PhD Thesis

Probabilistic Methods for Resource Efficiency in Machine Learning

Dipl.-Ing. Wolfgang Roth, BSc

DOCTORAL THESIS

to achieve the university degree of
Doktor der technischen Wissenschaften

submitted to
Graz University of Technology
Austria

Supervisor and First Assessor/Examiner:
Univ.-Prof. Dipl.-Ing. Dr.mont. Franz Pernkopf
Signal Processing and Speech Communication Laboratory
Graz University of Technology

Second Assessor/Examiner:
Assoc.Prof. Dr. Boris Flach
Department of Cybernetics
Czech Technical University in Prague

Graz, July 2021

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present doctoral thesis.

date	(signature)

Acknowledgements

I would like to thank my supervisor Franz Pernkopf for guiding me through my scientific development in the past six years. Especially in times of moderate scientific success, your motivational skills were vital for me to finally gain a foothold in the scientific world. I appreciate the scientific freedom that you gave me, which allowed me to discover many of the interesting topics of machine learning.

I also want to thank Robert Peharz for offering me the opportunity to visit the CBL lab in Cambridge and to see something else besides the halls of the SPSC lab. This research stay definitely contributed a lot to my positive development both as a researcher and as a person. I also want to take the opportunity to express my belated gratitude to you and to Sebastian Tschiatschek for advising me some years ago when I worked on my master's thesis, where I forgot to officially acknowledge your support. In hindsight, I believe it is in large parts due to both of you that I actually decided to do a PhD.

I really enjoyed the past years working at the SPSC lab, which is—besides my very interesting research topic—largely due to the many nice people who work(ed) there. Being fully aware that I run the risk of missing important people, I want to particularly thank Alex, Christian K., Christian T., Christoph, Elmar, Erik, Florian, Harald, Jamilla, Johanna, Johannes, Josef, Matthias, Markus, Martin T., Martin H., Michi, Sean, Stefan, Truc, and Vlad for sharing coffee breaks, having interesting discussions about research and life, and being exceptionally nice people.

I thank my family, especially my parents, for their endless support in everything I do. You made it possible that I am now close to receiving my PhD and that I have a beautiful life. I really appreciate that you always welcome me at home, allowing me to escape from work and recharge my batteries in the place where I grew up.

Finally, my deepest gratitude goes to my own little family, Anna and Valentina. Thank you, Anna, for always having an open ear and listening to my more or less interesting stories about my work, research, and all the other things that bother me from time to time. You provided the necessary support during the long period of writing this thesis and it is in large parts due to you that I could eventually finish it (especially since Valentina and COVID-19 have arrived almost simultaneously which made life in general an interesting challenge). Thank you, Valentina, for your smile in times when this thesis made only slow progress. That kept me motivated and enabled me to finally get it done.

Abstract

Deep neural networks (DNNs) have gained substantial attention in the past decade. A key factor for many of their recent success stories is the availability of increasing hardware capabilities that have enabled the training of ever-growing network architectures. As a result, the computational requirements of the resulting DNNs are often too high, preventing their use in many interesting real-world applications that must be operated on resource-constrained devices with limited memory, computation power, and battery capacity.

This thesis is dedicated to methods that improve the computational efficiency of machine learning models with a particular focus on DNNs. Our contributions are probabilistic in nature and closely related to Bayesian inference techniques. Probabilistic methods have the advantage that they offer a principled approach to obtaining prediction uncertainties. Furthermore, it is shown that probabilistic methods provide an effective means of converting combinatorial optimization problems into continuous ones that are easier to optimize.

The thesis is divided into two parts. The first part provides a thorough overview of the relevant background on supervised learning, deep neural networks, and Bayesian inference, and continues with an extensive overview on current state-of-the-art methods that improve resource efficiency in deep learning. The second part presents three individual contributions along with extensive experiments showing the effectiveness of the presented methods.

The first contribution is closely related to variational inference and considers weight and activation quantization in DNNs. This reduces the memory footprint of DNNs and enables faster predictions at test-time. We propose to learn discrete weights and activations by learning a distribution over the weights. This is accomplished by propagating distributions through the network using a central limit argument and propagating Gaussians through common building blocks and nonlinear activation functions. Once the weight distribution has been learned, a discrete-valued DNN is inferred by either taking its most probable value or by sampling from it.

The second contribution is concerned with weight sharing in Bayesian DNNs to reduce the memory footprint of storing a large ensemble of DNNs. The weight sharing is obtained by introducing a Dirichlet process prior on top of the weight prior. A sampling based inference scheme is presented that alternates between sampling assignments of weights to connections and sampling the weights themselves. Several algorithmic techniques are presented to overcome computational challenges in order to obtain a tractable algorithm.

The third contribution complements our discussion with an outlook on how methods for improving the resource efficiency of DNNs can be transferred to other model classes. In particular, we present a structure learning algorithm and a quantization approach for Bayesian network classifiers. The presented structure learning approach is closely related to differentiable neural architecture search for DNNs. The method learns a distribution over graph structures using continuous optimization techniques and subsequently selects the most probable structure from that distribution. By introducing a model size penalty to the objective, the method can be used to effectively trade off between model size and accuracy. The presented quantization approach relies on quantization-aware training using the straight-through gradient estimator. Quantization-aware training is currently the most widely used technique for weight and activation quantization in DNNs which allows for effective quantization with minimal changes to existing training pipelines. In extensive experiments, we contrast quantized small-scale DNNs and Bayesian network classifiers and show that both model classes offer benefits in different regimes of computational efficiency and accuracy.

Contents

Statutory Declaration III

Acknowledgements V

Abstract VII

1 Introduction 1
1.1 Scope . . . 2
1.2 Contributions . . . 3
1.3 Outline . . . 6
1.4 Symbols and Notation . . . 7

2 Machine Learning and Deep Neural Networks 11
2.1 Supervised Learning . . . 11
2.1.1 Training and Loss Function Minimization . . . 12
2.1.2 Gradient-Based Optimization . . . 15
2.1.3 Automatic Differentiation . . . 17
2.1.4 The Straight-Through Gradient Estimator . . . 22
2.2 Feed-Forward Deep Neural Networks . . . 23
2.2.1 The Basic Layout of Deep Neural Networks . . . 24
2.2.2 Training Deep Neural Networks . . . 27
2.2.3 Batch Normalization . . . 30
2.2.4 Dropout . . . 32
2.3 A Brief History of Deep Learning Architectures . . . 34
2.3.1 AlexNet . . . 34
2.3.2 VGGNet . . . 35
2.3.3 InceptionNet . . . 35
2.3.4 ResNet . . . 35
2.3.5 DenseNet . . . 36
2.3.6 EfficientNet . . . 37

3 Bayesian Deep Learning 39
3.1 Bayesian Inference . . . 39
3.1.1 Example: The Exponential Family and Conjugate Priors . . . 41
3.1.2 Bayesian Networks . . . 42
3.2 Approximate Bayesian Inference . . . 44
3.2.1 Maximum Likelihood and Maximum A Posteriori Estimation . . . 44
3.2.2 Laplace's Method . . . 46
3.2.3 Variational Inference . . . 47
3.2.4 Sampling Methods . . . 50
3.3 Bayesian Deep Neural Networks . . . 57
3.3.1 Linearization of the Network Output . . . 59
3.3.2 The Probabilistic Forward Pass . . . 60
3.4 Bayesian Neural Networks Using Variational Inference . . . 63
3.4.1 A Closed-Form Approximation Using the Probabilistic Forward Pass . . . 64
3.4.2 Optimization Using Monte Carlo Gradients . . . 65
3.4.3 The Log-Derivative Trick . . . 66
3.4.4 The Reparameterization Trick . . . 68
3.4.5 The Gumbel-Softmax Approximation . . . 70

3.5 Bayesian Neural Networks Using Sampling . . . 72
3.5.1 Stochastic Gradient Langevin Dynamics . . . 72
3.5.2 Stochastic Gradient Hamiltonian Monte Carlo . . . 73

4 Resource-Efficient Deep Neural Networks 75
4.1 Quantized Neural Networks . . . 76
4.1.1 Early Quantization Approaches . . . 76
4.1.2 Quantization-Aware Training . . . 77
4.1.3 Bayesian Approaches for Quantization . . . 79
4.2 Network Pruning . . . 81
4.2.1 Unstructured Pruning . . . 81
4.2.2 Structured Pruning . . . 82
4.2.3 Bayesian Approaches for Network Pruning . . . 83
4.2.4 Dynamic Network Pruning . . . 83
4.3 Structural Efficiency in Deep Neural Networks . . . 83
4.3.1 Weight Sharing . . . 84
4.3.2 Knowledge Distillation . . . 84
4.3.3 Special Matrix Structures . . . 86
4.3.4 Manual Architecture Design . . . 86
4.3.5 Neural Architecture Search (NAS) . . . 88

5 Learning Discrete-Valued Neural Networks Using Weight Distributions 91
5.1 Training with Discrete Weight Distributions . . . 92
5.1.1 The Probabilistic Loss . . . 92
5.1.2 Relation to Variational Inference . . . 94
5.1.3 Optimizing the Probabilistic Loss . . . 94
5.2 Model Details . . . 96
5.2.1 Model Layout . . . 96
5.2.2 Batch Normalization for Gaussian Distributions . . . 97
5.2.3 Max Pooling for Gaussian Distributions . . . 98
5.2.4 Parameterization and Initialization of Weight Distributions . . . 99
5.3 Experiments . . . 100
5.3.1 Dataset Setups . . . 101
5.3.2 Classification Results . . . 102
5.3.3 Straight-Through Gumbel Estimator and Probabilistic Forward Pass . . . 104
5.3.4 Different Max Pooling Methods . . . 105
5.3.5 The Influence of Parameter Initialization and Dropout . . . 106
5.3.6 The Influence of the Distribution Parameterization . . . 108
5.3.7 The Influence of Batch Normalization . . . 108
5.3.8 Model Averaging . . . 109
5.4 Discussion . . . 112
5.4.1 Limitations and Future Work . . . 113

6 Weight Sharing Using Dirichlet Processes 115
6.1 Dirichlet Processes: A Distribution over Distributions . . . 116
6.1.1 Dirichlet Process Mixtures . . . 117
6.1.2 Bayesian Inference for Dirichlet Process Mixtures . . . 119
6.2 Dirichlet Process Neural Networks . . . 122
6.2.1 Posterior Inference in Dirichlet Process Neural Networks . . . 123
6.2.2 Computational Tricks and Inference Complexity . . . 125
6.3 Experiments . . . 127
6.3.1 Classification Results . . . 128

6.3.2 Classification Results with Stochastic Gradient MCMC 129
6.3.3 Regression Results 130
6.3.4 Reducing the Number of Weights 130
6.3.5 Benefit over Random Weight Sharing 131
6.3.6 Running Time Experiments 131
6.3.7 Different Interpolation Methods 131
6.3.8 Influence of the Discretization Parameter 132
6.4 Discussion 132
6.4.1 Limitations and Future Work 133

7 Resource-Efficient Bayesian Network Classifiers 135
7.1 Bayesian Network Classifiers 136
7.1.1 Naïve Bayes and Tree-Augmented Naïve Bayes (TAN) Structures 137
7.1.2 Hybrid Generative-Discriminative Training 138
7.1.3 Structure Learning for Bayesian Networks 139
7.1.4 Relation between Bayesian Network Classifiers and Deep Neural Networks 140
7.2 Differentiable TAN Structure Learning 141
7.2.1 The Structure Learning Loss 141
7.2.2 Minimizing the Structure Learning Loss 142
7.2.3 Model-Size-Aware TAN Structure Learning 143
7.3 Parameter Quantization for Bayesian Network Classifiers 143
7.3.1 Quantization-Aware Bayesian Network Classifiers 143
7.3.2 Quantization-Aware Deep Neural Networks 144
7.4 Structure Learning Experiments 145
7.4.1 Classification Results 146
7.4.2 Heuristic Structures for Image Data 146
7.4.3 Influence of the Feature Ordering and Parent Subsets 148
7.4.4 Recovering the Chow-Liu Structure 148
7.4.5 Model-Size-Aware TAN Structure Learning 148
7.5 Quantization Experiments 150
7.5.1 Fixed Parameter Memory Budget 151
7.5.2 Fixed Number of Operations Budget 153
7.5.3 Quantization for BN Classifiers 153
7.5.4 Comparing Bayesian Network Classifiers and Deep Neural Networks . . . 154
7.6 Discussion 156
7.6.1 Limitations and Future Work 157

8 Conclusions and Outlook 159
8.1 Limitations and Future Work 161

9 List of Publications 163

A Datasets 165
A.1 MNIST 165
A.2 Variants of MNIST 165
A.3 Cifar-10 and Cifar-100 166
A.4 SVHN 166
A.5 USPS 166
A.6 UCI Datasets for Classification 166
A.6.1 Letter 166
A.6.2 Satimage 168

A.7 UCI Datasets for Regression . . . 168
A.7.1 Abalone . . . 168
A.7.2 Boston Housing . . . 168
A.7.3 Concrete Compressive Strength . . . 168
A.7.4 Combined Cycle Power Plant . . . 168
A.7.5 Wine Quality . . . 168

B Useful Calculations **169**
B.1 Full Covariance Gaussian Approximation of the Activation Distribution . . . 169
B.2 Expectation of a Quadratic Form with respect to a Gaussian . . . 170
B.3 Approximating the Logistic Sigmoid by a Gaussian CDF . . . 170
B.4 Approximating the Squared Logistic Sigmoid by a Logistic Sigmoid . . . 171
B.5 Convolving the Logistic Sigmoid with a Gaussian . . . 171
B.6 Convolving the Squared Logistic Sigmoid with a Gaussian . . . 172
B.7 Convolving the (Squared) Hyperbolic Tangent with a Gaussian . . . 172
B.8 Sampling from a Binary Gumbel-Softmax Distribution . . . 173

C List of Acronyms **175**

1 Introduction

Machine learning is nowadays at the core of many systems that simulate intelligent behavior. Consider the following motivating examples that are commonly solved by machine learning techniques. Distinguishing e-mails between spam and non-spam is among the prototypical examples solved by machine learning. For surveillance applications and autonomously driving cars, detecting objects in images such as faces, pedestrians, and cars is an important task. In a medical application, detecting and highlighting patterns in MRI and CT scans is a valuable tool to assist doctors in making a diagnosis. Automatic speech recognition, i.e., the task of transcribing raw audio data containing speech, increasingly finds its way into our living rooms through virtual assistants. Closely related is the task of machine translation, i.e., automatically translating text from one language into another. The recent major breakthrough in artificial intelligence for classical board games, namely achieving superhuman performance in the game of Go, was possible with machine learning [1, 2].

These are just the most prominent examples and it appears that this list can be extended indefinitely. Nevertheless, all of these examples have in common that the variety of different patterns in the data is huge and describing them by hand-crafted rules seems hopeless. For instance, objects in an image (e.g., cars) can be observed in varying views, lighting and weather conditions, objects might be occluded, and other artifacts might be present in the image. However, we expect an intelligent system to be robust with respect to these variations. Rather than devising hand-crafted application-specific rules to capture these patterns, machine learning follows a different approach. The underlying idea is to present a machine learning algorithm with a set of training data samples along with what we expect to be the system's output (i.e., the target). After a so called training or learning phase, the underlying machine learning model should be capable of predicting the target values for the training data. However, the most important point is that the model has implicitly learned to detect the important variations and patterns in the given data which allows it to make accurate predictions for previously unseen data—a property known as generalization. Therefore, considering machine learning as sort of *programming with data* is an appropriate description.

The number of required training samples to achieve a high degree of generalization depends on the specific application and the employed model. On the one side, the more patterns and variations a given dataset might exhibit, the more samples are required to sufficiently cover these patterns. On the other side, we can equip a model with certain prior knowledge such that meaningful patterns can be extracted from relatively few data samples. One particular method of incorporating prior knowledge is by designing hand-crafted features which are provided to the machine learning algorithm. For instance, if we were to know that the co-occurrence of certain words in an e-mail is highly indicative of spam, we could extend our input data by a binary variable (a binary feature) detecting this case. It should then be relatively easy for a machine learning algorithm to discover by itself that a message can be safely classified as spam if this variable is true. Intuitively, if we had not provided the model with this additional information, it would require much more data to discover such a relationship by itself. For a long time, most of the best performing models in various fields relied heavily on hand-crafted features, such as manually designed edge detectors in computer vision.

The situation changed with the renaissance of deep learning. Rather than relying on hand-

crafted features, deep neural networks (DNNs) learn to detect meaningful features during the training procedure. This is accomplished by processing the raw input data through a series of layers, each of which computes increasingly abstract features based on its input from preceding layers. Interestingly, DNNs were essentially known for decades but it took a very long time until they revealed their full potential. With the availability of very large datasets and massively parallel hardware such as graphics processing units (GPUs), it became possible to train large DNNs that outperform their classical counterparts relying on hand-crafted features. This became apparent with the ImageNet challenge in 2012 which is considered as one of the most important landmarks in deep learning. The task of this challenge was to classify images to one of 1,000 object categories using an immensely large training set containing 1.2 million images. The winning entry of this challenge employed a DNN and outperformed all competitors relying on traditional techniques by a large margin. Rather unsurprisingly, most competitors in subsequent challenges relied on deep learning techniques as well and soon afterwards DNNs found their way into many other application domains.

As it turned out, DNNs perform better if they are larger and deeper (i.e., more layers) which has led to ever-growing architectures pushing the state of the art. While we continue to achieve impressive results using deep learning, the resulting models are typically developed under scientific conditions with access to virtually unlimited computing resources. For instance, the initial version of AlphaGo that was able to beat a professional Go player was executed on multiple machines with access to 1,202 central processing units (CPUs) and 176 GPUs [1]. An improved single machine version thereof still runs on four tensor processing units (TPUs) and 44 CPUs [3]. Another extreme example is the recently proposed language model GPT-3 which contains an incredible 175 billion parameters [4]. These conditions often deviate from the real world where resource-constrained devices are ubiquitous. The resulting DNNs are often too large and cannot be deployed on devices with limited memory, computation power, and battery capacity. Therefore, reducing the complexity of DNNs is of central importance for their practical utility. As a result, a very active research area within deep learning is dedicated to improving resource efficiency of DNNs.

Another highly relevant, but often overlooked, aspect is that many machine learning algorithms lack providing meaningful prediction uncertainties. Such models supply us with predictions but they neither provide us with any intuition on how these predictions were computed nor do they tell us how certain they are about their predictions. Prediction uncertainties can be naturally obtained using Bayesian inference which provides us with a solid mathematical framework for probabilistic reasoning. The underlying idea is to introduce uncertainties over the parameters via a parameter prior distribution. Using Bayes' rule, the prior and a likelihood function specified by the model induce a posterior distribution over the parameters. By weighting the predictions obtained for every possible parameter combination according to the posterior, a distribution over the outputs is obtained.

To summarize, a good model must fulfill several aspects: (i) it should be accurate, (ii) it should be resource-efficient, and (iii) it should produce reasonable uncertainty estimates. In the remainder of this chapter, we state the scope of this thesis and define its boundaries. We continue with the contributions of this thesis and provide a brief outline of its content.

1.1 Scope

We have now set the stage to define the scope of this thesis. The topics covered in this thesis lie in the intersection of resource-efficient machine learning and probabilistic modeling. More precisely, we aim to develop probabilistic methods to improve resource efficiency in machine learning. We particularly focus on deep learning and DNNs. The thesis is complemented with a contribution on an inherently probabilistic model class, namely Bayesian networks (BNs).

Although the presented methods have their roots in Bayesian inference, the primary focus of this thesis remains on the resource efficiency aspect. For this reason, we sometimes deviate from the proper Bayesian path and emphasize that we do not claim to perform proper Bayesian inference at all times.

For instance, our contribution in Chapter 5 for quantization in DNNs employs probabilistic modeling to convert a combinatorial optimization problem into a continuous optimization problem. However, the resulting method is still closely related to variational inference, a well-established method for approximate Bayesian inference. We note that similar interpretations are valid for our BN structure learning approach presented in Chapter 7. In Chapter 6, on the other hand, we consider a full Bayesian treatment of DNNs. In this work, we aim to reduce the memory overhead inherent to sampling based inference. For this purpose, we extend the Bayesian model with a particular probabilistic object known as Dirichlet process (DP).

There are several aspects of computational complexity that one may consider in the analysis of resource-efficient models. The theoretical aspects are the model size, the memory requirements to compute predictions, and the number of operations per prediction. However, there also exist practical aspects that strongly depend on the underlying hardware, such as latency (i.e., the actual time to compute a prediction) and energy efficiency. The individual aspects and the model accuracy influence each other and cannot be viewed in isolation. Furthermore, the extent to which theoretical aspects are reflected in a practical implementation is strongly application-specific. In this thesis, we limit ourselves to a theoretical analysis of the computational complexity and refer to the literature for practical considerations if appropriate.

1.2 Contributions

The contributions of this thesis are twofold. On the one side, we provide a comprehensive literature overview of resource efficiency in deep learning which can be considered a contribution by itself. On the other side, we contribute explicitly to this literature. Our contributions to the resource efficiency literature comprise two deep learning methods and another contribution where we transfer techniques from deep learning to a different model class, namely BN classifiers.

Literature overview on resource-efficient deep neural networks: The literature on resource efficiency in deep learning is subject to an enormous growth and the developed techniques are very diverse. However, the objective of these techniques is always the same, i.e., reducing computational costs while maintaining accuracy of existing methods as much as possible. Our first contribution, presented in Chapter 4, provides a broad overview of the current state of the art. We have identified three major categories of methods to reduce the computational complexity of DNNs (see Figure 1.1), i.e., (i) quantization techniques, (ii) parameter or network pruning, and (iii) exploiting structural properties of DNNs. In the following, we give a brief outlook on the individual categories, allowing us to fit our subsequent contributions properly into the literature.

The first category is concerned with quantization of the weights and the activations of a DNN. The numerical quantities involved in the computations of a DNN are typically represented as 32 bit floating-point numbers. The aim of quantization is to map these numerical quantities to representations that require fewer bits and allow for more efficient arithmetic operations.

The second category is concerned with pruning techniques, i.e., achieving a high degree of sparsity in certain matrix and tensor structures. For a sufficient degree of sparsity and adequate data structures, the memory and computation requirements can be drastically reduced. Recently, structured sparsity approaches have been considered that eliminate entire dimensions of matrices and tensors such that no special data structures are required.

The third category exploits structural properties of DNNs to reduce their computational

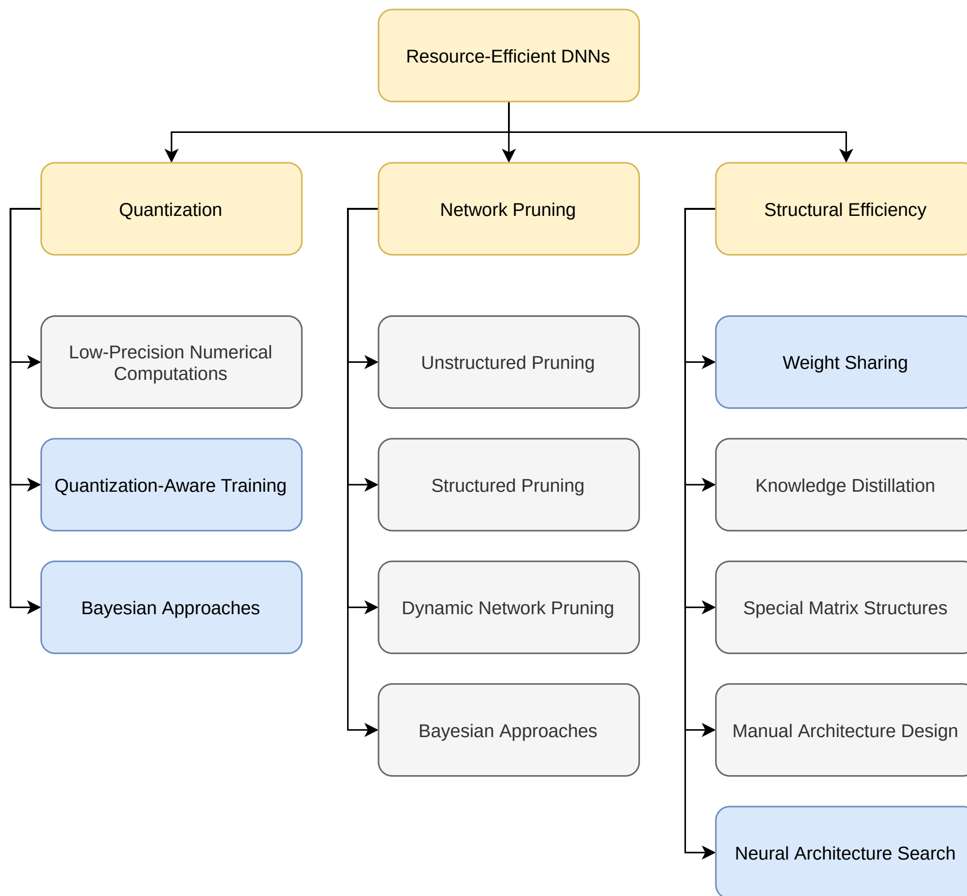


Figure 1.1: Literature overview on resource efficiency in DNNs. We have discovered three major topics: (i) quantization, (ii) network pruning, and (iii) structural efficiency. The individual categories are discussed in detail in Chapter 4. Our own contributions discussed in Chapters 5, 6, and 7 fall into the subcategories shown in blue.

complexity. This category is rather diverse and can be further split into the subcategories (i) weight sharing, (ii) knowledge distillation, (iii) exploiting special matrix structures, (iv) manual architecture design, and (v) neural architecture search (NAS) techniques.

We mention that our overview provides only a snapshot of the current research. Due to the incredibly fast developments in the recent past, we expect that our overview will soon lack many important contributions.

Training discrete-valued DNNs: This contribution belongs to the category of Bayesian quantization techniques. The proposed method is concerned with the training of DNNs having discrete weights and activation functions. We focus on ternary, quaternary, and quinary weights and binary activations using the sign activation function. For ternary weights, the memory footprint is reduced by at least a factor of 16 compared to 32 bit floating-point weights. In conjunction with binary activations from the sign function, multiply-accumulate operations—constituting the bulk of operations—are reduced to (integer) additions and subtractions.

The challenge of training discrete weights is that we are facing a combinatorial optimization problem in millions of variables for which it appears hopeless to apply traditional combinatorial optimization techniques. Common approaches to avoiding the combinatorial optimization are based on the straight-through gradient estimator (STE), a method that enables gradient-based training in the presence of piecewise constant quantization functions. Our method follows a different approach. We introduce a discrete weight distribution and formulate an expected loss with respect to this distribution. This allows us to train the discrete weight distribution by

means of continuous optimization. Once the discrete weight distribution has been trained, we infer a discrete-valued DNN from the weight distribution either by taking its most probable value or by sampling from it. The resulting method is closely related to variational inference, a widely used approximate Bayesian inference technique in the Bayesian deep learning community.

Our method improves on several aspects compared to previous works operating on discrete weight distributions. In particular, our method allows for arbitrary discrete weights whereas previous works are constrained to binary and ternary weights. This is achieved using simpler parameterization and initialization schemes for the weight distribution. Furthermore, previous works applied a max pooling approximation that does not fully utilize the distributional properties. Our work introduces a distribution-aware max pooling approximation based on iterated moment matching. We evaluate our model on several image classification datasets for which we achieve state-of-the-art performance. Our experiments provide thorough insights into various aspects of the proposed method. We empirically show that our parameterization and max pooling approximation facilitate training and result in higher accuracy. Our method allows us to trade off between computational complexity and accuracy in two natural ways. First, we can increase the expressiveness of the model by increasing the number of discrete weight values. Second, we can vary the number of models sampled from the weight distribution for model averaging.

Weight sharing in Bayesian DNNs: This contribution belongs to the category of weight sharing, a subcategory of structural efficiency approaches. The proposed method is concerned with sampling based methods for approximate Bayesian inference. For posterior distributions over the weights of a DNN, sampling is extremely difficult and time-consuming. As a consequence, generating samples on demand may not be an option. A possible solution is to generate samples offline, introducing a substantial memory overhead to store those samples.

To reduce the memory for storing an ensemble of DNNs, we introduce a DP on top of the weight prior in a Bayesian DNN. This induces a weight sharing that is subsequently utilized to drastically reduce the number of parameters. However, it is well-known that sampling based inference using DPs is inherently difficult. Along with unfavorable computational properties inherited from DNNs, the main challenge of this approach is to develop a feasible inference scheme in the first place. For this purpose, we adopt sampling techniques from mixture models—the traditional application of DPs—to DNNs. Moreover, we propose algorithmic techniques and approximations to avoid many redundant computations in order to overcome these difficulties.

The proposed method is applicable to fully connected DNNs for datasets of moderate size, the regime where Bayesian inference is most interesting. We demonstrate the effectiveness of our method in experiments on several image classification and regression tasks. Our method is competitive with DNNs without weight sharing and outperforms randomly shared weights. Especially if only few weights are shared among the connections of a DNN, the method clearly outperforms randomly shared weights. The degree of sharing can be determined by varying the concentration parameter of the DP. This allows us to effectively trade off between model size and accuracy.

Transferring deep learning methods to other model classes: This contribution employs techniques from two categories, i.e., (i) NAS from the category of structural efficiency approaches and (ii) quantization-aware training. With this contribution we positively answer the question whether recent advances in deep learning are transferable to other model classes. For this purpose, we select BN classifiers with naïve Bayes and tree-augmented naïve Bayes (TAN) structures as our model of study. We developed two particular methods. First, we perform TAN structure learning based on differentiable NAS. Second, we perform parameter quantization using quantization-aware training.

Structure learning, even for the relatively simple class of TAN structures, is a difficult combinatorial optimization problem. Traditionally, this problem is solved using score-based approaches

relying on combinatorial search heuristics such as greedy hill climbing. Here, we adopt *differentiable* NAS techniques that are used to train the parameters and the structure of a DNN by means of continuous optimization. The resulting method for TAN structure learning bears many similarities to our method for training discrete-valued DNNs above. In particular, we introduce a probability distribution over the space of TAN structures and formulate an expected loss with respect to this distribution. Subsequently, we jointly train this distribution and the BN parameters using continuous optimization techniques. Once the distribution has been trained, we infer a TAN structure using the most probable model from this distribution. A simple extension of the loss function allows us to also take the model size into account. The proposed method can be easily implemented using modern automatic differentiation frameworks without the need for combinatorial search heuristics.

Our quantization approach is based on quantization-aware training, the predominant approach to perform quantization in DNNs. Quantization-aware training relies on the STE which allows us to perform gradient-based learning in the presence of piecewise constant quantization functions. The typical workflow is as follows. We maintain a set of continuous parameters but evaluate a loss function for quantized versions of these parameters. During gradient computation when the chain rule of calculus is invoked, the zero derivative of the quantization function is replaced by the non-zero derivative of a similar function. In this way, we obtain a non-zero gradient for the continuous parameters which is subsequently used for gradient-based learning. After training, the continuous parameters are discarded and only the quantized versions are kept.

We conduct extensive experiments for both methods. Our structure learning approach consistently outperforms random TAN structures and generative Chow-Liu TAN structures. Using a model size penalty, we can trade off between model size and accuracy. With our quantization approach, we can quantize the parameters to only few bits without notable accuracy degradation. The proposed method outperforms a branch-and-bound algorithm specifically tailored to parameter quantization. We also conduct an extensive comparison of quantized BN classifiers and quantized DNNs. The results show that, depending on the requirements of a specific applications, both model classes are viable options. Furthermore, quantization-aware training performs well for DNNs in the small-scale setting. This implicitly closes a gap in the literature since the majority of papers are dedicated to the large-scale setting.

1.3 Outline

This thesis is divided into nine chapters. Chapter 2 and Chapter 3 introduce the necessary background for the remainder of the thesis. Our literature overview is provided in Chapter 4. Chapters 5, 6, and 7 present our particular contributions to the resource efficiency literature. Chapter 8 concludes this thesis and Chapter 9 provides a list of publications. Details about the datasets used in our experiments and some technical calculations can be found in the appendix. A detailed outline of the individual chapters is provided below.

