consistency of DTs has been established relatively early on in [BFSO] as well as some later works in [RG11]. Both works show consistency by utilizing a general version of the Glivenko-Cantelli theorem [VC15] that is based on the law of large numbers [DKLM05]. Theorem 3, on the other hand, is only based on arguments from real analysis in combination with the behavior of the top-down greedy algorithm: The property of DTs as universal function approximator does *not* stem from its function class per se, but it is only in combination with greedy top-down learning that grows the tree when a more data is available, i.e. the tree is fully-grown, that we obtain this property. This makes DT learning fundamentally different from other ML methods such as neural networks which are not adaptive. A neural network trained via SGD has a pre-determined structure that does not change during training, whereas DTs trained via greedy top-down algorithms are adaptive to the training data. Hence, most of the effort in training NN is to find a good architecture that fits the training data, whereas most of the effort in training DTs is how to restrict the learning process to not overfit the data. It is important to note, that both arguments only hold for the combination of model class (e.g. DT) and training algorithm (e.g. top-down). It is perfectly reasonable to train DTs using SGD as discussed in chapter 7 as a special type of neural network for which it is much more difficult to prove consistency and universal function approximation. Similarly, we may use more complex splits e.g. by using neural networks in each inner node of the tree which readily 'inherit' universal function approximation and consistency.

As discussed, *any* tree is good enough to represent a function as long as the predictions in the leaf nodes are consistent with the function values. This raises the question if there is a DT that is only as large as necessary, but does not contain any additional leaves or paths? In other words: Is there an optimal DT and if so, can we find it? Early research on this topic showed that finding an optimal axis-aligned binary DT is NP-hard [HR76] and recent research in [OS21] implies various other hardness results of DT learning in the context of parameterized complexity. Nevertheless, due to increases in computation power and a reoccurring interest in the interpretation of ML models, there is a continuous stream of work on finding small, interpretable DTs. Currently, there are competing methods to train optimal decision trees using dynamic programming [ANS20, DS21], Mixed Integer Linear Programming [Ben92, BB96], SAT solvers [BHO09, SS21] or combinations thereof [ZKN21]. Finding smaller DTs is also beneficial for small devices and hence it is natural to ask

whether we may apply these methods in this context. Unfortunately, finding a (near) optimal DT is still computationally very costly to the point that it is nearly impossible on small devices. Second, and arguably more important, single DTs do not seem to have enough predictive power in most practical situations and better results can be achieved with e.g. ensembles of trees. Last, the performance gap between optimal DTs and their heuristically computed siblings seems to be very small [MS95] so that in practice the greedy top-down algorithms dominate due to their fast runtime and near-optimal performance. This fast runtime enables us to train multiple DTs and combine them into an ensemble to improve the performance even further.

2.4.2 Binarized Neural Networks

Neural networks try to mimic the function of the human brain in which a large collection of neurons exchange information through their synapses. Neural networks are a comparably old technique that was first introduced in the early 1940s by McCulloch and Pits in [MP43] that models the mathematical function of a neuron. Their method was later refined by Rosenblatt in [Ros58], but research stagnated in favor of other methods. Later, in the 1980s, multiple researchers rediscovered and refined the backpropagation algorithm, which enables the efficient computation of gradients in a directed acyclic computation graph [Kel60, LC86, RHW86]. Hence, the training of multiple neurons arranged in layers became possible by gradient descent. After an initial burst of research, neural networks became unpopular again due to the excessive amounts of data and computation required to train these models. In the early 2010s, the research on neural networks then spiked again [KSH12]. Due to the availability of General Purpose Graphical Processing Units (GP-GPU) and advances in neural network design, as well as better optimization techniques, it was now possible to train deep neural networks with millions of parameters on large collections of data with manageable effort. Since then, the research interest in these deep neural networks with many layers has skyrocketed, most notably in application areas where the input data is unstructured such as image, speech, video, and text processing. Neither the history nor the current state of the art in deep learning research could be covered adequately in this thesis. Hence, this thesis will focus on resource-friendly variations of deep learning and, more specifically, on binarized neural networks. For a more general overview, including more detailed historical remarks, we refer readers to [GBC16]. Binarized neural networks are neural networks with parameters constraint to $\{-1, +1\}$, which can be executed without any floating-point operations. While these networks are still trained via (stochastic) gradient descent and backpropagation, their structure is more akin to discrete classifiers that count and compare the number of input features against pre-computed thresholds. This discreteness makes their training more difficult, and many well-known techniques from 'regular' neural networks do not translate to BNNs. The next chapter introduces the necessary notation and the basics of training feed-forward multi-layer perceptrons (MLP) and convolutional neural networks. After that, these techniques are discussed in the context of binarized neural networks.

Feed-Forward Neural Networks

Feed-forward neural networks are directed acyclic computation graphs in which the output of a layer forms the input of its subsequent layers. Each layer transforms its inputs by a pre-defined operation usually parameterized with a parameter vector w. Formally, let there be L layers and let f^l denote the operation of the l-th layer with

parameters $W_l \in \mathbb{R}^{p_l}$, then a feed-forward network computes:

$$f_W(x) = f^l_{W_L}(f^{L-1}_{W_{L-1}}(\dots f^1_{W_1}(x)))) = f^l_{W_L} \circ f^{L-1}_{W_{L-1}} \circ \dots \circ f^1_{W_1}(x) \tag{2.31}$$

where $W = (W_1, \dots, W_L) \in \mathbb{R}^p$ is the concatenation of all parameter vectors in the network. Per convention, we allow a layer l to access *all* previously computed outputs $1, \dots, l-1$ so that, e.g., certain layers can be skipped or ignored when desired. Crucial to the performance of a neural network is its architecture (e.g., the number of layers, their different types, etc.) as well as the specific weights in its layers. To obtain the weights, we again minimize a loss:

$$W^* = \arg\min_{W \in \mathbb{R}^p} L_{\mathcal{S}}(f_w) \tag{2.32}$$

The dominant training method for neural networks is stochastic descent. Central in SGD is the computation of the gradient. For convenience, let z^l denote the evaluation of the neural network up to layer l given some input x:

$$z^l = f^l_{W_l} \circ f^{l-1}_{W_{l-1}} \circ \dots \circ f^1_{W_1}(x) \tag{2.33}$$

The jacobian of a function $h \colon \mathbb{R}^p \to \mathbb{R}^m$ with respect to a parameter vector $w \in \mathbb{R}^p$ is given by:

$$J_w(h(x)) = \begin{pmatrix} \frac{\partial h_1(x)}{\partial W_1} & \frac{\partial h_1(x)}{\partial W_2} & \dots & \frac{\partial h_1(x)}{\partial W_p} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial h_m(x)}{\partial W_1} & \frac{\partial h_m(x)}{\partial W_2} & \dots & \frac{\partial h_m(x)}{\partial W_p} \end{pmatrix} \tag{2.34}$$

and the computation of the gradient of $L_{\mathcal{S}}(f_W)$ for the parameters W_l in layer l is given by a repeated application of the chain rule using the jacobian:

$$\begin{aligned} J_{W_l}\left(L_{\mathcal{S}} \circ f^L \circ f^{L-1} \circ \dots \circ f^l(z^{l-1})\right) &= J_{z^L}(L_{\mathcal{S}}(z^L)) J_{w_l}\left(f^L \circ f^{L-1} \circ \dots \circ f^l(z^{l-1})\right) \\ &= J_{z^L}(L_{\mathcal{S}}(z^L)) J_{z^{L-1}}(f^L(z^{L-1})) \cdot \dots \cdot J_{W_l}(f^l(z^{l-1})) \end{aligned} \tag{2.35}$$

There are a few interesting aspects to be noted in this formulation: First, recall that the loss $\ell : \mathbb{R}^d \to \mathbb{R}$ is a scalar function. Hence, the jacobian $J_{z^L}(L_{\mathcal{S}}(z^L))$ is equal to the partial derivative of $L_{\mathcal{S}}$ wrt. to its input z^L:

$$J_{z^L}(L_{\mathcal{S}}(z^L)) = \nabla_{z^L} L_{\mathcal{S}}(z^L) \tag{2.36}$$

Second, the computation of $J_{W_l}(L_{\mathcal{S}})$ depends on the output of the $l-1$ layer and also requires all jacobians from the subsequent layers l to L. The backpropagation algorithm is an efficient implementation of this gradient computation by using dynamic programming. It first computes all intermediate outputs z^l for all layers starting with the first layer in the forward pass. Then it computes the individual jacobians starting with the last layer in the backward pass. Since the error of the l-th layer depends on the errors of the subsequent layer, they are backpropagated through the network, which coins the name. Moreover, the jacobians do not need to be computed explicitly (which would result in a $p \times p$ jacobian), but an efficient implementation of the vector-jacobian product is sufficient to compute the entire backward-pass since $J_{z^l}(L_{\mathcal{S}}(z^l))$ is a vector. Each layer and its jacobian are only evaluated once, which results in a very fast gradient computation. The backpropagation algorithm is not limited to layer-wise networks but can be generalized to compute gradients in any directed

acyclic computational graph resulting in automatic differentiation systems (AD) that can compute the gradients in any DAG. For a more comprehensive introduction to this topic, we refer readers to [BPRS17]. AD is one of the core building blocks of deep learning as it allows one to quickly design and prototype new neural network architectures without manually computing gradients. This makes the design of new architectures much quicker, less error-prone, and enables a more engineering-style approach to neural network learning.

The architectural choices of a neural network impact the model class and can encode assumptions about the data as well as help the overall optimization process. Hence, the architecture has a critical impact on the performance of neural networks, and the optimal architecture must be developed for the current tasks at hand. In the following, we present some common layer types used in neural network design.

Linear Layers

Arguably the simplest layers are linear layers that have already been proposed in the early works on the perceptrons [MP43, Ros58]. Neural networks that only have linear layers are called multi-Layer perceptrons (MLP) or fully connected neural networks (FC). Let $f\colon \mathbb{R}^n \to \mathbb{R}^m$ be a linear function and let $z \in \mathbb{R}^n$ denote the input to a linear layer. Let $w \in \mathbb{R}^{m\times n}$ be its weights and let $b \in \mathbb{R}^m$ be an additional learnable bias so that $W = (w, b)$ are the parameters of this layer. Then the jacobian is given by:

$$
\begin{aligned}
f_{w,b}(z) &= \langle w, z\rangle + b \\
J_w(z) &= z \\
J_b(z) &= 1
\end{aligned}
\tag{2.37}
$$

Convolution Layers

A common operation used in signal processing (e.g., for audio or images) is a convolution that convolutes the input signal with a filter of a given size. Neural networks with convolutional layers are called convolutional neural networks, or sometimes ConvNets for short. The structure of ConvNets is such that they typically repeat multiple convolutions in the first few layers and use a fully connected linear layer in the output layer. ConvNets have first been proposed in [FM82], and its combination with backpropagation was due to [LBD+89].

For concreteness, consider a two-dimensional filter $f\colon \mathbb{R}^{m\times n} \to \mathbb{R}^{m'\times n'}$ in which images are filtered by a $K \times K$ kernel. Let $z \in \mathbb{R}^{m\times n}$ denote the input image to this layer and $w \in \mathbb{R}^{K\times K}$ the parameters of the filter kernel, then the output is given by:

$$
f_{w,b}(z)_{i,j} = \sum_{i'=0}^{K}\sum_{j'=0}^{K} w_{i,j} \cdot z_{i+i',j+j'} + b = w * z + b, \forall i = 1,\dots,m, j = 1,\dots,n \tag{2.38}
$$

where $b \in \mathbb{R}$ is again a learnable bias. This general idea can be expanded to different forms of inputs, e.g., one can reduce the second filter dimension to $K \times 1$ for time-series data and audio signals. Alternatively, one can expand the filter dimension to, e.g., $K \times K \times K$, which also includes a third dimension (e.g., a color channel in images). Moreover, the stride by how much the convolution is moved, as well as the padding of values at the edge of images, can be set according to assumptions about

the data. Formally, a discrete convolution can be expressed via a Toeplitz matrix:

$$T(W) = \begin{bmatrix} w_1 & 0 & \dots & 0 & 0 \\ w_2 & w_1 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & w_{K-1} & w_K \\ 0 & 0 & \dots & 0 & w_K \end{bmatrix} \tag{2.39}$$

With this, convolutional layers can be expressed as a linear layer that encodes a specific structure or assumption about the data into the network via $T(w)$ leading to the following jacobian:

$$\begin{aligned} f_{w,b}(z) &= \langle T(w), z\rangle + b \\ J_w(z) &= \langle T(w)^T, z\rangle \\ J_b(z) &= b \end{aligned} \tag{2.40}$$

where $T(w)^T$ denotes the transpose of $T(w)$. This makes the computation of gradients for convolutional layers particularly appealing because it means applying the convolution a second time with a transposed weight matrix.

Pooling Layer

Pooling layers are special convolutional layers *without* learnable parameters. They are usually used to reduce the size of the intermediate outputs inside the network to make them more manageable. Again, consider for concreteness a two-dimensional input in which images are filtered by a $K \times K$ kernel. Let $z \in \mathbb{R}^{m \times n}$ denote the input image to this layer. Then average pooling is defined as:

$$f(z)_{i,j} = \sum_{i'=0}^{K} \sum_{j'=0}^{K} \frac{1}{K^2} \cdot z_{i+i',j+j'} \tag{2.41}$$

which can be equivalently expressed as a convolution with $w = \left[\frac{1}{K^2}\right]_{i,j}$. Similarly, max-pooling is given by:

$$f(z)_{i,j} = \max\{z_{i+i',j+j'} | \forall i' = 0, \dots, K, j' = 0, \dots K\} \tag{2.42}$$

where the jacobian now contains the appropriate non-zero values during backpropagation.

Batch Normalization

Batch normalization (BN) is a crucial layer to stabilize the training process of deep networks and improve their overall accuracy. BN layers are tightly interconnected with the training process of deep nets and are only meaningful when a network is trained via mini-batch gradient-descent-like algorithms. It has been discovered relatively recently in [IS15]. The main goal of BN is to stabilize the training process by making sure that the outputs of a layer follow the same distribution in each batch during training. For concreteness, consider a 1-dimensional batch normalization $f\colon \mathbb{R}^n \to \mathbb{R}^n$ with learnable parameters $W = (\gamma, \beta), \gamma \in \mathbb{R}^n, \beta \in \mathbb{R}^n$ and a

parameter for numerical stability $\varepsilon \in \mathbb{R}$. Then, BN is defined as:

$$
\begin{aligned}
f_{\gamma,\beta}(z) &= \frac{z - \mathbb{E}_{\mathcal{B}}[z]}{\sqrt{\mathbb{V}_{\mathcal{B}}[z] + \varepsilon}} \odot \gamma + \beta \\
J_{\gamma}(z) &= \frac{z - \mathbb{E}_{\mathcal{B}}[z]}{\sqrt{\mathbb{V}_{\mathcal{B}}[z] + \varepsilon}} \\
J_{\beta}(z) &= 1
\end{aligned}
\tag{2.43}
$$

Here, $\mathbb{E}_{\mathcal{B}}[z]$ denotes the average input over the batch $\mathcal{B}$, and $\mathbb{V}_{\mathcal{B}}[z]$ is its variance. '$\odot$' is the element-wise multiplication so that the original input size does not change. Special care must be taken when the network is applied to a single example. In this case, the normalization becomes meaningless because $\mathbb{V}_{\mathcal{B}}[z]$ cannot be estimated. Hence, during training, a running average for $\mathbb{E}_{\mathcal{B}}[z]$ and $\mathbb{V}_{\mathcal{B}}[z]$ is stored, which can be used during model deployment when only one example is available. Similar to convolutional operations, there are different variants of BN, e.g., for two or three-dimensional inputs.

Skip Connections

Skip connections connect two different intermediate representations from two different layers by summing over them. Hence, a representation can skip certain layers. Skip connections have been part of neural network design for a long time, and it is difficult to pinpoint the first publication that proposed them. In the context of deep learning, they have been popularized by [HZRS16], that also coined the term Residual Networks or ResNets. The intuition behind this approach is that during the backward pass, the errors that are backpropagated can skip layers that would otherwise diminish the numerical value of the gradient[5]. For concreteness, again, consider two layers l and $l' < l$ with their respective outputs $z^l \in \mathbb{R}^n$ and $z^{l'} \in \mathbb{R}^m$, then a skip connection is defined as:

$$
\begin{aligned}
f(z^l, z^{l'}) &= z^l + \phi(z^{l'}) \\
J_{z^l}(z^l, z^{l'}) &= 1 \\
J_{z^{l'}}(z^l, z^{l'}) &= 1
\end{aligned}
\tag{2.44}
$$

where $\phi\colon \mathbb{R}^m \to \mathbb{R}^n$ transforms the input so that a summation is possible. Similar to convolutions, the specific dimensionality of the intermediate representations may vary, but the general approach remains the same. Note that the gradient of z^l and $z^{l'}$ is one so that the entire error of subsequent layers is directly backpropagated during training, essentially skipping the layers in between. Similar to BN, skip connections do not necessarily increase a network's size, but they often help to stabilize the learning process for especially deep networks since gradients from subsequent layers are backpropagated without any changes to them.

Activation Functions

In essence, each layer discussed so far performs a linear transformation of its input, and a neural network with only these layers would only be able to learn a linear function[6]. To solve this problem, non-linearities in the form of activations are introduced into the network. These usually do not have learnable parameters but are merely

[5] This phenomenon is commonly known as vanishing gradients [Hoc98].
[6] A linear combination of linear functions is again a linear function.

used to transform the input between the layers. However, they can have a significant impact on the training and on the final performance of the network. Historically, the sigmoid function

$$f(z)_i = \frac{1}{1 + \exp^{-z_i}}$$
$$J(z)_{i,j} = \begin{cases} f(z)_i(1 - f(z)_i) & \text{if } i = j \\ 0 & \text{else} \end{cases} \tag{2.45}$$

as well as the tanh-function

$$f(z)_i = \tanh(z_i)$$
$$J(z)_{i,j} = \begin{cases} 1 - \tanh^2(z_i) & \text{if } i = j \\ 0 & \text{else} \end{cases} \tag{2.46}$$

have been considered due to their elegant derivatives. However, these functions slow down the convergence in deeper networks because their gradients scale the errors of the subsequent layers by a factor ≤ 1 during backpropagation. Hence, different activation functions, such as Rectified Linear Activation (ReLU), have become popular:

$$f(z)_i = \max(0, z_i)$$
$$J(z)_{i,j} = \begin{cases} 1 & \text{if } i = j, z_i > 0 \\ 0 & \text{else} \end{cases} \tag{2.47}$$

that do not scale the errors but essentially leave them unchanged for positive values.

Feed-Forward Neural Networks with Constrained Weights

Recall that deep nets are trained via SGD that uses small, noisy updates during training to eventually converge against a good weight configuration. Hence, each parameter in a network must be stored as a floating-point value that can store these values. Depending on the system and specific implementation, that means that each parameter of the network requires 4 Bytes (i.e., a `float`) or 8 Bytes (i.e., a `double`) of storage which can quickly add up to hundreds of Megabyte for state-of-the-art networks (c.f. [BOFG20, CZZ+20]). In addition to the parameters, the intermediate values between layers must also be buffered, and each layer must be executed in the correct order using the same precision as the network parameters.

One natural idea to miniaturize deep learning is to consider neural networks with limited precision. For example, if one stores each parameter in 2 Bytes (e.g., in `half` if supported by the execution platform), the overall size is also halved. Going more extreme, one can use (fixed-point) quantization that stores each parameter with a fixed number of bits before and after the decimal. These custom data formats can be implemented with integer arithmetic, which also allows for a more efficient application of the network. Quantization is a field of study in its own right, and similar, the quantization of deep learning also contains a considerable amount of work that cannot be covered comprehensively here (see, e.g., [CZZ+20] for a broader overview). In this thesis, we focus on the most extreme version of quantization that only allows for two possible values represented by a single bit and hence are often called binarized neural networks (BNN), which was first[7] proposed in [HCS+16]. This has not

[7] Technically, [MP43] already discusses networks of logical expressions which can be viewed as binarized neural networks in today's terms.

only by far the greatest impact on the size of the network (i.e., a reduction by a factor of approximately 32), but it also allows for a very efficient forward pass that can be implemented in boolean logic.

The first question when using a quantization is to choose what values should be represented. Since computers naturally work with zeros and ones, the most intuitive choice here would be to constrain the weights of a neural network to $\{0, 1\}$. This approach has two down-sides: Recall that, in a linear layer and in a convolutional layer, the weight is multiplied by its input feature. Hence, a 0 ignores a certain feature, whereas a 1 activates it. This can be advantageous, but it does not allow the network to remove or correct intermediate computations in subsequent layers because either only positive values are added, or entire parts of the network are disabled. Second, and maybe more importantly, gradient-based approaches often have difficulties in properly optimizing 0 weights because the corresponding input feature does not have any impact on the output of the network. This makes the optimization of the network much more difficult. In practice, the quantization $\{-1, 1\}$ or sometimes $\{-a, a\}$ has been shown to be advantageous. This quantization does not ignore entire features but expresses their importance by either using -1 or $+1$. The network can correct intermediate outputs by subtracting values instead of just adding more positive values. And last, every feature now contributes to the output of the network so that gradient-based optimization simply works better.

The forward-pass of a BNN with weights $\{-1, +1\}$ can be implemented in boolean logic only. To do so, we conceptually map the weight $-1 \to 0$ and the weight $+1 \to 1$. Moreover, we ensure that all intermediate representations computed by the network are also binary, which can be ensured by a certain pattern of layers discussed at the end of this section. Under these assumptions, each layer can be implemented as follows:

Linear Layers

Recall that a linear layer computes the dot-product $f_w(z) = \langle w, z \rangle + b$ where $w \in \mathbb{B}^{m \times n}$ and $b \in \mathbb{B}^m$ are now binary parameters, and $z \in \mathbb{B}^n$ is a binary input. The multiplication of two binary values can be implemented with `XOR` operations, i.e. $1 \cdot 1$ and $0 \cdot 0$ both equal 1, whereas $1 \cdot 0$ and $0 \cdot 1$ equal 0. The sum over the resulting bitvector is given by the number of ones in the bitvector after the `XOR` operation. This operation is also known as `popcount`, and CPUs usually ship specific instructions for this operation. If such an operation is not available, it can be efficiently implemented by using multiple look-up tables that contain the number of set-bits for specific bit-strings [JF20]. The output type of linear layers is an `integer`.

Convolution Layers

Recall that a convolutional layer is a linear layer using a Toeplitz matrix $T(w)$ for the weights. Hence, it also computes the dot-product $f_{w,b}(z) = \langle T(w), z \rangle + b$ for binary parameters and inputs. As for linear layers, the multiplication of two binary values can be implemented with `XOR` operations, whereas their sum can be implemented using `popcount`. The output type of convolutional layers is an `integer`.

Pooling Layer

Caution must be exercised when using pooling layers in BNNs. Average pooling computes the average of the input, which cannot be implemented by boolean logic, but

requires more computations. As a resource-friendly variation, one may use sum-pooling

$$f(z)_{i,j} = \sum_{i'=0}^{K} \sum_{j'=0}^{K} z_{i+i',j+j'} \tag{2.48}$$

that simply sums over the inputs. Last, max-pooling that computes the max value across the inputs can also be used. Depending on the specific operation, the output type can be `float`, `integer`, or `boolean`.

Batch Normalization

Recall that BN re-scales and shifts the input to follow a normal distribution. If a BN layer precedes a step function (see below), then the scaling does not change the activation, but only shifting of the input does. Thus, the BN layer can be merged with subsequent step functions by adjusting the step threshold. Differently put, the combination of BN+Step function results in a *learnable* threshold for the activation. This is noteworthy because, in 'regular' NNs, the BN layer does not really impact the capacity of the network but mainly smoothens the optimization. In BNNs, on the other hand, the BN layer adds more capacity to the network by adding a learnable parameter. Note that the running mean of BN is usually a floating-point value and hence the adjusted value of the step is also a float. This is no problem because the IEEE-754 floating-point standard guarantees that float comparison can be implemented by comparing their binary representation directly[8] as if they were integers.

Skip Connections

Skip connections compute the sum of the outputs of two layers z^l and $z^{l'}$. This summation cannot be directly expressed by boolean operations, but a more complex circuit is required. Hence, skip connections are not possible in BNNs. However, a variation of this idea can be implemented by concatenating the inputs instead of summing them [BYBM19, LSSC20]. The concatenating connections simply concatenate the two inputs z^l and $z^{l'}$ into one input vector $[z^l, \phi(z^{l'})]$ where $\phi(z^{l'})$ again makes sure that the dimensionality is correct. As before, the jacobian preservers the gradient information and passes it to the respective inputs. No special operation is required here, and all data types are preserved.

Activation Functions

Besides adding non-linearity to the network, the activation function has a second purpose in BNNs. Here it makes sure that the output of a layer is binary, i.e., it is mapped to $\{0, 1\}$. Most activation functions like sigmoid and tanh cannot be directly expressed in boolean logic and require floating-point computations. The commonly used ReLU activation is simple but does not map the input to $\{0, 1\}$ as it leaves the input unchanged if it is positive. Hence, the step function

$$f(z)_i = \begin{cases} 1 & z_i \geq 0 \\ 0 & \text{else} \end{cases} \tag{2.49}$$

is often used. The forward pass of this activation can easily be implemented by a simple comparison, and it makes sure that the output of a layer is binary. However,

[8] Some edge cases such as `NaN` values must be checked manually, however.

since the step function is not continuous, it does not have a gradient everywhere, and most of the gradients are 0, which makes the optimization difficult. To circumvent this problem, one usually considers the straight-through estimator of $f(z)_i$:

$$J(z)_{i,j} = 1 \tag{2.50}$$

that essentially pretends that $f(z)_i$ is the identity function during optimization.

As hinted at the beginning of this section, the overall architecture of the BNN is vital to ensure that its forward pass can be implemented efficiently in boolean logic. Most layers transform the binary input into integer values, whereas the activation function maps non-binary inputs to binary values. Hence, these operations should always be used in conjunction. For every linear or convolutional layer, there should be a corresponding activation. The combination of BN+Step is also advantageous because it adds another learnable parameter to the network without hurting its memory consumption or forward-pass efficiency. Last, concatenations can be used where desired if the input sizes match, and pooling layers can help to control the size of intermediate representations. Summarizing, we found that a VGG-style network architecture [SZ15] generally works well for BNNs and can serve as a starting point during the architecture search:

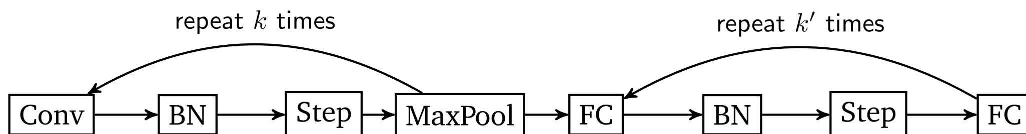


FIGURE 2.9: Example architecture for BNNs based on the VGG architecture [SZ15].

Bibliographic Remark: Before discussing the training of BNNs, a quick bibliographic remark is in order. The previous discussion of BNNs focuses on BNNs that can be executed entirely with boolean logic so that all intermediate computations (e.g., the activations) are also binary. Due to the flexibility of neural networks, it is also possible to use mixed-precision networks, in which, e.g., only the weights are binary, but activations are real-valued. There is a large collection of different NN architectures available that are often described as BNNs (see, e.g., [SL19, BYBM19]), but that use a mix of binary and non-binary data types. Clearly, these architectures allow for more flexible modeling and often have a better predictive performance than ‘pure’ BNNs. However, these architectures also require a much more deliberate implementation that is usually not considered in the corresponding papers: Constraining the weights to $\{-1, 1\}$ reduces the overall memory consumption of the network but does not necessarily improve the overall inferencing speed since floating-point operations are still required. Similarly, type conversion between `boolean/integer` to `float` and vice-versa can also impact the performance negatively if it involves the packing and unpacking of bit-level values for individual variables.

Training BNNs

Training BNNs, in principle, follows the same pattern as training ‘regular’ float networks by applying small changes to the current weights at each iteration to better

fit the training data. In the case of BNNs, we cannot perform gradient-based optimization directly for two reasons: First, the space of weights is discrete, and thus the parameter vector obtained by taking a small step in the opposite direction of the gradient is almost certainly not binary. Second, the sub-gradient of the step function is zero almost everywhere. Thus, arguably the most direct method to train BNNs is to store weights as floating-point numbers during training but round both – activations and weights – to $\{-1, +1\}$ during forward computations. More formally, Hubara et al. [HCS+16] propose a scheme that, during training, stores weights as floating-point numbers constrained to values between -1 and 1 and then *binarizes* the network during the forward pass. Let $b\colon \mathbb{R} \to \{-1, 1\}$ be a binarization function with

$$b(x) = \begin{cases} 1 & x \geq 0 \\ -1 & \text{else} \end{cases} \tag{2.51}$$

and let $B(W)$ denote the element-wise application of b to a tensor W, then we simply apply B during the forward pass to each weight tensor. Algorithm 5 summarizes the application (i.e., the forward pass) of a BNN.

Algorithm 5 Application of a BNN.

```
1: function APPLY_BNN(x, f^1, ..., f^L, W_1, ..., W_L)
2:     z_0 ← x
3:     for l = 1, ..., L do                    ▷ Iterate over all layers
4:         b_l = B(W_l)                        ▷ Binarize weights with Eq. 2.51
5:         z_l ← f^l_{b_l}(z_1, ..., z_{l-1})  ▷ Apply current layer
       return z_{l-1}
```

Algorithm 6 shows the training of BNNs. It starts with a random solution for the weights in line 2 and updates the weights for a given number of epochs. Before each epoch, the training data is shuffled and then iterated in a batch-wise fashion. The network is applied to each batch in line 6 using Algorithm 5. Then the error of the network on the batch is computed (line 7), and all errors are backpropagated through the network in line $8 - 9$ using Eq. 2.35. Finally, the weights are updated in line 10. The description in Algorithm 6 reflects the more engineering-style approach in designing neural networks in which building blocks are combined systematically to form a good training algorithm for a given network architecture. For example, it is well-known that the initialization (i.e., line 2) can have a severe impact on the performance of a neural network in general [SMDH13, FC19, HR18, ACB], but might even be more important in the context of BNNs [DK21]. The choice and possible augmentation of the different batches (i.e., line 5) during training is also a common approach to enhance the performance of deep nets further and BNNs, e.g., by augmentation of the training data [SK19] or by using a full-precision teacher network that generates different target labels for learning [MM18, LVV+20]. Last, the error propagation, as well as the weight updates (lines 8-10), also play a major role in the design of BNN training algorithms. While SGD is a common baseline for many deep networks, the Adam optimizer seems to be more favorable for BNNs[LSL+21], and more specialized variations such as proximal gradient descent have also been proposed [BWL19] as well as optimizers that purely depend on the momentum but not the gradient of a weight [HWG+19].

Algorithm 6 Training of a BNN.

```
1: function TRAIN_BNN(S, f^1, ..., f^L)
2:     W_1, ..., W_L ← init()                          ▷ Generate initial solution.
3:     for next epoch do                               ▷ Perform next epoch
4:         S ← shuffle(S)                              ▷ Shuffle data before epoch
5:         for (X, y) ← next_batch(S) do               ▷ Get next batch
6:             ŷ ← APPLY_BNN(X, f^1, ..., f^L, W_1, ..., W_L)          ▷
   Apply Alg. 5
7:             J_{L+1} ← ∇_ŷ ℓ(ŷ, y)                   ▷ Compute loss of network
8:             for l = L, ..., 1 do                    ▷ Iterate over all layers
9:                 J_l ← backprob_errors(J_{l+1})      ▷ Backpropagate errors
10:            W_1, ..., W_L ← update(J_1, ..., J_L)   ▷ Update weights.
       return W_1, ..., W_L
```

BNNs and Universal Function Approximation

In general, neural networks are universal function approximators: A neural network with a single linear layer can split the search space into two halves since it is a linear function. A network with two linear layers can identify convex sets, whereas a NN with three linear layers can identify arbitrary sets. This intuitive argument has been formally established in [Hor91], which shows that NNs with three (linear) layers and exponentially many neurons on the hidden layer can represent any continuous function. More recent results in [LPW+17] imply that we can trade the width of an NN for its depth and establish a similar result for a bounded width and arbitrary depth. Last, BNNs are universal function approximators as shown in [WZZ+19]. The consistency of neural networks has been established formally in [FL93], although – to the best of our knowledge – a similar result for BNNs is missing.

At this point, it is noteworthy to discuss the connection between neural networks and decision trees: Clearly, a DT in its core is a computation tree and, as such, a special case of a more general computation graph that is the backbone of a neural network. In fact, we can also view DTs as a special type of neural network and train it via gradient-based algorithms and backpropagation, as discussed more in chapter 7. Similarly, it is possible to restructure a tree ensemble as a collection of multilayered neural networks with a particular weight connection [BSW16]. Hence, it does not surprise that, for both models, we can show consistency and universal function approximation. In fact, the general proof idea for both is the same: As discussed in section 2.4.1 the greedy top-down induction of DTs can lead to the isolation of single data points in the leaves. Hence, we can – given the training data is large enough – approximate any function with it. The formal argument for NNs is similar in the sense that if we construct a NN with enough neurons on the hidden layer, then we can also isolate every single data point from the training sample. The difference between both models lies in the training algorithms themselves: For NNs, we choose the architecture *beforehand* incorporating prior knowledge into the architecture (e.g., by using convolutions for image data) and then fit the parameters as well as possible. For DTs, we do not fix the architecture beforehand, but it is *data dependent*.

2.4.3 Naive Bayes

For completeness, we will quickly visit the Naive Bayes classifier as a third example of a discrete classifier. Naive Bayes (NB) is one of the oldest classifiers and a direct

application of Bayes' Theorem [Bay63]. The general idea of Naive Bayes is to predict the majority class given the conditional probability distribution:

$$f(x) = \underset{c=0,\ldots,C-1}{\arg\max}\ P(y = c|x) \tag{2.52}$$

It is difficult to estimate $P(y = c|x) = \frac{P(c)}{P(x)}$ from the given data directly, because it would involve all the different combinations of classes and possible feature values. Hence, we use Bayes' Theorem:

$$P(y = c|x) = \frac{P(y = c)P(x|y = c)}{P(x)} \tag{2.53}$$

Unfortunately, this again involves the term $P(x|y = c) = \frac{P(x)}{P(c)}$ that also requires all the different combinations of feature values and classes. Hence, in Naive Bayes, we *assume* that the features are independent of each:

$$P(x|y = c) = \prod_{i=1}^{d} P(x_i|y = c) \tag{2.54}$$

Now combining Eq. 2.52 -2.54 leads to

$$f(x) = \underset{c=0,\ldots,C-1}{\arg\max} \frac{P(y = c)\prod_{i=1}^{d} P(x_i|y = c)}{P(x)} = \underset{c=0,\ldots,C-1}{\arg\max}\ P(y = c)\prod_{i=1}^{d} P(x_i|y = c) \tag{2.55}$$

where we noticed that the maximization wrt. c is independent of $P(x)$. In Eq. 2.55 we require an explicit model for $P(x_i|y = c)$ as well as an estimate for $P(y = c)$. The estimation of $P(y = c)$ can be done by counting the occurrences of each class in the training data:

$$P(y = c) = \frac{1}{N}\sum_{i=1}^{N} \mathbb{1}\{y_i = c\} = \frac{p_c}{N} \tag{2.56}$$

Due to the independence assumption, the estimation of $P(x_i|y = c)$ can be done independently of each other. In the case of binary or categorical features, we can use a Bernoulli distribution. For real-valued features, a Gaussian distribution is appropriate:

Bernoulli Naive Bayes

Assume that a feature takes a binary value $x_i \in \{0, 1\}$ so that $P(x_i|y = c)$ follows a Bernoulli distribution with the class probability $p_{i,c}$:

$$P(x_i|y = c) = p_{i,c}^{x_i}(1 - p_{i,c})^{x_i}$$

which is either $p_{i,c}$ or $(1 - p_{i,c})$ depending on the value of x_i. To estimate $p_{i,c}$ we use the Maximum-Likelihood estimator:

$$p_{i,c} = \underset{p\in[0,1]}{\arg\max} \prod_{j=1}^{N} \mathbb{1}\{y_j = c\}\, p^{x_{j,i}}(1 - p)^{x_{j,i}} = \frac{1}{N}\sum_{j=1}^{N} \mathbb{1}\{y_j = c\}\, x_{j,i} \tag{2.57}$$

where $\mathbb{1}\{y_j = c\}$ is '1' if the corresponding class equals to c and 0 otherwise. This process can be generalized to categorical features as well, e.g., by introducing a one-hot encoding for each category. More formally, consider a feature $x_i \in \{1, 2, \dots, K\}$ that can take up to K different values. Then we may replace the i-th feature with K binary features $e = (0, \dots, 0, 1, 0, \dots, 0) \in \{0, 1\}^K$ that contain a '1' at the corresponding entry of x_i, i.e. $e_{x_i} = 1$ whereas the remaining entries are 0.

Gaussian Naive Bayes

Assume that a feature takes a real value $x_i \in \mathbb{R}$. A reasonable assumption in this case is, that $P(x_i|y = c)$ follows a Gaussian distribution with mean $\mu_{i,c}$ and variance $\sigma_{i,c}$:

$$P(x_i|y = c) = \frac{1}{\sqrt{2\pi\sigma_{i,c}^2}} \exp\left(-\frac{(x - \mu_{i,c})^2}{2\sigma_{i,c}^2}\right)$$

Again using the Maximum-Likelihood estimator:

$$\mu_{i,c} = \frac{1}{N}\sum_{j=1}^{N} \mathbb{1}\{y_j = c\}\, x_{j,i} \tag{2.58}$$

$$\sigma_{i,c} = \frac{1}{N}\sum_{j=1}^{N} \mathbb{1}\{y_j = c\}\, (x_{j,i} - \mu_{i,c})^2 \tag{2.59}$$

To summarize, we combine both approaches into a single equation. Let D_r be the set of real-valued features and let D_b the set of binary features after a one-hot encoding has been computed, then the prediction of Naive Bayes is:

$$f(x) = \underset{c=0,\dots,C-1}{\arg\max} \frac{p_c}{N} \prod_{i\in D_b} \frac{f_{x_i,c}}{N} \prod_{i\in D_r} \frac{1}{\sqrt{2\pi\sigma_{i,c}^2}} \exp\left(-\frac{(x - \mu_{i,c})^2}{2\sigma_{i,c}^2}\right) \tag{2.60}$$

$$= \underset{c=0,\dots,C-1}{\arg\max}\ p_c \prod_{i\in D_b} f_{x_i,c} \prod_{i\in D_r} \frac{1}{\sqrt{2\pi\sigma_{i,c}^2}} \exp\left(-\frac{(x - \mu_{i,c})^2}{2\sigma_{i,c}^2}\right) \tag{2.61}$$

where p_c is the frequency of class c, $f_{x_i,c}$ are the frequencies for feature i depending on the value of x_i and class c and $\mu_{i,c}, \sigma_{i,c}$ are the parameters of the Gaussians per real-valued feature and class. The training of NB involves estimating p, f, μ, σ which is straightforward using Eq. 2.57 and Eq. 2.58.

Algorithm 7 shows the training algorithm for Naive Bayes. First, a one-hot encoding of all categorical variables is generated. Then in lines 5-7, the prior class frequencies are computed. Lines 8-13 compute the class frequencies per feature, and line 14 - 27 computes the mean and variance of the Gaussians. In order to apply a NB model we implement Eq. 2.61 in Algorithm 8. Again we first compute a one-hot encoding of the variables. Then, the probabilities for binary features are computed in lines 6-7, whereas lines 8-9 compute the probabilities for the real features.

It is worth noting, that, if there are no real-valued features in the data we only compute the product of frequency counts and hence there are no floating-point operations involved in either computing or applying a Naive Bayes model. Unfortunately, the multiplication of counts can quickly lead to large numbers and potentially overflow. A common solution to this problem is to apply the logarithm to Eq. 2.61 and

Algorithm 7 Training of Naive Bayes.

```
1: function TRAIN_NB(S)
2:     X ← one_hot_encoding(X)              ▷ Compute one-hot encoding
3:     D_b ← binary_features(X)             ▷ Get all binary features
4:     D_r ← real_features(X)               ▷ Get all real features
   ▷ Compute class prior
5:     p ← 0_{C,1}                          ▷ Start with C × 1 zero matrix
6:     for i = 1, ..., N do
7:         p_{y_i} ← p_{y_i} + 1            ▷ Update frequency
   ▷ Compute frequencies of all binary variables
8:     f ← 0_{|D_b|,C}                      ▷ Start with |D_b| × C zero matrix
9:     t ← 0                                ▷ Use temporary index variable
10:    for i ∈ D_b do
11:        t ← t + 1
12:        for j = 1, ..., N do
13:            f_{t,y_j} ← f_{t,y_j} + X_{j,i}    ▷ Update frequency
   ▷ Compute mean and variance of all real variables
14:    μ ← 0_{|D_r|,C}                      ▷ Start with |D_r| × C zero matrix
15:    σ ← 0_{|D_r|,C}                      ▷ Start with |D_r| × C zero matrix
16:    t ← 0                                ▷ Reset temporary variable
17:    for i ∈ D_r do
18:        t ← t + 1
19:        for j = 1, ..., N do
20:            μ_{t,y_j} ← μ_{t,y_j} + X_{j,i}    ▷ Update sum
21:    μ ← (1/N) μ                          ▷ Compute mean
22:    t ← 0                                ▷ Reset temporary variable
23:    for i ∈ D_r do
24:        t ← t + 1
25:        for j = 1, ..., N do
26:            σ_{t,y_j} ← σ_{t,y_j} + (X_{j,i} − μ_{t,y_j})^2    ▷ Update sum of squares
27:    σ ← (1/N) σ                          ▷ Compute variance
       return p, f, μ, σ                    ▷ Return estimates
```

replace the products with sums:

$$f(x) = \operatorname*{arg\,max}_{c=1,\ldots,C} \log \left(\frac{p_c}{N} \prod_{i \in D_b} \frac{f_{x_i,c}}{N} \prod_{i \in D_r} \frac{1}{\sqrt{2\pi\sigma_{i,c}^2}} \exp\left(-\frac{(x-\mu_{i,c})^2}{2\sigma_{i,c}^2} \right) \right) \tag{2.62}$$

$$= \operatorname*{arg\,max}_{c=1,\ldots,C} \log p_c + \sum_{i \in D_b} \log f_{x_i,c} + \sum_{i \in D_r} \log \left(\sqrt{2\pi\sigma_{i,c}^2} \right) \frac{(x-\mu_{i,c})^2}{2\sigma_{i,c}^2} \tag{2.63}$$

where we used the logarithm rules to further simplify. Again, if there are no real-valued features, we only need to evaluate the first part of the equation. Since the logarithm is monotone, removing it does not affect the maximization:

$$f(x) = \operatorname*{arg\,max}_{c=1,\ldots,C} \log p_c + \sum_{i \in D_b} \log f_{x_i,c} = \operatorname*{arg\,max}_{c=1,\ldots,C} p_c + \sum_{i \in D_b} f_{x_i,c}$$

Algorithm 8 Application of Naive Bayes.

```
1: function APPLY_NB(x, p, f, μ, σ)
2:     x ← one_hot_encoding(x)
3:     D_b ← binary_features(X_e)                      ▷ Get all binary features
4:     D_r ← real_features(X_e)                        ▷ Get all real features
5:     for c = 1, ..., C do
6:         for i = 1, ..., |D_b| do
7:             p_c ← p_c · f_{x_i,c}                    ▷ Probability for binary features
8:         for i = 1, ..., |D_r| do
9:             p_c ← p_c · 1/sqrt(2π σ²_{i,c}) exp(−(x−μ_{i,c})²/(2σ²_{i,c}))   ▷ Probability for real features
    return argmax_{c=1,...,C} p_c
```

1: **function** APPLY_NB(x, p, f, μ, σ)
2: $x \leftarrow$ one_hot_encoding(x)
3: $D_b \leftarrow$ binary_features(X_e) ▷ Get all binary features
4: $D_r \leftarrow$ real_features(X_e) ▷ Get all real features
5: **for** $c = 1, \ldots, C$ **do**
6: **for** $i = 1, \ldots, |D_b|$ **do**
7: $p_c \leftarrow p_c \cdot f_{x_i,c}$ ▷ Probability for binary features
8: **for** $i = 1, \ldots, |D_r|$ **do**
9: $p_c \leftarrow p_c \cdot \frac{1}{\sqrt{2\pi\sigma_{i,c}^2}} \exp\left(-\frac{(x-\mu_{i,c})^2}{2\sigma_{i,c}^2}\right)$ ▷ Probability for real features
return $\arg\max_{c=1,\ldots,C} p_c$

It follows, that, for categorical or binary features NB is a discrete classifier. To do so, we simply need to replace all products in Algorithm 8 with sums appropriately.

Naive Bayes and Universal Function Approximation

Naive Bayes is a simple algorithm that makes the extreme assumption that features are not correlated with each other. Consequently, it is easy to construct counterexamples that break this assumption, and hence Naive Bayes is neither a universal function approximator nor is consistent. Nevertheless, Naive Bayes works surprisingly well in practice, especially in the context of text data [DP96]. Other application fields such as online learning (see e.g. [BHKP10]) or as a part of other learners, e.g. in the leaf nodes of decision trees further show the usefulness of Naive Bayes [HKP05]. Last, for some Naive Bayes variations, e.g. when kernel density estimation is used, consistency can also be shown [JL95]. Due to its simplicity, Naive Bayes can also be an attractive model in the context of small devices as discussed more in chapter 7. However, it can be difficult to practically deploy NB without any floating-point operations because most problems in the context of small-devices stem from sensor data that is usually real-valued.

Part II

Additive Ensembles

3 | The Bias-Variance Decomposition for Additive Ensembles

In this thesis, we study additive ensembles of discrete classifiers. Formally, they are represented by the model class $\mathcal{F} = \{\sum_{i=1}^{M} w_i h_i | h \in \mathcal{H}, w_i \in \mathbb{R}\}$ that is the weighted combination of a set of M discrete classifiers $h \in \mathcal{H}$:

$$f(x) = \sum_{i=1}^{M} w_i h_i(x) \tag{3.1}$$

While it seems counterintuitive to train *multiple* models in the context of resource constraint devices that might already be challenged by a single model, ensembles offer a variety of advantages over a single model. First, the choice of appropriate base learners allows for great flexibility and already serves as an implicit regularization. Second, additive ensembles can easily remove certain ensemble members by setting their corresponding weight to $w_i = 0$ if required, further reducing the resources required by the entire ensemble. Third, additive ensembles are among the state of the art regarding predictive performance and are usually one of the first methods to boost the performance of single classifiers.

The ERM principle places the model at the center of an optimization problem to minimize a regularized training loss. In practice, we find that many optimization algorithms are not deterministic but incorporate some form of randomness. For example, the choice of the specific training sample $\mathcal{S} \sim \mathcal{D}^N$ already introduces randomness into the training process, the random initialization in BNNs is a more explicit form of randomization, and the choice of split functions in a DT can also be affected by randomization. The following bias-(co-)variance decomposition decomposes the training loss into a bias and a (co-)variance term under these randomizations to gain a better understanding of the structure of the loss and the algorithm's behavior.

The first bias-variance decomposition was proposed by Harry Markowitz in [Mar52] for the mean squared error (MSE) $(f(x) - y)^2$. It states that the expected error of a model $f(x)$ for an observation x with the true label y from a distribution $\mathcal{D}$ can be decomposed into its bias and variance:

$$\mathbb{E}_{\substack{\theta\sim\Theta \\ x,y\sim\mathcal{D}}}\left[(f(x) - y)^2\right] = (\mathbb{E}_{\substack{\theta\sim\Theta \\ x,y\sim\mathcal{D}}}\left[f_\theta(x)\right] - y)^2 + \mathbb{V}_{\theta\sim\Theta}\left[f_\theta(x)\right] \tag{3.2}$$

where Θ is a random process induced by the algorithm that generates f_θ. Its first appearance in the machine learning community was due to Geman et al. in [GBD] for the MSE, which then sparked a series of different decompositions (see, e.g., [UN96, Dom00, Jam03] and references therein). Most notable is the work by Domingos in [Dom00]. It provides a set of consistent definitions for bias and variance

and gives rise to a decomposition of the $0-1$ loss that fits the previous decomposition of the MSE. James gives in [Jam03] a set of different definitions for bias and variance for any symmetric loss function. However, as he notes, these definitions are only applicable to binary classification problems and not applicable to real-valued predictions. A similar decomposition has also been proposed in the context of Product Of Expert models called the ambiguity decomposition. This decomposition also first appeared for the MSE and is equal to the bias-variance decomposition although derived from a distributional point of view [KV94, Hes97]. Later, Hansen and Heskes give in [HH00] a generalized ambiguity decomposition for exponential families. The authors show that any exponential family loss has a decomposition of the form Error $=$ Bias $+$ Variance where 'Bias' and 'Variance' depend on the loss function. Closely related to the following derivation is the work by Jiang et al. in [JLFW17]. Here, the authors derive a generalized ambiguity decomposition for twice differentiable loss functions. Similar to the following discussion, the authors also use a second-order Taylor approximation but seem to ignore the remainder in their construction. Their paper focuses on binary classification losses with a single output and does not directly translate into a new learning algorithm. Our approach, on the other hand, also encapsulates multi-class problems as well as regression problems and therefore is a natural generalization of previous work.

Formally, the bias-variance decomposition studies which *algorithm* consistently produces the best model h with the smallest loss given its randomization Θ:

$$h = \arg\min_{\Theta} \mathbb{E}_{\substack{h\sim\Theta \\ x,y\sim\mathcal{D}}} [\ell(h(x),y)]$$

Theorem 4 decomposes the average error of the model into its bias, (co-)variance, and a remainder term using a second-order Taylor approximation.

Theorem 4 (Generalized Bias-Co-Variance Decomposition). *Let $\ell\colon \mathbb{R}^C \times \mathcal{Y} \to \mathbb{R}$ be a twice-differentiable loss function and let Θ be a random distribution induced by the random choices of an algorithm that result in model h. Further, let $\mu(x) = \mathbb{E}_{h\sim\Theta}[h(x)]$ be the average model and let $\phi(x) = (h(x) - \mu(x))$ be the difference between the average model and some model h. Then the bias-(co-)variance decomposition is*

$$\mathbb{E}_{\substack{h\sim\Theta \\ x,y\sim\mathcal{D}}} [\ell(h(x),y)] = \mathbb{E}_{\substack{h\sim\Theta \\ x,y\sim\mathcal{D}}} [\ell(\mu(x),y)] + \mathbb{E}_{\substack{h\sim\Theta \\ x,y\sim\mathcal{D}}} \left[\frac{1}{2}\phi(x)^T \left(\nabla^2_{\mu(x)}\ell(\mu(x),y)\right)\phi(x)\right] + \mathbb{E}_{\substack{h\sim\Theta \\ x,y\sim\mathcal{D}}} [R] \tag{3.3}$$

where $R \in \mathbb{R}$ is a loss specific constant. Last, let $m \in \mathbb{R}$ be a constant such that $\left|\frac{\partial^3 \ell(\mu(x),y)}{\partial\mu(x)\partial\mu(x)\partial\mu(x)}\right| \le m$ for all $x,y \sim \mathcal{D}$, then it holds that

$$R \le \frac{1}{6} m \max_{h(x)} \|h(x) - \mu(x)\|_1^3 \le \frac{1}{6} mC \max_{i=1,\dots,C} (h(x)_i - \mu(x)_i)^3 \tag{3.4}$$

Proof. We use a second-order Taylor approximation of ℓ around the average $\vec{\mu}(x) = \mathbb{E}_{h\sim\Theta}[h(x)]$. For readability, we now drop the subscript $h \sim \Theta, (x,y) \sim \mathcal{D}$. Similarly, we write $h(x) = \vec{h}$ and $\vec{\mu}(x) = \vec{\mu}$ and $\ell(h(x),y) = \ell(h)$:

$$\mathbb{E}[\ell(h)] = \mathbb{E}[\ell(\mu)] + \mathbb{E}\left[(\vec{h} - \vec{\mu})^T \nabla_{\vec{\mu}} \ell(\vec{\mu})\right] + \mathbb{E}\left[\frac{1}{2}(\vec{h} - \vec{\mu})^T \nabla^2_{\vec{\mu}} \ell(\vec{\mu})(\vec{h} - \vec{\mu})\right] + \mathbb{E}[R]$$

where R denotes the remainder of the Taylor approximation containing the third and higher derivatives.

We note, that $\nabla_{\vec{\mu}}\ell(\vec{\mu})$ does not depend on $\vec{h}$ since $\vec{\mu}$ is a constant given a fixed test point x and therefore $\mathbb{E}\left[\nabla_{\vec{\mu}}\ell(\vec{\mu})\right] = \nabla_{\vec{\mu}}\ell(\vec{\mu})$. Also note, that per definition $\mathbb{E}\left[h\right] = \vec{\mu}$ so that the second summand vanishes:

$$\begin{aligned}\mathbb{E}\left[(\vec{h}-\vec{\mu})^T\nabla_{\vec{\mu}}\ell(\vec{\mu}))\right] = \mathbb{E}\left[(\vec{h}-\vec{\mu})^T\right]\nabla_{\vec{\mu}}\ell(\vec{\mu}) &= \left(\mathbb{E}\left[\vec{h}\right]-\mathbb{E}\left[\vec{\mu}\right]\right)^T\nabla_{\vec{\mu}}\ell(\vec{\mu}) \\ &= (\vec{\mu}-\vec{\mu})^T\nabla_{\vec{\mu}}\ell(\vec{\mu}) = 0\end{aligned}$$

Naturally, the quality of this approximation depends on the magnitude of the remainder, and it becomes exact if the loss function does not have a third derivative. Otherwise, a classic textbook result (see e.g. [Edw73, Kön13]) bounds the magnitude of the remainder for functions that are $k+1$ times continuously differentiable. Let $M, r > 0$ be constants with

$$\begin{aligned}&||\vec{h}-\vec{\mu}||_1 < r \\ &|D^\alpha\ell(\vec{\mu},y)| \le M\frac{s!}{r^s} \text{ for } |\alpha| = s\end{aligned}$$

where we used the multi-index notation $\alpha = (\alpha_1,\dots,\alpha_n) \in \mathbb{N}_0^n$ with $|\alpha| = \alpha_1+\dots+\alpha_n$ and $D^\alpha f = D_1^{\alpha_1}f \dots D_n^{\alpha_n}f$. Then, every Taylor Series with $|\vec{h}-\vec{\mu}|_1 \le \rho < r$ converges with

$$R_s \cdot ||\vec{h}-\vec{\mu}||_1 \le M\left(\frac{\rho}{r}\right)^{s+1}$$

To be more useful, we reformulate this expression. Let there be some $m \in \mathbb{R}$ so that $|D^\alpha\ell(\vec{\mu},y)| \le m$ for all $y, \vec{\mu}$ then

$$m = M\frac{s!}{r^s} \Rightarrow M = \frac{1}{s!}mr^s$$

Re-substituting

$$\begin{aligned}R_s \cdot ||\vec{h}-\vec{\mu}||_1 \le M\left(\frac{\rho}{r}\right)^{s+1} &= \frac{1}{s!}mr^s\left(\frac{\rho}{r}\right)^{s+1} \\ &= \frac{1}{s!}mr^s\frac{\rho^{s+1}}{r^{s+1}} = \frac{1}{s!}m\frac{\rho}{r}\rho^s \\ &\le \frac{1}{s!}m\rho^s \le \frac{1}{s!}m\max_{\vec{h}}||\vec{h}-\vec{\mu}||_1^s \\ &= \frac{1}{s!}m\sum_{i=1}^{C}\max_{h_i}(h_i-\mu_i)^s \\ &\le \frac{1}{s!}mC\max_{h_1,\dots,h_C}(h_i-\mu_i)^s\end{aligned}$$

where the last line holds due to $\frac{\rho}{r} \le 1$ since $\rho < r = \max_{\vec{h}}||\vec{h}-\vec{\mu}||_1$. By setting $s = 3$ we find the remainder for the second-order Taylor approximation:

$$R = R_3(\vec{h}-\vec{\mu}) \le \frac{1}{6}m\max_{\vec{h}}||\vec{h}-\vec{\mu}||_1^3 \le \frac{1}{6}mC\max_{h_1,\dots,h_C}(h_i-\mu_i)^3$$

□

Theorem 4 decomposes the loss of model wrt. to its randomization Θ and the data distribution $\mathcal{D}$ using a second order Taylor approximation. For a sufficiently small remainder R we may write:

$$\mathbb{E}_{\substack{h\sim\Theta \\ x,y\sim\mathcal{D}}}[\ell(h(x),y)] \approx \mathbb{E}_{\substack{h\sim\Theta \\ x,y\sim\mathcal{D}}}[\ell(\mu(x),y)] + \mathbb{E}_{\substack{h\sim\Theta \\ x,y\sim\mathcal{D}}}\left[\frac{1}{2}\phi(x)^T\nabla^2_{\mu(x)}\ell(\vec{\mu}(x),y)\phi(x)\right] \tag{3.5}$$

$$= \mathbb{E}_{\substack{h\sim\Theta \\ x,y\sim\mathcal{D}}}[\ell(\mu(x),y)] + \frac{1}{2}tr\left(\nabla_{\mu(x)}\ell(\mu(x),y)\texttt{cov}(\phi(x),\phi(x))\right) \tag{3.6}$$

where the second line is the quadratic form of the expectation. We interpret this decomposition as a generalized Bias-(Co-)Variance decomposition: While the LHS depicts the expected error of a model h, the first term on the RHS depicts the error of the expected model - or differently coined the *algorithm's* bias. Moreover, the second term can be interpreted as the co-variance of h with respect to the expected model μ given a loss-specific multiplicative constant $\nabla^2_{\mu(x)}\ell(\mu(x),y)$. As shown in the examples below this decomposition is a natural extension of what was presented in the literature already and will serve as a guiding tool in this thesis. However, we also emphasize the fact that this decomposition is only meaningful if the remainder is small and negligible.

Example 1: Mean-squared error

Consider the mean squared error (MSE) of a one-dimensional regression task with $\mathcal{Y} = \mathbb{R}$ and let $z = h(x)$:

$$\begin{aligned}\ell(z,y) &= \frac{1}{2}(z-y)^2 \\ \frac{\partial\ell(z,y)}{\partial z} &= (z-y) \\ \frac{\partial^2\ell(z,y)}{\partial z\partial z} &= 1 \\ \frac{\partial^3\ell(z,y)}{\partial z\partial z\partial z} &= 0\end{aligned} \tag{3.7}$$

The third derivative of the MSE vanishes and thus the above approximation is exact. The resulting decomposition matches exactly the well-known bias-co-variance decomposition.

Example 2: Negative-log-likelihood

As a second example, we consider multi-class classification. Let $z = h(x) \in \mathbb{R}^C$ and let ℓ be the negative-log-likelihood loss (NLL):

$$\begin{aligned}
\ell(z,y) &= -\sum_{i=1}^{C} y_i \log(z_i) \\
\frac{\partial \ell(z,y)}{\partial z_i} &= -\frac{y_i}{z_i} \\
\frac{\partial^2 \ell(z,y)}{\partial z_i \partial z_j} &= -\frac{y_i}{z_i^2}(-1)\mathbb{1}\{i=j\} = \frac{y_i}{z_i^2}\mathbb{1}\{i=j\} \\
\frac{\partial^3 \ell(z,y)}{\partial z_i \partial z_j \partial z_k} &= -2\frac{y_i}{z_i^3}\mathbb{1}\{i=j=k\}
\end{aligned} \tag{3.8}$$

For this loss function, the third derivative does not vanish and thus the decomposition is not exact. Looking at the third derivative we also see, that it can get uncontrollably large for $z_i \to 0$ if $y_i = 1$. Thus, if a model completely fails with a wrong prediction then the decomposition error can be unbounded. Put differently, the performance of a model using the NLL loss cannot be completely explained in terms of 'Bias' and 'Variance' since the remainder is not neglectable.

Example 3: Cross Entropy Loss

Consider the common combination of the NLL loss with the softmax function, also called the cross entropy loss. Again, let $z = h(x) \in \mathbb{R}^C$ and let

$$q_i = \frac{e^{z_i}}{\sum_{i=1}^{C} e^{z_j}} \tag{3.9}$$

be the softmax function, then the cross-entropy loss is:

$$\begin{aligned}
\ell(z,y) &= -\sum_{i=1}^{C} y_i \log(q_i) = -\sum_{i=1}^{C} y_i \log\left(\frac{e^{z_i}}{\sum_{i=1}^{C} e^{z_j}}\right) \\
\frac{\partial \ell(z,y)}{\partial z_i} &= q_i - \mathbb{1}\{y_i = 1\} \\
\frac{\partial^2 \ell(z,y)}{\partial z_i \partial z_j} &= q_i\left(\mathbb{1}\{i=j\} - q_j\right) = \begin{cases} q_i((1-q_j)) & i == j \\ -q_i q_j & \text{else} \end{cases} \\
\frac{\partial^3 \ell(z,y)}{\partial z_i \partial z_j \partial z_k} &= \underbrace{\mathbb{1}\{i=j\} q_i \left(\mathbb{1}\{i=k\} - q_k\right)}_{\in[-1,1]} - \underbrace{q_i q_j \left(\mathbb{1}\{i=k\} - q_k\right)}_{\in[-1,1]} - \underbrace{q_i q_j \left(\mathbb{1}\{j=k\} - q_k\right)}_{\in[-1,1]}
\end{aligned} \tag{3.10}$$

Due to the softmax function we have $\sum_{c=1}^{C} q_c = 1, q_c > 0 \; \forall c = 1, \ldots, C$. The maximum of the third derivative is obtained for pairwise unequal i, j, k ($i \neq j, j \neq k, i \neq k$) and $q_i = q_j = q_k = \frac{1}{3}$:

$$2 \cdot q_i q_j q_k \leq \frac{1}{27} < 0.038 \tag{3.11}$$

Thus, the decomposition error for the cross entropy loss is bounded, and we can explain a model's performance in terms of its bias and variance (up to the bounded

remainder).

Example 4: Exponential loss

The exponential loss can also be used for (binary) classification problems with $\mathcal{Y} = \{-1, +1\}$ and $C = 1$ and is often used in ensembling algorithms such as AdaBoost. Let $z = h(x)$ be the prediction, then:

$$\begin{aligned}
\ell(z, y) &= \exp(-zy) \\
\frac{\partial \ell(z, y)}{\partial z} &= -y \exp(-zy) \\
\frac{\partial^2 \ell(z, y)}{\partial z \partial z} &= y^2 \exp(-zy) = \exp(-zy) \\
\frac{\partial^3 \ell}{\partial z \partial z \partial z} &= -y \exp(-zy)
\end{aligned} \tag{3.12}$$

For this loss function, the third derivative does not vanish, and thus the decomposition is not exact. We may estimate the remainder. Let $z \in [-1, +1]$, then

$$\frac{\partial^3 \ell}{\partial z \partial z \partial z} \leq \exp(1) \tag{3.13}$$

and therefore

$$R = R_3(\vec{h} - \vec{\mu}) \leq \frac{1}{6} \exp(1) \leq 0.454 \tag{3.14}$$

Note, that, if we predict one example wrong we already suffer a loss of $\exp(1) \approx 2.7$ which easily dominates the remainder. Therefore, the bias-variance decomposition for the exponential loss can be meaningful if the model has correct predictions most of the time.

Example 5: Gaussian Hinge Loss

Last, we present a variant of the popular hinge loss function. Since the normal hinge function is not differentiable and variants like smooth hinge and squared hinge do not have smooth second derivatives, we consider a continuously differentiable variant based on the Gaussian error function for a binary classification problem with $\mathcal{Y} = \{-1, +1\}$, $C = 1$ and $z = h(x)$:

$$\begin{aligned}
\ell(z, y) &= \frac{e^{-z^2}}{\sqrt{\pi}} - yz[1 + \operatorname{erf}(-yz)] \\
\frac{\partial \ell(z, y)}{\partial z} &= -y \exp(-zy) \\
\frac{\partial^2 \ell(z, y)}{\partial z \partial z} &= y^2 \exp(-zy) = \exp(-zy) \\
\frac{\partial^3 \ell(z, y)}{\partial z \partial z \partial z} &= -y \exp(-zy)
\end{aligned} \tag{3.15}$$

Similar to the exponential loss function, the third derivative does not vanish, and thus the decomposition is not exact. We may estimate the remainder. Let

$z \in [-1, +1]$, then

$$\frac{\partial^3 \ell}{\partial z \partial z \partial z} = -y \exp(-zy) \le \exp(1) \tag{3.16}$$

and therefore

$$R = R_3(\vec{h} - \vec{\mu}) \le \frac{1}{6} \exp(1) \le 0.454 \tag{3.17}$$

similar to the exponential loss.

4 | Training Additive Ensembles

Additive ensembles are arguably one of the most-used machine learning methods available and frequently place among the best performing models in data science competitions[1]. Recall that an additive ensemble $\mathcal{F} = \{\sum_{i=1}^M w_i h_i | h \in \mathcal{H}, w_i \in \mathbb{R}\}$ is the weighted combination of a set of M (discrete) classifiers $h \in \mathcal{H}$:

$$f(x) = \sum_{i=1}^{M} w_i h_i(x) \tag{4.1}$$

In order to train an additive ensemble at least three choices must be made:

- **Model class:** The model class $\mathcal{H}$ of the base models must be chosen and an appropriate learning algorithm for the base learners must be given.
- **Weights:** Each expert in the ensemble receives a weight that must be computed.
- **Number of experts:** The number of experts must be chosen.

Due to their wide success, there are many ensembling algorithms available in the literature. Given their long history in machine learning and the rapid advancements in ML research in the last years it is impossible to give a comprehensive overview of all the different ensemble methods available. Hence, this section is an attempt to categorize different ensembling algorithms into different classes. Our main guiding tool for this purpose is the generalized bias-co-variance decomposition that can explain many ensembling algorithms as methods to minimize bias and increase co-variance (sometimes also called diversity) in the ensemble.

4.1 Generalized Negative Correlation Learning

The previous chapter discussed the bias-variance decomposition mainly as a theoretical tool to understand the relationship of ensembles trained with different loss functions. In this section, we want to turn it into a practical framework and show how existing algorithms can be recovered with this framework. So far we implicitly assumed that we can evaluate $\mathbb{E}_{\substack{h\sim\Theta \\ (x,y)\sim\mathcal{D}}} [\ell(h(x), y)]$, but this is impossible in practice since we do not know the exact distribution $\mathcal{D}$ and in fact, this is part of the problem we would like to solve. Moreover, it is difficult to compute $\mu(x) = \mathbb{E}_{h\sim\Theta}[h(x)]$ exactly since the algorithm we use for computing h (e.g. SGD or CART) only implicitly induces a distribution over h and the exact nature of Θ for various model classes is ongoing research [BS, SMDH13, AGCH19, KKB17]. Hence, we first present an

[1] https://wandb.ai/site/articles/ama-with-anthony-goldbloom-ceo-of-kaggle

empirical version of Eq. 3.5 assuming that the remainder is sufficiently small, i.e.:

$$\mathbb{E}_{\substack{h\sim\Theta \\ (x,y)\sim\mathcal{D}}}\left[\ell(h(x),y)\right] = \mathbb{E}_{\substack{h\sim\Theta \\ (x,y)\sim\mathcal{D}}}\left[\ell(\mu(x),y)\right] + \mathbb{E}_{\substack{h\sim\Theta \\ (x,y)\sim\mathcal{D}}}\left[\frac{1}{2}\phi(x)^T\nabla^2_{\mu(x)}\ell(\vec{\mu}(x),y)\phi(x)\right] \tag{4.2}$$

and use it to derive a combined loss function that encapsulates many existing works in literature. As usual, we use the given labeled training set $\mathcal{S}$ to approximate the expected loss:

$$\mathbb{E}_{(x,y)\sim\mathcal{D}}\left[\ell(h(x),y)\right] \approx \frac{1}{N}\sum_{(x,y)\in\mathcal{S}}\ell(h(x),y) \tag{4.3}$$

Similarly, we may *approximate* the expected prediction μ with M models:

$$\begin{aligned} \mu(x) = \mathbb{E}_\Theta\left[h(x)\right] \approx f(x) = \frac{1}{M}\sum_{i=1}^{M}h_i(x) \\ \mathbb{E}_\Theta\left[\frac{1}{2}\phi^T\nabla^2_{\vec{\mu}}\ell(\vec{\mu})\phi\right] \approx \frac{1}{2M}\sum_{i=1}^{M}{d_i}^T D d_i \end{aligned} \tag{4.4}$$

where $D = \nabla^2_{f(x)}\ell(f(x),y)$ and $d_i = (h_i(x) - f(x))$. We stress the fact, that we *assume* that these are good approximations. For large M and large N, this is certainly a justified approximation as guaranteed by the law of large numbers. For smaller N and smaller M this is not necessarily the case. However, additive ensembles of this form are arguably the most common type of ensembles and undeniably work well in practice. We define the empirical bias-variance decomposition for any twice-differentiable loss function on $x,y\in\mathcal{S}$ as:

$$\ell(f(x),y) = \frac{1}{M}\sum_{i=1}^{M}\ell(h_i(x),y) - \frac{1}{2M}\sum_{i=1}^{M}{d_i}^T D d_i \tag{4.5}$$

We use Eq. 4.5 as a basis for a learning algorithm: We can either directly minimize its LHS and optimize the entire ensemble in an end-to-end fashion. Alternatively, we use its RHS to derive a regularized objective that trains each model independently with a coupling term enforcing diversity. To do so, let $\lambda\in\mathbb{R}$ be a regularization parameter, then we may minimize:

$$\frac{1}{M}\sum_{i=1}^{M}\ell(h_i(x),y) - \frac{\lambda}{2M}\sum_{i=1}^{M}{d_i}^T D d_i \tag{4.6}$$

Having such a regularized objective available naturally leads to the question of what the exact choice of λ should be. Clearly, $\lambda < 0$ actively *discourages* diversity. For $\lambda = 0$, we arrive at independent objectives. Similarly, a positive λ actively *encourages* diversity. The specific value of λ must be chosen accordingly for the specific problem at hand and can sometimes have a large impact on the optimization problem (see e.g. the experiments in the chapter 5). For some loss functions, e.g. the MSE, a range for different λ values can be given [BWT05], but a more general analysis is yet to be found.

Note that Eq. 4.5 implies two different objectives: Either we minimize $\ell(f(x),y)$ directly without considering the diversity at all or we minimize Eq. 4.6 and control the diversity manually. Having those two objectives available begs the question of which of both may lead to better results. Frankly, since both objectives are equal,

minimizing both will lead to similar, if not equal results. Thus, using either approach comes down to the more practical specifics of the problem at hand: Direct minimization of the loss seems favorable because it automatically finds a good trade-off between bias and variance and no hyperparameter tuning is necessary. Yet, using Eq. 4.6, on the other hand, enables us to train each model independently and only requires some synchronization between models (see e.g. [WRI+19] for a discussion on distributed training in this context). Moreover, this approach allows practitioners to fine-tune the trade-off between bias and variance which might be favorable for specific problems and base models. For example, in deep learning, it is common practice to train networks to achieve zero loss on the training data and sometimes train it even longer [ZBH+17, ZBH+21]. Recall that for a convex loss it holds that ${d_i}^T D d_i \geq 0$ and therefore Eq. 4.5 implies that an ensemble with powerful base learners having zero training loss should not have any variance on the training data. Therefore, as soon as the base learners achieve zero training loss there is no need to invest in variance because the best model (from the training data's perspective) has already been found. Clearly, this is neither the intuition behind the bias-variance decomposition nor is it what we want to achieve. And indeed, in most practical applications we can be sure that even though we have zero training loss, that we will suffer some loss when applying our model to new, unseen data. In this case, it might still be favorable to enforce some diversity between base models during training to achieve a better generalization error.