Chapter 2 introduces the problem setup of supervised learning and shows how to optimize a model for various performance metrics by means of gradient-based learning. We provide an overview on computation graphs and automatic differentiation—two components whose role is crucial for the widespread use of deep learning. Then we introduce DNNs along with commonly used building blocks required to successfully train them. The chapter concludes with a chronological overview of some of the most influential architectures from the past decade.

Chapter 3 introduces the Bayesian inference framework along with common challenges inherent to the subject. We also briefly introduce BNs, the main subject of Chapter 7. Since exact Bayesian inference is intractable for most models, we present common approximate inference

techniques such as variational inference and sampling. After a general treatment of Bayesian inference, we show how to apply the Bayesian framework to DNNs. In particular, we show how variational inference and sampling techniques can be applied to Bayesian DNNs.

Chapter 4 provides a comprehensive overview of the current state of the art to achieve resource efficiency in deep learning. The literature is split into three main categories, i.e., (i) quantization techniques, (ii) parameter pruning, and (iii) exploiting structural properties of DNNs. We review many of the most influential works from the individual categories. Throughout the chapter we highlight existing approaches that can be related to Bayesian inference.

Chapters 5, 6, and 7 present our contributions to the resource efficiency literature. Our first contribution presented in Chapter 5—training discrete-valued DNNs—is based on our paper

- **W. Roth**, G. Schindler, H. Fröning, and F. Pernkopf; *Training Discrete-Valued Neural Networks with Sign Activations Using Weight Distributions*; In: European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD); pp. 382–398, 2019.

Our second contribution presented in Chapter 6—weight sharing in Bayesian ensembles of DNNs—is based on our paper

- **W. Roth** and F. Pernkopf; *Bayesian Neural Networks with Weight Sharing Using Dirichlet Processes*; In: IEEE Transactions on Pattern Analysis and Machine Intelligence; vol. 42 (1), pp. 246–252, 2020.

Our third contribution presented in Chapter 7—transferring techniques for improving resource efficiency in deep learning to BNs—is based on our papers

- **W. Roth** and F. Pernkopf; *Differentiable TAN Structure Learning for Bayesian Network Classifiers*; In: International Conference on Probabilistic Graphical Models (PGM); pp. 389–400, 2020,

- **W. Roth**, G. Schindler, H. Fröning, and F. Pernkopf; *On Resource-Efficient Bayesian Network Classifiers and Deep Neural Networks*; In: International Conference on Pattern Recognition; pp. 10297–10304, 2020.

These contributions are largely independent from each other and can be read in arbitrary order. We note that each of these chapters contains its own experiments and discussion sections.

Chapter 8 concludes the thesis. We discuss the most important findings of the individual contributions and state limitations of the proposed methods. These limitations immediately lead to promising directions of future research that were not addressed in this thesis.

Chapter 9 provides a complete list of publications that emerged either directly as part of this thesis or in related contexts.

1.4 Symbols and Notation

General notation guidelines: The general notation guideline used throughout this thesis is as follows. Non-boldface symbols (e.g., a, x, L, N, ε, θ, ...) indicate scalar values for both uppercase and lowercase letters. Boldface lowercase symbols (e.g., $\mathbf{a}$, $\mathbf{x}$, ...) indicate (column) vectors. Boldface uppercase symbols (e.g., $\mathbf{B}$, $\mathbf{W}$, ...) indicate matrices (or tensors). We

emphasize, however, that there are several exceptions to these general notation guidelines. For instance, we generally treat inputs and outputs of DNN layers as vectors $\mathbf{x}$ and, therefore, we also denote the tensor-valued inputs and outputs of convolutional layers by lowercase symbols $\mathbf{x}$. Moreover, we sometimes abuse notation and use boldface uppercase and lowercase symbols to denote *sets* of vectors and matrices (or tensors), respectively. In all of these cases, the corresponding symbols will be properly introduced and, given the context, there should be no confusion about their meaning.

We aim for a consistent use of symbols throughout the thesis. However, due to the extent of this thesis, this was rather challenging and we sometimes had to overload symbols. If symbols are overloaded and only required in a narrow context, their meaning should be clear. If important symbols are required at several locations in the thesis, their specific use is often indicated by a subscript (e.g., $\beta_{\text{bn}}, \gamma_{\text{bn}}, \alpha_{\text{dp}}, \ldots$).

For probability distributions, we rely on the arguments of $p(\cdot)$ to identify the appropriate distribution. We use the symbol $q(\cdot)$ to indicate various kinds of approximating distributions. In our notation, we typically do not make a clear distinction between random variables and concrete instantiations thereof. However, we sometimes use the notation $p(A > 0)$ or $p(Y = 1)$ to refer to the probability of an event where the uppercase symbols correspond to the random variable of a lowercase symbol that is currently being discussed. Only in our discussion about BNs, we make a clearer distinction between uppercase random variables $\boldsymbol{X}, X, Y$ and lowercase instantiations $\mathbf{x}$, x, y. In particular, by $p(\boldsymbol{X})$ we refer to distributional properties while $p(\mathbf{x})$ is the probability density function (pdf) (or probability mass function (pmf)) evaluated at $\mathbf{x}$.

Computation graphs: This thesis contains several graphical illustrations of computation graphs. Circles correspond to value nodes. Boxes correspond to operation nodes. Diamonds correspond to sampling nodes. Blue value nodes correspond to parameters whose gradient is relevant for gradient-based learning. Yellow value nodes correspond to the output of the computation graph. Green operation nodes correspond to differentiable functions that are compatible with backpropagation. Red operation nodes correspond to operations that are not compatible with backpropagation, i.e., they are not differentiable or their gradient is zero almost everywhere. Sampling nodes are always shown in red. The forward path is indicated by solid red lines. The backward path is indicated by dashed green lines.

Bayesian networks: This thesis contains several graphical illustrations of BNs. Variables are shown as circles. Shaded circles correspond to known (observed) quantities whereas white circles are unknown variables. The replication of variables is indicated using plate notation, i.e., boxes around circles along with a number that indicates how often the variables are replicated. Dashed circles indicate that the corresponding variable is optional and not present in every setting.

Mathematical symbols: The following list provides descriptions of mathematical symbols used in this thesis.

$\mathbb{R}$	Real numbers
$\mathbb{Z}$	Integers
(a, b)	Open interval from a to b
$[a, b]$	Closed interval from a to b
$\otimes$	A (unspecified) linear operation
$*$	Convolution
$\odot$	Element-wise multiplication
$A \ll B$	A is much smaller than B (qualitative)
$A \propto B$	A is proportional to B

$p_{\boldsymbol{\theta}}$ / $f_{\boldsymbol{\theta}}$	Parametric distribution / function with parameters $\boldsymbol{\theta}$
$\mathbf{u} \sim p(\mathbf{u})$	$\mathbf{u}$ is sampled from (or distributed according to) $p(\mathbf{u})$
$\mathbb{E}[f(\mathbf{u})]$	Expectation of $f(\mathbf{u})$ (distribution clear from context)
$\mathbb{E}_{\mathbf{u}\sim p(\mathbf{u})}[f(\mathbf{u})]$	Expectation of $f(\mathbf{u})$ with respect to $p(\mathbf{u})$
$\mathbb{V}[f(\mathbf{u})]$	Variance of $f(\mathbf{u})$ (distribution clear from context)
$\mathrm{cov}(u_1, u_2)$	Covariance of u_1 and u_2 (distribution clear from context)
$D_{\mathrm{KL}}(p\|\|q)$	Kullback-Leibler divergence between p and q
$\mathbb{H}[p]$	(Differential) entropy of p
$I(X_1; X_2)$	Mutual information of X_1 and X_2
$\mathcal{N}(\mu, \sigma^2)$	Gaussian distribution with mean μ and variance σ^2
$\mathcal{U}(\mathcal{Z})$	Uniform distribution over the set $\mathcal{Z}$
$\mathrm{Bernoulli}(p)$	Bernoulli distribution with success probability p
$\mathrm{Discrete}(\mathbf{p})$	Discrete distribution with probabilities $\mathbf{p}$
$\mathrm{Gumbel}(0, 1)$	Standard Gumbel distribution
$\mathrm{Beta}(\alpha, \beta)$	Beta distribution with parameters α and β
$\mathrm{Dirichlet}(\alpha_1, \ldots, \alpha_K)$	K-dimensional Dirichlet distribution with parameters $\alpha_1, \ldots, \alpha_K$
$\mathcal{DP}(G_0, \alpha_{\mathrm{dp}})$	DP with base distribution G_0 and concentration parameter α_{dp}
$\delta_{\mathbf{z}}(\cdot)$	Point mass distribution at $\mathbf{z}$
$\phi(\cdot)$ / $\Phi(\cdot)$	pdf / cdf of the standard normal distribution
$\|u\|$	Absolute value of u
$\|\mathbf{u}\|$ / $\|\mathbf{U}\|$	Number of elements of $\mathbf{u}$ / $\mathbf{U}$
$\\|\mathbf{u}\\|$	ℓ^2-norm of $\mathbf{u}$
$\mathbf{u}^\top$ / $\mathbf{U}^\top$	Transpose of $\mathbf{u}$ / $\mathbf{U}$
$\mathbf{I}$	Identity matrix (dimension clear from context)
$\mathbf{e}_i$	i^{th} unit vector
$\det(\cdot)$	Determinant of a square matrix
$\mathrm{tr}(\cdot)$	Trace of a square matrix (i.e., sum of its diagonal entries)
$\mathrm{diag}(\mathbf{u})$	Square diagonal matrix with entries $\mathbf{u}$ on its diagonal
$f'(u)$	Derivative of $f(u)$
$\nabla_{\mathbf{u}} f(\mathbf{u})$	Gradient of $f(\mathbf{u})$ with respect to $\mathbf{u}$
$\mathbf{J}_f(\mathbf{u})$	Jacobian of $f(\mathbf{u})$ with respect to $\mathbf{u}$
$\nabla^2_{\mathbf{u}} f(\mathbf{u})$	Hessian of $f(\mathbf{u})$ with respect to $\mathbf{u}$
$\log(u)$	The natural logarithm
$\log_b(u)$	The logarithm to base b
$\mathrm{sign}(u)$	The sign function
$\tanh(u)$	The hyperbolic tangent
$\mathrm{sigm}(u)$	The logistic sigmoid function
$\mathrm{softmax}(\mathbf{u})$	The softmax function
$\mathrm{softmax}_i(\mathbf{u})$	i^{th} output of the softmax function
$\mathrm{clip}(v, l, u)$	Clipping function $\min(\max(v, l), u)$
$\mathrm{round}(u)$	Rounds u to the closest integer
$\mathrm{quant}(u)$	A quantization function
$\mathbb{I}[\text{condition}]$	The indicator function (outputs 1 if condition is true, 0 otherwise)
$\mathrm{pa}(v)$	Parents of v in a graph (graph clear from context)
$\mathrm{ch}(v)$	Children of v in a graph (graph clear from context)
$f(n) = \mathcal{O}(g(n))$	$\|f(n)/g(n)\|$ is bounded as $n \to \infty$

2 Machine Learning and Deep Neural Networks

This chapter introduces the general framework of supervised learning—the problem setup we are facing throughout this thesis. We focus on the concept of loss function minimization and demonstrate how computation graphs and automatic differentiation make gradient-based minimization particularly convenient. Subsequently, we delve into the specific model class of DNNs—the main subject of this thesis. We formally introduce DNNs along with common building blocks, such as batch normalization and dropout, that have enabled the training of very deep architectures. We argue how these tools—together with automatic differentiation, the use of highly parallel hardware, and the availability of very large (labeled) datasets—have opened the door to the widespread use of deep learning techniques in many different areas.

Especially since the ImageNet challenge in 2012 [5] when the AlexNet architecture [6] surpassed the accuracy of classical computer vision approaches by an unexpectedly large margin, many novel DNN architectures have emerged. We conclude this chapter with a brief chronological overview of a selection of the most influential DNN architectures from the past decade along with their fundamental design principles.

2.1 Supervised Learning

Let $\mathcal{D} = \{(\mathbf{x}_n, \mathbf{y}_n)\}_{n=1}^N$ be a set of N input-target pairs where $\mathbf{x}_n \in \mathcal{X}$ and $\mathbf{y}_n \in \mathcal{Y}$. The task of supervised learning is to infer a functional relationship $\hat{f} : \mathcal{X} \to \mathcal{Y}$ from $\mathcal{D}$. Subsequently, the function $\hat{f}$ can be used to predict unknown target values $\mathbf{y}$ for new inputs $\mathbf{x}$ that are not part of $\mathcal{D}$.[1] Consider the following examples that fall into this framework.

- Object classification in images: $\mathbf{x}$ is a vector of pixel intensities and $y \in \{1, \dots, \mathcal{C}\}$ indicates the object shown in that image.
- Automatic speech recognition: $\mathbf{x}$ represents a speech recording and, depending on the granularity of the task, $\mathbf{y}$ indicates a phoneme, a word, or even an entire sentence.
- Medicine: $\mathbf{x}$ is a vector of medical measurements and $y \in \{0, 1\}$ indicates whether a person suffers from a certain disease or not.
- Spam detection: $\mathbf{x}$ is a message and $y \in \{0, 1\}$ determines whether it is spam or non-spam.
- Document classification: $\mathbf{x}$ is a document and $y \in \{1, \dots, \mathcal{C}\}$ is a topic that best describes its content.
- Age prediction: $\mathbf{x}$ is a vector of observable features of a certain animal or plant and $y \in \mathbb{R}_+$ corresponds to its age (e.g., Appendix A.7.1).
- Stock market prediction: $\mathbf{x} \in \mathbb{R}_+^D$ are the stock prices of the past D days and $y \in \mathbb{R}_+$ is the stock price of tomorrow.

[1] We omit the subscript indicating the sample index when we make general statements about inputs and outputs.

The input spaces $\mathcal{X}$ and output spaces $\mathcal{Y}$ of these examples are rather diverse. Depending on the type of output space $\mathcal{Y}$, we distinguish between two fundamental tasks. For discrete $\mathcal{Y}$, we are facing a *classification* task. The medical and the spam example are special instances of *binary* classification where only two outcomes are possible. If there are more than two possible outcomes, this is referred to as *multiclass* classification. For continuous $\mathcal{Y}$, we are facing a *regression* task.

The examples above also exhibit a large diversity in the type of input spaces $\mathcal{X}$. For instance, audio recordings and documents are sequential data of variable length whereas the measurements of the medical example may be represented by a vector of fixed length. The practical consequences of this diversity are that we either must resort to special purpose methods tailored to a particular type of data or that we first map the given data to some more convenient space and then employ a well-established method. Throughout this thesis, we assume that the inputs are fixed-size D-dimensional vectors $\mathbf{x} \in \mathbb{R}^D$ and that any necessary data pre-processing steps have already been conducted.

In reality, the observed dataset $\mathcal{D}$ rarely corresponds to some actual input-output pairs of some function, but rather to some noisy observations thereof. The data acquisition process is typically subject to measurement errors for both the inputs and the outputs. For instance, even if we observe the same input values $\mathbf{x}_n = \mathbf{x}_{n'}$, the corresponding observed outputs $\mathbf{y}_n \neq \mathbf{y}_{n'}$ might differ. Another source of stochasticity arises due to incompleteness of the data. This occurs when some potentially important factors of the underlying process are simply not measured.

To account for all sorts of stochasticity in the acquisition process, we assume that the given data $\mathcal{D}$ is generated according to an underlying data distribution $p_{\text{data}}(\mathbf{x}, \mathbf{y})$. Given access to p_{data}, we can quantify the uncertainty about the target $\mathbf{y}$ for an observed input $\mathbf{x}$ in terms of a conditional distribution

$$p_{\text{data}}(\mathbf{y} \,|\, \mathbf{x}) = \frac{p_{\text{data}}(\mathbf{x}, \mathbf{y})}{\int_{\mathcal{Y}} p_{\text{data}}(\mathbf{x}, \mathbf{y}) \, d\mathbf{y}}. \tag{2.1}$$

For classification, the integral is replaced by a sum. We can then conveniently define a predictor $\hat{f}$ using (2.1). For classification, it is common to compute the most probable class, i.e.,

$$\hat{f}(\mathbf{x}) = \operatorname*{argmax}_{y \in \mathcal{Y}} p_{\text{data}}(y \,|\, \mathbf{x}) = \operatorname*{argmax}_{y \in \mathcal{Y}} p_{\text{data}}(\mathbf{x}, y). \tag{2.2}$$

The resulting classifier (2.2) is also referred to as the *Bayes classifier*. For regression, it is common to compute the conditional mean, i.e.,

$$\hat{f}(\mathbf{x}) = \mathbb{E}_{p_{\text{data}}(\mathbf{y}|\mathbf{x})}[\mathbf{y}]. \tag{2.3}$$

Both of these predictors, (2.2) and (2.3), are in a sense optimal. Predictor (2.2) minimizes the expected 0-1 loss with respect to p_{data} for classification. Predictor (2.3) minimizes the expected squared loss with respect to p_{data} for regression.

Given access to p_{data}, supervised learning is essentially solved. However, p_{data} is typically unknown and we only have access to a set of samples $\mathcal{D}$ generated from p_{data}. In the following, we show how to infer a predictor merely from the dataset $\mathcal{D}$.

2.1.1 Training and Loss Function Minimization

The preceding discussion shows that supervised learning can be reduced to estimating the underlying data distribution p_{data}. A common approach is to approximate p_{data} by a member $p_{\boldsymbol{\theta}}$ from a parametric family of distributions $\{p_{\boldsymbol{\theta}}\}_{\boldsymbol{\theta} \in \boldsymbol{\Theta}}$. Subsequently, $p_{\boldsymbol{\theta}}$ is used as a proxy for p_{data} in (2.2) and (2.3) to obtain a predictor $\hat{f}_{\boldsymbol{\theta}}$. This section discusses the central task of supervised learning: the training (or learning) procedure. We discuss the typical training procedure through

the minimization of a loss function. Furthermore, we highlight the subtleties that distinguish supervised learning from mere function optimization.

Let $\ell(\hat{\mathbf{y}}, \mathbf{y})$ be the per-sample loss incurred by predicting $\hat{\mathbf{y}}$ for a given target $\mathbf{y}$. Our ultimate goal is to find parameters $\boldsymbol{\theta}$ that minimize the expected per-sample loss with respect to p_{data}, i.e.,

$$\mathcal{L}_{\text{exp}}(\boldsymbol{\theta}) = \mathbb{E}_{(\mathbf{x},\mathbf{y})\sim p_{\text{data}}}[\ell(\hat{f}_{\boldsymbol{\theta}}(\mathbf{x}), \mathbf{y})]. \tag{2.4}$$

Since (2.4) is defined in terms of the unknown data distribution p_{data}, we cannot directly minimize it. Therefore, we approximate the expectation (2.4) through the samples in $\mathcal{D}$. This yields the empirical loss[2]

$$\mathcal{L}_{\text{emp}}(\boldsymbol{\theta}; \mathcal{D}) = \frac{1}{N} \sum_{n=1}^{N} \ell(\hat{f}_{\boldsymbol{\theta}}(\mathbf{x}_n), \mathbf{y}_n). \tag{2.5}$$

For regression, the per-sample loss is selected to be the squared error function $\ell(\hat{\mathbf{y}}, \mathbf{y}) = \|\hat{\mathbf{y}} - \mathbf{y}\|^2$. The corresponding empirical loss (2.5) yields the mean squared error (MSE) loss

$$\mathcal{L}_{\text{MSE}}(\boldsymbol{\theta}; \mathcal{D}) = \frac{1}{N} \sum_{n=1}^{N} \|\hat{f}_{\boldsymbol{\theta}}(\mathbf{x}_n) - \mathbf{y}_n\|^2. \tag{2.6}$$

For classification, the per-sample loss is selected as the 0-1 loss $\ell(\hat{\mathbf{y}}, \mathbf{y}) = \mathbb{I}[\hat{\mathbf{y}} \neq \mathbf{y}]$. Here, $\mathbb{I}$ is the indicator function that yields 1 if its argument is true and 0 otherwise. The corresponding empirical loss (2.5) is the classification error on the given dataset $\mathcal{D}$.

Many modern machine learning algorithms rely on gradient-based optimization. This requires that the loss function $\mathcal{L}$ is differentiable and its gradient $\nabla_{\boldsymbol{\theta}}\mathcal{L}$ is non-zero. However, the 0-1 loss for classification is piecewise constant and its gradient is zero almost everywhere. Therefore, we typically minimize a differentiable surrogate that correlates with the 0-1 loss. For binary classification with $\mathcal{Y} = \{0, 1\}$, we minimize the binary cross-entropy loss defined as

$$\mathcal{L}_{\text{BCE}}(\boldsymbol{\theta}; \mathcal{D}) = -\left(\frac{1}{N} \sum_{n=1}^{N} y_n \log p_{\boldsymbol{\theta}}(Y = 1 \,|\, \mathbf{x}_n) + (1 - y_n) \log\left(1 - p_{\boldsymbol{\theta}}(Y = 1 \,|\, \mathbf{x}_n)\right)\right). \tag{2.7}$$

For multiclass classification with targets $\mathcal{Y} = \{1, \ldots, \mathcal{C}\}$, the corresponding multiclass cross-entropy loss is defined as

$$\mathcal{L}_{\text{CE}}(\boldsymbol{\theta}; \mathcal{D}) = -\frac{1}{N} \sum_{n=1}^{N} \log p_{\boldsymbol{\theta}}(Y = y_n \,|\, \mathbf{x}_n). \tag{2.8}$$

The minimization of (2.7) and (2.8) requires a model that produces conditional probabilities $p_{\boldsymbol{\theta}}(Y = y \,|\, \mathbf{x})$. Fortunately, this holds for many models encountered in practice and, in particular, for DNNs (see Section 2.2).

Note that the cross-entropy loss is a suitable proxy for the 0-1 loss since it penalizes samples for which the output probabilities of the true class $p_{\boldsymbol{\theta}}(y_n \,|\, \mathbf{x}_n)$ are not close to one. In fact, the cross-entropy loss is a natural choice for a loss function. Both the MSE and the cross-entropy loss can be interpreted as likelihood maximization (see Section 3.2.1). The likelihood is an important metric since it also takes the output probabilities $p(\mathbf{y} \,|\, \mathbf{x})$ into account. The likelihood incurs a large loss for highly confident wrong predictions, but it does not severely penalize wrong predictions where the true target is still assigned a high probability. This is in contrast to the 0-1 loss that simply makes a binary decision.

According to our ongoing discussion, supervised learning reduces to an optimization problem.

[2] The empirical and expected loss are also known as empirical and expected *risk* in some contexts (e.g., see [7]).

However, there are some subtle differences that distinguish the training procedure from a pure optimization task. We have shown that we must resort to the minimization of an empirical loss $\mathcal{L}_{\text{emp}}$ defined in terms of a dataset $\mathcal{D}$. Furthermore, in the case of classification, the cross-entropy loss is a surrogate loss deviating from our true objective, the 0-1 loss. Therefore, a minimizer of the resulting empirical loss $\mathcal{L}_{\text{emp}}$ is generally not optimal for the expected loss $\mathcal{L}_{\text{exp}}$ that we ultimately wish to minimize. This must be taken into account to obtain a model that performs well on previously unseen data—a property known as *generalization*. Indeed, many models, such as DNNs [8], are universal function approximators capable of representing arbitrarily complex functional relationships. Such models are particularly prone to *overfitting*, meaning that they achieve a high accuracy on the given dataset but still perform poorly on unseen data.

Because of this, a typical supervised learning algorithm splits the given dataset $\mathcal{D}$ into three disjoint subsets: (i) a training set $\mathcal{D}_{\text{tr}}$, (ii) a validation set $\mathcal{D}_{\text{va}}$, and (iii) a test set $\mathcal{D}_{\text{te}}$. The training set $\mathcal{D}_{\text{tr}}$ is used to define the empirical loss $\mathcal{L}_{\text{emp}}$ which is typically minimized using gradient-based optimization. The true objective that we wish to minimize (e.g., the 0-1 loss for classification) is evaluated on the separate held-out validation set $\mathcal{D}_{\text{va}}$. The purpose of the validation set is twofold. First, since $\mathcal{D}_{\text{va}}$ is not used to specify the empirical loss $\mathcal{L}_{\text{emp}}$, the validation performance provides a better estimate of the generalization performance on previously unseen data. Second, the validation set is used for *model selection* and *hyperparameter optimization*. Hyperparameters are high-level parameters that govern the structure of a model, the loss function, or the optimization algorithm itself. These hyperparameters may have a large impact on the generalization performance but need to be selected before the optimization of $\mathcal{L}_{\text{emp}}$. Using a validation set, it is also common to apply *early stopping* to tune the number of iterations of an iterative optimization procedure. Early stopping selects an intermediate model that achieves the best validation performance during training and optionally terminates training early if no further progress can be expected. However, by tuning hyperparameters on a validation set, we obtain a model whose validation performance is an optimistically biased estimate of the generalization performance. To obtain an unbiased estimate of the generalization performance, we require another distinct test set $\mathcal{D}_{\text{te}}$ on which the selected model is finally evaluated.

The problem of overfitting is a major topic of supervised learning. Overfitting typically occurs when the selected model family $\{p_{\boldsymbol{\theta}}\}_{\boldsymbol{\theta}\in\boldsymbol{\Theta}}$ exhibits great expressiveness and training data is scarce. In this case, the training algorithm selects an overly complex model $p_{\boldsymbol{\theta}}$ that memorizes the individual data points, including all the noise that is present in the training data. As a result, such a model will fail to capture the underlying patterns in the data which is necessary to generalize well.

Overfitting is typically avoided by limiting the expressiveness of $p_{\boldsymbol{\theta}}$. A straightforward solution is to select a family of simple (low complexity) models $\{p_{\boldsymbol{\theta}}\}_{\boldsymbol{\theta}\in\boldsymbol{\Theta}}$. The number of free model parameters (i.e., the number of dimensions of $\boldsymbol{\theta}$) is a commonly used criterion for model complexity, suggesting to employ a model with few parameters. If such a model performs well on the training data, it is also expected to perform well on unseen data. Such simple models, however, may suffer from the reverse problem of underfitting in which case the model class is not rich enough to capture the given data.

Therefore, it is often more convenient to select a family of models $\{p_{\boldsymbol{\theta}}\}_{\boldsymbol{\theta}\in\boldsymbol{\Theta}}$ that is rich enough to model the data distribution p_{data}. Then we seek for the simplest model $p_{\boldsymbol{\theta}}$ in that family that still captures the training data well. Since the number of model parameters is now fixed, a different notion of model complexity is required. Intuitively, the output of a simple function does not vary much if we slightly vary its input. For many models, larger parameters $\boldsymbol{\theta}$ directly correspond to stronger variations in the induced function $\hat{f}_{\boldsymbol{\theta}}$. We can utilize this observation by incorporating a *regularization* term $\mathcal{R}(\boldsymbol{\theta})$ to the loss function which penalizes large parameters. This yields a regularized loss given by the sum of a data term and a regularizer, i.e.,

$$\mathcal{L}(\boldsymbol{\theta}; \mathcal{D}) = \mathcal{L}_{\text{data}}(\boldsymbol{\theta}; \mathcal{D}) + \lambda \mathcal{R}(\boldsymbol{\theta}), \tag{2.9}$$

Algorithm 1 Gradient descent

1: **Input:** Loss $\mathcal{L}(\boldsymbol{\theta})$, initial parameters $\boldsymbol{\theta}^0$, learning rates $(\eta_t)_{t\geq 1}$
2: $t \leftarrow 1$
3: **while** stopping criterion not met **do**
4: $\quad \boldsymbol{\theta}^t \leftarrow \boldsymbol{\theta}^{t-1} - \eta_t \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}^{t-1})$
5: $\quad t \leftarrow t + 1$
6: **end while**

where $\lambda > 0$ is a trade-off hyperparameter and $\mathcal{L}_{\text{data}}$ is an arbitrary loss. A common choice for the regularization term $\mathcal{R}(\boldsymbol{\theta})$ is the ℓ^2-norm over the parameters.[3] An optimal model with respect to (2.9) must fit the data well and be sufficiently simple. In practice, the challenge remains to select an optimal trade-off parameter λ to obtain a good balance between data fit and model complexity. As we will see in Section 3.2.1, minimizing the regularized loss (2.9) using an ℓ^2-norm regularizer is equivalent to maximizing a posterior density.

We will encounter several other regularization techniques in this thesis. For instance, many techniques achieve regularization by making training more difficult by injecting different kinds of noises (e.g., dropout [9]). In the following sections, we focus on the optimization aspect and ignore the specifics of the training procedure.

2.1.2 Gradient-Based Optimization

The minimization of a loss function is the central task of most machine learning algorithms. Especially in the context of deep learning, loss function minimization is extremely challenging. The encountered models are often specified by millions of parameters and we have to deal with non-convex, highly nonlinear loss functions that exhibit many flat regions and poor local minima.

In principle, we could treat a loss $\mathcal{L}(\boldsymbol{\theta})$ as a black box function without access to its derivatives. However, black box optimization tends to perform poorly for high-dimensional parameter spaces $\boldsymbol{\Theta}$. Especially for deep learning, most optimization techniques are first-order methods where we have access to the gradient of the loss function $\mathcal{L}$. The negative gradient of a function points into the direction of steepest descent, i.e., the direction in which the function locally decreases the most. By slightly moving the current parameters $\boldsymbol{\theta}$ into the direction of the negative gradient of $\mathcal{L}$, we decrease the loss and obtain a better model. The *gradient descent* algorithm (see Algorithm 1) implements this idea in an iterative manner to progressively reduce the loss.

Gradient descent is governed by a learning rate (or step size) hyperparameter $\eta_t > 0$ that determines how much the parameters $\boldsymbol{\theta}$ are changed in each iteration. Under mild conditions on the loss $\mathcal{L}(\boldsymbol{\theta})$ and using suitable learning rates η_t, gradient descent converges to a stationary point where the gradient vanishes [10]. In practice, the algorithm is iterated until some stopping criterion is met. Typical stopping criteria are based on a maximal number of iterations, a fixed time budget, diminishing changes of the loss $\mathcal{L}$ or the parameters $\boldsymbol{\theta}$, or a sufficiently small gradient norm $\|\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})\|$.

It turns out that the learning rate η_t is one of the most important hyperparameters whose choice is crucial for the success of many learning algorithms. The challenge is to select a suitable step size η_t that is large enough to sufficiently reduce the loss function while not running the risk of increasing the loss again.

Stochastic Gradient Descent

Gradient descent provides a powerful framework to minimize any differentiable loss $\mathcal{L}(\boldsymbol{\theta})$. However, the standard version as defined in Algorithm 1 is impractical for losses specified by very

[3] In the context of DNNs, an ℓ^2-norm regularizer is also called weight decay term.

Algorithm 2 Stochastic gradient descent (mini-batch version)

1: **Input:** Loss $\mathcal{L}(\boldsymbol{\theta};\mathcal{D})$, initial parameters $\boldsymbol{\theta}^0$, learning rates $(\eta_t)_{t\geq 1}$, mini-batch size N_B
2: $t \leftarrow 1$
3: **while** stopping criterion not met **do**
4: $\quad\mathcal{D} \leftarrow$ shuffle $\mathcal{D}$
5: $\quad(\mathcal{D}_1,\dots,\mathcal{D}_K) \leftarrow$ partition $\mathcal{D}$ into K mini-batches of size N_B
6: $\quad$**for** $k = 1$ **to** K **do**
7: $\qquad\boldsymbol{\theta}^t \leftarrow \boldsymbol{\theta}^{t-1} - \eta_t \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}^{t-1};\mathcal{D}_k)$
8: $\qquad t \leftarrow t+1$
9: $\quad$**end for**
10: **end while**

large datasets. As we have already seen, many common loss functions are defined by a sum over individual per-sample losses $\ell_n(\boldsymbol{\theta})$ as

$$\mathcal{L}(\boldsymbol{\theta};\mathcal{D}) = \frac{1}{N}\sum_{n=1}^{N} \ell_n(\boldsymbol{\theta}) \qquad \text{where} \qquad \ell_n(\boldsymbol{\theta}) = \ell(\hat{f}_{\boldsymbol{\theta}}(\mathbf{x}_n), \mathbf{y}_n). \tag{2.10}$$

The cost of evaluating (2.10) and its gradient scales linearly with the total number of samples N, rendering gradient descent impractical for large N as the gradient is required in every iteration.

The common solution is provided by the framework of *stochastic gradient descent (SGD)* where the parameters are updated based on cheaper stochastic Monte Carlo gradients. Instead of computing the gradient for the whole dataset $\mathcal{D}$, SGD estimates the gradient based on a single per-sample loss $\ell_n(\boldsymbol{\theta})$ for a randomly selected sample n. This allows us to estimate the gradient at a cost that is independent of the dataset size N. Consequently, we can perform several gradient updates for the same computational budget as required by a single update based on the exact gradient.[4] Furthermore, for an online setting with an indefinite number of samples, SGD might be the only option as computing exact gradients is not possible.

However, these approximate gradients exhibit a certain amount of variance that might occasionally drive the parameters into bad directions. Therefore, it is natural to ask whether performing multiple stochastic gradient updates provides any benefits compared to batch gradient descent. Similar to gradient descent, SGD converges under mild conditions on the loss $\mathcal{L}$ and a suitable learning rate schedule to a stationary point where the gradient vanishes [10]. However, when analyzing the asymptotic behavior of SGD on convex functions, batch gradient descent achieves an optimal convergence rate [10]. Nevertheless, Bottou et al. [10] also show that SGD outperforms batch gradient descent when considering a practical scenario using a fixed time budget. Hence, from a practical viewpoint, the fast progress made by SGD in the beginning of training outweighs the asymptotic properties of batch gradient descent.

In practice, stochastic gradients are rarely obtained by considering only a single sample, but rather using randomly selected subsets of $\mathcal{D}$ containing N_B samples, called mini-batches. Computing the gradients using mini-batches reduces the gradient variance while allowing to fully utilize efficient parallel batch operations of modern hardware. As a result, the computational cost for small mini-batches and per-sample gradients is typically of the same order. Selecting a proper mini-batch size N_B requires a careful trade-off between computational costs and gradient variance. In practice, the size of mini-batches N_B is rarely larger than a few hundred samples. Note that practical considerations regarding hardware efficiency are not considered in theoretical analyses such as [10] which assume that evaluating the exact gradient of $\mathcal{L}$ is N times more expensive than evaluating the per-sample gradients of ℓ_n.

A typical implementation of SGD (see Algorithm 2) repeatedly shuffles the order of the samples

[4] Gradient descent is also called *batch gradient descent* in this context.

Algorithm 3 Stochastic gradient descent (general version)

1: **Input:** Loss $\mathcal{L}(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{\varepsilon}\sim p(\boldsymbol{\varepsilon})}[\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\varepsilon})]$, initial parameters $\boldsymbol{\theta}^0$, learning rates $(\eta_t)_{t\geq 1}$
2: $t \leftarrow 1$
3: **while** stopping criterion not met **do**
4: $\quad \boldsymbol{\varepsilon} \sim p(\boldsymbol{\varepsilon})$
5: $\quad \boldsymbol{\theta}^t \leftarrow \boldsymbol{\theta}^{t-1} - \eta_t \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}^{t-1}, \boldsymbol{\varepsilon})$
6: $\quad t \leftarrow t + 1$
7: **end while**

in $\mathcal{D}$ and then partitions them into mini-batches. Iterating over these mini-batches then ensures that all samples are processed equally often. The common terminology is to call each gradient update an iteration whereas a full pass over the whole dataset is called an *epoch*. An interesting aspect is that during the first epoch, where each sample is processed for the first time, we actually minimize the expected loss $\mathcal{L}_{\text{exp}}$ with respect to the underling data distribution p_{data}. Only after the samples are processed multiple times we can distinguish between empirical and expected loss minimization.

There exists a number of extensions of SGD that aim to improve its practical convergence properties. For instance, by introducing *momentum* to the updates we also take previous gradients into account. The corresponding parameter update in line 7 of Algorithm 2 is replaced by

$$\begin{aligned} \mathbf{v}^t &= \xi_{\text{mom}} \mathbf{v}^{t-1} - \eta_t \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}^{t-1}; \mathcal{D}_k), \\ \boldsymbol{\theta}^t &= \boldsymbol{\theta}^{t-1} + \mathbf{v}^t \end{aligned} \tag{2.11}$$

where $\xi_{\text{mom}} > 0$ is a momentum hyperparameter and $\mathbf{v}^t$ is interpreted as a velocity term. Momentum reduces the influence of gradient variance and stabilizes training. Our experiments in Chapter 5 and Chapter 7 were performed using the stochastic optimization algorithm Adam [11] which maintains running averages of the gradient and the squared gradient entries to compute more sophisticated updates.

We emphasize that SGD is not limited to empirical loss functions defined in terms of per-sample losses ℓ_n. SGD is applicable whenever a loss can be defined as an expectation over a random variable $\boldsymbol{\varepsilon}$, i.e.,

$$\mathcal{L}(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{\varepsilon}\sim p(\boldsymbol{\varepsilon})}\left[\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\varepsilon})\right] \qquad \text{such that} \qquad \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{\varepsilon}\sim p(\boldsymbol{\varepsilon})}\left[\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\varepsilon})\right]. \tag{2.12}$$

This allows us to compute unbiased Monte Carlo estimates of the gradient by sampling $\boldsymbol{\varepsilon} \sim p(\boldsymbol{\varepsilon})$ and evaluating $\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\varepsilon})$. This is shown in Algorithm 3. Note that the empirical loss (2.10) is a specific instance of (2.12) by considering $p(\boldsymbol{\varepsilon})$ as a discrete uniform distribution over the samples of $\mathcal{D}$.

While SGD is applied to empirical losses (2.10) mainly for efficiency purposes, for some applications it allows us to perform gradient-based optimization in the first place. Indeed, the SGD framework allows us to effectively optimize expressions that cannot even be evaluated in closed form. For instance, variational inference, discussed in Section 3.2.3, minimizes the Kullback-Leibler (KL) divergence between two distributions which is given by an expectation. Another example is the induced loss of dropout training which is discussed in Section 2.2.4.

2.1.3 Automatic Differentiation

A computation graph is a graphical representation of a function that allows for the automatic computation of partial derivatives. This is particularly convenient as we only have to specify how a function is computed and we obtain its partial derivatives by an automated process.

This process does not only relieve us from the burden of calculating and implementing the partial derivatives by hand, but computing the gradient does also not take substantially longer than evaluating the function itself. Computation graphs and automatic differentiation are an integral part of modern deep learning frameworks to define loss functions that are subsequently minimized using gradient-based optimization. A comprehensive overview on different techniques and applications that benefit from automatic differentiation is provided in [12].

Computing Derivatives

Before we define computation graphs and automatic differentiation, we give a brief overview of the more conventional methods to compute gradients of an arbitrary differentiable function $\mathcal{L}(\mathbf{u})$.[5] The first approach is simply to derive analytic expressions for the gradient—typically using pen and paper—and to implement these expressions using the programming language at hand. Since this is a tedious and error-prone task, the implementation has to be tested carefully for correctness. This is done by comparing the numerical values to a different way of computing the partial derivatives—usually the *finite difference* approximation that we will describe shortly. If this numerical test fails, there is often a variety of possible causes and one typically does not get a good indication to where the failure stems from. Moreover, when the function $\mathcal{L}$ is changed slightly, all of the above steps have to be performed again which impedes the ability for fast prototyping of new models and functions $\mathcal{L}$.

The second approach, namely the finite difference approximation, essentially approximates the definition of a partial derivative

$$\frac{\partial \mathcal{L}}{\partial u_i}(\mathbf{u}_0) = \lim_{\Delta \to 0} \frac{\mathcal{L}(\mathbf{u}_0 + \Delta \mathbf{e}_i) - \mathcal{L}(\mathbf{u}_0)}{\Delta} \tag{2.13}$$

for some small $\Delta > 0$, where $\mathbf{e}_i$ denotes the ith unit vector. However, computing the finite difference approximation is not straightforward since the choice of a suitable Δ is crucial to obtain good approximations. On the one hand, if Δ is too large, the approximation quality might be poor as (2.13) requires Δ to be small. On the other hand, if Δ is too small, we might run into numerical issues. Furthermore, evaluating the gradient at $\mathbf{u}_0 \in \mathbb{R}^D$ requires $D + 1$ function evaluations which is prohibitive in many cases.

The third approach is called *symbolic differentiation*, where a given expression of $\mathcal{L}$ is differentiated using the laws of differential calculus to obtain a symbolic expression of the partial derivatives $\partial\mathcal{L}/\partial u_i$. This approach is appealing if we are interested in the symbolic expressions themselves. However, the resulting expressions of $\partial\mathcal{L}/\partial u_i$ are often substantially larger than the function $\mathcal{L}$ itself. Therefore, this approach is not recommended if we are only interested in a numerical evaluation of the gradient.

Given that most of these difficulties become obsolete when using automatic differentiation, the ease of developing new models becomes evident. Automatic differentiation is certainly one of the most important factors that have led to the rapid rise and widespread use of deep learning. Although automatic differentiation is by no means a new invention—it essentially performs a systematic computation of the chain rule of differential calculus—, it has only been in the past decade that corresponding tools have attracted much attention in machine learning.

Computation Graphs and Backpropagation

Automatic differentiation operates on a *computation graph* that specifies how to compute a scalar-valued function $\mathcal{L}$. We define a computation graph as a bipartite directed acyclic graph $\mathcal{G}$ where edges appear only between *value nodes* $\mathbf{u}_i$ and *operation nodes* f_i. The value nodes $\mathbf{u}_i$ correspond to actual numerical quantities that appear during the computation of the function.

[5] We use the symbol $\mathcal{L}$ since loss function minimization is the main application in this thesis.

These are the inputs of $\mathcal{L}$, intermediate values that appear during the computation of $\mathcal{L}$, and the scalar-valued output of $\mathcal{L}$. An operation node specifies how to compute its children from its parents. We refrain from using the term function node since we also allow for non-function operations such as stochastic sampling nodes. Note that other definitions of computation graphs are possible, e.g., in [12, 13] there is no clear distinction between operation and value nodes.

Note that the subscripts i of value and operation nodes are used to identify nodes irrespective of their type. We utilize the fact that every directed acyclic graph admits a topological ordering where every parent node appears before their children. In the following, we assume that the nodes are ordered in topological order, i.e., for every parent j of a node i it holds that $j < i$.

The strength of computation graphs is that they allow us to compute the partial derivatives of $\mathcal{L}$ with respect to *all* value nodes $\mathbf{u}_i$ efficiently in an automated manner. The algorithm we are going to present is known as automatic differentiation in the backward mode or, in short, simply backpropagation. The algorithm proceeds in two stages.

The first stage of the algorithm is called the forward pass (or forward propagation) which simply evaluates the function. This is accomplished by traversing the operation nodes in ascending order and computing the corresponding output nodes. Due to the topological ordering, we are guaranteed that the input value nodes have already been computed.

In the second stage, the backward pass (or backward propagation), the algorithm traverses the graph in reverse direction. In this stage, the algorithm systematically applies the chain rule of differential calculus to compute the partial derivatives of $\mathcal{L}$ with respect to all value nodes $\mathbf{u}_i$.

To be self-contained, we briefly review the chain rule of differential calculus here. Consider two functions $f : \mathbb{R}^K \to \mathbb{R}^{K'}$ and $\mathcal{L} : \mathbb{R}^{K'} \to \mathbb{R}$. Using the chain rule, the partial derivatives of $\mathcal{L}(f(\mathbf{u}))$ are given by[6]

$$\frac{\partial \mathcal{L}}{\partial u_i} = \sum_{j=1}^{K'} \frac{\partial \mathcal{L}}{\partial f_j} \frac{\partial f_j}{\partial u_i}. \tag{2.14}$$

By denoting $\mathbf{v} = f(\mathbf{u})$, the gradient of $\mathcal{L}(f(\mathbf{u}))$ can be compactly written as

$$\nabla_{\mathbf{u}} \mathcal{L} = \mathbf{J}_f^\top \nabla_{\mathbf{v}} \mathcal{L}, \tag{2.15}$$

where $\mathbf{J}_f \in \mathbb{R}^{K' \times K}$ denotes the Jacobian matrix of f, i.e.,

$$\mathbf{J}_f = \begin{pmatrix} \frac{\partial f_1}{\partial u_1} & \cdots & \frac{\partial f_1}{\partial u_K} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_{K'}}{\partial u_1} & \cdots & \frac{\partial f_{K'}}{\partial u_K} \end{pmatrix}. \tag{2.16}$$

Note that (2.15) integrates nicely into the computation graph framework by viewing f as a particular operation node and $\mathcal{L}$ being the remainder of the computation graph as a function of f's output $\mathbf{v}$. Therefore, to compute the gradient of $\mathcal{L}$ with respect to f's input $\mathbf{u}$, we require that the gradient with respect to f's output $\mathbf{v}$ has already been computed. Note that this is reminiscent of the forward pass where we require that the inputs of f must have been computed in order to compute the outputs of f.

To obtain a practical algorithm for computation graphs, we need to consider that each value node $\mathbf{u}_i$ might be an input to several operation nodes f_k. Furthermore, each of these operation nodes f_k may itself have multiple output nodes $\mathbf{v}_j$. However, this does not pose any major problems and we can utilize the additive nature of the gradient computation in (2.14). This is accomplished by initializing gradients $\nabla_{\mathbf{u}_i} \mathcal{L}$ with zero and accumulating several Jacobian-gradient products, one per value node $\mathbf{v}_j$ that directly depends on $\mathbf{u}_i$ through some operation

[6] Note that the subscripts in (2.14) correspond to individual dimensions and not to graph node indices.

Algorithm 4 Automatic differentiation

1: **Input:** Computation graph $\mathcal{G}$, assignments of input value nodes $val(\mathbf{u}_i)$
2: $(f_{i_1}, \ldots, f_{i_K}) \leftarrow$ topological ordering of K operation nodes
3: pa $\leftarrow$ parent function (returns parents of a node)
4: ch $\leftarrow$ child function (returns children of a node)
5: # Forward propagation: Compute value nodes $val(\cdot)$
6: **for** $k = 1$ **to** K **do**
7: $\quad \mathbf{u} \leftarrow \text{pa}(f_{i_k})$
8: $\quad$ **for** $\mathbf{v} \in \text{ch}(f_{i_k})$ **do**
9: $\quad\quad val(\mathbf{v}) \leftarrow f_{i_k,\mathbf{v}}(val(\mathbf{u}))$
10: $\quad$ **end for**
11: **end for**
12: # Backward propagation: Compute gradients $\delta(\cdot)$
13: $\delta(\mathbf{u}_i) \leftarrow 0 \qquad \forall i : \mathbf{u}_i$ value node
14: $\delta(\mathcal{L}) \leftarrow 1$
15: **for** $k = K$ **to** 1 **do**
16: $\quad$ **for** $\mathbf{v} \in \text{ch}(f_{i_k})$ **do**
17: $\quad\quad$ **for** $\mathbf{u} \in \text{pa}(f_{i_k})$ **do**
18: $\quad\quad\quad \delta(\mathbf{u}) \leftarrow \delta(\mathbf{u}) + \mathbf{J}^{\top}_{f_{i_k},\mathbf{u},\mathbf{v}}(val(\text{pa}(f_{i_k})))\,\delta(\mathbf{v})$
19: $\quad\quad$ **end for**
20: $\quad$ **end for**
21: **end for**
22: **return** δ

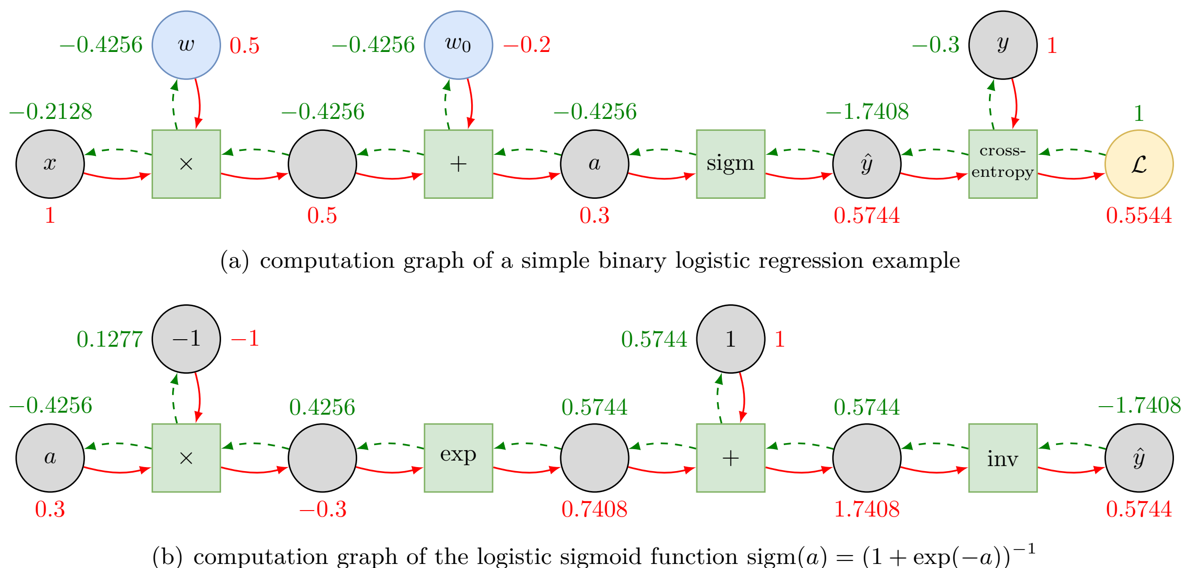
node f_k. Once every operation node f_k depending on $\mathbf{u}_i$ has been processed, the accumulated gradient contributions contain the correct gradient $\nabla_{\mathbf{u}_i}\mathcal{L}$.

Starting from the output node and setting $\partial\mathcal{L}/\partial\mathcal{L} = 1$, the backward pass traverses the operation nodes in descending order. At each operation node f_k, the gradient contributions from the Jacobian-gradient products are added to all of its inputs $\mathbf{u}_i$. Due to the topological ordering, we are guaranteed that the gradients $\nabla_{\mathbf{v}_j}\mathcal{L}$ for all children of f_k contain valid information. Once all operation nodes have been visited, the algorithm has correctly computed the gradient of $\mathcal{L}$ with respect to all value nodes $\mathbf{u}_i$. The full algorithm, forward and backward pass, is shown in Algorithm 4. The pseudocode introduces some new notations, i.e., ch(f) returns the children of node f, pa(f) returns the parents of node f, $f_{\mathbf{v}}$ computes the value of f's child $\mathbf{v}$ (f may have other children as well), and $\mathbf{J}_{f,\mathbf{u},\mathbf{v}}$ is the Jacobian of f restricted to its parent $\mathbf{u}$ and child $\mathbf{v}$.

There are some aspects that need to be considered for a practical algorithm. Note that for the forward pass, intermediate values can be discarded as soon as every operation nodes depending on them have been processed. However, when we intend to perform the backward pass, all of these values must be kept in memory as they are required to compute the Jacobians $\mathbf{J}_f$. This additional bookkeeping might pose a severe limitation due to a memory bottleneck. This problem can be mitigated by not keeping all value nodes in memory and recomputing value nodes on demand, effectively resulting in a trade-off between memory consumption and running time.

Furthermore, while we have so far only assumed vector-valued nodes, in practice we often encounter value nodes that contain matrix-valued or, more generally, tensor-valued nodes. In this case it is best thought of as temporarily reshaping any tensor-valued node $\mathbf{u}_i$ into a vector and restoring its original shape once processing has finished.

Another important point is that the algorithm only requires the *product* of the Jacobian $\mathbf{J}_f$ with another vector and does not require that the Jacobian $\mathbf{J}_f$ is ever explicitly computed. In fact, the Jacobian $\mathbf{J}_f$ typically exhibits a lot of structure and it would be wasteful to compute it explicitly. For instance, for an operation node $f(\mathbf{U}_i, \mathbf{U}_j) = \mathbf{U}_i\mathbf{U}_j = \mathbf{V}$ computing a matrix



(a) computation graph of a simple binary logistic regression example

(b) computation graph of the logistic sigmoid function $\mathrm{sigm}(a) = (1 + \exp(-a))^{-1}$

Figure 2.1: Backpropagation example. Value nodes are depicted as circles whereas operation nodes are depicted as boxes. The values computed in the forward path (red arrows) are shown in red next to the value nodes. The partial derivatives computed in the backward path (green dashed arrows) are shown in green next to the value nodes. (a) The loss of a simple binary logistic regression example (yellow node) is computed. The blue nodes indicate the parameters that are updated using the partial derivatives. (b) Unrolled version of the logistic sigmoid from (a).

multiplication, each entry of the output matrix $\mathbf{V}$ only depends on a single row of $\mathbf{U}_i$ and a single column of $\mathbf{U}_j$. As a result, the Jacobian $\mathbf{J}_f$ is highly sparse. Using $\mathbf{G} = \nabla_{\mathbf{V}}\mathcal{L}$, the Jacobian-gradient product with respect to $\mathbf{U}_i$ is given by $\mathbf{G}\mathbf{U}_j^\top$ and, similarly, with respect to $\mathbf{U}_j$ it is given by $\mathbf{U}_i^\top\mathbf{G}$. Another example are operation nodes computing an element-wise function $f(\mathbf{u}_i) = \mathbf{v}_j$ for which the Jacobian $\mathbf{J}_f$ is diagonal. In this case, it suffices to compute the element-wise product of the element-wise derivatives $f'(\mathbf{u}_i)$ with the gradient $\nabla_{\mathbf{v}_j}\mathcal{L}$.

Figure 2.1 illustrates automatic differentiation applied to a simple binary logistic regression example. The function defined by the computation graph in Figure 2.1(a) first computes $\hat{y} = \mathrm{sigm}(wx + w_0)$, where $\mathrm{sigm}(a) = (1 + \exp(-a))^{-1}$ is the logistic sigmoid. Then, the binary cross-entropy $\mathcal{L}(\hat{y}, y) = -y\log(\hat{y}) - (1-y)\log(1-\hat{y})$ for a given target $y \in \{0, 1\}$ is computed.

During forward propagation (red path), the values of intermediate nodes (red numbers) are computed until the loss function node $\mathcal{L}$ is reached. For backward propagation (green dashed path), the partial derivative $\partial\mathcal{L}/\partial\mathcal{L}$ is initialized to 1. At each operation node, the partial derivatives of its parents are computed. For instance, the partial derivative $\partial\mathcal{L}/\partial a$ is computed as $\mathrm{sigm}'(a) \cdot \partial\mathcal{L}/\partial\hat{y}$ where $\mathrm{sigm}'(a) = \mathrm{sigm}(a) \cdot (1 - \mathrm{sigm}(a))$.

The computation of $\mathrm{sigm}(a)$ can also be unrolled using the definition of the logistic sigmoid as shown in Figure 2.1(b). However, since for backpropagation all intermediate values need to be kept in memory, it is more efficient to use concise graph structures if possible. Moreover, we are mostly not interested in the partial derivatives of intermediate nodes. In fact, the graph in Figure 2.1(a) could be condensed by fusing the cross-entropy loss of the logistic sigmoid into a single operation $\mathcal{L}(a, y)$, which results in the convenient partial derivatives $\partial\mathcal{L}/\partial a = \mathrm{sigm}(a) - y$.

This example shows that the efficiency of the algorithm depends on the graph $\mathcal{G}$ itself. For instance, it would be highly inefficient to define the logistic sigmoid as in Figure 2.1(b). First, we would need to keep track of many unnecessary intermediate results and, second, we would not exploit the fact that the derivative of $\mathrm{sigm}(a)$ has a simple form. Although there exist algorithms that try to detect patterns in the graph in order to perform some sort of simplifications, it is advisable to incorporate as much prior knowledge as possible into the definition of the graph and its building blocks for backpropagation to be efficient.

Many modern automatic differentiation tools do not require the user to specify the computation graph explicitly. These tools build the graph implicitly by tracking computations which further improves their ease of use. In accordance with the DNN literature, in the remainder of this thesis we will call the automatic differentiation algorithm in backward mode simply the backpropagation algorithm. In the next section, we will discuss how the backpropagation algorithm can be used in the presence of operation nodes whose gradient is zero almost everywhere or not even defined. We will also come back to automatic differentiation in the context of the reparameterization trick in Section 3.4.4—a technique that rewrites stochastic sampling operations in a way that allows us to compute their gradient.

2.1.4 The Straight-Through Gradient Estimator

In our discussion on automatic differentiation we have silently assumed that the gradient of operation nodes exists and that it is non-zero. If an operation node $f(\mathbf{u})$ does not have a valid non-zero gradient, its parents $\mathbf{u}$ will not receive a gradient contribution from f during backpropagation. As a result, $\mathbf{u}$ cannot be updated via gradient-based learning unless it receives a gradient contribution from a different path in the computation graph. Even worse, if an operation is not differentiable, Algorithm 4 is not even well-defined.

The question arises whether backpropagation can still be used in the presence of such operations since this would open the door for many interesting applications (see below). The STE [14] provides a simple, yet effective solution to obtain approximate non-zero gradients. For simplicity, let $v = f(u)$ be an operation that takes a scalar input u and whose derivative is zero or undefined. The STE then approximates the partial derivative $\partial \mathcal{L} / \partial u$ by

$$\frac{\partial \mathcal{L}}{\partial u} = \frac{\partial \mathcal{L}}{\partial v} \frac{\partial f}{\partial u} \approx \frac{\partial \mathcal{L}}{\partial v} \frac{\partial \tilde{f}}{\partial u}, \tag{2.17}$$

where $\tilde{f}$ is an approximation to f that has a non-zero gradient. To put it more generally, let $\mathbf{v} = f(\mathbf{u})$ and $\mathcal{L}(f(\mathbf{u}))$ be as in (2.15). Furthermore, let $\tilde{f}$ be a function with non-zero gradient that approximates f. The STE then approximates the Jacobian-gradient product by

$$\nabla_{\mathbf{u}} \mathcal{L} = \mathbf{J}_f^\top \nabla_{\mathbf{v}} \mathcal{L} \approx \mathbf{J}_{\tilde{f}}^\top \nabla_{\mathbf{v}} \mathcal{L}. \tag{2.18}$$

This allows us to compute approximate non-zero gradients with respect to the parents of f, enabling gradient-based learning. Figure 2.2(a) illustrates the STE in terms of a computation graph.

The success of the STE depends crucially on the selected approximation $\tilde{f}$. It is important that f and $\tilde{f}$ exhibit a similar functional shape. In many cases, the identity function $\tilde{f}(\mathbf{u}) = \mathbf{u}$ is a reasonable choice. Indeed, the term STE stems from the particular choice of the identity function for which the gradient is simply passed "straight through" during backpropagation.

There are many interesting applications that arise in practice, most of which require some sort of discretization or stochasticity. As shown in Figure 2.2(b), the identity function can be seen as an approximation to staircase functions. Such staircase functions are heavily used as quantizers in deep learning (see Section 4.1.2).

For the piecewise constant sign function (see Figure 2.2(c)), the hyperbolic tangent (tanh) is a commonly used approximation. In some cases, a stochastic sign function given by

$$\operatorname{sign}_{\text{stoch}}(u) = \begin{cases} 1 & \text{if } \varepsilon \leq (1+u)/2 \\ -1 & \text{otherwise,} \end{cases} \tag{2.19}$$

yields better results. Here, $\varepsilon \sim \mathcal{U}([0,1])$ is drawn from a uniform distribution. Note that, due to its stochasticity, (2.19) is not even a function, but again $\tilde{f} = \tanh$ provides a suitable

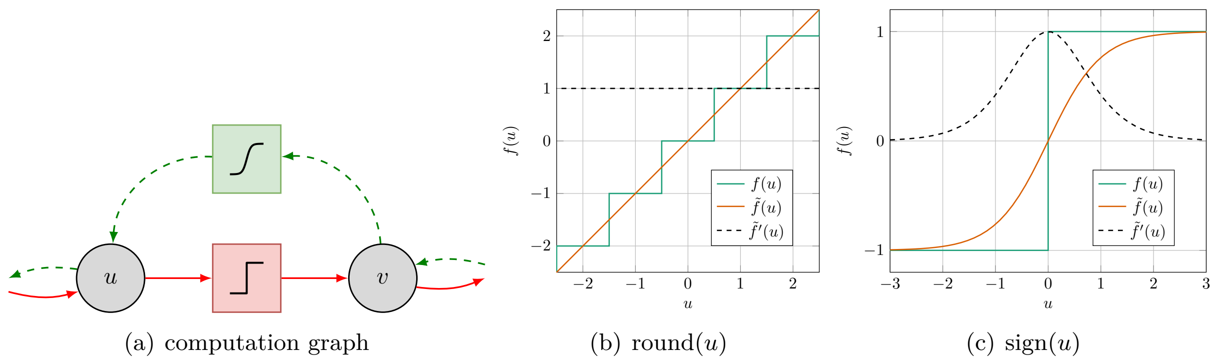


Figure 2.2: STE: (a) Computation graph: Green boxes indicate differentiable operations. Red boxes indicate piecewise constant functions whose derivative is zero almost everywhere. During forward propagation, the red path is followed. During backward propagation, the red boxes are avoided by following the green path. (b) The zero derivative of the staircase function $f(u)$ is approximated by the gradient of the identity function $\tilde{f}(u)$. (c) The zero derivative of the sign function $f(u)$ is approximated by the derivative of $\tilde{f}(u) = \tanh(u)$.

approximation for its derivative. The sign function is commonly used as a binary activation function in DNNs (see Section 4.1.2). Other applications of binary variables include binary gating functions and binary representations that enable a fast search in suitable data structures (e.g., semantic hashing [15]).

Consider the argmax function defined on $\mathbf{u} \in \mathbb{R}^D$ that computes a one-hot vector $\mathbf{e}_i$ as

$$\operatorname{argmax}(\mathbf{u}) = \mathbf{e}_i \quad \iff \quad u_i = \max\{u_1, \dots, u_D\}. \tag{2.20}$$

A differentiable approximation is obtained by the softmax function

$$\operatorname{softmax}(\mathbf{u}) = \left(\frac{\exp(u_1)}{\sum_{i=1}^{D} \exp(u_i)}, \dots, \frac{\exp(u_D)}{\sum_{i=1}^{D} \exp(u_i)} \right). \tag{2.21}$$

The one-hot version of the argmax is a convenient gating function to select among multiple options. For instance, we employ the argmax function in our structure learning approach in Chapter 7 to select a suitable parent in BNs.

These examples show that the STE is commonly used for various kinds of discretizations. We emphasize that applying the STE is fundamentally different from training with the differentiable approximation $\tilde{f}$ and using f only after training has finished (e.g., performing quantization as a post-processing step). In many cases, this would cause a substantial mismatch between the model behavior at training and test-time. Therefore, we typically expect the STE to yield improved results. Note that recent works provide more insights into the STE, e.g., [16].

The STE has been heavily used in the deep learning literature. In Chapter 4 we will encounter several works relying on the STE to improve resource efficiency in DNNs. Moreover, the STE is an important component of our proposed methods in Chapter 7 for structure learning and quantization in BN classifiers.

2.2 Feed-Forward Deep Neural Networks

DNNs are a class of function approximators that are heavily used for supervised learning. We start our discussion with the basic principles of DNNs and provide some intuitions about their inner workings. Then, we formally introduce the basic building blocks, linear operations and nonlinear activations, before we turn to more recent concepts such as dropout and batch nor-

malization. Finally, we end our discussion with a brief overview of the progression of DNN architecture trends in the past decade.