Interestingly, there is an upper bound of the empirical bias-variance decomposition that combines both approaches into a single objective. This upper bound rescales the individual contributions of the base learners and thus results in the *same* solution as minimizing $\ell(f(x), y)$ or Eq. 4.6 for appropriate choices of λ. As a bonus, this formulation does not depend on the costly computation of D and can also be used when the remainder is not neglectable[2]:

$$
\begin{aligned}
\ell(f(x), y) &\leq \ell(f(x), y) + \frac{1}{M}\sum_{i=1}^{M} \ell(h_i(x), y) \\
&= \frac{1}{M}\sum_{i=1}^{M} \ell(h_i(x), y) - \frac{1}{2M}\sum_{i=1}^{M} {d_i}^T D d_i + \frac{1}{M}\sum_{i=1}^{M} \ell(h_i(x), y) \\
&= \frac{2}{M}\sum_{i=1}^{M} \ell(h_i(x), y) - \frac{1}{2M}\sum_{i=1}^{M} {d_i}^T D d_i
\end{aligned}
$$

This leads us to the following Generalized Negative Correlation Learning (GNCL) objective for $\lambda \in [0, 1]$:

$$
\frac{1}{N}\sum_{j=1}^{N} \left(\lambda \ell(f(x_j), y_j) + \frac{1-\lambda}{M}\sum_{i=1}^{M} \ell(h_i(x_j), y_j) \right) \tag{4.7}
$$

For $\lambda = 0$ this trains M models independently and we will refer to this extreme case as independent training (Ind.). For $\lambda = 1$ all models are trained jointly in an end-to-end fashion and consequently, we call this approach E2E. For values between zero and one, we can smoothly interpolate between these to extremes making the entire

[2] For presentational purposes the remainder R has not been considered here, but this re-formulation also holds when the (empirical) remainder is part of Eq. 4.5.

spectrum available. At first glance, it seems superfluous that we first defined an empirical version of the bias-variance decomposition in Eq. 4.5 and then argued that for training the simpler version in Eq. 4.7 is sufficient. The main reason for this approach is, because Eq. 4.5 allows us to estimate the *exact* values of bias and variance for a trained model, whereas Eq. 4.7 allows for an indirect, yet simpler optimization of both quantities. This way Eq. 4.5 offers a theoretical tool for understanding the trade-off between bias and variance in existing algorithms and models, where Eq. 4.7 gives us an efficient algorithm. Generalized Negative Correlation learning works well for algorithms that allow us to directly minimize an object, e.g. via gradient-based learning. However, even for non-gradient-based algorithms the objectives Eq. 4.6 and Eq. 4.7 form the basis of many existing ensembling algorithms in the literature.

Negative Correlation Learning (NCL)

The earliest works [LY99, BWT05] on NCL-Learning propose to minimize the MSE with a coupling term including the ensembles' diversity (c.f. Eq. (17) in [BWT05]):

$$\frac{1}{M}\sum_{i=1}^{M}\frac{1}{2}(h_i(x)-y)^2-\lambda\frac{1}{M}\sum_{i=1}^{M}\frac{1}{2}(h_i(x)-f(x))^2 \tag{4.8}$$

Substituting the second derivative of the MSE loss in Eq. 4.6 directly leads to this formulation. NCL is a specialized version of GNCL for the MSE loss.

Modular loss

Webb et al. propose to minimize both, the ensemble loss and the loss of each individual expert in a modular loss function (c.f. Eq (4) in [WRI+19]):

$$\lambda \mathrm{KL}(f(x)\|y)+(1-\lambda)\frac{1}{M}\sum_{i=1}^{M}\mathrm{KL}(h_i(x)\|y) \tag{4.9}$$

where KL denotes the KL-Divergence and $\lambda \in [0,1]$ is the regularization strength. Substituting the cross-entropy loss into Eq. 4.7 yields the same formulation. The modular loss is a specialized version of GNCL with the cross-entropy loss.

DivLoss

Opitz et al. use NCL as inspiration to enforce diversity among neural networks by employing the cross-entropy loss between the individual experts' outputs while minimizing the individual and the ensemble loss. They propose to minimize (c.f. Eq. (15) in [OPB16]) the DivLoss:

$$\ell(f(x),y)+\frac{\lambda_1}{M}\sum_{i=1}^{M}\ell(h_i(x),y)-\frac{\lambda_2}{M(M-1)}\sum_{i=1}^{M}\sum_{j\neq i}\ell(h_i(x),h_j(x)) \tag{4.10}$$

where ℓ is the cross-entropy loss and $\lambda_1,\lambda_2 \in \mathbb{R}$ are regularization parameters. It is easy to see that GNCL recovers this objective for $\lambda_1 = 1-\lambda$ and $\lambda_2 = 0$. A more advanced analysis reveals a closer connection between both algorithms:

$$
\begin{aligned}
\ell(f(x),y) + \frac{\lambda_1}{M}\sum_{i=1}^{M}\ell(h_i(x),y) - \frac{\lambda_2}{M(M-1)}\sum_{i=1}^{M}\sum_{j\neq i}\ell(h_i(x),h_j(x)) \\
\geq \ell(f(x),y) - \frac{\lambda_2}{M^2}\sum_{i=1}^{M}\sum_{j=1}^{M}\ell(h_i(x),h_j(x))
\end{aligned}
$$

Since $\ell(h_i(x),h_j(x))$ is convex in its first argument we use Jensen's inequality:

$$
\begin{aligned}
\frac{1}{M}\sum_{i=1}^{M}\ell\left(\frac{1}{M}\sum_{j=1}^{M}h_j(x),h_i(x)\right) &\leq \frac{\lambda_2}{M^2}\sum_{i=1}^{M}\sum_{j=1}^{M}\ell(h_i(x),h_j(x)) \\
\frac{1}{M}\sum_{i=1}^{M}\ell\left(f(x),h_i(x)\right) &\leq \frac{\lambda_2}{M^2}\sum_{i=1}^{M}\sum_{j=1}^{M}\ell(h_i(x),h_j(x)) \\
-\frac{1}{M}\sum_{i=1}^{M}\ell\left(f(x),h_i(x)\right) &\geq -\frac{\lambda_2}{M^2}\sum_{i=1}^{M}\sum_{j=1}^{M}\ell(h_i(x),h_j(x)) \\
-\kappa\frac{1}{M}\sum_{i=1}^{M}\ell\left(f(x),h_i(x)\right) &\leq -\frac{\lambda_2}{M^2}\sum_{i=1}^{M}\sum_{j=1}^{M}\ell(h_i(x),h_j(x))
\end{aligned}
$$

for some $\kappa \leq 1$ and therefore

$$
\begin{aligned}
\ell(f(x),y) - \frac{\lambda_2}{M^2}\sum_{i=1}^{M}\sum_{j=1}^{M}\ell(h^i(x),h^j(x)) &\leq \ell(f(x),y) - \frac{\lambda_2\kappa}{M}\sum_{j=1}^{M}\ell(f(x),h^j(x)) \\
&= \ell(f(x),y) - \frac{\lambda}{M}\sum_{j=1}^{M}\ell(f(x),h^j(x))
\end{aligned}
\tag{4.11}
$$

with $\lambda = \kappa\lambda_2$. Interestingly, Webb et al. show in [WRI+19] that this formula is an alternative formulation of their modular loss when setting $\lambda_1 = 1$ and $\lambda_2 = \lambda \in [0,1]$. It follows, the objective proposed in [OPB16] is an upper bound of the modular loss proposed in [WRI+19], which in turn is a specialized version of GNCL learning for the cross entropy loss.

Diversity with Cooperation

Dvornik et al. propose in [DMS19] an ensemble approach that focuses on diversity and cooperation at the same time. More formally, they propose to use the following objective

$$
\sum_{i=1}^{M}\ell(h_i(x),y) + \frac{\lambda}{(M-1)}\sum_{i=1}^{M}\sum_{j\neq i}\psi(h_i(x),h^j(x)) \tag{4.12}
$$

where ψ is a penalty function to enforce diversity in the ensemble. By using the cross-entropy loss and setting $\psi = -\ell$ we arrive at the DivLoss function for $\lambda_1 = M$ and $\lambda_2 = 1$. Thus, the Diversity with Cooperation approach by Dvornik et al. is closely related to GNCL. However, we note that the authors combine arbitrary loss functions ℓ and penalties ψ. In particular, they propose to use either the cosine-similarly

$\psi(h_i(x), h_j(x)) = \cos(\widehat{h}_i(x), \widehat{h}_j(x))$ or the symmetric KL-divergence $\psi(h_i(x), h_j(x)) = \frac{1}{2}\left(D(\widehat{h}_i(x)\|\widehat{h}_j(x)) + D(\widehat{h}_j(x)\|\widehat{h}_i(x))\right)$ where $\widehat{h}_i(x)$ are the normalized classifier's predictions *without* the true label (the corresponding vector entry is set to 0). As the authors note, each penalty term seems to work for different ensemble sizes, and overall their paper shows mixed experimental results.

Independent Learning (Ind.)

Some articles argue, that the random initialization of deep nets combined with stochastic gradient descent promotes enough diversity. These approaches simply train M models independently thereby optimizing

$$\frac{1}{M}\sum_{i=1}^{M}\ell(h_i(x), y). \tag{4.13}$$

This training method sometimes occurs as a special case for certain hyperparameter settings [BWT05, WRI+19, WRC+20], but is also used as a standalone method [LPC+15, LPB17, SOF+19, DCLT19]. It is easy to see that for $\lambda = 0$ GNCL recovers this objective and thus allows for independent learning.

End-to-End Learning (E2E)

Joint training of the entire ensemble in an End-to-End fashion has also been proposed. This approach ignores the bias and variance of the individual experts but focuses on the ensemble's joint loss by minimizing

$$\ell(f(x), y) \tag{4.14}$$

directly. Here, the literature is slightly more fragmented. End-To-End training occurs in [BWT05, OPB16, WRI+19, WRC+20] as a special case for certain hyperparameter settings. Dutt et al. call this approach a coupled ensemble [DPQ17], whereas Lee et al. call this approach training under an ensemble-aware loss [LPC+15]. It is easy to see that for $\lambda = 1$ GNCL recovers this objective and thus allows for E2E learning.

As mentioned previously, GNCL is a new training objective that is often combined with gradient-based learning, especially in the context of neural networks. Here, the entire ensemble can be viewed as a specific neural network architecture that is trained with gradient descent algorithms minimizing the GNCL objective. As discussed in section 2.4.2, neural networks with enough hidden units and/or enough layers are consistent and universal function approximators. It follows that ensembles of neural networks trained via GNCL are also consistent and universal function approximators given they are large enough.

4.2 Bagging and Related Algorithms

Bagging is an ensemble technique that introduces randomness into the training to promote diversity. In its general form, each base learner receives a random seed that randomizes the individual training processes. Then each model is weighted equally

with $w_i = \frac{1}{M}$. A formal algorithm is given in algorithm 9. Bagging works with any type of base learner $h \in \mathcal{H}$ and the training process can easily be parallelized. Hence, it is one of the fastest ensembling techniques available.

Algorithm 9 Bagging.

1: **function** TRAIN_BAGGING($\Theta, \mathcal{H}, \mathcal{S}, M$)
2: **for** $i = 1, \dots, M$ **do**
3: Obtain random seed $\theta \sim \Theta$
4: Train hypothesis $h_{i,\theta} \in \mathcal{H}$ to minimise $\widehat{L}_{\mathcal{S}}(h_{i,\theta})$
5: $w_i = \frac{1}{M}$
return $f = \sum_{i=1}^{M} w_i h_i(x)$

There are two natural questions arising for bagging-like ensembles. First, what type of randomization should be used for training individual models, and then, how many classifiers M are required? To answer the second question we note, that we may view the prediction of a bagging model as the approximation of the expected model under the randomization Θ:

$$f(x) = \frac{1}{M} \sum_{i=1}^{M} h_i(x) \approx \mathbb{E}_{\theta \sim \Theta}\left[h_\theta(x)\right] \tag{4.15}$$

Hence the question becomes how quickly this approximation converges against its mean. We can use the empirical Bernstein inequality [AMS07] to bound the distance between $f(x)$ and $\mathbb{E}_{\theta \sim \Theta}\left[h_\theta(x)\right]$:

$$|f(x) - \mathbb{E}_{\theta \sim \Theta}\left[h_\theta(x)\right]| \leq C\sigma\sqrt{\frac{2\log(1/\delta)}{M}} + \frac{3R\log(1/\delta)}{M} \tag{4.16}$$

where C is the number of classes, R is the range for each entry in the probability vector and σ is the empirical variance between the prediction of each ensemble member. It follows that a bagging-like ensemble will converge against its mean prediction with a rate of $O\left(\frac{1}{\sqrt{M}}\right)$. The optimal choice for randomization is much more difficult to answer and multiple algorithms exist. The vanilla bagging algorithm in [Bre96] uses bootstrap samples that sample N observations with replacement from $\mathcal{S}$ and fit each model on its respective sample. If random subsets are considered then this algorithm is known as pasting [Bre99]. Random subspaces [Ho98] sample random features instead of sampling observations. The random forest algorithm [Bre01] is a combination of both ideas for DTs models. It uses a bootstrap sample for each tree and samples features for each split. Extremely randomized trees [GEW06] take this approach one step further by randomly sampling a set of splits that are scored on bootstrap samples for each tree. Arguably the most extreme version of this approach is perfect random trees [CZ01] that randomly select splits until the tree perfectly fits their respective bootstrap sample.

Bagging can be expressed in the previously discussed GNCL framework. As mentioned already the independent training (Ind.) of models uses the (random) initialization of neural networks to include diversity into the model training and hence naturally fits the GNCL framework. Bauer and Kohavi propose an extension to bagging called wagging in [BK99] that samples different weights instead of sampling examples or features directly. Oza and Russel show in [OR01] that wagging with weights sampled from a discrete Poisson distribution $w \sim Poisson(1)$ is the same as bagging. Similarly, Webb proposes in [Web00] to use continuous Poisson weights for

wagging, which improves the performance of certain base learners. We can add the wagging weights to the loss

$$\frac{1}{N}\sum_{j=1}^{N}\frac{1}{M}\sum_{i=1}^{M} w_{j,i}\widetilde{\ell}(h^i(x_j), y_j) \tag{4.17}$$

where $w_{j,i} \in \mathbb{R}_+$ is a precomputed Poisson weight for each base learner h_i, sample (x_j, y_j) and $\widetilde{\ell}\colon \mathcal{Y} \times \mathbb{R}^C \to \mathbb{R}_+$ is another loss function. Setting $\ell(h_i(x_j), y_j) = w_{i,j}\widetilde{\ell}(h_i(x_j), y_j)$ and $\lambda = 0$ in Eq. 4.7 yields the same formulation. Hence, we can simulate wagging and bagging with appropriate loss functions inside the GNCL framework. It is difficult to give a general statement on the consistency and universal function approximation of bagging-like ensembles as this is highly dependent on the base learners and the specific randomization. When using neural networks as base learners the previous discussion applies and bagging-like ensembles of such are universal function approximators. As shown later in Theorem 6 there is an equivalence between DTs and DT ensembles and hence, tree ensembles are also universal function approximators. The consistency of bagging-like ensembles, however, is much more difficult to show. The discussion above implies that for $M \to \infty$ the prediction $f(x)$ will converge against its mean if the output of base learners $h(x)$ is bounded. However, it does not guarantee that this mean is biased or not due to the randomization involved in training the individual base learners. And indeed, Biau et al. give in [BDL08] a series of theorems establishing the consistency of averaging classifiers in various scenarios that imply that many popular algorithms such as random forests are not universally consistent. A later study by the same authors in [SBV15] then turns this argument around to show that under mild assumptions a random forest classifier is consistent.

4.3 Boosting and Related Algorithms

Boosting is an iterative algorithm that trains new classifiers to correct the errors of the previous ones and thereby constructs a 'strong' classifier from 'weak' models. The first boosting algorithm was introduced by Freund and Schapire in [FS95], which then sparked a series of different algorithmic variations and studies on the behavior of boosting. There currently exist two views of boosting, namely that boosting can be seen as gradient-descent in function space and a more statistical view in which the errors of a model give rise to an error distribution on which subsequent models are fitted. In this thesis, we will adapt the gradient-descent viewpoint as it allows us to formulate boosting as a general algorithmic framework that encapsulates many different methods. A more detailed overview of boosting, its properties, and related methods are given in [SF12].

Consider a finite or countably infinite set $\mathcal{H} = \{h : \mathcal{X} \to \mathcal{Y}\}$ of $|\mathcal{H}| = K$ models. Recall the general minimization problem of the ERM principle is

$$f^* = \arg\min_{w\in\mathbb{R}^K} \frac{1}{N}\sum_{(x,y)\in\mathcal{S}} \ell\left(\sum_{i=1}^{K} h_i(x)w_i, y\right) \tag{4.18}$$

Hence, in order to find f^* we must compute the appropriate weight vector w for the ensemble $f_w(x) = \sum_{i=1}^{K} w_i h_i(x)$. Since K will presumably be very large, coordinate descent is a favorable choice for this problem. We adapt algorithm 1 (coordinate

descent with Gauß-Southwell update rule) for this problem. To obtain the coordinate-wise derivative we use the chain rule for coordinate i:

$$\frac{\partial}{\partial w_i}\left(\frac{1}{N}\sum_{(x,y)\in\mathcal{S}}\ell(f_w(x),y)\right)=\frac{1}{N}\sum_{(x,y)\in\mathcal{S}}\frac{\partial\ell(f_w(x),y)}{\partial f_w(x)}h_i(x)=\frac{1}{N}\langle\vec{\nabla},\vec{h}_i\rangle \quad (4.19)$$

where $\vec{\nabla}=\left(\frac{\partial\ell(f_w(x_1),y_1)}{\partial f_w(x_1)},\frac{\partial\ell(f_w(x_2),y_2)}{\partial f_w(x_2)},\dots\right)$ and $\vec{h}_i=(h_i(x_1),h_i(x_2),\dots)$. The Gauß-Southwell selection is given by:

$$i=\underset{i\in\{1,\dots,K\}}{\arg\max}\left|\frac{\partial\ell(f_w(x),y)}{\partial w_i}\right|=\underset{i\in\{1,\dots,K\}}{\arg\max}\left|\frac{1}{N}\langle\vec{\nabla},\vec{h}_i\rangle\right|=\underset{i\in\{1,\dots,K\}}{\arg\min}-\left|\frac{1}{N}\langle\vec{\nabla},\vec{h}_i\rangle\right| \quad (4.20)$$

Since we choose w optimally with respect to the minimization we may ignore the absolute value here[3] which leads to

$$i=\underset{i\in\{1,\dots,K\}}{\arg\min}\langle-\vec{\nabla},\vec{h}_i\rangle \quad (4.21)$$

It follows that the selection of the coordinate i is in fact the training of the model h on the residuals $\vec{\nabla}$ of the ensemble. Hence, in each iteration, we select that model from $\mathcal{H}$ that aligns best with the current gradient. To choose the optimal step size α_t we perform a line search:

$$\alpha=\underset{\alpha\in\mathbb{R}}{\arg\min}\frac{1}{N}\sum_{(x,y)\in\mathcal{S}}\ell\left(f_w(x)+h(x)\alpha,y\right)=\underset{\alpha\in\mathbb{R}}{\arg\min}\frac{1}{N}\ell_{\mathcal{S}}\left(\langle\vec{f}_w,\alpha\vec{h}\rangle\right) \quad (4.22)$$

where, with some abuse of notation, $\vec{f}_w=(f_w(x_1),f_w(x_2),\dots)$ and $\vec{h}=(h(x_1),h(x_2),\dots)$. Algorithm 10 offers a very general framework to fit an ensemble. We start with the zero vector that does not select any model from $\mathcal{H}$. Then we compute the optimal step size via a line search and finally update the ensemble. Thus, by construction, the solution $f(x)=\sum_{t=1}^{M}h_t(x)w_t$ is sparse in that sense, that we only select up to M hypothesis from $\mathcal{H}$ in M rounds. We do not need to store $\vec{h}$ explicitly, since most of its weights are 0 by initialization.

Algorithm 10 Boosting as Coordinate Descent with Gauß-Southwell rule.

1: **function** TRAIN_BOOSTING($\mathcal{H},\mathcal{S},M$)
2: $w_0\leftarrow\vec{0}$ ▷ Start with zero solution
3: **for** $t=1,\dots,M$ **do** ▷ Repeat until convergence
4: $\vec{\nabla}\leftarrow\left(\frac{\partial\ell(f_{t-1}(x_1),y_1)}{\partial f_{t-1}(x_1)},\dots\right)$ ▷ Compute residuals on $\mathcal{S}$
5: $h_t\leftarrow\arg\min_{h\in\mathcal{H}}\langle-\vec{\nabla},\vec{h}_t\rangle$ ▷ Train model h_t on residuals
6: $w_t\leftarrow\arg\min_{w\in\mathbb{R}}\frac{1}{N}\ell_{\mathcal{S}}\left(\langle\vec{f}_{w_t},\lambda\vec{h}_t\rangle\right)$ ▷ Linesearch
7: $f_{(t+1)}\leftarrow f_{(t)}+w_th_t$ ▷ Apply gradient step in dimension j
return f_M

An interesting case occurs when we re-sample a coordinate i meaning that we train the same model *twice*. In this case, the gradient $\vec{\nabla}$ also occurred in a previous round and the ensemble now has the same errors as before. Hence, fitting the same model twice is a good indicator that the ensemble is converged given that no other randomization is involved in the algorithm. Last note that if a coordinate is never

[3]We can choose $-w$ or w depending on the direction necessary for minimization.

re-sampled then its weight w_i is determined by its initial step size α_t used when the model was trained. Hence, the line search in line 4 can also be viewed as the computation of the weights for the respective model.

Technical note: We require $\mathcal{H}$ to be finite, which seems overly restrictive in practice. First, we note, that in practice every class of base learners can be viewed as finite for numerical reasons. Consider, without loss of generality, the case that h is defined by some parameter vector θ with m entries. Every practical implementation of h must represent θ with some numerical error, e.g. choose $\theta \in \mathbb{F}_{32}^m$ from the set of m dimensional vectors of 32-bit floating-point numbers. There are 2^{32} floating-point numbers and thus $\mathbb{F}_{32}$ is certainly finite. Since the cartesian product of finite sets is also finite, we note that $\mathbb{F}_{32}^m$ is finite. Second, we see, that it is enough to uniquely enumerate all items in $\mathcal{H}$ (or equivalent in Θ), which means that $\mathcal{H}$ can be infinite, but must be countable. Last, we can lighten these restrictions even more by only demanding that parameter vectors in Θ are measurable: If Θ is not countable, but the identity function $I(\theta) = \theta \ \forall \theta \in \Theta$ is measurable we can still sufficiently approximate it with a countable number of intervals. More formally (c.f Theorem 2), there exists a sequence $(I_n)_{n\in\mathbb{N}}$ of simple functions $I_n \in \{g = \sum_{j=1}^n a_k 1\{S_k\} | g \colon \theta \to \mathbb{R}\}$ with $a_k \in \mathbb{R}$ and $S_k = [l_k, u_k] \subseteq \mathbb{R}$ such that $\forall \theta \in \theta : \lim_{n\to\infty} I_n(\theta) = I(\theta)$. In this view, we can think of g as a stepwise approximation of I using a countable number of intervals S_k. Then again, we can assign a unique index to each element in Θ based on those intervals. Those requirements are rather technical and do not impact the algorithm in practical implementations. However, we wanted to point them out anyway to make not only the practical but also the theoretical boundaries of this representation clear.

AdaBoost

AdaBoost (short for Adaptive Boosting) was developed by Freund and Schapire in [FS95] and is the first boosting algorithm that adapts to the performance of the base classifiers[4]. We now show how to recover the original AdaBoost algorithm in algorithm 10. AdaBoost uses the exponential loss

$$\ell(f(x), y) = \exp(-f(x) \cdot y) \tag{4.23}$$

which has the following derivative

$$\frac{\partial \ell(f(x_i), y_i)}{\partial f(x_i)} = -y_i \exp(y_i \cdot f(x_i)) = -y_i \cdot D_i \tag{4.24}$$

where D_i is called the weight of example i. For brevity, we omit all iterations indices in the following, i.e. $\vec{h}_t$ becomes $\vec{h}$, etc. To better fit the view of the original AdaBoost algorithm, we also normalize the weights D_i:

$$\frac{\langle \vec{h}, -\nabla \rangle}{\sum_{i=1}^N D_i} = -\frac{\sum_{i=1}^N D_i y_i h(x_i)}{\sum_{i=1}^N D_i} = -\sum_{i=1}^N \frac{D_i}{\sum_{i=1}^N D_i} y_i h(x_i) = -\sum_{i=1}^N \tilde{D}_i y_i h(x_i) \tag{4.25}$$

Note that $\sum_{i=1}^N \tilde{D}_i = 1$ and that $\tilde{D}_i$ has a positive sign if $h(x_i) = y_i$ and a negative sign otherwise. Thus:

$$\sum_{i=1}^N \tilde{D}_i y_i h(x_i) = \sum_{h(x_i)=y_i} \tilde{D}_i - \sum_{h(x_i)\neq y_i} \tilde{D}_i = 1 \tag{4.26}$$

[4]Prior algorithms required the performance of the learner to be known before execution.

Re-substituting the normalization term and some rearranging leads to

$$\langle\vec{h}, -\nabla\rangle = -\sum_{i=1}^{N} \tilde{D}_i y_i h(x_i) = -\left(1 - \sum_{h(x_i)\neq y_i} \tilde{D}_i\right) = \frac{\sum_{h(x_i)\neq y_i} D_i}{\sum_{i=1}^{N} D_i} \tag{4.27}$$

where we omitted the constant in the last transformation because it does not affect the minimization. In order to compute the appropriate step size α we have to solve the following optimization problem

$$\begin{aligned}
\alpha &= \arg\min_{\alpha\in\mathbb{R}} \frac{1}{N} \sum_{(x,y)\in\mathcal{D}} L(f+\alpha h, x, y) = \arg\min_{\alpha\in\mathbb{R}} \sum_{i=1}^{N} \exp(-y_i(f+\alpha h)) \\
&= \arg\min_{\alpha\in\mathbb{R}} \sum_{i=1}^{N} \exp(-y_i f)\exp(-y_i\alpha h) \\
&= \arg\min_{\alpha\in\mathbb{R}} \sum_{h(x_i)\neq y_i} D_i \exp(\alpha) + \sum_{h(x_i)=y_i} D_i \exp(-\alpha)
\end{aligned} \tag{4.28}$$

We take the first derivative:

$$\frac{\partial}{\partial\alpha}\left(\sum_{h(x_i)\neq y_i} D_i \exp(\alpha) + \sum_{h(x_i)=y_i} D_i \exp(-\alpha)\right) = \sum_{h(x_i)\neq y_i} D_i \exp(\alpha) - \sum_{h(x_i)=y_i} D_i \exp(-\alpha) \tag{4.29}$$

Setting the first derivative to 0 and solving for α leads to

$$\alpha = \frac{1}{2}\ln\left(\frac{\sum_{h(x_i)=y} D_i}{\sum_{h(x_i)\neq y_i} D_i}\right) \tag{4.30}$$

Now combining algorithm 10 with equation 4.30 and equation 4.27 reproduces the well-known AdaBoost algorithm. This same framework can be followed to recover other boosting algorithms such as ConfidenceBoost or LogitBoost [MBBF99].

The two different views of boosting lead to two different convergences results. The convergence analysis of the CD algorithm as discussed in chapter 2.2 can be readily applied to Algorithm 10 showing that boosting will converge to a solution in $\mathcal{O}\left(\frac{K}{M}\right)$. From the statistical viewpoint of boosting the convergence analysis may also make use of the fact that we train a classifier in each round. Here the so-called weak-learnability assumption plays a central role. It assumes, that for every distribution there is a model $h \in \mathcal{H}$ in the hypothesis class so that its classification error is below $1/2$:

$$P_{(x,y)\sim\mathcal{D}}(h(x) \neq y) \leq \frac{1}{2} - \gamma \tag{4.31}$$

for some $\gamma > 0$. We call γ the edge of model h. For the convergence analysis, we are only interested in the training error, and hence we may replace $\mathcal{D}$ with $\mathcal{S}$ and arrive at the empirical weak-learnability assumption. It is possible to show exponential minimization of the error for AdaBoost when the weak-learnability assumption holds [SF12]. Both views guarantee the convergence of boosting against a stationary point. The coordinate descent view offers more freedom in the choice of the loss function and the statistical view places more emphasis on the base models in the ensemble. Intuitively, the weak-learnability assumption guarantees a constant change in the loss during minimization. Mukherjee et al. were able to integrate this intuition into a

formal proof for the convergence of AdaBoost in [MRS13]. They show, that AdaBoost requires at least $\Omega(\frac{1}{\varepsilon})$ rounds to achieve a loss within ε of the optimal loss *without* the weak-learning assumption thereby connecting both views.

The convergence analysis of boosting gives us the safety that the algorithm will converge which is a necessary condition for its consistency. Moreover, it also implies that its training error will reach 0 if the weak-learning assumption holds meaning that we can perfectly fit the training sample $\mathcal{S}$. Combining this result with the law of large numbers implies that every boosting algorithm can transform a weak learner that is better than random guessing (i.e.g it has a positive edge) into an arbitrary strong learner, i.e. into a universal function approximator. This argument has been first presented for AdaBoost in [FS95], and it was later shown that AdaBoost and some of its variations are also consistent [LV04, ZY05, BT06]. For a more comprehensive overview of this topic, we refer interested readers to [SF12].

Boosting is a very general framework to minimize the loss $\ell(f(x), y)$ via approximating the gradients in each iteration. Thus, it is closely related to GNCL with $\lambda = 1$, but behaves fundamentally different from GNCL because it is designed to greedily approximate gradients for non-differential base learners. Theoretically, both approaches could be combined: The proposed GNCL objective can e.g. be minimized via stochastic gradient descent (as usually done) or for example by boosting weak learners on the GNCL objective.

4.4 Dropout and Pseudo-Ensembles

Dropout [SHK+14] is a regularization method for deep nets, that randomly sets weights to zero during the forward pass. It is anecdotally sometimes referred to as 'the ensemble of possible subnetworks' [BS13, GG16]. Bachman et al. studied this connection more closely and proposed in [BAP14] the term pseudo-ensembles. Pseudo-ensembles are ensembles that are derived from a single large network by perturbing it with a noise process. Although not explicitly mentioned, snapshot ensembles [QZR+14, HLP+17] that store multiple versions of the same network (e.g. by storing the model every few epochs during training) can also be seen in this framework. Pseudo-ensembles minimize the following objective

$$\frac{1}{N}\sum_{j=1}^{N} \mathbb{E}_\theta \left[\ell_\theta(\mu(x_i), y_i)\right] + \lambda \mathbb{E}_\theta \left[Z\left(\mu(x_i), \mu_\theta(x_i)\right)\right] \tag{4.32}$$

where μ denotes the 'mother' net, μ_θ is a 'child net' under the noise process θ, ℓ is a loss function and Z is a regularizer with regularization strength λ. The bias-variance decomposition discussed earlier implies the same objective with $Z = \phi^T \nabla^2_{\vec{\mu}} \ell(\vec{\mu})\phi$ and by introducing λ as presented earlier. Unfortunately, the authors of [BAP14] do not discuss how to directly minimize this objective under the noise process θ. Interestingly, for their experiments they use the *same* GNCL objective in Eq. 4.7 with the cross entropy loss[5]. Hence, GNCL can be viewed as an empirical version of pseudo-ensembles. It is noteworthy however that both approaches have two different viewpoints: Pseudo-Ensembles train a single network and spawn a diverse set of offspring from this large network, whereas GNCL tries to combine a set of smaller models into a large one. In a sense, GNCL starts with many small learners and combines them into

[5]This is not explicitly stated in the paper, but can be observed in the original implementation `https://github.com/Philip-Bachman/Pseudo-Ensembles`.

one big model, whereas Pseudo-Ensembles start from a big model to extract multiple sub-models from it.

4.5 Stochastic Multiple Choice Learning

Lee et al. train in [LPC+15, LPC+16] a diverse ensemble of classifiers by using stochastic multiple choice learning (SMCL). Instead of training all ensemble members on all the available data, they only update that member with the smallest loss. This way, the diversity that naturally occurs e.g. due to the random initialization, is further promoted. We could not find a direct mathematical relationship between SMCL and GNCL.

5 On the Double-Descent Phenomenon in Random Forests

Before we discuss ensembles in the context of small devices we will visit one additional theoretical explanation for the performance of (decision tree) ensembles that has emerged recently as a by-product of deep learning research. Chapter 2.3 introduces the ERM principle as one of the core theoretical foundations of machine learning. A central part of ERM is to use a regularizer that punishes overly complex models so that learning focuses on small, concise models. The bias-variance trade-off (not to be confused with the bias-variance decomposition) is an arguably extremer version of this idea and forms another core theoretical foundation of ML. It states that a smaller, less complex model will generalize better than a large, very complex model if both have comparable empirical error [SSBD14]. Hence, one should always strive for a good balance between model complexity and empirical error. The last years of deep neural network research challenged this widely accepted notion by using larger and larger models with more and more parameters that seemingly do not overfit. One particular remarkable observation is that DNNs seem to exhibit a double u-shaped curve sometimes dubbed 'double descent' (see Figure 5.1): With increasing model complexity, the training error approaches 0 quickly and the test-error also shrinks up to a point where overfitting starts and the test-error rises again. However, further increasing the model complexity, e.g. beyond what is reasonable given the available training data suddenly leads to a drop in test error again.

The reasons for this behavior are not entirely discovered yet and the question is, whether SGD-like learning algorithms or the model architecture of deep nets are the decisive factors for this phenomenon [ZBH+21]. Similarly, a line of research asks if such a phenomenon exists for other, non-deep learning models and other learning algorithms such as random forests (RF). Belkin et al. showed in [BHMM19] that there seems to be a universal double-descent across different types of models. In particular, the authors show that RFs also exhibit a double-descent which could explain the miraculous performance of RFs in practice. We found that the discussion of the double-descent phenomenon is not properly placed in the context of existing research. In particular, the authors suggest using the *total* number of decision nodes in the random forests as their complexity measure, whereas learning theory implies that the *average* number of decision nodes in the forest would be a more appropriate measure. Similarly, training over-parameterized decision trees is virtually impossible because we cannot train larger trees than what we have data available. Last, the diversity among the trees inside the ensemble has long been cited as one of the cornerstones of a RF's good performance and hence cannot be ignored in this discussion. We revisit these explanations in the context of the recently emerged double-descent phenomenon and ask: Is there a double descent in RF? We emphasize that we couple existing theoretical knowledge with empirical evidence and leave a more thorough

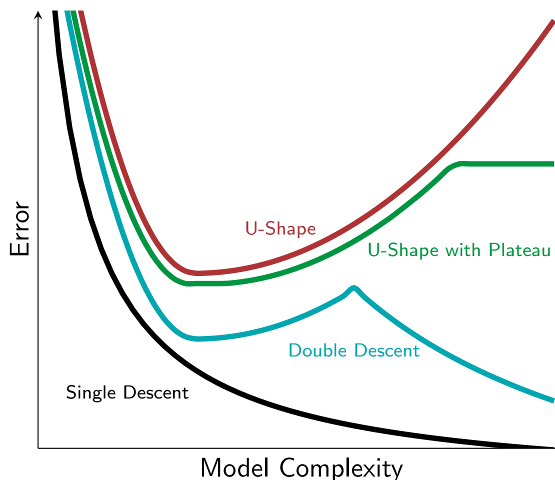


FIGURE 5.1: Bias-Variance trade-off: A single descent occurs if the error decreases while the model complexity increases. A U-Shaped curve can be seen when overfitting occurs and the error increases again after a certain model complexity is reached, either with or without a plateau. A double-descent can be seen where the loss suddenly again decreases if the model complexity is further increased beyond the point of overfitting.

theoretical analysis of this subject for future research. The list of datasets used for these experiments is depicted in Table 5.1.

TABLE 5.1: Datasets used for the experiments. We performed minimal pre-processing on each dataset removing instances that contain `NaN` values and computed a one-hot encoding for categorical features. Each dataset is available under `https://archive.ics.uci.edu/ml/datasets`.

Dataset	N	C	d
Adult	32 562	2	108
Bank	45 211	2	51
EEG	14 980	2	14
Magic	19 019	2	10
Nomao	34 465	2	174

5.1 The Complexity of Tree Ensembles

For clarity, we refer to overfitting as the u-shaped curve depicted in Figure 5.1 in which the test error increases again after a certain complexity is reached, but *not* the fact that in many practical applications there is a gap between the test and training error. Recall that a random forest is a convex combination of M axis-aligned decision trees $h_i \in \mathcal{H}$ each scaled by the same weight $w_i = \frac{1}{M}$:

$$f(x) = \sum_{i=1}^{M} w_i h_i(x) = \frac{1}{M} \sum_{i=1}^{M} h_i(x) \tag{5.1}$$

Here $\mathcal{H}$ is the model class of axis-aligned decision trees and f is a random forest. For completeness, Figure 11 depicts the random forest algorithm, but note that the following experiments can be repeated for all the bagging ensembles mentioned in section 4.2 with decision trees as base learners.

Algorithm 11 Training of a random forest.

```
 1: function TRAIN_RF(S, M, n_l, d)
 2:   for i = 1, ..., M do                          ▷ Train M models
 3:     S_i ← bootstrap_sample(X)                   ▷ Draw bootstrap sample
 4:     h_i ← new_root_node()                       ▷ Init. new tree
 5:     nodes ← [(h_i, S_i)]                        ▷ List of remaining nodes and data
 6:     l ← 0                                       ▷ Number of leafs
 7:     while len(nodes) > 0 do                     ▷ Over all remaining nodes
 8:       n, S_n ← nodes.pop()                      ▷ Get current node and data
 9:       d_i ← sample_features(d)                  ▷ Draw features
10:       split ← best_split(S_n, d_i)              ▷ Get best split
11:       n.set_split(S_n, split)                   ▷ Store split
12:       S_l, S_r ← split_data(S_n, split)         ▷ Split into left/right
13:       c_l, c_r ← new_children(n)                ▷ Init new children
14:       if valid(S_r, S_l) and l < n_l then       ▷ Check if split is valid
15:         nodes.append(c_l, S_l)                  ▷ Add left child
16:         nodes.append(c_r, S_r)                  ▷ Add right child
17:       else
18:         n.is_leaf ← true                        ▷ Set leaf node
19:         n.set_classes(S_i)                      ▷ Estimate class frequencies
20:         l ← l + 1                               ▷ Set leaf node
21:     trees.append(h_i)                           ▷ Add hypothesis
22:     weights.append(1/M)                         ▷ Add weight
      return h_1, ..., h_M
```

In statistical learning theory the generalization error of a model f is bounded in terms of its empirical error $\frac{1}{N}\sum_{i=1}^{N}\ell(f(x_i), y_i)$ given some loss function ℓ and a complexity measure $\mathcal{R}$ for the trained model. For concreteness consider a binary classification problem with $\mathcal{Y} = \{-1, +1\}$ and let $f\colon \mathcal{X} \to [-1, +1]$ be a prediction model and $\rho > 0$ be the classification margin. We denote the binary classification error of f on $\mathcal{D}$ with respect to ρ with $L_\rho(f)$ and the empirical classification error of f wrt. ρ on $\mathcal{S}$ with $\widehat{L}_{\rho,\mathcal{S}}(f)$:

$$L_\rho(f) = \mathop{\mathbb{E}}_{(x,y)\sim\mathcal{D}}\left[1\{yf(x) \le \rho\}\right] \tag{5.2}$$

$$\widehat{L}_{\rho,\mathcal{S}}(f) = \mathop{\mathbb{E}}_{(x,y)\sim\mathcal{S}}\left[1\{yf(x) \le \rho\}\right] = \frac{1}{N}\sum_{i=1}^{N} 1\{y_i f(x_i) \le \rho\} \tag{5.3}$$

The following theorem bounds the generalization error of a convex combination of classifiers in terms of their individual Rademacher complexities.

Theorem 5 (Convex combination of classifiers [CMS14]). *Let $\mathcal{H} = \bigcup_{j=1}^{k}\mathcal{H}_j$ denote a set of base classifiers and let $f = \sum_{i=1}^{M} w_i h_i$ with $w_i \in [0,1], \sum_{i=1}^{M} w_i = 1$ be the convex combination of classifiers $h_i \in \mathcal{H}$. Furthermore, let $\mathcal{R}(h_i)$ be the Rademacher complexity of the i-th classifier. Then, for a fixed margin $\rho > 0$ and for any $\delta > 0$, with probability at least $1-\delta$ over the choice of sample $\mathcal{S}$ of size N drawn i.i.d. according to $\mathcal{D}$, the*

following inequality holds:

$$L_0(f) \leq \widehat{L}_{\mathcal{S},\rho}(f) + \frac{4}{\rho}\sum_{i=1}^{M} w_i \mathcal{R}(h_i) + C(N,k,\rho)$$

where $C(N,k,\rho)$ is a constant depending on N,k,ρ which tends to 0 for $N \to \infty$ and for any k,ρ.

Theorem 5 offers two interesting insights: First, the Rademacher complexity of a convex combination of classifiers does *not* increase, but it is the weighted average of individual complexities $\mathcal{R}(h_i)$:

$$\mathcal{R}(f) = \sum_{i=1}^{M} w_i \mathcal{R}(h_i) \tag{5.4}$$

Hence, an ensemble is not more likely to overfit than each of its individual base learners. Second, the individual Rademacher complexities of each base learner are scaled by their respective weights, and therefore a very complex tree with little weight $w_i \to 0$ does not hurt the generalization performance of the forest. The key question now becomes how to compute the Rademacher complexity of the trees inside the forest. It is well-known that the Rademacher complexity $\mathcal{R}(h_i)$ is related to the VC-dimension $D(h_i)$ of each tree via (see e.g. [SSBD14])

$$\mathcal{R}(h_i) \leq \sqrt{\frac{2D(h_i)}{N}}. \tag{5.5}$$

Interestingly, the exact VC-Dimension of decision trees is unknown. Asian et al. performed an exhaustive search to compute the VC dimension of trees with depth up to 4 in [AYA], but so far no general formula has been discovered. However, there exist some useful bounds. A decision tree h_i with n_i nodes trained on d_i binary features has a VC-dimension of at most [Man97]:

$$D(h_i) \leq (2n_i + 1)\log_2(d_i + 1) \tag{5.6}$$

Leboeuf et al. extend this bound for continuous features in [LLM20] by introducing the concept of partition functions into the VC framework. They are able to show that the VC-dimension of a decision tree trained on d_i continuous features is of order $\mathcal{O}\left(n_i \log(n_i + d_i)\right)$. Unfortunately, the expression discovered by the authors is computationally expensive so that experiments with larger trees are impractical[1]. For our analysis in this chapter, we are interested in the asymptotic behavior of DTs and RFs. Hence, we use the following *asymptotic* Rademacher complexity:

$$\widehat{\mathcal{R}}(h_i) = \sqrt{\frac{2n_i \log(n_i + d_i)}{N}} \tag{5.7}$$

5.2 There is no Double-Descent in RF

Belkin et al. empirically showed in [BHMM19] that RF exhibits a double-descent curve. Similar to our discussion here, the authors introduce the number of nodes as a

[1] The authors provide a simplified version of their expression which works well for trees with less than 100 leaf nodes on our test system, but anything beyond that would take too long.

measure of complexity for single trees but then use the *total* number of nodes in the forest throughout their discussion. While we acknowledge that this is a very intuitive definition of complexity it is not consistent with the results in learning theory. Hence, we propose to use the *average* (asymptotic) Rademacher complexity as a capacity measure. We argue, that with this adapted definition, there is no double-descent occurring in RFs but rather a single descent in which we fit the training data better and better or there is a slight overfitting depending on the data.

We validate our hypothesis experimentally. To do so we train various RF models with different complexities and compare their overfitting behavior. We will focus on the five datasets depicted in 5.1. We specifically choose binary classification problems with a variable number of features $10 - 174$ on medium-sized datasets. We performed minimal pre-processing on each dataset removing instances that contain `NaN` values and computed a one-hot encoding for categorical features. By today's standards, these datasets are small to medium size which allows us to quickly train and evaluate different configurations, but large enough to train large trees. The code for these experiments is available under https://github.com/sbuschjaeger/rf-double-descent.

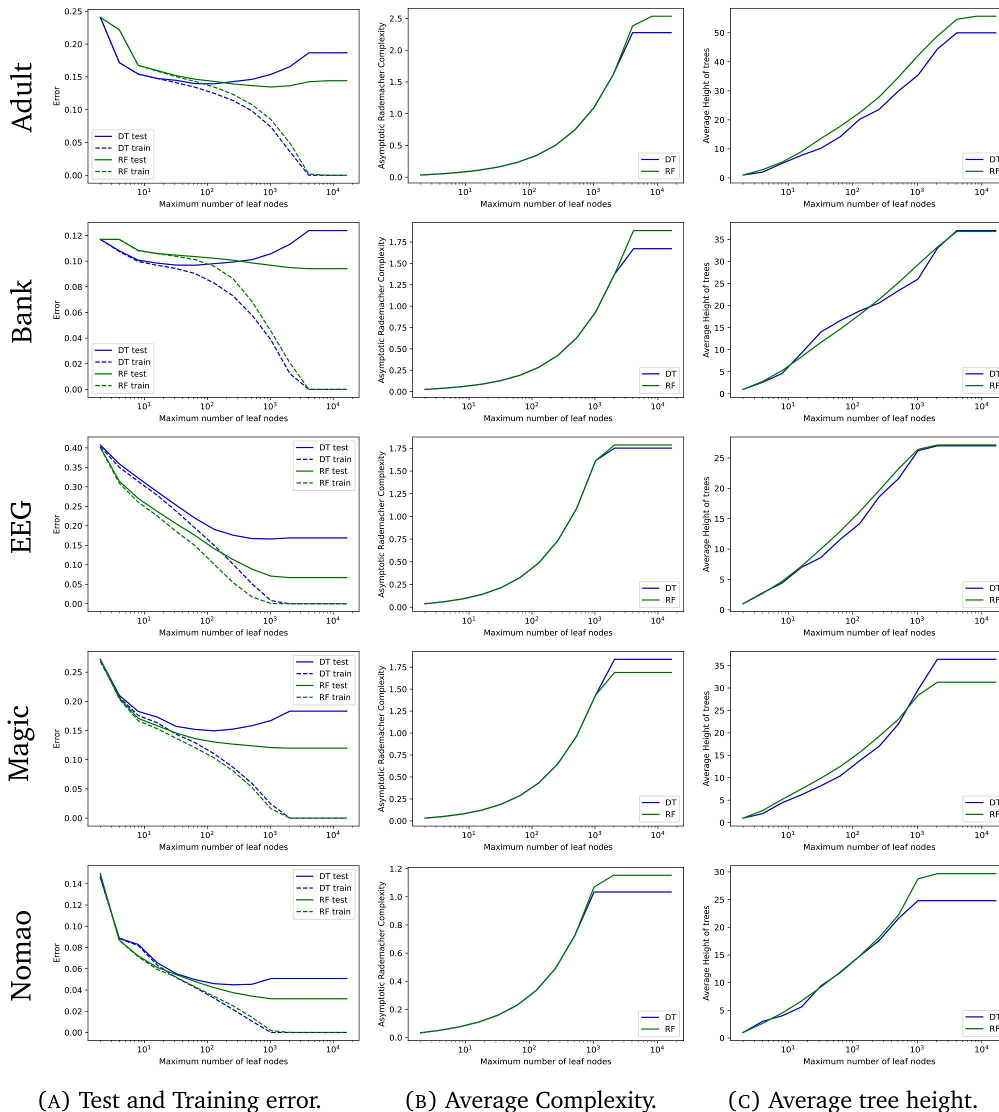
In our experimental evaluation, we want to study the overfitting behavior of DTs and RFs by training models with different Rademacher complexities. To do so we need to control the empirical error $\frac{1}{N}\sum_{i=1}^{N}\ell(f(x_i), y_i)$ as well as the Rademacher complexity of the model. Our experimental protocol is as follows: As implied by Theorem 5 the Rademacher complexity of a forest does not change when adding more trees, and it is bounded by the most complex tree, i.e. $R(f) \leq \max\{\mathcal{R}(h_1), \dots, \mathcal{R}(h_M)\}$. Second, Oshiro et al. showed in [OPB12] empirically that the predictions of a RF stabilizes between 128 and 256 trees, and adding more trees to the ensemble does not yield significantly better results. Hence, we train ensembles with $M = 256$ trees, and we do not expect the loss of the forests to change significantly when adding more trees. In order to control the Rademacher complexity of the individual trees we found that changing the maximum number of leaf nodes $n_l \in \{2, 4, 8, 16, 32, 64, \dots, 16384\}$ gives the most direct control over the Rademacher complexity. In all our experiments we perform a 5-fold cross-validation and report the average error across these runs.

Figure 5.2a shows the results of this experiment. Solid lines show the test error and dashed lines are the training error. Note the logarithmic scale on the x-axis. It can be clearly seen that for both, RF and DT, the training error decreases towards 0 for larger n_l values. On all but the EEG dataset, we see the classic u-shaped overfitting curve for a DT in which the error first improves and then suddenly increases again until it reaches a plateau in which no more overfitting occurs. Looking at the RF we see a single-descent curve on most datasets and some small signs of overfitting on the adult dataset.

When there is no double-descent in random forests, then why are they performing better than single trees? Interestingly, the Rademacher complexity may already offer a reasonable explanation of this behavior. First, a RF uses both, feature sampling and bootstrapping for training new trees. When done with care[2], then feature sampling can reduce the number of features to $d_i \ll d$ so that $\log_2(d_i + 1)$ also becomes smaller and thereby also reduces $\mathcal{R}$. Second, bootstrap sampling samples data points with replacement. Given a dataset with N observations, there are only $1 - \lim_{N\to\infty}\left(1 - \frac{1}{N}\right)^N = 1 - e^{-1} \approx 0.632$ unique data points per individual bootstrap sample. Thus, the effective size of each bootstrap sample reduces to roughly $N_i = 0.632 \cdot N$ which can lead to smaller trees because the entire training set is

[2] E.g. scikit-learn [PVG+11] may evaluate more than d_i features if no sufficient split has been found.

smaller and easier to learn due to duplicate observations. Last and maybe most important, tree induction algorithms such as CART are adaptive in the sense, that the tree structure is data-dependent. In the worst case, a complete tree is built in which single observations are isolated in the leaf nodes so that every leaf node contains exactly one example. However, it is impossible to grow a tree beyond isolating single observations because there simply is not any data left to split. Subsequently, the Rademacher complexity cannot grow beyond this point and is limited by an inherent, data-dependent limit. We summarize these arguments into the following hypothesis.



(A) Test and Training error. (B) Average Complexity. (C) Average tree height.

FIGURE 5.2: Test error (solid lines) and training error (dashed lines), the average Rademacher complexity and the average height of the trees over the maximum number of leaf nodes n_l. Each row depicts one dataset. Results are averaged over a 5-fold cross-validation. Best viewed in color.

5.3 The Complexity of RF is bounded by the Data

Figure 5.2b shows the Rademacher complexity of the DT and RF for the previous experiment. As one can see the Rademacher complexity steeply increases until it converges against a maximum from which it then plateaus. So indeed, a DT and a RF have both an inherent maximum Rademacher complexity given by the data as expected. Contrary to the above discussion, however, the RF has a *larger* Rademacher complexity than a DT on all but the magic datasets. For a better understanding, we look at the average height of trees in Figure 5.2c. Here we can see that RF – on average – has larger trees than the DT given the same number of maximum leaf nodes. We hypothesize that due to the feature sampling and bootstrapping that suboptimal features are chosen during the splits. Hence, a RF requires more splits in total to achieve a small loss leading to larger trees with larger Rademacher complexities.

Combining both experiments leads to a mixed explanation of why RF seems to be so resilient to overfitting: For trees trained via greedy algorithms such as CART one cannot (freely) over-parameterize the model because its complexity is inherently bounded by the data. Even if one allows for more leaf nodes, the algorithm simply cannot make use of more parameters. A similar argument holds for a random forest: Adding more trees does not increase the Rademacher complexity as implied by Theorem 5. Thus, one can add more and more trees without the risk of overfitting. Again increasing n_l only increases the Rademacher complexity up to the inherent limit given by the data. Hence – even if one allows for more parameters – a RF cannot make use of them. Its Rademacher complexity is inherently bounded by the data. However, as shown in Fig. 5.2 there does not seem to be a direct, data-independent connection between the maximum number of leaf nodes and this inherent maximum Rademacher complexity.

5.4 Complexity does not predict a RF's Error

The above discussion already shows that the Rademacher complexity of a forest does not seem to be an accurate predictor for the generalization error of the ensemble. In this section, we further challenge the notion that complexity is a predictor of the performance of a tree ensemble and construct ensembles with large complexities that do not overfit and trees with small complexities that do overfit. To do so, we first show that the decision boundary of a RF can be represented by a single DT and vice-versa. The idea is as follows: The decision boundary of a single DT is a set of d-dimensional hypercubes and similarly, the decision boundary of a RF is the average of a set of these hypercubes. While the decision boundary of a RF becomes smoother due to the averaging it can still be represented by a (larger) set of d-dimensional hypercubes. To represent this larger set of hypercubes we can simply append copies of a tree to the leaf nodes of the other trees and recursively repeat the process until all trees are merged into a single one. Theorem 6 formalizes this idea.

Theorem 6 (Equivalence of RF and DT). *Let $h\colon \mathcal{X} \to \mathcal{Y}$ be an axis-aligned decision tree and let $f\colon \mathcal{X} \to \mathcal{Y}$ be a random forest, then there exist*

1) *a forest f' so that $\forall x \in \mathcal{X} : h(x) = f'(x)$*

2) *a tree h' so that $\forall x \in \mathcal{X} : h'(x) = f(x)$*

Proof. The proof of 1) is straightforward. We simply copy the tree M times to form the forest:

$$f'(x) = \frac{1}{M}\sum_{i=1}^{M} h(x) = h(x)$$

which concludes the proof for 1).
For 2) we show how to construct h' from the forest. To do so recall that each path from the root node to a leaf node in an axis-aligned DTs represents a d-dimensional hypercube and each hypercube is then associated with a (constant) prediction. While a RF offers a much smoother overall decision boundary due to the averaging its basic building block still remains hypercubes. The overlap of two hypercubes in d-dimensions can be represented with 3^d smaller hypercubes which in turn can be represented with a regular DT. More formally, let there be two trees h_1 and h_2. Intuitively we now remove all leaf nodes in h_1 and replace them with copies of the second tree h_2. Then we replace all leaf nodes in the new tree with the average of the original predictions from h_1 as well as the predictions of the corresponding leaf nodes in h_2. More formally, consider a fixed $x \in \mathcal{X}$ and let i be that path for h_1 so that $s_{1,i}(x) = 1$ and let j be that path for h_2 with $s_{2,j}(x) = 1$. Further, let $s_{1,i,l}(x)$ be the split of the l-th node on the i-th path in h_1 and let $s_{2,j,l}(x)$ be the split of the l-th node on the j-th path in h_2 respectively. Then:

$$\begin{aligned}
h(x) &= \frac{1}{2}\left(g_{1,i}s_{1,i}(x) + g_{2,j}s_{2,j}(x)\right) \\
&= \frac{1}{2}\left(g_{1,i}\prod_{l\in L_{1,i}} s_{1,i,l}(x) + g_{2,j}\prod_{l\in L_{2,j}} s_{2,j,l}(x)\right) \\
&= \frac{1}{2}\left(g_{1,i} + g_{2,j}\right)\prod_{l\in L_{1,i}} s_{1,i,l}(x)\cdot\prod_{l\in L_{2,j}} s_{2,j,l}(x) \\
&= \sum_{i=1}^{L_1}\sum_{i=2}^{L_2}\frac{1}{2}\left(g_{1,i} + g_{2,j}\right)s_{1,i}(x)\cdot s_{2,j}(x) \\
&= h'(x)
\end{aligned}$$

Recursively repeating then yields the Theorem with

$$n'_i = \begin{cases} n_{i-1} + L_{i-1}n_i & i = 2 \\ n'_{i-1} + L_{i-1}n_i & i > 2 \end{cases}$$

number of nodes. Let $n = \max_i\{n_i\}$ and $L = \max_i\{L_i\}$, then the resulting tree has at most $n' \leq MLn$ nodes. □

When approximating a RF with a DT the new tree uses copies of each tree in the forest and appends them to its leaf nodes. The new tree, therefore, is larger – or to follow deep learning nomenclature it is *deeper* – which may explain the better performance of RF compared to DT. However, as discussed earlier, this new tree also has a huge Rademacher complexity which should lead to severe overfitting from a theoretical perspective.

While Theorem 6 offers an interesting theoretical insight it does not allow for direct control over different levels of complexity of the resulting decision tree. For the following experiments, we, therefore, use a slight variation of this idea: We randomly sample points along the decision boundary of the RF and then train a decision tree

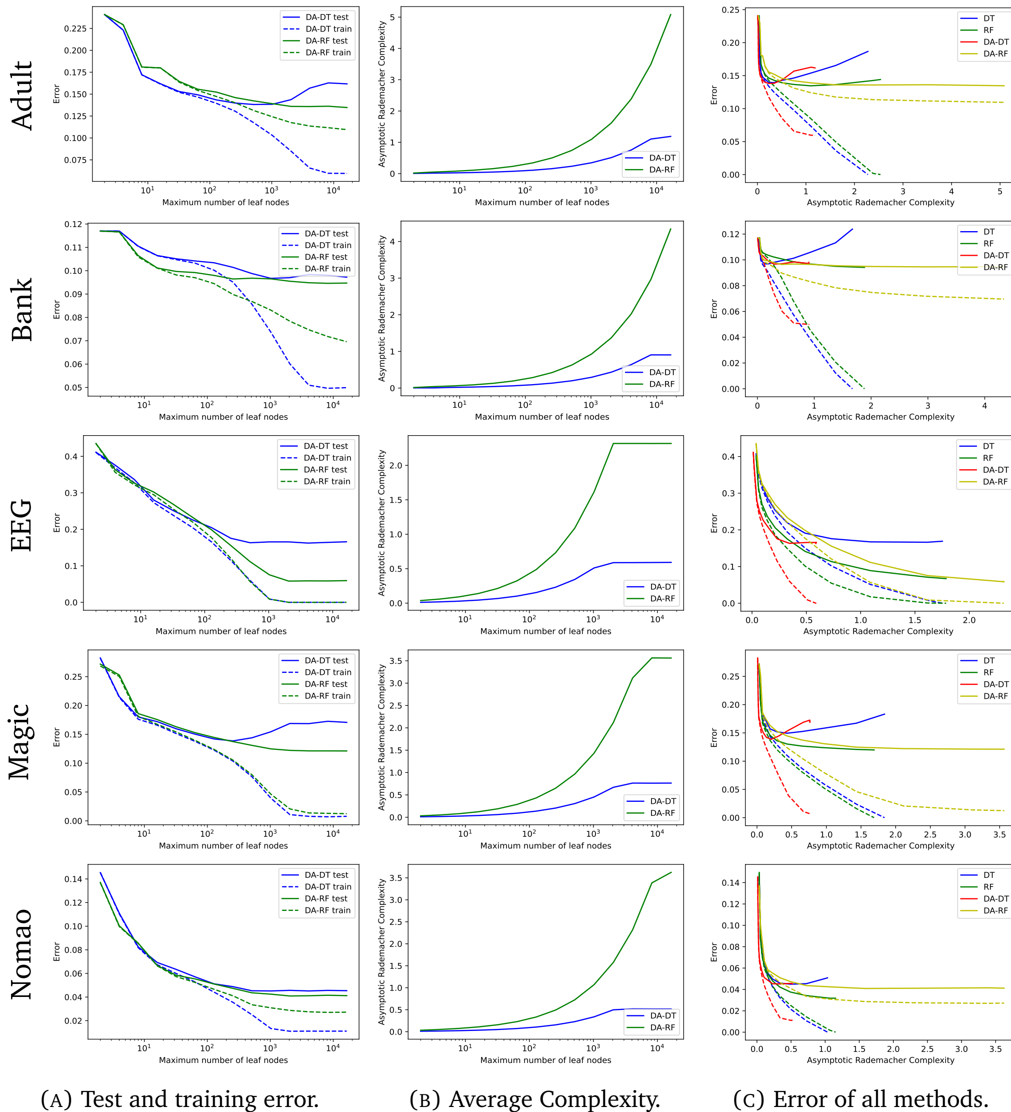
on these points using the predictions of the original forest. This idea is sometimes referred to as born-again trees [BS96]. Clearly, when sampling enough data points we can perfectly represent the decision boundary of the forest with a DT as implied by Theorem 6. However, we could also choose to sample fewer points and/or train a smaller tree with smaller complexity. It is conceivable that this DT 'inherits' the positive properties of the RF and hence it should not overfit. Likewise, we could reverse this approach and sample points from the decision boundary of a DT and train a RF on these points. Again we might hypothesize that such a RF has all the negative properties of the original DT. We call this approach training with data augmentation because we augment the original training data to sample the points along the decision boundary. Algorithm 12 summarizes our approach. We first train a reference model e.g. a regular RF given the original data $\mathcal{S}$ in line 2. Then, we sample $N \cdot T$ points along the decision boundary of the model by using augmented copies of the training data. Specifically, we copy the training data T times and add Gaussian noise to the observations in these copies as shown in line 5. Then we apply the reference model to this augmented data in line 6 and use its predictions as the new label for fitting the actual model in line 8.

Algorithm 12 Training with Data Augmentation.

```
1: function TRAIN_WITH_AUGMENTATION(S, T, ε)
2:     S ← X, y
3:     g ← train_reference_model(S)         ▷ Train a DT or RF
4:     for i = 1, ..., T do
5:         X_i ← X + N(0, ε)                 ▷ Augment data
6:         y_i ← g(X_i)                      ▷ Apply original model
7:         S = S ∪ (X_i, y_i)
8:     f ← train_model(S)                    ▷ Train a RF or DT
       return f
```

We repeat the previous experiment with data augmentation training. Again we limit the maximum number of leaf nodes $n_l \in \{2, 4, 8, \ldots, 16384\}$. We train a RF with $M = 256$ trees and approximate it with a DT using $T = 10$ and $\varepsilon = 0.01$. We call this algorithm DT with data augmentation (DA-DT). Similarly, we train a single DT and approximate it with a RF with $M = 256$ trees, $T = 10$ and $\varepsilon = 0.01$ denoted as RF with data augmentation (DA-RF). Figure 5.3a shows the error curves for this experiment. Again, note the logarithmic scale on the x-axis. First, we see that the training error approaches zero for larger n_l for both models as expected. Second, we see that the decision tree DA-DT despite fitting the decision boundary of a RF shows clear signs of overfitting. Third, and maybe even more remarkable, the forest DA-RF trained via data augmentation on the *bad, overfitted* labels from the DT still does *not* overfit but also has a single descent. To gain a better picture we can again look at the Rademacher complexities of these two models in Figure 5.3b. Similar to before there is a steep increase for both models. However, DA-DT now converges against a *smaller* Rademacher complexity compared to DA-RF which now has a much *larger* Rademacher complexity despite having a better test error. The forest does *not* overfit in an u-shaped curve as expected but also shows a single descent whereas the DT still *does* overfit in an u-shaped curve similar to before.

For a better comparison between the individual methods, we combine them in a single plot. Figure 5.3c shows the asymptotic Rademacher complexity over the test and train error of all methods. The dashed lines depict the training error, whereas the



(A) Test and training error. (B) Average Complexity. (C) Error of all methods.

FIGURE 5.3: Test error (solid lines) and training error (dashed lines), the average Rademacher complexity and the average height of the trees over the maximum number of leaf nodes n_l. Each row depicts one dataset. Results are averaged over a 5-fold cross-validation. Best viewed in color.

solid lines are the test error. Note that some curves stop early because their respective Rademacher complexities are not large enough to fill the entire plot. As one can see, DT and RF have a comparably small maximum Rademacher complexity. RF seems to minimize the training error more aggressively and reaches a smaller error with smaller complexities, whereas DT starts to overfit comparably early. DA-DT seems to have the smallest Rademacher complexity but also overfits the most. DA-RF has the largest complexity but does not seem to overfit at all. It slowly converges against the original RF's performance. Both, DT and DA-DT show an u-shaped curve whereas RF and DA-RF both show a single descent in most cases.

Technical note: Before continuing to the next section a quick note is in order. To the best of our knowledge, there does not exist a tight upper bound for the complexity of trees although the authors of [LLM20] conjecture that their formulation is indeed tight. Moreover, apart from an exhaustive search (such as performed in [AYA] for

small trees), we do not know of a way to compute the exact Rademacher complexity of trees. Hence, a better estimation of the Rademacher complexity could redeem its usefulness for tree ensembles, but we find this unlikely given the current state of knowledge.

5.5 Negative Correlation Forests

Clearly, the Rademacher complexity fails to explain the performance of the data-augmented trees and forests. In this section, we argue that the trade-off between bias and diversity also plays a crucial role as implied by the bias-variance decomposition. This decomposition does not directly relate the training error of a model to its generalization capabilities, but it shows how the individual training and testing losses are structured. Although suspected for some time and exploited in numerous ensembling algorithms, the exact connection between the diversity of an ensemble and its generalization error was only established relatively recently as shown in Theorem 7.

Theorem 7 (PAC-Style C-Bound [GLL+15]). *Let $\mathcal{H} = \bigcup_{j=1}^{k} \mathcal{H}_j$ denote a set of base classifiers and let $f = \frac{1}{M}\sum_{i=1}^{M} h_i$ be the ensemble. Then, for any $\delta > 0$, with probability at least $1-\delta$ over the choice of sample $\mathcal{S}$ of size N drawn i.i.d. according to $\mathcal{D}$, the following inequality holds:*

$$L_0(f) \leq 1 - \frac{\left(\max\left(0, \frac{1}{M}\sum\limits_{i=1}^{M} L_{0,\mathcal{S}}(h_i) - \tau_1(N,\delta)\right)\right)^2}{\min\left(1, \mathbb{V}_{i,\mathcal{S}}\left[h_i\right] + \tau_2(N,\delta)\right)}$$

where $\tau_1(N,\delta), \tau_2(N,\delta) \to 0$ for $N \to \infty$ and any δ and where $\mathbb{V}_{i,\mathcal{S}}\left[h_i\right]$ is the (co-)variance of the ensemble evaluated on the sample S.

Intuitively, this result shows that an ensemble of powerful learners with a small bias that sometimes disagree will be better than an ensemble with a comparable bias in which all models agree. Hence, we may hypothesize that a RF seems to find a good balance between bias and diversity explaining its excellent performance.

The original RF algorithm produces accurate ensembles, but it does not allow precise control of its diversity. To study this hypothesis we now present a RF variation that allows for explicit control over the diversity called Negative Correlation Forest. Theorem 7 is stated for the 0-1 loss which makes the direct minimization of the bound difficult as noted in [GLL+15]. Luckily, the minimization over the bias-variance decomposition of the MSE is much more approachable as discussed earlier. Recall the empirical bias-variance decomposition from Eq. 4.6:

$$\ell_\lambda(f(x), y) = \frac{1}{M}\sum_{i=1}^{M}(h_i(x) - y)^2 - \frac{\lambda}{2M}\sum_{i=1}^{M} {d_i}^T D d_i$$

where $d_i = (h_i(x) - f(x))$, $D = 2 \cdot I_C$ is the $C \times C$ identity matrix with 2 on the main diagonal and $\lambda \in \mathbb{R}$ is the regularization strength. We adapt this approach to RF by first training an initial RF which is then refined by optimizing the GNCL objective. Recall that DTs use a series of axis-aligned splits of the form $\mathbb{1}\{x_k \leq t\}$ and $\mathbb{1}\{x_k > t\}$ where k is a feature index, t is a threshold to determine the leaf nodes and each leaf node has a (constant) prediction $g \in \mathbb{R}^C$ associated with it. Unfortunately, the axis-aligned splits of a DT are not differentiable and thus it is difficult to refine them further with gradient-based approaches. However, the leaf predictions g are simple

constants that can easily be updated via SGD. Formally, we perform SGD to minimize Eq. 4.6 wrt. to $\beta = (\beta_1, \dots, \beta_M)$ where $\beta_i = (g_{i,1}, g_{i,2}, \dots)$ are the parameters for tree h_i. The gradient for a mini-batch $\mathcal{B}$ in this case is:

$$g_{\mathcal{B}}(\beta_i) = \frac{1}{|\mathcal{B}|}\left(\sum_{(x,y)\in\mathcal{B}} \frac{\partial \ell_\lambda(f_\beta(x), y)}{\partial f_\beta(x)} w_i s_{i,l}(x)\right)_{l=1,2,\dots,L_i} \tag{5.8}$$

Algorithm 13 summarizes the NCForest algorithm. First, an initial RF is trained with M trees using at most n_l leaf nodes and d_i features. Then, the leaf predictions are extracted from the forest in line 5 and SGD is performed in line 6 to 10. Finally, the updated leaf values are copied back into the original trees in line 11.

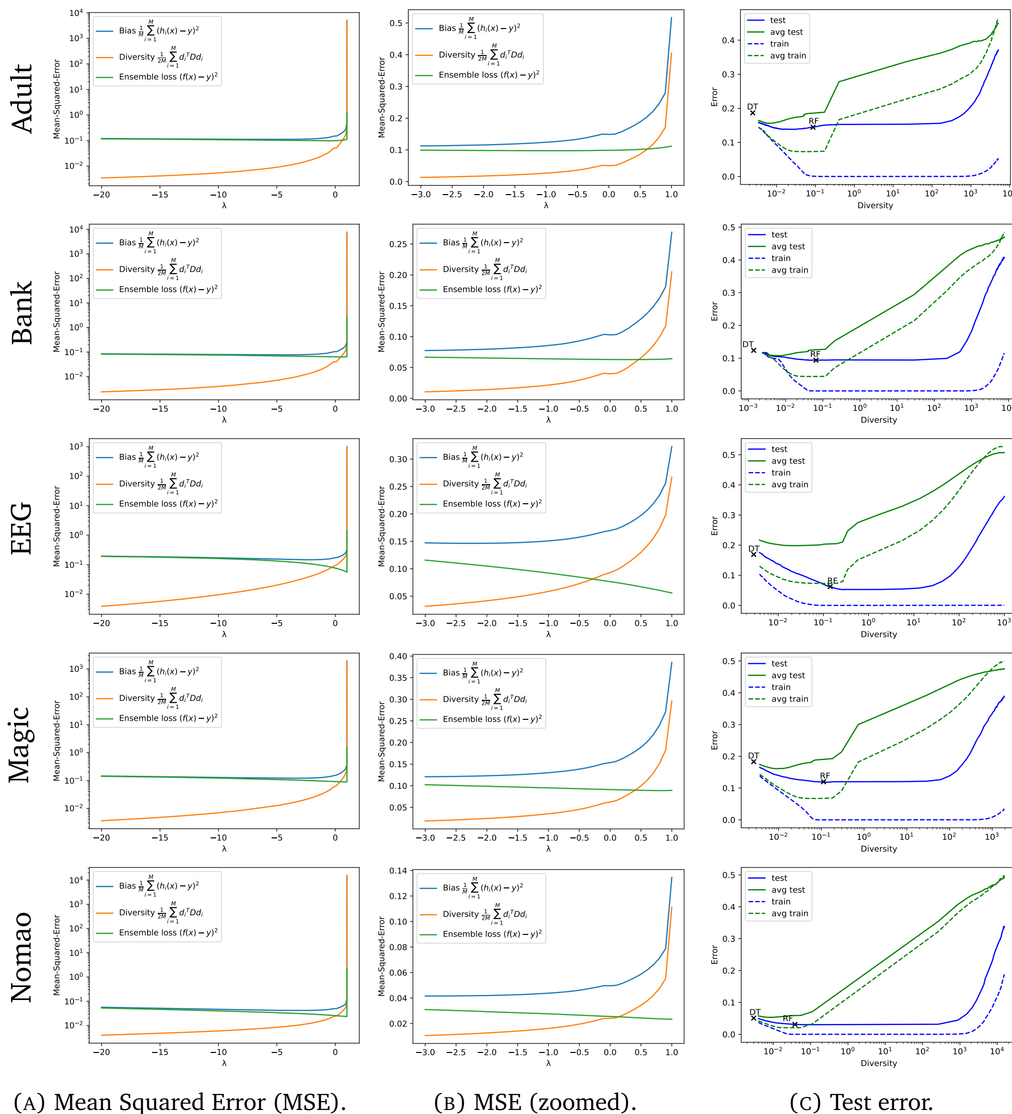
Algorithm 13 Training of a Negative Correlation Forest (NCForest).

```
1:  function TRAIN_NCFOREST(S, M, n_l, d)
2:      h ← train_rf(M, n_l, d)                 ▷ Algorithm 11
3:      w ← (1/M, ..., 1/M)                     ▷ Use constant weights
4:      for i = 1, ..., M do                    ▷ Init. leaf predictions
5:          β_i ← (g_{i,1}, g_{i,2}, ...)
6:      for next epoch do                       ▷ Perform SGD
7:          S ← shuffle(S)                      ▷ Shuffle data before epoch
8:          for B ← next_batch do
9:              for i = 1, ..., M do
10:                 β_i ← β_i − α g_B(β_i)      ▷ Using Eq. 4.6 + Eq. 5.8
11:     for i = 1, ..., M do
12:         h_i.update_leafs(β_i)               ▷ Copy new leafs into original trees
        return h
```

Again we validate our hypothesis experimentally. We train an initial RF with $M = 256$ trees with a maximum number of $n_l = 4096$ leaf nodes. Due to the bootstrap sampling and due to feature sampling, this initial RF already has some diversity. Hence, we use negative λ values to de-emphasize diversity and positive λ values to emphasize diversity. We noticed that between $\lambda = 1.0$ and $\lambda = 1.005$ there is a steep increase in the diversity because it starts to dominate the optimization ([BWT05] reports a similar effect for neural networks). Hence, we vary $\lambda \in \{-20, -19.9, -19.8, \dots, 1.0, 1.001, 1.002, \dots, 1.005\}$ and minimize the NCL objective over 50 epochs using the Adam optimizer with a step size of 0.001 and a batch size of 64 implemented in PyTorch[PGM+19]. We also experimented with more leaf nodes, different λ values, and more epochs but the test error would not improve further with different parameters.