2.2.1 The Basic Layout of Deep Neural Networks

A DNN is typically organized as a sequence of layers. Each layer takes the output of its preceding layer as input which is then processed to generate the input for the subsequent layer. More formally, a vanilla feed-forward DNN with L layers is a function that maps an input $\mathbf{x}^0$ to an output $\mathbf{x}^L$ by applying the iterative computation

$$\mathbf{a}^l = \mathbf{W}^l \otimes \mathbf{x}^{l-1} + \mathbf{w}_0^l, \tag{2.22}$$

$$\mathbf{x}^l = h^l(\mathbf{a}^l). \tag{2.23}$$

Here, $\otimes$ denotes a linear operation governed by *weights* $\mathbf{W}^l$, $\mathbf{w}_0^l$ is a *bias* vector, and h^l is a nonlinear activation function. We refer to $\mathbf{a}^l$ as the *activations* and to $\mathbf{x}^l$ as the layer's *output*. Individual dimensions i of a layer are commonly referred to as *neurons*, irrespective of their role as activation a_i^l or layer output x_i^l. The first layer $l = 1$ is called *input layer* and the last layer L is called *output layer*. Intermediate layers $1 \leq l < L$ are commonly referred to as *hidden layers*.

The parameters $\boldsymbol{\theta}$ of a vanilla DNN are given by the set of all weights $\mathbf{W} = (\mathbf{W}^1, \ldots, \mathbf{W}^L)$ and the set of all bias vectors $\mathbf{w}_0 = (\mathbf{w}_0^1, \ldots, \mathbf{w}_0^L)$. Nevertheless, the parameters of a DNN are often collectively referred to as weights which comprises both the weights $\mathbf{W}$ *and* the bias vectors $\mathbf{w}_0$. If a distinction between weights and bias vectors is important, this will be explicitly mentioned. Throughout this thesis we denote individual weights by lowercase $w \in \mathbf{W}$.

There is some debate about when a neural network is entitled to carry the attribute *deep*, especially since we have observed a trend towards ever increasing numbers of layers and the fact that depth is somewhat relative. Nevertheless, throughout this thesis we will stick to the most common convention, i.e., it holds that $L > 1$.

DNNs are *universal function approximators*, meaning that any continuous function can be approximated arbitrarily well with a DNN, given a sufficient number of neurons [8]. Interestingly, this also holds for only a single hidden layer ($L > 1$), but recent advances in deep learning indicate that deeper architectures achieve this using fewer neurons in total. By inspecting (2.22) and (2.23), it is evident that the expressive power of a DNN stems from its nonlinear activation function h^l. Surprisingly, very simple nonlinearities are sufficient (see below).

Before we delve into the details and explain the individual parts of a DNN, we want to give some intuition of what a DNN computes. On a high level, it is intuitive to think of a DNN as a feature extractor where different layers detect features at varying abstraction levels. More specifically, each neuron computes a numerical value representing the presence or absence of a specific property in the input $\mathbf{x}^0$. While the first layer of a DNN computes values based on the raw input features $\mathbf{x}^0$, subsequent layers have already access to numerical values $\mathbf{x}^{l-1}$ computed in a very specific way by the preceding layers. Therefore, we expect that the early layers of a DNN detect low-level features whereas neurons in deeper layers detect high-level features.

To what particular feature a neuron really corresponds is in general difficult to answer. It is perhaps also too simplistic to think that each neuron corresponds to a single feature. It is more realistic that several neurons compute several features in an intertwined way. This property makes DNNs hard to interpret which is one of their major weaknesses that must be considered when deploying DNNs in real-world applications. For instance, in a medical application it is very unlikely that DNNs will be part of a fully automatic system in the near future whereas it is conceivable that a DNN is used to assist experts in making their decisions. However, there is currently a substantial amount of research concerned with the interpretation of DNNs and developing more interpretable DNN architectures in the first place.

Activation Functions

Common nonlinear activation functions $h^l(a)$ of hidden layers are the ReLU function $\max(a, 0)$ and sigmoid functions such as $\tanh(a) = (\exp(a) - \exp(-a))/(\exp(a) + \exp(-a))$ and the logistic sigmoid $\operatorname{sigm}(a) = (1 + \exp(-a))^{-1}$. In the context of resource-efficient DNNs, the sign function $\operatorname{sign}(a) = \mathbb{I}[a \geq 0] - \mathbb{I}[a < 0]$ and the Heaviside step function $\mathbb{I}[a \geq 0]$ are widely used activation functions.

Output Activation Functions

The activation function h^L of the output layer depends on the type of prediction task. For regression, the output activation function is the identity $h^L(\mathbf{a}) = \mathbf{a}$, for binary classification, it is the logistic sigmoid $h^L(a) = \operatorname{sigm}(a)$, and for multiclass classification, it is the softmax function $h^L(\mathbf{a}) = \operatorname{softmax}(\mathbf{a})$ as defined in (2.21). These particular output activation functions are convenient since they allow us to interpret the DNN output $\mathbf{x}^L$ as parameters of a conditional distribution $p(\mathbf{y} \,|\, \mathbf{x}^0)$ that is used to define a probabilistic predictor. In particular, for regression, the outputs are typically assumed to be normally distributed with mean $\mathbf{x}^L$ and fixed (or homoscedastic) variance β^2, i.e., $\mathbf{y} \sim \mathcal{N}(\mathbf{x}^L, \mathbf{I}\beta^2)$ where $\mathbf{I}$ is the identity matrix. To obtain input-dependent (or heteroscedastic) variance $\beta^2(\mathbf{x}^0)$, it is common to introduce additional output neurons that explicitly model the variance.[7] For binary classification, the output of the logistic sigmoid, $x_i^L \in [0, 1]$, is interpreted as $x_i^L = p(Y_i = 1 \,|\, \mathbf{x}^0)$. For multiclass classification, the softmax output $\mathbf{x}^L$ is a probability vector (i.e., $x_i^L \geq 0$ and $\sum_i x_i^L = 1$) that is interpreted as $x_i^L = p(Y = i \,|\, \mathbf{x}^0)$. In either case, the predicted value of the DNN is computed as the most probable value according to

$$\hat{f}(\mathbf{x}^0) = \operatorname*{argmax}_{\mathbf{y}} p(\mathbf{y} \,|\, \mathbf{x}^0). \tag{2.24}$$

We emphasize that one needs to be cautious when interpreting the conditional probabilities $p(\mathbf{y} \,|\, \mathbf{x}^0)$ as prediction confidences. These probabilities are often spurious and do not provide meaningful uncertainties for inputs $\mathbf{x}^0$ that are not close to the training data. Better prediction uncertainties are obtained with the Bayesian framework discussed in Chapter 3.

Fully Connected Layers

There are essentially two types of linear transformations $\otimes$, namely (i) matrix-vector multiplications and (ii) linear convolutions. In case of a matrix-vector multiplication, the corresponding layer is also called a *fully connected layer* since every neuron x_i^l depends on every neuron x_j^{l-1} from the previous layer. This is illustrated in Figure 2.3(a).

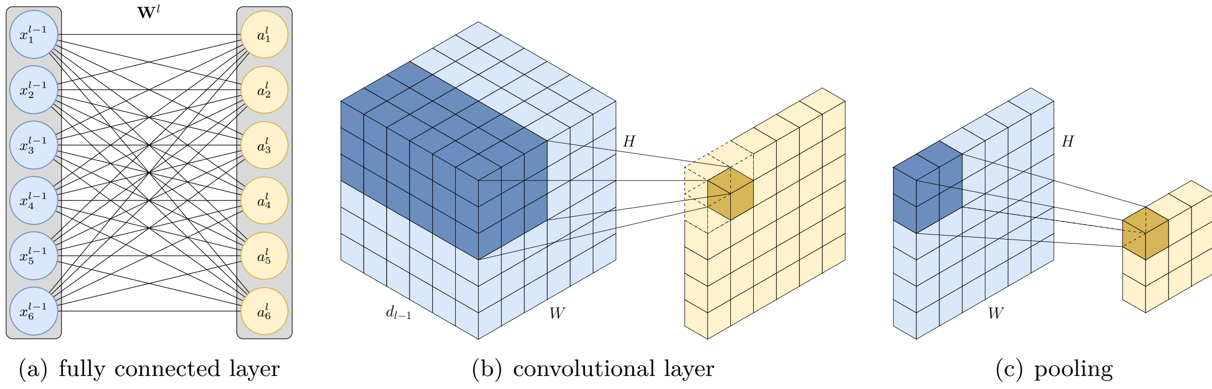
The linear transformation of layer l is determined by a weight matrix $\mathbf{W}^l \in \mathbb{R}^{d_l \times d_{l-1}}$ which transforms a d_{l-1}-dimensional vector into a d_l-dimensional vector. The number of dimensions d_l are tunable hyperparameters that determine the structure of the DNN.

Fully connected layers are typically used when the inputs $\mathbf{x}^{l-1}$ do not exhibit any a priori known structure among the individual dimensions that can be easily exploited. Fully connected layers are sometimes called *dense layers* due to the generally dense (i.e., non-sparse) matrix structure of $\mathbf{W}^l$.

Convolutional Layers

A layer that computes a linear convolution is called a *convolutional layer*, and we call a DNN that employs at least one convolutional layer a convolutional neural network (CNN). Convolutions

[7] As we will see in Chapter 3, heteroscedastic variance can also be achieved through parameter uncertainty in the Bayesian framework.



(a) fully connected layer (b) convolutional layer (c) pooling

Figure 2.3: DNN building blocks. (a) In a fully connected layer, every input neuron x_j^{l-1} is connected to every output neuron a_i^l via a weight $w_{i,j}^l$. (b) 3×3 convolution. Only a single output feature map is shown. The spatial locations in the output are computed by shifting the 3×3 kernel over the image and computing a weighted sum over the inputs. (c) 2×2 pooling to downscale the spatial dimensions.

are used if the data exhibits spatial dimensions, temporal dimensions, or both, such as image, audio, or video data, respectively.

In this thesis, our discussion is restricted to two-dimensional image data. Two-dimensional images can be represented as three-dimensional tensors $\mathbf{x}^l \in \mathbb{R}^{d_l \times W \times H}$ where d_l refers to the number of *channels* (or, equivalently, *feature maps*), and W and H refer to the width and the height of the image, respectively.

A $K_w \times K_h$ convolution $\mathbf{W}^l * \mathbf{x}^{l-1}$ using a rank-4 weight tensor $\mathbf{W}^l \in \mathbb{R}^{K_w \times K_h \times d_{l-1} \times d_l}$ mapping $\mathbf{x}^{l-1} \in \mathbb{R}^{d_{l-1} \times W \times H}$ to $\mathbf{a}^l \in \mathbb{R}^{d_l \times W \times H}$ is computed as

$$a_{i,w,h}^l = \sum_{k_w=1}^{K_w} \sum_{k_h=1}^{K_h} \sum_{j=1}^{d_{l-1}} w_{k_w,k_h,j,i}^l \cdot x_{j,\mathrm{idx}(w,k_w,K_w),\mathrm{idx}(h,k_h,K_h)}^{l-1}, \tag{2.25}$$

where idx is the auxiliary indexing function

$$\mathrm{idx}(pos, k, K) = pos - \left\lceil \frac{K}{2} \right\rceil + k. \tag{2.26}$$

This is illustrated in Figure 2.3(b). The bias $\mathbf{w}_0^l$ in (2.22) is typically added per channel, i.e., $\mathbf{w}_0^l \in \mathbb{R}^{d_l}$. For convolutional layers, the weight tensor $\mathbf{W}^l$ is often called *filter kernel*, and each sub-kernel $\mathbf{W}_i^l \in \mathbb{R}^{K_w \times K_h \times d_{l-1}}$ is referred to as a filter. The activations at each spatial location of $\mathbf{a}^l$ are computed from a spatial region of size $K_w \times K_h$ from the input feature maps $\mathbf{x}^{l-1}$. Since the same filter is used to compute the activations at different spatial locations, a translation invariant detection of features is obtained.

The spatial size of features detected within an image is bounded by the *receptive field*, i.e., the section of the input image $\mathbf{x}^0$ that influences the value of a particular spatial location in some hidden layer. For instance, the receptive field of $x_{i,w,h}^1$ in the first hidden layer after applying a 3×3 convolution is the 3×3 pixel section of $\mathbf{x}^0$ centered at (w, h). Note that the receptive field increases by stacking multiple convolutional layers, e.g., performing two consecutive 3×3 convolutions results in each output spatial location being influenced by a larger 5×5 region of the input. This leads to the intuition that CNNs detect low-level features (e.g., edges and corners) in early layers, while high-level features (e.g., objects) are detected in deeper layers.

Another form of translational invariance is achieved by *pooling* operations that downscale the spatial dimensions by merging neighboring locations within a feature map. The most common pooling operations are max pooling and average pooling which combine neighboring neurons by

computing their maximum or average, respectively. This is illustrated in Figure 2.3(c). The regions subject to the pooling operation are typically small (2×2 or 3×3) and non-overlapping, but there is no general restriction on how to select them. Note that pooling operations also increase the receptive field.

Computational Costs of Linear Operations

The computational costs of fully connected layers and convolutional layers are different. Computing the output of a fully connected layer (i.e., a matrix-vector product) requires $d_{l-1}d_l$ (multiply-accumulate) operations. For the analysis of convolutional layers we assume quadratic filters of size $K \times K$. Computing a single output feature map of a convolutional layer requires the computation of WH spatial locations, each of which requires $K^2 d_{l-1}$ operations. Since there are d_l output feature maps, computing the full output requires $WHK^2 d_{l-1} d_l$ operations. Therefore, convolutions are considered to be memory efficient since they compute many operations for a relatively small number of weights (i.e., $K^2 d_{l-1} d_l$).

2.2.2 Training Deep Neural Networks

The weights of a DNN are typically trained using SGD as discussed in Section 2.1.2. As we have seen in Section 2.1.3, the required gradient $\nabla_{\mathbf{W}} \mathcal{L}$ can be computed by specifying a computation graph and performing automatic differentiation. This is convenient since the computation graph of a DNN defined by (2.22) and (2.23) is a simple chain graph.

One key factor that has led to the widespread use of DNNs is their simple structure. The bulk of operations to evaluate DNNs are performed by the linear operations which can be massively parallelized. Using dedicated parallel hardware such as GPUs and TPUs, this allows for the training of large DNN architectures on large-scale datasets.

However, despite modern hardware capabilities, DNN training still remains a very challenging optimization task. It is not uncommon that DNN training takes on the order of days or even weeks [6]. In the following, we explain several difficulties that arise during DNN training, before we proceed with common solutions to these problems.

Vanishing Gradients

When training very deep architectures, the *vanishing gradient problem* might cause gradients to be very small such that optimization makes insufficient progress. To understand the vanishing gradient problem, it is convenient to consider DNNs that employ identity activation functions $h(\mathbf{a}) = \mathbf{a}$. By ignoring the biases $\mathbf{w}_0$, the resulting output $\mathbf{x}^L$ factorizes into a product of matrices

$$\mathbf{x}^L = \mathbf{W}^L \cdots \mathbf{W}^1 \mathbf{x}^0. \tag{2.27}$$

If we now assume that the eigenvalues of each $\mathbf{W}^l$ are less than $\alpha < 1$, the layer outputs $\mathbf{x}^l$ are exponentially scaled by α^l. As a result, the gradients $\nabla_{\mathbf{W}^l} \mathcal{L}$ for the early layers will be downscaled by α^{L-l}, preventing these layers from receiving significant updates during training. Although being a simplified view of general DNNs, these observations translate to general nonlinear activation functions h. In fact, for many commonly used activation functions these shrinking effects are even amplified. For instance, bounded activation functions, such as tanh and sigm, shrink their inputs and their derivatives are small.

A similar problem arises when all the eigenvalues are larger than $\alpha > 1$. This case is referred to as the *exploding gradient problem*. However, exploding gradients are more relevant for recurrent neural networks (RNNs) which we do not consider in this thesis.

Flat Regions and Dead Neurons

Many commonly used activation functions suffer from saturation effects causing very small gradients. For instance, the commonly used tanh activation function exhibits substantial variation around zero, but flattens out for inputs of large magnitude where the derivative approaches zero. Consequently, it would take an impractical number of update steps to escape from saturated regions due to tiny gradients.

Unbounded activation functions such as the ReLU do not suffer from this problem. On the downside, the ReLU function may suffer from the problem of *dead neurons*. This phenomenon arises when a particular activation a_i^l is negative for all data samples. This causes the derivative $\partial\mathcal{L}/\partial a_i^l$ to be zero such that the gradient with respect to all incoming weights $w_{i,j}^l$ is also zero. For the first layer, these neurons are considered *dead* since the incoming weights remain unchanged forever. For intermediate layers $l > 1$, the situation might change depending on weight updates applied to preceding layers. In any case, this is an unsatisfying state. Dead neurons can be avoided by the *leaky* ReLU activation function $\max(a, \alpha a)$ for $\alpha \in (0, 1)$. The leaky ReLU enables gradient flow for negative arguments while still exhibiting sufficient nonlinearity.

Local Optima and Saddle Points

A major issue when minimizing non-convex functions is getting stuck in bad local optima and saddle points. However, this seems to be less of a problem when training DNNs. Although the number of local optima is, due to weight space symmetries in DNNs, extremely large or even uncountable, it is believed that at least for large DNN architectures most local optima are not severely inferior to global optima [13]. Furthermore, typical SGD algorithms are not attracted to saddle points. Optimization algorithms that are attracted to saddle points are typically second-order methods such as Newton's method. Second-order methods, however, are rarely used for DNN training since they have larger memory requirements and need larger mini-batch sizes for stochastic optimization.

Weight Initialization

Considering the number of difficulties that may arise during training, it is interesting that most of them can be sufficiently solved by choosing appropriate initial weights. One role of initialization is *symmetry breaking*, meaning that each neuron should compute a different function when training starts. This is important since otherwise the resulting neurons would receive the same gradient updates and continue to compute the same functions throughout the optimization process. Consequently, initializing all weights to a constant value such as zero is not an option.

To achieve optimal symmetry breaking, the weight matrices must be orthogonal such that the detected features are as diverse as possible. Weight initialization based on orthogonal matrices has been studied in [17, 18]. However, generating orthogonal matrices is computationally expensive for large weight matrices. In practice, it typically suffices to sample the initial weights from a zero-mean distribution, resulting in approximately orthogonal matrices.

The specific form of the distribution used to initialize the weights (e.g., Gaussian or uniform) often does not have a severe impact on training. However, the variance of the selected distribution is typically crucial. On the one hand, large initial weights are beneficial for symmetry breaking but often result in saturated neurons or exploding gradients. On the other hand, small initial weights are prone to vanishing gradients but cause neurons to operate in the linear range (i.e., many sigmoid functions behave linearly in the vicinity of zero where learning is known to make fast progress). Therefore, a carefully chosen trade-off is necessary.

One widely used initialization scheme is the *Xavier initialization* [19]. The idea is to initialize the weights such that the activations in each layer approximately maintain their scale. As a consequence, we expect neither vanishing gradients nor exploding gradients to be an issue at the

beginning of optimization. Assume that the weights are initialized according to $w^l \sim \mathcal{N}(0, \sigma^2)$ and assume that the magnitude of the entries of $\mathbf{x}^{l-1}$ are on the order of one. Then, after the linear transformation using $\mathbf{W}^l$, the magnitude of the entries in $\mathbf{a}^l$ will be on the order of $\sqrt{d_{l-1}} \cdot \sigma$ where d_{l-1} is the number of incoming connections (or *fan-in* in this context).[8] To maintain equal activation scale, it suffices to select the weights according to

$$w^l \sim \mathcal{N}\left(0, \frac{1}{d_{l-1}}\right). \tag{2.28}$$

Since the above argument does not rely on a specific distribution but only on its variance, it is straightforward to extend these ideas to other zero-mean distributions by adjusting the variance accordingly. For instance, it is common to use a uniform distribution for weight initialization according to

$$w^l \sim \mathcal{U}\left(\left[-\sqrt{\frac{3}{d_{l-1}}}, \sqrt{\frac{3}{d_{l-1}}}\right]\right). \tag{2.29}$$

The initialization schemes (2.28) and (2.29) only consider forward propagation. It is also possible to incorporate approximate equality of gradient scale during backpropagation by replacing d_{l-1} with $(d_{l-1} + d_l)/2$, i.e.,

$$w^l \sim \mathcal{N}\left(0, \frac{2}{d_{l-1} + d_l}\right). \tag{2.30}$$

and

$$w^l \sim \mathcal{U}\left(\left[-\sqrt{\frac{6}{d_{l-1} + d_l}}, \sqrt{\frac{6}{d_{l-1} + d_l}}\right]\right). \tag{2.31}$$

Until now we have effectively ignored the nonlinear activation function in the ongoing discussion. It turns out that these initialization methods are particularly effective for symmetric activation functions that behave linearly around zero, such as sigm and tanh. The situation is slightly different for the ReLU which essentially removes any variance for negative activations. The *He initialization* [20] takes the ReLU activation function into account by doubling the respective variances of (2.28) and (2.29), i.e.,

$$w^l \sim \mathcal{N}\left(0, \frac{2}{d_{l-1}}\right) \tag{2.32}$$

and

$$w^l \sim \mathcal{U}\left(\left[-\sqrt{\frac{6}{d_{l-1}}}, \sqrt{\frac{6}{d_{l-1}}}\right]\right). \tag{2.33}$$

The biases $\mathbf{w}_0^l$ are typically initialized to constant values. Common values are zero or, in conjunction with the ReLU activation, small positive values to avoid dead neurons. For classification, the biases of the output layers can be used to compensate for a known class imbalance. For multiclass classification, this is achieved by setting $w_{0,i}^L = \log p(Y = i)$. For binary classification, this is achieved by setting the bias to the logits, i.e., $w_0^L = \log p(Y = 1)/p(Y = 0)$.

Another initialization approach is to extract knowledge from an existing pre-trained model. For instance, the low-level features detected by a pre-trained DNN might also be useful in a different context. This suggests to reuse the early layers of a pre-trained DNN and to randomly initialize the deeper layers. This is a particular instance of *transfer learning* [13] which is

[8] For convolutions the fan-in is given by $K_w K_h d_{l-1}$.

interesting if data for a given task is scarce, but data for a related task is abundant (e.g., pre-training on ImageNet [21]). Another common approach is to use *autoencoders* [22] for pre-training. Autoencoders are DNNs that are trained to predict their own inputs. By introducing a low-dimensional bottleneck layer, the input must be transformed into a concise representation that contains enough information to reconstruct it again. It is then expected that the early layers detect meaningful features that can be reused in a different context. Autoencoders are appealing since they allow us to perform pre-training using unlabeled data.

As we will see in Chapter 5, parameter initialization is a crucial component of our probabilistic method for training discrete-valued DNNs. Our method initializes weight distributions using the weights of a pre-trained model. This is interesting since the initialized parameters (probabilities) have a different semantics than the parameters used for initialization (fixed weights).

While initialization methods establish certain favorable properties that hold for the initial weights, it is not guaranteed that these properties persist throughout the course of training. In Section 2.2.3 we introduce *batch normalization*, a special building block that maintains such properties by reducing interactions between layers.

Optimization-Friendly Architectures

Despite several efforts to improve initialization methods and optimization techniques for the training of DNNs, the most progress has arguably been made by developing DNN architectures and building blocks that facilitate optimization. For instance, the nowadays commonly used ReLU activation function did not emerge due to some underlying theoretical or biological principles, but rather due to its appealing computational properties. In particular, the ReLU function is easy to compute and does not suffer from the above-mentioned disadvantages of sigmoid activation functions regarding vanishing gradients.

The vanishing gradient problem is still a common challenge when training very deep architectures. It arises mainly due to the rigid feed-forward structure where each layer receives inputs only from its immediate predecessor layer. During backpropagation, the gradient passes through several layers, each of which potentially weakens the gradient signal that eventually arrives at the early layers of a very deep architecture. In recent years, the concept of *shortcut connections* (or, equivalently, *skip connections*) has evolved to an effective building block to solve this problem. By connecting layers not only to their immediate predecessor, but also to other preceding layers, gradients can flow more directly from the output layer towards the input which mitigates the problem of vanishing gradients. For instance, by introducing skip connections between every other layer, the length of the shortest path from the output to the input is essentially halved. Shortcut connections are typically implemented either by adding the inputs of several preceding layers [23] or by stacking them [24].

Another approach to tackle the vanishing gradient problem introduces shallow side networks that branch off from intermediate layers. By additionally training these side networks to solve the given task, an additional gradient signal is introduced which does not need to pass through a potentially deep stack of layers. This facilitates training especially of the earlier layers of a DNN. After training, the side networks are discarded. For instance, such side networks are used in the InceptionNet architecture [25]. A similar concept has been used in capsule networks for regularization: a decoder branches off from the last hidden layer that—in addition to the standard classification objective—is trained to reconstruct the inputs [26].

2.2.3 Batch Normalization

The literature has established a consensus that increasing the number of layers improves the performance on difficult tasks. However, training very deep architectures has long been difficult due to issues such as the vanishing gradient problem, among others. The vanishing gradient

problem arises when several layers attenuate the backpropagated gradient such that early layers cannot make sufficient progress during learning. By normalizing the activation $\mathbf{a}^l$ in a particular way, *batch normalization* decouples the weight and activation scales of different layers such that gradient attenuations do not accumulate over several layers.

Batch normalization is tied to stochastic mini-batch optimization. Let N_B be the number of samples in a mini-batch. Then, batch normalization transforms the data according to

$$a_{n,i}^l \leftarrow \frac{a_{n,i}^l - \mu_{\mathrm{bn},i}^l}{\sigma_{\mathrm{bn},i}^l} \cdot \gamma_{\mathrm{bn},i}^l + \beta_{\mathrm{bn},i}^l \tag{2.34}$$

where

$$\mu_{\mathrm{bn},i}^l = \frac{1}{N_\mathrm{B}} \sum_{n=1}^{N_\mathrm{B}} a_{n,i}^l \qquad \text{and} \qquad \left(\sigma_{\mathrm{bn},i}^l\right)^2 = \frac{1}{N_\mathrm{B}-1} \sum_{n=1}^{N_\mathrm{B}} \left(a_{n,i}^l - \mu_{\mathrm{bn},i}^l\right)^2. \tag{2.35}$$

Here, $\beta_{\mathrm{bn},i}^l$ and $\gamma_{\mathrm{bn},i}^l$ are trainable batch normalization parameters. For convolutions the normalization is typically applied channel-wise to preserve the translation invariant detection of features. Note that the batch statistics $\mu_{\mathrm{bn},i}^l$ and $\sigma_{\mathrm{bn},i}^l$ are *not* treated as constants, but they are computed as part of the computation graph. This is important for the resulting gradient (see below).

It is instructive to view the computation of batch normalization in (2.34) as two consecutive affine transformations. First, each activation is normalized to exhibit zero mean and unit variance across all N_B samples of the current mini-batch. Next, the activations are subject to an affine transformation with newly introduced parameters $\beta_{\mathrm{bn},i}^l$ and $\gamma_{\mathrm{bn},i}^l$. This second affine transformation is necessary to preserve the capability of the DNN to express the same family of functions as without batch normalization.

To predict the outputs of unseen data, we use batch statistics $\mu_{\mathrm{tr},i}^l$ and $\sigma_{\mathrm{tr},i}^l$ computed over the whole training set $\mathcal{D}_\mathrm{tr}$. Since computing $\mu_{\mathrm{tr},i}^l$ and $\sigma_{\mathrm{tr},i}^l$ might be time-consuming for very large datasets, it is common practice to estimate these values using an exponential moving average during training, i.e.,

$$\mu_{\mathrm{tr},i}^{l,\mathrm{new}} \leftarrow \xi_\mathrm{bn} \, \mu_{\mathrm{bn},i}^l + (1 - \xi_\mathrm{bn}) \mu_{\mathrm{tr},i}^{l,\mathrm{old}}, \tag{2.36}$$

where $\xi_\mathrm{bn} \in (0, 1)$ is a hyperparameter, and we proceed similarly for $\sigma_{\mathrm{tr},i}^l$.

Note that the function computed by a DNN employing batch normalization can just as well be represented by a DNN without batch normalization. Indeed, after training it is common to *fuse* batch normalization and the preceding linear transformation to reduce the computational cost at test-time. This might raise the question how we benefit from batch normalization at all. Most importantly, batch normalization decouples the activation scales of different layers. The activation scale of a particular layer l now only depends on the parameters $\gamma_{\mathrm{bn},i}^l$. In contrast, without batch normalization the scale is determined by a complicated interaction of all preceding layers. As a result, gradients with respect to the weights, $\nabla_{\mathbf{W}^l} \mathcal{L}$, will not be guided by directions that change the activation scale. Such directions are essentially filtered out when backpropagating through the normalization (2.34). The same holds true for the activation means, rendering bias vectors $\mathbf{w}_0$ ineffective in conjunction with batch normalization.

The independence of the scale is beneficial as it reduces the risk of exponential blowup or shrinkage causing effects such as vanishing gradients. Furthermore, the independence of the mean is also relevant. As most activations are expected to be closer to zero, we expect DNN training to suffer less from saturating effects that typically occur for large activation magnitudes. As we will see in Chapter 5, batch normalization is crucial in conjunction with binary activation functions such as $\mathrm{sign}(a)$ since it encourages a significant portion of the activations to fall on either side of zero. This helps to preserve as much information as possible after applying the

sign activation function. In summary, batch normalization is essentially a new parameterization of a DNN that renders the induced loss surface substantially easier to optimize.

The idea of batch normalization has been extended by approaches that *whiten* (decorrelate) the activation statistics. In [27], it has been shown that whitening the activations can be seen as approximately performing natural gradient descent. However, their approach considers the whitening transformation as a fixed transformation and they do not backpropagate through it. A whitening approach with backpropagation is presented in [28].

We also note that performing SGD using batch normalization affects the induced loss function. For the sum of per-sample losses (2.10), we have assumed a loss $\mathcal{L}$ that decomposes into a sum of N *independent* terms ℓ_n. This does not hold true for batch normalization since the batch statistics (2.35) also depend on the other samples in the current mini-batch. Instead, the induced loss function $\mathcal{L}$ can be seen as a sum over all $\binom{N}{N_\mathrm{B}}$ mini-batches of size N_B where each possible mini-batch corresponds to a single term ℓ_n of (2.10).

The dependence on other samples within a mini-batch can also be seen as a particular kind of noise injection. From this point of view, $\mu^l_{\mathrm{bn},i}$ and $\sigma^l_{\mathrm{bn},i}$ are random quantities determined by the randomly selected samples in a mini-batch. Consequently, (2.34) introduces both additive and multiplicative noise. Injecting noise during training is known to improve robustness and to reduce overfitting. In the next section, we discuss one of the most prominent forms of noise injection, namely dropout.

2.2.4 Dropout

Dropout belongs to a wide class of methods that inject certain kinds of noises during training. Before we delve into the details of dropout, we provide some intuition why training can benefit from injected noise. When introducing noise during training, it becomes more difficult for a model to accomplish the given task. Consequently, the model must become robust to the noise which can be exploited to achieve favorable properties of the trained model.

Consider the following examples of introducing noise during training. By injecting noise to the inputs $\mathbf{x}^0$, a model becomes more robust to small variations in the input data. However, injecting input noise can also be used to trigger a very specific behavior. For instance, autoencoders have been trained to perform denoising by randomly corrupting the inputs [29]. Another specific behavior is achieved by adding Gaussian noise to the outputs of a logistic sigmoid $x^l_i = \mathrm{sigm}(a^l_i)$. The easiest way to become robust to this noise is by moving the inputs a^l_i away from zero towards the saturated regions of the logistic sigmoid. This causes the outputs x^l_i to be close to binary which is a desirable behavior in some applications, e.g., semantic hashing [15]. As we will see in Section 4.1.2, quantization-aware training using the STE can be seen as introducing quantization noise such that the weights become robust to quantization.

The idea of dropout is to temporarily remove randomly selected inputs and hidden neurons from the DNN to reduce overfitting. This is accomplished by injecting *multiplicative* noise during training. More specifically, during forward propagation, we generate binary *dropout masks* $\mathbf{z}^L_n$ according to $z^l_i \sim \mathrm{Bernoulli}(1 - p^l_\mathrm{do})$ where $p^l_\mathrm{do} \in (0, 1)$ is a hyperparameter determining the probability of z^l_i being zero. Then a layer employing dropout replaces the affine transformation in (2.22) by

$$\mathbf{a}^l = \mathbf{W}^l \otimes (\mathbf{x}^{l-1} \odot \mathbf{z}^l) + \mathbf{w}^l_0, \tag{2.37}$$

where $\odot$ denotes element-wise multiplication. We refer to p^l_do as the dropout probability (or dropout rate) of layer l. Note that the dropout mask $\mathbf{z}^l$ of layer l must have the same shape as the output $\mathbf{x}^{l-1}$ of layer $l - 1$.