As seen in Figure 5.4a and Figure 5.4b our NCForest method indeed allows for fine control over the diversity in the ensemble. Increasing λ from -20 to 1.0 leads to a larger bias and more diversity in the ensemble while the overall ensemble loss nearly remains constant as expected. Increasing $\lambda > 1.0$ leads to a steep increase in both where the ensemble loss also increases because the diversity starts to dominate the optimization. Looking at Figure 5.4c we can see the average training and testing error of the trees in the ensemble as well as the test and train error of the entire ensemble. Again, dashed lines depict the training error and solid lines are the test error.

Moreover, we marked the performance of a DT and a RF for better comparability[3]. We can see two effects here: First, there seems to be a large region in which the diversity does not affect the ensemble error, but only the individual errors of the trees. In this region the performance of each individual tree changes, but the overall ensemble error remains nearly constant. The corresponding plots are akin to a bathtub: If the diversity is too small or too large, then performance suffers. Only in the middle of both extremes, do we find a good trade-off between both. Second, we find that a RF seems to achieve a good balance between both quantities with its default parameters, although minor improvements are possible on the adult and EEG datasets. We conclude that a good balance between the bias and the diversity of the forest must be achieved. However, there seems to be a large region of similar performances where the *exact* trade-off does not matter.



(A) Mean Squared Error (MSE).

(B) MSE (zoomed).

(C) Test error.

FIGURE 5.4: Mean-Squared error (first column, second column) over different λ values and the test and train error (third column) over different diversities. Results are averaged over a 5-fold cross-validation. Best viewed in color.

[3]A single DT does not have any diversity. For presentational purposes, we assigned a near, but non-zero diversity to it.

Part III

Additive Ensembles and Small Devices

6 | Training Ensembles for Small Devices

So far, we discussed the theoretical properties of ensembles and how to train ensembles of discrete classifiers, but not how to deploy them to small devices. Recall that the main limitation of small devices are memory and computational resources. While discrete classifiers already reduce the computational burden for inferencing, they still require a lot of memory, as we will see later in this chapter (see, e.g., Table 6.3 or Table 6.4). There are two main research directions in literature for reducing the memory consumption of tree ensembles. The first research direction aims to find smaller models, such as Bonsai [KGV17], Decision Jungles [SSK+13] or X-CLEaVER [LNO+18] during training. Adaptions of model compression techniques to train smaller models are also available [CMGS20]. Last, leaf-refinement is a technique that jointly refines the probability estimates in a given tree ensemble (c.f. chapter 5 and [RCWS15]) to improve its performance which can sometimes lead to smaller and better ensembles [RCWS15]. The second approach aims to reduce the memory consumption of a given tree ensemble by post-processing it. 'Classic' decision tree pruning algorithms, as presented in section 2.4.1, as well as more recent adaptions such as cost-complexity forest pruning [KS17], already follow this approach. Ensemble pruning is another post-processing technique that removes unnecessary classifiers from the ensemble [TPV09, ZBS06]. Remarkably, this removal can sometimes result in a better predictive performance [MD97, MS06, LYZ12] leading to smaller and better ensembles at the same time. In this chapter, we present a method that combines both approaches into a unified objective and present a novel algorithm for solving it. To do so, we incorporate L_1 regularization into the leaf-refinement objective and adopt proximal gradient descent to solve this objective.

For the remainder of this chapter, we assume that we have given an *already trained* additive tree ensemble $\{h_1, \dots, h_M\}$ of M axis-aligned decision trees (DT) with

$$f(x) = \sum_{i=1}^{M} w_i h_i(x) \tag{6.1}$$

Recall that a DT partitions the input space $\mathcal{X}$ into d-dimensional hypercubes called leaves and uses independent predictions for each leaf in the tree. To do so, it uses a series of axis-aligned splits of the form $\mathbb{1}\{x_k \leq t\}$ and $\mathbb{1}\{x_k > t\}$ where k is a pre-computed feature index, and t is a pre-computed threshold to determine the leaf nodes. Each leaf node l contains a probability estimate $g_l \in \mathbb{R}^C$ using the class frequencies of the observations from the training points occurring in that leaf node, i.e.

$$h(x) = \sum_{l=1}^{L} g_l s_l(x) \tag{6.2}$$

where s_l is the indicator function that is 1 if x belongs to leaf l and 0 if not. We do not assume that a specific training algorithm was used to train the ensemble, but any additive tree ensemble can be used. For example, the tree ensemble can be a random forest or a forest of boosted decision trees, etc. For simplicity, we assume that each tree in the ensemble is equally weighted. If the forest is weighted (e.g., as in AdaBoost) so that each classifier h_i has a corresponding weight w_i, then we re-scale the individual classifier's predictions to include the weight

$$f(x) = \sum_{i=1}^{M} w_i {h'}_i(x) = \frac{1}{M}\sum_{i=1}^{M} M w_i {h'}_i(x) = \frac{1}{M}\sum_{i=1}^{M} h_i(x) \tag{6.3}$$

In addition to the trained ensemble, we are given a labeled pruning sample $\mathcal{T} = \{(x_i, y_i) \mid i = 1, \dots, N\}$. This sample can either be the original training data used to train f or another pruning data set not related to the training or test data. In this chapter, we will focus on classification problems, but note that our approach is also directly applicable to regression tasks. Moreover, we will focus on random forests (RF), but note that most of our discussion directly translates to other tree ensembles such as Bagging, ExtraTrees, Random Subspaces, or Random Patches.

6.1 Ensemble Pruning

The goal of ensemble pruning is to select a subset of K classifier from $\{h_1, \dots, h_M\}$ that forms a small and accurate sub-ensemble. Formally, each classifier h_i receives a corresponding pruning weight $w_i \in \{0, 1\}$ so that the ensemble's prediction can be expressed as

$$f(x) = \frac{1}{\|w\|_0}\sum_{i=1}^{M} w_i h_i(x) \tag{6.4}$$

where $\|w\|_0 = \sum_{i=1}^{M} 1\{w_i > 0\}$ is the L_0 norm that counts the number of nonzero entries in the weight vector $w = (w_1, \dots, w_M)$. Many effective ensemble pruning methods have been proposed in the literature. These methods usually differ in the specific loss function used to measure the performance of a sub-ensemble and the way this loss is minimized. Tsoumakas et al. give in [TPV09] a detailed taxonomy of pruning methods that was later expanded in [Zho12], which we follow here.

Ranking-based pruning

Early works on ensemble pruning focus on ranking-based approaches that assign a rank to each classifier depending on their individual performance and then select the top K classifier from that ranking. Formally, ranking-based approaches use the following optimization problem

$$\arg\min_{w \in \{0,1\}^M} \frac{1}{N} \sum_{(x,y)\in\mathcal{T}} \sum_{i=1}^{M} w_i \ell(h_i(x), y) \text{ st. } \|w\|_0 = K \tag{6.5}$$

where $\ell\colon \mathbb{R}^C \times \mathcal{Y} \to \mathbb{R}$ is a loss function. To solve this objective, the following approach can be used: First, all individual losses $\ell(h_i(x), y)$ are computed and sorted in decreasing order. Then, the K models with the smallest losses are selected, and their corresponding weights are set to 1. The remaining weights are set to 0. This

makes ranking-based pruning methods appealing since they are very fast, easy to implement, and the optimum is easily obtained. One of the earliest ranking-based pruning method was due to Margineantu and Dietterich, which employ the Cohen-Kappa statistic to rate the effectiveness of each classifier [MD97]. Later, Martinez-Munoz and Suarez propose the use of the cosine similarity to measure how close the ensemble prediction is to the sub-ensemble [MS06]. More recent approaches also incorporate the ensemble's diversity into the selection. Lu et al. propose to measure the individual contribution of each classifier to form a diverse and effective sub-ensemble [LWZB10], and Guo et al. propose to directly maximize the classification margin as well as the diversity of the sub-ensemble [GLL+18].

Mixed Quadratic Integer Programming (MQIP)

MQIP-based pruning methods enhance ranking-based methods by also adding a pairwise loss function that measures the relationship between two classifiers h_i and h_j. Formally, they use the following objective:

$$\underset{w \in \{0,1\}^M}{\arg\min} \frac{1}{N} \sum_{(x,y)\in\mathcal{T}} \left(\alpha \sum_{i=1}^{M} w_i \ell_1(h_i(x), y) + (1-\alpha) \sum_{i=1}^{M} \sum_{j=1}^{M} w_i w_j \ell_2(h_i(x), h_j(x), y) \right) \tag{6.6}$$

$$\text{st. } \|w\|_0 = K$$

where $\alpha \in [0,1]$ models the trade-off between the two losses $\ell_1 \colon \mathbb{R}^C \times \mathcal{Y} \to \mathbb{R}$ and $\ell_2 \colon \mathbb{R}^C \times \mathbb{R}^C \times \mathcal{Y} \to \mathbb{R}$. Here, ℓ_1 is again a loss function that relates the predictions of each classifier to the true label, and ℓ_2 is a loss that relates the predictions of two classifiers $h_i(x)$ and $h_j(x)$ to each other and potentially also to the true label y. Note that MQIP encapsulates ranking-based methods and recovers them for $\alpha = 1$. However, also note that solving MQIP problems can be difficult and often takes much more time compared to, e.g., ranking-based approaches. Originally this approach was proposed by Zhang et al. in [ZBS06], which uses the pairwise errors of each classifier and $\alpha = 0$ (ℓ_1 is not used). Cavalcanti et al. expand this idea in [COMC16] and combine 5 different measures into ℓ_1 and ℓ_2 including the diversity, correlation, kappa-statistic, disagreement, and double-fault measure.

Clustering-based pruning

Another approach for pruning is first to cluster the different models into groups and then select one representative from each group. To do so, let

$$H_i = (h_i(x_1)_1, \ldots, h_i(x_1)_C, h_i(x_2)_1, \ldots, h_i(x_2)_C, \ldots, h_i(x_N)_1, \ldots, h_i(x_N)_C)$$

denote the (stacked) vector of all predictions of classifier h_i on the sample $\mathcal{T}$ with $N \cdot C$ entries. Further, let

$$c(i) = \underset{j=1,\ldots,K}{\arg\min} \{d(\mu_j, H_i)\}$$

be the index of the closest cluster center $\{\mu_1, \ldots, \mu_K\} \subseteq \mathbb{R}^{NC}$ to H_i given a distance function $d \colon \mathbb{R}^{NC} \times \mathbb{R}^{NC} \to \mathbb{R}_+$. Then, clustering-based pruning formally solves the

following optimization problem:

$$\underset{\substack{w \in \{0,1\}^M \\ \mu_1,\ldots,\mu_K \in \mathbb{R}^{NC}}}{\arg\min} \frac{1}{N} \sum_{(x,y)\in\mathcal{T}} \ell\left(\frac{1}{\|w\|_0}\sum_{i=1}^{M} w_i h_i(x), y\right) + \sum_{i=1}^{M} d(\mu_{c(i)}, H_i) \tag{6.7}$$
$$\text{st. } \|w\|_0 = K \text{ and } \forall w_i = 1, w_j = 1, i \neq j : c(i) \neq c(j)$$

Eq. 6.7 has three parts: The first part $\frac{1}{N}\sum_{(x,y)\in\mathcal{T}} \ell\left(\sum_{i=1}^{M} w_i h_i(x), y\right)$ measures the error of the selected sub-ensemble whereas $\sum_{i=1}^{M} d(\mu_{c(i)}, H_i)$ computes the appropriate cluster centers. Last, the constraints combine both parts to select one representative from each cluster. This optimization problem can be solved with existing clustering algorithms in two steps: First, a clustering is obtained, and then representatives are selected from each cluster based on the loss ℓ. For example, Giacinto et al. propose in [GRF00] to use hierarchical agglomerative clustering using the pairwise error probability as the distance. Then, once the clusters have been obtained, they select the most distant representatives from each cluster to form a diverse ensemble. Lazarevic et al. propose to use K-means clustering with the Euclidean distance in [LO01]. In contrast to Giacinto et al., they iteratively remove the least accurate classifier from a cluster until only one classifier is left, which is then included in the sub-ensemble. More recent works on cluster-based pruning also directly include the diversity into the distance measure [ZW19, ZW20].

Ordering-based pruning

Ordering-based pruning orders all ensemble members according to their individual performances as well as their overall contribution to the ensemble and then picks the top K classifier from this list. In this sense, ordering-based approaches are the most general method for ensemble pruning as they allow to minimize the ensemble error directly:

$$\underset{w \in \{0,1\}^M}{\arg\min} \frac{1}{N} \sum_{(x,y)\in\mathcal{T}} \ell\left(\frac{1}{\|w\|_0}\sum_{i=1}^{M} w_i h_i(x), y\right) \text{ st. } \|w\|_0 \leq K \tag{6.8}$$

where $\ell\colon \mathbb{R}^C \times \mathcal{Y} \to \mathbb{R}$ is again a loss function. To do so, ordering-based approaches sort individual classifiers according to their performance and greedily select the tree that minimizes the overall ensemble error the most.
Algorithm 14 depicts the ordering-based optimization approach. First, the classifier with the best individual loss is selected in line 2. Then, line $4-6$ selects the classifier which minimizes ℓ the most given the already selected ensemble $\sum_{j=1}^{M} w_j h_j(x)$. In a way, ordering-based approaches are greedy because they select the model which improves the ensemble the most without considering all different combinations. Ordering-based pruning was also first presented by Margineantu and Dietterich in [MD97], which proposed to greedily minimize the overall ensemble error. A series of works by Martínez-Muñoz, Suárez and others [MMS04, MS06, MHS09] add upon this work proposing different error measures. More recently, theoretical insights from PAC theory and the bias-variance decomposition were also transformed into greedy pruning approaches [LYZ12, JLFW17].

Algorithm 14 Ordering-based pruning.

1: **function** PRUNE_ORDER($\mathcal{T}, h_1, \dots, h_M$)
2: $\quad w \leftarrow (0, \dots, 0)$
3: $\quad i \leftarrow \arg\min_{i=1,\dots,M} \left\{ \frac{1}{N} \sum_{(x,y)\in\mathcal{T}} \ell\left(h_i(x), y\right) \right\}$
4: $\quad w_i \leftarrow 1$
5: $\quad$ **for** $k = 1, \dots, K-1$ **do**
6: $\quad\quad i \leftarrow \arg\min_{i=1,\dots,M} \left\{ \sum_{(x,y)\in\mathcal{T}} \ell\left(\frac{1}{\|w\|_0+1} \sum_{j=1}^M w_j h_j(x) + h_i(x)\right) | w_i \neq 1 \right\}$
7: $\quad\quad w_i \leftarrow 1$
$\quad$ **return** $\{h_i | w_i \neq 0, i = 1, \dots, M\}$

6.2 Leaf-Refinement

In chapter 5 we already discussed the refinement of leaf nodes in a tree ensemble to study the impact of diversity on the ensemble. A similar approach has been proposed in [RCWS15] to obtain smaller and more accurate tree ensembles. Since we can incorporate the ensemble weights into the leaf nodes as described above, this leaf-refinement is a generalization of the re-weighting of ensembles in [AKA+19, SHP19, SH21] making it a very general framework for improving tree ensembles. Before discussing how to combine Leaf-Refinement and ensemble pruning, we will recap the gradient-based refinement used by the NCForest algorithm from chapter 5.

Let $\beta_i = (g_{i,1}, \dots, g_{i,L_i})$ be the probability estimates of all leaf nodes in tree h_i and let $h_{i,\beta_i}(x)$ denote the prediction of tree i using the probability estimates β_i. Further, let $\beta = [\beta_1, \dots, \beta_M]$ be the matrix of all probability estimates of all trees in the ensemble and let $f_\beta(x)$ denote the prediction of the ensemble with estimates β. Then, refinement proposes to minimize a global loss function

$$\beta = \arg\min_{\beta_1,\dots,\beta_M} \frac{1}{N} \sum_{(x_i,y_i)\in\mathcal{T}} \ell\left(\frac{1}{M}\sum_{j=1} h_{j,\beta_j}(x_i), y_i\right) \tag{6.9}$$

This global loss takes all the interactions between individual trees into account to refine the probability estimates in the leaves, but it does not change the structure of individual trees. Hence, it can be easily minimized via stochastic gradient descent (SGD). SGD is an iterative algorithm that takes a small step into the negative direction of the gradient in each iteration by using an estimation of the true gradient

$$\beta \leftarrow \beta - \alpha g_{\mathcal{B}}(\beta) \tag{6.10}$$

where $g_{\mathcal{B}}(\beta)$ is the gradient of ℓ wrt. to β computed on a mini-batch $\mathcal{B}$ and $\alpha \in \mathbb{R}_+$ is the step-size. The gradient for the individual entries β_i is given by the chain rule:

$$g_{\mathcal{B}}(\beta_i) = \frac{1}{|\mathcal{B}|}\left(\sum_{(x,y)\in\mathcal{B}} \frac{\partial \ell(f_\beta(x), y)}{\partial f_\beta(x)} \frac{1}{M} s_{i,l}(x)\right)_{l=1,2,\dots,L_i} \tag{6.11}$$

Algorithm 15 summarizes the Leaf-Refinement (LR) algorithm. First, the original probability estimates from the tree in the leaf nodes are used as an initialization for the parameter vectors β_i in line 2. Then, SGD is performed using Eq. 6.10 and Eq. 6.11 in lines 4-10. The loss function can be chosen for the specific task at hand.

Algorithm 15 Leaf-Refinement (LR)

```
1: function TRAIN_LR(𝒯, h_1, ..., h_M)
2:     for i = 1, ..., M do                          ▷ Init. leaf predictions
3:         β_i ← (g_{i,1}, g_{i,2}, ...)
4:     for next epoch do                             ▷ Perform SGD
5:         for 𝓑 ← next_batch do
6:             for i = 1, ..., M do
7:                 β_i ← β_i − α g_𝓑(β_i)            ▷ Perform update using Eq. 6.11
8:     for i = 1, ..., M do
9:         h_i.update_leafs(β_i)                     ▷ Copy new leafs into original trees
       return h_1, ..., h_M
```

Ren et al. propose in [RCWS15] to use the hinge-loss in combination with a L_2 regularization term similar to the SVM. Let $\lambda \in \mathbb{R}_+$ be a regularization strength, then they propose to minimize

$$\ell_\lambda(f_\beta(x), y) = \lambda \cdot \max(0, 1 - f_\beta(x) \cdot y) + \frac{1}{2}\|\beta\|_2^2 \tag{6.12}$$

where the $\|\cdot\|_2^2$ is the L_2 norm introduced to combat overfitting. In chapter 5 we already discussed how to adapt the negative correlation learning algorithm for leaf-refinement to enforce different levels of diversity:

$$\ell_\lambda(f_\beta(x), y) = \frac{1}{M}\sum_{i=1}^{M}(h_{i,\beta_i}(x) - y)^2 - \frac{\lambda}{2M}\sum_{i=1}^{M} {d_i}^T D d_i \tag{6.13}$$

where $d_i = (h_{i,\beta_i}(x) - f(x))$, $D = 2 \cdot I_C$ is the $C \times C$ identity matrix with 2 on the main diagonal and $\lambda \in \mathbb{R}_+$ is the regularization strength. For $\lambda = 0$, this trains M classifier independently and no further diversity among the ensemble members is enforced, for $\lambda > 0$ more diversity is enforced during training, and for $\lambda < 0$ diversity is discouraged. Last, following our previous discussion, we can also use the MSE (or any other classification loss) directly on the ensemble output without considering its diversity:

$$\ell(f_\beta(x), y) = (f_\beta(x) - y)^2 \tag{6.14}$$

6.3 Combining Leaf-Refinement and Ensemble Pruning

Leaf-Refinement as well as ensemble pruning enable better and smaller tree ensembles. However, both approaches tackle this challenge from a different point of view. Ensemble pruning removes entire trees from the ensemble to reduce its memory consumption and, as a byproduct, improve its predictive performance. Leaf-Refinement, on the other hand, refines the probability estimates in the trees to improve the performance and, as a byproduct, enables the use of smaller forests with similar performance.

This leads to two questions: First, which of the two methods is better suited to deploy tree ensembles to small devices? Second, can we combine both methods to further improve the predictive performance of the forest while having a smaller memory consumption at the same time? Arguably the simplest method to combine

both approaches is to first prune the ensemble and then refine it afterward. However, this method does not consider the interactions between the pruning algorithm and leaf-refinement. It is conceivable that pruning would select different trees if the probability estimates had been refined *before* the pruning process. Similarly, it is conceivable that refinement would compute different leaf values if it had been performed on the unpruned ensemble. We advocate that the selection of trees, as well as the refinement of the corresponding leaf values, should be performed *simultaneously* to find the overall smallest and best ensemble. The key challenge in this scenario is to incorporate the selection of trees into the gradient-based approach of leaf-refinement. In ensemble pruning, each tree either receives weight 0 (not selected) or 1 (selected). Unfortunately, it is difficult to optimize over discrete values $\{0,1\}^M$ with gradient-based approaches because we apply small, non-binary changes to the weights during optimization. One possible approach to solve this dilemma is to relax the constraints and optimize over real-valued weights $w \in \mathbb{R}^M$ in combination with a L_1 regularization penalty that enforces sparsity:

$$\beta, w = \arg\min_{\beta, w \in \mathbb{R}^M} \frac{1}{N} \sum_{(x_i,y_i)\in\mathcal{T}} \ell\left(\sum_{j=1}^{M} w_j h_{j,\beta_j}(x_i), y_i\right) + \lambda \|w\|_1 \tag{6.15}$$

Enforcing sparsity via a L_1 regularization has a long history in data mining and machine learning. Arguably the largest application of it can be found in feature selection via the LASSO and related methods (see e.g. [Tib96, LCW+18]), but also other application areas such as matrix factorization [KS15], neural network pruning [LKD+17], or dictionary learning [JNH15] have been explored.

Objective 6.15 is non-smooth due to the L_1 norm and hence cannot be minimized via SGD directly. We adopt proximal (stochastic) gradient descent to minimize it. Recall, that PSGD is an adaption of SGD that incorporates a projection operation into the updates so that it can cope with non-smooth objectives [PB14]: First, a gradient descent update of the objective function is performed without considering its non-smooth part (e.g. ignoring the L_1 regularizer). Then, a projection operator `prox` is applied to project the updated parameters onto the correct solution considering the non-smooth part of the objective. Let w be the current weight vector and let $g_{\mathcal{B}}(w)_i$ be the gradient of the i-th entry in w *without* considering the L_1 term. Further, let $\mathcal{P}_\alpha$ be the `prox` operator of $\lambda\|w\|_1$ with step size α, then PSGD performs the following updates

$$w \leftarrow \mathcal{P}_\alpha\left(w - \alpha g_{\mathcal{B}}(w)\right) \tag{6.16}$$

using the gradient

$$g_{\mathcal{B}}(w) = \frac{1}{|\mathcal{B}|}\left(\sum_{(x,y)\in\mathcal{B}} \frac{\partial \ell(f_{w,\beta}(x), y)}{\partial f_{w,\beta}(x)} h_{i,\beta_i}(x)\right)_{i=1,\dots,M} \tag{6.17}$$

and the `prox` $\mathcal{P}_\alpha \colon \mathbb{R}^M \to \mathbb{R}^M$ [PB14]:

$$\mathcal{P}_\alpha\left(w\right) = \left(\mathrm{sign}(w_i)\max(|w| - \lambda\alpha, 0)\right)_{i=1,\dots,M}. \tag{6.18}$$

Since there is no regularizer for the leaf nodes we can directly minimize the objective wrt. to β without using the `prox`. In this case the gradient for h_i now also contains

its weights:

$$g_{\mathcal{B}}(\beta_i) = \frac{1}{|\mathcal{B}|}\left(\sum_{(x,y)\in\mathcal{B}} \frac{\partial \ell(f_\beta(x), y)}{\partial f_\beta(x)} w_i s_{i,l}(x)\right)_{l=1,2,\dots,L_i}. \quad (6.19)$$

Algorithm 16 summarizes this approach. Similar to before, the probability estimates in the leaf nodes are used as an initialization for the parameter vectors β_i in line 2. Then, PSGD is performed for multiple epochs using Eq. 6.17, Eq. 6.19, and Eq. 6.18. To do so, the gradient for each weight $g_{\mathcal{B}}(w)_i$ is computed, and a regular weight update is performed in line 7. Similarly, the gradient for the leaf nodes of each tree $g_{\mathcal{B}}(\beta_i)$ is computed in line 8, and a regular gradient descent update is performed. After the leaf nodes of each tree as well as its weights have been updated the `prox` operator is applied in line 10. For $\lambda > 0$, we call this algorithm Leaf-Refinement with L_1 regularization (L1+LR). Setting $\lambda = 0$ and ignoring any weight updates (line 7) recovers the original Leaf-Refinement (LR) algorithm. Similarly, ignoring any updates for the leaf nodes in line 8 yields a new pruning algorithm that selects trees purely based on the L_1 norm which we call L_1 pruning.

Algorithm 16 Leaf-Refinement with L_1 regularization (L1+LR)

```
1:  function PRUNE_AND_REFINE(𝒯, h_1, ..., h_M)
2:      for i = 1, ..., M do                          ▷ Init. leaf predictions
3:          β_i ← (g_{i,1}, g_{i,2}, ...)
4:      for epoch 1, ..., E do                        ▷ Perform PSGD for E epochs
5:          for next batch 𝓑 in epoch do
6:              for i = 1, ..., M do
7:                  w_i ← w_i − α^t g_𝓑(w)_i          ▷ Perform update using Eq. 6.17
8:                  β_i ← β_i − α g_𝓑(β_i)            ▷ Perform update using Eq. 6.19
9:              w ← 𝒫_α(w)                            ▷ Apply the prox using Eq. 6.18
10:     H ← ∅, W ← ∅
11:     for i = 1, ..., M do
12:         if w_i ≠ 0 then
13:             h_i.update_leafs(β_i)                 ▷ Copy new leafs into original trees
14:             H ← H ∪ {h_i}
15:             W ← W ∪ {w_i}
        return H, W
```

6.4 Experiments

In this section, we experimentally evaluate the combination of Leaf-Refinement and Pruning (L1+LR) and compare its performance with vanilla random forests, pruned RFs, and vanilla Leaf-Refinement in the context of IoT. As argued before, our main concern is the final model size as it determines the resource consumption, runtime, and energy of the model application during deployment. Hence, we adopt a hardware-agnostic view and ask the following two questions:

- **Question 1:** What method has the best predictive performance?
- **Question 2:** What method has the best predictive performance under memory constraints?

An overview of all hyperparameters for our experiments is given in Table 6.1. We use the following experimental protocol: The basic idea of ensemble pruning is to overtrain the ensemble first and then prune away unnecessary classifiers from this overtrained pool. Oshiro et al. studied the impact of the number of trees on the performance of a regular RF and showed on a variety of datasets that there is no significant performance improvement when using more than 128 trees [OPB12]. Therefore, we 'overtrain' our base random forests with $M = 256$ trees to increase the classifier pool for pruning without increasing the training time significantly. To control the individual errors of trees we set the maximum number of leaf nodes n_l to values between $n_l \in \{16, 32, 64, 128, 256, 512, 1024, 2048\}$. For pruning, we use COMP, DREP, IC, IE, LMD, RE and task each pruning method to select $K \in \{2, 4, 8, 16, 32, 64, 128\}$ trees from the base forest. For DREP we additionally vary $\rho \in \{0.25, 0.3, 0.35, 0.4, 0.45, 0.5\}$. Finally, for leaf-refinement, we randomly select $K \in \{2, 4, 8, 16, 32, 64, 128\}$ trees from the random forests (which is similar to training a smaller forest directly) and minimize the MSE loss for 50 epochs with a batch size of 1024 using the Adam optimizer [KB15] implemented in PyTorch [PGM+19]. Recall that our L1+LR method indirectly controls the number of trees in the forest through the regularization strength $\lambda \in \{1, 0.5, 0.1, 0.05, 0.01\}$. As discussed previously, we study two variations of our algorithm. In the first version, we do not perform any leaf-refinement, but only select trees using the L_1 norm and call this algorithm L1. In the second version, we combine leaf-refinement with the L_1 regularization as outlined in algorithm 16 and call this algorithm L1+LR. For our experiments, we use 15 publicly available classification datasets with 7 195 to 70 000 examples depicted in Table 6.2. Here, N denotes the total number of data points, d is the dimensionality and C is the number of classes ranging from 2 to 11. The class distribution is also given for each dataset and each class. A dash "-" indicates that the corresponding dataset has fewer classes, e.g. adult has only two classes and hence entries for $C_2 - C_{11}$ are marked with a dash. In all experiments, we perform a 5-fold cross-validation except when the dataset has a dedicated train/test split. We use the training set for both, training the initial forest and pruning it. For a fair comparison, we made sure that each method receives the same forest in each cross-validation run. In all experiments, we use minimal pre-processing and encode categorical features as one-hot encoding. The random forests have been trained with scikit-Learn [PVG+11]. We implemented all pruning algorithms in a Python package for other researchers called PyPruning which is available under https://github.com/sbuschjaeger/PyPruning. The code for these experiments is available under https://github.com/sbuschjaeger/leaf-refinement-experiments. In total, we evaluated 920 hyperparameter configurations per dataset leading to a total of 13 800 experiments.

6.4.1 (Q1) What Method has the Best Predictive Performance?

In the first experiment, we study the predictive performance of pruning and leaf-refinement without considering any memory constraints. To do so, we pick the best hyperparameter configuration of each method that has the best predictive performance. To account for imbalanced datasets (e.g., ida2016) we study the predictive performance in terms of accuracy and the F_1 score.

Table 6.3 shows the accuracy of each method on each dataset with the corresponding model size. The highest accuracy is marked in bold. It can be clearly seen that the combination of Leaf-refinement and L1 regularization (LR + L1) offers the best accuracy on 10 datasets (adult, avila, connect, eeg, elec, fashion, gas-drift,

TABLE 6.1: Configurations used during the experiments. Here, n_l denotes the maximum number of leaf nodes in the random forest and K is the number of estimators in the forest. Each method receives a RF trained with $M = 256$ trees and $n_l \in \{16, 32, 64, 128, 256, 512, 1024, 2048\}$ leaf nodes. Each pruning method is tasked to select $K \in \{2, 4, 8, 16, 32, 64, 128\}$ trees from this forest. ρ denotes the hyperparameter of DREP as outlined in [LYZ12]. λ is the regularization strength as outlined in section 6.3.

Method	Abbreviation	Hyperparameter
Random forest [Bre01]	RF	$n_l \in \{16, 32, 64, 128, 256, 512, 1024, 2048\}$ $M \in \{2, 4, 8, 16, 32, 64, 128, 256\}$
Reduced error pruning [MD97]	RE	$K \in \{2, 4, 8, 16, 32, 64, 128\}$ RF with $M = 256$, $n_l \in \{16, 32, 64, 128, 256, 512, 1024, 2048\}$
Individual contribution pruning [LWZB10]	IC	$K \in \{2, 4, 8, 16, 32, 64, 128\}$ RF with $M = 256$, $n_l \in \{16, 32, 64, 128, 256, 512, 1024, 2048\}$
Individual error pruning [JLFW17]	IE	$K \in \{2, 4, 8, 16, 32, 64, 128\}$ RF with $M = 256$, $n_l \in \{16, 32, 64, 128, 256, 512, 1024, 2048\}$
Complementariness pruning [MMS04]	comp	$K \in \{2, 4, 8, 16, 32, 64, 128\}$ RF with $M = 256$, $n_l \in \{16, 32, 64, 128, 256, 512, 1024, 2048\}$
Largest mean distance pruning [GRF00]	LMD	$K \in \{2, 4, 8, 16, 32, 64, 128\}$ RF with $M = 256$, $n_l \in \{16, 32, 64, 128, 256, 512, 1024, 2048\}$
DREP [LYZ12]	DREP	$\rho \in \{0.25, 0.3, 0.35, 0.4, 0.45, 0.5\}, K \in \{2, 4, 8, 16, 32, 64, 128, 256\}$ RF with $M = 256$, $n_l \in \{16, 32, 64, 128, 256, 512, 1024, 2048\}$
leaf-refinement [RCWS15]	LR	$K \in \{2, 4, 8, 16, 32, 64, 128\}$, MSE loss Adam optimizer, $\alpha = 0.001$, $E = 50$, $\lvert\mathcal{B}\rvert = 1024$ RF with $M = 256$, $n_l \in \{16, 32, 64, 128, 256, 512, 1024, 2048\}$
L1	L1	$\lambda \in \{1, 0.5, 0.1, 0.05, 0.01\}$, MSE loss Adam optimizer, $\alpha = 0.001$, $E = 50$, $\lvert\mathcal{B}\rvert = 1024$ RF with $M = 256$, $n_l \in \{16, 32, 64, 128, 256, 512, 1024, 2048\}$
leaf-refinement + L1	LR + L1	$\lambda \in \{1, 0.5, 0.1, 0.05, 0.01\}$, MSE loss Adam optimizer, $\alpha = 0.001$, $E = 50$, $\lvert\mathcal{B}\rvert = 1024$ RF with $M = 256$, $n_l \in \{16, 32, 64, 128, 256, 512, 1024, 2048\}$

TABLE 6.2: Datasets used for the experiments (available in the UCI repository [DG17]). N denotes the total number of data points, d is the dimensionality and C is the number of classes ranging from 2 to 11. The class distribution is also given for each dataset and each class. A dash "-" indicates that the corresponding dataset has fewer classes, e.g. adult has only two classes and hence entries for $C_2 - C_{11}$ are marked with a dash.

Dataset	N	d	C	Class distribution											
				C_0	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9	C_{10}	C_{11}
adult	32 561	108	2	0.76	0.24	-	-	-	-	-	-	-	-	-	-
anuran	7 195	22	10	0.09	0.48	0.08	0.04	0.07	0.16	0.04	0.02	0.01	0.02	-	-
avila	20 867	10	12	0.41	0.01	0.01	0.03	0.10	0.18	0.04	0.05	0.08	0.00	0.05	0.03
bank	45 211	51	2	0.88	0.12	-	-	-	-	-	-	-	-	-	-
connect	67 557	42	3	0.10	0.25	0.66	-	-	-	-	-	-	-	-	-
eeg	14 980	14	2	0.55	0.45	-	-	-	-	-	-	-	-	-	-
elec	45 312	14	2	0.58	0.42	-	-	-	-	-	-	-	-	-	-
fashion	70 000	784	10	0.10	0.10	0.10	0.10	0.10	0.10	0.10	0.10	0.10	0.10	-	-
gas-drift	13 910	128	6	0.18	0.21	0.12	0.14	0.22	0.13	-	-	-	-	-	-
ida2016	76 000	170	2	0.98	0.02	-	-	-	-	-	-	-	-	-	-
japanese-vowels	9 961	14	9	0.11	0.10	0.16	0.15	0.08	0.10	0.12	0.10	0.09	-	-	-
magic	19 019	10	2	0.65	0.35	-	-	-	-	-	-	-	-	-	-
mnist	70 000	784	10	0.10	0.11	0.10	0.10	0.10	0.09	0.10	0.10	0.10	0.10	-	-
mozilla	15 545	5	2	0.33	0.67	-	-	-	-	-	-	-	-	-	-
postures	78 095	9	5	0.21	0.19	0.21	0.19	0.20	-	-	-	-	-	-	-

japanese-vowels, magic, postures) and is tied for the first place on 3 datasets (anuran, ida2016, mozilla). Only on the magic as well as the bank dataset it seems to underperform. Leaf-refinement (LR) is tied for a first place twice and LMD pruning ranks first on 2 datasets. Somewhat expectantly, random forest seems to underperform and improvements are possible due to leaf-refinement or pruning. However, it is also noteworthy that large improvements seem only to be possible with refinement and not with pruning. For example, RF only achieves 77.32% accuracy on the connect dataset and L1+LR achieves up to 83.95% whereas the best pruning method (here L1) achieve 79.32% accuracy. Table 6.4 shows the F_1 score of each method in each dataset. Again, the best method is marked in bold. Similar to before, L1+LR ranks first on 11 datasets (adult, avila, bank, connect, eeg, elec, fashion, gas-drift, ida2016,mnist, mozilla) and is tied for first place on two datasets (anuran,japanese-vowels,postures). Interestingly, L1+LR now also ranks first on the bank dataset using the F_1 score, which might be explained due to its more imbalanced class distribution. Only on the magic dataset, do we see that L1+LR still underperforms. Expectantly, the model size greatly varies between data sets in both tables, but there is also a sizable difference between the individual methods. RF has arguably the largest models, followed by the various pruning methods whereas LR, as well as L1+LR, seem to have the smallest models, although it is difficult to give a general recommendation here. We will examine the model size in more detail in the next section.

TABLE 6.3: The accuracy and model size of each method on each dataset computed over a five-fold cross-validation or the train/test split of the dataset (if any). Each entry is rounded to the second digit after to decimal point. Each row represents one dataset, and each column is one method. Larger is better. The best accuracy is marked in bold.

dataset	COMP	DREP	IC	IE	L1	L1+LR	LMD	LR	RE	RF
adult	86.99 % 3.12 MB	86.79 % 3.12 MB	86.95 % 3.12 MB	86.87 % 3.12 MB	86.91 % 3.24 MB	**87.25 %** 0.06 MB	86.78 % 12.49 MB	87.01 % 0.09 MB	86.98 % 6.25 MB	86.78 % 24.99 MB
anuran	98.89 % 1.57 MB	98.75 % 6.23 MB	98.89 % 1.56 MB	98.82 % 1.52 MB	98.8 % 6.23 MB	**99.24 %** 6.23 MB	98.75 % 6.23 MB	**99.24 %** 6.23 MB	98.89 % 1.56 MB	98.79 % 3.13 MB
avila	99.48 % 3.36 MB	98.66 % 16.37 MB	99.56 % 3.04 MB	99.55 % 3.03 MB	99.32 % 16.23 MB	**99.87 %** 3.52 MB	98.58 % 32.85 MB	99.77 % 4.05 MB	99.31 % 7.22 MB	98.58 % 32.85 MB
bank	90.44 % 3.12 MB	90.44 % 3.12 MB	90.46 % 6.25 MB	90.54 % 6.25 MB	90.42 % 4.16 MB	90.5 % 0.07 MB	**90.58 %** 3.12 MB	90.42 % 0.09 MB	90.48 % 12.5 MB	90.39 % 24.99 MB
connect	78.64 % 1.81 MB	77.52 % 1.81 MB	78.92 % 1.81 MB	78.77 % 1.81 MB	79.32 % 28.92 MB	**83.95 %** 1.88 MB	77.4 % 3.62 MB	82.88 % 0.9 MB	78.26 % 1.81 MB	77.32 % 3.62 MB
eeg	93.42 % 14.95 MB	93.42 % 14.95 MB	93.42 % 14.95 MB	93.42 % 14.95 MB	93.53 % 14.95 MB	**95.55 %** 5.88 MB	93.42 % 14.95 MB	95.35 % 6.24 MB	93.42 % 14.95 MB	93.42 % 14.95 MB
elec	89.12 % 12.5 MB	88.98 % 24.99 MB	89.27 % 3.12 MB	89.34 % 6.25 MB	89.72 % 24.86 MB	**92.49 %** 14.37 MB	88.98 % 24.99 MB	92.21 % 12.49 MB	89.14 % 6.25 MB	88.98 % 24.99 MB
fashion	87.3 % 28.49 MB	87.1 % 28.49 MB	87.22 % 28.49 MB	87.17 % 28.49 MB	87.25 % 49.2 MB	**89.4 %** 56.99 MB	87.21 % 28.49 MB	89.37 % 56.99 MB	87.09 % 28.49 MB	87.13 % 28.49 MB
gas-drift	99.53 % 0.32 MB	99.42 % 0.7 MB	99.5 % 0.63 MB	99.46 % 0.63 MB	99.39 % 5.64 MB	**99.59 %** 0.59 MB	99.46 % 0.74 MB	99.55 % 0.63 MB	99.46 % 0.36 MB	99.43 % 0.7 MB
ida2016	99.28 % 0.39 MB	99.23 % 1.56 MB	99.26 % 0.78 MB	99.22 % 2.07 MB	99.23 % 4.15 MB	**99.32 %** 3.12 MB	99.25 % 0.78 MB	**99.32 %** 3.12 MB	99.28 % 0.19 MB	99.25 % 2.07 MB
japanese-vowels	97.44 % 4.85 MB	97.14 % 9.66 MB	97.19 % 9.67 MB	97.19 % 9.5 MB	97.14 % 19.35 MB	**98.32 %** 3.3 MB	97.29 % 4.96 MB	98.31 % 3.3 MB	97.14 % 19.35 MB	97.14 % 19.35 MB
magic	87.67 % 16.68 MB	87.67 % 16.68 MB	87.67 % 12.49 MB	87.69 % 8.32 MB	**87.84 %** 12.48 MB	87.58 % 16.44 MB	87.69 % 8.38 MB	87.35 % 16.68 MB	87.67 % 12.49 MB	87.67 % 12.49 MB
mnist	96.53 % 56.99 MB	96.53 % 56.99 MB	96.56 % 28.49 MB	96.53 % 56.99 MB	96.53 % 55.21 MB	**98.05 %** 28.49 MB	96.53 % 56.99 MB	98.03 % 28.49 MB	96.53 % 56.99 MB	96.53 % 56.99 MB
mozilla	95.3 % 3.95 MB	95.27 % 7.86 MB	95.3 % 0.99 MB	**95.53 %** 0.47 MB	95.24 % 7.86 MB	**95.53 %** 0.67 MB	95.27 % 7.86 MB	95.32 % 3.12 MB	95.37 % 0.78 MB	95.27 % 7.86 MB
postures	97.11 % 4.62 MB	97.03 % 18.5 MB	97.2 % 18.5 MB	97.11 % 18.5 MB	97.23 % 36.85 MB	**98.64 %** 29.74 MB	96.96 % 36.99 MB	98.63 % 36.99 MB	97.02 % 9.25 MB	96.97 % 18.5 MB

TABLE 6.4: The F_1 score and model size score of each method on each dataset computed over a five-fold cross-validation or the train/test split of the dataset (if any). Each entry is rounded to the fourth digit after to decimal point. Each row represents one dataset, and each column is one method. Larger is better. The best F_1 score is marked in bold.

dataset	COMP	DREP	IC	IE	L1	L1+LR	LMD	LR	RE	RF
adult	0.8094	0.8066	0.8085	0.8066	0.8062	**0.8137**	0.8066	0.811	0.8081	0.8066
	3.12 MB	24.99 MB	3.12 MB	24.99 MB	10.87 MB	0.29 MB	24.99 MB	0.09 MB	6.25 MB	24.99 MB
anuran	0.9766	0.9706	0.974	0.9745	0.972	**0.9844**	0.9706	**0.9844**	0.9753	0.9715
	3.14 MB	6.23 MB	1.56 MB	1.52 MB	6.23 MB	6.23 MB	6.23 MB	6.23 MB	3.14 MB	3.13 MB
avila	0.9937	0.9872	0.9936	0.9933	0.9921	**0.9979**	0.9867	0.9967	0.9931	0.9867
	6.93 MB	16.37 MB	6.54 MB	3.03 MB	16.23 MB	3.52 MB	32.85 MB	4.05 MB	7.22 MB	32.85 MB
bank	0.7246	0.715	0.7249	0.7263	0.7216	**0.7502**	0.7188	0.7434	0.7283	0.7138
	3.12 MB	6.25 MB	6.25 MB	6.25 MB	24.76 MB	0.34 MB	0.0 MB	0.38 MB	6.25 MB	3.12 MB
connect	0.5472	0.5526	0.5571	0.5541	0.5545	**0.6784**	0.5162	0.6706	0.5436	0.5234
	0.45 MB	0.23 MB	0.91 MB	0.45 MB	28.92 MB	1.88 MB	0.45 MB	28.99 MB	0.45 MB	0.23 MB
eeg	0.9333	0.9333	0.9333	0.9333	0.9344	**0.9549**	0.9333	0.9529	0.9333	0.9333
	14.95 MB	14.95 MB	14.95 MB	14.95 MB	14.95 MB	5.88 MB	14.95 MB	6.24 MB	14.95 MB	14.95 MB
elec	0.888	0.8865	0.8897	0.8903	0.8944	**0.923**	0.8865	0.9201	0.8882	0.8865
	12.5 MB	24.99 MB	3.12 MB	6.25 MB	24.86 MB	14.37 MB	24.99 MB	12.49 MB	6.25 MB	24.99 MB
fashion	0.8714	0.8695	0.8707	0.8703	0.8709	**0.8932**	0.8704	0.8929	0.8693	0.8698
	28.49 MB	28.49 MB	28.49 MB	28.49 MB	49.2 MB	56.99 MB	28.49 MB	56.99 MB	56.99 MB	28.49 MB
gas-drift	0.9949	0.9937	0.9946	0.994	0.9932	**0.9956**	0.9939	0.995	0.994	0.9938
	0.32 MB	0.64 MB	0.63 MB	0.63 MB	5.64 MB	0.59 MB	0.74 MB	0.63 MB	0.36 MB	0.7 MB
ida2016	0.913	0.9053	0.9095	0.9041	0.9058	**0.9189**	0.9075	0.9179	0.9118	0.908
	0.39 MB	1.56 MB	0.78 MB	2.07 MB	4.15 MB	3.12 MB	0.78 MB	0.78 MB	0.19 MB	2.07 MB
japanese-vowels	0.9725	0.9692	0.9696	0.9696	0.9692	**0.9819**	0.9709	**0.9819**	0.9692	0.9692
	4.85 MB	19.35 MB	9.67 MB	4.7 MB	19.35 MB	6.61 MB	4.96 MB	3.3 MB	19.35 MB	9.69 MB
magic	0.8619	0.8619	0.8619	0.8622	**0.8639**	0.8613	0.862	0.8587	0.8619	0.8619
	16.68 MB	16.68 MB	16.68 MB	8.32 MB	12.48 MB	16.44 MB	8.38 MB	16.68 MB	16.68 MB	16.68 MB
mnist	0.965	0.965	0.9654	0.965	0.9651	**0.9804**	0.965	0.9802	0.965	0.965
	56.99 MB	56.99 MB	28.49 MB	56.99 MB	55.21 MB	28.49 MB	56.99 MB	28.49 MB	56.99 MB	56.99 MB
mozilla	0.9453	0.9449	0.9455	0.9482	0.9445	**0.9485**	0.9449	0.9459	0.946	0.9449
	3.95 MB	7.86 MB	0.99 MB	0.47 MB	7.86 MB	0.67 MB	7.86 MB	3.12 MB	0.78 MB	7.86 MB
postures	0.9709	0.9701	0.9718	0.9708	0.9721	**0.9863**	0.9694	**0.9863**	0.9699	0.9695
	4.62 MB	18.5 MB	18.5 MB	18.5 MB	36.85 MB	29.74 MB	36.99 MB	36.99 MB	9.25 MB	18.5 MB

To give a statistically meaningful comparison, we present the results in Table 6.3 and Table 6.4 as a CD diagram [Dem06]. In a CD diagram, each method is ranked according to its performance and a Friedman-Test is used to determine if there is a statistical difference between the average rank of each method. If this is the case, then a pairwise Wilcoxon-Test between all methods checks whether there is a statistical difference between the two classifiers. CD diagrams visualize this evaluation by plotting the average rank of each method on the x-axis and connecting all classifiers whose performances are statistically similar via a horizontal bar. Figure 6.1 shows the corresponding CD diagram for the accuracy (left side) and F_1 score (right side), where $p = 0.95$ was used for all statistical tests. In both cases, we see that L1+LR ranks first place in a single clique meaning that it is the statistically significant best method. On rank 2 we find leaf-refinement in both cases followed by IC, COMP, IE, L1, RE, and LMD. Random forest and DREP switch places in both plots: RF ranks last in terms of accuracy, whereas second to last in terms of F_1 score. Similarly, DREP is second to last place in terms of accuracy but ranks last place in terms of F_1 score. We conclude that pruning and leaf-refinement improve the accuracy over the base random forest in almost all cases confirming the results in the literature. However, leaf-refinement seems to perform better than pruning and larger improvements in terms of accuracy and F_1 are possible when leaf values are refined. Last, the joint selection and refinement of trees via the L1+LR algorithm seem to generally perform best ranking first in both cases thereby supporting our initial hypothesis that both, pruning and

refinement, should be integrated into each other for the best performance.

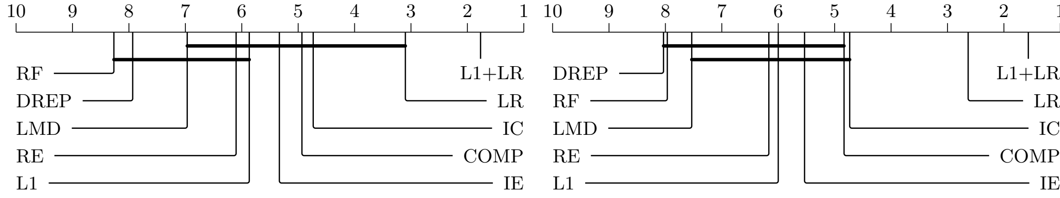


FIGURE 6.1: Critical Difference Diagram for the accuracy (left side) and F_1 score (right side) of the different methods over multiple datasets. For all statistical tests, $p = 0.95$ was used. More to the right (lower rank) is better. Methods in connected cliques are statistically similar.

6.4.2 (Q2) What Method has the Best Predictive Performance under Memory Constraints?

In the second experiment, we study the predictive performance of pruning and leaf-refinement under memory constraints. Recall that small IoT devices are often severely limited in terms of memory (c.f. Table 2.1) and we can only deploy models that fit the available memory. Note that many of the ensembles in Table 6.3 and Table 6.4 already exceed the small memory resources of many IoT devices, thereby making deployment impossible. For the following analysis, we adopt a hardware-agnostic view which assumes that we are given a fixed memory budget for our model, which should, naturally, maintain a state-of-the-art performance. To do so we pick the best hyperparameter configuration of each method that has the best predictive performance while having a model size smaller than $\{128, 256, 512, 1024, 2048\}$ KB. The model size is computed as follows: A baseline implementation (discussed in more detail in chapter 8) of DTs stores each node in an array and iterates over it. Each node inside the array requires a pointer to the left / right child (8 bytes in total assuming `int` is used), a boolean flag if it is a leaf node (1 byte), the feature index as well as the threshold to compare the feature against (8 bytes assuming `int` and `float` is used). Last, entries for the class probabilities are required for the leaf nodes (4 bytes per class assuming `float` is used). Thus, in total, a single node requires $17 + 4 \cdot C$ Bytes per node which we sum over all nodes in the entire ensemble.

We could not find meaningful differences between the F_1 score and the accuracy here and hence we will focus on the accuracy for now and revisit the F_1 score later on. Moreover, we will focus on $\{128, 512, 2048\}$ KB constraints. Additional tables with additional memory constraints as well as the F_1 score are given in the appendix B. Table 6.5 shows the accuracy for model sizes below 128 KB. Contrary to the accuracies without any memory constraints, this table is now more fragmented. L1+LR is the best method on 5 datasets (adult, bank, elec, mnist, mozilla), whereas vanilla LR ranks first on 6 datasets (anuran, connect, eeg, fashion, japanese-vowels, postures). RE pruning is the best option on two datasets (gas-drift, ida2016) and IC is the best option on the magic dataset. Somewhat surprisingly, pruning via L1 did not lead to valid models on many datasets, whereas L1+LR always produces valid models[1]. Going from 128 KB constraints to 512 KB constraints in Table 6.6 L1+LR seems to improve. It now ranks first on 8 datasets (adult, avila, bank, coonnect, eeg, elec, ida2016, japanese-voeels), followed by LR which ranks first on 4 datasets (anuran,

[1] We suspect that L1 and L1+LR require different values for λ to select a similar amount of trees.

fashion, mnist, postures), and IE that ranks first on two datasets (magic, mozilla). COMP ranks first on one dataset and IC shares its first place with IE the magic dataset. This trend continues for larger memory sizes. Table 6.7 shows the accuracy for a constraint of 2048 KB. Here, L1+LR now ranks first on 13 datasets with a performance close to the unconstrained ones in Table 6.3. LR, LMD, and IE each rank first on one dataset.

TABLE 6.5: The accuracy of each method on each dataset computed over a five-fold cross-validation or the train/test split of the dataset (if any) with a model size below 128 KB. A dash '-' indicates that a method did not produce any model that fits the memory constraint. Each entry is rounded to the second digit after to decimal point. Each row represents one dataset, and each column is one method. Larger is better. The best method is marked in bold.

dataset	COMP	DREP	IC	IE	L1	L1+LR	LMD	RE	RF	LR
adult	86.56	86.30	86.59	86.86	85.49	**87.25**	86.18	86.70	85.85	87.01
anuran	97.43	97.29	97.36	97.78	-	97.86	97.01	97.64	97.33	**98.05**
avila	**92.29**	89.82	91.68	91.77	-	78.76	83.39	91.88	87.75	88.80
bank	90.37	90.04	90.16	90.29	89.71	**90.50**	90.10	90.17	89.90	90.42
connect	75.51	75.49	76.15	75.81	69.50	78.11	73.84	75.38	73.67	**78.72**
eeg	87.15	86.68	86.48	87.05	77.45	88.22	85.61	87.52	85.57	**88.50**
elec	85.51	84.46	85.20	85.27	80.55	**86.54**	83.91	85.22	84.34	86.25
fashion	83.20	82.60	83.02	82.73	-	83.67	82.65	83.21	83.01	**84.22**
gas-drift	98.96	98.74	98.81	98.99	-	99.05	98.63	**99.07**	98.68	98.98
ida2016	99.13	99.12	99.13	99.17	98.91	99.08	99.12	**99.18**	99.11	99.16
japanese-vowels	91.11	90.11	91.16	89.41	-	91.46	90.46	91.16	90.40	**92.65**
magic	87.06	86.67	**87.35**	86.88	84.67	87.00	86.77	86.88	86.46	86.57
mnist	90.24	89.64	90.52	89.12	83.57	**92.54**	89.17	90.30	88.74	92.31
mozilla	94.85	94.66	94.79	94.85	-	**94.96**	94.85	94.76	94.60	94.60
postures	85.75	85.70	85.38	85.56	-	77.34	83.91	85.88	84.68	**86.63**

TABLE 6.6: The accuracy of each method on each dataset computed over a five-fold cross-validation or the train/test split of the dataset (if any) with a model size below 512 KB. A dash '-' indicates that a method did not produce any model that fits the memory constraint. Each entry is rounded to the second digit after to decimal point. Each row represents one dataset, and each column is one method. Larger is better. The best method is marked in bold.

dataset	COMP	DREP	IC	IE	L1	L1+LR	LMD	RE	RF	LR
adult	86.95	86.35	86.76	86.86	86.21	**87.25**	86.55	86.84	86.31	87.01
anuran	98.33	98.26	98.12	98.33	94.37	98.64	98.54	98.54	98.29	**98.75**
avila	98.20	94.62	98.06	98.33	66.70	**98.57**	89.33	98.06	95.59	97.53
bank	90.37	90.04	90.19	90.29	90.06	**90.50**	90.25	90.21	90.15	90.42
connect	77.35	77.31	77.57	77.58	73.62	**82.03**	75.58	77.19	75.61	81.67
eeg	90.79	89.52	90.49	90.25	83.26	**92.70**	89.59	89.79	89.67	91.01
elec	86.78	86.59	87.46	87.08	82.76	**89.01**	85.75	86.99	86.32	88.45
fashion	85.10	84.57	84.90	84.54	79.60	85.91	84.61	84.94	84.65	**86.28**
gas-drift	**99.53**	99.35	99.32	99.32	96.23	99.48	99.35	99.46	99.38	99.42
ida2016	99.28	99.22	99.24	99.21	99.03	**99.30**	99.19	99.28	99.22	99.25
japanese-vowels	94.93	93.67	94.28	94.43	82.78	**96.27**	93.67	94.63	93.54	95.82
magic	87.19	86.83	**87.35**	**87.35**	85.83	87.27	87.12	87.04	86.93	86.57
mnist	93.53	92.75	93.44	93.10	87.54	95.58	93.07	93.21	92.68	**95.79**
mozilla	95.27	95.18	95.21	**95.53**	94.27	95.21	94.92	95.34	95.13	95.08
postures	92.67	92.39	92.26	92.55	72.20	92.74	90.95	92.29	91.89	**93.50**

TABLE 6.7: The accuracy of each method on each dataset computed over a five-fold cross-validation or the train/test split of the dataset (if any) with a model size below 2048 KB. A dash '-' indicates that a method did not produce any model that fits the memory constraint. Each entry is rounded to the second digit after to decimal point. Each row represents one dataset, and each column is one method. Larger is better. The best method is marked in bold.

dataset	COMP	DREP	IC	IE	L1	L1+LR	LMD	RE	RF	LR
adult	86.95	86.67	86.92	86.86	86.82	**87.25**	86.55	86.92	86.55	87.01
anuran	98.89	98.54	98.89	98.82	97.50	99.15	98.54	98.89	98.67	**99.17**
avila	99.21	98.24	99.35	99.40	83.65	**99.84**	92.48	99.20	97.68	99.61
bank	90.42	90.36	90.36	90.46	90.33	**90.50**	90.38	90.38	90.29	90.42
connect	78.64	77.52	78.92	78.77	75.34	**83.95**	77.17	78.26	77.03	82.88
eeg	92.19	91.92	92.46	92.09	86.30	**94.86**	91.72	92.22	92.13	93.77
elec	88.90	88.05	89.12	89.24	85.13	**91.84**	87.75	88.87	87.97	89.77
fashion	85.86	86.06	85.91	86.07	83.45	**87.28**	85.97	86.16	85.81	87.24
gas-drift	99.53	99.42	99.50	99.46	98.31	**99.59**	99.46	99.46	99.43	99.55
ida2016	99.28	99.23	99.26	99.22	99.14	**99.33**	99.25	99.28	99.24	99.31
japanese-vowels	96.34	95.78	96.08	96.03	91.32	**98.09**	96.18	96.34	96.04	97.85
magic	87.40	87.06	87.40	87.54	87.05	87.27	**87.61**	87.56	87.21	86.57
mnist	95.12	94.87	95.21	94.99	90.86	**97.35**	94.88	95.00	94.67	97.31
mozilla	95.27	95.24	95.30	**95.53**	94.63	**95.53**	95.14	95.37	95.19	95.25
postures	95.79	95.66	95.92	95.76	84.30	**97.23**	95.06	95.94	95.37	96.93

We conclude that for small model sizes below 128 KB, pruning as well as refinement offer better predictive performance than a vanilla random forest, but it is difficult to give clear recommendations on what method works best in this scenario. We hypothesize that due to the small model size, each method can only pick a few comparably small trees, all with similar performance, and hence, we find similar performances across the methods. Moreover, LR seems to perform slightly better than L1+LR. Once more memory is available, each method can pick more and larger trees, thereby leaving more room for picking 'good' and 'bad' trees. Hence, we see more differences between the individual methods and a clear trend toward refinement. Finally, for larger models with 2048 KB constraints, there is a clear trend towards L1+LR for the best performance.

The difference between vanilla LR and L1+LR for smaller model sizes can be explained by the choice of hyperparameters in this experiment. LR considers $K \in \{2, 4, 8, 16, 32, 64, 128\}$ trees for refinement, whereas L1+LR indirectly chooses the number of trees via λ. We suspect that a more fine-grained hyperparameter selection of λ would have led to a more fine-grained distribution of different model sizes with potentially better performance. Figure 6.2 shows the average number of estimators across all datasets and all configurations selected for different λ values in L1+LR. The error band shows the standard deviation of selected trees across the datasets. As expected, increasing λ leads to a reduction in the number of trees. Between $\lambda = 0.1$ and $\lambda = 0.5$, there is a large drop in the number of estimators from more than 200 to below 100. Similar, but less steep, there is also a significant drop in the average number of estimators from $\lambda = 0.5$ to $\lambda = 1.0$. Hence, choosing additional values for $\lambda \in [0.01, 0.5]$ and $\lambda \in [0.5, 1.0]$ could give a more fair comparison here, and it is conceivable that L1+LR would perform better for smaller model sizes below 128 KB.

Similar to the previous section, we want to give a more statistical overview of our findings using CD diagrams. To do so, we expand them into two-dimensional CD diagrams where we apply memory constraints for each level on the y-axis. In the

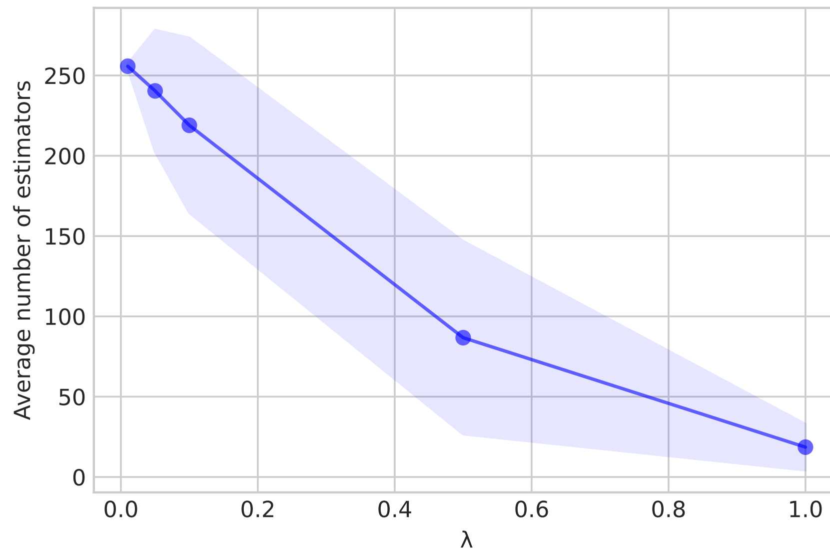


FIGURE 6.2: Average number of estimators across all datasets and configuration of L1+LR for different λ values. The error band shows the standard deviation.

first level, we apply very restrictive constraints, only allowing for models below 128 KB, and plot the average rank of each method. This will likely result in small ensembles of small trees. On the next level, we double the amount of memory allowed to 256 KB and again plot the average rank similar to a 'regular' CD diagram. We repeat this process for all constraints and plot 5 levels with $\{128, 256, 512, 1024, 2048\}$ KB constraints. Figure 6.3 shows the CD diagram for the accuracy. As indicated by the previous discussion, all methods are relatively close to each other if only limited memory is available. LR is the best method, followed by L1+LR, RE, COMP, IE, IC, DREP, LMD, RF, and L1 for 128 KB. As discussed previously, L1 on its own is the worst method for all memory constraints. Going to 256 KB constraints, we see that LR and L1+LR have equal performance while all the other methods seem to differentiate more. This general trend continues, and for 512 and 1025 KB, L1 and L1+LR form a single clique with statistically the best performance, and for 2048 KB constraint L1+LR is the best method on its own. Figure 6.4 shows the CD diagram for the F_1 score. The overall plot is similar to Figure 6.3: If limited memory is available, then it becomes more difficult to distinguish the performance of single methods, whereas, with more memory available, the average ranks seem to differentiate more. Moreover, L1 is the worst method, whereas LR is the best method for 128 KB constraints, and L1+LR is the best method for 512 - 2048 KB constraints. For 256 KB constraints, there is no clear winner, although LR seems to rank slightly better than L1+LR.

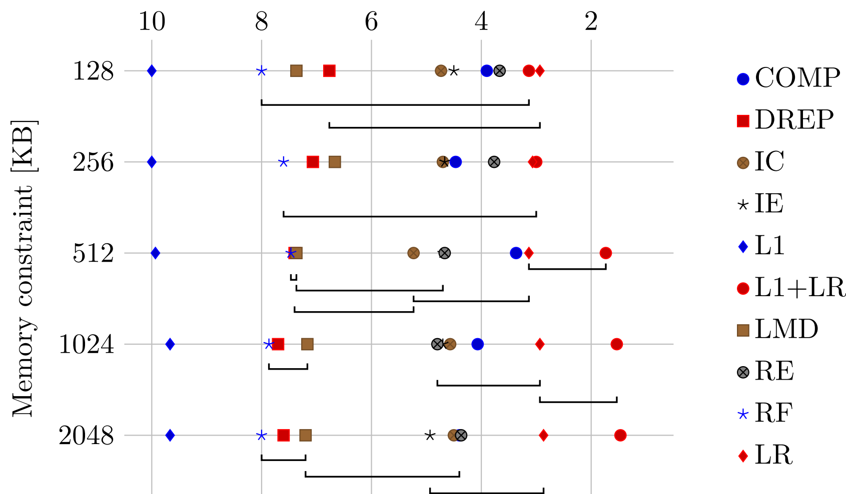


FIGURE 6.3: Two-dimensional critical difference diagram for the accuracy and $\{128, 256, 512, 1024, 2048\}$ KB memory constraints. For all statistical tests, $p = 0.95$ was used. More to the right (lower rank) is better. Methods in connected cliques are statistically similar.

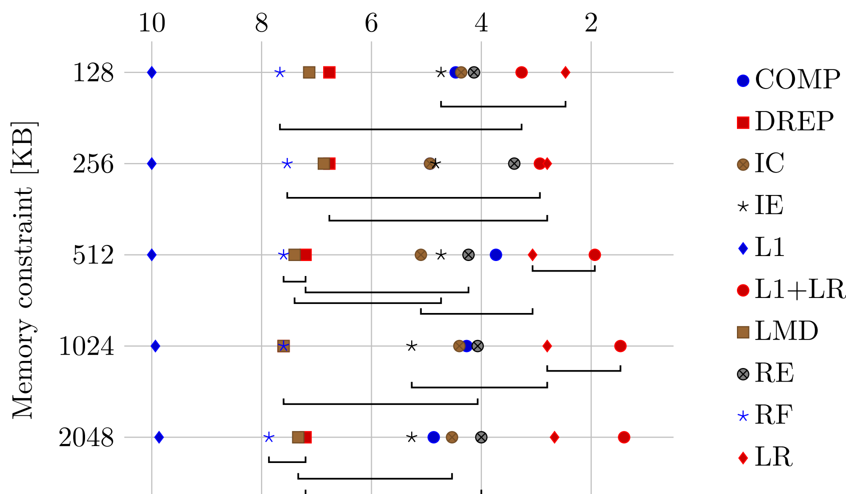


FIGURE 6.4: Two-dimensional critical difference diagram for the F_1 score and $\{128, 256, 512, 1024, 2048\}$ KB memory constraints. For all statistical tests, $p = 0.95$ was used. More to the right (lower rank) is better. Methods in connected cliques are statistically similar.

7 Training Ensembles on Small Devices

The last chapter discussed how to train ensembles for small devices, but with the increasing processing capabilities of small devices, the training of machine learning models directly on small devices also becomes more and more attractive. The advantages are clear: The training happens directly at the edge so no further communication between the edge and a (central) server is required. This not only enhances privacy but also minimizes latency and energy consumption by mitigating communication costs. On the other hand, training on the edge introduces new challenges. For model application, we can focus all available resources on the inference of a model. For model training, however, we additionally have to manage the resources of the training algorithm (e.g., additional buffers or data structures that do not belong to the model itself) as well as the training data itself.

The most natural idea to train ensembles in this context is to adapt the existing batch algorithms to small devices and / or specialized hardware architectures. The benefit of this approach is that training becomes much more efficient while we can often expect the same predictive performance as existing algorithms. For example, there are numerous works in literature that detail how to leverage parallelism in CPUs and GPUs to improve the split computations during DT construction [CG16, PGV+18, KMF+17] without hurting the predictive performance. Similar works also exist in the context of the map-reduce programming model and in distributed environments (see e.g. [LO02, PHBB09, YCCZ09, PR12, dLBH14]). Even more specialized approaches such as [DMD19] discuss how to use bit-level data structures during boosting. Similarly, [GI18] details how to compute large random forests with little resources by pre-partitioning the training data.

Discrete classifiers have the great advantage that they do not require floating-point resources for inference. Similarly, decision tree induction and the ensembling, e.g., via Bagging, can also be implemented with few to no floating-point operations. Hence, a carefully chosen ensemble of discrete classifiers can be trained without any floating-point operations. This makes the available memory of the small device the major limitation. Unfortunately, batch algorithms – by design – expect that the entire dataset is available on the computation device, which further challenges the available memory of the small device. Hence, we will now focus on online learning that consumes one data item after another for the training of ensembles on small devices.

7.1 Online Learning of Tree Ensembles

In online learning, the device consumes one data item at a time, never having access to the entire dataset at once. This has the benefit that we do not need to store the entire dataset on a small device. Additionally, online learning provides a model at

any point during training so that the performance of the model usually improves over time. The disadvantage of this approach is that it often requires entirely new training algorithms with their own theoretical properties that differ from their batch counterparts.

We start by adapting our notation to the online scenario. Let $\mathcal{S} = \{(x_i, y_i) | i \in \{0, \dots\}\}$ be an open-ended sequence of d-dimensional feature vectors $x_i \in \mathbb{R}^d$ and labels $y_i \in \mathcal{Y} \subseteq \mathbb{R}^C$. As before, for classification problems with $C \geq 2$ classes we encode each label as a one-hot vector $y = (0, \dots, 0, 1, 0, \dots, 0)$, having $y_c = 1$ for the assigned label $c \in \{0, \dots, C-1\}$; for regression problems we have $C = 1$ and $\mathcal{Y} = \mathbb{R}$. In this chapter, we focus on classification problems, even though the presented approach is directly applicable to regression, as well. Let $S[t : t+T] = \{(x_t, y_t), \dots, (x_{t+T}, y_{t+T})\}$ denote the sub-sequence of length T starting at element t. Our goal is to maintain a suitable model $f \colon \mathbb{R}^d \to \mathcal{Y}$, which integrates the knowledge of previously observed examples $S[0 : t-1]$, while also offering a good prediction $f(x_t)$ for the next data point x_t *before* the true label y_t is known. There are three crucial challenges in this setting:

- **Computational efficiency:** The algorithm must process examples at least as fast as new examples arrive.
- **Memory efficiency:** The algorithm has only a limited budget of memory and fails if more memory is required.
- **Evolving data streams:** The underlying distribution of the data might change over time, e.g., in the form of concept drift, and the algorithm must adapt to new data trends to preserve its performance.

7.1.1 Online DT Learning

In general, there are three different approaches for online DT induction:

Racing-based DT learning

Racing-based algorithms propose multiple hypotheses and let them race each other for the best performance [MM93, MM97, LN13]. Figure 17 shows this general idea for DT induction: The algorithm starts with initializing the root node. Then it consumes one example after another and accesses the corresponding leaf nodes for each example. Each leaf node contains a list of the possible split hypotheses that monitor their respective performances on the previous samples. Once a selection criterion deems one split significantly better than the other splits, that split is incorporated into the tree, and two new child splits are added. Domingos and Hulten proposed the first online DT induction algorithm called HoeffdingTree (HT) in [DH00]. Before discussing the algorithm, we will present a different variant of Hoeffding's inequality in Theorem 8 that is more useful in this context.

Theorem 8 (Hoeffding's inequality (Variant)). *Let $X_1, \dots, X_N$ be i.i.d random variables with $X_i \in [0, 1]$ $\forall i = 1, \dots, N$. Let $\mu = \frac{1}{N}\sum_{i=1}^{N} X_i$ be the empirical mean, then with probability $1 - \beta$ it holds that:*

$$|\mu - \mathbb{E}[X]| \leq \sqrt{\frac{1}{2N} \log\left(\frac{2}{\beta}\right)}$$

Algorithm 17 Online Training of a decision tree via racing.

```
1:  function UPDATE_TREE(x,y)
2:      if init then
3:          node ← new_node()                ▷ Init. new node
4:          init ← false
5:      else
6:          node ← apply (x)                 ▷ Find the leaf node for x
7:      for h ∈ node.H do
8:          h.update_statistics(y)           ▷ Update statistics for leaf hypothesis
9:      if node.is_converged then            ▷ Check if best hypothesis is converged
10:         node.h ← argmin_{h∈node.H} {e(h)} ▷ Node becomes inner node
11:         node.left ← new_node()           ▷ Add left child
12:         node.right ← new_node()          ▷ Add right child
```

Proof. We start with the original Hoeffding's inequality in Theorem 1 and use $a = 0, b = 1$:

$$
\begin{aligned}
&P\left(|\mu - \mathbb{E}[X]| \geq \varepsilon\right) \leq 2\exp\left(-2N\varepsilon^2\right) \\
\Leftrightarrow\ &P\left(|\mu - \mathbb{E}[X]| \leq \varepsilon\right) \geq 1 - 2\exp\left(-2N\varepsilon^2\right) \\
&\Leftrightarrow |\mu - \mathbb{E}[X]| \leq \varepsilon \text{ with prob. at-least } 1 - 2\exp\left(-2N\varepsilon^2\right)
\end{aligned}
$$

Now define

$$\varepsilon = \sqrt{\frac{1}{2N}\log\left(\frac{2}{\beta}\right)}$$

leading to

$$
\begin{aligned}
&|\mu - \mathbb{E}[X]| \leq \varepsilon \text{ with prob. at-least } 1 - 2\exp\left(-2N\left(\sqrt{\frac{1}{2N}\log\left(\frac{2}{\beta}\right)}\right)^2\right) \\
&|\mu - \mathbb{E}[X]| \leq \varepsilon \text{ with prob. at-least } 1 - 2\exp\left(-\log\left(\frac{2}{\beta}\right)\right) \\
&|\mu - \mathbb{E}[X]| \leq \varepsilon \text{ with prob. at-least } 1 - \beta
\end{aligned}
$$

which concludes the proof. □

The original algorithm by Domingos and Hulten only processes categorical features and uses Hoeffdings bound to rate the performance of each split. To do so, they introduce a split hypothesis for each categorical value of each feature in all current leaf nodes. The author notes that we are only interested in the best split during the race. Hence, it is enough to decide if the (current) best split is *reliably* better than the second-best split. To do so, they propose to use the Hoeffding Bound. More formally, let $s_1 = (k_1, t_1)$ denote the best and let $s_2 = (k_2, t_2)$ denote the second-best split. Let $\bar{\Gamma}(p_1)$ ($\bar{\Gamma}(p_2)$) be the average score $\Gamma(p_1)_i$ ($\Gamma(p_2)_i$) of the class probabilities p_1 (p_2) of

split s_1 (s_2) for the individual observations $i = 1, 2, \dots, T$:

$$\bar{\Gamma}(p_1) = \frac{1}{T}\sum_{i=1}^{T}\Gamma(p_1)_i \qquad \bar{\Gamma}(p_2) = \frac{1}{T}\sum_{i=1}^{T}\Gamma(p_2)_i \tag{7.1}$$

Then, according to Theorem 8 it holds with probability $1-\beta$:

$$|(\bar{\Gamma}(p_2) - \bar{\Gamma}(p_1)) - \mathbb{E}[\Gamma(p_2) - \Gamma(p_1)]| \leq \sqrt{\frac{\log(2/\beta)}{2T}} \tag{7.2}$$

and consequently, if

$$(\bar{\Gamma}(p_2) - \bar{\Gamma}(p_1)) > \sqrt{\frac{\log(2/\beta)}{2T}} \tag{7.3}$$

then with probability $1-\beta$ the split s_1 is surely better (in expectation) than s_2. The authors then continue to show that such a tree will converge against its batch counterpart if trained on the same data.

It is important to recognize that this formulation requires $\bar{\Gamma}(p_2) - \bar{\Gamma}(h_1)$ to be sum-decomposable, which implies that $\bar{\Gamma}(p_1)$ and $\bar{\Gamma}(p_2)$ should be decomposable meaning that they can be estimated via the sum of individual scores. As discussed previously, top-down DTs often use the Gini-Score or Entropy Score. Unfortunately, we cannot apply Hoeffding's-Bound in these cases because both scoring functions are not decomposable. The authors carefully circuit this issue in their motivating introduction of Hoeffding Trees and focus on the classification error instead. Unfortunately, this mathematical rigor did not carry over to either the experiments nor to recent implementations that feature both the Gini-Score and Entropy Score[1] as a splitting criterion. On a bibliographic note, it is interesting to add that the erroneous usage of the Hoffeding Bound was – to the best of our knowledge – first reported by Rutkowski et al. in [RPDJ13]. Rutkowski et al. propose to use McDiarmid's Bound, which does not require decomposability and thus also works with the Gini- and Entropy-score. Unfortunately, Rutkowski et al. do not offer an empirical comparison of their method to Hoeffding Trees. A reason for this might be that Hoeffding Trees simply work well in practice, and thus experiments did not yield significant differences. As discussed in section 2.4.1, a DT does not require optimal, or even 'good' splits to fit the sample $\mathcal{S}$ well as long as there are enough splits with consistent predictions in the leaf nodes. Hence, *any* split is sufficient if training continues, which is usually done in online learning.

The vanilla HT algorithm is only able to handle categorical inputs, and thus multiple extensions have been proposed to generalize it [HSD01, HKP05, PHK07, PHK08, BG09, RPDJ13, MHM21]. Most notably, the overall approach of HTs have been improved by Hoeffding Anytime Trees (HTT) [MWS18]. HTTs greedily select the best split nodes after a few examples before Hoeffding's bound deems it significantly better than the other splits but keeps evaluating *all* split candidates in all nodes. Then, it re-orders the entire tree if the initial greedy choice becomes sub-optimal due to Hoeffding's Bound.

[1] See, e.g., `https://www.cs.waikato.ac.nz/~abifet/MOA/API/annotated.html` or `https://github.com/online-ml/river`

Gradient-based DT learning

The second line of research studies the training of DTs using gradient-based algorithms such as stochastic gradient descent (SGD) [KFCB15, ST15, AİA18, SGW+18, IA21]. Here, the structure of the DT is pre-defined (e.g., its height and the number of leaf nodes) and the objective

$$\arg\min_{s,g} \sum_{t=1}^{T} \ell\left(\sum_{i=1}^{L} g_i(x_t) \prod_{l \in L_i} s_l(x_t), y_t\right) \tag{7.4}$$

is directly minimized over the prediction functions g and splits s. The commonly used axis-aligned splits $\mathbb{1}\{x_k \leq t\}$ are not smooth, which makes the optimization via gradient descent difficult. Thus, a *soft* DT with split-functions $\sigma(z(x))$ (for the children on the right side) and $1 - \sigma(z(x))$ (for the children on the left side) is used. Here σ is the sigmoid function, and $z \colon \mathbb{R}^d \to \mathbb{R}$ is another split function, e.g., a linear function $z(x) = \langle x, w \rangle$. The weight vector w is thereby a part of the optimization objective and determines the features which are relevant in the corresponding split by its nonzero entries. Some approaches introduce sparsity regularization terms for w in order to enforce using fewer features in z [YA14]. Other approaches apply dropout to the tree edges during training [IA21], add the possibility to resize the tree during learning [TAA+19], or map examples into a lower dimensional embedding space for smaller trees [KGV17]. Lastly, these soft decision trees can also be viewed as a specialized deep learning architecture as discussed previously [FH17, BSW16]. Gradient-based approaches naturally enable online learning and benefit from advances in optimization and tooling. However, there are two distinct drawbacks: First, the structure of a DT must be given beforehand, and hence the DT loses its adaptiveness to the data. Second, they require the costly computation of gradients through backpropagation over the *entire* (i.e., due to the soft splits, all paths in the tree must be considered during backpropagation) tree, which again requires floating-point operations. Last, gradient-based DTs suffer from the vanishing gradient problem that additionally slows down convergence [Hoc98].

Sliding Window

Sliding window approaches store a fixed-sized window of observations and then continuously train a new DT on the current window using a regular batch algorithm. Training a single DT on a sliding window has already been proposed in the 1980s in [Kub89]. The FLORA method constructs logical formulas of the form $A_1 \wedge A_2 \wedge \cdots \wedge A_n \implies B$ using so-called rough-sets [Paw82]. Clearly, the above formula also represents a decision tree, even though the training of such a tree does not follow the more established CART or ID3 algorithm. Street and Kim extend this approach by introducing a heuristic that re-trains individual trees of an ensemble on small batches of the data whenever the performance of a classifier deteriorates [SK01]. Unfortunately, we could not find any evidence that this simple baseline has been considered much beyond its original publication or the related variants FLORA2 - FLORA4 [WK96]. More 'recent' papers often train (batched) Naive Bayes, SVM, or KNN over a fixed-sized window [SK05, BG07] but not decision trees, while current work such as [GBR+17, GRB19] does not compare against fixed-sized windows at-all.

7.1.2 Online BNNs

The main training algorithm for BNNs is SGD. As mentioned above, SGD can easily be turned into an online algorithm, e.g., by using the current observation for a gradient step (i.e., batch size 1) or by buffering a batch of examples that is continuously replaced by new observations (e.g., a sliding window). Hence, the training of BNNs naturally fits into an online scenario. However, in contrast to DTs, the gradient computation of BNNs requires floating-point operations, which makes training BNNs on small devices less attractive. To the best of our knowledge, [WDM+21] is the only work that specifically optimizes BNN training for the edge by carefully designing a data structure for the activations of the network.

7.1.3 Online Naive Bayes

Naive Bayes can be directly translated into an online algorithm. Recall that for categorical variables, Bernoulli Naive Bayes treats every category as a one-hot encoding of binary features and computes the counts $f_{i,c}$ of feature i occurring together with class c to estimate the class probabilities $p_{i,c}$. Hence, we can simply update the counts during online learning.

Numerical features are modeled with a Gaussian distribution so that Naive Bayes uses a mean $\mu_{i,c}$ and a variance $\sigma_{i,c}$ for each numerical feature i and class c. Computing the mean is straightforward in an online scenario as it only requires the sum of as well as the total number of instances seen so far. Computing the variance is slightly more complex as it depends on the current mean. Let $x_{i,1}, x_{i,2}, \dots, ...x_{i,T-1}$ be the values of the i-th feature that have been processed so far in the stream and let $X_{i,T}$ be the i-th feature of the current example with the corresponding label y_T. Note that we only need to update the mean and variance estimates for the current label y_T, and all other estimates remain the same. Let $\mu_{i,y,T-1}$ be the mean of feature i and class y_T using the first $N-1$ data points and let $\sigma_{i,y,T-1}$ be the corresponding variance. Then, we can compute the current mean $\mu_{i,y,T}$ and the current variance $\sigma_{i,y,T-1}$ by using Welford's algorithm (c.f. [Knu97], p.232):

$$
\begin{aligned}
\mu_{i,y,T} &= \frac{(T-1)\mu_{i,y,T-1} + X_{i,T}}{T} = \mu_{i,y,N-1} + \frac{x_{i,T} - \mu_{T-1}}{T} \\
M_T &= M_{T-1} + (x_{i,T} - \mu_{i,y,T-1})(x_{i,N} - \mu_{i,y,T}) \\
\sigma^2_{i,y,T} &= \frac{M_T}{T}
\end{aligned}
\tag{7.5}
$$

7.1.4 Online Ensembling

Online GNCL-like algorithms

As mentioned before, GNCL-like algorithms are most often used in combination with neural networks. Hence, they naturally fit into an online learning scenario by leveraging SGD, as discussed above. However, they do not seem to be widely adopted in this scenario beyond the joint training of soft decision tree ensembles in [KFCB15, ST15, SGW+18] or as a general framework for online bagging.

Online Bagging

Bagging can be extended to online scenarios through their connection with GNCL-like algorithms discussed previously. Oza and Russel show in [OR01] that wagging with

weights sampled from a discrete Poisson distribution $w \sim Poisson(1)$ is the same as bagging and use this insight to formulate an online bagging algorithm. This algorithm requires an online learner such as Hoeffding Trees and then simulates bagging by presenting the training example $w \sim Poisson(1)$ times or weighting the loss of the learner with w. This general idea can be extended to various algorithms [Oza05] and more specialized approaches, e.g., specifically designed for concept drift [ASM08, BHP10, HCP11, GBR+17, GRB19]. For an overview, see [GBEB17, KMG+17] and references therein.

Online Boosting

Boosting has also been extended to the online scenario. Similar to online bagging, the basic idea is to use online base learners that can be updated with each new observation. The boosting weights (or losses) are computed on the fly by applying the models one after another, and the individual base learners are updated using their corresponding weights [OR01, RHS04, Oza05, PJVR09]. More specialized variations of this approach, e.g., for concept drift, are also available [KM05]. Again, for a more comprehensive overview, see [GBEB17, KMG+17] and references therein. It is worth mentioning that the theoretical performance of online boosting has remained somewhat elusive because a consistent definition of weak learnability in an online setting is not easy to come by. Chen et al. made the first effort to find a theoretically justified online boosting algorithm in [CLL12] that was later extended in [BHKL15, HSV+17, dCSdB20]. While multiple convergence rates for various online boosting variations now exist, there is – to the best of our knowledge – a general framework for online boosting still missing.

7.2 Shrub Ensembles

Tree ensembles are one of the most popular choices for online learning due to their ability to cope with drift. As mentioned above, the two main strategies to train online tree ensembles are to use incremental base learners such as Hoeffding Trees or gradient-based learning of DTs. The drawback of incremental learners such as HT is that they always keep adding new nodes to the tree without removing old ones. Thus, the size of the trees grows over time which is not suitable for applications on small devices. The drawback of gradient-based learning is that it requires the costly computation of gradients by backpropagation through the entire tree. In order to have a small *and* efficient learning algorithm, we are now re-visiting the training of individual trees over a sliding window.

We propose to maintain a bounded but dynamic ensemble of so-called decision shrubs. Just as in botany, decision shrubs are small- to medium-sized trees which compete against each other. Our algorithm trains shrubs on small windows and uses stochastic proximal gradient descent to learn the weights of individual shrubs in the ensemble. Shrubs with sub-optimal performance are aggressively pruned from the ensemble, while new shrubs are regularly introduced. This makes our algorithm fast and memory-efficient while it retains a high degree of adaptability to evolving data streams. In contrast to incremental learners, our trees never exceed a fixed size. In contrast to gradient-based approaches, we replace costly gradient computations with a continuous re-training of trees on small batches.

As usual, we assume an additive ensemble f of $K = |\mathcal{H}|$ base learners from some model class $\mathcal{H} = \{h \colon \mathbb{R}^d \to \mathbb{R}^C\}$, where K is potentially very large or even infinite.

Each of the K learners $h_i \in \mathcal{H}$ is associated with a weight $w_i \geq 0$ and the ensemble is given by:

$$f(x) = \sum_{i=1}^{K} w_i h_i(x) \tag{7.6}$$

Now, assume that the set of models $\mathcal{H}$ is fixed to a finite set *beforehand* (we will discuss how to adapt $\mathcal{H}$ during optimization later). Then our goal is to learn the optimal weights w_i for each base learner. Since we require a memory-efficient and adaptable algorithm, only $M \ll K$ trees should receive a nonzero weight, and the remaining $K - M$ hypotheses should have a zero weight $w_i = 0$. In this way, the computation of $f(x)$ becomes very efficient since only M instead of K models must be executed for prediction. Additionally, we are free to select another set of M hypotheses if there is a drift in the data, thus retaining the adaptability of the algorithm. Formally, we propose the following optimization objective

$$\arg\min_{w \in \mathbb{R}^K} \sum_{t=1}^{T} \ell\left(f_{S[0:t-1]}(x_t), y_t\right) \text{ s.t. } \|w\|_0 \leq M, w_i \geq 0, \sum_{i=1}^{K} w_i = 1 \tag{7.7}$$

where $M \geq 1$ is the maximum number of ensemble members, $\ell \colon \mathbb{R}^C \times \mathcal{Y} \to \mathbb{R}_+$ is a loss function, $\|w\|_0 = \sum_{i=1}^{K} \mathbb{1}\{w_i \neq 0\}$ is the $0-$norm which counts the number of nonzero entries in w and $f_{S[0:t-1]} \colon \mathbb{R}^d \to \mathbb{R}^C$ is the model at time t. This objective shares some overlap with the ensemble pruning objective in Eq. 6.8, but we now added additional constraints to it. More specifically, the L1+LR algorithm uses the L_1 norm during minimization whereas the objective here contains the L_0 norm. While the L_1 norm worked very well for ensemble pruning it cannot guarantee a maximum ensemble size, but it finds a good trade-off between the ensemble's error and the number of classifiers in it. If this trade-off is unfavorable, e.g., when the ensemble is too large, we can increase λ to force smaller ensembles and run the pruning algorithm again. In the case of on-device training, however, this is not possible. Here, we must ensure that the ensemble does not grow beyond a maximum size at all times during training and deal with the fact that there is no chance of re-running the algorithm. Hence, we use the L_0 norm instead of the L_1 norm to enforce an upper limit of at most M classifiers in the ensemble. For concreteness, we now focus on the (multi-class) MSE loss but note that our implementation also supports other loss functions such as the cross-entropy loss:

$$\ell(f_{S[0:t-1]}(x_t), y_t) = \frac{1}{C} \|f_{S[0:t-1]}(x_t) - y_t\|^2 \tag{7.8}$$

7.2.1 Optimizing the Weights when $\mathcal{H}$ is known

The direct minimization of $L(w, h) = \sum_{t=1}^{T} \ell(f_{S[0:t-1]}(x_t), y_t)$ is infeasible, since $\mathcal{S}$ is open-ended and unknown beforehand. However, we can store small batches $\mathcal{B} = S[t-B : t]$ of the incoming data (e.g. a sliding window) and use them to approximate our objective via the sample mean

$$L_{\mathcal{B}}(w, h) = \frac{1}{BC} \sum_{(x,y) \in \mathcal{B}} \|f(x) - y\|^2. \tag{7.9}$$

The function L is convex and smooth, and its global optimum can be easily derived via its stationary points. However, the feasible set of our constraints

$$\Delta = \left\{ w \in \mathbb{R}_+^K \middle| \sum_{i=1}^{K} w_i = 1, \|w\|_0 = M \right\} \tag{7.10}$$

is not convex, which makes the convex optimization problem to minimize $L(w, h)$ over $w \in \mathbb{R}^d$ a nonconvex problem, minimizing $L(w, h)$ over the feasible set $w \in \Delta$.

As mentioned in chapter 6, proximal stochastic gradient descent (PSGD) is a popular choice to integrate constraints into gradient-based optimization methods. Recall, that PSGD is an iterative algorithm, where every iteration consists of two steps: First, a gradient descent update of the objective function $L_{\mathcal{B}}(w, h)$ is performed without considering any constraint. Then, the `prox`-operator is applied to project its argument onto the feasible set Δ. The proximal gradient update for every iteration is then given as

$$w \leftarrow \mathcal{P}\left(w - \alpha \nabla_w L_{\mathcal{B}}(w)\right), \tag{7.11}$$

where $\alpha \in \mathbb{R}_+$ is the step-size and $\mathcal{P}\colon \mathbb{R}^K \to \Delta$ is the corresponding `prox`-operator. Let us consider an unbiased estimator of the true gradient by using a mini-batch $\mathcal{B}$ of a few examples. By the chain rule we have:

$$\nabla_w L(w, h) \approx \nabla_w L_{\mathcal{B}}(w) = \frac{1}{|\mathcal{B}|} \sum_{(x,y)\in\mathcal{B}} \left(\frac{\partial \ell(f_{S[0:t-1]}(x), y)}{\partial f_{S[0:t-1]}(x)} h_i(x) \right)_{i=1,\dots,K} \tag{7.12}$$

For $\nabla_w L(w, h) = \nabla_w L_{\mathcal{B}}(w)$ (e.g. if $\mathcal{B} = \mathcal{S}$), we obtain the 'regular' proximal gradient descent algorithm. [KBCK13] study the computation of the `prox`-operator for the combination of sparsity requirements and simplex constraints, which define our feasible set. Assuming that the vector w is decreasingly ordered, such that $w_1 \geq \ldots \geq w_K$, they present an operator that sets the $K - M$ smallest entries in w to zero and projects the M largest values onto the probability simplex:

$$\mathcal{P}(w)_i = \begin{cases} 0 & \text{if } i > M \\ [w_i - \tau]_+ & \text{otherwise} \end{cases}$$
$$\text{where } \tau = \frac{1}{\beta}\left(\sum_{i=1}^{\beta} w_i - 1\right), \beta = \max\left\{ j \middle| w_j > \frac{1}{j}\sum_{i=1}^{j}(w_i - 1), j \leq M \right\} \tag{7.13}$$

7.2.2 Optimizing $\mathcal{H}$ simultaneously with the Weights

If the set of classifiers $\mathcal{H}$ is large, then the computation of the weight-gradient can be costly, even when the `prox`-operator ensures that only M models receive a nonzero weight. In turn, a small candidate set $\mathcal{H}$ restricts the possibilities to adapt to the environment such that the model can not adequately react to concept drift. A natural solution to this problem is to drop the assumption that all models in $\mathcal{H}$ are known beforehand and to instead dynamically change $\mathcal{H}$ with new incoming data.

To do so, we maintain at most M trees in the ensemble whose corresponding entries in w are nonzero. In every iteration, we add a new tree to the ensemble and then update the $M + 1$ weights of w. The update ensures that at least one of the $M + 1$ trees obtains a weight of zero, which will be replaced in the subsequent step with a newly trained tree. The new tree will be a part of the ensemble as long as its weight is in subsequent updates not set to zero. Our method *Shrub Ensembles* (SE) is

outlined in Algorithm 18. We start with an empty buffer $\mathcal{B}$ and an empty set of trees $\mathcal{H}$ during the initialization phase. For every new data item, we update the sliding window buffer (lines 4-5), train a new classifier (e.g., via CART), and initialize its weight with zero (lines 6-9). Then, we perform the gradient step followed by the `prox`-operator (lines 10-12). Finally, we remove classifiers with a weight of 0 (line 13). The intuition of our approach is that a newly trained shrub that (significantly) improves the ensemble's prediction will likely receive a large enough weight after the gradient update to survive the subsequent `prox`-operator. If, however, the tree does not improve the ensemble's prediction much it might only receive little gradient mass, such that the tree is removed from the ensemble immediately.

Algorithm 18 Training of a Shrub Ensemble.

```
 1: function UPDATE_SE(x, y)
 2:     if init then
 3:         w ← (0); B ← [ ]; H ← [ ]                ▷ Init.
 4:     if |B| = B then                              ▷ Update batch
 5:         B.pop_first()
 6:     B.append((x, y))
 7:     h_new ← train(B)                             ▷ Add new classifier
 8:     H.append(h_new)
 9:     w ← (w_1, ..., w_M, 0)                       ▷ Initialize weight
10:     w ← w − α∇_w L_B(w)                          ▷ Gradient step using Eq. 7.12
11:     w, H ← sorted(w, H)                          ▷ Sort decreasing order
12:     w ← P(w)                                     ▷ Project on feasible set using Eq. 7.13
13:     w, H ← prune(w, H)                           ▷ Remove zero weights
```

7.2.3 Theoretical Performance of Shrub Ensembles

Theorem 9 formalizes the theoretical behavior of Shrub Ensembles. It shows that, whenever a new, previously unknown relationship (or concept) between observations and labels is discovered, then SE will include the newly trained tree in the ensemble, given an appropriate choice for the step size. In particular, this means two things: First, when $M = 1$ then SE resembles the continuous re-training of trees over a sliding window of fixed size, similar to the previously discussed FLORA algorithm. Second, SE will always incorporate a new concept into the ensemble while keeping track of past concepts, only replacing that tree with the smallest contribution to the entire ensemble. For large step sizes our approach is very aggressive as we introduce a new tree immediately in the ensemble when a single new concept arrives. For very fast-changing data, this can be beneficial, but in some settings, this can hurt the performance, e.g., if the data is very noisy.

Theorem 9 (Adaptability of Shrub Ensembles)**.** *Let $M \geq 1$ be the maximum ensemble size in the shrub ensembles (SE) algorithm and let B be the buffer size. Consider a classification problem with C classes. Further, let $m \leq M$ be the number of models in the ensemble. Now assume that a new observation (x_B, y_B) arrives, which was previously unknown to the ensemble so that $\forall i = 1, \ldots, m\colon h_i(x_B) \neq y_B$. Let SE train fully-grown trees with $h_j(x) \in \{0,1\}^C$ and let h be the new tree, trained on the current window, such that $\forall i = 1, \ldots, B-1\colon h(x_i) = y_i$. Then we have for $\alpha > \frac{BC}{4m}$ the following cases:*

- *(1) If $m < M$, then h is added to the ensemble*

- *(2) If $m = M$, then h replaces the tree with the smallest weight from the ensemble.*

Proof. We start by computing the gradient for weight w with its tree h:

$$\frac{\partial \ell(f(x), y)}{\partial w} = \frac{2}{BC} \left(\sum_{i=1}^{B} \sum_{c=1}^{C} (f(x_i)_c - y_{i,c}) h(x_i)_c \right)$$

Now consider the first case $m < M$. We show, that h receives a non-negative weight $w > 0$ and thus is kept after applying the prox-operator. To do so, we check the weight of h after the gradient step. Recall, that new trees receive an initial weight of $w = 0$ and thus:

$$w = 0 - \frac{2\alpha}{BC} \left(\sum_{i=1}^{B} \sum_{c=1}^{C} (f(x_i)_c - y_{i,c}) h(x_i)_c \right) \overset{?}{>} 0$$

simplifying and reordering lead to

$$\sum_{i=1}^{B} \sum_{c=1}^{C} (f(x_i)_c - y_{i,c}) h(x_i)_c \overset{?}{<} 0$$

$$\sum_{i=1}^{B} \sum_{c=1}^{C} f(x_i)_c h(x_i)_c \overset{?}{<} \sum_{i=1}^{B} \sum_{c=1}^{C} h(x_i)_c y_{i,c}$$

Note that h is a fully grown tree on the current batch so that $h(x_i)_c$ and $y_{i,c}$ are either both 0 or both 1. Thus, it follows that since only one entry in y_i is nonzero that the right side equals B. Now consider the left side. Recall that $h(x_B)_c$ is 1 for the new class j and $f(x_B)_j = 0$ per assumption. Thus, in the 'best' case the ensemble's prediction f and the prediction of h is the same on all but the last example. More formally, $\sum_{c=1}^{C} f(x_i)_c h(x_i)_c = 1$ for each, but the last item in $\mathcal{B}$. It follows that:

$$\sum_{i=1}^{B} \sum_{c=1}^{C} f(x_i)_c h(x_i)_c = B - 1 < B$$

which concludes the proof for the first case.

Now we consider the second case in which $m = M$. Here we must show that after the gradient step the weight w is larger than the smallest weight in the ensemble to replace the corresponding tree. Let w_k be the smallest weight in the ensemble. As noted, h replaces that tree h_k with the smallest weight w_k in the ensemble if $w_k < w$ after the gradient step. Let

$$G_k = \frac{2}{BC} \sum_{i=1}^{B-1} \sum_{c=1}^{C} (f(x_i)_c - y_{i,c}) h_k(x_i)_c$$

be the gradient on the first $B - 1$ examples in the window for the $k-$th member. Now consider the extreme case in which h_k is always correct but f is always wrong. Then G_k is at-least

$$G_k = \frac{2}{BC} \sum_{i=1}^{B-1} \sum_{c=1}^{C} f(x_i)_c h_k(x_i)_c - y_{i,c} h_k(x_i)_c \geq -\frac{2(B-1)}{BC}$$

A similar argument holds for h since we know per assumption that h is correct on the entire batch of the training data. Since $\sum_{i=1}^{m} w_i = 1$ we can further estimate that $w_k \leq \frac{1}{m}$. Thus:

$$
\begin{aligned}
w_k - \alpha G_k - \frac{\alpha}{BC}\sum_{c=1}^{C} 2(f(x_B)_c - y_{B,c})h_k(x)_c \\
&< w_k + \frac{2(B-1)}{BC} - \frac{\alpha}{BC}\sum_{c=1}^{C} 2(f(x_B)_c - y_{B,c})h_k(x)_c \\
&< \frac{1}{m} + \frac{2(B-1)}{BC} - \frac{\alpha}{BC}\sum_{c=1}^{C} 2(f(x_B)_c - y_{B,c})h_k(x)_c \\
&\overset{?}{<} w = 0 + \frac{2(B-1)}{BC} - \frac{\alpha}{BC}\sum_{c=1}^{C} 2(f(x_B)_c - y_{B,c})h(y)_c
\end{aligned}
$$

Subtracting $\frac{2(B-1)}{BC}$ on both sides leads to:

$$
\frac{1}{m} - \frac{2\alpha}{BC}\sum_{c=1}^{C}(f(x_B)_c - y_{B,c})h_k(x)_c \overset{?}{<} -\frac{2\alpha}{BC}\sum_{c=1}^{C}(f(x_B)_c - y_{B,c})h(x)_c
$$

Looking at the left side we note that $f(x_B)_j$ and $h_k(x)_j$ are 0 for the class of y_B and $h_k(x)_c = 1$ for exactly one other class c. Since $f(x_B)_c \leq 1$ we upper bound:

$$
\frac{1}{m} - \frac{2\alpha}{BC}\sum_{c \neq j} f(x_B)_c h_k(x_B)_c < \frac{1}{m} - \frac{2\alpha}{BC}
$$

Looking at the right side we note that $h(x)_j = 1$ whereas the remaining entries are 0. Moreover, $f(x_B)_j = 0$ and $y_{B,c} = 1$ leads to

$$
-\frac{\alpha}{BC}\sum_{c=1}^{C} 2(f(x_B)_c - y_{B,c})h(x)_c = -\frac{2\alpha}{BC}(-1)
$$

Combining both results:

$$
\frac{1}{m} - \frac{2\alpha}{BC} < \frac{2\alpha}{BC} \Leftrightarrow \alpha > \frac{BC}{4m}
$$

which concludes the proof. □

Theorem 9 has two crucial assumptions. First, shrubs are assumed to be fully grown so that they perfectly isolate the points in the current window. Second, the step size must be large enough. Turning this statement around, SE might decide *not* to include a new tree into the ensemble if the step size is smaller than $\frac{BC}{4m}$ or if trees are not perfectly fitting the current batch. In this case, a new concept might be ignored if it only appears a few times in the current window. If, however, the new concept appears multiple times in the window a newly trained tree will likely 'overfit' this new concept and therefore receive a big enough weight to replace one of the other trees. It follows that for large step sizes $\alpha > \frac{BC}{4m}$ and fully-grown trees SE will follow changes in the distribution very quickly, whereas for smaller step sizes and 'smaller' trees it will be more resilient to noise in the data.

7.2.4 Runtime and Memory Consumption of Shrub Ensembles

The continuous training of new models in SE might seem costly, but for reasonable choices of base learners and window sizes, the training is comparably quick. First, the CART and ID3 algorithms require $\mathcal{O}(dN^2 \log N)$ runtime where N is the number of data points. In our case we have $N = B$, that is, the runtime of tree induction is limited by the window size. Second, there are many efficient heuristics and implementations available for decision tree induction that improve the theoretical and practical runtime of tree learning, e.g. by only considering a well-chosen subset of splits [CG16, KMF+17, PGV+18, GEW06]. The computation of the `prox`-operator in Algorithm 18 is in $\mathcal{O}(M \log M)$ [WC13]. The complexity is dominated by the sorting of w. To further decrease the runtime we can maintain a sorted list of w and $\mathcal{H}$ instead of sorting them from scratch (line 11). This can efficiently be done via a binary search tree which only requires $\mathcal{O}(\log M)$ runtime for the insertion and deletion (line 13) of items. Hence, the total complexity of SE is $\mathcal{O}\left(dB^2 \log B + \log M\right)$.

Regarding memory consumption, we note that the size of the trained trees is inherently limited by the window size – regardless of the specific training algorithm. A fully-grown tree that perfectly separates the observations in $\mathcal{B}$ requires at most B leaf nodes. Therefore, the *total* number of nodes used by a tree is upper-bounded by the window size with $2^{\log_2 B+1} - 1 = 2 \cdot B - 1$. It follows that Algorithm 18 stores at most B examples and $M + 1$ models, each having at most $2 \cdot B - 1$ nodes. This makes our shrub ensembles an overall fast and memory-efficient algorithm.

7.3 Experiments

In our experimental evaluation, we are interested in the performance of our shrub ensembles in comparison to recent state-of-the-art methods. We are specifically interested in the accuracy-memory trade-off of these methods. For our analysis, we adopt a hardware-agnostic view which assumes that we are given a fixed memory budget for our model, which should, naturally, maintain a state-of-the-art performance. For racing-based algorithms, we use Online Naive Bayes (NB), Hoeffding Trees (HT), Hoeffding Anytime Trees (HTT), Streaming Random Patches (SRP), Adaptive Random Forest (ARF), Online Bagging (Bag) and Smooth Boost (SB) implemented in MOA [BHKP10]. For gradient-based approaches, we implemented soft decision tree ensembles (SDT) using PyTorch [PGM+19]. For Shrub Ensembles (SE) we used our own `C++` implementation. We compare the performance of each algorithm using the average test-then-train accuracy and the average model size (in kilobyte) on 12 different datasets depicted in Table 7.1.

We measure the model size as the entire model, including any stored variables (e.g., including the sliding window). A careful reader might view this comparison as slightly biased against MOA since it is implemented in Java, whereas the other algorithms are implemented in `C++` (with a Python interface). Unfortunately, there is currently no alternative, efficient MOA implementation available. Preliminary projects to implement MOA in `C++`[2] or Python[3] have not been finalized, yet. Thus, we put effort into making this comparison fair by computing the reference size of each MOA model first (before it received any data points), which is then subtracted from the

[2]`https://github.com/huawei-noah/streamDM-Cpp`

[3]The authors of `https://riverml.xyz/` confirmed that they currently strive for functional and 'feature complete' code and perform optimizations later on during development.

TABLE 7.1: Characteristics of employed datasets. The left group shows real-world datasets with unknown concept drift (available in the UCI repository [DG17]) and the right group shows artificial datasets with synthetic drift (available as part of MOA [BHKP10]).

Dataset	N	d	C
gas-sensor	13 910	128	5
weather	18 159	8	2
nomao	34 465	174	2
elec	45 312	14	2
airlines	539 383	614	2
covertype	581 012	98	7

Dataset	N	d	C
agrawal_a	1 000 000	40	2
agrawal_g	1 000 000	40	2
led_a	1 000 000	48	10
led_g	1 000 000	48	10
rbf_f	1 000 000	10	5
rbf_m	1 000 000	10	5

measurements. This way, we only account for changes in the model due to new items and do not include the 'static' overhead of Java.

To ensure a fair comparison between the hyperparameter choices of each individual algorithm, we follow the methodology presented in [BB12]. In a series of preliminary experiments, we identify reasonable ranges for each hyperparameter and method (e.g., number of trees in an ensemble, window size, step sizes, etc.). Then, for each method and dataset, we sample at most 50 random hyperparameter configurations from these ranges and evaluate their performance. If fewer configurations were available (e.g. Online Naive Bayes has no hyperparameters), then we sample all available configurations. Figure 7.1 gives an example of the hyperparameter optimization. For each method, a dictionary of hyperparameters has been defined. The `Var` keyword marks a variation of the corresponding hyperparameter which randomly selects one of the values from the list. For example, the depicted configuration might generate a SE model with $M = 128$ trees, trained on a window size of 2^{10} data points with a step size of 0.1 where each tree has a max depth of 12 which is randomly trained on d features. The detailed list of hyperparameters can be found in the source code. To ensure timely and realistic results, we remove each configuration that took longer than two hours. In summary, we test 312 different configurations per dataset totaling $3\,744$ experiments. For the experiments, we used a cluster node with 256 AMD EPYC 7742 CPUs and 1TB ram in total. The code for these experiments is available under https://github.com/sbuschjaeger/se-online.

```
"M": Var([4,8,16,32,64,128,256]),
"window_size": Var([2**i for i in range(4, 14)]),
"step_size": Var([1e-4,1e-3,1e-2,1e-1,2e-1,5e-1]),
"additional_tree_options" : {
        "max_depth": Var([2,4,8,12,15]),
        "splitter": Var(["train","random"]),
        "max_features": Var([d, np.sqrt(d)])
}
```

FIGURE 7.1: Example of a hyperparameter configuration for Shrub Ensembles (SE). The `Var` keyword marks a variation of the corresponding hyperparameter which randomly selects one of the values from the list. All trees receive the same hyperparameter configuration. Please consult the source code for the exact parameter choices for all methods.

7.3.1 Quantitative Analysis

As mentioned before, we are interested in the most accurate models with the smallest memory consumption. Clearly, these two metrics can contradict each other. In our first experiment, we assume that there are no memory restrictions at all. Table 7.2 displays the test-then-train accuracy, the average model size (in KB), and the total runtime (in seconds). The highest accuracies are highlighted and the fastest runtime and smallest memory consumption are marked in italic. We notice, that we obtain in comparison to [GRB19] which uses similar methods and the same datasets, slightly better results for all algorithms, which can be attributed to our more expensive hyperparameter optimization. SE, SRP, ARF, and Bagging generally offer the best accuracy. SRP offers the highest accuracy in 5 cases, followed by SE and ARF, performing best on 3 datasets. Bagging attains the highest accuracy on the airlines dataset. Looking at the size, however, we see a different picture. Most of the well-performing methods in terms of accuracy (SRP, ARF, and Bagging) generally consume the most memory, ranging in the order of megabytes, up to the hundreds. SDT, HT, and HTT place themselves in the middle, consuming a few hundred kilobytes to a few megabytes, whereas SE and NB generally consume the least memory (≤ 1 MB). Likewise, looking at the running time, we observe that SRP, ARF, SB, and Bagging require the most time, although SDT shows here some notable outliers. HT, HTT, and NB seem to be the quickest methods whereas SE sometimes is very fast and sometimes it is slower than HT or HTT. We conclude that SE is among the state-of-the-art algorithms for online learning while having a similar resource consumption as Online Naive Bayes, making it an ideal choice for small devices.

In the second experiment, we assume that we are given a limited memory budget, and we exclude every method which exceeds this limit. Table 7.3 shows the results when a maximum of 10 MB per model is allowed and Table 7.4 shows the results if only 1 MB is available. The best method is again highlighted for each dataset. A dash '-' indicates that an algorithm did not produce a model with a size in the given bounds for any hyperparameter configuration. For space reasons, we exclude the specific model size and runtime from the tables.

Looking at Table 7.3, we first notice that SE now also offers the best performance on the agrawal_a and agrwal_g dataset because neither ARF nor SRP managed to produce models which use less than 10 MB. Interestingly, HTT suddenly becomes the best method on the airlines dataset, and ARF manages to become the best method on the led_a and led_g datasets.

We see that this effect amplifies if only 1 MB is available in Table 7.4. Now, SE is the best method on all datasets except the weather dataset because ARF, SRP, SB, and Bagging do not produce valid models on most datasets. Only SDT, NB, and SE manage to consistently stay below 1 MB. We also performed experiments with more aggressive constraints, such as 128 KB but noticed that most algorithms except NB and SE would fail in this setting while SE generally outperformed NB. We conclude that SE or NB are well-prepared for resource-constraint devices, consistently producing small models. Looking at the accuracy of both approaches, we see that SE is the clear winner on all but one dataset. SE is able to produce excellent models while meeting even more aggressive resource constraints making it ideally suited for small, resource-constraint systems.

To summarize the algorithm's performance, we now compute the area under the Pareto front (APF) that contains those parameter configurations that are not dominated across one or more dimensions. For better comparability, we normalize the APF by the biggest model for the given dataset. We are specifically interested in the APF

TABLE 7.2: Best test-then-train accuracy for each method on the given datasets without any memory constraints.

dataset		ARF	Bag	HT	HTT	NB	SB	SDT	SE	SRP
gas-sensor	accuracy	92.674	91.608	84.798	92.535	72.898	76.023	17.164	**96.787**	95.715
	size [kb]	1796	5040	355	1041	67	6090	103	*1*	23650
	time [s]	34	30	14	17	14	112	1968	*8*	134
nomao	accuracy	98.566	97.760	94.733	96.871	92.792	96.889	88.628	**98.660**	98.558
	size [kb]	9292	90717	1091	411	42	12159	*7*	10	15997
	time [s]	148	637	26	28	28	147	97	*21*	204
elec	accuracy	91.057	89.774	86.452	87.748	76.165	88.372	42.514	**94.012**	91.415
	size [kb]	5727	10640	254	234	3	2113	4	*1*	18882
	time [s]	117	72	17	21	25	56	119	*12*	258
weather	accuracy	78.328	78.380	75.112	76.592	69.584	76.954	32.870	75.860	**79.192**
	size [kb]	22868	6034	96	95	*2*	794	106	333	4012
	time [s]	95	36	*8*	9	10	23	910	379	27
covtype	accuracy	93.647	92.256	86.460	88.688	64.591	85.236	3.742	92.844	**94.297**
	size [kb]	34204	1058285	11876	10432	45	51600	*10*	23	59589
	time [s]	2805	19558	*502*	611	589	2150	4583	1043	4527
airlines	accuracy	69.450	**69.501**	67.004	68.756	66.969	69.867	38.771	68.176	69.067
	size [kb]	203557	314285	49390	16826	113	348796	*60*	91	96996
	time [s]	5899	6801	801	1038	*780*	5202	3041	1413	3818
agrawal_a	accuracy	**94.413**	93.592	92.439	91.716	74.288	93.455	9.771	93.551	94.316
	size [kb]	102889	216892	5909	1537	8	45545	*5*	73	121708
	time [s]	2066	2430	*426*	495	567	1767	2782	803	2741
agrawal_g	accuracy	**92.277**	90.192	87.589	87.707	74.276	91.189	45.478	91.567	91.894
	size [kb]	132295	370692	10809	2807	8	61971	*5*	79	194785
	time [s]	2382	3176	*454*	497	563	1876	2601	827	3313
led_a	accuracy	73.789	72.277	68.849	71.480	64.417	72.754	10.112	72.329	**73.855**
	size [kb]	99900	49406	1154	1936	29	18363	*5*	320	45368
	time [s]	4528	3452	*1061*	1201	1192	4361	4567	1628	3256
led_g	accuracy	73.036	72.114	68.634	71.406	64.401	72.495	0	71.440	**73.109**
	size [kb]	115453	83044	1146	2407	29	18429	*5*	324	184343
	time [s]	5401	5128	*1089*	1220	1231	3727	4716	1602	6713
rbf_f	accuracy	75.788	56.659	38.186	40.326	29.747	42.529	7.246	73.028	**77.928**
	size [kb]	11285	629872	1644	4599	5	15575	*4*	241	35487
	time [s]	3109	9081	*542*	585	708	1469	4519	1271	7079
rbf_m	accuracy	**86.376**	81.335	62.963	69.513	32.948	65.467	16.672	79.632	86.134
	size [kb]	28217	589415	2726	5013	5	18409	*2*	916	73123
	time [s]	3251	5754	*547*	609	710	1537	2653	6468	7754

TABLE 7.3: Best test-then-train accuracy for each method on the given datasets with model size below 10 MB. The best method is depicted in bold for each dataset. A dash '-' indicates that a method did not have a model with a smaller size than the constraint in any hyperparameter configuration.

dataset	ARF	Bag	HT	HTT	NB	SB	SDT	SE	SRP
agrawal_a	-	93.029	92.439	91.716	74.288	92.879	9.771	**93.551**	-
agrawal_g	-	87.399	87.356	87.707	74.276	88.817	45.478	**91.567**	-
airlines	-	-	-	**68.491**	66.969	-	38.771	68.176	-
covtype	**93.194**	-	86.408	88.505	64.591	82.922	3.742	92.844	92.958
elec	91.057	89.122	86.452	87.748	76.165	88.372	42.514	**94.012**	91.020
gas-sensor	92.674	91.608	84.798	92.535	72.898	76.023	17.164	**96.787**	95.617
led_a	**73.501**	70.518	68.849	71.480	64.417	72.331	10.112	72.329	73.042
led_g	**72.680**	70.266	68.634	71.406	64.401	72.090	-	71.440	71.764
nomao	98.566	97.315	94.733	96.871	92.792	96.451	88.628	**98.660**	98.515
rbf_f	75.744	45.391	38.186	40.326	29.747	41.210	7.246	73.028	**77.444**
rbf_m	**85.030**	69.880	62.963	69.513	32.948	63.004	16.672	79.632	84.908
weather	78.244	78.380	75.112	76.592	69.584	76.954	32.870	75.860	**79.192**

TABLE 7.4: Best test-then-train accuracy for each method on the given datasets with model size below 1 MB. The best method is depicted in bold for each dataset. A dash '-' indicates that a method did not have a model with a smaller size than the constraint in any hyperparameter configuration.

dataset	ARF	Bag	HT	HTT	NB	SB	SDT	SE	SRP
agrawal_a	-	-	-	90.539	74.288	-	9.771	**93.551**	-
agrawal_g	-	-	-	85.846	74.276	-	45.478	**91.567**	-
airlines	-	-	-	-	66.969	-	38.771	**68.176**	-
covtype	-	-	-	-	64.591	-	3.742	**92.844**	-
elec	90.453	87.839	86.452	87.748	76.165	85.965	42.514	**94.012**	-
gas-sensor	92.336	81.173	84.798	92.177	72.898	73.471	17.164	**96.787**	93.815
led_a	-	-	-	-	64.417	-	10.112	**72.329**	-
led_g	-	-	-	-	64.401	-	-	**71.440**	-
nomao	-	-	94.564	96.871	92.792	-	88.628	**98.660**	-
rbf_f	69.554	-	36.207	35.618	29.747	-	7.246	**73.028**	-
rbf_m	-	-	55.465	-	32.948	-	16.672	**79.632**	-
weather	-	77.771	75.112	76.592	69.584	76.954	32.870	75.860	**77.806**

TABLE 7.5: Normalized area under the Pareto front (APF) for each method and each dataset with models smaller than 100 MB. Rounded to the fourth decimal digit. Larger is better. The best method is depicted in bold.

	ARF	Bag	HT	HTT	NB	SB	SDT	SE	SRP
agrawal_a	0.8259	0.8939	0.9145	0.9136	0.7429	0.9124	0.0977	**0.9355**	0.8877
agrawal_g	0.7738	0.8535	0.8601	0.8732	0.7427	0.8759	0.4548	**0.9157**	0.8571
airlines	0.5679	0.3461	0.5654	0.6588	0.6693	0.5048	0.3877	**0.6818**	0.5369
covtype	0.9274	0.8157	0.8487	0.8712	0.6458	0.8196	0.0374	0.9284	**0.9285**
elec	0.9081	0.8946	0.8641	0.8771	0.7616	0.8821	0.4251	**0.9401**	0.9091
gas-sensor	0.9238	0.9120	0.8474	0.9217	0.7287	0.7580	0.1716	**0.9679**	0.9537
led_a	0.7149	0.7050	0.6846	0.7095	0.6441	0.7199	0.1011	0.7233	**0.7240**
led_g	0.7085	0.7034	0.6825	0.7089	0.6439	0.7174	-	**0.7144**	0.7124
nomao	0.9793	0.9613	0.9443	0.9668	0.9277	0.9606	0.8863	**0.9866**	0.9752
rbf_f	0.7550	0.5171	0.3807	0.4014	0.2975	0.4217	0.0725	0.7302	**0.7739**
rbf_m	**0.8541**	0.7566	0.6262	0.6803	0.3295	0.6478	0.1667	0.7962	0.8479
weather	0.7788	0.7827	0.7510	0.7658	0.6958	0.7688	0.3287	0.7586	**0.7895**

in the context of small devices. From Table 2.1, it can be seen that the largest small devices offer around 256 MB - 1 GB of RAM. A standard Linux environment (e.g., Raspbian) requires around 50 - 150 MB[4] which leaves roughly $100 - 200$ MB for the model on a Raspberry Pi A+. Hence, we will now focus on models with less than 100 MB. Table 7.5 shows the normalized area under the Pareto front. It can be seen that SE and SRP generally perform best, followed by ARF. Our SE method ranks first on seven datasets offering the best accuracy-memory trade-off, followed by SRP, which ranks first on four datasets. In the third place, we find ARF, which ranks first on the rbf_m dataset. SDT does not perform well. We hypothesize that this is due to the random initialization combined with the vanishing gradient problem. On the led_g dataset, it did not finish the computation in under two hours and thus was removed for our evaluation.

To give a statistically meaningful comparison, we again present the results in Table 7.5 as a CD diagram [Dem06]. Recall that in a CD diagram, each method is

[4] https://www.arrow.com/en/research-and-events/articles/raspberry-pi-4-ram-options

ranked according to its performance, and a Friedman-Test is used to determine if there is a statistical difference between the average rank of each method. If this is the case, then a pairwise Wilcoxon-Test between all methods checks whether there is a statistical difference between the two classifiers. CD diagrams visualize this evaluation by plotting the average rank of each method on the x-axis and connecting all classifiers whose performances are statistically similar via a horizontal bar. Figure 7.2 shows the corresponding CD diagram, where $p = 0.95$ was used for all statistical tests. It can be seen that SE ranks first with an average rank between $1 - 2$ with some distance to SRP, which – on average – ranks between $3 - 4$th place closely followed by ARF. Next, there is $\{SRP, ARF, HTT, SB, Bag, HT, NB\}$ which forms a second clique, and $\{Bag, HT, NB, SDT\}$ which forms the last clique. While all three methods $\{SE, SRP, ARF\}$ are in the same clique and hence offer similar performance, SE has some distance. It is only present in this clique, meaning that it is statistically better than $\{HTT, SB, Bag, HT, NB, SDT\}$.

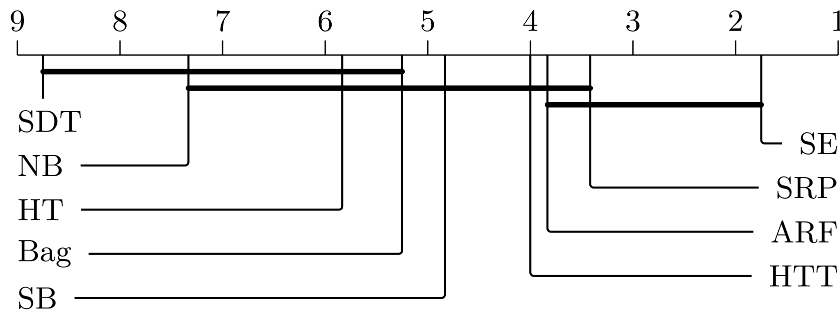


FIGURE 7.2: Critical Difference Diagram for the normalized area under the Pareto Front for different methods over multiple datasets. For all statistical tests, $p = 0.95$ was used. More to the right (lower rank) is better. Methods in connected cliques are statistically similar.

7.3.2 Qualitative Analysis

To gain a more complete picture, we now inspect the iterative development of the test-then-train accuracy and the model size over the learning process for the best-performing configuration of each algorithm without any memory constraints. Figures 7.3 and 7.4 plot the number of seen data points against the accuracy and memory requirement for the gas-sensor and led_a dataset. The gas-sensor dataset is interesting because it contains real-world data with a known time of drift (see Table 2 in [VVA+12]) whereas the led_a dataset contains artificial drift. More qualitative results on the other dataset can be found in appendix C. Looking at the accuracy in Figure 7.3 (top row), we notice a rather chaotic behavior in the beginning, which can be attributed to the fact that in the first ten months of measurements, new classes appear for the first time. Once each class was presented at least once to the algorithms, the accuracy approaches one, where SE has the highest accuracy, followed by SRP. After roughly $4\,000$ data points, we see a drop in the accuracy of ARF and NB, which can be attributed to sudden changes in the distribution after roughly 20 months of measurements. Here, the number of measurements, as well as the class distribution, heavily changes. We also observe that HTT seems to cope better with the dynamic changes in this dataset, compared to HT, which can be expected from the more greedy nature of the algorithm.

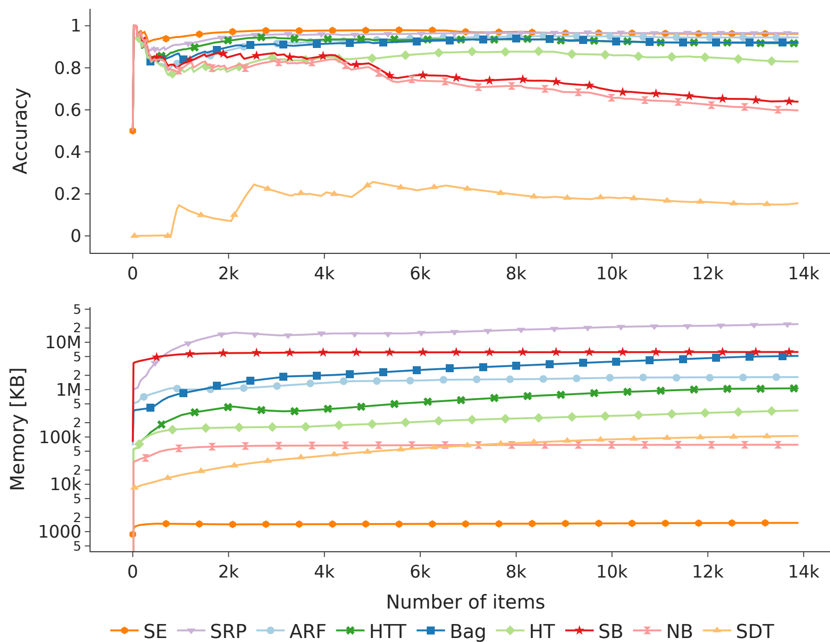


FIGURE 7.3: Accuracy and memory consumption over the number of items on the gas-sensor dataset. Best viewed in color.

Looking at the memory consumption in Figure 7.3 (bottom row), we notice an interesting behavior (note the logarithmic scale). We see that SE uses by far the fewest, strictly bounded resources, whereas the other algorithms require at least a magnitude more memory. As expected, the memory consumption of HT and HTT monotonously rises over time as these algorithms never remove any internal node from the tree. Moreover, HTT must maintain the list of all possible splits at all times, thereby requiring more memory than HT. Likewise, the use of multiple HT(T)s as base learners in ARF, SRP, Bagging, and SB is reflected in the plot. In addition, ARF and SRP utilize the ADWIN drift detector [BG07], which uses a variable-sized sliding window. The window size is computed by Hoeffding's Bound, which is ideally suited if no specific distribution can be assumed. On the downside, the window size convergences comparably slow. As a result, the algorithms store additional information for large windows to detect a possible drift, further increasing memory consumption. Naive Bayes (NB) is a strong competitor to SE but also requires roughly a magnitude more memory. Last, we notice that the memory consumption of SDT increases over time due to a large sliding window in the hyperparameter settings. We conclude that our SE method offers the best predictive performance on the gas-sensor dataset while using the fewest resources, making it an ideal algorithm for resource-constrained environments.

Figure 7.4 shows the test-then-train accuracy (top row) and average model size (bottom row) for the led_a dataset. While the accuracy is relatively stable in the beginning, we can see a clear drop around the 250 000 item mark and a smaller drop later at around 500 000 items. NB and HT seem to suffer the most from this concept drift, but also the other algorithms lose some predictive power. Again, SDT does not seem to learn anything at all. Looking at the memory consumption, we see a similar picture as before: HT(T) and ensembles of HT(T) learners steadily increase their memory consumption over time, requiring up to 100 MB. NB and SDT have the smallest memory consumption, whereas SE ranks third in memory consumption

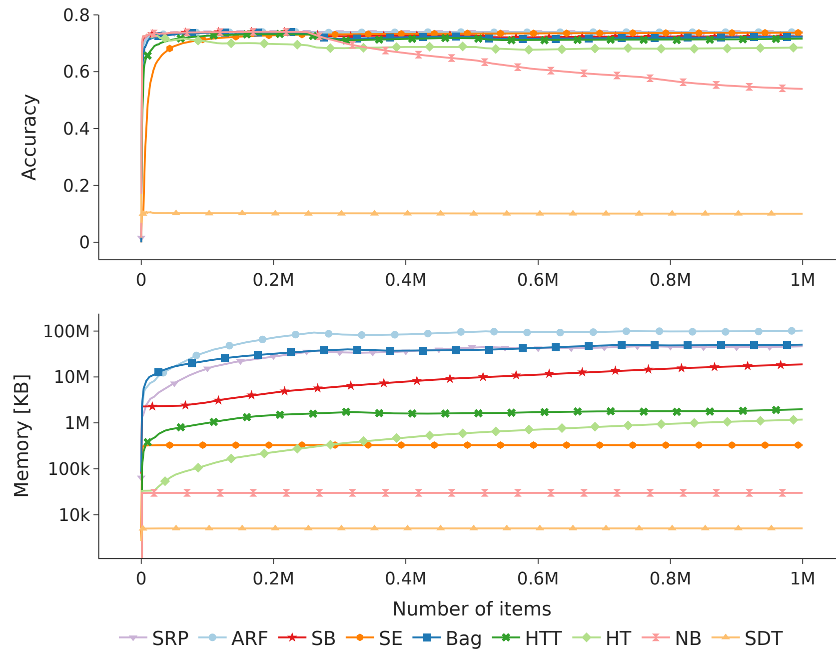


FIGURE 7.4: Accuracy and memory consumption over the number of items on the led_a dataset. Best viewed in color.

in this setting. Again, we find that SE offers excellent performance while ranking among the most resource-friendly methods.

8 | Implementing Ensembles on Small Devices

Model deployment is the last step after training and preparing a model. It determines many of the constraints for the overall pipeline and model. As argued before, discrete classifiers are ideally suited for deployment since they do not require any floating-point operations. Looking at the execution of these models, we find that binarized neural networks or Naive Bayes seem much more favorable than decision trees. BNNs and NB have a deterministic execution time that does not depend on the data but is the same regardless of its input. DTs, on the other hand, have a *probabilistic* inference time in which each observation in a tree may take a different path of different lengths. This unpredictability makes DTs less attractive for real-time systems with hard deadlines[1] or ultra-low power systems where a predictable amount of energy should be consumed by each operation. Yet, DT and ensembles, therefore, are one of the most powerful ML classifiers that practically do not require any computations during deployment but only comparisons. This brings three questions: First, what is the best way to implement DT ensembles on small devices? Second, is there a theoretical model for the execution of DTs so that we can make the execution time more predictable? Third, what would be the best hardware platform and the best implementation of a DT? In this section, we will tackle the three challenges by first introducing a probabilistic view of DT execution. Then, we use this view to derive an efficient memory layout of DT ensembles that maximizes caching. After that, we study the execution of DTs from a more theoretical perspective and tackle the question if there is an optimal hardware platform to execute DTs.

8.1 Implementing Decision Trees

Random forests and decision trees have been studied in the context of inferencing. Van Essen et al. present in [EMGP12] a comprehensive study of different architectures for implementing random forests on CPUs, FPGAs, and GPUs. Their approach is based on the CATE algorithm that trains DTs of a fixed height [PCMM13]. By utilizing these fixed-size trees, the authors show an effective pipelining approach for tree application on CPUs, FPGAs, and GPUs. The traversal of tree ensembles that are not trained via CATE can be categorized by the properties of the executing hardware platform.

[1] "A time-constraint is called hard if not meeting that constraint could result in a catastrophe" [Mar11].

CPUs

In [ALdV14], Asadi et al. introduce different implementation schemes for pre-trained tree ensemble in the context of learning-to-rank tasks: The first approach they present uses a while-loop to iterate over individual nodes of the tree, whereas the second approach decomposes each tree into its individual if-else structure. For the first implementation, the authors also consider a continuous data layout (i.e., an array of `structs`) to increase data locality but do not directly optimize each implementation. Again in the context of ranking models, Lucchese et al. present the QuickScorer algorithm for gradient boosted trees [LNO+15]. In this approach, the authors discard the tree structure but view each tree traversal as a series of bit operations on a 2^L dimensional bitvector, where L is the number of leaf nodes. Their approach offers significant speed-up but is limited to the word size of the CPU, i.e., to trees with at most 64 leaf nodes. Additionally, this approach evaluates *all* splits in all trees in the forest in the worst case.

Vectorization

Looking at vectorization more closely, Kim et al. present in [KCS+10] an implementation for binary search trees using SIMD vectorization on Intel CPUs and compare their implementation against a GPU implementation. The authors provide insight into how to tailor an implementation to Intel CPUs by taking into account register sizes, cache sizes as well as page sizes. Similarly, Lucchese et al. extend their QuickScorer algorithm to use vectorization in [LNO+16], offering a speed-up factor of around 3.2 against the vanilla QuickScorer algorithm. Ye et al. further extended the vectorized QuickScorer (vQS) algorithm by introducing a data structure called `epitome` in [YZZ+18]. Using this data structure, they are able to encode the tree traversal more efficiently so that it is also possible to traverse trees with more than $L > 64$ leaf nodes. However, the decoding of these epitomes requires additional computations so that the overall performance over vQS is in the range of $1.3 - 3.5$, depending on the forest and dataset.

GPUs

GPUs have also been investigated for the traversal of DT ensemble. The first work in this direction was due to Sharp in [Sha08], which showed how to encode DTs as 2D textures that are then processed by the GPU. A more recent re-implementation of this approach can be found in [NSY+20]. Here, the authors discuss multiple strategies that map the execution of a DT into tensor operations. The key insight is that mapping DT traversal to tensor operations usually leads to an increase in computation, but this increase is justified due to the availability of more efficient tensor libraries and tensor processing hardware. For example, the GEMM approach discussed in [NSY+20] has a runtime of $\mathcal{O}(n^2)$ where n is the number of nodes in the tree, which is much slower compared to the 'regular' node-by-node traversal (discussed below) that requires $\mathcal{O}(\log n)$. Yet, during the experiments, the authors report speed-up factors of around 60 when using their approach compared to the vanilla node-by-node traversal. Last, Lettich et al. propose a multicore and GPU version of the QuickScorer algorithm in [LLN+19] with speed-ups up to a factor of 102.6 against the original

version.

FPGAs

Going from fixed computing architectures to FPGAs, there are also multiple works available. In [SDP+15], Saqib et al. present a hardware-software co-design approach in which a software implementation of decision trees is improved by an FPGA decision tree hardware accelerator. The authors show a speed improvement in classification of around 3.5 compared to a software-only solution. Even though the authors propose a theoretical analysis of their FPGA implementation, they lack a thorough comparison with CPUs. More specifically, the authors compare their implementation against a comparably slow Microblaze CPU without vectorization units. Barbareschi et al. present in [BDPG+15] an implementation scheme for random forests on FPGAs focusing on the majority vote of a random forest. The authors present a fast and energy-efficient way of computing a majority vote on FPGAs. Additionally, they estimate the number of logic cells required for a decision tree during tree induction to stop the training of trees that are too large. The authors do not compare their findings against a software implementation.

Small devices often do not offer a GPU (c.f. Table 2.1), making GPU inferencing unattractive for these devices. Similarly, vectorization is not always available in small devices, making these approaches less attractive as well. In the following, we will therefore focus on implementations of DT ensembles on CPUs and – to some degree – also study FPGAs as means to implement the traversal of DTs. Programming a decision tree is a simple task in most programming languages. Take a binary decision tree in Figure 8.1 as an example. When we execute a node of the DT, we either report the associated prediction if the node is a leaf or just need to perform a simple comparison and decide whether the next node is the left child or the right child. In any modern programming language, there are at least two ways to implement such a decision tree:

- **native:** A simple implementation, named the native tree, uses a loop to iterate over each node of a tree within a continuous data structure, e.g., arranged by a one-dimensional array. An example code can be found in Figure 8.2.
- **if-else:** An alternative implementation, named the if-else tree, statically generates if-else blocks. Here, the split values of a tree are all hard-coded as constant values into the instructions. An example code can be found in Figure 8.3.

Bibliographic remark: For historical reasons, the following sections ignore the QuickScorer algorithm and its siblings. The QuickScorer algorithms are currently undergoing a patenting process, and hence they are not publicly available, making experimental comparisons difficult. Moreover, the original implementation only supports Intel CPUs[2] which are rare in embedded systems. During this thesis, I supervised a bachelor thesis implementing the QuickScorer algorithm on ARM [Kos21], and the results are very promising. At the time of writing this thesis, we submitted a research paper to publish our findings, but this remains a work in progress and is therefore not further discussed here.

[2] The author was kind enough to show us the source code under an NDA.

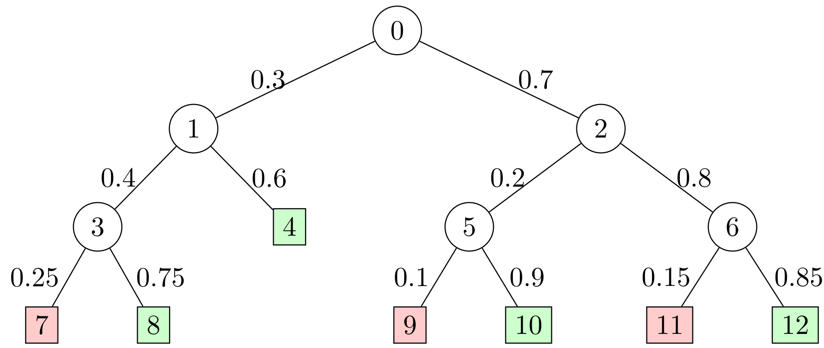


FIGURE 8.1: Binary DT with depth 4. Inner nodes are depicted with circles and leaf nodes are displayed as rectangles. Green nodes indicate a positive, and red nodes indicate a negative class. Each node has a unique id, and every path is associated with a probability. For example, the probability of going from the root node 0 to its right child 2 is $p(0 \to 2) = 0.7$.

8.2 A probabilistic View of DT Execution

To analyze the expected runtime of axis-aligned binary decision tree implementations, we want to use the following notation: Each node receives a unique identifier (e.g., in breath-first order) i. We denote the left child of i with $l(i)$ and the right child with $r(i)$. Let L denote the number of leaves in a decision tree, then there are L different paths from the root node to a leaf.

Recall that to classify a sample x, we begin to traverse the tree starting with its root node and follow the children according to the comparisons at each node until we find a leaf node. Then, we return the associated prediction value of the leaf node. Every observation takes exactly one path $\pi(x)$ from the root node to one leaf. Note that all comparisons performed on a path depend on each other: at each node, we decide which child node to visit next. To lighten the notation, we drop the argument x if we are not interested in the path of a specific observation.

The number of comparisons performed during a path effectively indicates the classification speed: The fewer comparisons we need to perform, the faster we can classify the given observation. By assuming a certain distribution of observations, we can calculate the expected number of comparisons needed for a given tree. More formally, we model each comparison at node i as a Bernoulli experiment in which we will take the path towards the left child with probability $p(i \to l(i))$ and respectively for the right child with $p(i \to r(i))$. It holds that $p(i \to l(i)) = 1 - p(i \to r(i))$ by the construction of the tree. An example can be found in Figure 8.1. Note that the probabilities $p(i \to l(i))$ and $p(i \to r(i))$ can be estimated during training by counting the number of samples at each node taking the left and right path. Assume a path of length Λ_k that ends in node i_k with $\pi_k = (i_1, i_2, \ldots, i_{\Lambda_k})$, where i_{j+1} is either the left or the right child of the j-th node on the path. Then, following this path consists of a series of Bernoulli experiments, each with probability $p(i_j \to i_{j+1})$. Let $\mathcal{P}$ denote the set of all paths in the tree. Then the probability to take path $\pi_k \in \mathcal{P}$ is given by

$$p(\pi_k) = p(i_0 \to i_1) \cdot \ldots \cdot p(i_{L-1} \to i_{\Lambda_k}) = \prod_{j=0}^{\Lambda_k} p(i_j \to i_{j+1}) \tag{8.1}$$

For short, we call $p(\pi_k)$ the probability of node i_k. We make an important observation:

```
struct Node {
  bool isLeaf;
  bool prediction; // Predicted Label
  unsigned int feature; // Targeted feature
  float split; // Threshold
  unsigned int leftChild;
  unsigned int rightChild;
};
Node tree[] = {{0,0,0,3.14,1,2},{0,0,1,1.337,3,4},/* .. */]}
bool predict(float const x[3]){
  unsigned int i = 0;
  while(!tree[i].isLeaf) {
    if (x[tree[i].feature] <= tree[i].split) {
      i = tree[i].leftChild;
    } else {
      i = tree[i].rightChild;
    }
  }
  return tree[i].prediction;
}
```

FIGURE 8.2: Example implementation of a decision tree using the native implementation in C++.

The probability of a node is always greater or equal to the probability of its children. More formally, consider node k with children $l(k)$ and $r(k)$, then it holds that:

$$
\begin{aligned}
p(\pi_k) \geq p(\pi_{l(k)}) &= \prod_{j=1}^{\Lambda_{l(k)}} p(i_j \to i_{j+1}) = p(\pi_k) \cdot p(i_k \to i_{l(k)}) \\
p(\pi_k) \geq p(\pi_{r(k)}) &= \prod_{j=1}^{\Lambda_{r(k)}} p(i_j \to i_{j+1}) = p(\pi_k) \cdot p(i_k \to i_{r(k)})
\end{aligned}
\tag{8.2}
$$

since $p(i_k \to i_{l(k)}) \leq 1$ and $p(i_k \to i_{r(k)}) \leq 1$. By recursion, it follows that the probability of a node is always greater or equal to the probability of any node in its sub-tree. Last, the expected number of comparisons of a tree with L different paths $(\pi_1, \pi_2, \ldots, \pi_L)$ from the root node to one of the leaf nodes, each with length $(\Lambda_1, \Lambda_2, \ldots, \Lambda_L)$ is given by

$$
\mathbb{E}[\Lambda] = \sum_{i=1}^{L} p(\pi_i) \cdot \Lambda_i \tag{8.3}
$$

8.3 Optimizing the Memory Layout of Trees

Recall that due to the significant performance gap between the main memory (DRAM) and the processor, modern computer architectures have introduced a memory hierarchy. The key assumption of the memory hierarchy is the *locality*:

- **Temporal locality:** Recently accessed items will be accessed in the near future, e.g., small program loops.

```
bool predict(float const x[3]){
  if(x[0] <= 3.14){
    if(x[1] <= 1.337){
      return true;
    } else {
      return false;
    }
  } else {
    if(x[2] <= -3.69){
      return true;
    } else {
      return false;
    }
  }
}
```

FIGURE 8.3: Example implementation of a decision tree using the if-else implementation in C++.

- **Spatial locality:** Items at addresses close to the addresses of recently accessed items will be accessed in the near future, e.g., sequential accesses to elements of an array.

Unfortunately, naive implementations of DTs do not exploit such locality when they classify a set of input data. The benefit of the native tree implementation is the temporal locality of the program, i.e., executing a tree is a simple loop with a few lines of code. However, the accesses to the nodes of the tree do not have any spatial locality. The execution of a DT follows a unique path from the root to a leaf, which is stored in memory addresses that are unfortunately arranged discontinuously if no attention is made. As a result, the cached data will not be further used if the distance between each node of the path is greater than the number of nodes that can be loaded into a cache set at once. As for the if-else tree implementation, since the thresholds and the values required for a split node of a tree are all hard-coded into the instructions, this avoids indirect memory accesses and has a clear advantage of reducing the latency. Therefore, the if-else tree implementation does not suffer from missing data locality. However, without awareness of instruction-cache design, the hard-coded instructions may just be loaded into the data cache once and only used once so that the advantage of the temporal locality in the instruction cache is completely abandoned.

Implementing a tree should take the layout of the data (for the native tree), the instructions of the branches (for the if-else trees), and the size of caches of the particular platform of execution into consideration. To do so, we first discuss the downsides of each implementation regarding their caching behavior. Then, we present optimizations to improve caching.

8.3.1 Optimization of If-Else Trees

As already mentioned, we can unroll the comparisons of a DT into conditional statements forming an if-else structure. Since the entire tree is transformed into if-else

blocks without indirect memory accesses, we can expect this implementation to perform better than the native tree structure. However, cache misses may still occur.

Reducing compulsory cache misses

When an instruction cache miss occurs, several instructions are sequentially fetched into the instruction cache. When a branch is executed, these prefetched instructions may not be utilized. If we can increase the chance of actually using prefetched instructions, we can reduce the number of compulsory cache misses. However, DTs are naturally composed of many branches. To reduce the possibility of branch executions for a tree, we can traverse all its paths and swap the children of every node i when $p(i \rightarrow l(i)) \geq p(i \rightarrow r(i))$. In this way, we can decrease the possibility of branching out of the current block, which in turn increases the utilization of prefetched code blocks.

Reducing capacity and conflict cache misses

The best case for exploiting the instruction cache fully is having all the instructions of the if-else tree loaded into the instruction cache. However, if the size of the instructions from the overall tree structure is greater than the size of the instruction cache, the cached instructions may be evicted by loading other instructions due to the capacity and conflict cache misses. Considering the usage of DTs, we can notice that keeping the instructions of those nodes utilized frequently in the instruction cache can improve the utilization of the cached instructions, resulting in better performance. With this idea, we can define a computation kernel that contains those nodes which are used most of the time. For example, note that the root node of a tree is used in every case, and thus it should be kept inside the cache all the time. Let $\mathcal{K}$ denote the kernel and let $s(i)$ be a function estimating the instruction size of node i. Let β be a given budget related to the size of the instruction cache on the target architecture. Then we want to solve the following optimization problem:

$$\mathcal{K} = \arg\max \left\{ p(T) \middle| T \subseteq \mathcal{T} \text{s.t.} \sum_{i \in T} s(i) \leq \beta \right\} \tag{8.4}$$

Given $\mathcal{K}$, we can ensure that these nodes are likely to remain in the cache, whereas the remaining nodes $\mathcal{L} = \mathcal{P} \setminus \mathcal{K}$ may be evicted.

In order to solve Eq. 8.4 we need to iterate over all possible subsets of all the nodes $\mathcal{T}$, which might be difficult for large trees. Thus, we propose a greedy approach in which we look at a complete path from the root to the leaf node: First, we swap the children depending on their probabilities, as already explained in the former section. Then, we sort all paths in the tree by their probability. After that, we greedily add a node one by one into $\mathcal{K}$ until the accumulated size of the added nodes is greater than the given budget β. The remaining nodes are placed into another set $\mathcal{L}$ that represents the nodes that might be evicted more often. Algorithm 19 summarizes the presented approach.