There are $2^{|\mathbf{z}|}$ possible dropout masks, each of which encodes a particular DNN structure. Therefore, dropout can be interpreted as inducing an expected loss $\mathcal{L}_\mathrm{do}$ defined by an exponen-

tially large ensemble of DNNs with shared weights, i.e.,

$$\mathcal{L}_{\text{do}}(\mathbf{W};\mathcal{D}) = \mathbb{E}_{\mathbf{z}}\left[\mathcal{L}_{\text{do}}(\mathbf{W},\mathbf{z};\mathcal{D})\right]. \tag{2.38}$$

The ensemble view helps to explain the improved generalization of dropout with the same arguments as those used for common ensemble methods. However, (2.38) also suggests that computing predictions requires exponentially many forward passes or, at least, a Monte Carlo estimate using sufficiently many dropout masks $\mathbf{z}$. Fortunately, a simple approximation requiring only a single forward pass is necessary to obtain sufficient accuracy in practice. In particular, by scaling the weights $\mathbf{W}^l$ by $(1 - p_{\text{do}}^l)$, the activations $\mathbf{a}^l$ are close to their expected value.[9] Nevertheless, evaluating many DNNs may still provide useful prediction uncertainties as suggested in [30].

Although dropout is straightforward to implement, its implications on the trained models are far more intricate than mere perturbation of the inputs $\mathbf{x}^0$. During training with dropout, a DNN cannot rely on the presence of individual features to accomplish its task. Therefore, it has to develop a certain degree of feature redundancy to become robust enough. A different aspect is that we generally do not believe that individual neurons of a DNN detect meaningful features on their own, but we rather expect that features are simultaneously detected by several different neurons in an entangled way—a phenomenon called co-adaptation of features. By randomly removing neurons from the computation, individual neurons cannot rely on the presence of other neurons anymore. Hence, they are forced to detect features on their own, essentially breaking the co-adaptation. We also expect the detected features to be somehow meaningful since they must provide useful information to accomplish the given task regardless of the specific dropout pattern.

One shortcoming of dropout is that training takes generally longer due to the increased gradient variance. Wang and Manning [31] proposed a closed-form approximation of the intractable objective (2.38) based on the central limit theorem. They achieve similar performance using less training time, indicating that dropout does not inherently rely on the stochasticity during training. Note that a closely related application of the central limit theorem is an integral part of our method for training discrete-valued DNNs presented in Chapter 5.

By studying dropout on simpler models such as linear and logistic regression, we can gain further useful insights. Wager et al. [32] have shown that dropout applied to linear regression corresponds to ℓ^2-norm regularization where features exhibiting more variance are penalized stronger. Although this does not hold exactly for DNNs, these insights are still useful to explain the improved generalization when using dropout.

Interestingly, dropout also performs well using other noises as long as the noise is injected in a multiplicative manner. For instance, *Gaussian dropout* uses dropout masks generated according to $z_i^l \sim \mathcal{N}(1, \alpha_{\text{do}}^l)$, where α_{do}^l takes the role of the hyperparameter p_{do}^l. In some contexts it is more convenient to analyze dropout using Gaussian noise, e.g., in the variational dropout framework [33]. Note that batch normalization [34] can also be seen as a means of introducing both multiplicative and additive noise through the randomly sampled mini-batches during SGD. This might explain both the regularization effect of batch normalization and the reduced effectiveness of dropout when used in conjunction with batch normalization.

Dropout has also been applied to remove individual connections during training by randomly setting the corresponding weights to zero [35]. This is a generalization of dropout since dropout can also be seen as setting entire rows and columns of $\mathbf{W}^l$ to zero. For CNNs applied to image classification, Ghiasi et al. [36] have shown that zeroing out whole blocks of neighboring pixels outperforms setting individual pixels to zero.

[9] Instead of scaling the weights, some implementations scale the dropout masks z_i^l by $1/(1 - p_{\text{do}}^l)$.

2.3 A Brief History of Deep Learning Architectures

Deep learning has gained tremendous attention since 2012 when the AlexNet architecture [6] outperformed classical computer vision methods on the *ImageNet Large Scale Visual Recognition Challenge (ILSVRC)* for image classification [5] by an unprecedented margin. After the success of AlexNet, it was rather unsurprising that all winning entries of subsequent ILSVRC challenges relied on deep learning techniques. However, the achieved progress since then is staggering and the efficiency and accuracy of the developed architectures have improved considerably.

The ImageNet dataset for image classification [21] is a large-scale dataset containing approximately 1.2 million labeled images that must be classified into 1,000 classes. We believe that the availability of large-scale datasets such as ImageNet along with the possibility to train large architectures using dedicated hardware (e.g., GPUs and TPUs) are the key factors for the success of deep learning.

We conclude this chapter with a chronological overview of the arguably most influential DNN architectures for image classification that emerged within the past decade. By investigating the recent progress, it becomes evident that network depth plays a crucial role for obtaining state-of-the-art performance. Over the years, the accuracy and the depth of DNNs have gradually increased. However, the ability to train deeper architectures did not merely arise from the ever-growing hardware capabilities, but rather from new design principles and building blocks (e.g., batch normalization) that have been developed.

Although the design principles of the architectures discussed in the remainder of this section were developed for image data (in particular, image classification), many of the techniques have been successfully transferred to other domains as well. Note that while the individual architectures sometimes deviate from the simplified layout presented in Section 2.2.1, they still adhere to the core feed-forward structure organized in simple layers.

2.3.1 AlexNet

The AlexNet architecture [6] was the first DNN capable of improving performance over conventional hand-crafted computer vision techniques. It achieved a top-5 error[10] of 16.4% at the ILSVRC12 challenge—an improvement of approximately 10% absolute error compared to the runner-up entry that relied on well-established computer vision techniques at that time. This most influential work essentially started the advent of deep learning. Since then DNNs have spread over virtually any scientific field and achieved improved performances compared to well-established methods in the respective fields.

The architecture consists of eight layers: five convolutional layers followed by three fully connected layers. AlexNet was designed to optimally utilize the available hardware at that time rather than following some clear design principles. This involves the choice of heterogeneous filter sizes $K_w \times K_h$ and seemingly arbitrary numbers of channels per layer d_l. The bulk of the weights (approximately 90%) are located in the first two fully connected layers. The overfitting caused by the heavy overparameterization of these layers was reduced using dropout [9].

Some convolutional layers compute two independent paths, each computing one half of the output channels from one half of the input channels. This corresponds to an instance of *grouped convolutions* which halves the number of operations and parameters compared to a full convolution. The main purpose of this was to facilitate training of a larger architecture on two GPUs. Interestingly, it took several years until grouped convolutions have regained attention as a means of reducing model complexity (see Section 4.3.4).

[10] The top-5 error reports a misclassification if the target is not within the five highest ranked predictions.

2.3.2 VGGNet

The VGGNet architecture [37] was the runner-up entry at the ILSVRC14 challenge achieving 7.3% top-5 error. Compared to AlexNet, the structure of VGGNet is more homogeneous and with up to 19 layers much deeper. The design of VGGNet is guided by two main principles. (i) VGGNet employs mostly 3×3 convolutions and it increases the receptive field by stacking several of them. (ii) After downscaling the spatial dimension with 2×2 max pooling, the number of channels is doubled to avoid information loss.

This is in contrast to earlier architectures that often relied on larger convolutions, e.g., AlexNet employed a 11×11 convolution in the input layer. Using several 3×3 convolutions to achieve the same receptive field as a larger convolution does not only increase the depth of the network, but it also increases the computational efficiency. For instance, computing two 3×3 convolutions instead of a single 5×5 convolutions reduces the number of weights and operations by a factor of $25/18 = 1.39$, and the computational gains even grow for larger receptive fields. Moreover, by stacking several layers, additional nonlinearities are introduced, possibly increasing the expressiveness of the model. From a hardware perspective, VGGNet is often preferred over other architectures due to its homogeneous structure.

2.3.3 InceptionNet

InceptionNet (or, equivalently, GoogLeNet) [25] won the ILSVRC14 challenge with 6.7% top-5 error using an even deeper architecture consisting of 22 layers. The main feature of this architecture is the *inception module* which concatenates the results of several independent operations—a 1×1, a 3×3, and a 5×5 convolution, as well as a pooling operation—that are performed in parallel on the inputs from the previous layer. The use of different filter sizes allows for the detection of features at different scales within a single inception module. To reduce the computational burden, InceptionNet performs cheaper 1×1 convolutions as proposed in [38] to reduce the number of channels immediately before the larger 3×3 and 5×5 convolutions.

The training of InceptionNet was aided by two shallower auxiliary networks that branch off from earlier hidden layers. By additionally training these shallower side networks to perform the predictive task, the gradient signal is improved and vanishing gradient effects are mitigated. Once training has finished, these auxiliary networks are discarded.

The InceptionNet architecture has been substantially improved by exploiting various design principles in follow-up work [39]. The computational efficiency has been improved by splitting up expensive convolutions into several cheaper convolutions. Similarly as in VGGNet, 5×5 convolutions have been split up into two consecutive 3×3 convolutions. Furthermore, filters have also been factorized along spatial dimensions to improve efficiency. For instance, approximating a 7×7 convolution by a 1×7 and a 7×1 convolution results in $49/14 = 3.5$ fewer weights and operations.

Another design principle of the follow-up work is concerned with *representational bottlenecks* that impair the expressiveness of the model. Such representational bottlenecks might appear if the dimensionality of the data is reduced too quickly, e.g., at pooling operations. To overcome representational bottlenecks, it is proposed to not only downscale the feature maps using a pooling operation, but also by a strided convolution executed in parallel to maintain a large dimensionality.

2.3.4 ResNet

Although the scientific community has established a consensus that deeper architectures are capable of improving predictive performance, training became more and more difficult as depth increases. Even worse, deeper models did not just fail to generalize well, but they also achieved inferior training set performance compared to their shallower counterparts—a phenomenon called

degradation problem. This was rather unsatisfying since the accuracy of a deep model obtained by the odd construction of first training a shallower network followed by adding artificial identity layers was hardly achievable by training the deep model from scratch.

This observation inspired the rather new paradigm of residual networks (ResNets). He et al. [23] concluded from the above observation that identity mappings might play an important role and layers should compute residuals that are added to the layer's inputs, i.e.,

$$\mathbf{a}^l = f^l(\mathbf{x}^{l-1}, \mathbf{W}^l) + \mathbf{x}^{l-1}. \tag{2.39}$$

Graphically the computation of a residual block (2.39) corresponds to two parallel paths: (i) the residual path transforming the input and (ii) the identity path (or skip connection).

Assuming that DNNs of a given depth are capable of representing a sufficiently wide range of functions, they should also be able to represent the residual function $f(\mathbf{x}^0) - \mathbf{x}^0$. Consequently, the class of representable functions using residual layers remains approximately the same. As a result, the benefits of residual layers cannot be attributed to improved model expressiveness, but rather to the induced loss surface being easier to optimize. It is instructive to think in terms of a reparameterization such that weights are optimized with reference to the more natural identity function instead of a zero function.

From the perspective of computation graphs, it becomes evident why ResNets are easier to train. By following the identity paths from the DNN output back to the inputs, gradients do not have to pass through several linear transformations that potentially cause vanishing or exploding gradients. In this way, extremely deep architectures have been successfully trained, e.g., up to 152 layers on ImageNet and even up to 1,000 layers on Cifar-10. ResNet won the ILSVRC15 challenge with 3.6% top-5 error.

In follow-up work, some structural rearrangements have been made to obtain an even cleaner identity path, further improving the learning characteristics [40]. Most notably, instead of computing the nonlinear activation function *after* adding the residual to the identity path, the nonlinear activation function has been moved completely *inside* the residual path. In another follow-up work called ResNeXt [41], grouped convolutions (see Section 4.3.4) were employed to increase the number of channels. This improves accuracy while leaving the overall computational complexity approximately the same.

2.3.5 DenseNet

Inspired by ResNets whose skip connections have been shown to improve the learning characteristics, densely connected CNNs (DenseNets) [24] drive this idea even further by connecting each layer to *every* preceding layer. On a high level, DenseNets are conceptually very similar to ResNets. Instead of adding the output and the input of a layer, DenseNets *stack* them.

However, the implications of stacking the layer outputs are different from adding them. Each layer is now a direct input to every subsequent layer which further facilitates backpropagation through shorter paths from the output to the input. Furthermore, when thinking of a DNN in terms of carrying a *state* that is transformed by each layer, DenseNets make an explicit distinction between old and new state information by design. In contrast, it is less obvious how the state is changed in a ResNet. This allows for the use of small layers introducing only a relatively small number of new channels per layer (the *growth rate*).

Nevertheless, stacking layer outputs necessarily increases the number of channels with each layer, which might result in impractical input sizes for deeper layers. Therefore, it is proposed to use *compression layers* (i.e., a 1×1 convolution to reduce the number of feature maps) after downscaling the spatial dimension with pooling. Compared to ResNets, DenseNets achieve similar performance, allow for even deeper architectures, and they are more parameter and computation efficient.

2.3.6 EfficientNet

The EfficientNet architecture [42] emerged from a thorough study of scaling architectural dimensions of DNNs, namely the depth (number of layers), the width (number of channels), and the resolution (number of pixels in the input image). Extensive experiments have shown that scaling up each of these dimensions individually quickly results in diminishing accuracy gains whereas simultaneously scaling these dimensions considerably improves accuracy. Intuitively this makes sense as, for instance, increasing the resolution of an image also increases the spatial extension of interesting features which, in turn, requires a deeper architecture for an increased receptive field to detect these features. The proposed *compound scaling method* scales each of these dimensions simultaneously by constant factors determined using a grid search.

The particular EfficientNet model was discovered using NAS [43, 44] (see Section 4.3.5) to find a relatively small but decently performing DNN which was subsequently scaled up using the proposed compound scaling approach. NAS is currently a very active research area concerned with the automatic discovery of good DNN architectures within a prespecified space of architectures. The specific architecture space used for EfficientNet comprises building blocks that are specifically tailored towards resource-efficient computation such as mobile inverted bottleneck convolutions [45].

By scaling up the discovered EfficientNet architecture using the proposed compound scaling, a whole set of architectures of various complexities can be obtained. In particular, their largest reported model achieves overall state-of-the-art performance. Nevertheless, the proposed approach is also remarkable as it surpasses the performance of the previous state of the art while being substantially more efficient. When comparing the smallest EfficientNet architecture (the one obtained with NAS) to AlexNet, the computational gains have improved by a factor of 44 within a period of only seven years [46]. This shows that progress of deep learning cannot be merely attributed to larger datasets and increased computational capabilities, but rather to improved algorithms and a better understanding of architecture design. Interestingly, EfficientNet was designed by starting from a small model which is subsequently scaled up. This is opposed to many common network pruning approaches (see Section 4.2) that start from a large model and then prune unnecessary parts to obtain an efficient architecture.

3 Bayesian Deep Learning

In the previous chapter, we considered loss function minimization to obtain a fixed set of parameters $\boldsymbol{\theta}$. Once those parameters are found, we are committed to using these parameters for all our future predictions. Any valuable information of the training data that was not absorbed during the training process is essentially lost. This chapter introduces the framework of Bayesian inference which does not suffer from this information loss. The core idea of Bayesian inference is to consider the parameters $\boldsymbol{\theta}$ as random quantities. Using Bayes' rule, this allows us to summarize the whole information of our prior belief (the prior distribution) and empirical observations (the training data via the likelihood) in a posterior distribution over the parameters. Subsequently, the whole parameter space is considered by computing expectations with respect to this posterior.

The Bayesian framework has several appealing properties. It is less prone to overfitting, it naturally handles the online setting where data samples arrive in an indefinite sequence, and it provides a well-justified means of obtaining prediction uncertainties. However, exact Bayesian inference requires solving integrals that are often intractable. As a consequence, the field of approximate Bayesian inference is a very active research area devoted to developing effective approximations in various contexts.

We start our discussion by introducing the basic concepts of Bayesian inference and provide examples where exact inference is possible. We continue with common approximate Bayesian inference techniques for models where exact inference is not possible. The focus of our discussion is on the two most widely used approximation schemes, i.e., variational inference and sampling methods. Our discussion also shows how loss function minimization as discussed in the previous chapter can be seen as a specific way of performing approximate Bayesian inference. After a general treatment of these topics, we turn to the specific field of Bayesian deep learning and show how these techniques can be applied to DNNs.

3.1 Bayesian Inference

The core idea of Bayesian learning is to treat the parameters $\boldsymbol{\theta}$ as uncertain quantities by modeling them as random variables. We consider a parametric family of distributions $p(\mathcal{D} | \boldsymbol{\theta})$ to model a given dataset $\mathcal{D}$. In this context, $p(\mathcal{D} | \boldsymbol{\theta})$ is referred to as the *likelihood* of the parameters $\boldsymbol{\theta}$. Note that, to emphasize that the parameters $\boldsymbol{\theta}$ are now random variables, they appear in the condition rather than in the subscript. The dataset $\mathcal{D}$ may contain either unlabeled data $\mathcal{D} = \{\mathbf{x}_n\}_{n=1}^N$ or, as in the previous chapter, labeled data $\mathcal{D} = \{(\mathbf{x}_n, \mathbf{y}_n)\}_{n=1}^N$. We assume that the individual samples $\mathbf{x}_n$ are conditionally independent given the parameters $\boldsymbol{\theta}$, i.e., they are sampled i.i.d. Then, for unlabeled data, the distribution $p(\mathcal{D} | \boldsymbol{\theta})$ factorizes as

$$p(\mathcal{D} | \boldsymbol{\theta}) = \prod_{n=1}^{N} p(\mathbf{x}_n | \boldsymbol{\theta}). \tag{3.1}$$

For labeled data, the distribution factorizes as

$$p(\mathcal{D}|\boldsymbol{\theta}) = \prod_{n=1}^{N} p(\mathbf{y}_n|\mathbf{x}_n, \boldsymbol{\theta}) p(\mathbf{x}_n|\boldsymbol{\theta}). \tag{3.2}$$

For the labeled case, we adopt the common practice and do not model the distribution $p(\mathbf{x}|\boldsymbol{\theta})$ over the inputs. This corresponds to assuming a uniform distribution over $\mathbf{x}$. For the sake of brevity, we consider either the labeled or the unlabeled data case in the following discussions. Nevertheless, the conclusions are valid for both cases.

To introduce uncertainties over the parameters $\boldsymbol{\theta}$, the first step of Bayesian learning is to introduce a *prior distribution* $p(\boldsymbol{\theta})$ that expresses our initial belief about the parameters. If no particular prior knowledge is available, the prior is typically selected to be rather uninformative such that no parameters are assigned a probability of zero. As discussed in Section 2.1.1, for some models it is reasonable to assume that their parameters are small, suggesting a prior distribution assigning more mass to parameters close to zero. One might also select a more informative prior to incorporate expert knowledge or experience gained from previous experiments. For some applications it is even reasonable to exclude certain parameter configurations $\boldsymbol{\theta}$ by setting their prior mass to zero. This can be used to introduce hard constraints over the parameters $\boldsymbol{\theta}$, e.g., sparsity or equality constraints. For instance, convolutional layers in CNNs can be seen as arising from a Bayesian treatment of fully connected layers where the weight matrix $\mathbf{W}$ exhibits a certain sparsity pattern and several weights are shared.

In the next step of Bayesian learning, a set of data samples $\mathcal{D}$ is collected. We can then employ Bayes' theorem to update our prior belief about the parameters according to

$$p(\boldsymbol{\theta}|\mathcal{D}) = \frac{p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta})}{p(\mathcal{D})} \propto p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta}). \tag{3.3}$$

The left hand side of (3.3) is known as the *posterior distribution* over the parameters, and the denominator $p(\mathcal{D})$ is a normalization constant referred to as *marginal likelihood* or *evidence*. The posterior summarizes the prior knowledge and the whole information about the parameters $\boldsymbol{\theta}$ contained in the dataset $\mathcal{D}$. Subsequently, the posterior is used to obtain a *predictive distribution* by marginalizing out the parameters $\boldsymbol{\theta}$ as

$$p(\mathbf{x}|\mathcal{D}) = \int_{\boldsymbol{\Theta}} p(\mathbf{x}, \boldsymbol{\theta}|\mathcal{D})\, d\boldsymbol{\theta} = \int_{\boldsymbol{\Theta}} p(\mathbf{x}|\boldsymbol{\theta}) p(\boldsymbol{\theta}|\mathcal{D})\, d\boldsymbol{\theta} = \mathbb{E}_{\boldsymbol{\theta} \sim p(\boldsymbol{\theta}|\mathcal{D})} \left[p(\mathbf{x}|\boldsymbol{\theta}) \right], \tag{3.4}$$

where we have used the conditional independence assumption in the second equality. Equation (3.4) has several appealing properties. Rather than computing predictions based on a single model $\boldsymbol{\theta}$, the predictive distribution (3.4) is obtained by computing predictions using every possible model $\boldsymbol{\theta} \in \boldsymbol{\Theta}$ and averaging their outputs according to the posterior $p(\boldsymbol{\theta}|\mathcal{D})$.

This avoids several problems associated with predictions based on individual models.[11] Most importantly, the Bayesian framework is much less susceptible to overfitting. This renders Bayesian inference especially interesting for applications where data is scarce. In fact, as we will see in Section 3.2.1, certain regularizers emerge from loss functions whose minima correspond to modes in the posterior distribution.

Moreover, the probabilistic semantics obtained via the predictive distribution (3.4) are well supported through the Bayesian framework. Consequently, the prediction uncertainties from (3.4) typically reflect our intuition well. In contrast, the uncertainties from $p(\mathbf{x}|\boldsymbol{\theta})$ obtained using a point estimate $\boldsymbol{\theta}$ are often spurious and yield overconfident predictions [30].

The Bayesian framework also provides an elegant framework to handle the online learning setting where samples $\mathbf{x}_n$ arrive as an indefinite sequence and are only processed once and never considered again. For this purpose, let $\mathcal{D}^{(N)} = \{\mathbf{x}_n\}_{n=1}^{N}$ be a dataset containing N samples such

[11] Individual models are also called *point estimates* in this context.

that $\mathcal{D}^{(N)} \subset \mathcal{D}^{(N+1)}$ for all N. We can then define a posterior distribution after observing the first N samples as

$$p(\boldsymbol{\theta} \,|\, \mathcal{D}^{(N)}) \propto p(\mathcal{D}^{(N)} \,|\, \boldsymbol{\theta})\, p(\boldsymbol{\theta}). \tag{3.5}$$

When we observe another data sample $\mathbf{x}_{N+1}$, the posterior (3.5) obtained from the first N samples can serve as a new prior to obtain an updated posterior as

$$p(\boldsymbol{\theta} \,|\, \mathcal{D}^{(N+1)}) \propto p(\mathbf{x}_{N+1} \,|\, \boldsymbol{\theta})\, p(\boldsymbol{\theta} \,|\, \mathcal{D}^{(N)}). \tag{3.6}$$

Despite all these appealing properties of Bayesian inference, its practical application is still limited. In most cases, the posterior distribution exhibits a complicated structure, rendering the required computations (e.g., marginalization) difficult if not infeasible. For instance, when employing Bayesian inference for DNNs, the posterior $p(\mathbf{W} \,|\, \mathcal{D})$ inherits all the unfavorable computational properties of DNNs such as their high nonlinearity. In these cases, we have to rely on approximation techniques which are discussed in a general context in Section 3.2 and specifically for DNNs in Sections 3.3–3.5.

For some models and distributions, however, the posterior distribution admits a concise representation, reducing the knowledge contained in a dataset $\mathcal{D}$ to only a few numerical values. Furthermore, these models allow us to compute the predictive distribution (3.4) in closed form. In the following, we discuss such a class of models, namely the exponential family.

3.1.1 Example: The Exponential Family and Conjugate Priors

Exact Bayesian inference (3.3) and computing the posterior predictive distribution (3.4) is generally infeasible for distributions with high-dimensional parameter spaces $\boldsymbol{\Theta}$. However, for certain combinations of the prior and the likelihood, posterior inference admits an exact solution. In this section, we discuss an entire family of distributions with favorable properties for posterior inference, namely the exponential family. Many familiar distributions belong to the exponential family, e.g., the Gaussian, Bernoulli, categorical, exponential, and the gamma distribution.

More specifically, a distribution belongs to the exponential family if its pdf or pmf can be phrased as

$$p(\mathbf{x} \,|\, \boldsymbol{\theta}) = h(\mathbf{x}) \exp\Big(\boldsymbol{\theta}^\top \boldsymbol{\phi}(\mathbf{x}) - A(\boldsymbol{\theta})\Big). \tag{3.7}$$

Here $\boldsymbol{\phi} : \mathbb{R}^D \to \mathbb{R}^M$ is called the sufficient statistics function, $\boldsymbol{\theta} \in \mathbb{R}^M$ are the natural parameters[12], $A(\boldsymbol{\theta})$ is called the log-partition function that ensures normalization, and $h(\mathbf{x})$ is the base measure.

Distributions from the exponential family have several interesting properties, many of which stem from a connection to convex analysis [47]. We restrict our discussion to a particular property that is especially appealing for Bayesian inference, namely the existence of *conjugate priors*. A prior $p(\boldsymbol{\theta})$ is said to be conjugate to a likelihood function $p(\mathcal{D} \,|\, \boldsymbol{\theta})$, if the induced posterior $p(\boldsymbol{\theta} \,|\, \mathcal{D})$ according to (3.3) exhibits the same form as the prior. In our case, this means that the posterior is from the same exponential family as the prior.

To see why this holds, consider the prior

$$p(\boldsymbol{\theta} \,|\, \boldsymbol{\alpha}, N_0) \propto \exp\Big(\boldsymbol{\theta}^\top \boldsymbol{\alpha} - N_0\, A(\boldsymbol{\theta})\Big), \tag{3.8}$$

where $(\boldsymbol{\alpha}, N_0)$ are natural parameters and the sufficient statistics are $\boldsymbol{\theta} \mapsto (\boldsymbol{\theta}, -A(\boldsymbol{\theta}))$. The

[12] The parameters are distinctively called *natural*, since exponential family distributions are typically represented in a non-exponential family style using a different parameterization.

likelihood for a dataset $\mathcal{D}$ containing N i.i.d. samples can be written as

$$p(\mathcal{D}|\boldsymbol{\theta}) = \left(\prod_{n=1}^{N} h(\mathbf{x}_n)\right) \exp\left(\boldsymbol{\theta}^\top \left(\sum_{n=1}^{N} \boldsymbol{\phi}(\mathbf{x}_n)\right) - NA(\boldsymbol{\theta})\right). \tag{3.9}$$

Multiplying the prior (3.8) with the likelihood (3.9) yields the posterior

$$p(\boldsymbol{\theta}|\mathcal{D}) \propto \exp\left(\boldsymbol{\theta}^\top \left(\boldsymbol{\alpha} + \sum_{n=1}^{N} \boldsymbol{\phi}(\mathbf{x}_n)\right) - (N_0 + N)A(\boldsymbol{\theta})\right), \tag{3.10}$$

which is of the same form as the prior (3.8).

The form of the posterior (3.10) reveals interesting properties of the posterior. First, the posterior depends on the data only via a sum of the sufficient statistics over the dataset $\mathcal{D}$. Consequently, the entire information of the dataset $\mathcal{D}$ can be compressed to an M-dimensional vector. This is especially convenient from a computational perspective as it allows us to perform online inference by simply updating the sum of the sufficient statistics. Furthermore, the prior has the intuitive interpretation as observing N_0 pseudo samples whose sum of sufficient statistics equals $\boldsymbol{\alpha}$.

The key property that allows us to perform exact inference for exponential family distributions and conjugate priors is the ability to exactly compute the normalization factor $p(\mathcal{D})$ in (3.3). From this property and by observing that the predictive distribution factorizes as $p(\mathbf{x}|\mathcal{D}) = p(\mathbf{x}, \mathcal{D})/p(\mathcal{D})$, it immediately follows that computing the predictive distribution (3.4) is feasible in the exponential family setting with conjugate priors.

3.1.2 Bayesian Networks

BNs provide an elegant framework to define properties of a multivariate distribution by means of a graphical representation. We introduce BNs here because they allow us to put posterior inference as introduced above in a broader context. Moreover, we will refer to them in the remainder of this thesis on several occasions; particularly in Chapter 7 which is fully dedicated to a specific model class known as BN classifiers.

Let $\boldsymbol{X} = \{X_1, \ldots, X_D\}$ be a set of random variables. Furthermore, let $\mathcal{G}$ be a directed acyclic graph whose nodes correspond to $\boldsymbol{X}$. Then a BN with graph structure $\mathcal{G}$ defines a factorization of the joint distribution $p(\boldsymbol{X})$ as

$$p(\boldsymbol{X}) = \prod_{i=1}^{D} p(X_i \,|\, \mathrm{pa}(X_i)), \tag{3.11}$$

where $\mathrm{pa}(X_i)$ are the parents of X_i in $\mathcal{G}$. For nodes X_i without parents, the corresponding factor in (3.11) is an unconditional distribution $p(X_i)$. This factorization is convenient as it allows us to specify the joint distribution $p(\boldsymbol{X})$ in terms of local factors $p(X_i \,|\, \mathrm{pa}(X_i))$. The factorization in (3.11) is general since any distribution can be factorized in this way by the product rule of probability, i.e.,

$$p(\boldsymbol{X}) = p(X_1) \prod_{i=2}^{D} p(X_i|X_1, \ldots, X_{i-1}). \tag{3.12}$$

The corresponding graph $\mathcal{G}$ of (3.12) is complete in the sense that it contains a directed edge between every pair of variables X_i.

However, it turns out that BNs reveal interesting properties of the underlying distributions only if edges are missing. The graph structure $\mathcal{G}$ also encodes a set of conditional independence assumptions that hold for any distribution that factorizes according to (3.11). For instance, one

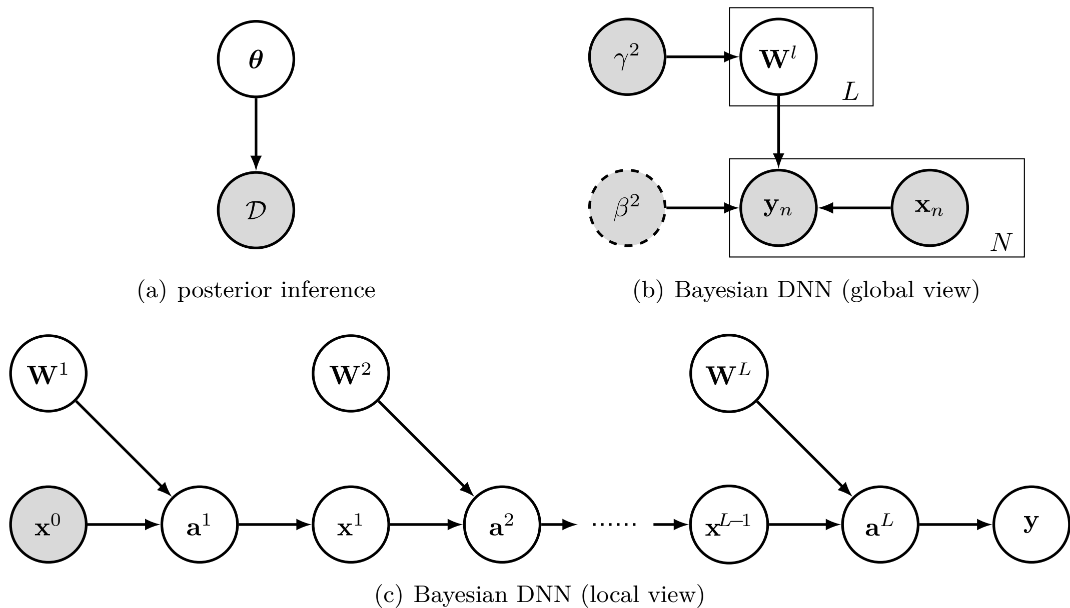


Figure 3.1: (a) Simple graphical model of $p(\boldsymbol{\theta}, \mathcal{D}) = p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta})$. (b) A Bayesian model of DNNs as a BN. (c) BN of a Bayesian DNN at the local level of individual layers.

can show that for a given graph $\mathcal{G}$, a variable X_i is conditionally independent of non-descendants X_j given its parents $\text{pa}(X_i)$. The set of independencies obtained in this way are called *local independencies*. These local independencies, however, are only a subset of all independencies encoded by a graph $\mathcal{G}$. We note that a characterization of the complete set of conditional independencies encoded by $\mathcal{G}$ is provided by the d-separation criterion. In short, d-separation characterizes conditional independence statements between two nodes X_i and X_j through a certain blocking property of undirected paths between them. We refer to [48] for a thorough treatment of the topic.