Recall that the probability of a node is always greater-or-equal to the probability of its sub-tree. Hence, as soon as a node is sorted into $\mathcal{L}$, its entire sub-tree is also added to $\mathcal{L}$. We exploit this insight by using `goto` statements to break the sequential generation of if-else blocks: First, we generate if-else blocks for all nodes in $\mathcal{K}$. Once

Algorithm 19 Optimization of the if-else tree.

```
1:  function OPTIMIZE_IF-ELSE(𝒫)
2:      𝒫 ← swap(𝒫)                     ▷ Swap nodes based on probabilities
3:      𝒫 ← sorted(𝒫)                   ▷ Sort decreasing using probabilities
4:      b ← 0, ℒ ← ∅, 𝒦 ← ∅
5:      for π ∈ 𝒫 do                    ▷ Over all path in tree
6:          for i ∈ π do                ▷ Over all nodes on path
7:              if b + s(i) > β then
8:                  Add i to ℒ          ▷ Node does not fit into kernel
9:              else
10:                 Add i to 𝒦          ▷ Node fits into kernel
11:                 b ← b + s(i)
        return 𝒦, ℒ
```

the left/right child of one of those nodes is in $\mathcal{L}$, a goto statement is generated at the same position to replace the original if-else statement. Then, the corresponding if-else statements of this node and its children are all generated into a label block at the end. Figure 8.4 shows an example based on Figure 8.3 by applying Algorithm 19.

```
bool predict(short const x[3]){
  if(x[0] > 3.14){
    if(x[2] <= -3.69){
      return true;
    } else {
      return false;
    }
  } else {
    goto Label0;
  }
Label0:
  {
    if(x[1] <= 1.337){
      return true;
    } else {
      return false;
    }
  }
}
```

FIGURE 8.4: Example implementation of a decision tree using the goto implementation in C++.

The question remains, how to estimate the instruction size $s(\cdot)$ of each node? The instruction set size generally differs for the two different types of nodes:

- **Split nodes** require three types of instructions. First, the values of the target feature and the corresponding threshold are loaded into registers. Second, the values inside the registers are compared against constant values, and last, a jump out of the current block is performed based on the comparison.

- **Leaf nodes** need two types of instructions. First, the return values of the prediction are stored in a register, and second, a jump back to the caller of the if-else tree is performed.

Based on the above analysis, we can estimate $s(\cdot)$ by counting the number of generated instructions on the targeted architecture in an isolated example. However, note that in a real application, the actual number of instructions may differ based on the compiler, compiler options, and specific tree used. Thus, this mapping function helps to choose appropriate parameters but should be viewed as the expected size of a node. Table 8.1 summarizes the expected size of instructions for ARM, X86 (Intel) and PPC.[3]

TABLE 8.1: The expected size of instructions for a split node and a leaf node in a decision tree on ARM (`Raspberry PI 2`), PPC (`NXP T4240 processors`) and Intel (`Intel Core i7-6700`) processors.

	ARM [Bytes]		PPC [Bytes]		Intel [Bytes]	
Type	Int	Float	Int	Float	Int	Float
Split	20	32	20	48	28	17
Leaf	8	8	8	8	10	10

8.3.2 Optimization of Native Tree

As shown in Figure 8.2 we can implement a DT by placing the nodes sequentially in an array and by traversing this array using a simple while loop. We observe that half of the nodes in a tree are leaf nodes, which only store a prediction value. The native implementation however assumes the same data type for each node, leading to unnecessary overhead. Second, considering the usage of DTs for predicting classes, we notice that the data access pattern in the array is mostly non-sequential. The distance between each accessed element becomes bigger when the depth of targeted nodes in the DT becomes greater. This phenomenon violates the spatial locality of the array and abandons the advantage of the cache design, which may result in high cache misses.

Reducing compulsory cache misses

Nodes are prefetched into the instruction cache sequentially. If we can reduce the amount of memory each node requires, we can fit more nodes into the cache and thus reduce compulsory cache misses. For the native implementation, we recognize that a leaf node only stores a prediction value but does not use the pointer to its children, nor does it use the feature index or the split value. A simple way to reduce memory consumption is to remove all leaf nodes from this array and move them to a separate array with a specialized data type. However, then we have to lay out two arrays in the memory, which might be difficult. Thus, we propose to abandon the `isLeaf` and `prediction` fields of the native solution but store the prediction of the left (right) child directly in the respective fields `left` (`right`) if it is a leaf node. This method only requires us to lay out one array but offers the same size reduction as

[3] We adopt GNU g++ compiler version 4.8.3 for ARM, version 4.9.2 for PPC, and version 5.4.0 for Intel with `-O0` option.

using two arrays.

Reducing capacity and conflict cache misses

As mentioned previously, if no attention is paid, the nodes stored in memory are arranged discontinuously. Thus, when a node is loaded into the cache, the nearby nodes should be on the same path to reduce capacity and conflict cache misses. A sensible way to exploit the data locality is to allocate as many nodes as possible on the same path into the same cache set. To do so, we propose the following approach, where τ denotes the cache set size: Let `A` be the array in which we place all nodes of the tree. Furthermore, let $\mathcal{C}$ be the candidate list of nodes that have not been placed in `A` yet and let $\mathcal{S}$ denote the nodes which should be placed in the same cache set. For each node, we greedily choose that child with the largest probability on the current path and try to place it in $\mathcal{S}$. Once $\mathcal{S}$ contains $\tau - 1$ elements and hence is full, we append all nodes from $\mathcal{S}$ to the array `A`, reset $\mathcal{S}$ and continue with the next cache set. Algorithm 20 summarizes this method.

Algorithm 20 Optimization of the native tree.

```
 1: function OPTIMIZE_NATIVE(P, τ)
 2:     A = [ ]                                    ▷ Init. array
 3:     C ← {0, 1, . . . , n}                       ▷ Init. remaining indices
 4:     while C ≠ ∅ do                              ▷ Repeat as long as there are nodes
 5:         i ← arg max_{j∈C} {p(π(j))}             ▷ Get most probable node
 6:         C ← C \ {i}                             ▷ Repeat as long as there are nodes
 7:         S ← {i}                                 ▷ Add node to S
 8:         while |S| ≠ τ do                        ▷ Follow path until cache-line is full
 9:             if i is leaf node and C ≠ ∅ then
10:                 i ← arg max_{j∈C} {p(π(j))}     ▷ Start new path if cache-line is not full
11:                 C ← C \ {i}
12:             else
13:                 ▷ Follow left or right path depending on the probability
14:                 C ← C ∪ arg min{p(i → l(i)), p(i → r(i))}
15:                 i ← arg max{p(i → l(i)), p(i → r(i))}
16:                 if |S| = τ − 1 then             ▷ Update C if cache-line is now full
17:                     C ← C ∪ {l(i), r(i)}
18:             S ← S ∪ {i}                         ▷ Extend path
19:         A.append(S)                             ▷ Append path to array
        return A
```

When adding a new node to $\mathcal{S}$, attention has to be paid because there are two types of nodes (lines 9-18):

- The current node is a split node. Then we pick the next node based on the children's probabilities and put the more probable child into $\mathcal{S}$ and the other child into the candidate list $\mathcal{C}$.
- The current node is a leaf node, i.e., it is the end of the path. Then we pick up a sub-root with the highest probability from the candidate list $\mathcal{C}$ as long as it is not empty. The traverse starts again until $\mathcal{S}$ is full.

If the current set is full before finishing a traverse of a path (line 16), two children are put back into the candidate list $\mathcal{C}$ (line 17). A sub-root that has the highest probability is then picked up from $\mathcal{C}$ for the next new set $\mathcal{S}$. Once a set is finished, the nodes in it will be allocated into the data array sequentially. To the end, the output of the algorithm is the data array with a path-oriented layout, in which path-oriented sets are sequentially allocated into the array.

We want to give a quick example to illustrate Algorithm 20. Consider the DT in Figure 8.1 and set the size τ to three. The first path starts from the root node 0, which is a split node. Accordingly, node 2 with a higher probability is chosen, and node 1 is put into $\mathcal{C}$. From the children of node 2, the leaf node 6 is chosen (node 5 is put into $\mathcal{C}$). As now, $\mathcal{S}$ is full with three nodes, the algorithm adds the current $\mathcal{S}$ into the output array, prepares a new set $\mathcal{S}$, and picks up a sub-root with the highest probability from $\mathcal{C}$. In the list $\mathcal{C}$, there are currently nodes 1 and 5. Since the probability of node 1 is higher than the probability of node 5, node 1 is the next chosen sub-root. To the end, the delivered sets are: $\{0, 2, 6\}$ and $\{1, 3, 5\}$.

8.3.3 Experiments

In this section, we experimentally evaluate the proposed optimizations. We have performed 1800 different experiments by training decision trees (DT), random forests (RF), and extremely randomized trees (ET) on 12 different data sets with varying tree depths to generate the aforementioned implementations for different architectures, i.e., X86, PPC and ARM CPUs.

Table 8.2 shows the data sets we used during the experiments. All data sets are available in the UCI Machine Learning Repository [DG17] except for MNIST (`http://yann.lecun.com/exdb/mnist/`), IMDB [MDP+11] and FACT [ABB+13]. In addition to the number of features and the number of examples during test time, we also report the range of accuracy for the three different models DT, RF, and ET. In all experiments, we used the CART algorithm with the Gini-Score criterion for node-splitting, and trained models using the `sklearn` package [PVG+11]. For RF and ET, we used 25 trees. If the respective data set comes with a pre-computed train/test split, we use this. Otherwise, we use 75% of the data for training and 25% of the data for testing. Expectantly, DTs often do not achieve high accuracy, whereas RF and ET perform best with large trees. We want to emphasize that we did not perform any hyperparameter optimization with respect to the classification accuracy but report the accuracy here to validate our toolchain.

After training, we export the models into a JSON format which is used by a code-generator that generates the optimized implementations for each individual forest. During code generation, we make sure that optimized trees retain their accuracy. Note that sklearn always produces floating-point split values. For data sets with integer features (e.g., letter or MNIST), this was rounded down toward the next integer to circumvent the use of floats. This does not change the accuracy. The original code for these experiments is publicly available at `https://bitbucket.org/sbuschjaeger/arch-forest`, but a recent re-implementation of our tool-chain can be found under `https://github.com/sbuschjaeger/fastinference`. After the implementations have been generated, we use the `GNU` toolchain to compile the code with the most aggressive optimizations (`-O3`) enabled. Each implementation is tested individually by using the following protocol: For minimizing unfairness due to caching, we first iterate twice over the test data and perform predictions (burn-in phase). Then, we measure the runtime required to classify all examples in the test set and repeat this 50 times.

TABLE 8.2: Datasets used for our experiments. We performed minimal pre-processing on each dataset, removing instances that contain `NaN` values and computed a one-hot encoding for categorical features

Dataset	# Examples	# Features	Accuracy
adult	8141	64	0.76 - 0.86
bank	10297	59	0.86 - 0.90
covertype	145253	54	0.51 - 0.88
fact	369450	16	0.81 - 0.87
imdb	25000	10000	0.54 - 0.80
letter	5000	16	0.06 - 0.95
magic	4755	10	0.64 - 0.87
mnist	10000	784	0.17 - 0.96
satlog	2000	36	0.40 - 0.90
sensorless	14628	48	0.10 - 0.99
wearable	41409	17	0.57 - 0.99
wine-quality	1625	11	0.49 - 0.68

We find that the performance of our implementation compared to sklearn might be of interest since sklearn is arguably one of the most-used machine learning library and thus well-known to many practitioners. We found that our implementation is, on average, $500-1500$ times faster than sklearn. However, we admit that this comparison is biased because large parts of sklearn are written in Python and optimized for batch execution. Therefore, we will focus the remaining evaluation on our implementation and RF models in the remaining parts of this section. We notice that DT and ET result in similar behaviors across all systems and thus do not add much more value to the discussion here. We use a naive native implementation that does not perform any optimizations as a baseline for all experiments and measure the average speed-up for each data set of each optimization against this implementation.

In the following, we denote 'native' as the native implementation with reduced memory consumption but without cache optimizations and 'optimized native' as the native solution with reduced memory consumption and cache optimizations. We denote 'if-else' as the regular if-else implementation and 'optimized if-else' as the cache-optimized if-else variant, including node swapping. For native optimizations, we choose $\tau = 25$ on X86, $\tau = 8$ on ARM, and $\tau = 8$ for the PPC CPU. For if-else optimizations, we use an instruction-cache size $\beta = 128000$ Bytes on X86, $\beta = 32000$ Bytes on ARM, and $\beta = 32000$ Bytes on the PPC CPU. For `X86`, we performed the experiments on a `Intel Core i7-6700` desktop machine with 16 GB RAM. For `PPC`, we use a `NXP Reference Design Board` with `T4240 processors` and 6 GB RAM. Last, for `ARM`, we use a `Raspberry PI 2` with a `ARMv7` CPU and 1 GB RAM.

Experiments on the X86 CPU architecture

Figure 8.5 depicts the average speed-up of the four different implementations on Intel for different tree depths. The error band shows the standard deviation across the different datasets. First, we note that the if-else tree versions are the fastest on Intel and offer a speed-up of around 3 across all tree depths. For smaller tree depths from $1-10$, we see that optimizing if-else trees only offers marginal speed-up. However, for larger tree depths around 15 and 20, we can see that optimized

if-else trees can retain their speed-up and outperform un-optimized if-else trees with a speed-up factor larger than 3. Native trees do not perform as well as if-else trees on Intel CPUs. Overall, the speed-up compared to naive native trees is only marginal for smaller trees below a depth of 15. Here, both versions, i.e., the native tree and the optimized native tree, offer a speed-up of 1.5 at most. Interestingly, for larger trees around depth 15 and more, we again notice that our optimizations improve performance.

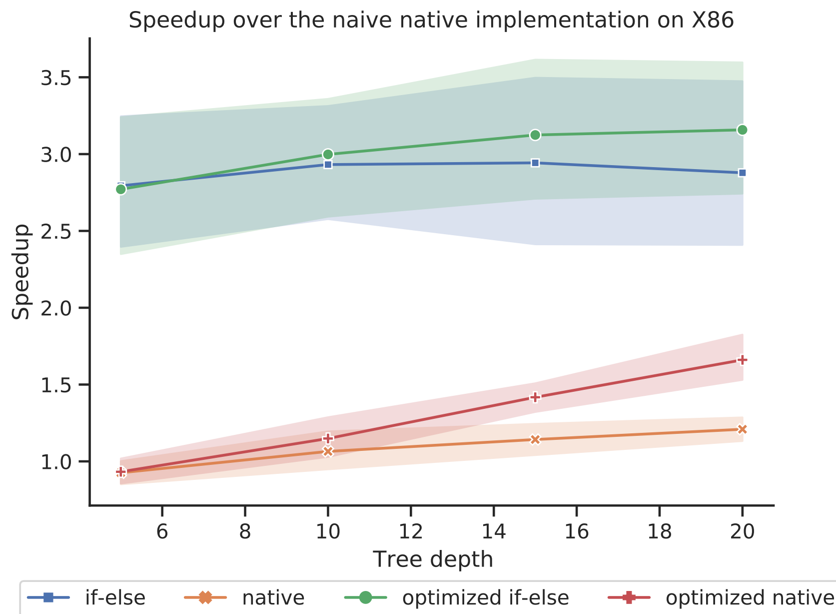


FIGURE 8.5: Average speed-up for real-time execution compared to the naive native implementation on Intel averaged across the different datasets. The error band depicts the standard deviation.

Experiments on the PPC CPU architecture

Figure 8.6 depicts the average speed-up of the four different implementations on PPC for different tree depths. Again the error band shows the standard deviation across the different datasets. We can observe that the results here are similar to Figure 8.5, in which if-else trees always outperform native trees with a speed-up in the range of $2-5$. Along with the increment of tree depth, the speed-up from both if-else versions drops, but especially un-optimized if-else trees suffer from degraded performance, dropping to almost 2, whereas the optimized version can retain a speed-up of around 3.5. Similar to X86 CPU, the native implementation does not seem to be the best choice here by providing a speed-up under 2 in all cases. However, we also notice that with increasing tree depths, optimizations are more important. It is worth noting that we can observe some cases where the native trees outperform if-else trees when tree depth is bigger than 15.

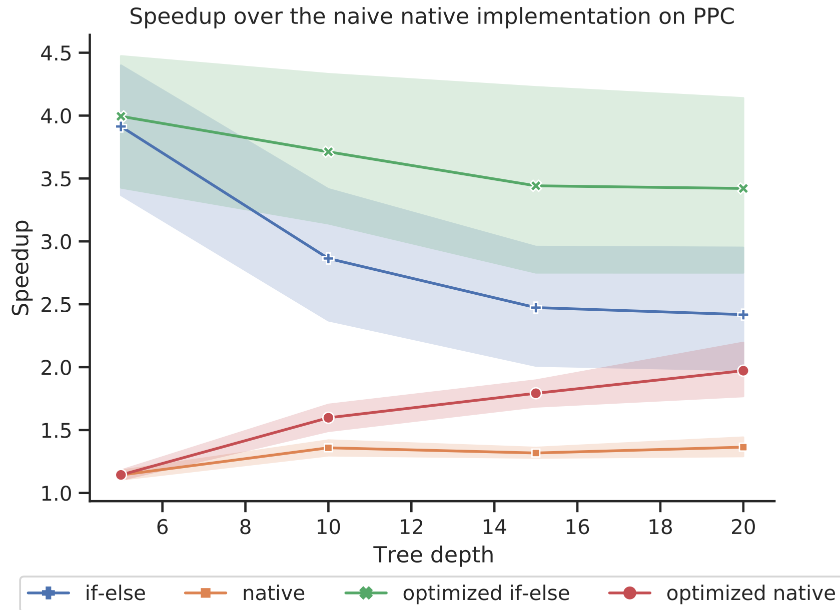


FIGURE 8.6: Average speed-up for real-time execution compared to the naive native implementation on PPC averaged across the different datasets. The error band depicts the standard deviation.

Experiments on the ARM CPU architecture

Figure 8.7 depicts the average speed-up of the four different implementations on ARM for different tree depths. As before, the error band shows the standard deviation across the different datasets. We observe that the situation on ARM is more fragmented compared to X86 and PPC. In general, we are able to achieve a speed-up of around 4 for small trees, which drops to around $2-3$ for larger trees. Both implementations roughly start with the same speed-up factor for small trees but then quickly diverge for tree depth around $5-15$. In this range of tree depth, we see that if-else trees are the fastest choice on ARM. Additionally, we notice that with increasing tree depth, cache optimizations become more important and consistently outperform their un-optimized counterpart. Once trees are sufficiently large, we see that the native trees match the performance of if-else trees again and even outperform them for tree depth of 15 and 20 in some cases. In this sense, the results are similar to what we have seen on the PPC architecture.

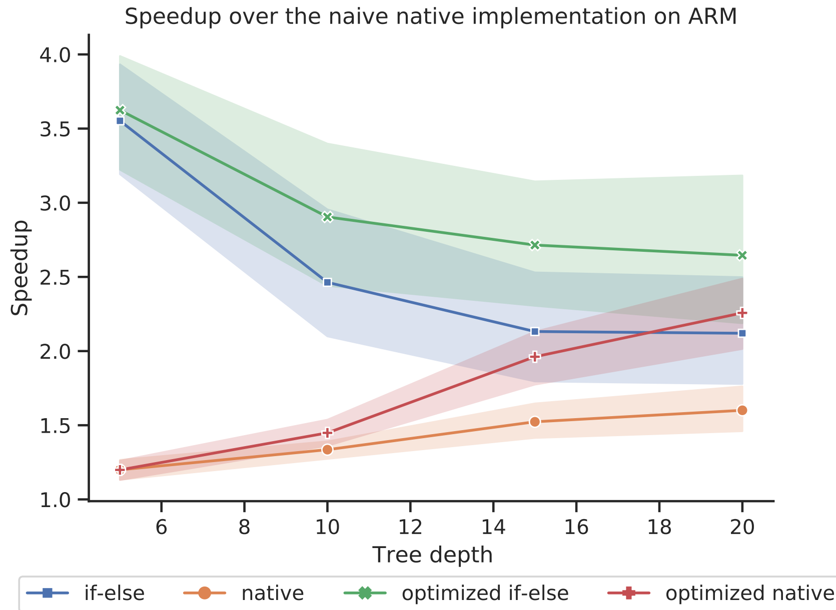


FIGURE 8.7: Average speed-up for real-time execution compared to the naive native implementation on Intel averaged across the different datasets. The error band depicts the standard deviation.

Discussion of the Experiments

The experiments show overall different behaviors across the three different architectures but also depict some similarities. Here, we want to discuss these phenomena in terms of the properties of the specific architectures, as well as the concrete CPU models used for experiments. We note that one of the main architectural differences between X86, ARM, and PPC are the available instructions. Since native trees only use a small amount of hot code, the differences between CPU architectures will likely not matter much here. However, looking at if-else trees, we can expect a larger difference. To further investigate the interplay between CPU architectures and code size, we consider Table 8.3 in the following. Table 8.3 depicts the number of instructions of a single tree for varying tree depths for the fact data set (containing floating-point features) and the covertype data set (containing integer features) of the regular if-else implementation.

TABLE 8.3: Number of instructions for an unoptimized if-else decision tree implementation on different architectures compiled with O3 option. The left side shows a tree for the covertype dataset with integer features, and the right side shows a tree for the fact dataset with floating-point features.

DateType	DT-1	DT-5	DT-10	DT-15	DT-20
Intel	224	575	8185	51005	167644
PPC	232	604	7732	51840	170772
ARM	204	604	9040	55012	180628

DateType	DT-1	DT-5	DT-10	DT-15	DT-20
Intel	96	415	17023	127330	404722
PPC	96	556	20996	169696	577952
ARM	88	428	18436	154992	542020

Clearly, the if-else trees are the best choice for Intel CPUs, but why is that so? We find two reasons for that, one of which is related to the architectural specifics of X86 architecture and one which is related to the specific CPU we used: First, X86 CPUs are Complex Instruction Set Computers (CISC) offering a very rich set of instructions

which include all sorts of specialized operations. Since if-else trees unroll the complete tree structure into code, they allow the compiler to utilize this multitude of instructions to the fullest by encoding larger parts of the tree in single instructions. And indeed, looking at Table 8.3 we see that the Intel CPU almost always requires the fewest instructions per decision tree. Second, the `Intel Core i7-6700` CPU used for experiments has a comparably large instruction cache of 256 KB combined with two larger shared caches of 1 MB (L2 Cache) and of 8 MB (L3 Cache). Thus, by encoding a single tree in only a few instructions, it is likely to fit into the larger instruction cache. In contrast, native trees do not utilize the CISC architecture and "waste" additional data cache by encoding the tree as data and not as instructions.

Similar to the `X86` architecture, we have seen that if-else trees perform very well on the `PPC` architecture, but to a lesser extent. Again, we can try to explain this behavior in terms of the `PPC` instruction set architecture and the specific CPU model used for experiments. The `PPC` CPU architecture is a Reduced Instruction Set Computer (RISC) with performance enhancement for high-performance computing. RISC does not offer instructions for specialized operations as CISC does. Thus, the compiler must largely rely on the combination of (comparably) simple instructions to implement if-else trees. This, in turn, results in more code which is less likely to fit into the instruction cache. Comparing the instruction size of `PPC` to `X86` in Table 8.3 we see that the `PPC` architecture indeed requires more instructions compared to `X86`. Interestingly, this case is less severe for integer features, which can be attributed to the high-performance enhancements in this instruction set architecture. Looking at the cache sizes of the `T4240 processors`, we see that it only has 32 KB instruction cache but also comes with a 2 MB shared L2 cache, which is even larger than the `Intel Core i7-6700` CPU. For smaller trees, around $5 - 10$, the cache sizes are still enough to hold all trees, and thus if-else trees are still the fastest choice. If trees become large (depth 10 and more), the instruction cache is not enough to hold all trees anymore, and we must rely on the larger L2 cache. However, this cache is slower, and thus we suffer some performance penalty, which in combination with the larger code size, explains the performance drop for larger trees.

Last, we want to discuss the fragmented behavior of the `ARM` architecture. Again we try to answer this question in terms of ARM's instruction set architecture and the specific CPU used for experiments. Much like its `PPC` counterpart, `ARM` also uses a reduced instruction set architecture (RISC). However, `ARM`'s RISC does not come with specialized instructions for high-performance computing, and thus, the compiler has to rely to a larger extent on the combination of simple instructions for if-else implementation. This, in turn, results in even more code for integer features, which is less likely to fit into the instruction cache, as shown in Table 8.3. Interestingly, for floating-point features, we see that the `ARM` CPU uses fewer instructions than the `PPC` CPU, which can be attributed to the specific CPU models used during experiments. The `T4240 processors` are optimized towards high-performance computing in a low-power, embedded computing setting, such as networking applications, and thus are optimized towards integer operations. In contrast, the `ARMv7` CPU of the `Raspberry PI 2` is a general-purpose CPU aimed at the needs of the average user, and thus it lays a larger emphasis on floating-point operations compared to the `T4240` processors. It has a 32 KB instruction cache in combination with a significantly smaller 512 KB L2 shared cache. Compared to the other CPUs, this means that the `ARM` CPU has $2 - 16$ times less L2/L3 cache available. For smaller trees around depth $5-10$, the cache sizes are still enough to hold all trees, and thus if-else trees are still the fastest choice. For larger tree depths, however, the instruction cache is not enough anymore, and native structures using the data cache become faster. However, since the data cache is also

small, we quickly fill both caches to their maximum. Interestingly, if we optimize both if-else and native trees, we end up with roughly the same performance.

8.4 A Theoretical Execution Model of DT ensembles

As discussed previously, the optimal implementation of a DT ensemble depends on the trained trees as well as the target architecture. To make matters more complicated, the previous experiments indicate that it is not even possible to give a general guideline on which implementation should be favored. This raises the question if the von Neumann architecture is best suited to execute DTs in general or if there might be a better circuit for this task. In this section, we will tackle this question from a theoretical point of view. To do so, we first derive a theoretical von Neumann model and implement decision trees in it. Then, we repeat the same process without the restrictions of the von Neumann architecture using the building blocks of an FPGA to compare the theoretical runtime of both circuits.

8.4.1 A Theoretical von Neumann Architecture

As already discussed in section 2.1, there is a vast landscape of different processes available that are all based on the von Neumann architecture. Hence, to make a theoretically more grounded evaluation of existing processors, we now formalize the instructions of a theoretical von Neumann CPU together with the number of clocks required to execute the instructions. We tried to keep this theoretical instruction set architecture close to existing real-world hardware but do not restrict ourselves to instructions that are available in every CPU. A comparison between this theoretical model and real-world hardware is given at the end of this section.

Our theoretical CPU is connected via a common communication bus to the main memory as well as peripheral devices. The CPU operates on words of size s_w and has registers of the same size. A cache with size M_c is used, which is organized in cache lines with size s_c that is a multiple of s_w. It has a vectorization unit that operates on vectorization registers with size $s_v = v \cdot s_w$, where v denotes the degree of vectorization. `load` and `store` instructions can be used to load and store values from the cache. In the case of vectorization units, the `load` operation can only load continuous memory from the cache into the vector register. If we want to load values from different, non-continuous memory locations, we first need to store the corresponding memory addresses into one register using the `load` instruction. Once the addresses are present in one register, we can use a `gather` instruction to load the values placed at the different memory addresses into one register. In order to extract scalar values from the vector registers, we can load specific `lanes` from the register unit. The results of vector comparisons are saved into the vector register. Since the outcome of a single comparison can be saved using one bit, we only need v bits to save the vector comparison. In order to access these v comparison bits, we use an `extract` instruction, loading the v comparison bits into a scalar register. For arrays, we assume that they are stored in continuous memory. Memory access on arbitrary indices can also be performed with a single `load` operation if the specified index is saved in a register. Our theoretical CPU is clocked. We denote the clock frequency with C^{CPU}. To further simplify our analysis, we assume that the complete forest will fit into the cache, and we will operate only on elements saved in the registers as well as in the cache. Additionally, we use the following cost model:

- **load:** Accessing a specific word of size s_w already present in the cache takes one clock cycle. Loading continuous memory into a vector register also takes one clock cycle. Loading Words smaller than s_w requires one load instruction and an additional $lane$ access to extract smaller words.
- **gather:** The gather instruction takes one clock cycle.
- **store:** Saving one data item of size s_w or smaller inside the cache takes one clock cycle. Saving a vector register into continuous memory also takes one clock cycle.
- **compare:** The comparison consists of two operations. First, the comparison is performed by taking one clock cycle. Then, a conditional jump to the next instruction based on the outcome of the comparison is executed, taking another cycle.
- **arithmetic and logic operations:** Boolean operations, as well as accessing specific parts of vectorization registers, take one clock cycle.

Our model largely follows the X86 architecture with some differences for SIMD operations. Recent X86 implementations have registers with 64 bit and vectorization registers ranging from 128 bit up to 512 bit. Loading values into scalar register is achieved using the `MOV` instruction, which supports address resolution for memory access in one clock cycle. Similarly, vectorization registers are loaded using one of the many SIMD counterparts of the `MOV` instruction, e.g., `MOVDQA` in the SSE extension. Performing scalar comparison is achieved by first performing a comparison using the `CMP` instruction and then by performing the corresponding jump to the next code segment, e.g., using the `je` (jump-if-equal) instruction. Performing SIMD comparisons can be implemented by using the corresponding vector instructions, e.g., `PCMPGTB` for the SSE. The resulting bitmask can be extracted using the `MOVMSKPS` instruction in SSE. The support for gather instructions is available with the Advanced Vector Extensions 2 (AVX 2), which is, for example, implemented in Intel Haswell CPUs. Looking at ARM's RISC instruction set architecture, we find more similarities for scalar instructions but larger differences for SIMD instructions. Until 2011, the ARM ISA supported 32 bit but switched then towards 64 bit registers. A vectorization extension called NEON with 128 bit registers is also available. Similar to the X86 architecture, `MOV` instructions are used to load memory content into registers. The `MOV` instruction may perform address resolution and thus can also load memory content in one clock cycle. In order to load content into the vectorization registers, the `VLD` instruction can be used. Comparisons work the same as in X86 CPUs: First, the comparison is performed using the `CMP` instruction, and then a conditional jump is performed, e.g., using the `BLT` instruction. In the case of vector comparison, the NEON extension provides a `VCMP` instruction. Unfortunately, there is no dedicated instruction to extract the comparisons bitmask. This instruction must be emulated with individual lane accesses on the registers. Also, there is no `gather` instruction in NEON.

8.4.2 A Theoretical Model of FPGAs

While the von Neumann architecture is arguably the most common computing architecture to date, it is somewhat restrictive as it has a fixed execution pipeline. What would happen if we break this established pipeline for the execution of DTs and implement a decision tree specific integrated circuit?

Since FPGAs are essentially free to image every possible circuit, they offer the perfect platform to implement this approach. FPGAs can be used to build any hardware architecture, and hence they do not operate on fixed words. Data access and computation can be tailored specifically for the given task at hand. Additionally, it is not required that FPGAs are clocked. Note, however, that block memory, as well as DSP units, are implemented in fixed hardware and thus use a standardized clocked interface. For a fair comparison, we take into account that the CLBs are a scarce resource. We model the accesses to block ram also with `load` instructions, which – in the case of a CPU – are already given by the instruction set architecture of the CPU. In the case of FPGAs, these instructions need to be implemented. Assuming we want to access a specific entry i inside an array `arr[i]`, we need to compute the address of that element first in order to issue the corresponding `load` instruction. Given the array is stored continuously in the block ram starting at address `arr`, and it contains data items of size s, the address of `arr[i]` is given by $arr + i \cdot s$. Thus, we need to implement summation and multiplication for address resolution. Integer summation and multiplication is a well-studied problem in literature, and many solutions such as carry-look-ahead adder, Ladner-fisher adder, or Wallace trees exist (see, e.g., Appendix I in [HP11] or chapter 12 in [WH10]). Additionally, the FPGA's DSP elements may already offer a corresponding multiply-accumulate operation in fixed hardware. Therefore, we assume that we can implement the address resolution mechanism with a constant delay of one clock cycle using at most r_{addr} CLB resources. If we want to compare a single bit against a constant value 0 or 1, we can utilize the fact that FPGAs can address arbitrary bit length. More formally, we need to realize a function $eq\colon \{0,1\}^2 \rightarrow \{0,1\}$, which can be implemented using $r_{eq} = 1$ CLBs given that the CLB tables are larger or equal to two, i.e., $t \geq 2$. In order to compute the inequality $a \leq b$ of two unsigned integers $a = (a_1 a_2 \dots a_{s_w})$ and $b = (b_1 b_2 \dots b_{s_w})$ we need to implement a boolean function $le\colon \{0,1\}^{2s_w} \rightarrow \{0,1\}$. To do so, we can observe that the comparison between a_i and b_i is independent of any other comparison of a_j and b_j in a and b, given $i \neq j$. Therefore, we can build a tree structure in which packs of $t/2$ bits of a and $t/2$ bits of b are fed into the same CLB. Then, we can merge the output of the t CLBs and feed its output into the next layer. We repeat the process until the remaining inputs fit into one CLB. More precisely, we build a tree with $n_l = \lceil \frac{2 \cdot s_w}{t} \rceil$ leafs and t children per node. The number of inner nodes n_i can be obtained by using the well known fact that $n_l = (t-1)n_i + 1$ for t-ary trees:

$$\begin{aligned} n_l &= (t-1)n_i + 1 \\ \left\lceil \frac{2 \cdot s_w}{t} \right\rceil &= (t-1)n_i + 1 \\ \frac{\lceil \frac{2 \cdot s_w}{t} \rceil - 1}{t-1} &= n_i \end{aligned} \tag{8.5}$$

Let $t > 4$, then this leads to a total of

$$r_{le} = n_i + n_l = \frac{\lceil \frac{2 \cdot s_w}{t} \rceil - 1}{t-1} + \left\lceil \frac{2 \cdot s_w}{t} \right\rceil = \left\lceil \frac{2 \cdot s_w}{t} \right\rceil \left(1 + \frac{1}{t-1}\right) - \frac{1}{t-1} \leq 1.25 \left\lceil \frac{2 \cdot s_w}{t} \right\rceil \tag{8.6}$$

CLBs are required to implement the comparison of two s_w-bit unsigned integers. In CPUs, we need to perform conditional jumps based on the outcome of the comparisons. In the case of FPGAs, we can use the 1-bit output of the comparison to control the functional units directly. Thus, the comparison combined with branching only needs one clock cycle in FPGAs. Realizing a register of size s_w takes s_w CLBs, as each

CLB can store up to one bit. Realizing a constant of s_w bit can be achieved using $\lceil \frac{s_w}{t} \rceil$ truth tables. In summary, we assume that the FPGA is running with a clock frequency C^{FPGA} and the following operations are available:

- **load:** Loading a specific word of size s_w from the block memory takes two clock cycles (address resolution and actual memory access) and uses r_{addr} resources.
- **load:** Loading arbitrary words from registers realized with CLBs take one clock.
- **load:** Loading constant words takes zero clocks.
- **store:** Storing a specific word of size s_w in the block memory takes two clock cycles and uses r_{addr} resources.
- **store:** Storing arbitrary words in register realized with CLBs take one clock cycle.
- **store:** Storing a s_w-bit constant value in a register is performed in zero clock cycles and requires s_w CLBs (on CLB saves one bit).
- **compare:** Comparisons are performed using one clock cycle. Comparison against a constant single-bit value needs r_{eq} CLBs, whereas comparison of two s_w-bit integers requires r_{le} CLBs.

The reader might wonder why loading a constant word inside the FPGA takes zero clock cycles. In order to execute a specific operation, the corresponding functional unit must load the desired operands first. If, however, one operand is a constant value, we can directly hard-wire this constant value into the functional unit. Therefore, loading of constant operands is not required leading to a clock delay of zero clocks. Because FPGAs do not offer the possibility to perform address resolution combined with loading in one instruction, the access to block memory is slower compared to CPUs. In contrast, FPGAs offer the possibility to perform multiple instructions in parallel. To encourage this property, we will allow up to two concurrent accesses to block memory in one clock.

Xilinx is one of the largest FPGA manufacturers with a variety of different FPGA models [Xil]. Xilinx combines 4 look-up tables with 6 inputs and 1 output into one CLB and offers models with 2000 to $33\ 650$ CLBs. Additionally, between 720 KB to $13\ 140$ KB block memory is available. Block memory is implemented in 36 KB dual-port memory cells, meaning two separate 18 KB address spaces can be accessed simultaneously. Adding to this, 40 to 740 DSP elements, including a pre-adder, a multiplier, an adder as well as an accumulator, are available.

8.4.3 Implementing DT ensembles on von Neumann Architectures

As discussed in section 8.1 there are at least two ways to implement DTs, namely as native trees and as if-else trees. In addition, we can leverage SIMD instructions that give us two additional implementation schemes for DTs.

Native Implementation: The native implementation stores the nodes of a tree inside an array and then iterates over that array following the decisions at each node. Typically, the nodes of a tree are represented as a single entity containing the split value, a pointer to the children, and the prediction. An example of this entity can be found in Figure 8.8. We explicitly denote the corresponding data types to model the size of the node. `boolean_t` denotes a boolean data type, and all other

types represent an unsigned integer of necessary size except `split_t`, which might also be a float depending on the input features. Since loading words smaller than s_w requires an extra $lane$ access, we align the size of each data type towards s_w. Thus, if values below 2^{s_w} need to be represented, they are stored inside a variable of size s_w. If values above 2^{s_w} need to be represented, we can store them in the next smallest multiple of s_w. The complete size of a node is given by $s_n = 2 \cdot sizeof(\texttt{boolean_t}) + sizeof(\texttt{feature_t}) + sizeof(\texttt{split_t}) + 2 \cdot sizeof(\texttt{node_t})$. For the following, we assume that data types fit into scalar registers of size s_w, and thus only one $load$ instruction is necessary to access each field of a node. A tree is represented by all its nodes, which can be stored in a simple array structure. The variables `leftChild` and `rightChild` point to the next index in that array.

```
1: entity Node:
2:     boolean_t prediction          ▷ true or false
3:     boolean_t isLeaf              ▷ true or false
4:     feature_t feature             ▷ feature index
5:     split_t split                 ▷ split value
6:     node_t leftChild              ▷ index of left child
7:     node_t rightChild             ▷ index of right child
```

FIGURE 8.8: Implementation of single decision tree node for the native implementation in the theoretical von Neumann architecture.

```
 1: function PREDICT(x, tree)
 2:     i ← load(0)                                   ▷ 1 clock cycles
 3:     r1 ← load(tree[0].isLeaf)                     ▷ 1 clock cycles
 4:     while cmp_eq_false(r1) do                     ▷ 2 clock cycles
 5:         r1 ← load(tree[i].feature)                ▷ 1 clock cycles
 6:         r1 ← load(x[r1])                          ▷ 1 clock cycles
 7:         r2 ← load(tree[i].split)                  ▷ 1 clock cycles
 8:         if cmp_le(r1, r2) then                    ▷ 2 clock cycles
 9:             i ← load(tree[i].leftChild)           ▷ 1 clock cycles
10:         else
11:             i ← load(tree[i].rightChild)          ▷ 1 clock cycles
12:         r1 ← load(tree[i].isLeaf)                 ▷ 1 clock cycles
        return tree[i].prediction                     ▷ 1 clock cycles
```

FIGURE 8.9: Implementation of a decision tree using the native approach in the theoretical von Neumann architecture.

For prediction, one can traverse the tree starting from index zero until a leaf is reached. Let `tree[i]` denote node i in the array `tree`, then we may access a field of node i by writing `tree[i].field`. If i is already present inside a register, we allow `load` instructions directly on `tree[i].field` with one clock cycle. A native approach implementation can be found in Figure 8.9. First, we initialize the intermediate register `i` to zero and load the field `tree[0].isLeaf` into another intermediate register $r1$ (line 2 and 3). Then we compare the contents of register $r1$ against `false` (line 4). If we have not reached a leaf yet, we start to execute the `while`-loop. In essence, we need to compare the current split point `tree[i].split` with the corresponding feature value `x[tree[i].feature]`. In order to access `x[tree[i].feature]`, we first need to load `tree[i].feature` into register $r1$ and then load `x[r1]` into the same register (line $5-6$). Additionally, we load `tree[i].split` into register $r2$. Once $r1$

and $r2$ contain the desired values, we can compute the comparison and branch accordingly (line $8-11$). Depending on the outcome of the branch, we update i either with `tree[i].leftChild` or `tree[i].rightChild`. At the end of each loop iteration, we need to update the content of register $r1$ with `tree[i].isLeaf` (line 13). Once a leaf node is reached, the corresponding prediction is returned. The number of clock cycles needed for each instruction is depicted at the end of each line. Notice that we either need to update register i with the left or right child taking the same time. Thus, one pass through the loop uses a total of 9 clock cycles. Given the expected number of comparisons $\mathbb{E}[\Lambda]$, we see that one prediction for a tree takes

$$c_{CPU}^{native} = 9 \cdot E[\Lambda] + 3 \tag{8.7}$$

clock cycles in total.

If-else-trees

In the native implementation, the CPU indirectly accesses the field `tree[i].feature` to load the corresponding feature value from x. This increases the number of clocks required, reducing the overall throughput. We exploit the observation that `tree[i].feature` and `tree[i].threshold` are already known at compile time, which enables us to unroll the tree in its if-else structure replacing `tree[i].feature` and `tree[i].threshold` respectively by their constant values k_i and t_i. This way, the CPU does not need to access `tree[i].feature`, but can directly load the necessary feature $x[k_i]$. Figure 8.10 shows a scheme of this approach in the theoretical von Neumann model. This approach does not require intermediate values to be stored, but only two registers are required for every comparison (see line $2, 3$ or line $5, 6$). Since k_i is a constant, there is no need to compute the address for accessing x, but the CPU can directly load the array entry $x[k_i]$ needed. Once the corresponding split value s_i is also loaded into an intermediate register, the comparison can be performed (line 4 and line 7). After this, both registers are free to be used by the next $if-$branch. Once a leaf node is reached, the prediction is given by a constant `true` or `false`, which is loaded into a register for the function's caller to use. For every if-branch taken, there are 4 operations, and thus 4 clocks are needed. Taking the expected height of the tree into account, we see that `if-else` trees are expected to need

$$c_{CPU}^{if-else} = 4 \cdot E[\Lambda] + 1 \tag{8.8}$$

clock cycles.

SIMD Implementation

If-else reduces memory accesses but still needs to perform $E[\Lambda]$ comparisons to traverse the tree. The vectorization unit of the CPU can perform up to v comparisons in one clock cycle and thus offers the possibility to traverse a tree in only $\lceil \frac{E[\Lambda]}{v} \rceil$. As already mentioned, Kim et al. presented a SIMD implementation in [KCS+10], which we will use as a basis here. In order to utilize vector instructions, we store up to v feature indices, v split values, and v children in one node, as shown in Figure 8.11. We prepare these structures during compile-time and hence can place them in continuous memory. Thus, loading them into the vectorization register requires just one cycle. Figure 8.12 shows how the prediction can be performed. First, one loads the v split values as well as the indices for the features into vector register $v1$ and $v2$.

```
1: function PREDICT(x, tree)
2:     r1 ← load(x[k_0])                          ▷ 1 clock cycles
3:     r2 ← load(t_0)                             ▷ 1 clock cycles
4:     if cmp_le(r1, r2) then                     ▷ 2 clock cycles
5:         r1 ← load(x[k_1])                      ▷ 1 clock cycles
6:         r2 ← load(t_1)                         ▷ 1 clock cycles
7:         if cmp_le(r1, r2) then                 ▷ 2 clock cycles
8:             ...
9:         else
10:            ...
11:    elsereturn tree[i].prediction              ▷ 1 clock cycles
```

FIGURE 8.10: Implementation of a decision tree using the if-else approach in the theoretical von Neumann architecture.

Second, we need to load all the corresponding feature values from x into another register $v2$. Since we cannot guarantee that we require a continuous part of x, we need to gather different parts of x using the $gather$ instruction. Third, we can perform the actual comparison, which is saved into a vector register of size s_v. Since we are only interested in the v bits corresponding to the comparison, we can use the $extract$ instruction to extract a bitmask of size v. Last, we can reinterpret this bitmask as an index and use it to access the next child node in the tree. Unfortunately, we cannot access `tree[i].children[mask]` directly since we would need to perform two array look-ups in one instruction. Thus, we split this indirect access and first load the base address `tree[i].children` into register $r1$ and then perform the actual lookup depending on $r1$. One pass through the `while`-loop takes 10 cycles. Initialization takes again 2 cycles, and returning the prediction takes another clock cycle. The question remains, how many loop iterations are expected to be performed for a tree?

```
1: entity Node:
2:     boolean_t prediction        ▷ true or false
3:     boolean_t isLeaf            ▷ true or false
4:     feature_t features[v]       ▷ v feature indices
5:     split_t splits[v]           ▷ v split values
6:     node_t children[v]          ▷ v indices of children
```

FIGURE 8.11: Implementation of single decision tree node for the SIMD implementation in the theoretical von Neumann architecture.

Depth-first SIMD

Given we have a skewed distribution of positive and negative labels inside the decision tree, there might be a path l from the root node to a leaf node which is taken the majority of the time. In other words, the probability of using a path π_i may be much higher than any other path, thus $p(\pi_i) \gg p(\pi_j)\ \forall i \neq j$. We can utilize this fact by performing v comparisons on the most probable path. Then, in the best case, we can skip up to v comparisons if the first v nodes of π_i match the first v nodes in $v1$ and $v2$. However, in sub-optimal cases where only the first $u < v$ nodes of both paths match, we can only skip the first u nodes. In the worst case, only the first node in both paths matches, and thus we effectively perform only one comparison. Therefore, we call this type of comparison strategy a depth-first comparison in which we

```
1:  function PREDICT(x, tree)
2:      i ← load(0)                              ▷ 1 clock cycles
3:      r1 ← load(tree[0].isLeaf)                ▷ 1 clock cycles
4:      while cmp_eq_false(r1) do                ▷ 2 clock cycles
5:          v1 ← load(tree[i].splits)            ▷ 1 clock cycles
6:          v2 ← load(tree[i].features)          ▷ 1 clock cycles
7:          v2 ← gather(x[r2])                   ▷ 1 clock cycles
8:          v2 ← compare(v1, v2)                 ▷ 1 clock cycles
9:          mask ← extract(v2)                   ▷ 1 clock cycles
10:         r1 ← tree[i].children                ▷ 1 clock cycles
11:         i ← load(r1[mask])                   ▷ 1 clock cycles
12:         r1 ← load(tree[i].isLeaf)            ▷ 1 clock cycles
        return tree[i].prediction                ▷ 1 clock cycles
```

FIGURE 8.12: Implementation of a decision tree using the SIMD approach in the theoretical von Neumann architecture.

always perform v comparisons on the most probable path of the tree. The number of loop iterations we expect to perform can be expressed as the number of successful Bernoulli experiments in a row. Let k be the most probable path in the tree and let $\pi_k[0:i]$ denote the sub-path of it, which is given by the first i comparisons, then the number of expected loop iterations is given by the sum of every sub-path in π_k:

$$\mathbb{E}^{SIMD}[\Lambda] = \sum_{i=1}^{L_k} p_{\pi_k[0:i]} \cdot i \tag{8.9}$$

This leads us to an expected number of clock cycles needed:

$$c_{CPU}^{depth-first} = 10 \cdot \frac{\mathbb{E}[\Lambda]}{\min(v, \mathbb{E}^{SIMD}[\Lambda])} \tag{8.10}$$

Breadth-first SIMD

In a depth-first comparison, we try to skip up to v comparisons in one clock cycle. This approach works well if there are only a few paths in the tree which are taken the majority of the time. If the decision tree is more balanced, we are more likely to take different paths with each observation x and thus will effectively only perform one comparison per clock. In order to utilize SIMD instructions in a more controlled way, we can perform multiple comparisons on different paths in the same instruction. We can do this because each node in the tree has only 2 children. Regardless of the outcome of the comparison at a parent node, we will have to perform one of the two comparisons given by the children. Thus, if we perform the comparison at the current node, as well as both children with one vectorization instruction, we make sure that two of the comparisons are useful. More formally, we are guaranteed to skip at-least $\lfloor \log_2(v+1) \rfloor$ comparisons with one instruction. In case v is not an exponential of 2, we still have some comparison entries left in the vectors. We can use the remaining $m = v - (2^{\lfloor \log_2(v+1) \rfloor} - 1)$ slots in the vectorization units and use them to perform the most probable comparisons on the next layer. Given the highest probabilities $p_1, \ldots, p_m$ of that layer, we will match one of the comparisons needed with expectation $\sum_{i=1}^{m} p_i$. This leads us in total to an expected number of clock cycles

needed for a breadth-first approach of

$$c_{CPU}^{breadth-first} = 10 \cdot \frac{\mathbb{E}[\Lambda]}{\lfloor \log_2(v+1) \rfloor + \sum_{i=1}^{m} p_i} \tag{8.11}$$

(Soft) Majority Vote Implementation

So far, we have discussed the implementation of decision trees. We can use these implementations as building blocks to implement a tree ensemble by applying one tree after another and by keeping track of the sum of votes[4]. Figure 8.13 shows the implementation of a majority vote. First, we create an array in which we store the sum of class probabilities from the trees. Then, we perform the prediction using the first tree and save the result in an intermediate register $r1$ (lines 5 and 6). The prediction is then added to the `predictions` array, and the overall procedure is repeated for every tree in the tree ensemble. In the end, we compute the average by multiplying the probabilities with $\frac{1}{M}$. Let c_{CPU}^{m} denote the number of cycles needed for method m and tree i and assume that we do not use SIMD operations for the averaging[5], then we need a total number of cycles for the majority vote of:

$$C_{CPU}^{total} = 2C + \sum_{i=1}^{M} (c_i^m + C) = C\,(M+2) + \sum_{i=1}^{M} c_{CPU}^{m} \tag{8.12}$$

1: **function** MAJORITYVOTE($x, tree_1, \dots, tree_M$)
2: `predictions` $\leftarrow load(0, \dots, 0)$ ▷ 1 or C clock cycles
3: $r1 \leftarrow predict(x, tree_1)$ ▷ c_{CPU}^{m} clock cycles
4: `predictions` $\leftarrow$ `predictions` $+ r1$ ▷ C or $\lceil \frac{C}{v} \rceil$ clock cycles
5: $\dots$
6: $r1 \leftarrow predict(x, tree_M)$ ▷ c_{CPU}^{m} clock cycles
7: `predictions` $\leftarrow$ `predictions` $+ r1$ ▷ C or $\lceil \frac{C}{v} \rceil$ clock cycles
8: `predictions` $\leftarrow \frac{1}{M} \cdot$ `predictions` ▷ C or $\lceil \frac{C}{v} \rceil$ clock cycles
return `predictions`

FIGURE 8.13: Implementation of a majority vote in the theoretical von Neumann architecture.

8.4.4 Implementing DT Ensembles on FPGAs

Field programmable gate arrays offer reconfigurable hardware and thus do not offer any computing architecture but are free to mimic every architecture required. In a naive approach, we can simply reuse the implementations presented so far and find the same theoretical conclusions as discussed. This, however, would not take the very flexible nature of FPGAs into account. First, it has to be noted that register accesses do not need to be aligned towards given word sizes s_w or vector sizes s_v, but they can exactly be tailored to the problem at hand. Thus, we can use the same node entity depicted in Figure 8.8, but can also tailor the corresponding data sizes exactly to the specific tree at hand. Similar to the CPU implementation, we assume that the complete forest fits into the block ram of the FPGA, and thus, we only operate on

[4] If individual trees are weighted, we can include the weights into the leaf predictions as discussed earlier.

[5] If SIMD is used then we replace C by $\frac{C}{v}$.

block memory or the logic blocks of the FPGA.

Native Implementation

First, we analyze the native implementation shown in Figure 8.9 for FPGAs. The initialization before the `while`-loop can be executed during power-on of the FPGA and thus takes no time (line $2-3$). Unlike for the CPU, we can issue two `load` instructions at the same time on the FPGA, but each load takes two clock cycles because of address resolution. Thus, we can perform the first `load` operations on $r1$ and $r2$ in parallel (line 5 and 6). However, the remaining `load` instructions depend on each other and thus need to be executed in sequence, leading to a sequence of 4 `load` instructions inside the `while`-loop. Additionally, two comparisons are performed (line 4 and 8), which take 1 clock cycle each, leading to:

$$C_{FPGA}^{Native} = (0 + 2 \cdot 4 + 2) \cdot \mathbb{E}[\Lambda] + 1 = 10 \cdot \mathbb{E}[\Lambda] + 1 \tag{8.13}$$

In order to estimate the resources used by this implementation, we observe that two functional units for comparisons and two functional units for address resolution are required. Additionally, 3 registers are needed giving a total of

$$r^{Native} = 2 \cdot r_{addr} + 3 \cdot s_w + r_{eq} + r_{le} \tag{8.14}$$

resources used by this implementation.

If-Else Implementation

In the native implementation, we observe that memory access is a costly operation since we have to perform address resolution first. Similar to the CPU implementation, we can bypass this problem if we unroll the tree in its if-else structure as depicted in Figure 8.10. As before, we need to access a constant t_i, which is known at compile time. Similarly, we see, that we need to access $x[k_i]$, where k_i is also a constant. Thus, the memory address of $x[k_i]$ is known at compile time, and no address resolution unit is required. So far, we implicitly assumed that the observation x has already been copied into the FPGA's block memory. It is reasonable to assume, that we could instead copy it directly into the FPGA CLBs cells because we need to communicate with the FPGAs in any case[6]. Then, we can hard-wire the entries in x and their split thresholds t_i directly into the corresponding comparators. This way, the operands for comparison do not need to be loaded, but only the comparison itself needs to be performed, taking 1 clock cycle in total:

$$c_{FPGA}^{if-else} = \mathbb{E}[\Lambda] + 1 \tag{8.15}$$

Since we hard-wire constants into the comparison blocks, we cannot reuse any comparison unit and thus have to implement the comparison of every node of the complete tree. Additionally, we need to store all split thresholds t_i as well as the entire observation x inside the CLBs of the FPGA. Given we implement a tree with n nodes, we need

$$r_{FPGA}^{if-else} = n \cdot r_{le} + n \cdot \left\lceil \frac{s_w}{t} \right\rceil + d \cdot s_w \tag{8.16}$$

[6]Assuming x is always copied into the block ram due to hardware constraints, we can load it into the FPGAs CLBs in $\lceil \frac{d}{2} \rceil$ clock cycles.

resources.

SIMD Implementation

FPGAs are well suited for vector operations and thus, the SIMD implementation of the CPU can directly be mapped onto the FPGA as presented in Figure 8.12. In order to access v features at once, we again store the complete tree using the FPGAs CLBs. However, unlike if-else trees, we need to keep track of the current node i and issue `load` instructions towards the memory using CLBs. Thus, memory access still has a delay of two clock cycles. Again, the number of expected loop iterations depends on the comparison strategy. However, the number of clock cycles per iteration is different. The two load operations before entering the `while`-loop can be performed during power-on and thus require no clock cycles (line $2, 3$). Loading values into register $v1$ and $v2$ can be performed in parallel (line $5, 6$), taking two cycles. Additionally, we need to gather the entries in x, taking another two cycles. The comparisons in lines 4 and 8 can be performed in one clock cycle each. Generating the corresponding bitmask from the comparison is free on the FPGA since we can hard-wire the comparison bits directly it into the next functional unit. After that, three loading operations are performed in sequence, leading to a total of 12 clock cycles. Looking at the resource consumption of this implementation, we observe that we need a total of v functional units for address lookup since the gather instruction performs v lookups in parallel. Also, we need to perform v comparisons using v comparators. Additionally, another comparator, which compares $r1$ against a fixed value, is required. Last, we need to materialize all nodes, as well as the vector registers and $r1$ and i with the FPGAs logic cells leading to a total of:

$$r_{FPGA}^{SIMD} = v \cdot r_{addr} + v \cdot r_{le} + v \cdot r_{eq} + n \cdot node_t + 2 \cdot v \cdot s_w + 2 \cdot s_w + d \cdot s_w \quad (8.17)$$

DNF Implementation

So far, we mapped CPU implementations to FPGAs, not taking the FPGA's flexible nature into account. To do so, we can formulate an extreme case of the SIMD trees, in which we perform all comparisons in one clock cycle. This can be done, by observing two things: First, the comparisons in a tree do not depend on each other. Only once we perform the actual prediction, we need to traverse the tree. Thus, we can perform all comparisons of all nodes in the tree given x first and then traverse the tree given the pre-computed comparisons. Second, a tree can be represented by a disjunctive normal form (DNF). In a disjunctive normal form, a boolean function is represented as a series of conjunctions connected by disjunctions. We can view the comparison performed at node i as a boolean variable, which is either true or false. Then, a particular path from the root node to a leaf is represented as a conjunction of all – possibly inverted – comparisons on that path. Let c_i denote the comparison for node i, then the DNF of the tree depicted in Figure 8.1 is for example $c_{x_2} \vee (\neg c_{x_2} \wedge \neg c_{x_3} \wedge c_{x_1})$. The DNF for a given tree is independent of the specific observation x but only depends on the structure of the tree. Figure 8.14 shows an implementation of this approach.

Similar to the if-else trees, we can store the observation x inside the FPGAs CLBs and hard-wire the split values and features directly into the comparators. Thus, given n nodes in the tree, we have to perform n comparisons in parallel taking only 1 clock cycles. The DNF computation of a tree thus consists of three operations. First, necessary variables need to be inverted, then conjunctions are computed, and finally, the disjunction can be evaluated. In a naive approach, this takes 3 clock cycles, but

```
1: function PREDICT(x, tree)
2:     r1 ← cmp_le(r1, x[k_1], t_1)                ▷ 1 clock cycles
3:     r2 ← cmp_le(r2, x[k_2], t_2)                ▷ 1 clock cycles
4:     ...
5:     r1 ← computeDNF(r1, r2, ...)               ▷ 1 clock cycles
        return predictions[r1]                     ▷ 1 clock cycles
```

FIGURE 8.14: Implementation of a decision tree using the DNF approach in the theoretical FPGA.

since the tree structure is known, we can directly encode this into the look-up tables of the FPGA similarly to the if-else trees. Thus, computing the DNF takes only one clock cycle. Adding another clock cycle for getting the prediction value and returning the prediction values, we see, that DNF-Trees only require

$$c_{FPGA}^{DNF} = 3$$

clock cycles. In order to traverse the complete tree in only one clock cycle, we trade run-time for space. In short, we need n comparators performing comparisons in parallel. Additionally, we need n times s_w bits to materialize all split thresholds using the FPGAs CLBs and $d \cdot s_w$ bits to store the observation x. Computing the DNF for the tree can be viewed as computing a boolean function with $2^{\max\{L_1,\dots,L_k\}}$ input variables $DNF\colon \{0,1\}^n \rightarrow \{0,1\}$. We can distribute this across multiple logic blocks so that in total, this implementation needs

$$r_{SIMD}^{DNF} = n \cdot r_{le} + n \cdot s_w + d \cdot s_w + \left\lceil \frac{6 \cdot n}{t} \right\rceil + 1 \tag{8.18}$$

resources.

(Soft) Majority Vote Implementation

For FPGAs, we will follow the same general procedure as for von Neumann CPUs shown in Figure 8.13 to implement the majority vote. However, unlike CPUs, FPGAs do not need to execute trees in sequence but can execute the trees in parallel leading to

$$C_{FPGA}^{total} = 2C + \max\{c_1^m, \dots, c_M^m\} \tag{8.19}$$

with a resource consumption of

$$r_{total}^{FPGA} = M \cdot r^m + r_{mac} \tag{8.20}$$

where r_{mac} denotes the resource consumption of a multiply-accumulate circuit. Here, the special case of hard voting schemes for two classes in which each tree either predicts $true$ or $false$ is interesting. In order to compute the majority vote in this case we need to count the number of 1-bits inside the bit vector of all predictions where 1 corresponds to $true$ and 0 corresponds to $false$. This is known as the hamming weight of a bit vector and has been studied in literature [SS13]. Similar to comparing two integers, the hamming weight of a bit vector with M bits can be implemented using $r_{ham} = \lceil \log_2(M+1) \rceil$ CLBs, taking one clock cycle to complete its operation. After that, we can compare the value against $\lfloor \frac{M}{2} \rfloor$, also taking one clock cycle requiring r_{le}

resources. Thus, the total clock delay of a tree ensemble given method m is given by:

$$C_{total}^{FPGA} = C_{FPGA}^{m} + 2 \tag{8.21}$$

The amount of resources used is given by:

$$r_{total}^{FPGA} = M \cdot r^{m} + r_{ham} + r_{le} \tag{8.22}$$

Part IV

Discrete Classifiers and Small Devices in Practice

9 | Software and Libraries

The last section already highlighted the impact of different implementations on the performance of ML models. Naturally, the software ecosystem surrounding all parts of the ML model from data processing, model training, and deployment plays an equally, if not more important role when applying ML methods. The next section quickly surveys existing libraries and software frameworks to train (discrete) classifier ensembles that have been used throughout this thesis, whereas section 9.2 and section 9.3 discuss two software frameworks that have been created as a part of this thesis.

9.1 Overview of existing Software

In order to deploy machine learning models they must be trained first. In the following, we will give a compact overview of existing software that has been used during this thesis. For a more comprehensive overview of ML frameworks, we refer interested readers to [NDB+19].

Weka (`https://www.cs.waikato.ac.nz/ml/weka/`) is arguably the oldest software framework that offers a multitude of different data processing techniques as well as ML algorithms, whereas RapidMiner (`https://rapidminer.com/`) is younger, but with a comparable feature set. Last, scitkit-learn (`https://scikit-learn.org/stable/`) is the youngest, but arguably also the most widely used general-purpose machine learning framework. These frameworks offer a similar set of features and are still actively in development. It is interesting to note, that both, older and more recent methods are actively added to these frameworks. For example, minimal cost complexity decision tree pruning was originally proposed in the 1980s [BFSO], but only added to scikit-learn `0.22` in 2020, whereas the other frameworks do not support it at all[1].

All these general-purpose frameworks also support discrete classifier ensembles such as gradient boosted trees or random forests, but there are three additional libraries dedicated to training gradient boosted decision trees. XGBoost (`https://xgboost.ai/`) started as a GPU implementation for gradient boosted decision trees and has since become the de-facto standard boosting library. The original paper proposed a novel column layout for distributing features on the GPU in combination with a regularization term during boosting. In addition, splits are computed heuristically instead of iterating all possible splits leading to a very efficient and competitive system [CG16]. Similar libraries such as LightGBM (`https://github.com/microsoft/LightGBM`, [KMF+17]) and CatBoost (`https://catboost.ai/`, [PGV+18]) have been proposed that implement similar ideas for DT boosting. The performance of the three frameworks is comparable, although there are subtle differences in their models

[1] However, they support other post-pruning methods for DTs.

[APP+18]. For example, CatBoost uses balanced trees whereas XGBoost and LightGBM utilize unbalanced trees.

Looking at deep learning libraries there are two main frameworks namely TensorFlow (https://www.tensorflow.org/) and PyTorch (https://pytorch.org/) available. Both frameworks offer automatic differentiation of arbitrary computation graphs constructed from a set of basic operations such as convolution, matrix multiplication, etc. Both frameworks support the execution via CPUs and GP-GPUs (e.g. CUDA) as well as more specialized hardware such as TPUs. The main difference between both frameworks is that PyTorch uses a dynamic graph structure whereas TensorFlow uses a static graph meaning that the computation graph remains the same during execution. While this difference is neglectable on a surface level for most feed-forward networks it can lead to subtle differences in the user experience as well as to differences when implementing recurrent neural networks (RNN). In TensorFlow, the maximum number of recurrent steps must be set beforehand, whereas in PyTorch it can be changed during execution. Both frameworks are actively maintained and improved. In addition, there are multiple extensions available such as Keras (https://keras.io/) or PyTorch Lightning (https://www.pytorchlightning.ai/) that try to reduce the complexity of using these frameworks by providing a common interface implementing a set of best practices. Most interesting for this thesis is the larq (https://larq.dev/) framework that adds native support for binarized neural networks to TensorFlow for training and deploying models. Unfortunately, the official beta release of this framework was in May 2019 (https://github.com/larq/larq/releases/tag/v0.1.0) and hence not available during parts of this research. Last, we note, that researchers seem to gravitate more towards PyTorch whereas in industry TensorFlow is more widely used (https://tinyurl.com/2mdb7n59).

Last, looking at online learning there are two main frameworks available. Massive Online Analysis (MOA, https://moa.cms.waikato.ac.nz/) is arguably the most-used online learning framework to date and is still under active development. It contains many of the most recent advances in online learning and many papers and advancements are nearly indistinguishable from the software contribution to this framework. Unfortunately, MOA does not integrate well into the current machine learning ecosystem that gravitates more toward Python. Hence, River (https://riverml.xyz/) attempts to re-implement moa in Python. However, River is currently in its early development stage and is not comparable to MOA in terms of performance and features.

9.2 PyPruning

Large parts of this thesis encapsulate theoretical and algorithmic results that are usually backed by experiments. Hence, it is inevitable that related methods, as well as novel ideas for model pruning discussed in section 6, had to be implemented for the corresponding experiments. Contrary to the very active field of deep learning inferencing and optimization (discussed in the next section), research in ensemble pruning moves at a much slower pace. Somewhat surprisingly, however, there does not seem to be *any* common library for ensemble pruning. Hence, the ensemble pruning library PyPruning was implemented as part of this thesis. PyPruning is – to the best of our knowledge – the only available implementation of common ensemble pruning algorithms. It follows the taxonomy of pruning algorithm presented in section 6 and currently offers six types of pruning algorithms:

- **RandomPruningClassifier**: Selects a random subset of classifiers. This is mainly used as a baseline.
- **RankPruningClassifier**: Rank each classifier according to a given metric and then select the best K classifier.
- **ClusterPruningClassifier**: Cluster the classifiers according to a clustering method and then select a representative from each cluster to form the sub-ensemble.
- **GreedyPruningClassifier**: Selects one classifier per round in a total of K rounds. In each round, the best classifier for the current sub-ensemble is selected.
- **MIQPPruningClassifier**: Constructs a mixed-integer quadratic problem and optimizes this to compute the best sub-ensemble.
- **ProxPruningClassifier**: Minimize a (regularized) loss function via (stochastic) proximal gradient descent over the ensemble's weights.

PyPruning is designed to be flexible while implementing the most common pruning methods. To do so, there is a common interface for each class of pruning algorithm that implements the general pruning approach as discussed previously but receives specific functions that reproduce the specific pruning method from the literature. For example, individual error (IE pruning [JLFW17]) pruning can be implemented by passing the corresponding error function (c.f. Figure 9.1) to the `GreedyPruningClassifier` object. Similarly, cluster pruning via K-Means can be implemented by passing scitkit-learns `sklearn.cluster.KMeans` object to the `ClusterPruningClassifier` object whereas cluster pruning with agglomerative clustering is implemented by passing `sklearn.cluster.AgglomerativeClustering` to the same object.

```
def individual_error(i, ensemble_proba, target):
    iproba = ensemble_proba[i,:,:]
    return (iproba.argmax(axis=1) != target).mean()
```

FIGURE 9.1: Example of a custom metric for pruning in PyPruning.

In total, PyPruning offers 16 different pruning algorithms that have been proposed in literature [GLL+18, LWZB10, JLFW17, MD97, MMS04, LYZ12, COMC16, ZBS06, MS06, GRF00, LO01, BH03], although more combinations are possible. If a new pruning method does not fit the established taxonomy (e.g. L_1 pruning) then a custom pruner can be implemented as well. In this case, the `PruningClassifier` class has to be implemented which requires a `prune_` method. This function receives a list of all predictions of all classifiers as well as the corresponding data and targets and is supposed to return a list of indices corresponding to the chosen estimators as well as the corresponding weights. An example can be found in Figure 9.2. PyPruning is implemented in Python and available under https://github.com/sbuschjaeger/pypruning. A more thorough documentation can be found under https://buschjaeger.it/PyPruning/html/index.html.

```python
class PruningClassifier(ABC):
    def __init__(self):
        # omitted for space reasons

    @abstractmethod
    def prune_(self, proba, target, data = None):
        # to be implemented

    def prune(self, X, y, estimators, classes = None, n_classes = None):
        """ Main entry point for pruning """
        # omitted for space reasons

    # ...
    # Additional function are omitted for space reasons
    # ...
    def _individual_proba(self, X):
        """ Helper function """

class RandomPruningClassifier(PruningClassifier):

    def __init__(self):
        super().__init__()

    def prune_(self, proba, target, data = None):
        n_received = len(proba)
        n_est = self.n_estimators
        if n_est >= n_received:
            idx = range(0, n_received)
            weights =  [1.0 / n_received for _ in range(n_received)]
        else:
            idx = np.random.choice(range(0, n_received),size=n_est)
            weights = [1.0 / n_est for _ in range(n_est)]

        return (idx, weights)
```

FIGURE 9.2: Example of a custom pruner in PyPruning.

9.3 Fastinference

As argued before, the deployment and application of trained models are one of the most important aspects of the ML pipeline when solving real-world problems with machine learning. Hence, it is not surprising that the exchange of pre-trained models, as well as their deployment, has been considered by almost all the major companies. The following contains a survey of existing tools and libraries at the time of writing this thesis. I do not attempt to make a complete survey here, nor do I think that the following list will be accurate in the future due to the volatility of the current market. Nevertheless, such an overview not only gives valuable pointers to existing tools, but it also places the software that originated as part of this thesis properly in the context

of currently existing software.

Exchange Formats for ML models

The goal of an inference engine is to execute a pre-trained ML model as resource-efficient and / or as fast as possible. Hence, loading the model is the first step which, in turn, requires a common model exchange format. During this thesis, the Open Neural Network Exchange Format ONNX (https://onnx.ai/) emerged as one of the standard formats to exchange trained neural networks between different frameworks. ONNX offers a common language to express computation graphs in which each node represents a computation and tensors are passed between the nodes to exchange information. ONNX is – from an industry point of view – in a relatively early standardization state, and it is still actively changing. However, most basic operations such as Generalized Matrix Multiplication (GEMM) or convolutions over multiple dimensions are firmly standardized by now. One of the major drawbacks of ONNX is that it does not standardize the structure of a computation graph, but only offers the basic operation which can be used inside of it. This freedom also has the unfortunate side effect that the same neural network can be expressed in a variety of different ways, and often times tools output very different ONNX files for the same neural network architecture. Hence, the *same* network can have different runtimes using the *same* inferencing tool because the ONNX files were exported by two different training frameworks. Looking beyond neural networks, there is the Predictive Model Markup Language PMML. PMML is an XML dialect for ML model exchange that is much older than ONNX and has a broader focus. It aims to encapsulate the entire inferencing pipeline, including the actual model, any data transformation and pre-processing, as well as additional operations such as outlier detection or missing data imputation. In this sense, PMML offers a much more complete approach to model inferencing. At the same time, it is also more restrictive than ONNX, because it does not allow for general computation graphs but defines a classifier as a single operation instead of decomposing it into a set of common operations. Last, it is noteworthy that each training framework usually also comes with its own set of exporting options, e.g. through the use of custom JSON formats (see e.g. https://tinyurl.com/5fdset26) or as custom binary objects (see e.g. https://tinyurl.com/ymwfzx3k). However, importing these custom formats into another framework is usually impossible unless a custom loader is implemented for this specific use case.

Deploying on Standard Hardware

Looking at deep learning, it seems that every major tech company nowadays maintains its own inferencing system for neural models: Googles Tensorflow Lite (https://www.tensorflow.org/), Facebooks's PyTorch Glow (https://github.com/pytorch/glow), Microsoft's ONNX Runtime (https://onnxruntime.ai/), Nvidia's Tensor RT (https://developer.nvidia.com/tensorrt), Huawei's Bolt (https://github.com/huawei-noah/bolt), Intel's OpenVino (https://tinyurl.com/36f3evcr), Xiaomi's MACE (https://github.com/XiaoMi/mace) just to list a few. While all these tools support a different subset of tensor operations standardized by ONNX they are also somewhat similar in the sense that they usually target smartphone-like platforms and are often benchmarked in image recognition tasks using common architectures such as MobileNet or EfficientNet. These tools oftentimes lack behind training frameworks

in terms of features because the optimization process for different neural network architectures as well as computational architectures is more time-consuming.

Looking beyond deep learning, the situation changes a bit. Here inferencing tools are much more scarce. Scikit-learn is one of the most-used frameworks for training "classical" ML models, but they do not offer a dedicated inferencing tool outside the framework. However, there is an adaptor for ONNX available that supports many of the classifiers in scikit-learn (http://onnx.ai/sklearn-onnx/) which are expressed as a series of tensor operations to form the computation graph. Similarly, there is an adaptor for PMML available (https://github.com/jpmml/sklearn2pmml). One interesting exception in this area is Treelite (https://treelite.readthedocs.io/en/latest/) which supports scikit-learn and other frameworks through their Python objects or a custom JSON format. It unrolls a given tree into its if-else structure and then automatically compiles the resulting C code to an inferencing binary that can be used by other programs as well. It is noteworthy that WEKA has a similar built-in mechanism that exports trees as their Java source code (see, e.g., `to_source` in https://tinyurl.com/mrksfct8). Last, cloud providers built upon these tools to offer ML model applications as-a-service in a scalable manner, e.g. over Amazon's AWS (https://aws.amazon.com/sagemaker/) or Microsoft's Azure (https://azure.microsoft.com/en-us/services/machine-learning/).