Therefore, BNs establish a link between distributional properties of p and graph theoretic properties of $\mathcal{G}$. Besides these interesting theoretical properties, they provide a practical tool to formally specify distributions by incorporating domain knowledge. For instance, the edges in $\mathcal{G}$ might have a direct interpretation as causal relationships among the connected variables. This often implies a certain degree of interpretability of the resulting models. Not least of all, BNs provide a powerful language to talk about probability distributions in more intuitive terms.

We can now represent the simple Bayesian model $p(\boldsymbol{\theta}, \mathcal{D}) = p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta})$ from above in terms of a simple BN. This is shown in Figure 3.1(a). In our representation, shaded nodes indicate that the corresponding values are observed. The direction of the edge indicates the generative process of how the dataset $\mathcal{D}$ is generated from a distribution $p(\mathcal{D}|\boldsymbol{\theta})$. In this light, inferring the posterior distribution $p(\boldsymbol{\theta}|\mathcal{D})$ can be seen as deducing the reverse direction of the generative process. We emphasize that BNs are much more powerful when used to model probabilistic inference scenarios for graphs with more structure than our simple posterior inference sample shown here.

Deep Neural Networks as Bayesian Networks

Next, we show how DNNs—when viewed in a Bayesian framework—can be seen as a specific BN. Assume a prior distribution $p(\mathbf{W}|\gamma^2)$ over the weights governed by a variance hyperparameter γ^2. The corresponding BN is shown in Figure 3.1(b). We use the plate notation (i.e., the boxes around certain variables) to denote that the corresponding variables within the boxes are

replicated L and N times. The conditional distribution over the targets

$$p(\mathbf{y}_n \,|\, \mathbf{x}_n, \mathbf{W}, [\beta^2]) \tag{3.13}$$

is specified by the output of the DNN. The square brackets in (3.13) and the dashed circle denote that the output variance β^2 is only relevant for regression (see Section 3.2.1). Furthermore, the graph explicitly shows the hyperparameters γ^2 and β^2, but since these are known quantities they could also be absorbed into $p(\mathbf{W})$ and (3.13), respectively. Using the assumption that $p(\mathbf{W})$ factorizes over the individual layers as

$$p(\mathbf{W}) = \prod_{l=1}^{L} p(\mathbf{W}^l), \tag{3.14}$$

we obtain a more fine-grained BN at the local level of individual layers. This is shown in Figure 3.1(c). Any hyperparameters governing the weights $\mathbf{W}$ and the outputs $\mathbf{y}$ are not explicitly shown. We can see that the uncertainty from the weight distributions $p(\mathbf{W}^l)$ induces a probability distribution over the hidden layers $p(\mathbf{a}^l \,|\, \mathbf{W}^l, \mathbf{x}^{l-1})$. This allows for the interpretation that DNNs with uncertain weights propagate distributions from layer to layer. This process ends at the output layer where we eventually obtain a distribution over the targets $\mathbf{y}$. Indeed, we will utilize this view of DNNs extensively in Section 3.3 and Section 3.4 to perform approximate inference for Bayesian DNNs.

3.2 Approximate Bayesian Inference

For most models, exact Bayesian inference is intractable and we must resort to approximations. In this section, we introduce common approximation techniques in a general setting, before we show how they can be applied to DNNs in particular in Sections 3.3–3.5.

Although typically not considered as Bayesian methods, we start our discussion with point estimates and show how computing a mode in a posterior distribution $p(\boldsymbol{\theta} \,|\, \mathcal{D})$ relates to regularized loss functions. We continue our discussion with the Laplace method which utilizes curvature information from the log-posterior to approximate the posterior by a Gaussian centered at a mode.

The main part of this section discusses the two predominant approximate Bayesian inference methods, i.e., variational inference and sampling methods. These two approaches operate in a fundamentally different way. On the one hand, variational inference methods approximate the posterior by a simpler distribution. Variational inference methods are typically biased in the sense that they are unable to recover the true posterior that is typically not contained in the family of simpler distributions. However, these methods enjoy properties from the rich optimization literature [49] to determine when no further approximation progress can be expected. On the other hand, sampling based methods approximate the posterior distribution by generating a collection of samples that are representative for the posterior. Sampling based methods are guaranteed to asymptotically recover the true posterior, but it is difficult to evaluate the approximation quality when stopping the algorithm after finite time.

3.2.1 Maximum Likelihood and Maximum A Posteriori Estimation

One of the most widely used paradigms to estimate model parameters $\boldsymbol{\theta}$ that explain a given dataset $\mathcal{D}$ are the maximum a posteriori (MAP) and the maximum likelihood (ML) principles. The MAP approach aims to find a point of maximum density (or mass) in the posterior distribution by means of optimization, therefore essentially condensing the whole posterior to a single

point estimate. The ML approach is very similar in that it aims to find a point of maximum likelihood $p(\mathcal{D}\,|\,\boldsymbol{\theta})$. In fact, ML estimation can be seen as MAP estimation under a uniform prior $p(\boldsymbol{\theta})$.

We apply the common assumption of an i.i.d. dataset $\mathcal{D}$ such that the likelihood $p(\mathcal{D}\,|\,\boldsymbol{\theta})$ factorizes over the individual samples. Furthermore, it is common to apply the logarithm which does not change the location of optimal parameters $\boldsymbol{\theta}_{\text{MAP}}$ and simplifies optimization by turning products into sums. Then, MAP estimation is performed by computing

$$\boldsymbol{\theta}_{\text{MAP}} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \sum_{n=1}^{N} \log p(\mathbf{y}_n\,|\,\mathbf{x}_n, \boldsymbol{\theta}) + \log p(\boldsymbol{\theta}) + \text{const}, \tag{3.15}$$

where the constant arising from $\log p(\mathcal{D})$ can be ignored for optimization. We will now show how the commonly used loss functions discussed in Section 2.1.1 emerge from MAP estimation. Without specific knowledge about the parameters, it is common to assume a Gaussian prior with zero mean and fixed variance γ^2, i.e., $\boldsymbol{\theta} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}\gamma^2)$. For regression, it is common to assume that the targets y are normally distributed according to

$$y \sim \mathcal{N}(\hat{f}_{\boldsymbol{\theta}}(\mathbf{x}), \beta^2), \tag{3.16}$$

where $\hat{f}_{\boldsymbol{\theta}}$ is some model governed by parameters $\boldsymbol{\theta}$ (e.g., a DNN) and β^2 is some fixed variance. For this specific case, the MAP parameters (3.15) are given by

$$\boldsymbol{\theta}_{\text{MAP}} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} -\frac{1}{2\beta^2} \sum_{n=1}^{N} \left(\hat{f}_{\boldsymbol{\theta}}(\mathbf{x}_n) - y_n\right)^2 - \frac{1}{2\gamma^2} \sum_{\theta \in \boldsymbol{\theta}} \theta^2 + \text{const}, \tag{3.17}$$

where additive terms, corresponding to the normalization factors of the Gaussian distributions, have been absorbed into the constant. Note that (3.17) is equivalent to minimizing the regularized loss (2.9) for a suitable trade-off parameter λ and $\mathcal{L}_{\text{data}}$ being the MSE loss (2.6).

We can proceed similarly for classification problems. For binary classification, the target $y \in \{0, 1\}$ is assumed to be Bernoulli distributed with pmf

$$p(y\,|\,\mathbf{x}, \boldsymbol{\theta}) = \hat{f}_{\boldsymbol{\theta}}(\mathbf{x})^{y} \cdot (1 - \hat{f}_{\boldsymbol{\theta}}(\mathbf{x}))^{1-y}, \tag{3.18}$$

where $\hat{f}_{\boldsymbol{\theta}}(\mathbf{x}) \in [0, 1]$ (e.g., the output of a logistic sigmoid). For the likelihood (3.18), the MAP parameters are given by

$$\boldsymbol{\theta}_{\text{MAP}} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \sum_{n=1}^{N} \Big(y \log \hat{f}_{\boldsymbol{\theta}}(\mathbf{x}_n) + (1-y) \log(1 - \hat{f}_{\boldsymbol{\theta}}(\mathbf{x}_n))\Big) - \frac{1}{2\gamma^2} \sum_{\theta \in \boldsymbol{\theta}} \theta^2 + \text{const}, \tag{3.19}$$

which corresponds to the binary cross-entropy loss (2.7). For multiclass classification, the target $y \in \{1, \ldots, \mathcal{C}\}$ is assumed to be categorically distributed with pmf

$$p(y\,|\,\mathbf{x}, \boldsymbol{\theta}) = \prod_{y'=1}^{\mathcal{C}} \left(\hat{f}_{\boldsymbol{\theta}}(\mathbf{x})_{y'}\right)^{\mathbb{I}[y=y']} = \hat{f}_{\boldsymbol{\theta}}(\mathbf{x})_y, \tag{3.20}$$

where $\sum_{y'=1}^{\mathcal{C}} \hat{f}_{\boldsymbol{\theta}}(\mathbf{x})_{y'} = 1$ and $\hat{f}_{\boldsymbol{\theta}}(\mathbf{x})_{y'} \geq 0$ (e.g, the output of a softmax). The resulting MAP parameters are given by

$$\boldsymbol{\theta}_{\text{MAP}} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \sum_{n=1}^{N} \left(\log \hat{f}_{\boldsymbol{\theta}}(\mathbf{x}_n)_{y_n}\right) - \frac{1}{2\gamma^2} \sum_{\theta \in \boldsymbol{\theta}} \theta^2 + \text{const}. \tag{3.21}$$

Once again this recovers a well-known loss function, i.e., the multiclass cross-entropy loss (2.8).

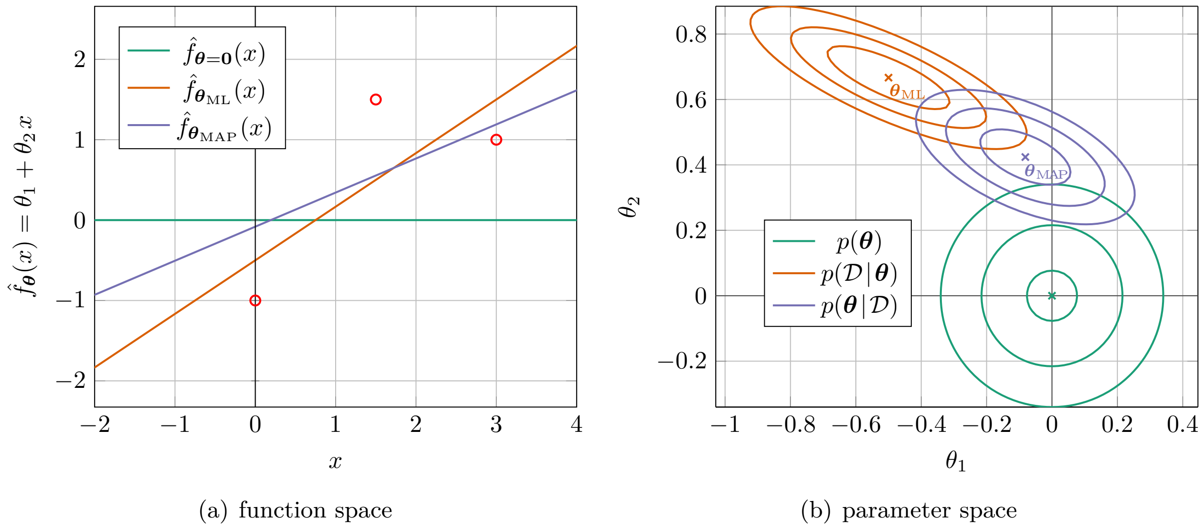


Figure 3.2: Bayesian inference for the model $f_{\boldsymbol{\theta}}(x) = \theta_1 + \theta_2 x$. (a) Function space view. The red circles correspond to the labeled dataset $\mathcal{D}$. The three lines correspond to the functions $f_{\boldsymbol{\theta}}(x)$ obtained for the prior mean $\boldsymbol{\theta} = \mathbf{0}$, the ML parameters $\boldsymbol{\theta}_{\mathrm{ML}}$, and the MAP parameters $\boldsymbol{\theta}_{\mathrm{MAP}}$. (b) Parameter space view. Contour lines of the prior $p(\boldsymbol{\theta})$, the likelihood $p(\mathcal{D}\,|\,\boldsymbol{\theta})$, and the posterior $p(\boldsymbol{\theta}\,|\,\mathcal{D})$.

Note that regression introduces an additional variance hyperparameter β^2 in the likelihood (3.16). This hyperparameter specifies the degree of label noise that we assume for the continuous targets y. For classification, we typically do not assume label noise and, therefore, no additional parameter is required.

Figure 3.2 illustrates a simple linear regression example for the model $f_{\boldsymbol{\theta}}(x) = \theta_1 + \theta_2 x$. The ML parameters $\boldsymbol{\theta}_{\mathrm{ML}}$ provide the best fit to the given data in the least squares sense. Compared to $\boldsymbol{\theta}_{\mathrm{ML}}$, the MAP parameters $\boldsymbol{\theta}_{\mathrm{MAP}}$ are pulled towards the origin due to the prior $p(\boldsymbol{\theta})$ that assumes small parameter values. Consequently, the function for $\boldsymbol{\theta}_{\mathrm{MAP}}$ is somewhat simpler than that for $\boldsymbol{\theta}_{\mathrm{ML}}$: It exhibits a smaller slope (θ_2) and its y-intercept (θ_1) is closer to zero.

Although MAP estimation is typically not considered as a Bayesian method, we will see that it can be interpreted as the special case of performing variational inference where the posterior is approximated by a single point mass $\delta_{\boldsymbol{\theta}}$. Computing expectations with respect to this point mass $\delta_{\boldsymbol{\theta}}$ then reduces to simply evaluating the model function $\hat{f}_{\boldsymbol{\theta}}$.

3.2.2 Laplace's Method

While point estimates only consider the parameters $\boldsymbol{\theta}_{\mathrm{MAP}}$ at a point of maximum density in the posterior $p(\boldsymbol{\theta}\,|\,\mathcal{D})$, Laplace's method additionally takes the curvature of $\log p(\boldsymbol{\theta}\,|\,\mathcal{D})$ at $\boldsymbol{\theta}_{\mathrm{MAP}}$ into account. In particular, Laplace's method approximates the posterior $p(\boldsymbol{\theta}\,|\,\mathcal{D})$ by a Gaussian distribution $q(\boldsymbol{\theta})$ centered at a mode $\boldsymbol{\theta}_{\mathrm{MAP}}$ such that the curvature of $\log p(\boldsymbol{\theta}\,|\,\mathcal{D})$ and $\log q(\boldsymbol{\theta})$ at $\boldsymbol{\theta}_{\mathrm{MAP}}$ are equal. This is accomplished by computing a second-order Taylor expansion of $\log p(\boldsymbol{\theta}\,|\,\mathcal{D})$ around a mode $\boldsymbol{\theta}_{\mathrm{MAP}}$, i.e.,

$$\log p(\boldsymbol{\theta}\,|\,\mathcal{D}) \approx \log p(\boldsymbol{\theta}_{\mathrm{MAP}}\,|\,\mathcal{D}) - \frac{1}{2}\left(\boldsymbol{\theta} - \boldsymbol{\theta}_{\mathrm{MAP}}\right)^{\top}(-\mathbf{H})\left(\boldsymbol{\theta} - \boldsymbol{\theta}_{\mathrm{MAP}}\right), \tag{3.22}$$

where

$$\mathbf{H} = \frac{\partial^2}{\partial\theta_i \partial\theta_j} \log p(\boldsymbol{\theta}\,|\,\mathcal{D})\bigg|_{\boldsymbol{\theta}=\boldsymbol{\theta}_{\mathrm{MAP}}}. \tag{3.23}$$

Note that since the Taylor expansion is computed around a maximum, the first-order term vanishes and $-\mathbf{H}$ is positive semidefinite. Exponentiation of (3.22) yields

$$p(\boldsymbol{\theta}\,|\,\mathcal{D}) \approx q(\boldsymbol{\theta}) \propto \exp\!\left(-\frac{1}{2}\,(\boldsymbol{\theta} - \boldsymbol{\theta}_{\mathrm{MAP}})^{\top}\,(-\mathbf{H})\,(\boldsymbol{\theta} - \boldsymbol{\theta}_{\mathrm{MAP}})\right), \tag{3.24}$$

which has the form of a Gaussian distribution with mean $\boldsymbol{\theta}_{\mathrm{MAP}}$ and covariance matrix $(-\mathbf{H})^{-1}$. The normalization constant of (3.24) is given by $(2\pi)^{-D/2}\det(-\mathbf{H})^{1/2}$, where D denotes the dimensionality of $\boldsymbol{\theta}$.

In contrast to point estimates, the Laplace approximation provides uncertainty information around a mode $\boldsymbol{\theta}_{\mathrm{MAP}}$. These uncertainties often allow for better approximations of predictive distributions by replacing $p(\boldsymbol{\theta}\,|\,\mathcal{D})$ in (3.4) by $q(\boldsymbol{\theta})$. The Gaussian approximation is convenient as it often admits a closed-form solution of the expectation in (3.4). In cases where no closed-form solution is available, the expectation (3.4) can still be approximated by Monte Carlo averaging using samples from the Gaussian distribution $q(\boldsymbol{\theta})$.

The Laplace approximation typically provides good estimates if many data points are observed and if the true distribution tends to be normally distributed as a consequence of the central limit theorem. Furthermore, many distributions are known to become unimodal in the large sample limit where the posterior $p(\boldsymbol{\theta}\,|\,\mathcal{D})$ concentrates around the true parameters, which justifies the approximation with a unimodal distribution. However, for multimodal distributions, the approximation quality might depend heavily on the particular MAP estimate $\boldsymbol{\theta}_{\mathrm{MAP}}$ discovered during optimization. Nevertheless, the benefits of the Laplace approximation over point estimates are questionable considering that it also only depends on properties of the posterior $p(\boldsymbol{\theta}\,|\,\mathcal{D})$ at a mode $\boldsymbol{\theta}_{\mathrm{MAP}}$. Consequently, important global properties might not be modeled well.

3.2.3 Variational Inference

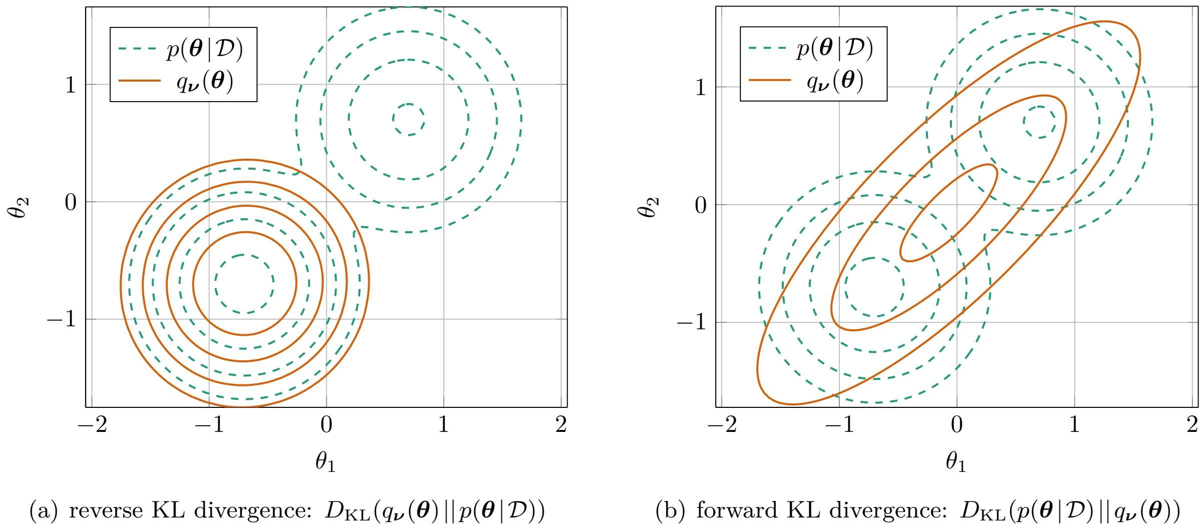
Variational inference is concerned with approximating the intractable posterior $p(\boldsymbol{\theta}\,|\,\mathcal{D})$ by a simpler distribution $q(\boldsymbol{\theta})$. Subsequently, we can approximate expectations such as the predictive distribution (3.4) by replacing the true posterior $p(\boldsymbol{\theta}\,|\,\mathcal{D})$ with the approximation $q(\boldsymbol{\theta})$.

Different variational inference methods can essentially be distinguished by (i) their choice of a particular family of approximating distributions $\{q_{\boldsymbol{\nu}}\}$ and (ii) by the criterion according to which a particular element $q_{\boldsymbol{\nu}}$ is selected from that family. Here, $\boldsymbol{\nu}$ denotes the variational parameters that specify a particular distribution, e.g., the mean and the covariance matrix of a Gaussian distribution.

Considering the choice of the particular family of distributions $\{q_{\boldsymbol{\nu}}\}$, one aims to satisfy two opposing objectives. On the one hand, the family of distributions should be rich enough to contain distributions that are in some sense close to the true posterior $p(\boldsymbol{\theta}\,|\,\mathcal{D})$. On the other hand, the distribution $q_{\boldsymbol{\nu}}(\boldsymbol{\theta})$ should ideally admit closed-form solutions to the predictive distribution (3.4) or allow for efficient Monte Carlo approximations by sampling from $q_{\boldsymbol{\nu}}(\boldsymbol{\theta})$. It is common to assume independence among the individual dimensions of $\boldsymbol{\theta}$ such that $q_{\boldsymbol{\nu}}(\boldsymbol{\theta})$ factorizes into a product over the individual dimensions $q_i(\theta_i)$ where each factor is governed by its individual variational parameters $\boldsymbol{\nu}_i$. This independence assumption is also referred to as the *mean field assumption*.

Given a family of approximating distributions, variational inference aims to find a distribution $q_{\boldsymbol{\nu}}(\boldsymbol{\theta})$ by optimizing an objective that measures the similarity between the approximation $q_{\boldsymbol{\nu}}(\boldsymbol{\theta})$ and the true posterior $p(\boldsymbol{\theta}\,|\,\mathcal{D})$. The term *variational* originates from a relation to the calculus of variations, i.e., by considering a function space view where optimization is performed over the *space of distributions* $\{q_{\boldsymbol{\nu}}\}$. However, in practice, variational inference typically employs standard (continuous) optimization techniques over the space of the variational parameters $\boldsymbol{\nu}$.

There exists a range of optimization criteria for variational inference. The most common criteria are based on the KL divergence which measures the dissimilarity between two distribu-



(a) reverse KL divergence: $D_{\mathrm{KL}}(q_{\boldsymbol{\nu}}(\boldsymbol{\theta})\,||\,p(\boldsymbol{\theta}|\mathcal{D}))$

(b) forward KL divergence: $D_{\mathrm{KL}}(p(\boldsymbol{\theta}|\mathcal{D})\,||\,q_{\boldsymbol{\nu}}(\boldsymbol{\theta}))$

Figure 3.3: Reverse KL divergence vs forward KL divergence. The posterior $p(\boldsymbol{\theta}|\mathcal{D})$ is a bimodal distribution with a slightly larger mode at the bottom left. (a) For the reverse KL divergence, $q_{\boldsymbol{\nu}}(\boldsymbol{\theta})$ captures the larger mode due to the mode-seeking behavior. (b) For the forward KL divergence, $q_{\boldsymbol{\nu}}(\boldsymbol{\theta})$ captures both modes due to the mass-covering behavior.

tions. Consequently, by minimizing the KL divergence between the posterior $p(\boldsymbol{\theta}|\mathcal{D})$ and the approximation $q_{\boldsymbol{\nu}}(\boldsymbol{\theta})$, we expect to obtain a good approximation of the posterior within the selected family of approximating distributions. Note that other divergence measures, most of which are generalizations of the KL divergence, have been studied in the literature, e.g., [50–52].

Since the KL divergence is not symmetric and, hence, does not fulfill the properties of a distance functional, we distinguish between the reverse and the forward KL divergence.[13] Perhaps the most common variational inference objective is the *reverse* KL divergence defined as

$$D_{\mathrm{KL}}(q_{\boldsymbol{\nu}}(\boldsymbol{\theta})\,||\,p(\boldsymbol{\theta}|\mathcal{D})) = \int_{\boldsymbol{\Theta}} q_{\boldsymbol{\nu}}(\boldsymbol{\theta}) \log \frac{q_{\boldsymbol{\nu}}(\boldsymbol{\theta})}{p(\boldsymbol{\theta}|\mathcal{D})} d\boldsymbol{\theta} = \mathbb{E}_{\boldsymbol{\theta} \sim q_{\boldsymbol{\nu}}(\boldsymbol{\theta})} \left[\log \frac{q_{\boldsymbol{\nu}}(\boldsymbol{\theta})}{p(\boldsymbol{\theta}|\mathcal{D})} \right]. \tag{3.25}$$

For the *forward* KL divergence, the roles of the posterior $p(\boldsymbol{\theta}|\mathcal{D})$ and the approximation $q(\boldsymbol{\theta})$ are reversed, i.e.,

$$D_{\mathrm{KL}}(p(\boldsymbol{\theta}|\mathcal{D})\,||\,q_{\boldsymbol{\nu}}(\boldsymbol{\theta})) = \mathbb{E}_{\boldsymbol{\theta} \sim p(\boldsymbol{\theta}|\mathcal{D})} \left[\log \frac{p(\boldsymbol{\theta}|\mathcal{D})}{q_{\boldsymbol{\nu}}(\boldsymbol{\theta})} \right]. \tag{3.26}$$

The reverse and the forward KL divergence result in substantially different approximation behaviors when used for variational inference. To see why this holds, it is instructive to consider necessary conditions for (3.25) and (3.26) to become small. On the one hand, for (3.25) to be small, $q_{\boldsymbol{\nu}}(\boldsymbol{\theta})$ must be small whenever $p(\boldsymbol{\theta}|\mathcal{D})$ is small. This results in an behavior called *zero-avoiding* or *mode-seeking*, where the approximation $q(\boldsymbol{\theta})$ concentrates around a mode and avoids putting mass in regions where $p(\boldsymbol{\theta}|\mathcal{D})$ is small. On the other hand, for (3.26) to be small, $q_{\boldsymbol{\nu}}(\boldsymbol{\theta})$ is not allowed to be small whenever $p(\boldsymbol{\theta}|\mathcal{D})$ is large. In this case, the resulting behavior is called *mass-covering*, where the approximation $q_{\boldsymbol{\nu}}(\boldsymbol{\theta})$ tends to put mass in all regions where the posterior $p(\boldsymbol{\theta}|\mathcal{D})$ exhibits a substantial amount of mass.

The two different behaviors are best understood by considering a unimodal approximation $q_{\boldsymbol{\nu}}(\boldsymbol{\theta})$ for a multimodal posterior $p(\boldsymbol{\theta}|\mathcal{D})$ where the modes are separated by sufficiently large regions of low density. This is illustrated in Figure 3.3. In this case, the mode-seeking approximation tends to cover the region around one of the modes and, therefore, potentially misses to

[13] We adopt the terminology from [53] for the meaning of the terms *forward* and *reverse*.

cover substantial mass spread around the remaining modes. The mass-covering behavior, on the other hand, would distribute its mass across all modes and, therefore, put a substantial amount of mass in low density regions between the modes. Which one of these two behaviors is desirable depends on the application at hand. For instance, when the approximation is used to generate samples from the posterior, the mass-covering approximation runs the risk of generating samples that are very unlikely under the true posterior $p(\boldsymbol{\theta}|\mathcal{D})$.

However, regardless of the desirable properties of the approximation $q_{\boldsymbol{\nu}}(\boldsymbol{\theta})$, the reverse KL divergence (3.25) is predominantly used in practice due to its computational convenience. For instance, it can be optimized using samples from $q_{\boldsymbol{\nu}}(\boldsymbol{\theta})$ (see below). This is typically not possible for the forward KL divergence (3.26) where the expectation is evaluated with respect to the intractable posterior $p(\boldsymbol{\theta}|\mathcal{D})$ for which generating independent samples is assumed to be difficult. However, there also exist algorithms that aim to minimize (3.26). For instance, the *expectation propagation* algorithm [54] can be seen as approximately minimizing the forward KL divergence by iteratively minimizing the forward KL divergence of local factors. Interestingly, when minimizing (3.26) in conjunction with the mean field assumption, it can be shown that the optimal individual factors $q_i(\theta_i)$ correspond to the marginals of the true posterior $p(\boldsymbol{\theta}|\mathcal{D})$ [55].

Traditional methods for optimizing the reverse KL divergence operate in a coordinate-wise manner. Consider an approximating distribution $q_{\boldsymbol{\nu}}(\boldsymbol{\theta})$ that adopts the mean field assumption, i.e., it factorizes into individual factors $q_i(\theta_i)$, each of which is governed by individual parameters $\boldsymbol{\nu}_i$. It can be shown [55] that for the reverse KL divergence, the optimal factor $q_i^*(\theta_i)$ must satisfy

$$\log q_i^*(\theta_i) = \mathbb{E}_{\boldsymbol{\theta}_{-i}\sim q_{-i}(\boldsymbol{\theta}_{-i})}\left[\log p(\boldsymbol{\theta}, \mathcal{D})\right] + \text{const}, \tag{3.27}$$

where the expectation is taken with respect to the distribution specified by the product of the remaining factors, i.e., $q_{-i}(\boldsymbol{\theta}_{-i}) = \prod_{j\neq i} q_j(\theta_j)$. Equation (3.27) admits a closed-form solution if $p(\boldsymbol{\theta}, \mathcal{D})$ and $q_{\boldsymbol{\nu}}(\boldsymbol{\theta})$ are selected appropriately, e.g., if they satisfy certain conjugacy assumptions [56, 57]. This suggests an iterative coordinate ascent algorithm where the parameters $\boldsymbol{\nu}_i$ of each factor $q_i(\theta_i)$ are updated in turn. These updates do not increase the KL divergence such that convergence of the iterative algorithm is guaranteed.

However, these closed-form coordinate ascent updates are model specific and their derivation is typically cumbersome and error-prone, hindering rapid development of modeling assumptions. Furthermore, the class of models for which (3.27) admits closed-form solutions is limited. To obtain a more generally applicable approach, note that it is straightforward to evaluate the posterior $p(\boldsymbol{\theta}|\mathcal{D})$ up to the intractable normalization constant, the evidence $p(\mathcal{D})$. The reverse KL divergence can be rephrased by separating the intractable evidence $p(\mathcal{D})$ from terms that only depend on the variational parameters $\boldsymbol{\nu}$. More specifically, by applying Bayes' rule to the posterior $p(\boldsymbol{\theta}|\mathcal{D})$ in (3.25) and rearranging terms we obtain

$$D_{\mathrm{KL}}(q_{\boldsymbol{\nu}}(\boldsymbol{\theta})\,||\,p(\boldsymbol{\theta}|\mathcal{D})) = -\mathbb{E}_{\boldsymbol{\theta}\sim q_{\boldsymbol{\nu}}(\boldsymbol{\theta})}\left[\log p(\mathcal{D}|\boldsymbol{\theta})\right] + D_{\mathrm{KL}}(q_{\boldsymbol{\nu}}(\boldsymbol{\theta})\,||\,p(\boldsymbol{\theta})) + \log p(\mathcal{D}). \tag{3.28}$$

When optimizing (3.28) with respect to the variational parameters $\boldsymbol{\nu}$, the intractable term $\log p(\mathcal{D})$ is a constant that can be ignored. Moreover, the KL divergence between the approximation $q_{\boldsymbol{\nu}}(\boldsymbol{\theta})$ and the prior $p(\boldsymbol{\theta})$ admits a closed-form expression for many commonly used distributions (e.g., if both are Gaussians). The remaining intractable part is the expected log-likelihood term. This term, however, can be approximated using samples from $q_{\boldsymbol{\nu}}(\boldsymbol{\theta})$, allowing us to perform SGD using Monte Carlo gradients of (3.28). This is interesting since we cannot approximate (3.28) itself using Monte Carlo methods due to the evidence $p(\mathcal{D})$. This approach is sometimes also called black box variational inference [58]. These methods require fewer assumptions on $p(\boldsymbol{\theta}, \mathcal{D})$, enabling variational inference for many different kinds of models. In Section 3.4, we show how to apply this method to perform variational inference for Bayesian DNNs.