Deploying on Non-Standard Hardware

All the above tools are targeted towards existing computer architectures (e.g. ARM, X86, or CUDA), and depending on a given architecture, a different selection of tools is available. Moving towards non-standard hardware the situation becomes increasingly convoluted. A well-maintained list of available and/or announced AI chips can be found under https://github.com/basicmi/AI-Chip that currently contains over 60 startups, 17 IC vendors, 16 HPC firms, and 7 IP vendors. Usually, all of these AI chips come with their own tooling that is integrated into one of the existing frameworks. For example, Google's Tensor Processing Units are tightly integrated into TensorFlow and cannot be used directly with other frameworks such as PyTorch[2]. Similarly, firms such as Sambanova offer an intermediate compilation layer that translates PyTorch calls onto their custom AI chip (see e.g. https://tinyurl.com/2p9b6s58). FPGAs offer a middle-ground between GP-GPUs and these custom inferencing chips. Here, the two major FPGA manufacturers also offer solutions for neural network inference: Xilinx offers the Vitis Toolkit that offers a set of tools to compile ONNX models down to hardware code (https://tinyurl.com/2svfssjk) and similar Intel integrated FPGA support into their OpenVino toolkit (https://tinyurl.com/mt4u7nn5). Here, we again see a similar lack of features as mentioned above. Many of these tools only support a subset of the features offered by training frameworks in exchange for more optimized inferencing solutions for certain neural architecture and hardware combinations. In the context of this thesis, it is worth mentioning that Xilinx offers a dedicated toolkit to compile binarized neural networks for FPGAs (https://github.com/Xilinx/finn), but to the best of my knowledge, there does not exist a similar tool for e.g. tree ensembles.

Naturally, the question arises if the discrepancy between features offered by high-level training tools and optimized inferencing is desired or if there might be a way

[2]PyTorch and other frameworks increasingly add improved support for TPUs however.

to integrate both into a single tool. And indeed, there are efforts to develop a common machine-learning language. This way the high-level frameworks can implement their operations against this common language while inferencing engines can offer optimized implementations for a subset of operations on a target platform. The Multi Level IR (MLIR) Compiler Framework (https://mlir.llvm.org/), which is part of the LLVM infrastructure (https://llvm.org/) offers a standard way to express language dialects in the context of LLVM. A language developed using MLIR hence benefits from the existing tooling in the LLVM project and can therefore be developed much quicker. Tensorflow-lite offers a MLIR dialect that standardizes the operations offered by TensorFlow lite (https://www.tensorflow.org/mlir/tfl_ops) and an open-source version called tosa (https://mlir.llvm.org/docs/Dialects/TOSA/) for tensor operations is also available. Hence, a computation graph of neural networks can be expressed as a series of e.g. tosa instructions and any inferencing framework that supports this dialect can then execute this network. The inference engine does not need to know specifics about the neural network, but merely offers optimized implementations of the operations defined by the dialect, whereas a high-level framework can e.g. optimize the computation graph itself. The iree framework (https://github.com/google/iree) gives a reference on this approach that can read TensorFlow lite models, emit the corresponding dialect code, and then executes it using a CPU or GPU. Using these dialects it is also easy to integrate new operations by extending the dialect. For example, the larq (https://github.com/larq/compute-engine) framework extends the existing TensorFlow lite dialect by adding binarization operations to it. This way, the runtime engine of larq simply re-uses existing inference engines for regular TensorFlow lite operations and inserts optimized implementations for the binarization in the appropriate places.

Unfortunately, to the best of our knowledge *all* inferencing engines treat the model as a “second-class” citizen. They break the model into its computation graph, possibly perform optimizations on it and then use optimized implementations for specific computations in that graph. However, they do not utilize additional available knowledge about the model, such as the distribution of visited nodes in a decision tree or the fact that BNNs can be executed via XNOR and popcount instructions. Additionally, researchers are often pushing the limits of these tools by inventing new models or by deploying models to entirely new devices. Hence – even though the landscape is maturing – we do not find these tools to adequately address the issues of inferencing from a researchers’ perspective but are focused on the “average” industry use case.

Hence, to offer a tool that allows researchers to quickly test new inferencing strategies, FastInference has been developed as part of this thesis. FastInference is a machine learning model optimizer and model compiler that generates the optimal implementation for a given ML model and hardware architecture. It contains all the optimizations and implementations discussed in Section 8 for decision trees, but also supports additional models such as (binarized) neural networks. The goal of this tool is to ease the development of new optimization techniques as well as implementations for ML models. A general overview is depicted in Figure 9.3. FastInference works a three-step process: First, the pre-trained model is loaded from a JSON or ONNX file into an internal representation. This un-optimized model is then optimized by a number of possible optimizers. The resulting optimized model is further passed to the backend that generates the code for specific target architectures. The backend utilizes template-based code generation to generate optimized code for the *entire* model, e.g. outputting a single source file that represents the entire model.

FastInference focuses on three main concerns:

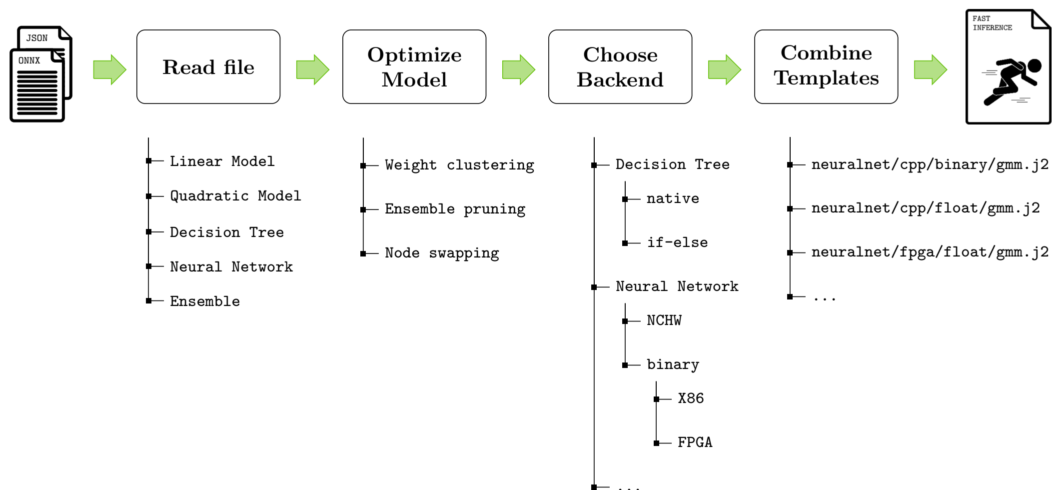


FIGURE 9.3: Workflow of FastInference. A pe-trained model is loaded from a JSON or ONNX file. Then a series of model optimizations are performed. The optimized model is passed to the backend, which then emits the optimized code using template-based programming.

- **Rapid prototyping is key**: New implementations for new target architectures as well as new optimizations should be easily added to the existing framework.
- **The model is central for inferencing**: Model optimizations should be available regardless of the target execution platform and implementations for a specific platform should be independent of specific optimizations. The ML model connects both.
- **The user comes first**: The user knows best what implementation and what type of optimizations should be performed. Hence, the generated code should be readable and users should be able to manually adapt the implementation if necessary.

The model plays a central role in this workflow as it connects the individual stages with each other. The base class of each model can be seen in Figure 9.4. Each model must implement `from_dict` and `to_dict` methods for loading / storing the model. Additionally, `predict_proba` must be provided that applies the model to the given data. The model itself offers an `optimize` function that applies all the given optimizations to it whereas `implementation` actually implements it.

Optimizations are decoupled from FastInference in the sense that they only require the model and any optimization-specific parameters. As an example, Figure 9.5 shows the *entire* code for the swap-optimization discussed in Section 8. Once the corresponding python file is placed in the search path of FastInference, it will be automatically loaded and the corresponding `optimize` function is executed. This way, optimizations can be developed independently of each other without interfering with multiple locations in the code.

Models and hardware architectures are generally too different to benefit from sharing parts of their implementation. For example, a decision tree is simply too different from a linear regression to share meaningful parts of their respective implementation. Similarly, a CPU implementation of BNNs is different from an FPGA implementation even though both might be done in a C-like language. Hence, there is also no coupling of the implementations themselves, and they can function completely

```python
class Model(ABC):
    def __init__(self, classes, n_features,
        category, accuracy = None, name = "Model"):
        # omitted for space reasons

    def optimize(self, optimizers, args):
        """ Applies a list of optimizers to this model """
        # omitted for space reasons

    def predict(self, X):
        """ Obtains predictions of this model to the given data """
        # omitted for space reasons

    def implement(self, out_path, out_name, impl_type, **kwargs):
        """ Implements this model using the impl_type"""
        # omitted for space reasons

    @abstractmethod
    def from_dict(self):
        # to be implemented

    @abstractmethod
    def to_dict(self):
        # to be implemented

    @abstractmethod
    def predict_proba(self,X):
        # to be implemented
```

FIGURE 9.4: Base class for every model in FastInference.

independently of each other. However, to ease the development of new backends for e.g. new programming languages, most of the implementations in FastInference utilize the jinja template engine (https://jinja.palletsprojects.com/en/3.0.x/). The jinja template engine is a template engine originally used for web development, that also offers more advanced features such as macro expansion. Most backends in FastInference utilize jinja templates to generate common interfaces and manage common data types e.g. by using `feature_type` as a placeholder for the actual data type whenever a feature is used. Figure 9.6 shows the shortened template that generates the if-else implementation discussed in Section 8 in C++. By using the `macro` keyword jinja is essentially expanding the tree by itself so that only minimal processing is required by Fastinference as depicted in Figure 9.7.

Fastinference is available under https://github.com/sbuschjaeger/fastinference and a more thorough documentation can be found under https://buschjaeger.it/fastinference/html/index.html. It is still under active development. Table 9.1 shows the currently supported backends that can be combined with the following optimizations:

- **Decision Trees**: swap, quantization

```python
def optimize(model, **kwargs):
    """
    Main entry point for every optimization.
    """
    remaining_nodes = [model.head]

    # Implement breadth-first traversal of the given DT
    while(len(remaining_nodes) > 0):
        cur_node = remaining_nodes.pop(0)

        # Swap the nodes depending on the probability
        if cur_node.probLeft < cur_node.probRight:
            left = cur_node.leftChild
            right = cur_node.rightChild
            cur_node.leftChild = right
            cur_node.rightChild = left

        if cur_node.prediction is not None:
            remaining_nodes.append(cur_node.leftChild)
            remaining_nodes.append(cur_node.rightChild)

    return model
```

FIGURE 9.5: Example of the swap optimization discussed in Section 8 implemented in FastInference.

- **Ensemble**: leaf-refinement, weight-refinement, pruning, weight-refinement, linear-merging
- **MLP + CNN**: merge and remove nodes

```
{% macro ifelse(nodes, node, tree_weight) %}
    {% if node.prediction is not none %}
    {% for pred in node.prediction %}
    pred[{{loop.index-1}}] += {{pred*tree_weight}};
    {% endfor %}
    return;
    {% else %}
    if (x[{{node.feature}}] <= {{node.split}}) {
        {{ifelse(nodes,node.leftChild, tree_weight)|indent}}
    } else {
        {{ifelse(nodes,node.rightChild, tree_weight)|indent}}
    }
    {% endif %}
{% endmacro %}

void predict_{{model.name}}(
    {{ feature_type }} const * const x,
    {{ label_type }} * pred
) {
    {{ifelse(model.nodes,model.head,weight)|indent}}
}
```

FIGURE 9.6: Template code for the if-else implementation of DTs discussed in Section 8 in FastInference for the C++ backend (simplified version).

TABLE 9.1: Supported languages in FastInference. CPP denotes the C++ programming languages whereas FPGAs denotes C++ code adapted for Xilinx High-Level Synthesis tool (https://tinyurl.com/ynrw4hmk). Haxe (https://haxe.org/) is a strictly-typed language for cross-compiling to e.g. Java, Javascript, PHP, Lua, etc. Tosa denotes the LLVM tosa dialect (https://mlir.llvm.org/docs/Dialects/TOSA/) that can e.g. be executed via iree (https://github.com/google/iree).

Method	cpp	FPGA	Haxe	Tosa
Decision Trees	ifelse, native			
Linear	native, unroll		native	
Discriminant	native			
MLP	native, binarized	binarized		native
CNN	native, binarized	binarized		native
Ensemble	native			

```python
def to_implementation(model,out_path,out_name,weight = 1.0,**kwargs):
    """
    Main entry point for every implementation.
    """

    # ...
    # Perform pre-processing
    # ...


    # Load the Jinja2 enviroment to find all templates
    env = Environment(
        loader=FileSystemLoader(
            os.path.join(os.path.dirname(os.path.abspath(__file__)))
        )
    )

    # Load the base.j2 that contains the actual implementation
    implementation = env.get_template('base.j2').render(
        model = model,
        model_name = model.name,
        weight = weight,
        # ...
    )

    # Load the header.j2 that contains the the C++ header
    header = env.get_template('header.j2').render(
        model = model,
        model_name = model.name,
        model_weight = weight
        # ...
    )

    # Store the implementation on disk.
    cpp_file = os.path.join(out_path, "{}.{}".format(out_name,"cpp")
    with open(cpp_file ), 'w') as out_file:
        out_file.write(implementation)

    header_file = os.path.join(out_path, "{}.{}".format(out_name,"h")
    with open(header_file), 'w') as out_file:
        out_file.write(header)
```

FIGURE 9.7: Realization of the if-else implementation of DTs discussed in Section 8 in FastInference for the C++ backend (simplified version).

10 | Case Studies with Discrete Classifiers

So far, this thesis discussed the theoretical and algorithmic properties of discrete classifier ensembles as well as the deployment of such models from an application-agnostic point of view. In this chapter, we will discuss the usage of discrete classifiers in more specific application contexts and show how the software that has been presented in the previous chapter can be used in real-world use cases.

10.1 Discrete Classifiers and the PhyNetLab

We start by showcasing the effectiveness of ensemble pruning in the context of the PhyNetLab warehouse [MRVT+18]. The PhyNetLab is a hardware test platform for the evaluation and analysis of IoT-based warehouses that has been developed as a part of the Collaborative Research Center SFB 876 (http://phynetlab.com/). It consists of small, ultra-low power, energy-neutral devices called PhyNodes that are placed on storage boxes inside the warehouse. The nodes are connected with various access points and form a wireless sensor network. Each node measures the current light intensity, the current temperature, and its acceleration as well as the Wifi signal strength to the access points in the warehouse. The goal is to estimate the current position of each node and thereby allow for efficient routing and detection of the storage boxes in the warehouse. While machine learning is ideally suited for such a task, the challenge lies in the deployment of the models. The PhyNode has a MSP430 MCU with a total of 64 KB of Ferroelectric Random Access Memory (FRAM) available, of which 48 KB are accessible by the instruction set. Roughly one-third of this memory is used for the operating system and drivers, leaving about 30 KB of memory for the top-level application, including the model. Subtracting additional top-level application code of around 10 KB leaves roughly 20 KB for the localization model [MRVT+18]. Hence, our goal is to find the best localization model that still fits into the remaining 20 KB of memory.

During 42 experiments conducted at various light and temperature levels, a total of 41 431 measurements at 31 different locations inside the warehouse have been taken. Each measurement consists of the acceleration (X,Y,Z) of the box, the current temperature, the current light intensity, as well as the Wifi signal strength to 3 different access points and a unique identifier for each box. During earlier experiments, we noted that the acceleration can have a huge impact on the performance because, in some experiments, the boxes would not be leveled, introducing biases into the acceleration. Hence, the model would overfit on this feature, although – by design – the acceleration of a (standing) box should not impact the performance of the classification. Hence, we ignore the acceleration for these experiments. To further reduce overfitting against specific environmental properties (e.g., a particular shiny or warm

day), we train the models on the data from the first 41 experiments and test them on the last experiment. The resulting training data has $N_{train} = 40\,444$ samples with $d = 6$ features, $C = 31$ classes, and the test set contains $N_{test} = 987$ test samples.

Recall that the localization model must fit into 20 KB of memory. The size of the implemented model is highly dependent on the specific implementation and can vary across models, MCUs and implementations. Hence, we perform a two-step process to find good models that fit into 20 KB of RAM: First, we train small models that approximately fit into the memory of the PhyNode. For these experiments, we focus on random forests and pruned tree ensembles discussed in chapter 6. A study of the performance of other classifiers, such as the SVM or Naive Bayes, is given in [MRVT+18], although the results are not entirely comparable due to slightly different training data.

Our experimental setup is as follows: We first train a base random forest via scikit-learn and then prune it with PyPruning. We estimate the size of the model by counting the total number of nodes n_{total} in all trees inside the forest and then by computing the size via $(17 + 4 \cdot C) \cdot n_{total}$ as described in section 6.4. In the second step, we use FastInference to generate the implementation of these models, automatically compile them and remove all models that result in an overflow during the cross-compilation, thereby leaving only models that can actually be deployed to the PhyNode. Unfortunately, if-else trees result in very large code sizes and hence would use too much memory, not fitting the MCU. Thus, we opted for native trees in this experiment. Some additional pre-processing was required to make the models fit into 20 KB: Recall that there are $C = 31$ classes, and hence a tree with 16 leaf nodes requires 2 KB to store the class probabilities in leaf nodes if a `float` variable is used. To reduce the memory consumption, we, therefore, employed a fixed-point quantization that scales each probability by a factor of $10\,000$ and rounds it down towards the next integer. This way, the probabilities in each leaf node can be stored inside a 2 Byte (i.e., a `short` variable), effectively halving the size. This operation is also implemented in FastInference, and we could not detect any change in the accuracy with this quantization in this experiment.

In a series of pre-experiments, we determined reasonable ranges for the hyperparameters of each algorithm so that the estimated model size is below 24 KB. Similar to the experiments in section 6.4, we train a base random forest with $M = 256$ trees and $n_l \in \{4, 8, 12\}$ leaf-nodes and then prune away trees from the ensemble. Each pruning method is tasked to select $K \in \{2, 4, 8\}$ trees. For DREP we used $\rho \in \{0.25, 0.3, \dots, 0.5\}$. For L1 and L1+LR we minimized the MSE over 20 epochs with the Adam optimizer using $\alpha = 0.01$, $|\mathcal{B}| = 1024$ and $\lambda \in \{1.0465, 1.0466, \dots, 1.047\}$. The code for this experiment is available under `https://github.com/sbuschjaeger/leaf-refinement-experiments`.

Table 10.1 shows the accuracy and F_1 score for the best models that still could fit on the PhyNode. As one can see, L1+LR offers the best predictive accuracy as well as the best F_1 score highlighting the usefulness of our approach. Moreover, we found that the accuracies seem to vary a lot between the different methods. For example, DREP is the worst method with 51.32% accuracy, whereas L1+LR is nearly 20 percentage points better with an accuracy of 71.04%. Given that all models are derived from the RF, these large differences seem surprising to us, but we could not find any errors in our evaluation pipeline. In particular, we made sure that all methods receive the same base forest so that no re-training of the forest would occur.

TABLE 10.1: Accuracy (rounded to the second decimal digit) and F_1 score (rounded to the fourth decimal digit) of the best model per method that can still fit into the memory of the PhyNode. The best model is marked in bold.

	COMP	DREP	IC	IE	L1	L1+LR	LMD	LR	RE	RF
Accuracy [%]	68.39	51.32	56.50	56.80	62.50	**71.04**	61.28	68.09	63.21	56.40
F_1	0.5757	0.5180	0.4266	0.4383	0.5234	**0.5758**	0.5300	0.5627	0.5163	0.4220

10.2 Discrete Classifiers and the FACT Telescope

As a second example, we discuss the usage of discrete classifiers in the context of an astrophysics experiment. Basic research in physics remains one of the main driving forces in many research fields outside of physics itself, including computer science and machine learning. From a computer science perspective, astrophysics is a particularly interesting subject because it involves large amounts of automatic data gathering through a radio telescope while, at the same time, it enables us to justify the research by one of the oldest questions to humankind: Where does the universe come from? While – this rather flamboyant – question may be asked by physics and philosophers, we try to avoid astrophysics itself as much as possible and leave it to the physicists. Our focus in this chapter is how discrete classifiers can be deployed to the telescope itself so that parts of the enormous processing pipeline (c.f. [Boc15]) can be executed on the device in real time. Discrete classifiers are a natural fit for radio astronomy since a telescope is continuously gathering data at a high data rate and a telescope is usually deployed to remote mountains for the best view, thereby limiting processing and communication resources as well as memory and physical space. The next section will give a brief overview of the FACT telescope, which inspired many ideas in this thesis. Section 10.2.2 shows how additive ensembles can be used to filter background noise from the telescope's data, and section 10.2.3 shows how the calibrated raw signals of the telescope can be analyzed in an end-to-end fashion by using BNNs.

Technical Note: The experiments in this section involve code from different projects written by different people at different points in time. Hence, they require some manual preparation of the data and computing environment. Unfortunately, there is no comprehensive and publicly available code repository that exactly reproduces these results. Parts of the code used for these experiments can be found under `https://bitbucket.org/sbuschjaeger/arch-forest/src`, `https://github.com/sbuschjaeger/Pysembles/`,
`https://github.com/sbuschjaeger/fastinference` and
`https://sfb876.tu-dortmund.de/FACT/`.

10.2.1 The FACT Telescope

Celestial objects several hundred million light-years away from the earth are recognized by observing the energy beams emitted by these sources. The energy beams have an effect on a detector medium. For example, particles interact with the earth's atmosphere, producing cascading air showers. These showers emit Cherenkov light, which, in turn, can be measured by telescopes such as the First G-APD Cherenkov Telescope (FACT). Figure 10.1 shows an air shower triggered by some cosmic ray beam, emitting Cherenkov light that is captured by the FACT telescope (left side).

The telescope can be viewed as a camera with 1440 pixels arranged in hexagonal form. Each pixel consists of a small light sensor that samples light pulses at a 2 GHz frequency. The right side of Figure 10.1 shows the resulting images taken by the telescope. Green indicates the telescope's surface, whereas blue indicates the amount of light hitting the sensors (the shower). Red indicates padding pixels that are used to form quadratic images from the telescope (discussed in more detail later). The camera continuously samples all the pixels into a ring buffer and a hardware trigger initiates a write-out to disk storage if some pixels exceed a specified threshold indicating that a shower is hitting the telescope. Upon trigger activation, a series of camera samples which amount to a time period of 150 nanoseconds called the region of interest (ROI), are written to disk. This time series of sensor voltages represents an event and corresponds to the light cone induced by the airshower. The FACT telescope records roughly 60 events per second, where each event amounts to up to 3 MB of raw data, resulting in a rate of about 180 MB/s.

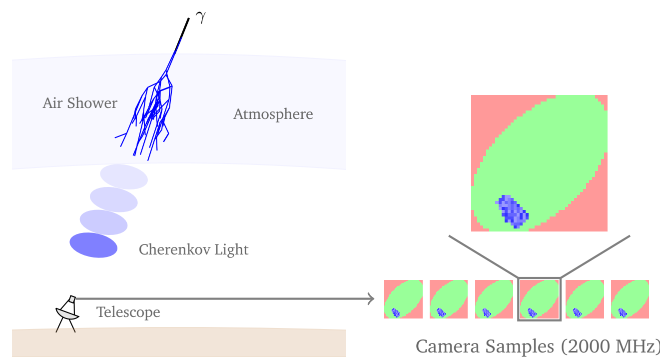


FIGURE 10.1: An air shower produced by a particle beam hitting the atmosphere (left) and the corresponding measurements (right side). The figure was taken and modified from [BBB+15].

A central problem in radio astronomy is the distinction between gamma rays which indicate a celestial object and background noise which is mostly produced by cosmic rays from hadrons that do not allow for conclusion on a particular source – the *gamma-hadron separation problem*. Data analysis has been established as an effective tool for analyzing modern high-energy particle experiments and solving the gamma-hadron separation problem [BBB+15]. The processing pipeline that is used to analyze the data performs multiple steps as depicted in Figure 10.2. First, the data are calibrated in order to account for environmental changes such as, e.g., the day-night cycle. Second, the resulting raw data are cleaned, i.e., values from broken sensors are corrected and artifacts are removed. Third, the pipeline extracts high-level features based on hand-crafted rules from domain experts. Finally, the ML model for the gamma-hadron separation is applied. For a detailed explanation of the over 80 individual steps involving this pipeline and its very fast execution on commodity hardware, we refer interested readers to [BBB+15].

10.2.2 Optimal Implementations of Random Forests

As depicted in Figure 10.2 the last processing step involves the execution of the learned machine learning model to classify the air showers into gamma or hadron particles. The standard classifier used to perform this separation is a random forest [BBB+15] and hence it is ideally suited for the implementations discussed previously. We start by giving a theoretical recommendation for FACT based on the theoretical

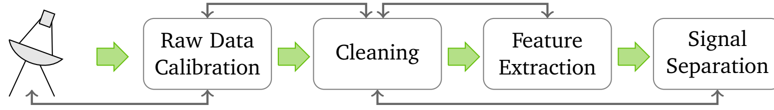


FIGURE 10.2: Data processing steps from raw data acquisition to signal separation. The workflow of using a simple trigger with data calibration / cleaning as well as feature extraction. The figure was taken and modified from [BBB+15].

von Neumann architecture and the theoretical FPGA developed in Section 8. Then, we conclude this analysis with an experimental evaluation of the best implementations for the fact data. Note that the fact data has already been used in multiple experiments in part 3 of this thesis. Here we extend these experiments to also include FPGAs. Since FPGAs do not have a memory hierarchy and to facilitate a fair comparison we will not perform any tree optimizations, but use the different implementations as-is on both ARM CPUs and FPGAs[1].

For filtering-out unwanted events, we train a random forest classifier with $M = 50$ decision trees on $N = 60\ 000$ training instances. To improve the quality of the trees, we prepare the training data to contain 50% of background noise and 50% gamma-ray events. Each tree contains an average of $1\ 349$ nodes and roughly 675 different paths from the root node to a leaf node. A 10-fold cross-validation shows, that the trained ensemble offers a prediction accuracy close to 80% for gamma-hadron separation.

Which hardware and which implementation would be the best fit for this ensemble? First, we will compare the implementations based on the theoretical analysis: Judging by the number of clock cycles, the FPGA clearly seems to be at an advantage, if we either implement if-else trees or one of the SIMD derivatives. Assuming a $s_w = 32$ bit float, we require $n \cdot r_{le} + n \cdot \lceil \frac{s_w}{t} \rceil + d \cdot s_w = 1349 \cdot (1.25 \lceil \frac{2 \cdot 32}{6} \rceil) + 1349 \cdot 32 + 1440 \cdot 32 = 107\ 797$ CLBs per tree for if-else trees. Therefore, we identify that if-else simply will not fit into smaller FPGAs. In the case of SIMD derivatives, we see that DNF trees will also not fit since they use more CLBs than if-else trees. The amount of resources required by a SIMD implementation depends on v. Using $v = 8$, we see that a tree needs $8 \cdot r_{addr} + 8 \cdot (1.25 \lceil \frac{2 \cdot 32}{6} \rceil) + 8 \cdot 1 + 1348 \cdot sizeof(\texttt{node_t}) + 2 \cdot 2 \cdot 32 + 2 \cdot 32 + 1440 \cdot 32 = 8 \cdot r_{addr} + 1348 \cdot sizeof(\texttt{node_t}) + 46\ 390$ logic blocks. Since we have $1\ 348$ nodes in a tree, we need at least 11 bits to index each node. In order to index $1\ 440$ features, we also need 11 bits. Thus, we can assume, that $sizeof(\texttt{node_t}) \approx 67$. Therefore, a SIMD implementation on FPGAs needs at least $1\ 348 \cdot 67 + 46\ 390 = 136\ 706$ CLBs, which is also unlikely to fit on smaller FPGAs.

We conclude, that for the particular problem at hand, FPGAs cannot be employed. In the case of a CPU, we find that if-else trees offer a fast and reliable clock delay. For the presented trees, we see that we need $4 \cdot 13 + 1 = 53$ clock cycles on average, leading to a total of $2\ 650$ clocks required to compute the complete forest. At peak performance, up to 300 measurements must be classified per second, and hence a processor with at least $795\ 000 Hz = 0.795$ Mhz. Thus, a small, embedded system with ≈ 1Mhz clock speed will do the job.

Last we compare three different implementations for FACT using a Xilinx Zedboard (http://zedboard.org/). The Zedboard contains an ARM Cortex-A9 with 666

[1] The main reason here is of historical nature. The experiments presented here are part of [BM18], whereas techniques for optimal memory layouting have been developed later in its successor [BCCM18]. While FastInference now supports all optimizations as well as different backends such as ARM or FPGAs I decided against re-doing these experiments to preserve the original analysis.

Mhz, 512 Mb DDR RAM, and 512 KB cache. Additionally, this board contains a Xilinx Artix-7 Z-7020 FPGA with 53 200 lookup tables, 106 400 flip-flops (FF) in total combined with 4.9 MB block ram, and 220 DSP units. The Zedboard contains four Advanced eXtensibel Interface (AXI) ports, each running at 142 Mhz with 32 bit word sizes leading to an aggregated bandwidth of up to 3.8 GB/s. We used `SDSoC` in version `2016.2` to run and compile the experiments. Power consumption was estimated using `Vivado` in version `2016.2`. All experiments were performed in the standalone mode of this board so that no operating system is involved during measurements. We activated the most aggressive optimization level `O3`. FPGA implementations are clocked with 100 Mhz giving an actual bandwidth of up to 1.6 GB/s. Table 10.2 depicts the average classification throughput in measurements per millisecond for a random forest and for a single decision tree. All tests were repeated 20 times.

TABLE 10.2: Throughput comparison for different implementations of random forests and decision trees on the FACT data. Larger is better. A dash "-" indicates that the corresponding model did not fit on the device.

	FPGA $\left[\frac{\text{elem}}{\text{ms}}\right]$	ARM $\left[\frac{\text{elem}}{\text{ms}}\right]$
If-Else (Tree)	$1480 \pm 2.7 \cdot 10^{-9}$	29000 ± 0.0027
If-Else (Forest)	-	$780 \pm 2.7 \cdot 10^{-9}$
Native (Tree)	1170 ± 0.00034	14500 ± 0.00054
Native (Forest)	-	$460 \pm 4.9 \cdot 10^{-9}$
DNF (Tree)	$1100 \pm 4.7 \cdot 10^{-10}$	$1900 \pm 4.9 \cdot 10^{-9}$
DNF (Forest)	-	$30 \pm 1.4 \cdot 10^{-13}$

One can observe that the if-else trees offer the highest throughput for single trees, as well as for random forests. Native tree implementations also do fairly well on the CPU, whereas DNF trees offer the smallest throughput. Looking at the FPGA, we first see that the random forest implementations did not fit onto the FPGA. Thus, the corresponding entries in Table 10.2 are missing. The decision tree implementations all fit on the FPGA with a throughput ranging from 1100 to 1480 elements per second, where if-else trees are the fastest and DNF trees are the slowest implementation.

Table 10.3 depicts the resource usage of all implementations. For the FPGA we depict the resource usage reported by the synthesis tool. For the CPU, we present the binary size, which is loaded by the first-stage bootloader directly after the power-on of the board. Please note, that since the board is used in standalone mode, the binary contains all necessary libraries, e.g. functions for time measurements and output over UART. One can see, that the native tree uses the least resources fitting up to 40 trees of a random forest onto the FPGA. The DNF trees as well as if-else trees also fit nicely on the FPGA but use more resources than native trees. With this implementation, one could roughly fit 5 trees on the FPGA. Looking at the CPU, one can observe that the binary sizes are around 1.3 MB to 2.4 MB. The DNF tree forest implementation is an exception in this regard with 6.8 MB.

Last, table 10.4 displays the power consumption of all models. It is well established that the complete zedboard uses around $4-6$ W in total [MWH13, MK15]. This also takes peripheral devices such as the audio controller or the VGA controller into account, which are not required during deployment for FACT. Therefore, we want to focus on the energy consumption of the ARM processor as well as the FPGA. Direct measuring of these quantities is difficult because these parts are integrated into the

TABLE 10.3: Resource comparison for different implementations of random forests and decision trees on the FACT data. Smaller is better. A dash "-" indicates that the corresponding model did not fit on the device.

	FPGA				ARM [MB]
	LUT	FF	BRAM	DSP	
If-Else (Tree)	10244	1317	0	0	1.3
If-Else (Forest)	-	-	-	-	2.4
Native (Tree)	31	64	5	0	1.3
Native (Forest)	-	-	-	-	1.8
DNF (Tree)	9609	3183	0	0	1.4
DNF (Forest)	-	-	-	-	6.8

board so we will rely on the estimations of the power consumption given by the synthesis tool. We report all estimates for the maximum power consumption during full load. The complete chip uses in total less than 2 W in all configurations, from which the ARM processor uses 1.53 W. The FPGA implementations greatly vary in power consumption ranging from 0.008W to 0.068W, but are all two to three magnitudes lower than what the ARM processor uses. Using the throughput measurements from table 10.2, we compute the amount of energy needed to process one measurement. One can observe, that on the ARM the if-else implementation offers the smallest power consumption because of the large throughput of this implementation. On the FPGA, however, the native implementation dominates, as this also uses the least resources. All in all, we see that the FPGA – despite smaller throughput – wins in terms of energy per element for all implementations.

TABLE 10.4: Power consumption for different random forests and decision implementations on the FACT data. Smaller is better. A dash "-" indicates that the corresponding model did not fit on the device.

	Power [W]		Power per Element $\left[\frac{\text{nJ}}{\text{elem}}\right]$	
	FPGA	ARM	FPGA	ARM
If-Else (Tree)	0.068	1.53	45.95	52.76
If-Else (Forest)	-	1.53	-	1961.54
Native (Tree)	0.008	1.53	6.84	105.51
Native (Forest)	-	1.53	-	3326.08
DNF (Tree)	0.023	1.53	20.9	805.26
DNF (Forest)	-	1.53	-	51000

10.2.3 On-Site Gamma-Hadron Separation with BNNs

The FACT pipeline heavily relies on hand-crafted features. It uses a basic trigger, i.e., it begins the recording of an event when sufficient energy hits the detector. Then the data is communicated to a larger processing system (server or desktop) which executes the remaining processing pipeline. Unfortunately, this approach does not scale well with higher data rates and larger telescopes. For example, the Cherenkov Telescope Ring (CTR) aims to connect multiple telescopes around the globe thereby

increasing the data-rate orders of magnitudes [RER+19]. Moreover, we note that even a Mac mini (as used in [BBB+15]) requires up to 85W as well as space and cooling.

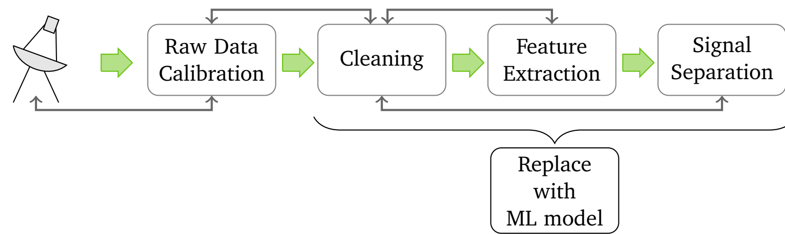


FIGURE 10.3: Data processing steps from raw data acquisition to signal separation. The workflow of using a simple trigger with data calibration / cleaning as well as feature extraction that is now to be replaced by a model learned from the raw observations. The figure was taken and modified from [BBB+15].

A natural question is if we can bring this data analysis closer to the telescope and mitigate the costly feature engineering at the same time as depicted in Figure 10.3. To do so, we will use binarized CNNs that directly operate on the raw data of the telescope's camera and output the corresponding class (gamma or hadron) of the observed shower. We specifically investigate if this model can be executed on commodity hardware available at the telescope to handle its data stream in real time and also study the use of FPGAs for inference. We assume the following minimal pre-processing:

- **Sensor calibration:** The detectors' sensors behave differently in different environmental situations, e.g., the temperature has an effect on the sensor which should be corrected. This calibration involves the correcting of sensor values by multiplicative constants and additive biases and can therefore easily be performed either by an FPGA or a digital signal processor.

- **Extracting photon counts:** The FACT telescope produces 1440 time series each with a length of 150 nanoseconds. We remove noise from the time series and focus on a time window of 50 ns which contains most of the relevant Cherenkov photons. Calibration measurements for the sensors depict a typical voltage curve when a single photon hits the sensor. This baseline measurement is subtracted as often as possible from the actual measurement until there is no signal left [MAA+17]. The number of subtractions can be considered the number of photons that arrived during the time series. The resulting image then shows the photon counts for each sensor in each pixel.

- **Image mapping:** The FACT sensors are arranged in a hexagonal form. In hexagonal grids, each pixel has up to six neighbors instead of four as in regular Euclidean grids. Recall, that CNNs apply rectangular convolution filters to extract and generate higher-level features. Although the neighborhood of pixels in rectangular and in hexagonal grids are slightly different, in a series of pre-experiments as well as student theses, no performance difference could be found between using rectangular and hexagonal filters for FACT [Rö17, May18]. We, therefore, choose to transform the data into 45×45 images in which the hexagonal grid is slightly rotated into the middle of the image. This allows us to use regular CNNs architectures and filters together with common frameworks. Figure 10.1 (right side) depicts the sensor mapping. Here, the

red color denotes unused pixels which are always '0', whereas green (and blue) pixels are mapped to the corresponding sensors.

Experiments are designed to answer three key questions: First, can (binarized) CNNs replace hand-crafted features with reasonable accuracy on simulated data? Second, will these models generalize well enough to be used for real data? Third, can we execute these models on-site and in real-time using commodity hardware and/or FPGAs? We now tackle each question individually. For the first two questions, we want to emphasize, that random forests are still state-of-the-art for the FACT data due to their resilience against overfitting and thereby overcoming the gap between simulation (see below) and real-world data.

Models

For our study, we investigate one small and one large VGG-style neural network architecture [SZ15] each in a regular floating-point version and as a binarized version. These networks are composed of blocks, where each block consists of two 3×3 convolutional layers each followed by a batch normalization layer and a ReLU activation as well as a single max-pooling layer of stride 2 (see also Figure 2.9). The small network is composed of two of these blocks, while the large network is composed of four. The number of channels in the convolution layers starts with 128 in the first block and then increases by 128 with every following block. Finally, we compute a linear layer of size 128 or 512, respectively. We apply batch normalization before we compute the class probabilities using a final softmax layer. We have also experimented with residual neural networks, as suggested by Zhang et al. for training BNNs [ZST+19], but found that they did not outperform our purely convolutional models.

The networks are trained using the AMSGrad optimizer [RKK18] with a batch size of 128 examples minimizing the cross entropy loss. We train our models for 100 epochs and use an initial learning rate of 0.001 which we reduce by a factor of 0.1 every 25 epochs. The neural networks learn from raw photon counts (denoted by PhC).

Since the state of the art is the random forest (RF) learner, it is applied for comparison. It learns from the hand-crafted high-level features (denoted by DL2) as well as from PhC data. Our RFs consist of 128 decision tree estimators of unlimited depth. Its decision trees are built using bootstrap samples of the training data and each of its splits is selected by maximizing the Gini score on a random subset of features of size $\sqrt{d}$, where $d = 22$ (DL2) or $d = 45^2$ (PhC). For deep learning, we use PyTorch [PGM+19] and for fitting RF we use scikit-learn [PVG+11].

Experiments on simulation data

When applying machine learning in astrophysics, it is difficult to obtain labeled data since particles from outer space can come from any source. A common approach to solve this problem is to combine Monte Carlo simulations with careful training of the classifier.

Astrophysics has a profound understanding of particle interactions in the atmosphere: Given the energy and direction of some parent particle (gamma, proton, etc.), its interaction can be described by a probabilistic model which gives a probability for particle collisions, possibly resulting in secondary particles, which again may

interact with each other. This results in a cascade of levels of interactions that form the air shower, which can be simulated by particle simulation software like CORSIKA [HKC+98]. The output is a simulated air shower, which needs to be run through a simulation of the telescope and camera device to produce realistic raw data mimicking a shower that would have been recorded using the telescope. We can simulate interesting particles (e.g. gamma) and uninteresting particles (e.g. proton) and label the resulting raw data accordingly.

Using CORSIKA we simulate 200 000 training data and 100 000 test data, each set with a perfectly balanced class frequency of hadron and gamma events. The simulation comes in two variants, with or without quality cuts. Physicists have identified regions, where the simulation is inaccurate and does not resemble real events sufficiently well. The so-called quality cuts eliminate such unrealistic simulated events.

We train our models on both variants of the simulation data and summarize the findings in Table 10.5. Neural networks beat the RF baseline trained on high-level features by a large margin. Possibly due to the neatness of the simulation data, a RF trained on photon counts also beats the baseline. As the results show, the dataset with quality cuts poses an easier classification problem where higher accuracies are achieved. Float models show overfitting after 100 epochs. Hence, we also tried training with early stopping after 10 epochs, which results in a small positive effect for floating points trained with quality cuts. BNNs, however, need more training epochs to achieve good test accuracy. Overall, our BNNs perform slightly worse than their floating-point counterparts. However, for the large models, this difference is small. In the next section, we investigate if our findings carry over from simulation data to data recorded by a real telescope.

TABLE 10.5: Accuracy on simulation data. We distinguish models trained on simulations with and without quality cuts (QC). For the neural networks, we also report accuracies for models trained with early stopping after 10 epochs.

Model	Data	Accuracy, no QC		Accuracy, QC	
		epochs:100	epochs:10	epochs:100	epochs:10
RF	DL2	0.70959		0.78483	
RF	PhC	0.74711		0.78839	
CNN(small)	PhC	0.90825	0.88867	0.93441	0.93846
BNN(small)	PhC	0.90861	0.88644	0.90440	0.88866
CNN(large)	PhC	**0.91094**	0.90251	0.93735	**0.94228**
BNN(large)	PhC	0.90011	0.89925	0.93112	0.91369

Experiments on real-world Crab Nebula observations

Now we evaluate our trained models on real-world data collected by the FACT telescope at the Observatorio del Roque de los Muchachos (La Palma, Canary Islands, Spain). The telescope has been directed once towards a known gamma source, the Crab Nebula which emits large amounts of gamma rays. On these recorded data, we run the full source detection pipeline of the FACT experiment and investigate the influence of the gamma-hadron separation models on the overall quality of the source detection.

The evaluation proceeds in the following steps: We take the publicly available (`https://fact-project.org/data/`) Crab Nebula observation data [ABB+13, B+14], which consists of 17.7 hours or 3 972 043 recorded events. Our gamma-hadron classification models are applied to classify which events are gamma rays. Then, an established model estimates the direction of the incoming gamma rays [Nö17]. From the direction, we compute the angle between the trajectory of any incoming ray and the known direction of the Crab Nebula. The number of gamma rays can be regarded as a distribution that depends on the angle. High counts are to be expected for small angles. The distribution with respect to the direction of the Crab Nebula is called an *on-distribution* [FSL+94]. Contrasting the distribution of counts with distributions for five different positions with no known gamma sources yields the *off-distributions*. A uniform distribution of counts over angles is expected, where the majority of counted rays can be attributed to misclassified hadronic rays. We state the null hypothesis that on-distribution and off-distribution follow the same distribution, or, intuitively, that there is no gamma source at the direction of the Crab Nebula. The margin by which a significance test rejects this null hypothesis gives us a significance of detection $S_{li\&ma}$, reported by the number of standard deviations σ [LM83]. This $S_{li\&ma}$ is the performance metric for gamma-hadron classifiers, where larger numbers are better.

The trained classifiers output probabilities that an event is a gamma ray, and we can control the classification behavior by varying the threshold for actually predicting gamma. A large threshold yields fewer events and also fewer misclassified events because the classifier is more certain. If we set the threshold too large, we get too few total events which results in a small statistical significance. In contrast, if we decrease the threshold, we obtain more events, but also more misclassifications. If we set the value too small, we count too many noise events, and the difference between on- and off-distribution shrinks, which also yields a low significance of detection.

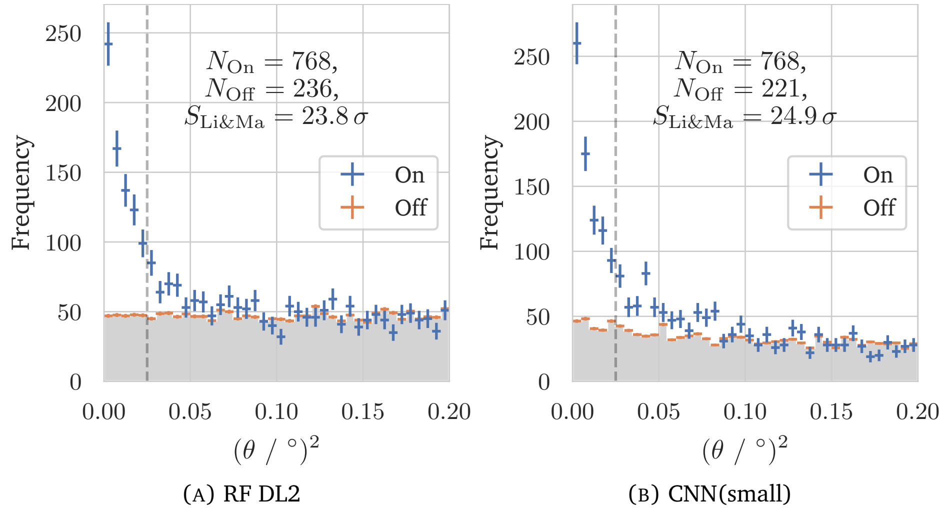
To ensure that the output probabilities of the models are meaningful estimates of the classifiers' confidence, we apply isotonic probability calibration [ZE01] using the simulated test examples as calibration data. If not explicitly mentioned otherwise, the RFs use a threshold of 0.85, CNNs use a threshold of 0.6, and BNNs a threshold of 0.75.

TABLE 10.6: Significance of detection.

Model	Data	$\mathrm{S_{li\&ma}}$, no QC		$\mathrm{S_{li\&ma}}$, QC	
		epoch: 100	epoch: best loss	epoch: 100	epochs:best loss
RF	DL2	22.86σ		23.82σ	
RF	PhC	2.09σ		3.35σ	
CNN(small)	PhC	24.09σ	25.83σ	24.12σ	24.89σ
BNN(small)	PhC	19.55σ	$\mathbf{25.87}\sigma$	22.96σ	21.67σ
CNN(large)	PhC	23.68σ	24.64σ	24.20σ	23.17σ
BNN(large)	PhC	22.70σ	22.92σ	22.35σ	22.26σ

In table 10.6 we summarize the results for all models. Using the established RF classifier on high-level features, we obtain a significance of detection of 23.8σ. Small and large float CNNs outperform the baseline with a significance of 24.12σ and 24.20σ, respectively. BNNs achieve a slightly smaller significance of 22.96σ and 22.35σ. We hypothesize that this is again due to overfitting. To investigate this, during the training of our models, we compute the validation loss on the simulated test data

after each epoch and use the best epoch for classification. When we inspect the $S_{li\&ma}$ scores for the epoch with the best loss, we see that these more carefully selected models indeed perform better: Both the small BNN and CNN now achieve significance over 25.8σ. For large models, however, we do not see the same benefits, further analysis is needed to better understand the connection between loss on simulation data and detection significance on real data. Last, we see that the random forests trained on photon counts are not useful at all for real-world data.



(A) RF DL2 (B) CNN(small)

FIGURE 10.4: Histogram of the frequencies as a function of the squared angular distance between the trajectory of any incoming ray and a position in the sky. On-events show the frequency with respect to the position of the Crab Nebula, while Off-events are w.r.t. positions with no known sources. The significance of the detection test only considers angles smaller than 0.025 (left of the dashed vertical line).

Visual inspection of the on- and off-distribution in Figure 10.4 reveals the different classification behavior of the random forest baseline and our float CNN: We see that the random forest has a uniform off-distribution, while the CNN has a decaying distribution with smaller counts for larger angles. This suggests that our classifier is biased to predict gamma at the camera positions used for the off-distributions. Indeed, the gamma rays in the simulation data are not generated uniformly, which can explain this bias.

Proof-of-concept with FPGAs

We want to measure the impact of using BNNs compared to floating-point nets running on different hardware close to the telescope. Since FACT produces data at a rate of roughly 60 events per second, on average, we cannot spend more than 16ms to classify a single event.

As explained above, we use FastInference to generate c-code for each model and compile this for the target architecture, either by High-Level Synthesis or by a regular compiler for CPUs. For CPUs, we enabled the most aggressive optimizations `-Ofast -march=native -mtune=native` using `gcc` version 8.3. We compare our results with the deep learning inference engine ONNX Runtime, which is optimized toward real-time model inference. Experiments are run on commodity hardware, namely an Intel i7-6700 CPU with 16 GB RAM. For consistent runtime measurements, we randomly

sampled 1000 events and measure the total runtime to process these events, then compute the average runtime per event in this batch. This process is repeated 20 times, and we report the average runtime and standard deviation per single event across all batches.

For the FPGA, we use a Xilinx Virtex UltraScale VCU110 Evaluation Board with 805 680 lookup-tables (LUTs) and 132.9 MB Block-Ram (BRAM). The synthesis was performed with `Xilinx Vivado HLS` 2018.3. We used the generated c-code as a basis and performed a minimal design space exploration by either pipe-lining or unrolling loops in the design to maximize the performance without exceeding the available LUTs and BRAM. The design for the small BNN is clocked at 25 MHz[2] whereas the design for the large BNN is clocked at 100 Mhz.

Note, that we allocate independent input/output buffers for each layer. For classifying a single event this is wasteful, because we only use one buffer pair at the same time, while all other pairs are not used. However, we expect our design to run continuously so that a stream of events is available. This enables efficient pipe-lining of the entire design: For each event, we process one layer so that the classification of the first event is delayed by the number of layers L in the entire network. Processing a single layer is much quicker than processing all layers, which means that, despite the initial delay, we can classify events at a faster overall rate.

Table 10.7 shows the latency of the different neural network configurations using different inference engines. We see that ONNX Runtime offers the fastest classification rate for small and large float networks, whereas our code generator outperforms Onnx Runtime in the case of BNNs. It was not possible to synthesize a working FPGA design for float networks, because they utilize too much BRAM. The FPGA offers the fastest (small BNN, pipelined) and slowest execution time (large binary, pipeline) for BNNs depending on the specific configuration. In summary, for floating-point networks, Onnx Runtime is the fastest method, whereas for smaller BNNs the FPGA is the fastest and for larger BNNs, our generated code seems to be the best method. The reasons for this are three-fold: Onnx Runtime is highly optimized for floating-point operations utilizing vectorization instructions to their fullest. In contrast, the code generator relies on the compiler to vectorize loops. Looking at BNNs, the situation reverses. Where our implementation exploits the specific structure of BNNs to gain performance, Onnx Runtime does not support this. Finally, small models have the lowest latency on FPGAs, because large parts of the network can be unrolled so that they fit entirely on the FPGA. In contrast, larger models which do not fit well on the FPGA suffer tremendously. If most loops cannot be unrolled, the result is a very slow design. For the application at the Cherenkov telescope, large CNNs are not an option with either inference system, since none of them meet the required 16 ms latency. It is interesting to note that small float nets can be executed slightly quicker than BNNs on commodity hardware. We attribute this fact to the floating-point vectorization instructions available on current Intel CPUs. If these are not available, BNNs are a very attractive alternative, especially for large networks. All in all, the generated code satisfies the most scenarios enabling small CNNs, small BNNs, and large BNNs making it the best overall choice.

10.3 Discrete Classifiers and Approximate Memory

As a third example, we study the behavior of discrete classifiers on systems that use approximate memory. Approximate memory is a relatively new research direction in

[2] We found that using fewer clocks improved latency because loops can be unrolled.

TABLE 10.7: Latency of different neural net configurations using different inference engines. The best inference engine for float and binary is marked in bold. Smaller is better. A dash "-" indicates that a model did not fit on the FPGA.

System	Type	Runtime [ms/event]	
		float	binary
ONNX Runtime	large	$\mathbf{21.083 \pm 0.078}$	26.642 ± 0.100
	small	$\mathbf{0.957 \pm 0.020}$	1.861 ± 0.037
Generated Code	large	78.583 ± 1.704	$\mathbf{11.250 \pm 0.077}$
	small	2.757 ± 0.026	1.574 ± 0.014
FPGA	large	-	561.588 ± 0.000
	small	-	4.221 ± 0.000
FPGA pipelined	large	-	72.657 ± 0.000
	small	-	$\mathbf{0.662 \pm 0.000}$

the design of new memory architectures that reduces the memory supply voltage and changes the latency parameters (e.g. for writing to the memory) with the goal of lower power consumption and faster access. Hence, they are ideally suited for small devices with limited power resources.

If these novel designs are pushed to the limit, high bit error rates (BERs) can occur. For modern memory technologies, such as volatile memories (SRAM [RWA+16, YM17], DRAM [KOY+19]) and emerging non-volatile memories (e.g. STT-RAM [HPK+19, SWH+17], RRAM [HBK+19]) the BER increases steeply when reducing the voltage and tightening the timing.

To cope with the increase in BER two approaches are possible: First, a correction mechanism can be adopted into the system's design that detects and potentially corrects bit errors once they occur. Second, the middleware and the user code running on the system is designed with errors in mind, so that it still operates well with errors. While both approaches are orthogonal and should be used in combination, for the deployment of models to small devices the latter must be studied more carefully. Here, the ML system must, by design, take the structure of the underlying hardware into account and therefore the trained model must already be error tolerant.

Due to the excessive amount of resources required by neural networks and deep learning, the design of error-tolerant NN architectures was in the focus early on, and we will follow this route in this case study. The survey in [TG17] provides a comprehensive overview of the recent and further back work about fault and error-tolerant NNs, from which we summarize some representative studies. For example, the study in [EM97] proposes a penalty term that aims at distributing the computation to neurons optimally to achieve error tolerance. Another study [CM99] distributes the absolute values of weights evenly to neurons, while the work in [Sim01] aims at low weight importance.

Binarized neural networks play an important role in this research direction as they present a low-resource, error-tolerant alternative to regular NN architectures. Recall that BNNs use one bit to store individual weights and activations and hence, changing the status of one bit leads to a flip of a single weight or activation in the network. While this is still a severe change in the network's configuration it is much

more manageable than changes in a regular NN. In regular floating-point NN, a single bitflip can lead to changes in the sign, the mantissa, or the exponent of the float leading to unbounded changes in the weight or activation.

Technical Note: The experiments in this section involve code from different projects written by different people at different points in time. They require some manual preparation of the data and computing environment. Hence, there is no comprehensive and publicly available code repository that exactly reproduces these results. Parts of the code used for these experiments can be found under https://github.com/sbuschjaeger/Pysembles/.

10.3.1 Margin-Maximization for Error Tolerant BNNs

In this study, we assume that bit error rates (BERs) are transient and symmetric, i.e., the probability for 0 to flip to 1 is the same as the probability for 1 to flip to 0. This assumption characterizes the probability of bit flips every time when a bit is read from the approximate memory. This matches the assumptions in recent studies about approximate memories (SRAM [SLC+17, YBM+18, HLS20], DRAM [KOY+19]), and non-volatile memories (RRAM [HBK+19], MRAM or STT-RAM [HPK+19]).

Currently, the only known method to enhance the bit error tolerance of BNNs is training with bit flip injections according to the error model. Training with bit flip injections trains the BNN via SGD as usual but introduces random bit flips during the forward pass while the backward pass is untouched. While training with bit flip injection is straightforward, it has disadvantages. First, recent studies have reported that injecting bit flips during training can significantly degrade accuracy. The higher the BER during training, the more significant the accuracy degradation [HBK+19, KOY+19, BCC+21b]. Another disadvantage is the additional overhead [MVS+19]. During the training with bit flip injection, for every bit of the error-prone data, a decision has to be made whether to inject a bit flip, which adds numerous additional steps in the BNN training.

We propose a novel approach to overcome this problem that considers the classification margin. Let $c = \arg\max_{i=1,\ldots,C} \hat{y}_i$ be the largest output of the BNN and let $c' = \arg\max_{j=1,\ldots,C, i \neq j} \hat{y}_j$ be the second-largest output of the BNN. Further, let

$$m = \hat{y}_c - \hat{y}_{c'} > 0 \tag{10.1}$$

be the margin of the BNN, then the output layer of the BNN tolerates

$$\max(0, \left\lfloor \frac{m}{2} \right\rfloor - 1) \tag{10.2}$$

bit flips [BCC+21a]. Clearly, for a high error tolerance, the margin must be maximized. However, optimizing with respect to m without considering the other entries $\hat{y}_c$ of $\hat{y}$ may not exhaust the full potential of the margin between $\hat{y}_{c'}$ and the output of the other classes $\hat{y}_c$. The larger the margin between $\hat{y}_{c'}$ and $\hat{y}_c$ of other classes c, i.e. $m_c = \hat{y}_{c'} - \hat{y}_c$, the more bit errors can be tolerated in the neuron that calculates $\hat{y}_c$ without a change of the prediction. To put it concisely, for a bit error tolerant output layer, $\hat{y}_{c'}$ needs to be as large as possible, while the other $\hat{y}_c$ need to be as small as possible. To achieve this, we build upon the hinge loss for maximum margin classification. The hinge loss (c.f. [RVC+04]) for multi-class classification is defined as

$$\ell_{MHL}(\hat{y}, y) = \max\{0, (b - y \cdot \hat{y})\}. \tag{10.3}$$

where we introduced a parameter $b \in \mathbb{R}$ that serves as a slack for a high penalty for wrong predictions and a small penalty for correct predictions.

10.3.2 Experiments

In our experimental analysis, we compare the error tolerance of BNNs trained via bit-flip injection with the max-margin training. Last, we also study the error tolerance when both approaches are combined. We evaluate three types of BNNs: Fully connected BNNs (MLP) and small convolutional BNNs (CNNs) for the FashionMNIST dataset [XRV17], and a larger CNNs for the CIFAR10 dataset [Kri09]. The specific BNN architectures are presented in Table 10.8. The BNNs use convolutional (C) layers with size 3×3, fully connected (FC) layers, maxpool (MP) with size 2×2, and batch normalization (BN) layers followed by activation.

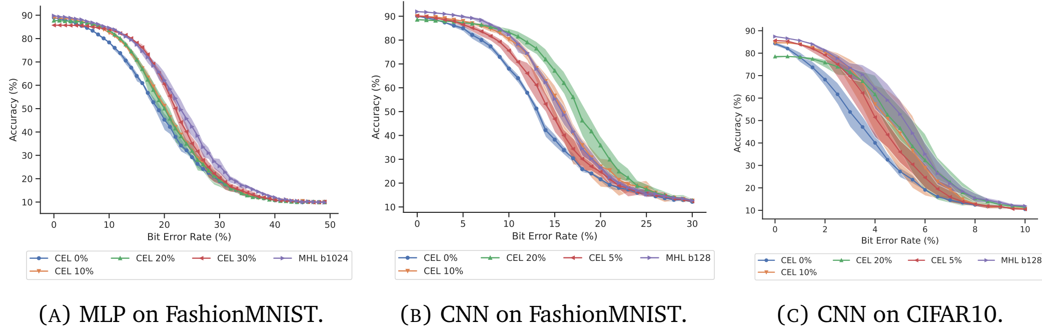
Parameter	Range
Fashion FCNN	In → FC 2048 → FC 2048 → FC 10
Fashion CNN	In → C64 → MP2 → BN → C64 → MP2 → BN → FC2048 → BN → FC10
CIFAR10 CNN	In → C128 → BN → C128 → MP2 → BN → C256 → BN → C256 → MP2 → BN → C512 → BN → C512 → MP2 → BN → FC1024 → BN → FC10

TABLE 10.8: BNN architectures used for the bit-error tolerance experiments.

For training, we run the Adam optimizer for 200 epochs for FashionMNIST and 500 epochs for CIFAR10, with either cross-entropy loss (CEL) or modified hinge loss (MHL). We use a batch size of 256 and an initial learning rate of 10^{-3}. To stabilize training we exponentially decrease the learning rate every 25 epochs by 50 percent. To cover a wide spectrum of bit errors, for testing we use bit error rates (BERs) from 0% (no bit errors) up to 35%, with increments of either 1% for Fashion and 0.5% for CIFAR10. For training with bit flips we use different BERs, from 1% up to 30% BER, such that accuracy degradation is below 10% from the original accuracy. Depending on the approximate memory and its properties, accepting BERs of this extent can improve the approximate memory, such as in energy consumption, timing parameters, production cost, etc.

For each data set, five BNNs were trained using MHL without any bit-flip injections and CEL with different BERs for bit-flip injections. Moreover, for all BNNs trained with MHL, we employed a parameter search for b, testing powers of two, up to two times the maximum value the neurons in the output layer can compute (maximum output value of a neuron in the output layer is the number of neurons in the layer before the output layer). Among these configurations of b, the best one was chosen.

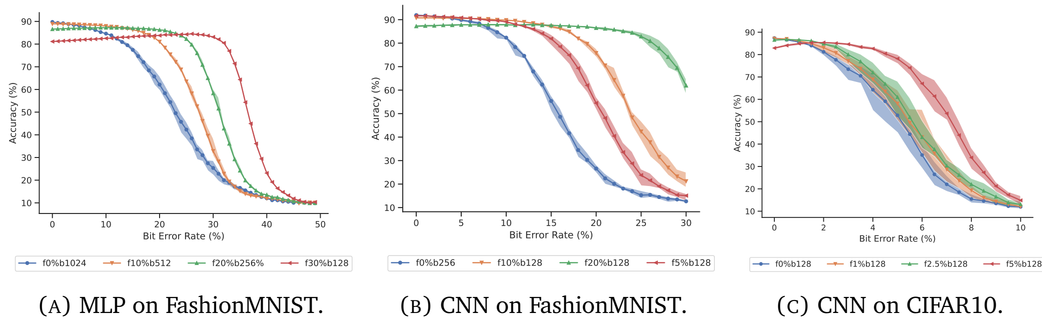
Figure 10.5 presents the results of this experiment. We observe that BNNs trained with MHL without bit flip injections have better accuracy over BER than the BNNs trained with CEL under bit flip injections. The BNNs trained with CEL suffer from a significant accuracy drop for lower BERs, when the BER during training is high, e.g., CEL 20% and/or CEL 30% at low BER. The BNNs trained with MHL, however, do not suffer from this accuracy drop. Although the CNNs trained with CEL 20% and bit-flip injections have better accuracy for FashionMNIST when the error rate is higher than 10%, the overall accuracy drops by a significant amount, which may be unacceptable.



(A) MLP on FashionMNIST. (B) CNN on FashionMNIST. (C) CNN on CIFAR10.

FIGURE 10.5: Test accuracy over bit error rate for BNNs trained with CEL under a given bit flip injection rate (specified in the legend, 0%, 5%, 10%, etc.) and BNNs trained with MHL without bit flip injections for a specified b in Eq. (10.3).

Training with the max-margin loss seems favorable compared to bit-flip injections. However, to further improve the bit error tolerance, we can also combine bit-flip injection with the max-margin loss. To do so we repeat the same experimental protocol as described above and detect reasonable ranges for b through a set of pre-experiments, but now minimize the MHL loss while applying bit-flip injections during training.



(A) MLP on FashionMNIST. (B) CNN on FashionMNIST. (C) CNN on CIFAR10.

FIGURE 10.6: Test accuracy over bit error rate for BNNs trained with MHL and bit flip injections (denoted as flip 0%, 1%, etc). The number after the b is the value to which the parameter b in the MHL is set during training (see Eq. (10.3))

Figure 10.6 presents the experimental results of different BNNs with respect to the accuracy over BER. In all experiments, we observe that the accuracy over the BER of the BNNs trained under MHL and bit-flip injections is significantly higher than that of the baseline trained by only MHL. For example, for FashionMNIST in Figure 10.6a and 10.6b, the BER at which the accuracy degrades significantly is extended from 5% (baseline, green curve) to 20% and 15% respectively, with a small trade-off in the accuracy at 0% BER. If more accuracy at low error bit rates is traded, the BER at which accuracy degrades steeply can be shifted even further. For CIFAR10 in Figure 10.6c, this breaking point can also be increased. However, more accuracy has to be traded compared to the previous cases.

11 | Conclusion and Future Work

Effective machine learning requires high-quality data as well as time and energy to explore data pre-processing, classifier training, and the evaluation of the entire ML pipeline. This leads to a tight feedback loop in which practitioners re-design the ML pipeline and evaluate it in rapid succession to find the best combination of pre-processing, classifier as well as training configuration. While practitioners often spend the majority of their time in this design stage, the overall energy consumption is often dominated by the continuous application of the ML system once it is deployed.

Discrete classifiers are models that can be applied without the need for costly floating-point operations and hence are ideally suited for energy-efficient deployment of ML systems even on the tiniest of devices. However, the performance of these classifiers is often weaker than models that utilize floating-point operations. To strengthen the predictive performance, an ensemble of discrete classifiers can be used that combines multiple discrete classifiers into a single model. While the deployment of multiple models to reduce resource consumption seems counterintuitive at first, it offers more flexibility in the design of the ML system while mitigating the costly floating-point operations.

In this thesis, we studied discrete classifier ensembles from a theoretical, algorithmic, and practical point of view. In the theoretical study of ensembles, we found that the bias-(co-)variance decomposition plays a major role in the design of new ensembles. While originally presented for the mean squared error, we presented a generalization based on a second-order Taylor approximation that recovers the original decomposition while generalizing it to other loss functions. So far, it has been difficult in literature to decompose classification losses such as the zero-one loss into bias and variance while recovering the original decomposition for the MSE. The key insight of our generalized bias-variance decomposition is that – depending on the loss function – the remainder of this decomposition is non-zero, and many classification losses simply do not have a non-zero remainder. Second, we studied the double-descent phenomenon that occurs when very large models are trained on comparably few data points: First, the model starts to overfit in an u-shaped curve, but then, once a certain model complexity is reached, it starts to improve its performance again. For the first time, we studied this phenomenon in the context of random forests and found that it would not occur in these models. Moreover, we found evidence that there is a bathtub-like relationship between the bias and the diversity of the forest in which the exact bias-diversity trade-off does not matter, but a large range of hyperparameters offer similar performance. The key insight here is that the PAC-learning theory and the Rademacher complexity do not seem to predict the performance of RFs well, but the algorithm training the RF seems to play a more important role than it is given credit for by this framework.

On an algorithmic level, we presented the novel Generalized Negative Correlation

Learning (GNCL) algorithm that is derived from our novel Generalized Bias-Variance decomposition and showed how many existing works in the literature can be expressed in the GNCL framework. We then continued to study the deployment of tree ensembles to small devices more carefully. We discussed ensemble pruning and leaf-refinement in detail and compared both approaches. The key insight here is that both approaches should be unified into a single objective that refines and prunes the ensemble at the same time. To do so, we introduced L_1 regularization into the objective and minimized it via proximal gradient descent resulting in a statistically significant better performance of the ensemble. Taking this idea one step further, we expanded it to the training of tree ensembles on the devices themselves. The resulting online algorithm trains classifiers on small batches of incoming data and then performs an online pruning step. The challenge for online learning is that the L_1 regularization acts like a soft constraint that does not guarantee a maximum number of classifiers at all times, possibly challenging the memory of a small device. Hence, we introduced a L_0 regularization that guarantees a maximum number of classifiers that strictly bounds the memory consumption of the model during training but also makes the optimization much more challenging. Our novel Shrub Ensembles algorithm utilizes proximal gradient descent to effectively minimize the overall ensemble error while adhering to the L_0 constraints. The key insight here is, that training an ensemble on sliding windows outperforms other online learning algorithms under resource constraints while offering similar performance in the general case.

On an implementation level, we studied both, the best implementation for tree ensembles and the surrounding software ecosystem for model deployment. We discussed the different types of tree implementations and showed, for the first time, how these implementations can benefit from a better memory layout and more effective caching. The key insight for this work is that DTs have a probabilistic runtime that can be estimated during training. During deployment, this estimated runtime is then exploited to compute the optimal memory layout for caching through our tree-framing algorithms. We continued to study these implementations more theoretically by defining a theoretical computer architecture and compared the different implementations in terms of their expected number of clock cycles and resource consumption. The key insight here is that a well-defined theoretical architecture can guide the implementation selection early in the design process, e.g., when it is likely that too many resources are consumed by a certain implementation.

In the last part of the thesis, we presented practical applications of discrete classifiers. We surveyed the existing landscape of model deployment frameworks and found that most frameworks for deployment focus on deep learning and neglect classic methods. Moreover, while these frameworks offer excellent performance, they treat models as computation graphs while ignoring their inherent structure and the training process involved in computing them. Hence, it is impossible to apply optimizations such as ensemble pruning or tree framing in these frameworks. We presented FastInferece as a model compiler that generates model- and architecture-specific inference code for classic models as well as (binarized) feed-forward nets that allow for optimizations of both the model and its implementation. The key insight for FastInference is that the optimization and implementation of a model are connected via the model itself and are mostly independent of the specific target system. Hence, optimizations of the model result in a new model that can be chained together into a list of optimizations whereas the implementation can perform implementation-specific optimizations right before deployment. Finally, the realization of the code for the specific target system can then be computed by instantiating the correct templates from a template library. To the best of our knowledge, FastInference is the only tool

that combines optimizations of models and implementations into a single framework.

The research in this thesis raises some important questions. Looking at the theoretical understanding of tree ensembles, neither PAC-learning theory nor the double-descent phenomenon can really explain the overwhelming performance of random forests in practice. Although a better estimation of the Rademacher complexity of decision trees might redeem the usefulness of the PAC-learning theory in this context, our experiments indicate that the complexity of models is not the correct tool to understand the performance of tree ensembles, but the training algorithm seems to play a more important role than previously thought. Taking this argument one step further, we may hypothesize that the top-down greedy learning of DTs seems to play a major role in this setting. And indeed, other researchers report that part of RF's success is due to the fact that DTs can learn noise from the data [WOBM17]. This, in turn, is a property derived from the training algorithm of DTs, as argued in this thesis. Tightly related here is the fact that tree ensembles that are *not* trained via greedy top-down algorithms but, e.g., by SGD, often have mixed performance. Although more careful experiments are required here, the personal experience during this thesis implies that the main performance improvement when training DT ensembles with SGD is that linear functions are used for splitting (i.e., oblique splits). Again, taking this one step further, it is clear that neural networks empirically do not benefit as much as decision trees from ensembling, although improvements are possible. The main difference between the training of NNs and DTs is the training algorithm, and hence the question becomes: Are tree ensembles more effective because trees are trained greedily and – in turn – are ensembles of NNs less effective because deep nets are trained via SGD-like algorithms?

Looking at the training of ensembles on small devices, we have seen that training models on sliding windows of the data can offer better performance in resource-constraint environments. For the training of DTs, we only require a minimal amount of floating-point operations by design, but for BNNs, we still have to compute floating-point gradients for SGD. Hence, we may ask if there is a training algorithm for BNNs that does not require floating-point operations, e.g., by fitting a BNN ensemble in an online environment using our Shrub Ensembles algorithm. The central question here is if there exists a learning algorithm for BNNs that does not require gradients and that is adaptive to the data, similar to top-down DT algorithms?

Looking at the deployment of ML models, we find that the verification and certification of the entire pipeline is an import issue. Model verification has already attracted significant attention in recent years. This thesis additionally raised the issue of implementing ML models and – by extension – the verification of these implementations as well. While there is a large body of literature in the context of software verification, it has not yet been adapted to machine learning. More specifically, the entire ML pipeline starting from the data-gathering process, over pre-processing to serving predictions, must be verified both for the modeling part as well as its implementation. At the same time, it is clear that machine learning has a profound impact on the design of new computing architectures that already impact the current landscape of hardware manufacturing. Similarly, novel hardware also impacts the design and research in machine learning as discussed in this thesis. Putting this all together, we may ask what is the best combination of hardware, software as well as ML pipeline, and how do they influence each other?

A | Bibliographical Remarks

This thesis is a combination of original work and already published work:

- Parts of the general explanation of the von Neumann architecture in section 2.1 have already been published in [BM18] which was written together with Katharina Morik. I wrote the majority of the paper, implemented the software and performed the experiments. Katharina Morik helped writing the paper.
- Parts of the explanations of caches in section 2.1 have already been published in [BCCM18] which was written together with Kuan-Hsun Chen, Jian-Jia Chen and Katharina Morik. Kuan-Hsun Chen and I wrote the majority of the paper, code and experiments. Jian-Jia Chen and Katharina Morik helped writing the paper.
- Parts of the explanation that DTs are universal function approximators in section 2.4.1 have been published in [BHM20] which was written together with Jan-Philipp Honysz and Katharina Morik. I had originally conceived the paper and wrote the majority of it. Jan-Philipp Honysz did all experiments and implementations for the paper and wrote the corresponding sections in the paper. Katharina Morik helped in writing the paper.
- Parts of the explanation of binarized neural networks in section 2.4.2 have been published in [BPB+20] which was written together with Lukas Pfahler, Jens Buss, Katharina Morik and Wolfgang Rhode. Lukas Pfahler and I conceived the paper and did the majority of implementations and writing. Jens Buss advised us in the details of the FACT telescope and helped us with the data processing. Katharina Morik and Wolfgang Rhode helped writing the paper.
- Parts of the general explanation of the bias-variance decomposition, the resulting GNCL framework and its relationship to existing ensembling methods in chapter 3 and chapter 4 have been published as a pre-print in [BPM20] which was written together with Lukas Pfahler and Katharina Morik. The paper was originally conceived by me and I wrote most of it including the experiments that are not part of this thesis. Lukas Pfahler helped with the derivation of the formulas, wrote parts of the paper and helped with the implementation. Katharina Morik helped writing the paper.
- The study on the double-descent phenomenon in chapter 5 is available as a pre-print in [BM21b] and currently under review. This paper was written together with Katharina Morik. I had originally conceived the paper, did all the implementations and experiments for it and wrote the majority of it. Katharina Morik helped in writing the paper.

- Parts of theoretical execution model of DTs in chapter 8 have already been published in [BM18] which was written together with Katharina Morik. I wrote the majority of the paper, implemented the software and performed the experiments. Katharina Morik helped writing the paper.

- Parts of optimal memory layout of DTs in chapter 8 have already been published in [BCCM18] which was written together with Kuan-Hsun Chen, Jian-Jia Chen and Katharina Morik. Kuan-Hsun Chen and I wrote the majority of the paper, code and experiments. Jian-Jia Chen and Katharina Morik helped writing the paper.

- Parts of chapter 6 have been published as a pre-print in [BM21a] which was written together with Katharina Morik and is currently under review. The paper was originally conceived by me and I wrote most of it, including the implementation and execution of the experiments. Katharina Morik helped writing the paper.

- Parts of chapter 7 have been published in [BHM22] which was written together with Sibylle Hess and Katharina Morik. I had originally conceived the paper, did all the implementations and experiments for it and wrote the majority of it. Sibylle Hess helped in the mathematical derivations and helped writing the paper. Katharina Morik helped in writing the paper

- Parts of the application of the theoretical execution model to FACT in chapter 10.2.2 have already been published in [BM18] which was written together with Katharina Morik. I wrote the majority of the paper, implemented the software and performed the experiments. Katharina Morik helped writing the paper.

- Parts of the application of binarized neural networks in chapter 10.2.3 have been published in [BPB+20] which was written together with Lukas Pfahler, Jens Buss, Katharina Morik and Wolfgang Rhode. Lukas Pfahler and I conceived the paper and did the majority of implementations and writing. Jens Buss advised us in the details of the FACT telescope and helped us with the data processing. Katharina Morik and Wolfgang Rhode helped writing the paper.

- Parts of the application of binarized neural networks in chapter 10.3 have been published in [BCC+21a] and [BCC+21b] that were written together with Jian-Jia Chen, Kuan-Hsun Chen, Mario Günzel, Christian Hakert, Katharina Morik, Rodion Novkin, Lukas Pfahler, and Mikail Yayla. [BCC+21b] is the conceptual predecessor of [BCC+21a]. The main idea for these papers originated from a project meeting in the SFB876-A1 project. The main authors of these papers are Lukas Pfahler, Mikail Yayla and me. I contributed the training code for the experiments. Lukas Pfahler had the idea on max-margin maximization, whereas Mikail Yayla performed the experiments and conducted a theoretical investigation of the bit error tolerance of BNNs (not discussed in this thesis). The remaining authors contributed through discussions and gave feedback during writing.

B | Additional Results for the Experiments in Chapter 6

B.1 Accuracy under Memory Constraints

TABLE B.1: The accuracy score of each method on each dataset computed over a five-fold cross-validation or the train/test split of the dataset (if any) with a model size below 128 KB. A dash '-' indicates that a method did not produce any model that fits the memory constraint. Each entry is rounded to the second digit after to decimal point. Each row represents one dataset and each column is one method. Larger is better. The best method is marked in bold.

	COMP	DREP	IC	IE	L1	L1+LR	LMD	LR	RE	RF
adult	86.56	86.30	86.59	86.86	85.49	**87.25**	86.18	87.01	86.70	85.85
anuran	97.43	97.29	97.36	97.78	0.00	97.86	97.01	**98.05**	97.64	97.33
avila	**92.29**	89.82	91.68	91.77	0.00	78.76	83.39	88.80	91.88	87.75
bank	90.37	90.04	90.16	90.29	89.71	**90.50**	90.10	90.42	90.17	89.90
connect	75.51	75.49	76.15	75.81	69.50	78.11	73.84	**78.72**	75.38	73.67
eeg	87.15	86.68	86.48	87.05	77.45	88.22	85.61	**88.50**	87.52	85.57
elec	85.51	84.46	85.20	85.27	80.55	**86.54**	83.91	86.25	85.22	84.34
fashion	83.20	82.60	83.02	82.73	0.00	83.67	82.65	**84.22**	83.21	83.01
gas-drift	98.96	98.74	98.81	98.99	0.00	99.05	98.63	98.98	**99.07**	98.68
ida2016	99.13	99.12	99.13	99.17	98.91	99.08	99.12	99.16	**99.18**	99.11
japanese-vowels	91.11	90.11	91.16	89.41	0.00	91.46	90.46	**92.65**	91.16	90.40
magic	87.06	86.67	**87.35**	86.88	84.67	87.00	86.77	86.57	86.88	86.46
mnist	90.24	89.64	90.52	89.12	83.57	**92.54**	89.17	92.31	90.30	88.74
mozilla	94.85	94.66	94.79	94.85	0.00	**94.96**	94.85	94.60	94.76	94.60
postures	85.75	85.70	85.38	85.56	0.00	77.34	83.91	**86.63**	85.88	84.68

TABLE B.2: The accuracy score of each method on each dataset computed over a five-fold cross-validation or the train/test split of the dataset (if any) with a model size below 256 KB. A dash '-' indicates that a method did not produce any model that fits the memory constraint. Each entry is rounded to the second digit after to decimal point. Each row represents one dataset and each column is one method. Larger is better. The best method is marked in bold.

	COMP	DREP	IC	IE	L1	L1+LR	LMD	LR	RE	RF
adult	86.75	86.35	86.76	86.86	85.82	**87.25**	86.36	87.01	86.84	86.31
anuran	98.05	97.71	98.05	98.12	0.00	98.30	97.92	**98.37**	98.33	97.93
avila	95.50	94.60	**95.77**	**95.77**	0.00	89.20	87.27	95.24	95.22	89.66
bank	90.37	90.04	90.18	90.29	89.93	**90.50**	90.23	90.42	90.17	90.12
connect	76.69	76.57	77.09	76.77	72.67	**80.27**	74.78	80.06	76.53	74.74
eeg	88.85	88.15	89.09	89.32	80.41	**90.28**	88.18	90.03	89.19	88.28
elec	86.11	85.71	86.22	86.29	81.86	**87.81**	84.97	87.44	86.28	85.14
fashion	83.77	83.84	83.86	83.82	77.46	84.85	83.91	**85.02**	84.01	83.54
gas-drift	99.28	99.21	99.25	99.25	91.66	99.15	**99.35**	99.31	**99.35**	99.20
ida2016	99.20	99.22	99.16	99.18	99.01	**99.29**	99.18	99.19	99.28	99.19
japanese-vowels	92.72	92.17	92.37	92.22	0.00	94.28	92.52	**94.94**	93.07	92.41
magic	87.06	86.83	**87.35**	86.88	85.80	87.27	86.88	86.57	86.88	86.79
mnist	92.42	91.41	92.26	91.66	83.57	93.80	90.99	**94.20**	92.06	91.10
mozilla	95.08	94.85	95.11	95.08	94.18	**95.21**	94.92	94.60	**95.21**	94.78
postures	89.81	88.83	89.26	89.33	65.21	84.06	87.65	**90.49**	89.46	88.35

TABLE B.3: The accuracy score of each method on each dataset computed over a five-fold cross-validation or the train/test split of the dataset (if any) with a model size below 512 KB. A dash '-' indicates that a method did not produce any model that fits the memory constraint. Each entry is rounded to the second digit after to decimal point. Each row represents one dataset and each column is one method. Larger is better. The best method is marked in bold.

	COMP	DREP	IC	IE	L1	L1+LR	LMD	LR	RE	RF
adult	86.95	86.35	86.76	86.86	86.21	**87.25**	86.55	87.01	86.84	86.31
anuran	98.33	98.26	98.12	98.33	94.37	98.64	98.54	**98.75**	98.54	98.29
avila	98.20	94.62	98.06	98.33	66.70	**98.57**	89.33	97.53	98.06	95.59
bank	90.37	90.04	90.19	90.29	90.06	**90.50**	90.25	90.42	90.21	90.15
connect	77.35	77.31	77.57	77.58	73.62	**82.03**	75.58	81.67	77.19	75.61
eeg	90.79	89.52	90.49	90.25	83.26	**92.70**	89.59	91.01	89.79	89.67
elec	86.78	86.59	87.46	87.08	82.76	**89.01**	85.75	88.45	86.99	86.32
fashion	85.10	84.57	84.90	84.54	79.60	85.91	84.61	**86.28**	84.94	84.65
gas-drift	**99.53**	99.35	99.32	99.32	96.23	99.48	99.35	99.42	99.46	99.38
ida2016	99.28	99.22	99.24	99.21	99.03	**99.30**	99.19	99.25	99.28	99.22
japanese-vowels	94.93	93.67	94.28	94.43	82.78	**96.27**	93.67	95.82	94.63	93.54
magic	87.19	86.83	**87.35**	**87.35**	85.83	87.27	87.12	86.57	87.04	86.93
mnist	93.53	92.75	93.44	93.10	87.54	95.58	93.07	**95.79**	93.21	92.68
mozilla	95.27	95.18	95.21	**95.53**	94.27	95.21	94.92	95.08	95.34	95.13
postures	92.67	92.39	92.26	92.55	72.20	92.74	90.95	**93.50**	92.29	91.89

TABLE B.4: The accuracy score of each method on each dataset computed over a five-fold cross-validation or the train/test split of the dataset (if any) with a model size below 1024 KB. A dash '-' indicates that a method did not produce any model that fits the memory constraint. Each entry is rounded to the second digit after to decimal point. Each row represents one dataset and each column is one method. Larger is better. The best method is marked in bold.

	COMP	DREP	IC	IE	L1	L1+LR	LMD	LR	RE	RF
adult	86.95	86.49	86.92	86.86	86.49	**87.25**	86.55	87.01	86.84	86.46
anuran	98.61	98.47	98.47	98.75	96.39	98.74	98.54	**98.99**	98.54	98.46
avila	99.09	98.03	98.90	98.90	75.37	**99.84**	92.03	99.17	98.75	97.23
bank	90.37	90.28	90.36	90.46	90.33	**90.50**	90.38	90.42	90.38	90.15
connect	78.00	77.34	78.37	78.21	74.41	**82.98**	76.64	82.88	77.90	76.59
eeg	91.69	90.85	91.42	91.32	86.26	**93.73**	91.15	91.38	91.66	91.00
elec	88.03	87.30	88.78	88.60	84.22	**90.08**	86.98	89.03	87.77	87.53
fashion	85.50	85.36	85.39	85.39	80.41	86.74	85.21	**87.16**	85.52	85.53
gas-drift	99.53	99.42	99.50	99.46	96.36	**99.59**	99.46	99.55	99.46	99.43
ida2016	99.28	99.22	99.26	99.21	99.11	99.30	99.25	**99.31**	99.28	99.24
japanese-vowels	95.73	95.13	95.33	95.23	87.50	**97.27**	95.03	97.07	95.68	94.96
magic	87.19	87.06	87.38	**87.54**	86.69	87.27	87.22	86.57	87.04	87.14
mnist	94.35	94.10	94.23	94.14	90.86	**96.79**	94.14	96.75	94.37	94.07
mozilla	95.27	95.24	95.30	**95.53**	94.44	**95.53**	95.14	95.19	95.37	95.19
postures	94.21	94.16	94.40	94.14	78.56	94.98	93.19	**95.79**	94.50	94.04

TABLE B.5: The accuracy of each method on each dataset computed over a five-fold cross-validation or the train/test split of the dataset (if any) with a model size below 2048 KB. A dash '-' indicates that a method did not produce any model that fits the memory constraint. Each entry is rounded to the second digit after to decimal point. Each row represents one dataset and each column is one method. Larger is better. The best method is marked in bold.

	COMP	DREP	IC	IE	L1	L1+LR	LMD	LR	RE	RF
adult	86.95	86.67	86.92	86.86	86.82	**87.25**	86.55	87.01	86.92	86.55
anuran	98.89	98.54	98.89	98.82	97.50	99.15	98.54	**99.17**	98.89	98.67
avila	99.21	98.24	99.35	99.40	83.65	**99.84**	92.48	99.61	99.20	97.68
bank	90.42	90.36	90.36	90.46	90.33	**90.50**	90.38	90.42	90.38	90.29
connect	78.64	77.52	78.92	78.77	75.34	**83.95**	77.17	82.88	78.26	77.03
eeg	92.19	91.92	92.46	92.09	86.30	**94.86**	91.72	93.77	92.22	92.13
elec	88.90	88.05	89.12	89.24	85.13	**91.84**	87.75	89.77	88.87	87.97
fashion	85.86	86.06	85.91	86.07	83.45	**87.28**	85.97	87.24	86.16	85.81
gas-drift	99.53	99.42	99.50	99.46	98.31	**99.59**	99.46	99.55	99.46	99.43
ida2016	99.28	99.23	99.26	99.22	99.14	**99.33**	99.25	99.31	99.28	99.24
japanese-vowels	96.34	95.78	96.08	96.03	91.32	**98.09**	96.18	97.85	96.34	96.04
magic	87.40	87.06	87.40	87.54	87.05	87.27	**87.61**	86.57	87.56	87.21
mnist	95.12	94.87	95.21	94.99	90.86	**97.35**	94.88	97.31	95.00	94.67
mozilla	95.27	95.24	95.30	**95.53**	94.63	**95.53**	95.14	95.25	95.37	95.19
postures	95.79	95.66	95.92	95.76	84.30	**97.23**	95.06	96.93	95.94	95.37

B.2 F_1 score under Memory Constraints

TABLE B.6: The F_1 score of each method on each dataset computed over a five-fold cross-validation or the train/test split of the dataset (if any) with a model size below 128 KB. A dash '-' indicates that a method did not produce any model that fits the memory constraint. Each entry is rounded to the fourth digit after to decimal point. Each row represents one dataset and each column is one method. Larger is better. The best method is marked in bold.

	COMP	DREP	IC	IE	L1	L1+LR	LMD	LR	RE	RF
adult	0.8002	0.7957	0.8008	0.8022	0.7699	**0.8136**	0.7950	0.8110	0.8040	0.7860
anuran	0.9409	0.9322	0.9442	0.9505	0.0000	0.9530	0.9333	**0.9566**	0.9374	0.9400
avila	0.8280	0.7948	0.8387	**0.8897**	0.0000	0.6399	0.7273	0.8746	0.8271	0.8278
bank	0.7128	0.6787	0.7155	0.7088	0.6343	0.7423	0.7188	**0.7430**	0.7173	0.6880
connect	0.5135	0.5305	0.5305	0.5233	0.3624	0.5274	0.4881	**0.5637**	0.5118	0.4834
eeg	0.8696	0.8642	0.8629	0.8685	0.7675	0.8805	0.8534	**0.8837**	0.8733	0.8532
elec	0.8508	0.8397	0.8466	0.8484	0.7972	**0.8616**	0.8331	0.8589	0.8474	0.8384
fashion	0.8290	0.8233	0.8269	0.8241	0.0000	0.8339	0.8240	**0.8402**	0.8296	0.8267
gas-drift	0.9888	0.9862	0.9871	0.9890	0.0000	0.9895	0.9852	0.9889	**0.9897**	0.9858
ida2016	0.8904	0.8894	0.8926	0.8973	0.8532	0.8871	0.8884	**0.9008**	0.8976	0.8892
japanese-vowels	0.9047	0.8940	0.9051	0.8877	0.0000	0.9074	0.8995	**0.9209**	0.9041	0.8967
magic	0.8540	0.8498	**0.8564**	0.8515	0.8219	0.8542	0.8504	0.8491	0.8519	0.8479
mnist	0.9008	0.8948	0.9037	0.8895	0.8296	**0.9244**	0.8899	0.9220	0.9016	0.8860
mozilla	0.9397	0.9377	0.9388	0.9400	0.0000	**0.9413**	0.9398	0.9369	0.9384	0.9370
postures	0.8558	0.8550	0.8519	0.8538	0.0000	0.7690	0.8363	**0.8645**	0.8569	0.8446

TABLE B.7: The F_1 score of each method on each dataset computed over a five-fold cross-validation or the train/test split of the dataset (if any) with a model size below 256 KB. A dash '-' indicates that a method did not produce any model that fits the memory constraint. Each entry is rounded to the fourth digit after to decimal point. Each row represents one dataset and each column is one method. Larger is better. The best method is marked in bold.

	COMP	DREP	IC	IE	L1	L1+LR	LMD	LR	RE	RF
adult	0.8015	0.7991	0.8014	0.8022	0.7810	**0.8136**	0.7955	0.8110	0.8067	0.7944
anuran	0.9562	0.9431	0.9547	0.9572	0.0000	0.9574	0.9551	**0.9635**	0.9584	0.9543
avila	0.9216	0.9337	**0.9434**	**0.9434**	0.0000	0.8796	0.7654	0.9374	0.9346	0.8606
bank	0.7128	0.6916	0.7155	0.7094	0.6792	**0.7480**	0.7188	0.7430	0.7173	0.6960
connect	0.5450	0.5526	0.5526	0.5537	0.4440	0.5758	0.5149	**0.5997**	0.5346	0.5234
eeg	0.8870	0.8796	0.8893	0.8915	0.7978	**0.9016**	0.8801	0.8991	0.8902	0.8810
elec	0.8573	0.8528	0.8580	0.8590	0.8120	**0.8748**	0.8447	0.8711	0.8588	0.8465
fashion	0.8352	0.8365	0.8370	0.8354	0.7651	0.8472	0.8362	**0.8495**	0.8375	0.8336
gas-drift	0.9921	0.9912	0.9918	0.9921	0.9118	0.9907	**0.9929**	0.9923	0.9928	0.9913
ida2016	0.9007	0.9041	0.8935	0.8991	0.8713	**0.9126**	0.8988	0.9008	0.9118	0.9001
japanese-vowels	0.9228	0.9164	0.9181	0.9157	0.0000	0.9389	0.9194	**0.9463**	0.9266	0.9190
magic	0.8541	0.8513	0.8564	0.8517	0.8372	**0.8578**	0.8516	0.8491	0.8519	0.8516
mnist	0.9231	0.9128	0.9215	0.9155	0.8296	0.9372	0.9085	**0.9412**	0.9195	0.9099
mozilla	0.9432	0.9406	0.9435	0.9433	0.9312	0.9444	0.9405	0.9369	**0.9448**	0.9399
postures	0.8969	0.8870	0.8913	0.8921	0.6454	0.8377	0.8745	**0.9038**	0.8933	0.8821

TABLE B.8: The F_1 score of each method on each dataset computed over a five-fold cross-validation or the train/test split of the dataset (if any) with a model size below 512 KB. A dash '-' indicates that a method did not produce any model that fits the memory constraint. Each entry is rounded to the fourth digit after to decimal point. Each row represents one dataset and each column is one method. Larger is better. The best method is marked in bold.

	COMP	DREP	IC	IE	L1	L1+LR	LMD	LR	RE	RF
adult	0.8066	0.7991	0.8045	0.8022	0.7872	**0.8137**	0.8004	0.8110	0.8067	0.7955
anuran	0.9616	0.9581	0.9547	0.9636	0.8398	0.9716	0.9645	**0.9741**	0.9676	0.9593
avila	0.9812	0.9338	0.9758	**0.9825**	0.4289	0.9756	0.8242	0.9720	0.9758	0.9358
bank	0.7128	0.7005	0.7155	0.7130	0.6886	**0.7502**	0.7188	0.7434	0.7173	0.6982
connect	0.5472	0.5526	0.5566	0.5541	0.4575	0.6226	0.5162	**0.6335**	0.5436	0.5234
eeg	0.9065	0.8936	0.9034	0.9009	0.8280	**0.9261**	0.8942	0.9091	0.8957	0.8952
elec	0.8639	0.8617	0.8708	0.8672	0.8215	**0.8872**	0.8529	0.8816	0.8664	0.8591
fashion	0.8492	0.8435	0.8472	0.8433	0.7888	0.8579	0.8431	**0.8621**	0.8475	0.8442
gas-drift	**0.9949**	0.9928	0.9926	0.9927	0.9606	0.9942	0.9929	0.9936	0.9940	0.9931
ida2016	0.9130	0.9041	0.9068	0.9040	0.8733	**0.9163**	0.9004	0.9106	0.9118	0.9029
japanese-vowels	0.9454	0.9317	0.9394	0.9395	0.8063	**0.9611**	0.9334	0.9552	0.9429	0.9314
magic	0.8559	0.8513	0.8564	**0.8583**	0.8376	0.8578	0.8549	0.8491	0.8551	0.8533
mnist	0.9345	0.9265	0.9335	0.9301	0.8733	0.9552	0.9298	**0.9575**	0.9312	0.9257
mozilla	0.9452	0.9439	0.9444	**0.9482**	0.9325	0.9447	0.9405	0.9433	0.9458	0.9436
postures	0.9259	0.9232	0.9217	0.9246	0.7161	0.9264	0.9083	**0.9343**	0.9221	0.9180

TABLE B.9: The F_1 score of each method on each dataset computed over a five-fold cross-validation or the train/test split of the dataset (if any) with a model size below 1024 KB. A dash '-' indicates that a method did not produce any model that fits the memory constraint. Each entry is rounded to the fourth digit after to decimal point. Each row represents one dataset and each column is one method. Larger is better. The best method is marked in bold.

	COMP	DREP	IC	IE	L1	L1+LR	LMD	LR	RE	RF
adult	0.8066	0.8002	0.8071	0.8022	0.7944	**0.8137**	0.8004	0.8110	0.8067	0.8006
anuran	0.9688	0.9598	0.9669	0.9708	0.9082	0.9732	0.9645	**0.9762**	0.9712	0.9652
avila	0.9882	0.9802	0.9842	0.9838	0.5684	**0.9973**	0.8896	0.9867	0.9876	0.9660
bank	0.7159	0.7117	0.7155	0.7206	0.7064	**0.7502**	0.7188	0.7434	0.7276	0.7049
connect	0.5472	0.5526	0.5571	0.5541	0.4675	0.6517	0.5162	**0.6545**	0.5436	0.5234
eeg	0.9156	0.9072	0.9129	0.9119	0.8595	**0.9365**	0.9102	0.9127	0.9154	0.9087
elec	0.8768	0.8693	0.8846	0.8828	0.8371	**0.8982**	0.8658	0.8877	0.8744	0.8717
fashion	0.8531	0.8516	0.8523	0.8522	0.7973	0.8662	0.8498	**0.8711**	0.8532	0.8531
gas-drift	0.9949	0.9937	0.9946	0.9940	0.9620	**0.9956**	0.9939	0.9950	0.9940	0.9938
ida2016	0.9130	0.9041	0.9095	0.9040	0.8871	0.9163	0.9075	**0.9179**	0.9118	0.9056
japanese-vowels	0.9536	0.9479	0.9502	0.9477	0.8640	**0.9712**	0.9463	0.9688	0.9537	0.9464
magic	0.8563	0.8543	0.8590	**0.8601**	0.8488	0.8578	0.8563	0.8491	0.8551	0.8552
mnist	0.9428	0.9403	0.9417	0.9406	0.9073	**0.9675**	0.9407	0.9671	0.9431	0.9400
mozilla	0.9452	0.9448	0.9455	0.9482	0.9345	**0.9485**	0.9438	0.9446	0.9460	0.9440
postures	0.9416	0.9410	0.9434	0.9408	0.7809	0.9494	0.9311	**0.9575**	0.9444	0.9397

TABLE B.10: The F_1 score of each method on each dataset computed over a five-fold cross-validation or the train/test split of the dataset (if any) with a model size below 2048 KB. A dash '-' indicates that a method did not produce any model that fits the memory constraint. Each entry is rounded to the fourth digit after to decimal point. Each row represents one dataset and each column is one method. Larger is better. The best method is marked in bold.

	COMP	DREP	IC	IE	L1	L1+LR	LMD	LR	RE	RF
adult	0.8072	0.8023	0.8071	0.8023	0.8013	**0.8137**	0.8020	0.8110	0.8075	0.8006
anuran	0.9754	0.9666	0.9740	0.9745	0.9376	0.9804	0.9652	**0.9809**	0.9750	0.9680
avila	0.9906	0.9826	0.9914	0.9899	0.7245	**0.9973**	0.9191	0.9946	0.9921	0.9758
bank	0.7183	0.7117	0.7191	0.7244	0.7064	**0.7502**	0.7188	0.7434	0.7276	0.7114
connect	0.5472	0.5526	0.5571	0.5541	0.4781	**0.6784**	0.5162	0.6681	0.5436	0.5234
eeg	0.9206	0.9179	0.9234	0.9198	0.8599	**0.9479**	0.9158	0.9369	0.9209	0.9202
elec	0.8858	0.8769	0.8882	0.8894	0.8467	**0.9163**	0.8736	0.8954	0.8854	0.8762
fashion	0.8567	0.8591	0.8572	0.8590	0.8305	**0.8720**	0.8578	0.8719	0.8599	0.8563
gas-drift	0.9949	0.9937	0.9946	0.9940	0.9824	**0.9956**	0.9939	0.9950	0.9940	0.9938
ida2016	0.9130	0.9053	0.9095	0.9040	0.8910	**0.9186**	0.9075	0.9179	0.9118	0.9056
japanese-vowels	0.9599	0.9549	0.9576	0.9565	0.9069	**0.9795**	0.9593	0.9768	0.9605	0.9576
magic	0.8585	0.8548	0.8590	0.8601	0.8536	0.8578	**0.8612**	0.8491	0.8603	0.8570
mnist	0.9507	0.9482	0.9516	0.9494	0.9073	**0.9732**	0.9482	0.9728	0.9494	0.9461
mozilla	0.9452	0.9448	0.9455	0.9482	0.9369	**0.9485**	0.9438	0.9452	0.9460	0.9440
postures	0.9575	0.9562	0.9587	0.9573	0.8404	**0.9721**	0.9501	0.9690	0.9590	0.9532

C | Additional Results for the Experiments in Chapter 7

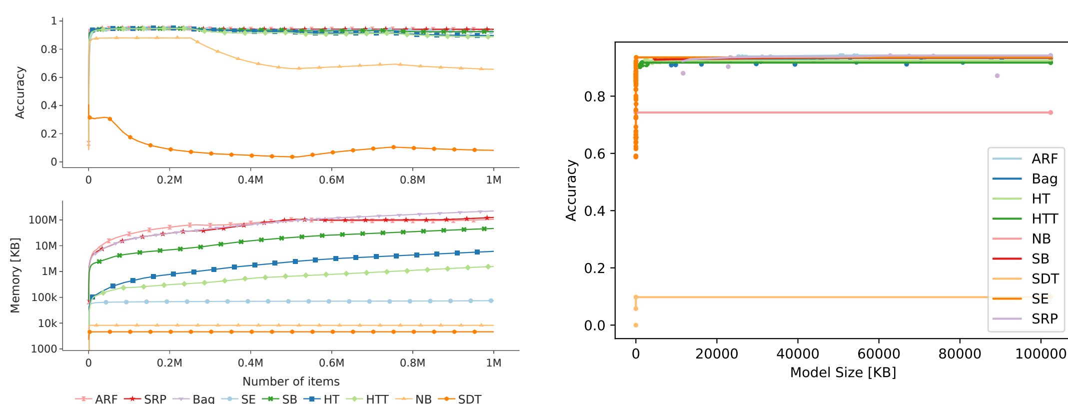


FIGURE C.1: (left) Test-then-train accuracy and memory consumption on the agrawal_a dataset of the best configuration over the number of data items in the stream. (right) Pareto front on the agrawal_a dataset of each method.

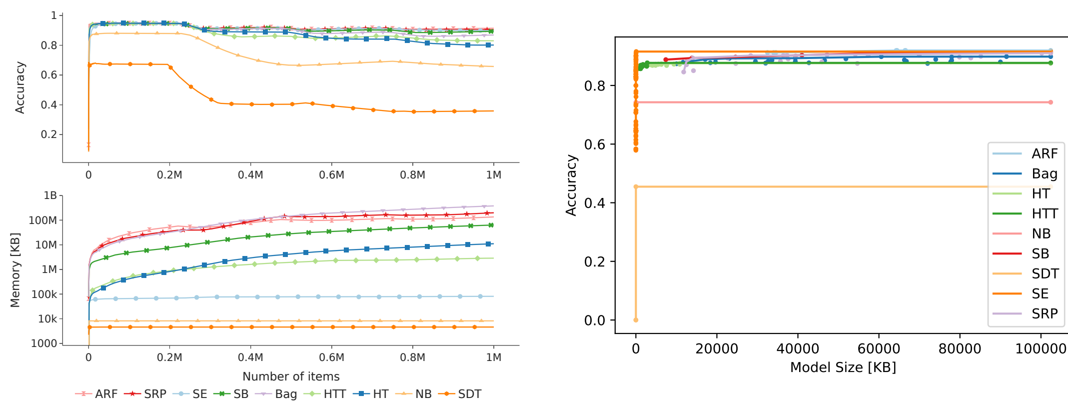


FIGURE C.2: (left) Test-then-train accuracy and memory consumption on the agrawal_g dataset of the best configuration over the number of data items in the stream. (right) Pareto front on the agrawal_g dataset of each method.

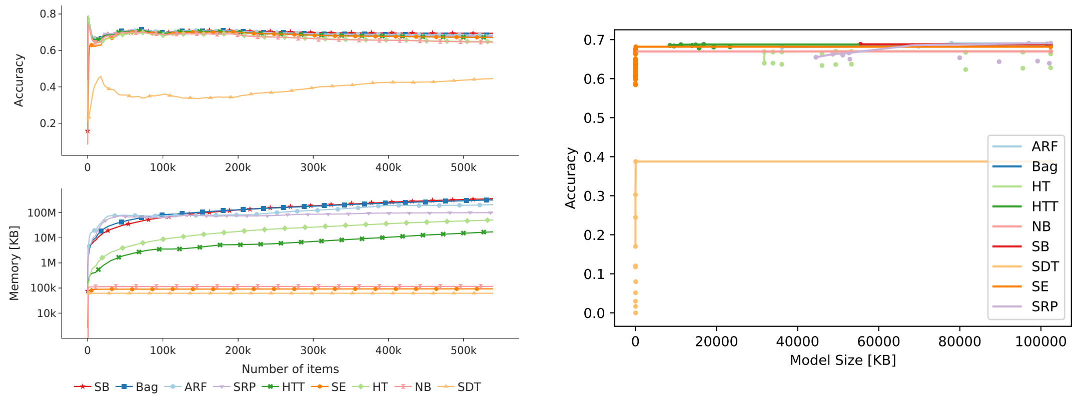


FIGURE C.3: (left) Test-then-train accuracy and memory consumption on the airlines dataset of the best configuration over the number of data items in the stream. (right) Pareto front on the airlines dataset of each method.

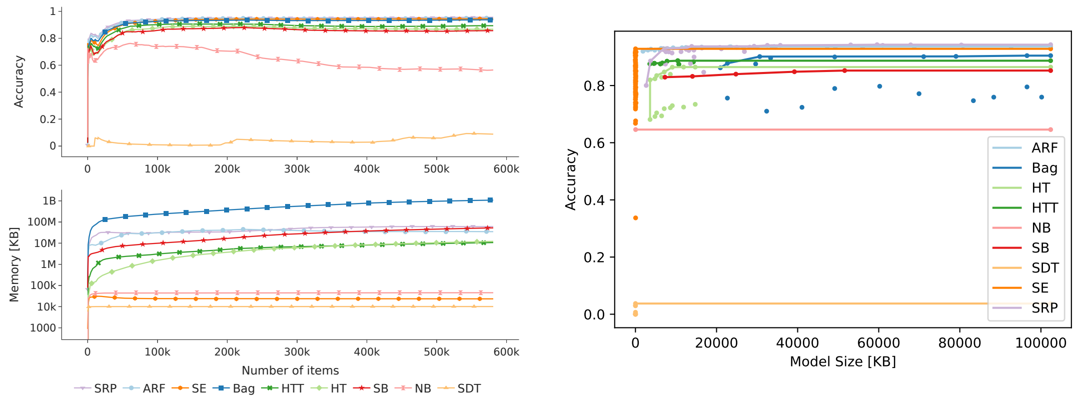


FIGURE C.4: (left) Test-then-train accuracy and memory consumption on the covtype dataset of the best configuration over the number of data items in the stream. (right) Pareto front on the covtype dataset of each method.

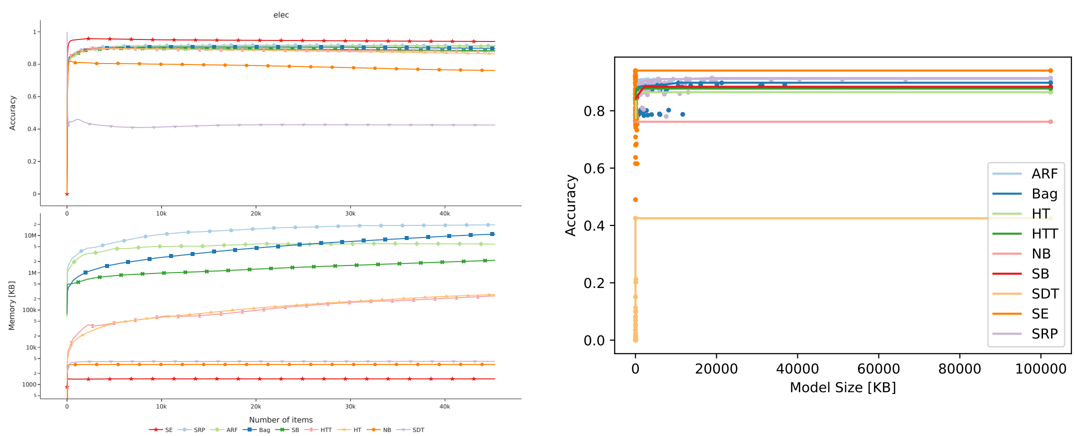


FIGURE C.5: (left) Test-then-train accuracy and memory consumption on the elec dataset of the best configuration over the number of data items in the stream. (right) Pareto front on the elec dataset of each method.

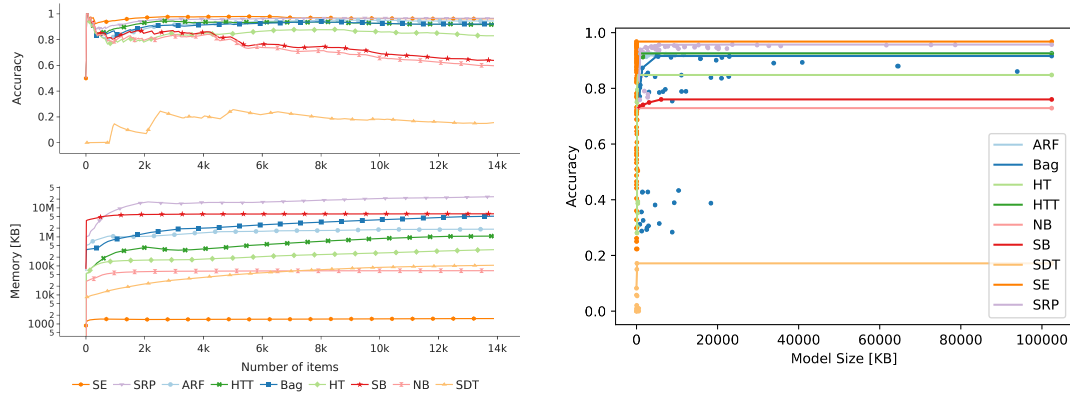


FIGURE C.6: (left) Test-then-train accuracy and memory consumption on the gas-sensor dataset of the best configuration over the number of data items in the stream. (right) Pareto front on the gas-sensor dataset of each method.

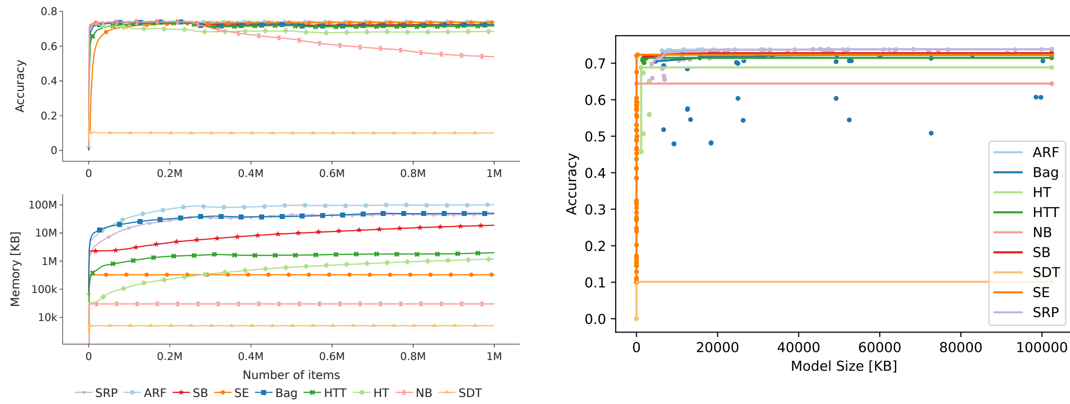


FIGURE C.7: (left) Test-then-train accuracy and memory consumption on the led_a dataset of the best configuration over the number of data items in the stream. (right) Pareto front on the led_a dataset of each method.

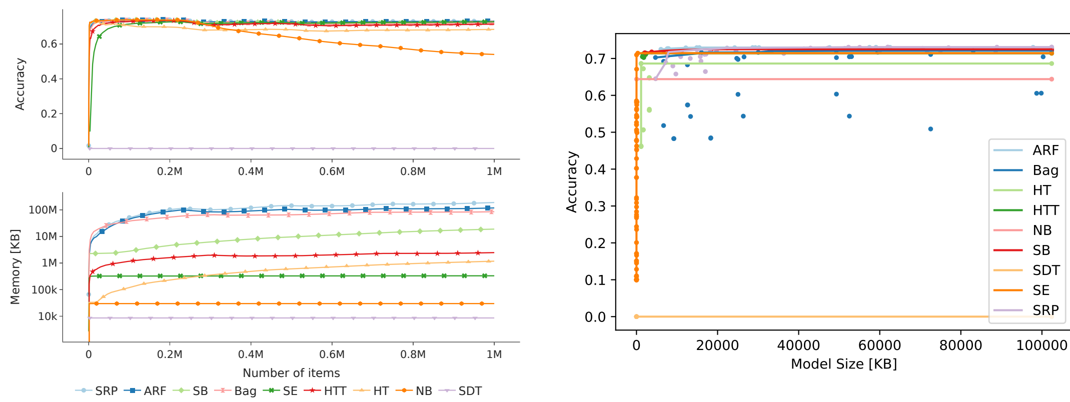


FIGURE C.8: (left) Test-then-train accuracy and memory consumption on the led_g dataset of the best configuration over the number of data items in the stream. (right) Pareto front on the led_g dataset of each method.

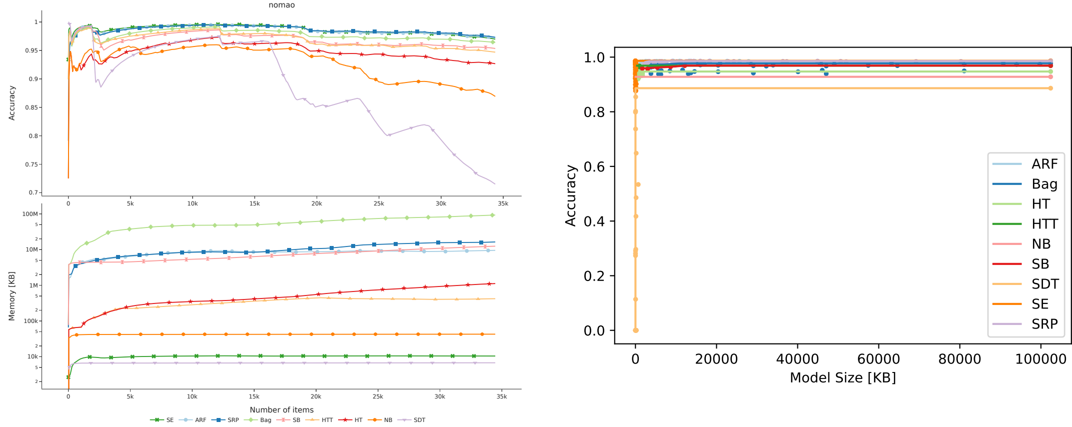


FIGURE C.9: (left) Test-then-train accuracy and memory consumption on the nomao dataset of the best configuration over the number of data items in the stream. (right) Pareto front on the nomao dataset of each method.

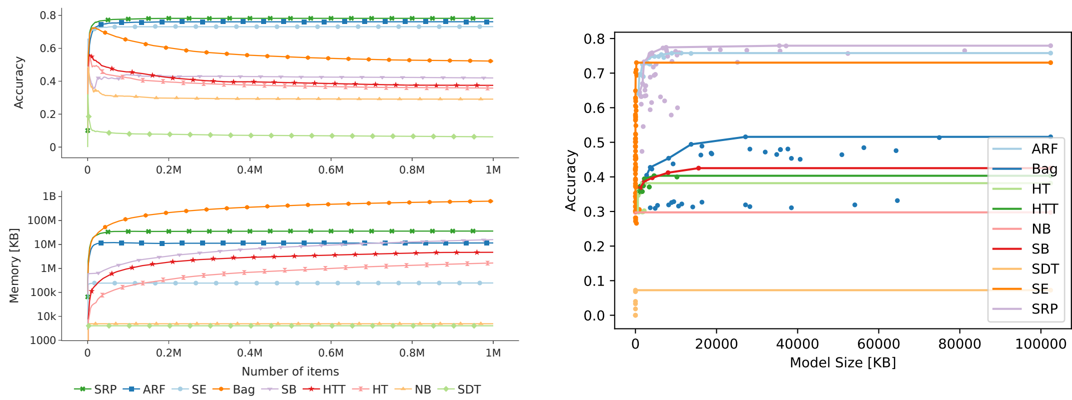


FIGURE C.10: (left) Test-then-train accuracy and memory consumption on the rbf_f dataset of the best configuration over the number of data items in the stream. (right) Pareto front on the rbf_f dataset of each method.

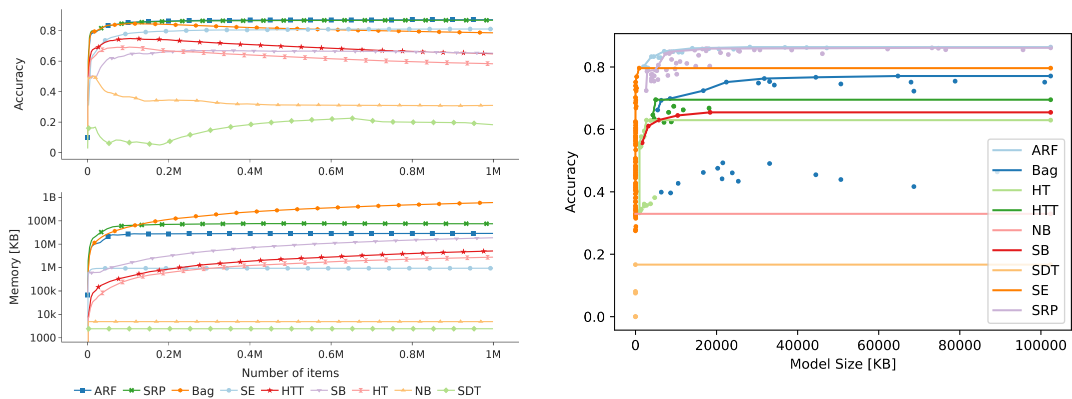


FIGURE C.11: (left) Test-then-train accuracy and memory consumption on the rbf_m dataset of the best configuration over the number of data items in the stream. (right) Pareto front on the rbf_m dataset of each method.

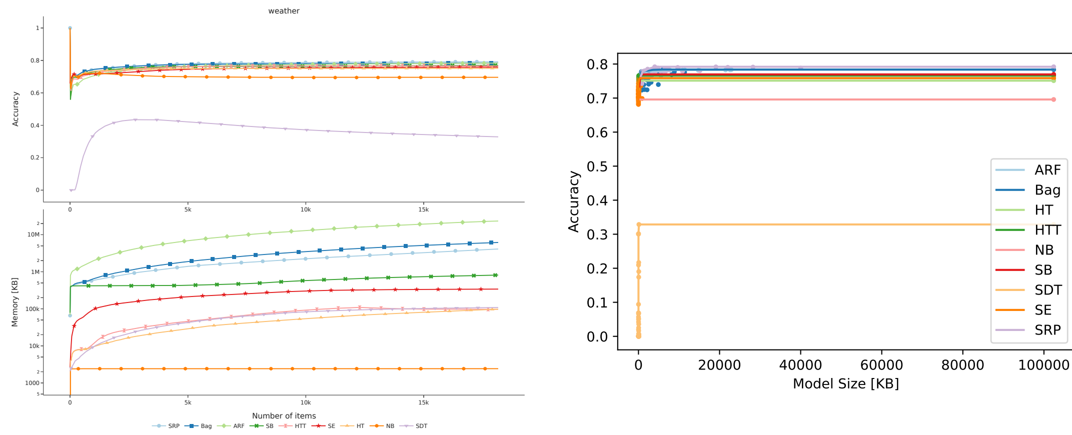


FIGURE C.12: (left) Test-then-train accuracy and memory consumption on the weather dataset of the best configuration over the number of data items in the stream. (right) Pareto front on the weather dataset of each method.

Bibliography

[ABB$^+$13] H. Anderhub, Michael Backes, A. Biland, Vittorio Boccone, Isabel Braun, Thomas Bretz, Jens Bjoern Buss, F. Cadoux, V. Commichau, L. Djambazov, D. Dorner, S. Einecke, Dorit Eisenacher Glawion, A. Gendotti, O. Grimm, H. Gunten, C. Haller, D. Hildebrand, U. Horisberger, and M. Z"anglein. Design and operation of FACT-the first G-APD Cherenkov telescope. *Journal of Instrumentation*, 8(6), 2013. doi:10.1088/1748-0221/8/06/P06008.

[ACB] Devansh Arpit, Víctor Campos, and Yoshua Bengio. How to initialize your network? robust initialization for weightnorm & resnets. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 10900–10909. URL: https://proceedings.neurips.cc/paper/2019/hash/e520f70ac3930490458892665cda6620-Abstract.html.

[AGCH19] Sanjeev Arora, Noah Golowich, Nadav Cohen, and Wei Hu. A convergence analysis of gradient descent for deep linear neural networks. In *7th International Conference on Learning Representations, ICLR 2019*, 2019. arXiv:arXiv:1810.02281v3.

[AGPR07] Davide Anguita, Alessandro Ghio, Stefano Pischiutta, and Sandro Ridella. A hardware-friendly support vector machine for embedded automotive applications. In *2007 International Joint Conference on Neural Networks*, pages 1360 – 1364. IEEE, 2007. doi:10.1109/IJCNN.2007.4371156.

[AİA18] Alper Ahmetoğlu, Ozan İrsoy, and Ethem Alpaydın. Convolutional soft decision trees. In *International Conference on Artificial Neural Networks*, pages 134–141. Springer, 2018. doi:10.1007/978-3-030-01418-6_14.

[AKA$^+$19] Pritom Saha Akash, Md. Eusha Kadir, Amin Ahsan Ali, Md. Nurul Ahad Tawhid, and Mohammad Shoyaib. Introducing confidence as a weight in random forest. In *2019 International Conference on Robotics, Electrical and Signal Processing Techniques (ICREST)*, pages 611–616. IEEE, 2019. doi:10.1109/icrest.2019.8644396.

[ALdV14] Nima Asadi, Jimmy Lin, and Arjen P. de Vries. Runtime optimizations for tree-based machine learning models. *IEEE Transactions on Knowledge and Data Engineering*, 26(9):2281–2292, Sept 2014. doi:10.1109/TKDE.2013.73.

[AMS07] Jean-Yves Audibert, Rémi Munos, and Csaba Szepesvári. Variance estimates and exploration function in multi-armed bandit. In *CERTIS Research Report 07–31*. 2007. URL: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.136.546&rep=rep1&type=pdf.

[ANS20] Gaël Aglin, Siegfried Nijssen, and Pierre Schaus. Learning optimal decision trees using caching branch-and-bound search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 3146–3153, 2020. doi:10.1609/aaai.v34i04.5711.

[APP+18] Andreea Anghel, Nikolaos Papandreou, Thomas Parnell, Alessandro De Palma, and Haralampos Pozidis. Benchmarking and optimization of gradient boosting decision tree algorithms, 2018. doi:10.48550/ARXIV.1809.04559.

[ASM08] Hanady Abdulsalam, David B. Skillicorn, and Patrick Martin. Classifying evolving data streams using dynamic streaming random forests. In *International Conference on Database and Expert Systems Applications*, volume 5181, pages 643–651, 2008. doi:10.1007/978-3-540-85654-2_54.

[AYA] Ozlem Asian, Olcay Taner Yildiz, and Ethem Alpaydin. Calculating the vc-dimension of decision trees. In *24th International Symposium on Computer and Information Sciences*, pages 193–198. IEEE. doi:10.1109/iscis.2009.5291847.

[AZTK18] Ahmad Al-Zoubi, Konstantinos Tatas, and Costas Kyriacou. Design space exploration of the knn imputation on fpga. In *2018 7th International Conference on Modern Circuits and Systems Technologies (MOCAST)*, pages 1–4. IEEE, 2018. doi:10.1109/MOCAST.2018.8376606.

[B+14] A. Biland et al. Calibration and performance of the photon sensor response of FACT - The first G-APD Cherenkov telescope. *Journal of Instrumentation*, 9(10), 2014. doi:10.1088/1748-0221/9/10/P10012.

[BAP14] Philip Bachman, Ouais Alsharif, and Doina Precup. Learning with pseudo-ensembles. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014. URL: https://proceedings.neurips.cc/paper/2014/file/66be31e4c40d676991f2405aaecc6934-Paper.pdf.

[Bay63] Thomas Bayes. Lii. an essay towards solving a problem in the doctrine of chances. by the late rev. mr. bayes, frs communicated by mr. price, in a letter to john canton, amfr s. *Philosophical transactions of the Royal Society of London*, (53):370–418, 1763. URL: http://www.jstor.org/stable/105741.

[BB96] Kristin P. Bennett and Jennifer A. Blue. Optimal decision trees. Technical report, Rensselaer Polytechnic Institute Math Report, 1996. URL: https://www.researchgate.net/profile/Kristin-Bennett/publication/2796065_Optimal_Decision_Trees/links/00b4951d958261226a000000/Optimal-Decision-Trees.pdf.

[BB12] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13:281–305, 2012. URL: https://dl.acm.org/doi/10.5555/2503308.2188395, doi:10.5555/2503308.2188395.

[BBB+15] Christian Bockermann, Kai Brügge, Jens Buß, Alexey Egorov, Katharina Morik, Wolfgang Rhode, and Tim Ruhe. Online analysis of high-volume data streams in astroparticle physics. In Albert Bifet, Michael May, Bianca Zadrozny, Ricard Gavaldà, Dino Pedreschi, Francesco Bonchi, Jaime S. Cardoso, and Myra Spiliopoulou, editors, *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2015, Porto, Portugal, September 7-11, 2015, Proceedings, Part III*, volume 9286 of *Lecture Notes in Computer Science*, pages 100–115. Springer, 2015. doi:10.1007/978-3-319-23461-8_7.

[BC] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77. doi:10.1145/1941487.1941507.

[BCC+21a] Sebastian Buschjäger, Jian-Jia Chen, Kuan-Hsun Chen, Mario Günzel, Christian Hakert, Katharina Morik, Rodion Novkin, Lukas Pfahler, and Mikail Yayla. Margin-maximization in binarized neural networks for optimizing bit error tolerance. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2021, Grenoble, France, February 1-5, 2021*, pages 673–678. IEEE, 2021. doi:10.23919/DATE51398.2021.9473918.

[BCC+21b] Sebastian Buschjäger, Jian-Jia Chen, Kuan-Hsun Chen, Mario Günzel, Katharina Morik, Rodion Novkin, Lukas Pfahler, and Mikail Yayla. Bit error tolerance metrics for binarized neural networks. *CoRR*, abs/2102.01344, 2021. URL: https://arxiv.org/abs/2102.01344, arXiv:2102.01344.

[BCCM18] Sebastian Buschjäger, Kuan-Hsun Chen, Jian-Jia Chen, and Katharina Morik. Realization of random forest for real-time evaluation through tree framing. In *The IEEE International Conference on Data Mining series (ICDM)*, pages 19–28, November 2018. URL: https://ieeexplore.ieee.org/document/8594826, doi:10.1109/ICDM.2018.00017.

[BCN16] Léon Bottou, Frank E. Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. *CoRR*, abs/1606.04838, 2016. URL: http://arxiv.org/abs/1606.04838, arXiv:1606.04838.

[BdCF15] Rodrigo C. Barros, André C. P. L. F. de Carvalho, and Alex A. Freitas. *Decision-Tree Induction*. Springer International Publishing, Cham, 2015. doi:10.1007/978-3-319-14231-9_2.

[BDL08] Gérard Biau, Luc Devroye, and Gábor Lugosi. Consistency of random forests and other averaging classifiers. *Journal of Machine Learning Research*, 9:2015–2033, 2008. URL: https://dl.acm.org/doi/10.5555/1390681.1442799, doi:10.5555/1390681.1442799.

[BDPG+15] Mario Barbareschi, Salvatore Del Prete, Francesco Gargiulo, Antonino Mazzeo, and Carlo Sansone. Decision tree-based multiple classifier

systems: An fpga perspective. In *International Workshop on Multiple Classifier Systems*, pages 194–205. Springer, 2015. doi:10.1007/978-3-319-20248-8_17.

[Ben92] Kristin Bennett. Decision tree construction via linear programming. Technical report, Computer Sciences Department, University of Wisconsin, 1992. URL: https://minds.wisconsin.edu/bitstream/handle/1793/59564/TR1067.pdf?sequence=1.

[BFC19] Sérgio Branco, André G. Ferreira, and Jorge Cabral. Machine learning in resource-scarce embedded systems, fpgas, and end-devices: A survey. *Electronics*, 8(11), 2019. URL: https://www.mdpi.com/2079-9292/8/11/1289, doi:10.3390/electronics8111289.

[BFSO] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen. *Classification and Regression Trees*. Taylor & Francis. The reprint from 1993 was used. URL: https://books.google.de/books?id=JwQx-WOmSyQC.

[BG07] Albert Bifet and Ricard Gavaldà. Learning from time-changing data with adaptive windowing. In *Proceedings of the Seventh SIAM International Conference on Data Mining, April 26-28, 2007, Minneapolis, Minnesota, USA*, pages 443–448. SIAM, 2007. doi:10.1137/1.9781611972771.42.

[BG09] Albert Bifet and Ricard Gavaldà. Adaptive learning from evolving data streams. In Niall M. Adams, Céline Robardet, Arno Siebes, and Jean-François Boulicaut, editors, *Advances in Intelligent Data Analysis VIII, 8th International Symposium on Intelligent Data Analysis, IDA 2009, Lyon, France, August 31 - September 2, 2009. Proceedings*, volume 5772 of *Lecture Notes in Computer Science*, pages 249–260. Springer, 2009. doi:10.1007/978-3-642-03915-7_22.

[BH03] Bart Bakker and Tom Heskes. Clustering ensembles of neural network models. *Neural networks*, 16(2):261–269, 2003. doi:10.1016/s0893-6080(02)00187-9.

[BHKL15] Alina Beygelzimer, Elad Hazan, Satyen Kale, and Haipeng Luo. Online gradient boosting. In Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 2458–2466, 2015. URL: https://proceedings.neurips.cc/paper/2015/hash/0a1bf96b7165e962e90cb14648c9462d-Abstract.html.

[BHKP10] Albert Bifet, Geoff Holmes, Richard Kirkby, and Bernhard Pfahringer. MOA: massive online analysis. *Journal of Machine Learning Research*, 11:1601–1604, 2010. URL: http://portal.acm.org/citation.cfm?id=1859903.

[BHM20] Sebastian Buschjäger, Philipp-jan Honysz, and Katharina Morik. Randomized outlier detection with trees. *International Journal of Data Science and Analytics*, 13(2):91–104, 2020. doi:10.1007/s41060-020-00238-w.

[BHM22] Sebastian Buschjäger, Sibylle Hess, and Katharina Morik. Shrub ensembles for online classification. In *Proceedings of the Thirty-Sixth AAAI Conference on Artificial Intelligence (AAAI-22)*. AAAI Press, 2022. URL: https://arxiv.org/abs/2112.03723, doi:10.1609/aaai.v36i6.20560.

[BHMM19] Mikhail Belkin, Daniel Hsu, Siyuan Ma, and Soumik Mandal. Reconciling modern machine-learning practice and the classical bias–variance trade-off. *Proceedings of the National Academy of Sciences*, 116(32):15849–15854, 2019. URL: https://www.pnas.org/doi/abs/10.1073/pnas.1903070116, arXiv:https://www.pnas.org/doi/pdf/10.1073/pnas.1903070116, doi:10.1073/pnas.1903070116.

[BHO09] Christian Bessiere, Emmanuel Hebrard, and Barry O'Sullivan. Minimising decision tree size as combinatorial optimisation. In *International Conference on Principles and Practice of Constraint Programming*, pages 173–187. Springer, 2009. doi:10.1007/978-3-642-04244-7_16.

[BHP10] Albert Bifet, Geoffrey Holmes, and Bernhard Pfahringer. Leveraging bagging for evolving data streams. In José L. Balcázar, Francesco Bonchi, Aristides Gionis, and Michèle Sebag, editors, *Machine Learning and Knowledge Discovery in Databases, European Conference, ECML PKDD 2010, Barcelona, Spain, September 20-24, 2010, Proceedings, Part I*, volume 6321 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 2010. doi:10.1007/978-3-642-15880-3_15.

[BHPM21] Sebastian Buschjäger, Philipp-Jan Honysz, Lukas Pfahler, and Katharina Morik. Very fast streaming submodular function maximization. In Nuria Oliver, Fernando Pérez-Cruz, Stefan Kramer, Jesse Read, and José Antonio Lozano, editors, *Machine Learning and Knowledge Discovery in Databases. Research Track - European Conference, ECML PKDD 2021, Bilbao, Spain, September 13-17, 2021, Proceedings, Part III*, volume 12977 of *Lecture Notes in Computer Science*, pages 151–166. Springer, 2021. doi:10.1007/978-3-030-86523-8_10.

[BK99] Eric Bauer and Ron Kohavi. An empirical comparison of voting classification algorithms: Bagging, boosting, and variants. *Machine Learning*, 36:105–139, 1999. doi:10.1023/A:1007515423169.

[BLM19] Sebastian Buschjäger, Thomas Liebig, and Katharina Morik. Gaussian model trees for traffic imputation. In *Proceedings of the International Conference on Pattern Recognition Applications and Methods (ICPRAM)*, pages 243–254. SciTePress, 2019. URL: https://www.scitepress.org/PublicationsDetail.aspx?ID=g+tVIY+KNts=&t=1.

[BM18] Sebastian Buschjäger and Katharina Morik. Decision tree and random forest implementations for fast filtering of sensor data. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 65-I(1):209–222, 2018. doi:10.1109/TCSI.2017.2710627.

[BM21a] Sebastian Buschjäger and Katharina Morik. Joint leaf-refinement and ensemble pruning through l1 regularization. *CoRR*, abs/2110.10075, 2021. URL: https://arxiv.org/abs/2110.10075, arXiv:2110.10075.

[BM21b] Sebastian Buschjäger and Katharina Morik. There is no double-descent in random forests. *CoRR*, abs/2111.04409, 2021. URL: https://arxiv.org/abs/2111.04409, arXiv:2111.04409.

[BMS17] Sebastian Buschjäger, Katharina Morik, and Maik Schmidt. Summary extraction on data streams in embedded systems. In Moamar Sayed Mouchaweh, Albert Bifet, Hamid Bouchachia, João Gama, and Rita Paula Ribeiro, editors, *Proceedings of the Workshop on IoT Large Scale Learning from Data Streams co-located with the 2017 European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML-PKDD 2017), Skopje, Macedonia, September 18-22, 2017*, volume 1958 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2017. URL: http://ceur-ws.org/Vol-1958/IOTSTREAMING3.pdf.

[Boc15] Christian Bockermann. *Mining big data streams for multiple concepts*. PhD thesis, Technical University of Dortmund, Germany, 2015. URL: https://hdl.handle.net/2003/34363.

[BOFG20] Davis W. Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John V. Guttag. What is the state of neural network pruning? In Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze, editors, *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*. mlsys.org, 2020. URL: https://proceedings.mlsys.org/book/296.pdf.

[BPB+20] Sebastian Buschjäger, Lukas Pfahler, Jens Buß, Katharina Morik, and Wolfgang Rhode. On-site gamma-hadron separation with deep learning on fpgas. In Yuxiao Dong, Dunja Mladenic, and Craig Saunders, editors, *Machine Learning and Knowledge Discovery in Databases: Applied Data Science Track - European Conference, ECML PKDD 2020, Ghent, Belgium, September 14-18, 2020, Proceedings, Part IV*, volume 12460 of *Lecture Notes in Computer Science*, pages 478–493. Springer, 2020. doi:10.1007/978-3-030-67667-4_29.

[BPM20] Sebastian Buschjäger, Lukas Pfahler, and Katharina Morik. Generalized negative correlation learning for deep ensembling. *CoRR*, abs/2011.02952, 2020. URL: https://arxiv.org/abs/2011.02952, arXiv:2011.02952.

[BPRS17] Atılım Günes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: A survey. *Journal of Machine Learning Research*, 18(1):5595–5637, jan 2017. URL: http://jmlr.org/papers/v18/17-468.html.

[Bre96] Leo Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996. doi:10.1007/BF00058655.

[Bre99] Leo Breiman. Pasting small votes for classification in large databases and on-line. *Machine Learning*, 36(1-2):85–103, 1999. doi:10.1023/A:1007563306331.

[Bre01] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001. doi:10.1023/A:1010933404324.

[BRS13] David Bacon, Rodric Rabbah, and Sunil Shukla. Fpga programming for the masses. *Communications of the ACM*, 56(4):56–63, 2013. doi:10.1145/2436696.2443836.

[BS] G. Biau and E. Scornet. A random forest guided tour. *Test*, 25(2):197–227. doi:10.1007/s11749-016-0481-7.

[BS96] Leo Breiman and Nong Shang. Born again trees. Technical report, University of California, Berkeley, 1996. URL: https://www.stat.berkeley.edu/~breiman/BAtrees.pdf.

[BS13] Pierre Baldi and Peter J Sadowski. Understanding dropout. In C.J. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc., 2013. URL: https://proceedings.neurips.cc/paper/2013/file/71f6278d140af599e06ad9bf1ba03cb0-Paper.pdf.

[BSW16] Gérard Biau, Erwan Scornet, and Johannes Welbl. Neural random forests. *CoRR*, abs/1604.07143, 2016. URL: http://arxiv.org/abs/1604.07143, arXiv:1604.07143.

[BT06] Peter Bartlett and Mikhail Traskin. Adaboost is consistent. In B. Schölkopf, J. Platt, and T. Hoffman, editors, *Advances in Neural Information Processing Systems*, volume 19. MIT Press, 2006. URL: https://proceedings.neurips.cc/paper/2006/file/b887d8d5e65ac4dec3934028fe23ad72-Paper.pdf.

[BV14] Stephen P. Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2014. URL: https://web.stanford.edu/%7Eboyd/cvxbook/, doi:10.1017/CBO9780511804441.

[BWL19] Yu Bai, Yu-Xiang Wang, and Edo Liberty. Proxquant: Quantized neural networks via proximal operators. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL: https://openreview.net/forum?id=HyzMyhCcK7.

[BWT05] Gavin Brown, Jeremy L. Wyatt, and Peter Tiño. Managing diversity in regression ensembles. *Journal of Machine Learning Research*, 6:1621–1650, 2005. URL: http://jmlr.org/papers/v6/brown05a.html.

[BYBM19] Joseph Bethge, Haojin Yang, Marvin Bornstein, and Christoph Meinel. Binarydensenet: Developing an architecture for binary neural networks. In *2019 IEEE/CVF International Conference on Computer Vision Workshops, ICCV Workshops 2019, Seoul, Korea (South), October 27-28, 2019*, pages 1951–1960. IEEE, 2019. doi:10.1109/ICCVW.2019.00244.

[CG16] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In Balaji Krishnapuram, Mohak Shah, Alexander J. Smola, Charu C. Aggarwal, Dou Shen, and Rajeev Rastogi, editors, *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, pages 785–794. ACM, 2016. doi:10.1145/2939672.2939785.

[CKL+06] Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary R. Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. In Bernhard Schölkopf, John C. Platt, and Thomas Hofmann, editors, *Advances in Neural Information Processing Systems 19, Proceedings of the Twentieth Annual Conference on Neural Information Processing Systems, Vancouver, British Columbia, Canada, December 4-7, 2006*, pages 281–288. MIT Press, 2006. URL: https://proceedings.neurips.cc/paper/2006/hash/77ee3bc58ce560b86c2b59363281e914-Abstract.html.

[CLL12] Shang-Tse Chen, Hsuan-Tien Lin, and Chi-Jen Lu. An online boosting algorithm with theoretical justifications. In *Proceedings of the 29th International Conference on Machine Learning, ICML 2012, Edinburgh, Scotland, UK, June 26 - July 1, 2012*, 2012. URL: http://icml.cc/2012/papers/538.pdf.

[CM99] Salvatore Cavalieri and Orazio Mirabella. A novel learning algorithm which improves the partial fault tolerance of multilayer neural networks. *Neural Networks*, 12(1):91–106, 1999. doi:10.1016/S0893-6080(98)00094-X.

[CMGS20] Tejalal Choudhary, Vipul Kumar Mishra, Anurag Goswami, and Jagannathan Sarangapani. A comprehensive survey on model compression and acceleration. *Artificial Intelligence Review*, 53(7):5113–5155, 2020. doi:10.1007/s10462-020-09816-7.

[CMS14] Corinna Cortes, Mehryar Mohri, and Umar Syed. Deep boosting. In *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014*, volume 32 of *JMLR Workshop and Conference Proceedings*, pages 1179–1187. JMLR.org, 2014. URL: http://proceedings.mlr.press/v32/cortesb14.html.

[COMC16] George D. C. Cavalcanti, Luiz S. Oliveira, Thiago J. M. Moura, and Guilherme V. Carvalho. Combining diversity measures for ensemble pruning. *Pattern Recognition Letters*, 74:38–45, 2016. doi:10.1016/j.patrec.2016.01.029.

[CP11] Patrick L. Combettes and Jean-Christophe Pesquet. Proximal splitting methods in signal processing. In Heinz H. Bauschke, Regina Sandra Burachik, Patrick L. Combettes, Veit Elser, D. Russell Luke, and Henry Wolkowicz, editors, *Fixed-Point Algorithms for Inverse Problems in Science and Engineering*, volume 49 of *Springer Optimization and Its Applications*, pages 185–212. Springer, 2011. doi:10.1007/978-1-4419-9569-8_10.

[CSH+21] Kuan-Hsun Chen, ChiaHui Su, Christian Hakert, Sebastian Buschjäger, Chao-Lin Lee, Jenq-Kuen Lee, Katharina Morik, and Jian-Jia Chen. Efficient realization of decision trees for real-time inference. *ACM Transactions on Embedded Computing Systems*, dec 2021. doi:10.1145/3508019.

[CZ01] Adele Cutler and Guohua Zhao. Pert-perfect random tree ensembles. *Computing Science and Statistics*, 33:490–497, 2001. URL: https://www.researchgate.net/profile/

Adele-Cutler/publication/268424569_PERT-perfect_random_tree_ensembles/links/551940c00cf2d241f355ee7b/PERT-perfect-random-tree-ensembles.pdf.

[CZZ+20] Yanjiao Chen, Baolin Zheng, Zihan Zhang, Qian Wang, Chao Shen, and Qian Zhang. Deep learning on mobile and embedded devices: State-of-the-art, challenges, and future directions. *ACM Computing Surveys (CSUR)*, 53(4):84:1–84:37, 2020. doi:10.1145/3398209.

[DCLT19] Jacob Devlin, Ming Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. *NAACL HLT 2019 - 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies - Proceedings of the Conference*, 1(Mlm):4171–4186, 2019. URL: https://arxiv.org/pdf/1810.04805.pdf, arXiv:1810.04805.

[dCSdB20] Silas Garrido Teixeira de Carvalho Santos and Roberto Souto Maior de Barros. Online adaboost-based methods for multiclass problems. *Artificial Intelligence Review*, 53(2):1293–1322, 2020. doi:10.1007/s10462-019-09696-6.

[Dem06] Janez Demšar. Statistical comparisons of classifiers over multiple data sets. *Journal of Machine Learning Research*, 7:1–30, 2006. URL: http://jmlr.org/papers/v7/demsar06a.html.

[DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. URL: https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf, doi:10.1145/1327452.1327492.

[DG17] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017. URL: http://archive.ics.uci.edu/ml.

[DH00] Pedro M. Domingos and Geoff Hulten. Mining high-speed data streams. In Raghu Ramakrishnan, Salvatore J. Stolfo, Roberto J. Bayardo, and Ismail Parsa, editors, *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining, Boston, MA, USA, August 20-23, 2000*, pages 71–80. ACM, 2000. doi:10.1145/347090.347107.

[DK21] James Diffenderfer and Bhavya Kailkhura. Multi-prize lottery ticket hypothesis: Finding accurate binary neural networks by pruning A randomly weighted network. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. URL: https://openreview.net/forum?id=U_mat0b9iv.

[DKLM05] Frederik Michel Dekking, Cornelis Kraaikamp, Hendrik Paul Lopuhaä, and Ludolf Erwin Meester. *A Modern Introduction to Probability and Statistics: Understanding why and how*. Springer, 2005. doi:10.1007/1-84628-168-7.

[dLBH14] Sara del Río, Victoria López, José Manuel Benítez, and Francisco Herrera. On the use of mapreduce for imbalanced big data using random forest. *Information Sciences*, 285:112–137, 2014. Processing and Mining Complex Data Streams. URL: https://www.sciencedirect.com/science/article/pii/S0020025514003272, doi:https://doi.org/10.1016/j.ins.2014.03.043.

[DMD19] Laurens Devos, Wannes Meert, and Jesse Davis. Fast gradient boosting decision trees with bit-level data structures. In Ulf Brefeld, Élisa Fromont, Andreas Hotho, Arno J. Knobbe, Marloes H. Maathuis, and Céline Robardet, editors, *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2019, Würzburg, Germany, September 16-20, 2019, Proceedings, Part I*, volume 11906 of *Lecture Notes in Computer Science*, pages 590–606. Springer, 2019. doi:10.1007/978-3-030-46150-8_35.

[DMS19] Nikita Dvornik, Julien Mairal, and Cordelia Schmid. Diversity with cooperation: Ensemble methods for few-shot classification. In *2019 IEEE/CVF International Conference on Computer Vision, ICCV 2019, Seoul, Korea (South), October 27 - November 2, 2019*, pages 3722–3730. IEEE, 2019. doi:10.1109/ICCV.2019.00382.

[Dom00] Pedro M. Domingos. A unified bias-variance decomposition for zero-one and squared loss. In Henry A. Kautz and Bruce W. Porter, editors, *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on on Innovative Applications of Artificial Intelligence, July 30 - August 3, 2000, Austin, Texas, USA*, pages 564–569. AAAI Press / The MIT Press, 2000. URL: http://www.aaai.org/Library/AAAI/2000/aaai00-086.php.

[DP96] Pedro M. Domingos and Michael J. Pazzani. Beyond independence: Conditions for the optimality of the simple bayesian classifier. In Lorenza Saitta, editor, *Machine Learning, Proceedings of the Thirteenth International Conference (ICML '96), Bari, Italy, July 3-6, 1996*, pages 105–112. Morgan Kaufmann, 1996. URL: https://homes.cs.washington.edu/~pedrod/papers/mlc96.pdf.

[DPQ17] Anuvabh Dutt, Denis Pellerin, and Georges Quénot. Coupled ensembles of neural networks. *Neurocomputing*, 396:346–357, sep 2017. URL: https://arxiv.org/abs/1709.06053, arXiv:1709.06053, doi:10.1016/j.neucom.2018.10.092.