Variational inference has been an active research area for the past twenty years and we refer

to [59] for a comprehensive overview. Besides applying variational inference to different kinds of probabilistic models and investigating the optimization of new divergence measures, much work has been devoted to more expressive approximating distributions $q_{\boldsymbol{\nu}}(\boldsymbol{\theta})$ that do not rely on the mean field assumption. For instance, *normalizing flows* [60, 61] transform a simple distribution, such as a Gaussian with zero mean and unit variance, by a sequence of invertible transformations to obtain a more complicated distribution. The variational parameters $\boldsymbol{\nu}$ of the normalizing flow correspond to the learnable parameters of the invertible transformations. Normalizing flows have the advantage that in the limit of an infinitely long sequence of invertible transformations they are able to model arbitrary distributions [60], reducing the systematic bias of variational inference due to an inexpressive approximation $q_{\boldsymbol{\nu}}(\boldsymbol{\theta})$.

Note that instead of viewing variational inference in terms of minimizing a KL divergence, one can also view variational inference as the maximization of a lower bound. In particular, by applying the non-negativity of the KL divergence to the left hand side of (3.28), we obtain the evidence lower bound

$$\log p(\mathcal{D}) \geq \mathbb{E}_{\boldsymbol{\theta} \sim q_{\boldsymbol{\nu}}(\boldsymbol{\theta})} \left[\log p(\mathcal{D} \,|\, \boldsymbol{\theta})\right] - D_{\mathrm{KL}}(q_{\boldsymbol{\nu}}(\boldsymbol{\theta}) \,||\, p(\boldsymbol{\theta})). \tag{3.29}$$

The right hand side of (3.29) is often maximized as a surrogate for the intractable marginal log-likelihood (or evidence), e.g., for the training of variational autoencoders [62].

Example: ML Estimation as Variational Inference

Interestingly, ML estimation as discussed in Section 3.2.1 also arises by minimizing a particular kind of KL divergence. Consider the KL divergence between the data joint distribution $p_{\mathrm{data}}(\mathbf{x}, \mathbf{y})$ and the model joint distribution $p(\mathbf{x}, \mathbf{y} \,|\, \boldsymbol{\theta})$ governed by parameters $\boldsymbol{\theta}$, i.e.,

$$\begin{aligned} D_{\mathrm{KL}}(p_{\mathrm{data}}(\mathbf{x}, \mathbf{y}) \,||\, p(\mathbf{x}, \mathbf{y} \,|\, \boldsymbol{\theta})) &= \int p_{\mathrm{data}}(\mathbf{x}, \mathbf{y}) \log \frac{p_{\mathrm{data}}(\mathbf{x}, \mathbf{y})}{p(\mathbf{x}, \mathbf{y} \,|\, \boldsymbol{\theta})} \, d\mathbf{x} \, d\mathbf{y} \\ &= -\, \mathbb{H}\left[p_{\mathrm{data}}(\mathbf{x}, \mathbf{y})\right] - \mathbb{E}_{(\mathbf{x}, \mathbf{y}) \sim p_{\mathrm{data}}} \left[\log p(\mathbf{x}, \mathbf{y} \,|\, \boldsymbol{\theta})\right], \end{aligned} \tag{3.30}$$

where $\mathbb{H}$ denotes the (differential) entropy of a distribution. The entropy term is constant with respect to the parameters $\boldsymbol{\theta}$ and can be ignored for minimization. Note that the expected log-likelihood cannot be computed since the true data generating distribution p_{data} is assumed to be unknown, but it can be approximated by a Monte Carlo sum using the observed dataset $\mathcal{D}$ as

$$\mathbb{E}_{(\mathbf{x}, \mathbf{y}) \sim p_{\mathrm{data}}} \left[\log p(\mathbf{x}, \mathbf{y} \,|\, \boldsymbol{\theta})\right] \approx \frac{1}{N} \left(\sum_{n=1}^{N} \log p(\mathbf{y}_n \,|\, \mathbf{x}_n, \boldsymbol{\theta}) + \log p(\mathbf{x}_n \,|\, \boldsymbol{\theta}) \right). \tag{3.31}$$

Assuming that we do not model the input distribution $p(\mathbf{x}_n \,|\, \boldsymbol{\theta})$, we recover the MAP objective from (3.15) with an assumed uniform prior $p(\boldsymbol{\theta})$, which in turn corresponds to the ML objective. This connection between ML estimation and KL divergence minimization is interesting since the properties of the particular kind of KL divergence minimization (forward or backward) directly transfer to the ML framework. In particular, (3.30) results in a mass-covering behavior. Note that the reversed KL divergence for this example cannot be easily minimized since the required density of the true data generating distribution p_{data} is assumed to be unknown.

3.2.4 Sampling Methods

Besides variational inference, the second pillar of approximate Bayesian inference are sampling methods. These methods approach the problem in a completely different way. Assume that we

want to approximate an intractable distribution $p(\mathbf{z})$.[14] Whereas variational inference approximates $p(\mathbf{z})$ by a simpler distribution $q(\mathbf{z})$ using optimization, the aim of sampling methods is to simulate artificial instantiations $\mathbf{z}$, called samples, from $p(\mathbf{z})$. The resulting samples are then expected to be drawn from regions where $p(\mathbf{z})$ exhibits substantial mass. In the limit of infinitely many samples, the proportion of samples falling into a particular region becomes equal to the probability that $p(\mathbf{z})$ assigns to that region.

It is convenient to think of the collection of the drawn samples $\mathbf{z}^1, \dots, \mathbf{z}^M$ as a mixture $q(\mathbf{z})$ of equally probable delta distributions that approximates the intractable distribution $p(\mathbf{z})$, i.e.,

$$p(\mathbf{z}) \approx q(\mathbf{z}) = \frac{1}{M} \sum_{i=1}^{M} \delta_{\mathbf{z}^i}(\mathbf{z}). \tag{3.32}$$

This allows us to approximate intractable expectations with respect to the original distribution by averaging over these samples, i.e.,

$$\mathbb{E}_{\mathbf{z} \sim p(\mathbf{z})}\left[f(\mathbf{z})\right] \approx \frac{1}{M} \sum_{i=1}^{M} f(\mathbf{z}^i). \tag{3.33}$$

Given that the samples $\mathbf{z}^i$ are independent, the variance of the approximated expectation (3.33) decreases linearly with the number of samples M. Importantly, this holds irrespective of the number of dimensions of $\mathbf{z}$. However, generating *independent* samples tends to be difficult if exact Bayesian inference is intractable.

Note that generating independent samples from many common distributions is tractable, e.g., the uniform, Gaussian, exponential, beta, gamma, or the Gumbel distribution. Many sampling methods for these distributions operate by transforming a sample from a different distribution for which a sampling algorithm is already available. For instance, if the inverse cumulative distribution function (cdf) Φ_p^{-1} of a distribution $p(z)$ is available in closed form, we can generate samples by evaluating $\Phi_p^{-1}(\varepsilon)$ for samples $\varepsilon \sim \mathcal{U}([0,1])$ from a uniform distribution.

We will omit details about sampling from these common distributions and start our discussion with the general Markov chain Monte Carlo (MCMC) algorithm that is widely used in the literature. Then we proceed with two special instances of the MCMC algorithm that are relevant for the remainder of this thesis, namely Gibbs sampling and Hamiltonian Monte Carlo (HMC). These two algorithms are located on opposing extremes of the MCMC spectrum in the sense that Gibbs sampling only updates one variable at a time, whereas HMC updates many (if not all) variables simultaneously. We note that there exist stochastic sampling methods that, similarly to SGD, rely only on a subset of the data to generate samples. These methods are discussed in the context of DNNs in Section 3.5.

Markov Chain Monte Carlo

A Markov chain is a stochastic process defined as a sequence of random variables $\{\mathbf{z}^t\}_{t \geq 1}$ where the concrete instantiations $\mathbf{z}^t$ only depend on their immediate predecessors $\mathbf{z}^{t-1}$ according to a transition probability $p(\mathbf{z}^t \,|\, \mathbf{z}^{t-1})$. Under mild assumptions, a Markov chain possesses a unique *stationary distribution* $p(\mathbf{z})$ such that by simulating the Markov chain for long enough—i.e., by iterated sampling from $p(\mathbf{z}^t \,|\, \mathbf{z}^{t-1})$—the samples $\mathbf{z}^t$ will eventually be distributed according to $p(\mathbf{z})$ as $t \to \infty$.[15] Most importantly, this holds irrespective of the initial value $\mathbf{z}^1$. Consequently, given an appropriately constructed Markov chain whose stationary distribution corresponds to $p(\mathbf{z})$, we can generate samples by performing a random walk and reporting $\mathbf{z}^t$ after a sufficiently large number of steps t. MCMC is a class of sampling algorithms that follow exactly this idea.

[14] The distribution $p(\mathbf{z})$ is an arbitrary distribution and therefore may also represent a posterior distribution $p(\boldsymbol{\theta} \,|\, \mathcal{D})$. We use $p(\mathbf{z})$ in this section to keep notation uncluttered.

[15] The process of simulating the Markov chain is also called a *random walk*.

The question remains how to construct a Markov chain whose unique stationary distribution corresponds to $p(\mathbf{z})$. A stationary distribution $p(\mathbf{z})$ is characterized by the property that it remains *invariant* when applying a random step according to the transition probability $p(\mathbf{z}^t \,|\, \mathbf{z}^{t-1})$, i.e.,

$$p(\mathbf{z}^t) = \sum_{\mathbf{z}^{t-1}} p(\mathbf{z}^{t-1}) p(\mathbf{z}^t \,|\, \mathbf{z}^{t-1}). \tag{3.34}$$

Since $p(\mathbf{z})$ is fixed, we need to select transition probabilities $p(\mathbf{z}^t \,|\, \mathbf{z}^{t-1})$ such that $p(\mathbf{z})$ is a stationary distribution. A sufficient condition satisfying the stationarity property is the *detailed balance* condition defined by

$$p(\mathbf{z}^{t-1}) p(\mathbf{z}^t \,|\, \mathbf{z}^{t-1}) = p(\mathbf{z}^t) p(\mathbf{z}^{t-1} \,|\, \mathbf{z}^t). \tag{3.35}$$

It is straightforward to show that the detailed balance condition implies stationarity according to (3.34), i.e.,

$$\sum_{\mathbf{z}^{t-1}} p(\mathbf{z}^{t-1}) p(\mathbf{z}^t \,|\, \mathbf{z}^{t-1}) = p(\mathbf{z}^t) \sum_{\mathbf{z}^{t-1}} p(\mathbf{z}^{t-1} \,|\, \mathbf{z}^t) = p(\mathbf{z}^t). \tag{3.36}$$

It remains to show that after simulating the Markov chain for long enough, the observed samples will be distributed according to the stationary distribution regardless of the initial value $\mathbf{z}_1$, i.e.,

$$p(\mathbf{z}^t \,|\, \mathbf{z}^1) \xrightarrow{t \to \infty} p(\mathbf{z}). \tag{3.37}$$

Fortunately, this holds under relatively mild assumptions [63]. Most notably, the Markov chain must be irreducible such that, intuitively speaking, each state is reachable with positive probability from every other state within a finite number of steps. Furthermore, the Markov chain must be time-homogeneous such that the transition probabilities $p(\mathbf{z}^t \,|\, \mathbf{z}^{t-1})$ do not depend on the time t, i.e., $p(\mathbf{z}^t \,|\, \mathbf{z}^{t-1}) = p(\mathbf{z}^{t+1} | \mathbf{z}^t)$ must hold for all t. These results transfer to continuous space Markov chains, albeit slightly more care needs to be taken in the respective proofs [63].

The *Metropolis-Hastings algorithm* [64] provides a general scheme to construct a Markov chain that satisfies the detailed balance condition. Let $\hat{p}(\mathbf{z}^t \,|\, \mathbf{z}^{t-1})$ be an arbitrary *proposal distribution* that depends on the state $\mathbf{z}^{t-1}$ and from which we can efficiently draw samples. The idea of the Metropolis-Hastings algorithm is to generate a sample $\mathbf{z}^t$ from the proposal distribution $\hat{p}(\mathbf{z}^t \,|\, \mathbf{z}^{t-1})$ and to accept this sample with an *acceptance probability* given by

$$\mathrm{acc}(\mathbf{z}^t, \mathbf{z}^{t-1}) = \min \left\{ 1, \frac{p(\mathbf{z}^t) \hat{p}(\mathbf{z}^{t-1} \,|\, \mathbf{z}^t)}{p(\mathbf{z}^{t-1}) \hat{p}(\mathbf{z}^t \,|\, \mathbf{z}^{t-1})} \right\}. \tag{3.38}$$

If the sample gets rejected, we set $\mathbf{z}^t = \mathbf{z}^{t-1}$. It is straightforward to show that this scheme satisfies the detailed balance condition (3.35), i.e.,

$$\begin{aligned} p(\mathbf{z}^{t-1}) p(\mathbf{z}^t \,|\, \mathbf{z}^{t-1}) &= p(\mathbf{z}^{t-1}) \hat{p}(\mathbf{z}^t \,|\, \mathbf{z}^{t-1}) \,\mathrm{acc}(\mathbf{z}^t, \mathbf{z}^{t-1}) & (3.39) \\ &= p(\mathbf{z}^{t-1}) \hat{p}(\mathbf{z}^t \,|\, \mathbf{z}^{t-1}) \min \left\{ 1, \frac{p(\mathbf{z}^t) \hat{p}(\mathbf{z}^{t-1} \,|\, \mathbf{z}^t)}{p(\mathbf{z}^{t-1}) \hat{p}(\mathbf{z}^t \,|\, \mathbf{z}^{t-1})} \right\} & (3.40) \\ &= \min \left\{ p(\mathbf{z}^{t-1}) \hat{p}(\mathbf{z}^t \,|\, \mathbf{z}^{t-1}), p(\mathbf{z}^t) \hat{p}(\mathbf{z}^{t-1} \,|\, \mathbf{z}^t) \right\}. & (3.41) \end{aligned}$$

Reversing steps (3.39)–(3.41) with the role of $\mathbf{z}^t$ and $\mathbf{z}^{t-1}$ interchanged yields the desired result.

A remarkable property of the Metropolis-Hastings algorithm is that we only need to know the distribution $p(\mathbf{z})$ up to a normalizing constant. This property can be easily seen from (3.38) as the potentially unknown normalization constant of $p(\mathbf{z})$ would simply cancel. In practice, we often encounter distributions that are only available up to a normalizing constant. For instance,

it is easy to evaluate an expression that is proportional to the posterior distribution (3.3), but computing the normalization constant itself is typically intractable.

This already gives us a powerful tool to perform approximate posterior inference. However, the efficiency of the algorithm depends crucially on the particular proposal distribution $\hat{p}(\mathbf{z}^t | \mathbf{z}^{t-1})$. A good proposal distribution must satisfy two opposing properties. First, it must be exploratory in the sense that consecutive samples in our Markov chain are decorrelated as much as possible. Recall that samples are required to be drawn *independently* in order to obtain a linear reduction of the variance of a Monte Carlo estimator. Second, it should induce a high acceptance probability. In practice, one typically has to make a trade-off by proposing samples $\mathbf{z}^t$ that are somewhat close to the previous samples $\mathbf{z}^{t-1}$ to achieve reasonable acceptance rates. If samples are highly correlated, it is sometimes convenient to subsample the sequence of generated samples to reduce their correlation by only collecting samples after every few steps. However, this is not necessary from a theoretical viewpoint and the computational cost for computing intermediate steps persists.

Another practical consideration is the choice of the initial value $\mathbf{z}^1$. MCMC convergences to the stationary distribution for arbitrary initial values $\mathbf{z}^1$ and it is common to select it at random. In this case $\mathbf{z}^1$ is often not representative of $p(\mathbf{z})$. Therefore, a *burn-in* phase is required where the first few samples of the Markov chain are discarded. However, determining how long the burn-in phase should last and when the stationary distribution is reached are questions that are difficult to answer in practice.

In the following, we discuss two particular sampling algorithms, both of which can be seen as special cases of the Metropolis-Hastings algorithm. The first one, Gibbs sampling, employs a proposal distribution that only samples one dimension at a time conditioned on all the others. The second one, HMC, employs the gradient of the log-density of $p(\mathbf{z})$ to update all variables at a time while being able to maintain low rejection rates.

Gibbs Sampling

Gibbs sampling is a conceptually simple MCMC algorithm that iteratively updates individual dimensions z_i of the state $\mathbf{z}$ conditioned on all the remaining dimensions $\mathbf{z}_{-i}$ by sampling from $p(z_i | \mathbf{z}_{-i})$. In this process, the algorithm remains valid regardless of the particular order in which individual dimensions i are selected. In particular, we can cycle over the individual dimensions in a predefined order or we can select individual dimensions at random in each step.

We start our discussion by showing that Gibbs sampling is a valid MCMC algorithm. To show that each step leaves the stationary distribution $p(\mathbf{z})$ invariant, assume that $\mathbf{z}$ is already drawn from $p(\mathbf{z})$. After applying a single step to update z_i, the marginal $p(\mathbf{z}_{-i})$ remains invariant since $\mathbf{z}_{-i}$ remains unchanged. Since each step updates z_i by sampling from $p(z_i | \mathbf{z}_{-i})$, and the product of $p(\mathbf{z}_{-i})$ and $p(z_i | \mathbf{z}_{-i})$ equals the joint distribution $p(\mathbf{z})$, we obtain a valid sample from $p(\mathbf{z})$.

To show that the Markov chain also converges to the stationary distribution irrespective of its initial state, we assume that a dimension i is selected randomly with uniform probability in each step. Then the Markov chain is obviously time-homogeneous. A sufficient condition for the second major condition, irreducibility of the Markov chain, is that each conditional distribution satisfies $p(z_i | \mathbf{z}_{-i}) > 0$ for all values of z_i. If this condition holds, each state $\mathbf{z}$ can be reached from every other state $\mathbf{z}'$ after cycling once over the individual dimensions. Although this sufficient condition often holds in practice, one might have to prove explicitly that the constructed Markov chain is irreducible in the rare case where this condition does not hold.

Interestingly, the Gibbs sampling algorithm can be seen as a special instance of the Metropolis-Hastings algorithm discussed above. From the viewpoint of the Metropolis-Hastings algorithm, the proposal distribution $\hat{p}(\mathbf{z}^t | \mathbf{z}^{t-1})$ is given by the conditional distribution $p(z_i^t | \mathbf{z}_{-i}^{t-1})$ where i

is the dimension currently being updated. The acceptance probability is then given by

$$\begin{aligned}\operatorname{acc}(\mathbf{z}^t, \mathbf{z}^{t-1}) &= \min\left\{1, \frac{p(\mathbf{z}^t)\, p(z_i^{t-1}\,|\,\mathbf{z}_{-i}^t)}{p(\mathbf{z}^{t-1})\, p(z_i^t\,|\,\mathbf{z}_{-i}^{t-1})}\right\} \\ &= \min\left\{1, \frac{p(z_i^t\,|\,\mathbf{z}_{-i}^t)\, p(\mathbf{z}_{-i}^t)\, p(z_i^{t-1}\,|\,\mathbf{z}_{-i}^t)}{p(z_i^{t-1}\,|\,\mathbf{z}_{-i}^{t-1})\, p(\mathbf{z}_{-i}^{t-1})\, p(z_i^t\,|\,\mathbf{z}_{-i}^{t-1})}\right\} = 1, \end{aligned} \tag{3.42}$$

where we have used $\mathbf{z}_{-i}^{t-1} = \mathbf{z}_{-i}^t$ in the last equation.

We have shown that Gibbs sampling can be used to reduce MCMC in a potentially high-dimensional state space to iterated sampling in a one-dimensional space. Note that the conditional distribution $p(z_i\,|\,\mathbf{z}_{-i})$ is proportional to the joint distribution $p(z_i, \mathbf{z}_{-i})$ if $\mathbf{z}_{-i}$ is considered fixed. Given that the joint distribution $p(\mathbf{z})$ factorizes into a product of several terms, the conditional $p(z_i\,|\,\mathbf{z}_{-i})$ is fully specified by the factors that depend on z_i. For instance, when the joint distribution $p(\mathbf{z})$ is specified by a BN, it suffices to only consider the Markov blanket of z_i. This is particularly convenient for finite discrete distributions, i.e., one-dimensional distributions are not subject to the curse of dimensionality and the conditional distribution $p(z_i\,|\,\mathbf{z}_{-i})$ is obtained by evaluating the joint distribution $p(z_i, \mathbf{z}_{-i})$ for every possible value of z_i followed by renormalization. For continuous state spaces, the form of the conditional $p(z_i\,|\,\mathbf{z}_{-i})$ depends on the joint distribution $p(\mathbf{z})$. In practice, the joint distribution $p(\mathbf{z})$ is often designed to exhibit certain conjugacy properties that induce a particular form of the conditional $p(z_i\,|\,\mathbf{z}_{-i})$ from which we can easily draw samples. If this is not the case, there still exist efficient algorithms for sampling from continuous one-dimensional distributions that are only known up to a normalizing constant, such as slice sampling [65].

Due to its simplicity and the fact that it is typically easy to sample from one-dimensional distributions, Gibbs sampling is widely applicable. However, on the downside, the generated samples are highly correlated since consecutive samples can only differ in a single dimension and traversing the state space might take very long. A generalization of Gibbs sampling, called *block Gibbs sampling*, partially solves this issue by sampling from a larger subset of dimensions in each step. For some models it is possible to split the dimensions into subsets such that it is easy to sample from the joint distribution of each subset conditioned on all the other dimensions. For instance, the states of a restricted Boltzmann machine can be partitioned into visible and hidden states, and conditionally sampling each subset jointly given the other is easy [66]. In Chapter 6, we introduce a block Gibbs sampling algorithm where we split the state space into discrete and continuous states. The discrete dimensions of the posterior are then sampled using one-dimensional Gibbs sampling and the continuous dimensions are sampled jointly using HMC.

Another means of reducing the correlation among consecutive samples is provided by *collapsed Gibbs sampling*. The idea of collapsed Gibbs sampling is to reduce the dimensionality of the state space by analytically integrating out certain dimensions. Performing Gibbs sampling on the remaining dimensions then makes faster progress through the state space. This idea has, for instance, been successfully applied to the latent Dirichlet allocation model [67] in [68]. Collapsing certain dimensions of the state space (i.e., the mixture probabilities $\boldsymbol{\pi}$) is an integral part of our proposed Gibbs sampling scheme in Chapter 6.

To conclude our discussion about Gibbs sampling, we want to highlight the similarity between Gibbs sampling and the coordinate ascent variational inference algorithm based on (3.27). The major differences between both algorithms are that Gibbs sampling maintains concrete instantiations and updates the values by sampling, whereas coordinate ascent variational inference maintains distributions and updates the distribution parameters by evaluating expectations.

Sampling with Gradient Information: Hamiltonian Monte Carlo

The HMC algorithm [55, 69, 70] is a sampling algorithm for continuous distributions. It utilizes the gradient of the log-density to make larger steps which reduces the correlation between

consecutive samples and enables a fast exploration of the state space. The idea of using gradients bears resemblance to continuous optimization techniques such as gradient descent that typically outperform methods that only have access to the function itself (see Section 2.1.2).

HMC belongs to a class of algorithms called *auxiliary variable MCMC*. The idea of auxiliary variable methods is to augment the state space by auxiliary variables $\mathbf{u}$ and to sample from the joint distribution $p(\mathbf{z}, \mathbf{u})$ in the augmented space. The joint distribution $p(\mathbf{z}, \mathbf{u})$ must satisfy the property that its marginal $p(\mathbf{z})$ corresponds to the original distribution we want to generate samples from. Then, by simply discarding the auxiliary variables $\mathbf{u}$ from the generated samples $(\mathbf{z}, \mathbf{u})$, we obtain valid samples $\mathbf{z}$ from the distribution $p(\mathbf{z})$ that we care about. Note that this is contrary to previously discussed methods, such as collapsed Gibbs sampling, that reduce the number of dimensions to increase efficiency. However, as we will see, properties that hold in the augmented space can be exploited to obtain a sampling scheme with desirable properties that let us explore the original state space more efficiently. Some other popular auxiliary variable sampling algorithms are slice sampling [65] and the Swendsen-Wang algorithm [71].

In the context of HMC, it is common to write the distribution $p(\mathbf{z})$ as

$$p(\mathbf{z}) = \frac{1}{Z_E} \exp\left(-E(\mathbf{z})\right), \tag{3.43}$$

where Z_E is a normalization constant ensuring that $p(\mathbf{z})$ integrates to one and $E(\mathbf{z})$ is the unnormalized negative log-density of $p(\mathbf{z})$. Here, the term $E(\mathbf{z})$ is also called the *potential energy*. Assuming $\mathbf{z} \in \mathbb{R}^D$, we introduce the auxiliary variables $\mathbf{u} \in \mathbb{R}^D$ called *momentum variables*. We can then define the *kinetic energy* as

$$K(\mathbf{u}) = \mathbf{u}^\top \mathbf{u}/2. \tag{3.44}$$

Finally, we define the Hamiltonian function $H(\mathbf{z}, \mathbf{u})$ as the sum of the potential and the kinetic energy, i.e.,

$$H(\mathbf{z}, \mathbf{u}) = E(\mathbf{z}) + K(\mathbf{u}), \tag{3.45}$$

to obtain the joint density

$$p(\mathbf{z}, \mathbf{u}) = \frac{1}{Z_H} \exp\left(-H(\mathbf{z}, \mathbf{u})\right). \tag{3.46}$$

For the joint distribution (3.46), the variables $\mathbf{z}$ and $\mathbf{u}$ are independent such that, as required by auxiliary variable methods, the marginal $p(\mathbf{z})$ corresponds to the distribution we care about. Furthermore, from (3.44) we can see that $p(\mathbf{u})$ is given by an isotropic Gaussian with zero mean and unit variance.

The HMC algorithm alternates between two sampling steps which together form a Markov chain whose stationary distribution corresponds to (3.46). In the first step, we perform a block Gibbs update for the momentum variables $\mathbf{u}$. Since $\mathbf{z}$ and $\mathbf{u}$ are independent, sampling from the conditional $p(\mathbf{u}\,|\,\mathbf{z})$ is equivalent to sampling from the Gaussian marginal $p(\mathbf{u})$.

In the second step, the joint state $(\mathbf{z}, \mathbf{u})$ is updated according to a Metropolis-Hastings step. In particular, we perform a finite-time simulation of Hamiltonian dynamics according to the Hamiltonian equations

$$\frac{dz_i}{dt} = \frac{\partial H}{\partial u_i} \qquad \text{and} \qquad \frac{du_i}{dt} = -\frac{\partial H}{\partial z_i}. \tag{3.47}$$

It can be shown that the Hamiltonian $H(\mathbf{z}, \mathbf{u})$ remains constant when simulating the joint state according to (3.47). Therefore, this simulation will always be accepted according to the Metropolis-Hastings acceptance probability. For all but very simple distributions, however,

Algorithm 5 Hamiltonian Monte Carlo (HMC)

1: **Input:** $E(\mathbf{z}) \coloneqq -\log p(\mathbf{z}) + \text{const}$, $\mathbf{z}^0$, T, η
2: **for** $t = 1$ **to** ... **do**
3: Draw $\mathbf{u} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
4: $\bar{\mathbf{z}}^0 \leftarrow \mathbf{z}^{t-1}$
5: $\bar{\mathbf{u}}^0 \leftarrow \mathbf{u} - (\eta/2)\nabla_{\mathbf{z}} E(\bar{\mathbf{z}}^0)$
6: **for** $k = 1$ **to** T **do**
7: $\bar{\mathbf{z}}^k \leftarrow \bar{\mathbf{z}}^{k-1} + \eta \bar{\mathbf{u}}^{k-1}$
8: $\bar{\mathbf{u}}^k \leftarrow \bar{\mathbf{u}}^{k-1} - \eta \nabla_{\mathbf{z}} E(\bar{\mathbf{z}}^k)$
9: **end for**
10: $\bar{\mathbf{u}}^T \leftarrow \bar{\mathbf{u}}^T + (\eta/2)\nabla_{\mathbf{z}} E(\bar{\mathbf{z}}^T)$ # correct for last half step
11: Draw $\varepsilon \sim \mathcal{U}([0, 1])$
12: **if** $\varepsilon < \min\{1, \exp(E(\mathbf{z}^{t-1}) + K(\mathbf{u}) - E(\bar{\mathbf{z}}^T) - K(\bar{\mathbf{u}}^T))\}$ **then**
13: $\mathbf{z}^t \leftarrow \bar{\mathbf{z}}^T$ # accept sample
14: **else**
15: $\mathbf{z}^t \leftarrow \mathbf{z}^{t-1}$ # reject sample
16: **end if**
17: **end for**

we are unable to simulate (3.47) analytically, and we must resort to numerical approximations. Such approximations will necessarily introduce numerical errors and the value of the Hamiltonian $H(\mathbf{z}, \mathbf{u})$ might change. For the HMC algorithm, it is common to apply the *leapfrog algorithm* to perform the numerical simulation of (3.47). The leapfrog algorithm simulates (3.47) by iteratively computing

$$\mathbf{u}(t + \eta/2) = \mathbf{u}(t) - (\eta/2)\nabla_{\mathbf{z}} E(\mathbf{z}(t)) \tag{3.48}$$

$$\mathbf{z}(t + \eta) = \mathbf{z}(t) + \eta \mathbf{u}(t + \eta/2) \tag{3.49}$$

$$\mathbf{u}(t + \eta) = \mathbf{u}(t + \eta/2) - (\eta/2)\nabla_{\mathbf{z}} E(\mathbf{z}(t + \eta)) \tag{3.50}$$

for a fixed number of iterations $T > 0$, where $\eta > 0$ is a fixed step size. The leapfrog algorithm is more accurate than a simple time discretization and satisfies important properties to show the correctness of HMC, namely the preservation of volume in the joint space $(\mathbf{z}, \mathbf{u})$ and the property that the leapfrog iteration is time reversible. The leapfrog algorithm in combination with a Metropolis-Hastings acceptance step to account for numerical errors ensures that the algorithm asymptotically samples from the correct distribution. Note that resampling the momentum variables $\mathbf{u}$ in the first step is an integral part of the algorithm as otherwise the algorithm would only explore points of equal $H(\mathbf{z}, \mathbf{u})$.

The complete HMC algorithm is shown in Algorithm 5. The first step of HMC, resampling the momentum $\mathbf{u}$, happens in line 3. The leapfrog algorithm is performed in lines 5–10. The shown implementation utilizes the fact that by applying (3.48)–(3.50) in succession, the half steps (3.48) and (3.50) can be combined to full steps and half steps remain only once at the beginning and once at the end. The Metropolis-Hastings acceptance step to account for numerical errors is performed in lines 11–16.