[DS21] Emir Demirovic and Peter J. Stuckey. Optimal decision trees for nonlinear metrics. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pages 3733–3741. AAAI Press, 2021. URL: https://ojs.aaai.org/index.php/AAAI/article/view/16490.

[Edw73] Charles Henry Edwards. *Advanced calculus of several variables*. Academic Press, 1973. doi:10.1016/C2013-0-10610-1.

[EM97] Peter J. Edwards and Alan F. Murray. Penalty terms for fault tolerance. In *Proceedings of International Conference on Neural Networks (ICNN'97), Houston, TX, USA, June 9-12, 1997*, pages 943–947. IEEE, 1997. doi:10.1109/ICNN.1997.616152.

[EMGP12] Brian Van Essen, Chris Macaraeg, Maya B. Gokhale, and Ryan Prenger. Accelerating a random forest classifier: Multi-core, gp-gpu, or fpga? In *2012 IEEE 20th Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2012, 29 April - 1 May 2012, Toronto, Ontario, Canada*, pages 232–239. IEEE Computer Society, 2012. doi:10.1109/FCCM.2012.47.

[FC19] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL: https://openreview.net/forum?id=rJl-b3RcF7.

[FH17] Nicholas Frosst and Geoffrey E. Hinton. Distilling a neural network into a soft decision tree. In Tarek R. Besold and Oliver Kutz, editors, *Proceedings of the First International Workshop on Comprehensibility and Explanation in AI and ML 2017 co-located with 16th International Conference of the Italian Association for Artificial Intelligence (AI*IA 2017), Bari, Italy, November 16th and 17th, 2017*, volume 2071 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2017. URL: http://ceur-ws.org/Vol-2071/CExAIIA_2017_paper_3.pdf.

[FL93] András Faragó and Gábor Lugosi. Strong universal consistency of neural network classifiers. *IEEE Transactions on Information Theory*, 39(4):1146–1151, 1993. doi:10.1109/18.243433.

[FM82] Kunihiko Fukushima and Sei Miyake. Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In *Competition and cooperation in neural nets*, pages 267–285. Springer, 1982. doi:10.1007/978-3-642-46466-9_18.

[For55] George E. Forsythe. Relaxation methods. Technical report, California Univ. Los Angeles Numerical Analysis Research, 1955. URL: https://apps.dtic.mil/sti/pdfs/AD0059170.pdf.

[FS95] Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. In Paul M. B. Vitányi, editor, *Computational Learning Theory, Second European Conference, EuroCOLT '95, Barcelona, Spain, March 13-15, 1995, Proceedings*, volume 904 of *Lecture Notes in Computer Science*, pages 23–37. Springer, 1995. doi:10.1007/3-540-59119-2_166.

[FSL+94] V. P. Fomin, A. A. Stepanian, R. C. Lamb, D. A. Lewis, M. Punch, and T. C. Weekes. New methods of atmospheric Cherenkov imaging for gamma-ray astronomy. I. The false source method. *Astroparticle Physics*, 2(2):137–150, 1994. doi:10.1016/0927-6505(94)90036-1.

[FW60] G. W. Forsythe and Wolfgang R. Wasow. *Finite-Difference Methods for Partial Differential Equations*. Applied Mathematical Series. John Wiley & Sons, New York, 1960.

[GAGN15] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In Francis R. Bach and David M. Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, volume 37 of *JMLR Workshop and Conference Proceedings*, pages 1737–1746. JMLR.org, 2015. URL: http://proceedings.mlr.press/v37/gupta15.html.

[Gau23] Carl Friedrich Gauss. Briefwechsel zwischen c.f. gauss und c. l. gerling, 1823. Brief Nr 163, 26. Dezember 1823. URL: https://gdz.sub.uni-goettingen.de/id/PPN335994989?tify={%22pages%22:[11],%22panX%22:0.422,%22panY%22:0.742,%22view%22:%22export%22,%22zoom%22:0.427}.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[GBD] Stuart Geman, Elie Bienenstock, and René Doursat. Neural Networks and the Bias/Variance Dilemma. URL: http://www.dam.brown.edu/people/geman/Homepage/Essaysandideasaboutneurobiology/bias-variance.pdf, doi:10.1162/neco.1992.4.1.1.

[GBEB17] Heitor Murilo Gomes, Jean Paul Barddal, Fabrício Enembreck, and Albert Bifet. A survey on ensemble learning for data stream classification. *ACM Compututing Surveys*, 50(2):23:1–23:36, 2017. doi:10.1145/3054925.

[GBR+17] Heitor Murilo Gomes, Albert Bifet, Jesse Read, Jean Paul Barddal, Fabrício Enembreck, Bernhard Pfahringer, Geoff Holmes, and Talel Abdessalem. Adaptive random forests for evolving data stream classification. *Machine Learning*, 106(9-10):1469–1495, 2017. doi:10.1007/s10994-017-5642-8.

[GEW06] Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Machine learning*, 63(1):3–42, 2006. doi:10.1007/s10994-006-6226-1.

[GG16] Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In Maria-Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 1050–1059. JMLR.org, 2016. URL: http://proceedings.mlr.press/v48/gal16.html.

[GI18] Fabian Gieseke and Christian Igel. Training big random forests with little resources. In Yike Guo and Faisal Farooq, editors, *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018*, pages 1445–1454. ACM, 2018. doi:10.1145/3219819.3220124.

[GKD+21] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference. *CoRR*, abs/2103.13630, 2021. URL: https://arxiv.org/abs/2103.13630, arXiv:2103.13630.

[GLL+15] Pascal Germain, Alexandre Lacasse, François Laviolette, Mario Marchand, and Jean-Francis Roy. Risk bounds for the majority vote: from a pac-bayesian analysis to a learning algorithm. *Journal of Machine Learning Research*, 16:787–860, 2015. URL: http://dl.acm.org/citation.cfm?id=2831140.

[GLL+18] Huaping Guo, Hongbing Liu, Ran Li, Chang-an Wu, Yibo Guo, and Mingliang Xu. Margin & diversity based ordering ensemble pruning. *Neurocomputing*, 275:237–246, 2018. doi:10.1016/j.neucom.2017.06.052.

[Gow19] Robert M. Gower. Convergence theorems for gradient descent, 2019. https://gowerrobert.github.io/pdf/M2_statistique_optimisation/grad_conv.pdf, accessed June 2022. URL: https://gowerrobert.github.io/pdf/M2_statistique_optimisation/grad_conv.pdf.

[GRB19] Heitor Murilo Gomes, Jesse Read, and Albert Bifet. Streaming random patches for evolving data stream classification. In Jianyong Wang, Kyuseok Shim, and Xindong Wu, editors, *2019 IEEE International Conference on Data Mining, ICDM 2019, Beijing, China, November 8-11, 2019*, pages 240–249. IEEE, 2019. doi:10.1109/ICDM.2019.00034.

[GRF00] Giorgio Giacinto, Fabio Roli, and Giorgio Fumera. Design of effective multiple classifier systems by clustering of classifiers. In *15th International Conference on Pattern Recognition, ICPR'00, Barcelona, Spain, September 3-8, 2000*, pages 2160–2163. IEEE Computer Society, 2000. doi:10.1109/ICPR.2000.906039.

[Guo18] Yunhui Guo. A survey on methods and theories of quantized neural networks. *CoRR*, abs/1808.04752, 2018. URL: http://arxiv.org/abs/1808.04752, arXiv:1808.04752.

[HB] Danny Hernandez and Tom B. Brown. Measuring the algorithmic efficiency of neural networks. abs/2005.04305. URL: https://arxiv.org/abs/2005.04305, arXiv:2005.04305.

[HBK+19] Tifenn Hirtzlin, Marc Bocquet, Jacques-Olivier Klein, Etienne Nowak, Elisa Vianello, Jean Michel Portal, and Damien Querlioz. Outstanding bit error tolerance of resistive ram-based binarized neural networks. In *International Conference on Artificial Intelligence Circuits and Systems, AICAS*, pages 288–292, 2019. doi:10.1109/AICAS.2019.8771544.

[HBM21] Philipp-Jan Honysz, Sebastian Buschjäger, and Katharina Morik. Gpu-accelerated optimizer-aware evaluation of submodular exemplar clustering. *CoRR*, abs/2101.08763, 2021. URL: https://arxiv.org/abs/2101.08763, arXiv:2101.08763.

[HCP11] T. Ryan Hoens, Nitesh V. Chawla, and Robi Polikar. Heuristic updatable weighted random subspaces for non-stationary environments. In Diane J. Cook, Jian Pei, Wei Wang, Osmar R. Zaïane, and Xindong Wu, editors, *11th IEEE International Conference on Data Mining, ICDM 2011, Vancouver, BC, Canada, December 11-14, 2011*, pages 241–250. IEEE Computer Society, 2011. doi:10.1109/ICDM.2011.75.

[HCS+16] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 4107–4115, 2016. URL: https://proceedings.neurips.cc/paper/2016/hash/d8330f857a17c53d217014ee776bfd50-Abstract.html.

[HD08] S. Hauck and A. DeHon. *Reconfigurable computing: the theory and practice of FPGA-based computation*. Morgan Kaufmann, 2008. URL: https://dl.acm.org/doi/10.5555/1564780.

[Hes97] Tom Heskes. Selecting weighting factors in logarithmic opinion pools. In Michael I. Jordan, Michael J. Kearns, and Sara A. Solla, editors, *Advances in Neural Information Processing Systems 10, [NIPS Conference, Denver, Colorado, USA, 1997]*, pages 266–272. The MIT Press, 1997. URL: http://papers.nips.cc/paper/1413-selecting-weighting-factors-in-logarithmic-opinion-pools.

[HH00] Jakob Vogdrup Hansen and Tom Heskes. General bias/variance decomposition with target independent variance of error functions derived from the exponential family of distributions. In *15th International Conference on Pattern Recognition, ICPR'00, Barcelona, Spain, September 3-8, 2000*, pages 2207–2210. IEEE Computer Society, 2000. doi:10.1109/ICPR.2000.906049.

[HKC+98] D. Heck, J. Knapp, J. N. Capdevielle, G. Schatz, and T. Thouw. CORSIKA: a Monte Carlo code to simulate extensive air showers. Technical report, Forschungszentrum Karlsruhe GmbH, Karlsruhe (Germany), February 1998. URL: https://publikationen.bibliothek.kit.edu/270043064/3813660.

[HKP05] Geoffrey Holmes, Richard Kirkby, and Bernhard Pfahringer. Stress-testing hoeffding trees. In Alípio Jorge, Luís Torgo, Pavel Brazdil, Rui Camacho, and João Gama, editors, *Knowledge Discovery in Databases: PKDD 2005, 9th European Conference on Principles and Practice of Knowledge Discovery in Databases, Porto, Portugal, October 3-7, 2005, Proceedings*, volume 3721 of *Lecture Notes in Computer Science*, pages 495–502. Springer, 2005. doi:10.1007/11564126_50.

[HLP+17] Gao Huang, Yixuan Li, Geoff Pleiss, Zhuang Liu, John E. Hopcroft, and Kilian Q. Weinberger. Snapshot ensembles: Train 1, get M for free. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL: https://openreview.net/forum?id=BJYwwY9ll.

[HLS20] Sébastien Henwood, François Leduc-Primeau, and Yvon Savaria. Layer-wise noise maximisation to train low-energy deep neural networks. In *2nd IEEE International Conference on Artificial Intelligence Circuits and Systems, AICAS 2020, Genova, Italy, August 31 - September 2, 2020*, pages 271–275. IEEE, 2020. doi:10.1109/AICAS48895.2020.9073854.

[Ho98] Tin Kam Ho. The random subspace method for constructing decision forests. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(8):832–844, 1998. doi:10.1109/34.709601.

[Hoc98] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *Int. J. Uncertain. Fuzziness Knowl. Based Syst.*, 6(2):107–116, 1998. doi:10.1142/S0218488598000094.

[Hoe63] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963. URL: http://www.jstor.org/stable/2282952.

[Hor91] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257, 1991. doi:10.1016/0893-6080(91)90009-T.

[Hor14] Mark Horowitz. 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE International Conference on Solid-State Circuits Conference, ISSCC 2014, Digest of Technical Papers, San Francisco, CA, USA, February 9-13, 2014*, pages 10–14. IEEE, 2014. doi:10.1109/ISSCC.2014.6757323.

[HP11] J. Hennessy and D. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011. URL: https://dl.acm.org/doi/10.5555/1999263.

[HPK+19] Tifenn Hirtzlin, Bogdan Penkovsky, Jacques-Olivier Klein, Nicolas Locatelli, Adrien F. Vincent, Marc Bocquet, Jean Michel Portal, and Damien Querlioz. Implementing binarized neural networks with magnetoresistive RAM without error correction. *arXiv:1908.04085*, 2019. URL: http://arxiv.org/abs/1908.04085, arXiv:1908.04085.

[HR76] Laurent Hyafil and Ronald L. Rivest. Constructing optimal binary decision trees is np-complete. *Inf. Process. Lett.*, 5(1):15–17, 1976. doi:10.1016/0020-0190(76)90095-8.

[HR18] Boris Hanin and David Rolnick. How to start training: The effect of initialization and architecture. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 569–579, 2018. URL: https://proceedings.neurips.cc/paper/2018/hash/d81f9c1be2e08964bf9f24b15f0e4900-Abstract.html.

[HSBM21] Philipp-Jan Honysz, Alexander Schulze-Struchtrup, Sebastian Buschjäger, and Katharina Morik. Providing meaningful data summarizations using examplar-based clustering in industry 4.0. *CoRR*, abs/2105.12026, 2021. URL: https://arxiv.org/abs/2105.12026, arXiv:2105.12026.

[HSD01] Geoff Hulten, Laurie Spencer, and Pedro M. Domingos. Mining time-changing data streams. In Doheon Lee, Mario Schkolnick, Foster J.

Provost, and Ramakrishnan Srikant, editors, *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining, San Francisco, CA, USA, August 26-29, 2001*, pages 97–106. ACM, 2001. doi:10.1145/502512.502529.

[HSV+17] Hanzhang Hu, Wen Sun, Arun Venkatraman, Martial Hebert, and J. Andrew Bagnell. Gradient boosting on stochastic data streams. In Aarti Singh and Xiaojin (Jerry) Zhu, editors, *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, AISTATS 2017, 20-22 April 2017, Fort Lauderdale, FL, USA*, volume 54 of *Proceedings of Machine Learning Research*, pages 595–603. PMLR, 2017. URL: http://proceedings.mlr.press/v54/hu17a.html.

[HWG+19] Koen Helwegen, James Widdicombe, Lukas Geiger, Zechun Liu, Kwang-Ting Cheng, and Roeland Nusselder. Latent weights do not exist: Rethinking binarized neural network optimization. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 7531–7542, 2019. URL: https://proceedings.neurips.cc/paper/2019/hash/9ca8c9b0996bbf05ae7753d34667a6fd-Abstract.html.

[HYC+19] Christian Hakert, Mikail Yayla, Kuan-Hsun Chen, Georg von der Brüggen, Jian-Jia Chen, Sebastian Buschjäger, Katharina Morik, Paul R. Genssler, Lars Bauer, Hussam Amrouch, and Jörg Henkel. Stack usage analysis for efficient wear leveling in non-volatile main memory systems. In *1st ACM/IEEE Workshop on Machine Learning for CAD (MLCAD)*, 2019. URL: https://ieeexplore.ieee.org/abstract/document/9142113.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770–778. IEEE Computer Society, 2016. doi:10.1109/CVPR.2016.90.

[IA21] Ozan Irsoy and Ethem Alpaydin. Dropout regularization in hierarchical mixture of experts. *Neurocomputing*, 419:148–156, 2021. doi:10.1016/j.neucom.2020.08.052.

[IEE19] IEEE standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019. doi:10.1109/IEEESTD.2019.8766229.

[Int18] Intel. Intel neural compute stick 2. Technical report, Intel, 2018. https://www.intel.com/content/dam/support/us/en/documents/boardsandkits/neural-compute-sticks/NCS2_Product-Brief-English.pdf.

[IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Francis R. Bach and David M. Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, volume 37 of *JMLR Workshop and Conference Proceedings*, pages 448–456. JMLR.org, 2015. URL: http://proceedings.mlr.press/v37/ioffe15.html.

[Jam03] Gareth M. James. Variance and bias for general loss functions. *Machine Learning*, 51(2):115–135, May 2003. doi:10.1023/A:1022899518027.

[JF20] Zheming Jin and Hal Finkel. Population count on intel® cpu, GPU and FPGA. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2020, New Orleans, LA, USA, May 18-22, 2020*, pages 432–439. IEEE, 2020. doi:10.1109/IPDPSW50202.2020.00081.

[JKC+18] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew G. Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 2704–2713. Computer Vision Foundation / IEEE Computer Society, 2018. URL: http://openaccess.thecvf.com/content_cvpr_2018/html/Jacob_Quantization_and_Training_CVPR_2018_paper.html, doi:10.1109/CVPR.2018.00286.

[JL95] George H. John and Pat Langley. Estimating continuous distributions in bayesian classifiers. In Philippe Besnard and Steve Hanks, editors, *UAI '95: Proceedings of the Eleventh Annual Conference on Uncertainty in Artificial Intelligence, Montreal, Quebec, Canada, August 18-20, 1995*, pages 338–345. Morgan Kaufmann, 1995. URL: https://dslpitt.org/uai/displayArticleDetails.jsp?mmnu=1&smnu=2&article_id=450&proceeding_id=11.

[JLFW17] Zhengshen Jiang, Hongzhi Liu, Bin Fu, and Zhonghai Wu. Generalized ambiguity decompositions for classification with applications in active learning and unsupervised ensemble pruning. In Satinder Singh and Shaul Markovitch, editors, *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, pages 2073–2079. AAAI Press, 2017. URL: http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14475.

[JNH15] Wenhao Jiang, Feiping Nie, and Heng Huang. Robust dictionary learning with capped l1-norm. In Qiang Yang and Michael J. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 3590–3596. AAAI Press, 2015. URL: http://ijcai.org/Abstract/15/505.

[JYP+17] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden,

Al Borchers, Rick Boyle, Pierre Luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. *Proceedings - International Symposium on Computer Architecture*, Part F128643:1–12, 2017. arXiv:1704.04760, doi:10.1145/3079856.3080246.

[KB15] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL: http://arxiv.org/abs/1412.6980.

[KBCK13] Anastasios Kyrillidis, Stephen Becker, Volkan Cevher, and Christoph Koch. Sparse projections onto the simplex. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, volume 28 of *JMLR Workshop and Conference Proceedings*, pages 235–243. JMLR.org, 2013. URL: http://proceedings.mlr.press/v28/kyrillidis13.html.

[KCS+10] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. FAST: fast architecture sensitive tree search on modern cpus and gpus. In Ahmed K. Elmagarmid and Divyakant Agrawal, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 339–350. ACM, 2010. doi:10.1145/1807167.1807206.

[Kel60] Henry J. Kelley. Gradient theory of optimal flight paths. *Ars Journal*, 30(10):947–954, 1960. URL: https://www.gwern.net/docs/statistics/decision/1960-kelley.pdf.

[KFCB15] Peter Kontschieder, Madalina Fiterau, Antonio Criminisi, and Samuel Rota Bulò. Deep neural decision forests. In *2015 IEEE International Conference on Computer Vision, ICCV 2015, Santiago, Chile, December 7-13, 2015*, pages 1467–1475. IEEE Computer Society, 2015. doi:10.1109/ICCV.2015.172.

[KGV17] Ashish Kumar, Saurabh Goyal, and Manik Varma. Resource-efficient machine learning in 2 KB RAM for the internet of things. In Doina Precup

and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 1935–1944. PMLR, 2017. URL: http://proceedings.mlr.press/v70/kumar17a.html.

[KKB17] Kenji Kawaguchi, Leslie Pack Kaelbling, and Yoshua Bengio. Generalization in deep learning. *CoRR*, abs/1710.05468, 2017. URL: http://arxiv.org/abs/1710.05468, arXiv:1710.05468.

[Kle13] A. Klenke. *Wahrscheinlichkeitstheorie*. Masterclass. Springer Berlin Heidelberg, 2013. URL: https://books.google.de/books?id=K9EoBAAAQBAJ.

[KM96] Michael J. Kearns and Yishay Mansour. On the boosting ability of top-down decision tree learning algorithms. In Gary L. Miller, editor, *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 459–468. ACM, 1996. doi:10.1145/237814.237994.

[KM05] Jeremy Z. Kolter and Marcus A. Maloof. Using additive expert ensembles to cope with concept drift. In Luc De Raedt and Stefan Wrobel, editors, *Machine Learning, Proceedings of the Twenty-Second International Conference (ICML 2005), Bonn, Germany, August 7-11, 2005*, volume 119 of *ACM International Conference Proceeding Series*, pages 449–456. ACM, 2005. doi:10.1145/1102351.1102408.

[KMF+17] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 3146–3154, 2017. URL: https://proceedings.neurips.cc/paper/2017/hash/6449f44a102fde848669bdd9eb6b76fa-Abstract.html.

[KMG+17] Bartosz Krawczyk, Leandro L. Minku, João Gama, Jerzy Stefanowski, and Michal Wozniak. Ensemble learning for data stream analysis: A survey. *Inf. Fusion*, 37:132–156, 2017. doi:10.1016/j.inffus.2017.02.004.

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, Boston, third edition, 1997. URL: https://dl.acm.org/doi/10.5555/270146.

[Kön13] Konrad Königsberger. *Analysis 2*. Springer-Lehrbuch. Springer, 2013. URL: https://link.springer.com/book/10.1007/978-3-662-05699-8, doi:10.1007/978-3-662-05699-8.

[Kos21] Simon Koschel. Vergleich einer einheitlichen Implementierung von QuickScorer und RapidScorer mit OpenMP. Bachelor's Thesis TU Dortmund University, 2021. URL: https://www-ai.cs.tu-dortmund.de/BachelorArbeit_SimonKoschel.pdf?self=%24gjww14g0sg&part=data.

[Kot13] Sotiris B. Kotsiantis. Decision trees: a recent overview. *Artificial Intelligence Review*, 39(4):261–283, 2013. doi:10.1007/s10462-011-9272-4.

[KOY+19] Skanda Koppula, Lois Orosa, Abdullah Giray Yaglikçi, Roknoddin Azizi, Taha Shahroodi, Konstantinos Kanellopoulos, and Onur Mutlu. EDEN: enabling energy-efficient, high-performance deep neural network inference using approximate DRAM. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*, pages 166–181. ACM, 2019. doi:10.1145/3352460.3358280.

[Kri09] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009. http://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf. URL: http://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

[KS15] Abhishek Kumar and Vikas Sindhwani. Near-separable non-negative matrix factorization with l1 and bregman loss functions. In *Proceedings of the 2015 SIAM International Conference on Data Mining*, pages 343–351. SIAM, 2015. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611974010.39, arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9781611974010.39, doi:10.1137/1.9781611974010.39.

[KS17] Bangalore Ravi Kiran and Jean Serra. Cost-complexity pruning of random forests. In Jesús Angulo, Santiago Velasco-Forero, and Fernand Meyer, editors, *Mathematical Morphology and Its Applications to Signal and Image Processing - 13th International Symposium, ISMM 2017, Fontainebleau, France, May 15-17, 2017, Proceedings*, volume 10225 of *Lecture Notes in Computer Science*, pages 222–232, 2017. doi:10.1007/978-3-319-57240-6_18.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In Peter L. Bartlett, Fernando C. N. Pereira, Christopher J. C. Burges, Léon Bottou, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, pages 1106–1114, 2012. URL: https://proceedings.neurips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html.

[Kub89] Miroslav Kubat. Floating approximation in time-varying knowledge bases. *Pattern recognition letters*, 10(4):223–227, 1989. doi:10.1016/0167-8655(89)90092-5.

[KV94] Anders Krogh and Jesper Vedelsby. Neural network ensembles, cross validation, and active learning. In Gerald Tesauro, David S. Touretzky, and Todd K. Leen, editors, *Advances in Neural Information Processing Systems 7, [NIPS Conference, Denver, Colorado, USA, 1994]*, pages 231–238. MIT Press, 1994. URL: http://papers.nips.cc/paper/1001-neural-network-ensembles-cross-validation-and-active-learning.

[LBD+89] Yann LeCun, Bernhard E. Boser, John S. Denker, Donnie Henderson, Richard E. Howard, Wayne E. Hubbard, and Lawrence D. Jackel. Handwritten digit recognition with a back-propagation network. In David S. Touretzky, editor, *Advances in Neural Information Processing Systems 2, [NIPS Conference, Denver, Colorado, USA, November 27-30, 1989]*, pages 396–404. Morgan Kaufmann, 1989. URL: http://papers.nips.cc/paper/293-handwritten-digit-recognition-with-a-back-propagation-network.

[LC86] Yann Le Cun. Learning process in an asymmetric threshold network. In E. Bienenstock, F. Fogelman Soulié, and G. Weisbuch, editors, *Disordered Systems and Biological Organization*, pages 233–240, Berlin, Heidelberg, 1986. Springer Berlin Heidelberg. URL: https://link.springer.com/chapter/10.1007/978-3-642-82657-3_24.

[LCW+18] Jundong Li, Kewei Cheng, Suhang Wang, Fred Morstatter, Robert P. Trevino, Jiliang Tang, and Huan Liu. Feature selection: A data perspective. *ACM Computing Surveys*, 50(6):94:1–94:45, 2018. doi:10.1145/3136625.

[LHF05] Niels Landwehr, Mark A. Hall, and Eibe Frank. Logistic model trees. *Machine Learning*, 59(1-2):161–205, 2005. doi:10.1007/s10994-005-0466-3.

[Lim14] Manuel Lima. *The Book of Trees – Visualizing Branches of Knowledge*. Princeton Architectural Press, New York, 2014.

[LKD+17] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL: https://openreview.net/forum?id=rJqFGTslg.

[LLM20] Jean-Samuel Leboeuf, Frédéric Leblanc, and Mario Marchand. Decision trees as partitioning machines to characterize their generalization properties. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL: https://proceedings.neurips.cc/paper/2020/hash/d2a10b0bd670e442b1d3caa3fbf9e695-Abstract.html.

[LLN+19] Francesco Lettich, Claudio Lucchese, Franco Maria Nardini, Salvatore Orlando, Raffaele Perego, Nicola Tonellotto, and Rossano Venturini. Parallel traversal of large ensembles of decision trees. *IEEE Transactions on Parallel and Distributed Systems*, 30(9):2075–2089, 2019. doi:10.1109/TPDS.2018.2860982.

[LM83] Ti-Pei Li and Yu-Qian Ma. Analysis methods for results in gamma-ray astronomy. *The Astrophysical Journal*, 272, 9 1983. doi:10.1086/161295.

[LN13] Po-Ling Loh and Sebastian Nowozin. Faster hoeffding racing: Bernstein races via jackknife estimates. In Sanjay Jain, Rémi Munos, Frank Stephan, and Thomas Zeugmann, editors, *Algorithmic Learning Theory -*

24th International Conference, ALT 2013, Singapore, October 6-9, 2013. Proceedings, volume 8139 of *Lecture Notes in Computer Science*, pages 203–217. Springer, 2013. doi:10.1007/978-3-642-40935-6_15.

[LNO+15] Claudio Lucchese, Franco Maria Nardini, Salvatore Orlando, Raffaele Perego, Nicola Tonellotto, and Rossano Venturini. Quickscorer: A fast algorithm to rank documents with additive ensembles of regression trees. In Ricardo Baeza-Yates, Mounia Lalmas, Alistair Moffat, and Berthier A. Ribeiro-Neto, editors, *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval, Santiago, Chile, August 9-13, 2015*, pages 73–82. ACM, 2015. doi:10.1145/2766462.2767733.

[LNO+16] Claudio Lucchese, Franco Maria Nardini, Salvatore Orlando, Raffaele Perego, Nicola Tonellotto, and Rossano Venturini. Exploiting CPU SIMD extensions to speed-up document scoring with tree ensembles. In Raffaele Perego, Fabrizio Sebastiani, Javed A. Aslam, Ian Ruthven, and Justin Zobel, editors, *Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval, SIGIR 2016, Pisa, Italy, July 17-21, 2016*, pages 833–836. ACM, 2016. doi:10.1145/2911451.2914758.

[LNO+18] Claudio Lucchese, Franco Maria Nardini, Salvatore Orlando, Raffaele Perego, Fabrizio Silvestri, and Salvatore Trani. X-cleaver: Learning ranking ensembles by growing and pruning trees. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 9(6):62:1–62:26, 2018. doi:10.1145/3205453.

[LO01] A. Lazarevic and Z. Obradovic. Effective pruning of neural network classifier ensembles. In *IJCNN'01. International Joint Conference on Neural Networks. Proceedings (Cat. No.01CH37222)*, volume 2, pages 796–801, 2001. doi:10.1109/IJCNN.2001.939461.

[LO02] Aleksandar Lazarevic and Zoran Obradovic. Boosting algorithms for parallel and distributed learning. *Distributed Parallel Databases*, 11(2):203–229, 2002. doi:10.1023/A:1013992203485.

[Loh14] Wei-Yin Loh. Fifty years of classification and regression trees. *International Statistical Review / Revue Internationale de Statistique*, 82(3):329–348, 2014. URL: http://www.jstor.org/stable/43298996.

[LPB17] Balaji Lakshminarayanan, Alexander Pritzel, and Charles Blundell. Simple and scalable predictive uncertainty estimation using deep ensembles. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 6402–6413, 2017. URL: https://proceedings.neurips.cc/paper/2017/hash/9ef2ed4b7fd2c810847ffa5fa85bce38-Abstract.html.

[LPC+15] Stefan Lee, Senthil Purushwalkam, Michael Cogswell, David J. Crandall, and Dhruv Batra. Why M heads are better than one: Training a diverse

ensemble of deep networks. *CoRR*, abs/1511.06314, 2015. URL: http://arxiv.org/abs/1511.06314, arXiv:1511.06314.

[LPC+16] Stefan Lee, Senthil Purushwalkam, Michael Cogswell, Viresh Ranjan, David J. Crandall, and Dhruv Batra. Stochastic multiple choice learning for training diverse deep ensembles. In Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 2119–2127, 2016. URL: https://proceedings.neurips.cc/paper/2016/hash/20d135f0f28185b84a4cf7aa51f29500-Abstract.html.

[LPW+17] Zhou Lu, Hongming Pu, Feicheng Wang, Zhiqiang Hu, and Liwei Wang. The expressive power of neural networks: A view from the width. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 6231–6239, 2017. URL: https://proceedings.neurips.cc/paper/2017/hash/32cbf687880eb1674a07bf717761dd3a-Abstract.html.

[LSJR16] Jason D. Lee, Max Simchowitz, Michael I. Jordan, and Benjamin Recht. Gradient descent only converges to minimizers. In Vitaly Feldman, Alexander Rakhlin, and Ohad Shamir, editors, *Proceedings of the 29th Conference on Learning Theory, COLT 2016, New York, USA, June 23-26, 2016*, volume 49 of *JMLR Workshop and Conference Proceedings*, pages 1246–1257. JMLR.org, 2016. URL: http://proceedings.mlr.press/v49/lee16.html.

[LSL+21] Zechun Liu, Zhiqiang Shen, Shichao Li, Koen Helwegen, Dong Huang, and Kwang-Ting Cheng. How do adam and training strategies help bnns optimization. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 6936–6946. PMLR, 2021. URL: http://proceedings.mlr.press/v139/liu21t.html.

[LSSC20] Zechun Liu, Zhiqiang Shen, Marios Savvides, and Kwang-Ting Cheng. Reactnet: Towards precise binary neural network with generalized activation functions. In Andrea Vedaldi, Horst Bischof, Thomas Brox, and Jan-Michael Frahm, editors, *Computer Vision - ECCV 2020 - 16th European Conference, Glasgow, UK, August 23-28, 2020, Proceedings, Part XIV*, volume 12359 of *Lecture Notes in Computer Science*, pages 143–159. Springer, 2020. doi:10.1007/978-3-030-58568-6_9.

[LV04] Gábor Lugosi and Nicolas Vayatis. On the bayes-risk consistency of regularized boosting methods. *The Annals of statistics*, 32(1):30–55, 2004.

[LVV+20] Sam Leroux, Bert Vankeirsbilck, Tim Verbelen, Pieter Simoens, and Bart Dhoedt. Training binary neural networks with knowledge transfer. *Neurocomputing*, 396:534–541, 2020. doi:10.1016/j.neucom.2018.09.103.

[LWZB10] Zhenyu Lu, Xindong Wu, Xingquan Zhu, and Josh C. Bongard. Ensemble pruning via individual contribution ordering. In Bharat Rao, Balaji Krishnapuram, Andrew Tomkins, and Qiang Yang, editors, *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, July 25-28, 2010*, pages 871–880. ACM, 2010. doi:10.1145/1835804.1835914.

[LY99] Y. Liu and X. Yao. Ensemble learning via negative correlation. *Neural Networks*, 12(10):1399–1404, 1999. URL: https://www.cs.bham.ac.uk/{~}pxt/NC/ncl.pdf, doi:10.1016/S0893-6080(99)00073-8.

[LYZ12] Nan Li, Yang Yu, and Zhi-Hua Zhou. Diversity regularized ensemble pruning. In Peter A. Flach, Tijl De Bie, and Nello Cristianini, editors, *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2012, Bristol, UK, September 24-28, 2012. Proceedings, Part I*, volume 7523 of *Lecture Notes in Computer Science*, pages 330–345. Springer, 2012. doi:10.1007/978-3-642-33460-3_27.

[MAA+17] Sebastian Achim Mueller, J. Adam, Max Ludwig Ahnen, D. Baack, Matteo Balbo, Adrian Biland, M. Blank, Thomas Bretz, K Bruegge, Jens Buss, A. Dmytriiev, Daniela Dorner, Sabrina Einecke, Dominik Elsaesser, Christina Hempfling, Tina Herbst, Dorothee Hildebrand, L. Kortmann, L. Linhoff, Max Mahlke, Karl Mannheim, Dominik Neise, Andrii Neronov, Maximilian Noethe, J. Oberkirch, Aleksander Paravac, Felicitas Pauss, Wolfgang Rhode, B. Schleicher, F. Schulz, Amit Shukla, Vitalii Sliusar, Fabian Temme, Julia Thaele, and Roland Walter. Single Photon Extraction for FACT's SiPMs allows for Novel IACT Event Representation. *PoS*, ICRC2017:801, 2017. doi:10.22323/1.301.0801.

[Man97] Yishay Mansour. Pessimistic decision tree pruning based on tree size. In *Proceedings of the Fourteenth International Conference on Machine Learning*, pages 195–201. Morgan Kaufmann, 1997. URL: https://www.tau.ac.il/~mansour/papers/97-ml.ps.gz.

[Mar52] Harry Markowitz. The utility of wealth. *Journal of Political Economy*, 60(2):151–158, 1952. URL: http://www.jstor.org/stable/1825964.

[Mar11] Peter Marwedel. *Embedded System Design - Embedded Systems Foundations of Cyber-Physical Systems, Second Edition*. Embedded Systems. Springer, 2011. doi:10.1007/978-94-007-0257-8.

[May18] Michael May. Effizente bildverarbeitung hexagonaler strukturen mittels deep convolutional neural networks. Master's thesis, 2018. URL: https://www-ai.cs.tu-dortmund.de/PublicPublicationFiles/may_2018a.pdf.

[MBB18] Katharina Morik, Christian Bockermann, and Sebastian Buschjäger. Big data science. *Künstliche Intelligenz*, 32(1):27–36, 2018. doi:10.1007/s13218-017-0522-8.

[MBBF99] Llew Mason, Jonathan Baxter, Peter L. Bartlett, and Marcus R. Frean. Boosting algorithms as gradient descent. In Sara A. Solla, Todd K. Leen, and Klaus-Robert Müller, editors, *Advances in*

Neural Information Processing Systems 12, [NIPS Conference, Denver, Colorado, USA, November 29 - December 4, 1999], pages 512–518. The MIT Press, 1999. URL: http://papers.nips.cc/paper/1766-boosting-algorithms-as-gradient-descent.

[MD97] Dragos D. Margineantu and Thomas G. Dietterich. Pruning adaptive boosting. In Douglas H. Fisher, editor, *Proceedings of the Fourteenth International Conference on Machine Learning (ICML 1997), Nashville, Tennessee, USA, July 8-12, 1997*, pages 211–218. Morgan Kaufmann, 1997. URL: https://web.engr.oregonstate.edu/~tgd/publications/ml97-pruning-adaboost.ps.gz.

[MDP+11] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In Dekang Lin, Yuji Matsumoto, and Rada Mihalcea, editors, *The 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, Proceedings of the Conference, 19-24 June, 2011, Portland, Oregon, USA*, pages 142–150. The Association for Computer Linguistics, 2011. URL: https://aclanthology.org/P11-1015/.

[MHM21] Mehran Mirkhan, Maryam Amir Haeri, and Mohammad Reza Meybodi. Analytical Split Value Calculation for Numerical Attributes in Hoeffding Trees with Misclassification-Based Impurity. *Annals of Data Science*, 8(3):645–665, September 2021. URL: https://ideas.repec.org/a/spr/aodasc/v8y2021i3d10.1007_s40745-019-00225-4.html, doi:10.1007/s40745-019-00225-.

[MHS09] Gonzalo Martínez-Muñoz, Daniel Hernández-Lobato, and Alberto Suárez. An analysis of ensemble pruning techniques based on ordered aggregation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(2):245–259, 2009. doi:10.1109/TPAMI.2008.78.

[MK15] Pradeep Moorthy and Nachiket Kapre. Zedwulf: Power-performance tradeoffs of a 32-node zynq soc cluster. In *23rd IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2015, Vancouver, BC, Canada, May 2-6, 2015*, pages 68–75. IEEE Computer Society, 2015. doi:10.1109/FCCM.2015.37.

[MM93] Oded Maron and Andrew W. Moore. Hoeffding races: Accelerating model selection search for classification and function approximation. In Jack D. Cowan, Gerald Tesauro, and Joshua Alspector, editors, *Advances in Neural Information Processing Systems 6, [7th NIPS Conference, Denver, Colorado, USA, 1993]*, pages 59–66. Morgan Kaufmann, 1993. URL: tinyurl.com/bdjebn4c.

[MM97] Oded Maron and Andrew W. Moore. The racing algorithm: Model selection for lazy learners. *Artificial Intelligence Review*, 11(1-5):193–225, 1997. doi:10.1023/A:1006556606079.

[MM18] Asit K. Mishra and Debbie Marr. Apprentice: Using knowledge distillation techniques to improve low-precision network accuracy. In *6th*

International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings. OpenReview.net, 2018. URL: https://openreview.net/forum?id=B1ae1lZRb.

[MMS04] Gonzalo Martınez-Munoz and Alberto Suárez. Aggregation ordering in bagging. In *Proceedings of the International Conference on Artificial Intelligence and Applications (IASTED)*, pages 258–263, 2004. URL: http://arantxa.ii.uam.es/~gonzalo/publications/MarSua-aia2004_preprint.pdf.

[MP43] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943. URL: https://link.springer.com/article/10.1007/BF02478259.

[MPM19] Sascha Mücke, Nico Piatkowski, and Katharina Morik. Hardware accelerated learning at the edge. In Michael Kamp, Daniel Paurat, and Yamuna Krishnamurthy, editors, *Decentralized Machine Learning at the Edge*. Springer, 2019. URL: https://dmle.iais.fraunhofer.de/papers/muecke2019hardware.pdf.

[MRS13] Indraneel Mukherjee, Cynthia Rudin, and Robert E. Schapire. The rate of convergence of adaboost. *Journal of Machine Learning Research*, 14(1):2315–2347, 2013. URL: http://dl.acm.org/citation.cfm?id=2567735.

[MRVT+18] Mojtaba Masoudinejad, Aswin Karthik Ramachandran Venkatapathy, David Tondorf, Danny Heinrich, Robert Falkenberg, and Markus Buschhoff. Machine learning based indoor localisation using environmental data in phynetlab warehouse. In *Smart SysTech 2018; European Conference on Smart Objects, Systems and Technologies*, pages 1–8, 2018. URL: https://ieeexplore.ieee.org/document/8435922.

[MS63] James N. Morgan and John A. Sonquist. Problems in the analysis of survey data, and a proposal. *Journal of the American statistical association*, 58(302):415–434, 1963. URL: https://www.jstor.org/stable/2283276#metadata_info_tab_contents, doi:10.2307/2283276.

[MS95] Sreerama K. Murthy and Steven Salzberg. Decision tree induction: How effective is the greedy heuristic? In Usama M. Fayyad and Ramasamy Uthurusamy, editors, *Proceedings of the First International Conference on Knowledge Discovery and Data Mining (KDD-95), Montreal, Canada, August 20-21, 1995*, pages 222–227. AAAI Press, 1995. URL: http://www.aaai.org/Library/KDD/1995/kdd95-054.php.

[MS06] Gonzalo Martínez-Muñoz and Alberto Suárez. Pruning in ordered bagging ensembles. In William W. Cohen and Andrew W. Moore, editors, *Machine Learning, Proceedings of the Twenty-Third International Conference (ICML 2006), Pittsburgh, Pennsylvania, USA, June 25-29, 2006*, volume 148 of *ACM International Conference Proceeding Series*, pages 609–616. ACM, 2006. doi:10.1145/1143844.1143921.

[MS15] Zelda Mariet and Suvrit Sra. Fixed-point algorithms for learning determinantal point processes. In Francis R. Bach and David M. Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, volume 37 of *JMLR Workshop and Conference Proceedings*, pages 2389–2397. JMLR.org, 2015. URL: http://proceedings.mlr.press/v37/mariet15.html.

[MVS+19] Vojtech Mrazek, Zdenek Vasícek, Lukás Sekanina, Muhammad Abdullah Hanif, and Muhammad Shafique. ALWANN: automatic layer-wise approximation of deep neural network accelerators without retraining. In David Z. Pan, editor, *Proceedings of the International Conference on Computer-Aided Design, ICCAD 2019, Westminster, CO, USA, November 4-7, 2019*, pages 1–8. ACM, 2019. doi:10.1109/ICCAD45719.2019.8942068.

[MWH13] Joshua S. Monson, Michael J. Wirthlin, and Brad L. Hutchings. Implementing high-performance, low-power fpga-based optical flow accelerators in C. In *24th International Conference on Application-Specific Systems, Architectures and Processors, ASAP 2013, Washington, DC, USA, June 5-7, 2013*, pages 363–369. IEEE Computer Society, 2013. doi:10.1109/ASAP.2013.6567602.

[MWS18] Chaitanya Manapragada, Geoffrey I. Webb, and Mahsa Salehi. Extremely fast decision tree. In Yike Guo and Faisal Farooq, editors, *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018*, pages 1953–1962. ACM, 2018. doi:10.1145/3219819.3220005.

[NDB+19] Giang Nguyen, Stefan Dlugolinsky, Martin Bobák, Viet D. Tran, Álvaro López García, Ignacio Heredia, Peter Malík, and Ladislav Hluchý. Machine learning and deep learning frameworks and libraries for large-scale data mining: a survey. *Artificial Intelligence Review*, 52(1):77–124, 2019. doi:10.1007/s10462-018-09679-z.

[NJLS09] A. Nemirovski, A. Juditsky, G. Lan, and A. Shapiro. Robust stochastic approximation approach to stochastic programming. *SIAM Journal on Optimization*, 19(4):1574–1609, 2009. arXiv:https://doi.org/10.1137/070704277, doi:10.1137/070704277.

[NSL+15] Julie Nutini, Mark Schmidt, Issam H. Laradji, Michael P. Friedlander, and Hoyt A. Koepke. Coordinate descent converges faster with the gauss-southwell rule than random selection. In Francis R. Bach and David M. Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, volume 37 of *JMLR Workshop and Conference Proceedings*, pages 1632–1641. JMLR.org, 2015. URL: http://proceedings.mlr.press/v37/nutini15.html.

[NSY+20] Supun Nakandala, Karla Saur, Gyeong-In Yu, Konstantinos Karanasos, Carlo Curino, Markus Weimer, and Matteo Interlandi. A tensor compiler for unified machine learning prediction serving. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI*

2020, Virtual Event, November 4-6, 2020, pages 899–917. USENIX Association, 2020. URL: https://www.usenix.org/conference/osdi20/presentation/nakandala.

[Nvi18] Nvidia. Nvidia a100 tensor core gpu architecture. Technical report, Nvidia, 2018. URL: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.

[Nö17] Maximilian Nöthe. Improving the Angular Resolution of FACT Using Machine Learning. Technical report, Tu Dortmund University, Dortmund, 2017. URL: https://sfb876.tu-dortmund.de/PublicPublicationFiles/sfbgk_etal_2017b.pdf.

[OPB12] Thais Mayumi Oshiro, Pedro Santoro Perez, and José Augusto Baranauskas. How many trees in a random forest? In Petra Perner, editor, *Machine Learning and Data Mining in Pattern Recognition - 8th International Conference, MLDM 2012, Berlin, Germany, July 13-20, 2012. Proceedings*, volume 7376 of *Lecture Notes in Computer Science*, pages 154–168. Springer, 2012. doi:10.1007/978-3-642-31537-4_13.

[OPB16] Michael Opitz, Horst Possegger, and Horst Bischof. Efficient model averaging for deep neural networks. In Shang-Hong Lai, Vincent Lepetit, Ko Nishino, and Yoichi Sato, editors, *Computer Vision - ACCV 2016 - 13th Asian Conference on Computer Vision, Taipei, Taiwan, November 20-24, 2016, Revised Selected Papers, Part II*, volume 10112 of *Lecture Notes in Computer Science*, pages 205–220, 2016. doi:10.1007/978-3-319-54184-6_13.

[OR01] Nikunj C. Oza and Stuart Russell. Online bagging and boosting. In Thomas S. Richardson and Tommi S. Jaakkola, editors, *Proceedings of the Eighth International Workshop on Artificial Intelligence and Statistics, AISTATS 2001, Key West, Florida, USA, January 4-7, 2001*. Society for Artificial Intelligence and Statistics, 2001. URL: http://www.gatsby.ucl.ac.uk/aistats/aistats2001/files/oza149.ps.

[OS21] Sebastian Ordyniak and Stefan Szeider. Parameterized complexity of small decision tree learning. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pages 6454–6462. AAAI Press, 2021. URL: https://ojs.aaai.org/index.php/AAAI/article/view/16800.

[Oza05] Nikunj C. Oza. Online bagging and boosting. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, Waikoloa, Hawaii, USA, October 10-12, 2005*, pages 2340–2345. IEEE, 2005. doi:10.1109/ICSMC.2005.1571498.

[Paw82] Zdzislaw Pawlak. Rough sets. *International Journal of Computer & Information Sciences*, 11(5):341–356, 1982. doi:10.1007/BF01001956.

[PB08a] Markos Papadonikolakis and Christos-Savvas Bouganis. Efficient FPGA mapping of gilbert's algorithm for SVM training on large-scale classification problems. In *FPL 2008, International Conference on Field Programmable Logic and Applications, Heidelberg, Germany, 8-10 September 2008*, pages 385–390. IEEE, 2008. doi:10.1109/FPL.2008.4629968.

[PB08b] Markos Papadonikolakis and Christos-Savvas Bouganis. A scalable FPGA architecture for non-linear SVM training. In Tarek A. El-Ghazawi, Yao-Wen Chang, Juinn-Dar Huang, and Proshanta Saha, editors, *2008 International Conference on Field-Programmable Technology, FPT 2008, Taipei, Taiwan, December 7-10, 2008*, pages 337–340. IEEE, 2008. doi:10.1109/FPT.2008.4762412.

[PB14] Neal Parikh and Stephen P. Boyd. Proximal algorithms. *Found. Trends Optim.*, 1(3):127–239, 2014. doi:10.1561/2400000003.

[PCMM13] R. Prenger, B. Chen, T. Marlatt, and D. Merl. Fast map search for compact additive tree ensembles (cate). Technical report, Tech. rep., Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2013. URL: https://www.osti.gov/servlets/purl/1078539.

[PFM+12] G. Pasaoglu, D. Fiorello, A. Martino, G. Scarcella, A. Alemanno, A. Zubaryeva, and C. Thiel. *Driving and parking patterns of European car drivers - a mobility survey*. 2012. doi:10.2790/7028.

[PGM+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 8024–8035, 2019. URL: https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

[PGV+18] Liudmila Ostroumova Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, and Andrey Gulin. Catboost: unbiased boosting with categorical features. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 6639–6649, 2018. URL: https://proceedings.neurips.cc/paper/2018/hash/14491b756b3a51daac41c24863285549-Abstract.html.

[PHBB09] Biswanath Panda, Joshua Herbach, Sugato Basu, and Roberto J. Bayardo. PLANET: massively parallel learning of tree ensembles with mapreduce. *Proc. VLDB Endow.*, 2(2):1426–1437, 2009. URL: http://www.vldb.org/pvldb/vol2/vldb09-537.pdf, doi:10.14778/1687553.1687569.

[PHK07] Bernhard Pfahringer, Geoffrey Holmes, and Richard Kirkby. New options for hoeffding trees. In Mehmet A. Orgun and John Thornton, editors, *AI 2007: Advances in Artificial Intelligence, 20th Australian Joint Conference on Artificial Intelligence, Gold Coast, Australia, December 2-6, 2007, Proceedings*, volume 4830 of *Lecture Notes in Computer Science*, pages 90–99. Springer, 2007. doi:10.1007/978-3-540-76928-6_11.

[PHK08] Bernhard Pfahringer, Geoffrey Holmes, and Richard Kirkby. Handling numeric attributes in hoeffding trees. In Takashi Washio, Einoshin Suzuki, Kai Ming Ting, and Akihiro Inokuchi, editors, *Advances in Knowledge Discovery and Data Mining, 12th Pacific-Asia Conference, PAKDD 2008, Osaka, Japan, May 20-23, 2008 Proceedings*, volume 5012 of *Lecture Notes in Computer Science*, pages 296–307. Springer, 2008. doi:10.1007/978-3-540-68125-0_27.

[PJVR09] Raphael Pelossof, Michael Jones, Ilia Vovsha, and Cynthia Rudin. Online coordinate boosting. In *12th IEEE International Conference on Computer Vision Workshops, ICCV Workshops 2009, Kyoto, Japan, September 27 - October 4, 2009*, pages 1354–1361. IEEE Computer Society, 2009. doi:10.1109/ICCVW.2009.5457454.

[PLM16] Nico Piatkowski, Sangkyun Lee, and Katharina Morik. Integer undirected graphical models for resource-constrained systems. *Neurocomputing*, 173:9–23, 2016. doi:10.1016/j.neucom.2015.01.091.

[PN15] Andrei Patrascu and Ion Necoara. Efficient random coordinate descent algorithms for large-scale structured nonconvex optimization. *J. Glob. Optim.*, 61(1):19–46, 2015. doi:10.1007/s10898-014-0151-9.

[Pol64] B. T. Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964. URL: https://www.sciencedirect.com/science/article/pii/0041555364901375, doi:https://doi.org/10.1016/0041-5553(64)90137-5.

[PR12] Indranil Palit and Chandan K. Reddy. Scalable and parallel boosting with mapreduce. *IEEE Transactions on Knowledge and Data Engineering*, 24(10):1904–1916, 2012. doi:10.1109/TKDE.2011.208.

[PVG+11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake VanderPlas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, 12:2825–2830, 2011. URL: http://dl.acm.org/citation.cfm?id=2078195.

[Qia99] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural Networks*, 12(1):145–151, 1999. URL: https://www.sciencedirect.com/science/article/pii/S0893608098001166, doi:https://doi.org/10.1016/S0893-6080(98)00116-6.

[Qui86] J. Ross Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986. doi:10.1023/A:1022643204877.

[QZR+14] Xueheng Qiu, Le Zhang, Ye Ren, Ponnuthurai N. Suganthan, and Gehan A. J. Amaratunga. Ensemble deep learning for regression and time series forecasting. In *2014 IEEE Symposium on Computational Intelligence in Ensemble Learning, CIEL 2014, Orlando, FL, USA, December 9-12, 2014*, pages 21–26. IEEE, 2014. doi:10.1109/CIEL.2014.7015739.

[RCWS15] Shaoqing Ren, Xudong Cao, Yichen Wei, and Jian Sun. Global refinement of random forest. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, pages 723–730. IEEE Computer Society, 2015. doi:10.1109/CVPR.2015.7298672.

[RER+19] Tim Ruhe, Dominik Elsässer, Wolfgang Rhode, Maximilian Nöthe, and Kai Brügge. Cherenkov Telescope Ring-An Idea for World Wide Monitoring of the VHE Sky. In *Proceedings of the EPJ Web Conference*, 2019. doi:10.1051/epjconf/201920703002.

[RF10] H. L. Royden and P. Fitzpatrick. *Real Analysis*. Prentice Hall, 2010. URL: https://books.google.de/books?id=0Y5fAAAACAAJ.

[RG11] Parikshit Ram and Alexander G. Gray. Density estimation trees. In Chid Apté, Joydeep Ghosh, and Padhraic Smyth, editors, *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Diego, CA, USA, August 21-24, 2011*, pages 627–635. ACM, 2011. doi:10.1145/2020408.2020507.

[RHS04] Jesse A. Reichler, Harlan D. Harris, and Michael A. Savchenko. Online parallel boosting. In Deborah L. McGuinness and George Ferguson, editors, *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA*, pages 366–371. AAAI Press / The MIT Press, 2004. URL: http://www.aaai.org/Library/AAAI/2004/aaai04-059.php.

[RHW86] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986. doi:https://doi.org/10.1038/323533a0.

[RKK18] Sashank J. Reddi, Satyen Kale, and Sanjiv Kumar. On the convergence of adam and beyond. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. URL: https://openreview.net/forum?id=ryQu7f-RZ.

[Ros58] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958. doi:10.1037/h0042519.

[RPDJ13] Leszek Rutkowski, Lena Pietruczuk, Piotr Duda, and Maciej Jaworski. Decision trees for mining data streams based on the mcdiarmid's bound. *IEEE Transactions on Knowledge and Data Engineering*, 25(6):1272–1279, 2013. doi:10.1109/TKDE.2012.66.

[Rud16] Sebastian Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016. URL: http://arxiv.org/abs/1609.04747, arXiv:1609.04747.

[RVC+04] Lorenzo Rosasco, Ernesto De Vito, Andrea Caponnetto, Michele Piana, and Alessandro Verri. Are loss functions all the same? *Neural Comput.*, 16(5):1063–107, 2004. doi:10.1162/089976604773135104.

[RVV20] Lorenzo Rosasco, Silvia Villa, and Bang Công Vũ. Convergence of stochastic proximal gradient algorithm. *Applied Mathematics & Optimization*, 82(3):891–917, 12 2020. doi:10.1007/s00245-019-09617-7.

[RWA+16] Brandon Reagen, Paul N. Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David M. Brooks. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016*, pages 267–278. IEEE Computer Society, 2016. doi:10.1109/ISCA.2016.32.

[Rö17] Stefan Rötner. Deep learning on raw telescope data. Master's thesis, 2017. URL: https://www-ai.cs.tu-dortmund.de/FactDeepLearning.pdf?self=%24ffzcz7qww0&part=data.

[SBV15] Erwan Scornet, Gérard Biau, and Jean-Philippe Vert. Consistency of random forests. *The Annals of Statistics*, 43(4):1716 – 1741, 2015. doi:10.1214/15-AOS1321.

[SDP+15] Fareena Saqib, Aindrik Dutta, Jim Plusquellic, Philip Ortiz, and Marios S. Pattichis. Pipelined decision tree classification accelerator implementation in FPGA (DT-CAIF). *IEEE Transactions on Computers*, 64(1):280–285, 2015. doi:10.1109/TC.2013.204.

[SF12] Robert E. Schapire and Yoav Freund. *Boosting: Foundations and Algorithms*. The MIT Press, 2012. doi:mitpress/8291.001.0001.

[SGW+18] Wei Shen, Yilu Guo, Yan Wang, Kai Zhao, Bo Wang, and Alan Yuille. Deep Regression Forests for Age Estimation. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2018. arXiv:1712.07195, doi:10.1109/CVPR.2018.00245.

[SH21] Mohsen Shahhosseini and Guiping Hu. Improved weighted random forest for classification problems. In Tofigh Allahviranloo, Soheil Salahshour, and Nafiz Arica, editors, *Progress in Intelligent Decision Science*, pages 42–56, Cham, 2021. Springer International Publishing. URL: https://link.springer.com/chapter/10.1007/978-3-030-66501-2_4.

[Sha08] Toby Sharp. Implementing decision trees and forests on a GPU. In David A. Forsyth, Philip H. S. Torr, and Andrew Zisserman, editors, *Computer Vision - ECCV 2008, 10th European Conference on Computer Vision, Marseille, France, October 12-18, 2008, Proceedings, Part IV*, volume 5305 of *Lecture Notes in Computer Science*, pages 595–608. Springer, 2008. doi:10.1007/978-3-540-88693-8_44.

[SHK+14] Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, 2014. URL: http://dl.acm.org/citation.cfm?id=2670313.

[SHP19] Mohsen Shahhosseini, Guiping Hu, and Hieu Pham. Optimizing ensemble weights and hyperparameters of machine learning models for regression problems. *CoRR*, abs/1908.05287, 2019. URL: http://arxiv.org/abs/1908.05287, arXiv:1908.05287.

[Sim01] Daniel J. Simon. Distributed fault tolerance in optimal interpolative nets. *IEEE Transactions Neural Networks*, 12(6):1348–1357, 2001. doi:10.1109/72.963771.

[SK01] W. Nick Street and YongSeog Kim. A streaming ensemble algorithm (SEA) for large-scale classification. In Doheon Lee, Mario Schkolnick, Foster J. Provost, and Ramakrishnan Srikant, editors, *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining, San Francisco, CA, USA, August 26-29, 2001*, pages 377–382. ACM, 2001. doi:10.1145/502512.502568.

[SK05] Martin Scholz and Ralf Klinkenberg. An ensemble classifier for drifting concepts. In *Second International Workshop on Knowledge Discovery in Data Streams*, volume 6, pages 53–64. Porto, Portugal, 2005. URL: https://www.researchgate.net/profile/Ralf-Klinkenberg/publication/228622206_An_ensemble_classifier_for_drifting_concepts/links/0fcfd51498433608f9000000/An-ensemble-classifier-for-drifting-concepts.pdf.

[SK19] Connor Shorten and Taghi M. Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of big data*, 6:60, 2019. doi:10.1186/s40537-019-0197-0.

[SKS17] Charbel Sakr, Yongjune Kim, and Naresh R. Shanbhag. Analytical guarantees on numerical precision of deep neural networks. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 3007–3016. PMLR, 2017. URL: http://proceedings.mlr.press/v70/sakr17a.html.

[SL19] Taylor Simons and Dah-Jye Lee. A review of binarized neural networks. *Electronics*, 8(6), 2019. URL: https://www.mdpi.com/2079-9292/8/6/661, doi:10.3390/electronics8060661.

[SLC+17] Xiaoyu Sun, Rui Liu, Yi-Ju Chen, Hsiao-Yun Chiu, Wei-Hao Chen, Meng-Fan Chang, and Shimeng Yu. Low-vdd operation of SRAM synaptic array for implementing ternary neural network. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(10):2962–2965, 2017. doi:10.1109/TVLSI.2017.2727528.

[SMDH13] Ilya Sutskever, James Martens, George E. Dahl, and Geoffrey E. Hinton. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on Machine Learning,*

ICML 2013, Atlanta, GA, USA, 16-21 June 2013, volume 28 of *JMLR Workshop and Conference Proceedings*, pages 1139–1147. JMLR.org, 2013. URL: http://proceedings.mlr.press/v28/sutskever13.html.

[SOF+19] Jasper Snoek, Yaniv Ovadia, Emily Fertig, Balaji Lakshminarayanan, Sebastian Nowozin, D. Sculley, Joshua V. Dillon, Jie Ren, and Zachary Nado. Can you trust your model's uncertainty? evaluating predictive uncertainty under dataset shift. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 13969–13980, 2019. URL: https://proceedings.neurips.cc/paper/2019/hash/8558cb408c1d76621371888657d2eb1d-Abstract.html.

[Sou46] Richard Vynne Southwell. *Relaxation Methods in Theoretical Physics: A Continuation of the Treatise, Relaxation Methods in Engineering Science*, volume 2. The Clarendon Press, 1946.

[SPZ+17] Charbel Sakr, Ameya D. Patil, Sai Zhang, Yongjune Kim, and Naresh R. Shanbhag. Minimum precision requirements for the SVM-SGD learning algorithm. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2017, New Orleans, LA, USA, March 5-9, 2017*, pages 1138–1142. IEEE, 2017. doi:10.1109/ICASSP.2017.7952334.

[SS13] Valery Sklyarov and Iouliia Skliarova. Digital hamming weight and distance analyzers for binary vectors and matrices. *International Journal of Innovative Computing, Information and Control*, 9(12):4825–4849, 2013. URL: http://www.ijicic.org/ijicic-12-12021.pdf.

[SS21] André Schidler and Stefan Szeider. Sat-based decision tree learning for large data sets. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pages 3904–3912. AAAI Press, 2021. URL: https://ojs.aaai.org/index.php/AAAI/article/view/16509.

[SSBD14] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014. doi:10.1017/CBO9781107298019.

[SSK+13] Jamie Shotton, Toby Sharp, Pushmeet Kohli, Sebastian Nowozin, John M. Winn, and Antonio Criminisi. Decision jungles: Compact and rich models for classification. In Christopher J. C. Burges, Léon Bottou, Zoubin Ghahramani, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*, pages 234–242, 2013. URL: https://proceedings.neurips.cc/paper/2013/hash/69adc1e107f7f7d035d7baf04342e1ca-Abstract.html.

[ST15] Mojtaba Seyedhosseini and Tolga Tasdizen. Disjunctive normal random forests. *Pattern Recognition*, 48(3):976–983, 2015. doi:10.1016/j.patcog.2014.08.023.

[SWH+17] Lili Song, Ying Wang, Yinhe Han, Huawei Li, Yuanqing Cheng, and Xiaowei Li. STT-RAM buffer design for precision-tunable general-purpose neural network accelerator. *IEEE Transactions on Very Large Scale Integration Systems*, 25(4):1285–1296, 2017. doi:10.1109/TVLSI.2016.2644279.

[SZ15] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL: http://arxiv.org/abs/1409.1556.

[TAA+19] Ryutaro Tanno, Kai Arulkumaran, Daniel C. Alexander, Antonio Criminisi, and Aditya V. Nori. Adaptive neural trees. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 6166–6175. PMLR, 2019. URL: http://proceedings.mlr.press/v97/tanno19a.html.

[Tan] Andrew S. Tanenbaum. *Structured computer organization, 5th Edition*. Pearson Education, Boston, MA.

[TG17] César Torres-Huitzil and Bernard Girau. Fault and error tolerance in neural networks: A review. *IEEE Access*, 5:17322–17341, 2017. doi:10.1109/ACCESS.2017.2742698.

[Tib96] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, 58(1):267–288, 1996. URL: http://www.jstor.org/stable/2346178.

[TPV09] Grigorios Tsoumakas, Ioannis Partalas, and Ioannis P. Vlahavas. An ensemble pruning primer. In Oleg Okun and Giorgio Valentini, editors, *Applications of Supervised and Unsupervised Ensemble Methods*, volume 245 of *Studies in Computational Intelligence*, pages 1–13. Springer, 2009. doi:10.1007/978-3-642-03999-7_1.

[UN96] Naonori Ueda and Ryohei Nakano. Generalization error of ensemble estimators. In *Proceedings of International Conference on Neural Networks (ICNN'96), Washington, DC, USA, June 3-6, 1996*, pages 90–95. IEEE, 1996. doi:10.1109/ICNN.1996.548872.

[VC15] V. N. Vapnik and A. Ya. Chervonenkis. *On the Uniform Convergence of Relative Frequencies of Events to Their Probabilities*, pages 11–30. Springer International Publishing, Cham, 2015. doi:10.1007/978-3-319-21852-6_3.

[VVA+12] Alexander Vergara, Shankar Vembu, Tuba Ayhan, Margaret A. Ryan, Margie L. Homer, and Ramón Huerta. Chemical gas sensor drift compensation using classifier ensembles. *Sensors and Actuators B: Chemical*, 166-167:320–329, 2012. URL: https://www.

sciencedirect.com/science/article/pii/S0925400512002018, doi:https://doi.org/10.1016/j.snb.2012.01.074.

[WC13] Weiran Wang and Miguel Á. Carreira-Perpiñán. Projection onto the probability simplex: An efficient algorithm with a simple proof, and an application. *CoRR*, abs/1309.1541, 2013. URL: http://arxiv.org/abs/1309.1541, arXiv:1309.1541.

[WDM+21] Erwei Wang, James J. Davis, Daniele Moro, Piotr Zielinski, Jia Jie Lim, Claudionor Coelho, Satrajit Chatterjee, Peter Y. K. Cheung, and George A. Constantinides. Enabling binary neural network training on the edge. In *EMDL@MobiSys 2021: Proceedings of the 5th International Workshop on Embedded and Mobile Deep Learning, Virtual Event, Wisconsin, USA, June 25, 2021*, pages 37–38. ACM, 2021. doi:10.1145/3469116.3470015.

[Web00] Geoffrey I. Webb. Multiboosting: A technique for combining boosting and wagging. *Mach. Learn.*, 40(2):159–196, 2000. doi:10.1023/A:1007659514849.

[WH10] Neil Weste and David Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. Addison-Wesley Publishing Company, USA, 4 edition, 2010. URL: https://dl.acm.org/doi/10.5555/1841628.

[WK96] Gerhard Widmer and Miroslav Kubat. Learning in the presence of concept drift and hidden contexts. *Mach. Learn.*, 23(1):69–101, 1996. doi:10.1007/BF00116900.

[WOBM17] Abraham J. Wyner, Matthew Olson, Justin Bleich, and David Mease. Explaining the success of adaboost and random forests as interpolating classifiers. *Journal of Machine Learning Research*, 18:48:1–48:33, 2017. URL: http://jmlr.org/papers/v18/15-240.html.

[WRC+20] Andrew M. Webb, Charles Reynolds, Wenlin Chen, Henry W. J. Reeve, Dan-Andrei Iliescu, Mikel Luján, and Gavin Brown. To ensemble or not ensemble: When does end-to-end training fail? In Frank Hutter, Kristian Kersting, Jefrey Lijffijt, and Isabel Valera, editors, *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2020, Ghent, Belgium, September 14-18, 2020, Proceedings, Part III*, volume 12459 of *Lecture Notes in Computer Science*, pages 109–123. Springer, 2020. doi:10.1007/978-3-030-67664-3_7.

[Wri15] Stephen J. Wright. Coordinate descent algorithms. *Mathematical Programming*, 151(1):3–34, 2015. doi:10.1007/s10107-015-0892-3.

[WRI+19] Andrew M. Webb, Charles Reynolds, Dan-Andrei Iliescu, Henry W. J. Reeve, Mikel Luján, and Gavin Brown. Joint training of neural network ensembles. *CoRR*, abs/1902.04422, 2019. URL: http://arxiv.org/abs/1902.04422, arXiv:1902.04422.

[WZZ+19] Yanzhi Wang, Zheng Zhan, Liang Zhao, Jian Tang, Siyue Wang, Jiayu Li, Bo Yuan, Wujie Wen, and Xue Lin. Universal approximation property and equivalence of stochastic computing-based neural networks and binary neural networks. In *The Thirty-Third AAAI Conference on Artificial*

Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019, pages 5369–5376. AAAI Press, 2019. doi:10.1609/aaai.v33i01.33015369.

[Xil] 7 series fpgas overview. https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf, accessed January 2022. URL: https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf.

[XRV17] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017. https://github.com/zalandoresearch/fashion-mnist, accessed June 2022. arXiv:cs.LG/1708.07747.

[YA14] Olcay Taner Yildiz and Ethem Alpaydin. Regularizing soft decision trees. *Lecture Notes in Electrical Engineering*, 264 LNEE:15–21, 2014. doi:10.1007/978-3-319-01604-7_2.

[YBG+21] Mikail Yayla, Sebastian Buschjäger, Aniket Gupta, Jian-Jia Chen, Jorg Henkel, Katharina Morik, Kuan-Hsun Chen, and Hussam Amrouch. Fefet-based binarized neural networks under temperature-dependent bit errors. *IEEE Transactions on Computers*, pages 1–1, 2021. URL: https://ieeexplore.ieee.org/document/9513530, doi:10.1109/TC.2021.3104736.

[YBM+18] Lita Yang, Daniel Bankman, Bert Moons, Marian Verhelst, and Boris Murmann. Bit error tolerance of a CIFAR-10 binarized convolutional neural network processor. In *IEEE International Symposium on Circuits and Systems, ISCAS 2018, 27-30 May 2018, Florence, Italy*, pages 1–5. IEEE, 2018. doi:10.1109/ISCAS.2018.8351255.

[YCCZ09] Jerry Ye, Jyh-Herng Chow, Jiang Chen, and Zhaohui Zheng. Stochastic gradient boosted distributed decision trees. In David Wai-Lok Cheung, Il-Yeol Song, Wesley W. Chu, Xiaohua Hu, and Jimmy Lin, editors, *Proceedings of the 18th ACM Conference on Information and Knowledge Management, CIKM 2009, Hong Kong, China, November 2-6, 2009*, pages 2061–2064. ACM, 2009. doi:10.1145/1645953.1646301.

[YM17] Lita Yang and Boris Murmann. SRAM voltage scaling for energy-efficient convolutional neural networks. In *18th International Symposium on Quality Electronic Design, ISQED 2017, Santa Clara, CA, USA, March 14-15, 2017*, pages 7–12. IEEE, 2017. doi:10.1109/ISQED.2017.7918284.

[YZZ+18] Ting Ye, Hucheng Zhou, Will Y. Zou, Bin Gao, and Ruofei Zhang. Rapidscorer: Fast tree ensemble evaluation by maximizing compactness in data level parallelization. In Yike Guo and Faisal Farooq, editors, *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018*, pages 941–950. ACM, 2018. doi:10.1145/3219819.3219857.

[ZBH+17] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. In *5th International Conference on Learning Representations, ICLR*

2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings. OpenReview.net, 2017. URL: https://openreview.net/forum?id=Sy8gdB9xx.

[ZBH+21] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning (still) requires rethinking generalization. *Communications of the ACM*, 64(3):107–115, 2021. doi:10.1145/3446776.

[ZBS06] Yi Zhang, Samuel Burer, and W. Nick Street. Ensemble pruning via semi-definite programming. *Journal of machine learning research*, 7:1315–1338, 2006. URL: http://jmlr.org/papers/v7/zhang06a.html.

[ZE01] Bianca Zadrozny and Charles Elkan. Obtaining calibrated probability estimates from decision trees and naive bayesian classifiers. In Carla E. Brodley and Andrea Pohoreckyj Danyluk, editors, *Proceedings of the Eighteenth International Conference on Machine Learning (ICML 2001), Williams College, Williamstown, MA, USA, June 28 - July 1, 2001*, pages 609–616. Morgan Kaufmann, 2001. URL: https://cseweb.ucsd.edu/~elkan/calibrated.pdf.

[Zho12] Zhi-Hua Zhou. *Ensemble methods: foundations and algorithms*. Chapman & Hall/CRC, 1st edition, 2012. URL: https://dl.acm.org/doi/10.5555/2381019.

[ZKN21] Valentina Zantedeschi, Matt J. Kusner, and Vlad Niculae. Learning binary decision trees by argmin differentiation. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 12298–12309. PMLR, 2021. URL: http://proceedings.mlr.press/v139/zantedeschi21a.html.

[ZST+19] Bohan Zhuang, Chunhua Shen, Mingkui Tan, Lingqiao Liu, and Ian D. Reid. Structured binary neural networks for accurate image classification and semantic segmentation. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, pages 413–422. Computer Vision Foundation / IEEE, 2019. URL: http://openaccess.thecvf.com/content_CVPR_2019/html/Zhuang_Structured_Binary_Neural_Networks_for_Accurate_Image_Classification_and_Semantic_CVPR_2019_paper.html, doi:10.1109/CVPR.2019.00050.

[ZW19] Pawel Zyblewski and Michal Wozniak. Clustering-based ensemble pruning and multistage organization using diversity. In Hilde Pérez García, Lidia Sánchez-González, Manuel Castejón Limas, Héctor Quintián-Pardo, and Emilio S. Corchado Rodríguez, editors, *Hybrid Artificial Intelligent Systems - 14th International Conference, HAIS 2019, León, Spain, September 4-6, 2019, Proceedings*, volume 11734 of *Lecture Notes in Computer Science*, pages 287–298. Springer, 2019. doi:10.1007/978-3-030-29859-3_25.

[ZW20] Pawel Zyblewski and Michal Wozniak. Novel clustering-based pruning algorithms. *Pattern Anal. Appl.*, 23(3):1049–1058, 2020. doi:10.1007/s10044-020-00867-8.

[ZY05] Tong Zhang and Bin Yu. Boosting with early stopping: Convergence and consistency. *The Annals of Statistics*, 33(4):1538 – 1579, 2005. doi:10.1214/009053605000000255.