It is intuitive to think of the algorithm as simulating a ball that moves on a surface defined by $E(\mathbf{z})$ where the ball's velocity $\mathbf{u}$ changes according to the laws of motion. For instance, when the ball is moving downwards, its kinetic energy $K(\mathbf{u})$ increases while its potential energy $E(\mathbf{z})$ decreases such that both energies maintain balance at all times. Hence, the HMC algorithm repeatedly pushes the ball into a random direction $\mathbf{u}$ (the first step) and reports the position $\mathbf{z}$ of the ball after simulating the laws of motion for a finite time. This scheme allows us to make substantially larger steps than most other MCMC methods based on simpler proposal

distributions, mitigating random walk behavior and resulting in weaker correlation between consecutive samples.

In practice, the efficiency of the algorithm depends crucially on the choice of the two leapfrog parameters η and T. Ideally, the leapfrog algorithm satisfies three competing objectives, i.e., (i) small numerical errors to maintain high Metropolis-Hastings acceptance rates, (ii) large progress in the state space to obtain weakly correlated samples, and (iii) little computation time. To see why this is challenging, we need to consider the effect of setting η and T in particular ways. For small η, we can ensure small numerical errors, but the progress through the state space might be insufficiently small. For small T, we obtain a fast algorithm (the number of iterations directly relates to the computation time) and achieve small numerical errors, but exploration of the state space might again make insufficient progress. Some algorithms have been proposed to automate the selection of η and T, e.g., adaptive Hamiltonian Monte Carlo (AHMC) [72] or the No-U-Turn sampler [73]. In Chapter 6, we employ AHMC to sample from the continuous space of DNN weights.

Many distributions encountered in practice correspond to posterior distributions whose log-density is given by a sum over the whole dataset $\mathcal{D}$. As discussed in the context of stochastic optimization (see Section 2.1.2), this becomes difficult for very large datasets. In Section 3.5, we discuss how sampling methods can be generalized to the stochastic gradient framework similarly to how gradient descent has been generalized to SGD.

3.3 Bayesian Deep Neural Networks

In Chapter 2, we have discussed how to compute a point estimate of the DNN weights $\mathbf{W}$ by means of optimization (see Figure 3.4(a)). Subsequently, predictions are computed solely based on these weights. Here, we consider distributions over the weights (see Figure 3.4(b)). In particular, the aim of Bayesian DNNs is to infer the posterior distribution over the weights $\mathbf{W}$ using Bayesian inference, i.e.,

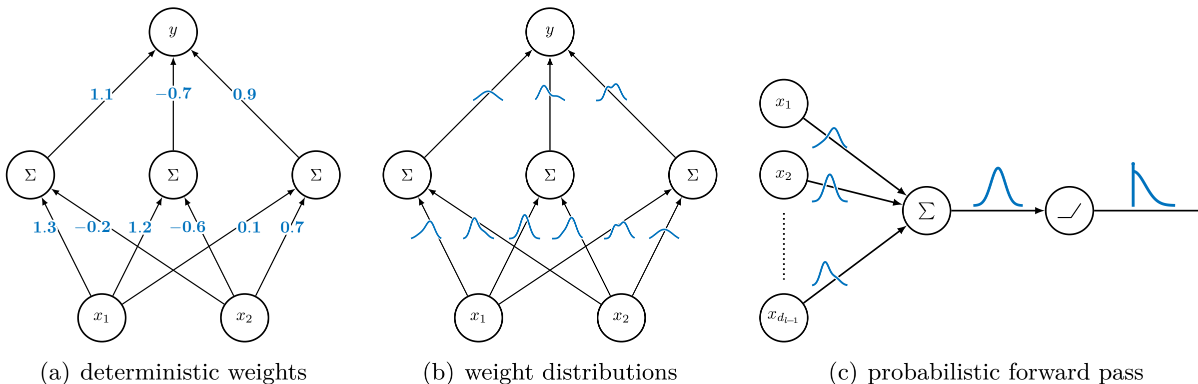
$$p(\mathbf{W} \mid \mathcal{D}) = \frac{p(\mathcal{D} \mid \mathbf{W}) p(\mathbf{W})}{p(\mathcal{D})} \propto p(\mathcal{D} \mid \mathbf{W}) p(\mathbf{W}). \tag{3.51}$$

Subsequently, predictions are computed by considering the whole weight space using expectations with respect to this posterior, i.e.,

$$p(\mathbf{y} \mid \mathbf{x}, \mathcal{D}) = \mathbb{E}_{\mathbf{W} \sim p(\mathbf{W} \mid \mathcal{D})}[p(\mathbf{y} \mid \mathbf{x}, \mathbf{W})] = \int p(\mathbf{y} \mid \mathbf{x}, \mathbf{W}) p(\mathbf{W} \mid \mathcal{D}) \, d\mathbf{W}. \tag{3.52}$$

Using this procedure, DNNs inherit all the favorable properties of Bayesian inference such as less overfitting and the ability to compute meaningful prediction uncertainties. The Bayesian framework requires us to select a prior distribution $p(\mathbf{W})$ and a likelihood function $p(\mathcal{D} \mid \mathbf{W})$. We employ the likelihood functions discussed in Section 3.2.1. In particular, we assume Gaussian outputs (3.16) for regression, and Bernoulli (3.18) and categorical outputs (3.20) for binary and multiclass classification, respectively.

For the prior $p(\mathbf{W})$, there are several reasonable options in the context of DNNs. Without specific knowledge about the weights $\mathbf{W}$, the prior $p(\mathbf{W})$ is typically selected to be an isotropic Gaussian with zero mean and variance γ^2. Using this prior, the MAP objective becomes equivalent to a regularized loss (2.9) where $\mathcal{R}(\mathbf{W})$ corresponds to an ℓ^2-norm regularizer. Another common choice is a Laplace prior, for which the MAP objective recovers an ℓ^1-norm regularizer. The ℓ^1-norm regularizer is known to enforce sparsity and is frequently used in the pruning literature (see Section 4.2). Blundell et al. [74] employ a scale mixture prior using two zero-mean Gaussians with different variances γ_1^2 and γ_2^2. This enforces a portion of the weights to concentrate around zero while also allowing some of the weights to have larger magnitudes.



(a) deterministic weights (b) weight distributions (c) probabilistic forward pass

Figure 3.4: (a) Conventional DNN with deterministic weights. (b) Bayesian DNN using the mean field approximation. Each connection is associated with a weight distribution instead of a deterministic weight. (c) The probabilistic forward pass operates in two steps. In the first step, the activation distribution is approximated by a Gaussian using the central limit theorem. In the second step, the Gaussian approximation is passed through the nonlinear activation function.

A more elaborate prior is based on general Gaussian mixture models (GMMs) [75] where the mass is spread around the means of several Gaussian components. By performing MAP inference using a GMM prior, the weights tend to cluster around the component means. This results in a *soft weight sharing* where many of the weights are similar but not equal. However, it is not straightforward to set the component means in a meaningful way. Therefore, the authors of [75] propose to adapt the prior $p(\mathbf{W})$ during training.

An interesting perspective on Bayesian DNNs emerges if we attribute structural constraints on the architecture to specific choices of priors $p(\mathbf{W})$. For instance, a convolutional layer in a CNN can be seen as a fully connected layer where an infinitely strong prior $p(\mathbf{W})$ enforces a particular sparsity and weight sharing pattern on $\mathbf{W}$. Other common examples of imposing structural restrictions on the weights are RNNs, autoencoders [22] (decoder weight matrices are transposed versions of the encoder weight matrices), and random weight sharing [76].

Bayesian inference for DNNs is extremely challenging. Considering the difficulty of finding a mode of the posterior $p(\mathbf{W} \mid \mathcal{D})$, it is unsurprising that exact computation of (3.52) is intractable for DNNs of any reasonable size. The posterior inherits all the unfavorable computational properties of DNNs, i.e., it is highly nonlinear and multimodal. Furthermore, the number of weights is typically on the order of thousands or even millions, preventing the use of general purpose numeric integration methods. Even approximating (3.52) by replacing the true posterior with a more convenient distribution $q(\mathbf{W})$ is intractable due to the nonlinearities of $p(\mathbf{y} \mid \mathbf{x}, \mathbf{W})$.

In the remainder of this section, we focus on approximating expected predictions (3.52). For this purpose, we assume that we are given a suitable approximation $q(\mathbf{W})$ of the posterior (3.51) that has already been computed in a previous step. We defer the discussion on how to actually obtain an approximation $q(\mathbf{W})$ to Section 3.4 since the presented methods to approximate (3.51) and (3.52) rely on the same concepts. Provided that we can efficiently generate independent samples from $q(\mathbf{W})$, the most straightforward approximation of (3.52) is to compute a Monte Carlo average as

$$\mathbb{E}_{\mathbf{W} \sim q(\mathbf{W})}[p(\mathbf{y} \mid \mathbf{x}, \mathbf{W})] \approx \frac{1}{M} \sum_{i=1}^{M} p(\mathbf{y} \mid \mathbf{x}, \mathbf{W}^i) \qquad \text{with} \qquad \mathbf{W}^i \sim q(\mathbf{W}). \tag{3.53}$$

In the remainder, we derive closed-form approximations of (3.52).

3.3.1 Linearization of the Network Output

In the following, we assume a Gaussian approximation $q(\mathbf{W}) = \mathcal{N}(\mathbf{W}|\boldsymbol{\mu}_W, \boldsymbol{\Sigma}_W)$. Possible ways to obtain a Gaussian approximation are Laplace's method (see Section 3.2.2) or variational inference (see Section 3.2.3 and Section 3.4).

Using the Gaussian approximation $q(\mathbf{W})$, it is possible to obtain a closed-form approximation based on a linearization of the DNN output activations $\mathbf{a}^L$. Assume that the variance of $q(\mathbf{W})$ is small such that the output activations $\mathbf{a}^L_{\mathbf{W}}(\mathbf{x})$ behave linearly in $\mathbf{W}$ in the region where $q(\mathbf{W})$ exhibits significant mass, i.e., in the vicinity of $\boldsymbol{\mu}_W$. This assumption justifies a first-order Taylor expansion of $\mathbf{a}^L_{\mathbf{W}}(\mathbf{x})$ around $\boldsymbol{\mu}_W$ to approximate the DNN output as

$$\mathbf{a}^L_{\mathbf{W}}(\mathbf{x}) \approx \mathbf{a}^L_{\boldsymbol{\mu}_W}(\mathbf{x}) + \mathbf{g}(\mathbf{x})^\top (\mathbf{W} - \boldsymbol{\mu}_W), \tag{3.54}$$

where we have defined the Jacobian

$$\mathbf{g}(\mathbf{x}) = \nabla_{\mathbf{W}} \mathbf{a}^L_{\mathbf{W}}(\mathbf{x})\Big|_{\mathbf{W}=\boldsymbol{\mu}_W}. \tag{3.55}$$

Using $\mathbf{W} \sim \mathcal{N}(\boldsymbol{\mu}_W, \boldsymbol{\Sigma}_W)$, we substitute $\mathbf{W} = \boldsymbol{\mu}_W + \boldsymbol{\Sigma}_W^{\frac{1}{2}} \mathbf{Z}$ for $\mathbf{Z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ into the right hand side of (3.54) to obtain

$$\mathbf{a}^L_{\mathbf{W}}(\mathbf{x}) \approx \mathbf{a}^L_{\boldsymbol{\mu}_W}(\mathbf{x}) + \mathbf{g}(\mathbf{x})^\top \boldsymbol{\Sigma}_W^{\frac{1}{2}} \mathbf{Z}, \tag{3.56}$$

which we recognize as following the Gaussian distribution

$$p(\mathbf{a}^L|\mathbf{x}, \mathcal{D}) \approx \mathcal{N}\left(\mathbf{a}^L \middle| \mathbf{a}^L_{\boldsymbol{\mu}_W}(\mathbf{x}), \mathbf{g}(\mathbf{x})^\top \boldsymbol{\Sigma}_W \mathbf{g}(\mathbf{x})\right). \tag{3.57}$$

For regression, we assume that the outputs $\mathbf{y}$ are normally distributed according to

$$p(\mathbf{y}|\mathbf{a}^L) = \mathcal{N}\left(\mathbf{y} \middle| \mathbf{a}^L, \mathbf{I}\beta^2\right), \tag{3.58}$$

which results in the approximate output distribution

$$p(\mathbf{y}|\mathbf{x}, \mathcal{D}) \approx \int p(\mathbf{a}^L|\mathbf{x}, \mathcal{D}) p(\mathbf{y}|\mathbf{a}^L) \, d\mathbf{a}^L = \mathcal{N}\left(\mathbf{y} \middle| \mathbf{a}^L_{\boldsymbol{\mu}_W}(\mathbf{x}), \mathbf{g}(\mathbf{x})^\top \boldsymbol{\Sigma}_W \mathbf{g}(\mathbf{x}) + \mathbf{I}\beta^2\right). \tag{3.59}$$

This approximation has, for instance, been used in [77]. For binary classification using a single output a^L, the outputs $y \in \{0, 1\}$ are distributed according to

$$p(Y = 1|a^L) = \mathrm{sigm}(a^L). \tag{3.60}$$

For the logistic sigmoid, the marginalization over a^L does not admit an analytic solution. However, by approximating the logistic sigmoid by the cdf of a standard normal distribution, $\Phi(a^L)$, we obtain the closed-form approximation (B.27) as

$$p(Y = 1|\mathbf{x}, \mathcal{D}) = \int \mathrm{sigm}(a^L) \mathcal{N}(a^L|\mu_{a^L}, \sigma^2_{a^L}) \, da^L \approx \mathrm{sigm}\left(\frac{\mu_{a^L}}{\sqrt{1 + \tilde{c}^2 \sigma^2_{a^L}}}\right), \tag{3.61}$$

for $\tilde{c} = \sqrt{\pi/8}$, and μ_{a^L} and $\sigma^2_{a^L}$ being the mean and the variance of the Gaussian approximation in (3.57), respectively. The multiclass case is more challenging. We refer to [78] and [79] for discussions on approximations for the multiclass case.

3.3.2 The Probabilistic Forward Pass

The previous approximation can be considered as a global approximation scheme in the sense that the linearization (3.54) takes into account how individual weights influence the DNN output activations $\mathbf{a}^L$. A local approximation scheme is obtained by considering how individual weights influence the activations of their layer. This is achieved by successively performing Gaussian approximations of the activation distributions $p(\mathbf{a}^l)$ of the individual layers (see Figure 3.4(c)). This method, which we call *probabilistic forward pass*, is an integral part of a paper we have published at the Workshop on Bayesian Deep Learning at the NIPS conference in 2016 [80].

Throughout our discussion, we assume a factorized distribution $q(\mathbf{W})$ and that the weights $\mathbf{W}^l$ are independent from the layer inputs $\mathbf{x}^{l-1}$. We emphasize that the probabilistic forward pass is not constrained to particular forms of $q(\mathbf{W})$. The only requirement on $q(\mathbf{W})$ is that its mean $\mathbb{E}[\mathbf{W}]$ and variance $\mathbb{V}[\mathbf{W}]$ can be evaluated efficiently. Indeed, we will utilize this freedom in Chapter 5 to approximate expected predictions for discrete distributions $q(\mathbf{W})$.

Let $\mathbf{W} = (\mathbf{W}^1, \dots, \mathbf{W}^L)$ be the set of weight matrices of all layers. We want to approximate the expected prediction (3.52) by replacing the true posterior $p(\mathbf{W} \,|\, \mathcal{D})$ by the approximation $q(\mathbf{W})$, i.e.,

$$\mathbb{E}_{\mathbf{W} \sim p(\mathbf{W}|\mathcal{D})}[p(\mathbf{y} \,|\, \mathbf{x}^0, \mathbf{W})] \approx \int q(\mathbf{W}) \, p(\mathbf{y} \,|\, \mathbf{x}^0, \mathbf{W}) \, d\mathbf{W}. \tag{3.62}$$

Approximate $\mathbf{a}^l$: The first step of the probabilistic forward pass approximates the distribution over activations $\mathbf{a}^l$. Assume we are given a distribution $q(\mathbf{x}^{l-1})$ over the inputs from the previous layer. Since the activations are computed as a sum of typically many random variables, we can invoke the central limit theorem and approximate the activation distribution as

$$q(\mathbf{a}^l \,|\, \mathbf{x}^{l-1}, \mathbf{W}^l) = \mathcal{N}(\mathbf{a}^L \,|\, \boldsymbol{\mu}_{a^l}, \boldsymbol{\Sigma}_{a^l}), \tag{3.63}$$

where

$$\mu_{a_i^l} = \sum_k \mathbb{E}[w_{i,k}^l] \, \mathbb{E}[x_k^{l-1}] \tag{3.64}$$

and

$$\operatorname{cov}(a_i^l, a_j^l) = \sum_k \sum_{k'} \mathbb{E}[w_{i,k}^l] \, \mathbb{E}[w_{j,k'}^l] \operatorname{cov}(x_k^{l-1}, x_{k'}^{l-1}) + \mathbb{I}[i = j] \sum_k \mathbb{V}[w_{i,k}^l] \, \mathbb{E}[(x_k^{l-1})^2]. \tag{3.65}$$

We refer to Appendix B.1 for a detailed derivation of (3.65). To reduce the computational overhead, we only compute $\operatorname{cov}(a_i^l, a_j^l)$ for $i = j$, i.e., we approximate $\boldsymbol{\Sigma}_{a^l}$ by its diagonal as $\boldsymbol{\Sigma}_{a^l} \approx \operatorname{diag}(\boldsymbol{\sigma}_{a^l}^2)$ where $\sigma_{a_i^l}^2 = \mathbb{V}[a_i^l]$. For element-wise activation functions $h^l(\mathbf{a}^l)$, this implies $\operatorname{cov}(x_k^l, x_{k'}^l) = 0$ for $k \neq k'$ for the inputs of the subsequent layer. By assuming deterministic inputs $\mathbf{x}^0$, this further implies $\operatorname{cov}(x_k^l, x_{k'}^l) = 0$ for $k \neq k'$ for *all* layers l. Therefore, the diagonal entries $\operatorname{cov}(a_i^l, a_i^l)$ of $\boldsymbol{\Sigma}_{a^l}$, simplify to

$$\sigma_{a_i^l}^2 = \sum_k \mathbb{E}[w_{i,k}^l]^2 \, \mathbb{V}[x_k^{l-1}] + \mathbb{V}[w_{i,k}^l] \, \mathbb{E}[(x_k^{l-1})^2] \tag{3.66}$$

$$= \sum_k \mathbb{E}[w_{i,k}^l]^2 \, \mathbb{V}[x_k^{l-1}] + \mathbb{V}[w_{i,k}^l] \, \mathbb{E}[x_k^{l-1}]^2 + \mathbb{V}[w_{i,k}^l] \, \mathbb{V}[x_k^{l-1}]. \tag{3.67}$$

Note that for the first layer $l = 1$, the values $\mathbf{x}^0$ are deterministic, simplifying expressions to

$$\mu_{a_i^1} = \sum_k \mathbb{E}[w_{i,k}^1] \, x_k^0 \qquad \text{and} \qquad \sigma_{a_i^1}^2 = \sum_k \mathbb{V}[w_{i,k}^1] \, (x_k^0)^2. \tag{3.68}$$

Indeed, the diagonal approximation is exact for the first layer, which can be easily seen by noting that $\mathrm{cov}(x_k^{l-1}, x_{k'}^{l-1}) = 0$ in (3.65). It is intuitive to think of the Gaussian approximation as pushing the uncertainty over the weights $\mathbf{W}^l$ to the subsequent activations $\mathbf{a}^l$. This allows us to approximate the integral (3.62) as

$$\mathbb{E}_{\mathbf{W}\sim p(\mathbf{W}|\mathcal{D})}[p(\mathbf{y}|\mathbf{x}^0, \mathbf{W})] \approx \int q(\mathbf{W}^{>l})\mathcal{N}\Big(\mathbf{a}^l \,|\, \boldsymbol{\mu}_{a^l}, \mathrm{diag}(\boldsymbol{\sigma}_{a^l}^2)\Big)\, p(\mathbf{y}|\mathbf{a}^l, \mathbf{W}^{>l})\, d\mathbf{a}^l d\mathbf{W}^{>l}, \tag{3.69}$$

where we have defined $\mathbf{W}^{>l} = (\mathbf{W}^{l+1}, \ldots, \mathbf{W}^L)$.

Approximate $\mathbf{x}^l$: In the next step, the Gaussian distribution of the activations $\mathbf{a}^l$ is passed through the nonlinear activation function h^l to obtain a distribution over the inputs $\mathbf{x}^l$ of the next layer. Depending on the activation function h^l, we compute the means and the variances

$$\mathbb{E}[x_i^l] = \mathbb{E}_{a_i^l\sim\mathcal{N}(\mu_{a_i^l}, \sigma^2_{a_i^l})}[h^l(a_i^l)] \qquad \text{and} \qquad \mathbb{V}[x_i^l] = \mathbb{V}_{a_i^l\sim\mathcal{N}(\mu_{a_i^l}, \sigma^2_{a_i^l})}[h^l(a_i^l)] \tag{3.70}$$

either in closed form or we approximate them if the activation function h^l does not permit a closed-form solution. This further approximates (3.69) as

$$\mathbb{E}_{\mathbf{W}\sim p(\mathbf{W}|\mathcal{D})}[p(\mathbf{y}|\mathbf{x}^0, \mathbf{W})] \approx \int q(\mathbf{W}^{>l})\, q(\mathbf{x}^l)\, p(\mathbf{y}|\mathbf{x}^l, \mathbf{W}^{>l})\, d\mathbf{x}^l d\mathbf{W}^{>l}. \tag{3.71}$$

Next, we show how to compute the mean $\mathbb{E}[x_i^l]$ and the raw second moment $\mathbb{E}[(x_i^l)^2]$ for commonly used activation functions h^l. The variance follows from $\mathbb{V}[x_i^l] = \mathbb{E}[(x_i^l)^2] - \mathbb{E}[x_i^l]^2$.

ReLU: For the ReLU function $\max(a, 0)$, we obtain the expectation

$$\mathbb{E}\Big[x_i^l\Big] = \frac{\mu_{a_i^l}}{2}\left(1 + \mathrm{erf}\left(\frac{\mu_{a_i^l}}{\sqrt{2\sigma^2_{a_i^l}}}\right)\right) + \sqrt{\frac{\sigma^2_{a_i^l}}{2\pi}}\exp\left(-\frac{\mu^2_{a_i^l}}{2\sigma^2_{a_i^l}}\right) \tag{3.72}$$

and the raw second moment

$$\mathbb{E}\Big[(x_i^l)^2\Big] = \frac{\sigma^2_{a_i^l} + \mu^2_{a_i^l}}{2}\left(1 + \mathrm{erf}\left(\frac{\mu_{a_i^l}}{\sqrt{2\sigma^2_{a_i^l}}}\right)\right) + \mu_{a_i^l}\sqrt{\frac{\sigma^2_{a_i^l}}{2\pi}}\exp\left(-\frac{\mu^2_{a_i^l}}{2\sigma^2_{a_i^l}}\right), \tag{3.73}$$

where erf denotes the error function defined as

$$\mathrm{erf}(u) = \frac{2}{\sqrt{\pi}}\int_0^u \exp(-z^2)\, dz. \tag{3.74}$$

The integral in the definition of the error function (3.74) does not allow for an analytic solution, but efficient implementations thereof exist in any reasonable package for numeric computations.

sigm and tanh: For the logistic sigmoid and the hyperbolic tangent, the required first and second moments cannot be computed in closed form. However, by approximating the logistic sigmoid by the cdf of a standard normal distribution, $\Phi(a)$, we can approximate the required first and second moments (see Appendices B.5 and B.6) by

$$\mathbb{E}\Big[x_i^l\Big] = \mathrm{sigm}\left(\frac{\mu_{a_i^l}}{\sqrt{1 + \tilde{c}^2\sigma^2_{a_i^l}}}\right) \qquad \text{and} \qquad \mathbb{E}\Big[(x_i^l)^2\Big] = \mathrm{sigm}\left(\frac{\tilde{\alpha}(\mu_{a_i^l} - \tilde{b})}{\sqrt{1 + \tilde{\alpha}^2\tilde{c}^2\sigma^2_{a_i^l}}}\right) \tag{3.75}$$

for $\tilde{\alpha} = 4 - 2\sqrt{2}$, $\tilde{b} = -\log(\sqrt{2} - 1)$, and $\tilde{c} = \sqrt{\pi/8}$. By noting that $\tanh(a) = 2\,\mathrm{sigm}(2a) - 1$, we obtain a similar approximation for the hyperbolic tangent (see Appendix B.7).

sign: An interesting case is obtained for the sign activation function, for which we obtain a binary output distribution over $\{-1, 1\}$. Using a standard normal random variable $Z \sim \mathcal{N}(0, 1)$, the corresponding output distribution is given by

$$q(X_i^l = 1) = q(A_i^l > 0) = p(\sigma_{a_i^l} Z + \mu_{a_i^l} > 0) = p(Z > -\mu_{a_i^l}/\sigma_{a_i^l}) = \Phi(\mu_{a_i^l}/\sigma_{a_i^l}). \tag{3.76}$$

The corresponding mean and respective raw second moment are given by

$$\mathbb{E}[x_i^l] = 2\Phi\Big(\mu_{a_i^l}/\sigma_{a_i^l}\Big) - 1 \qquad \text{and} \qquad \mathbb{E}[(x_i^l)^2] = 1. \tag{3.77}$$

Most importantly, the mean $\mathbb{E}[x_i^l]$ and the variance $\mathbb{V}[x_i^l]$ of this binary distribution are differentiable in the mean $\mu_{a_i^l}$ and the variance $\sigma^2_{a_i^l}$ of the Gaussian activation distribution. We utilize this in Chapter 5 where we train DNNs using the sign activation function.

The two steps discussed so far—approximating the distribution over activations by a Gaussian and propagating this Gaussian through the nonlinear activation function—are iterated for all layers up to the output layer to obtain a Gaussian approximation over the activations $\mathbf{a}^L$, i.e.,

$$\mathbb{E}_{\mathbf{W}\sim p(\mathbf{W}|\mathcal{D})}[p(\mathbf{y}\,|\,\mathbf{x}^0, \mathbf{W})] \approx \int \mathcal{N}\Big(\mathbf{a}^L\,|\,\boldsymbol{\mu}_{a^L}, \mathrm{diag}(\boldsymbol{\sigma}^2_{a^L})\Big)\, p(\mathbf{y}\,|\,\mathbf{a}^L)\, d\mathbf{a}^L. \tag{3.78}$$

The form of the integral in (3.78) is similar to the integral of the linearization approximation (3.59). For regression, we obtain the approximate output distribution

$$p(\mathbf{y}\,|\,\mathbf{x}, \mathcal{D}) = \mathbb{E}_{\mathbf{W}\sim p(\mathbf{W}|\mathcal{D})}[p(\mathbf{y}\,|\,\mathbf{x}^0, \mathbf{W})] \approx \mathcal{N}\Big(\mathbf{y}\,\Big|\,\boldsymbol{\mu}_{a^L}, \mathrm{diag}(\boldsymbol{\sigma}^2_{a^L}) + \mathbf{I}\beta^2\Big)\,. \tag{3.79}$$

For binary classification, we can apply the approximation in (3.61). For a discussion of the multiclass case, we again refer to [78] and [79].

The idea of approximating a sum over several random variables by a Gaussian using the central limit theorem has been used in different contexts before. Teh et al. [81] treat certain count variables as sums over several independent Bernoulli variables and approximate them by Gaussians. This approximation dramatically improves the computational efficiency of their collapsed variational inference algorithm for the latent Dirichlet allocation model. More closely related to our current discussion, Ribeiro and Opper [82] approximate sums over several random variables as they arise in single layer neural network models (e.g., linear or logistic regression) to obtain a tractable expectation propagation [54] algorithm. Several works have extended this technique to DNNs by performing successive Gaussian approximations of the activation distribution $p(\mathbf{a}^l)$. Wang and Manning [31] transfer the stochasticity of dropout noise to subsequent layers to obtain a closed-form approximation of the induced dropout objective. They showed that this reduces the training time dramatically due to the reduced gradient variance while still enjoying the regularizing effect of dropout. Soudry et al. [83] apply the Gaussian approximation to obtain closed-form expectation propagation style updates for discrete and continuous weight distributions of DNNs with sign activations. However, for continuous weight distributions, their method assumes a Gaussian distribution with fixed variance. Hernández-Lobato and Adams [84] propose a similar method inspired by expectation propagation to allow for continuous Gaussian weight distributions with learnable variance parameters. Moreover, they also approximate the posterior over the hyperparameters, i.e., the prior variances of both the weights and the DNN outputs for regression.

The probabilistic forward pass has been extended in [85] to the non-diagonal case by also modeling correlations between activations $\mathbf{a}^l$. They propose approximations to the required

quantities when correlated Gaussians are propagated through the commonly used ReLU and Heaviside step activation functions. In Section 3.4.4, we will see that, conceptually, the probabilistic forward pass is closely related to the *local reparameterization trick* [33] which propagates samples from the activation distributions $q(\mathbf{a}^l)$ through the DNN.

We note that using the probabilistic forward pass for convolutional layers, recurrent architectures, or any other architecture that employs some kind of weight sharing, introduces an independence assumption. Convolutions use the same weights to compute activations at different spatial locations, and recurrent architectures use the same weights to compute activations at different time steps. So far, we have implicitly assumed that each individual weight $w_{i,j}^l$ only participates in the computation of a single activation a_i^l. For models with shared weights, a single weight participates in the computation of several activations a_i^l and the corresponding activation distributions $q(\mathbf{a}^l)$ become dependent if the shared weights are random variables.

Note that we have ignored commonly used building blocks in our discussion of the probabilistic forward pass. In Chapter 5, we present methods for propagating Gaussian distributions through batch normalization and max pooling operations.

3.4 Bayesian Neural Networks Using Variational Inference

In the previous section, we have assumed that we are already given an approximation $q(\mathbf{W})$ to the intractable posterior $p(\mathbf{W}|\mathcal{D})$. In this section, we discuss how such an approximation can actually be obtained. More specifically, we aim to minimize the reverse KL divergence objective (3.25) where the parameters $\boldsymbol{\theta}$ correspond to the weights $\mathbf{W}$ of a DNN. For convenience, we state the reverse KL divergence objective again here, i.e.,

$$D_{\mathrm{KL}}(q_{\boldsymbol{\nu}}(\mathbf{W})||p(\mathbf{W}|\mathcal{D})) = \mathbb{E}_{\mathbf{W}\sim q_{\boldsymbol{\nu}}(\mathbf{W})}\left[\log \frac{q_{\boldsymbol{\nu}}(\mathbf{W})}{p(\mathbf{W}|\mathcal{D})}\right]. \tag{3.80}$$

One of the pioneering works concerned with the minimization of (3.80) was conducted by Hinton and van Camp [86], albeit drawing motivation from a different background. They phrased the minimization of (3.80) in terms of the information theoretic minimum description length (MDL) framework [87] which states that one should minimize the number of bits required to communicate both the model parameters and the discrepancy between model predictions and target values. For this purpose, we can apply the decomposition (3.28) to obtain

$$D_{\mathrm{KL}}(q_{\boldsymbol{\nu}}(\mathbf{W})||p(\mathbf{W}|\mathcal{D})) = -\mathbb{E}_{\mathbf{W}\sim q_{\boldsymbol{\nu}}(\mathbf{W})}\left[\log p(\mathcal{D}|\mathbf{W})\right] + D_{\mathrm{KL}}(q_{\boldsymbol{\nu}}(\mathbf{W})||p(\mathbf{W})) + \log p(\mathcal{D}). \tag{3.81}$$

The MDL loss is defined as the sum of the first two terms of (3.81). The third term can be ignored as it does not depend on the variational parameters $\boldsymbol{\nu}$. In the MDL framework, the first term is interpreted as the amount of information required to encode the targets $\mathbf{y}$ for known inputs $\mathbf{x}^0$ using the outputs $\hat{\mathbf{y}}$ of a DNN whose weights $\mathbf{W}$ are sampled from $q_{\boldsymbol{\nu}}(\mathbf{W})$. The second term is interpreted as the amount of information required to encode weights $\mathbf{W}$ sampled from $q_{\boldsymbol{\nu}}(\mathbf{W})$ using a coding scheme defined by the prior $p(\mathbf{W})$. In this context, the first term can be seen as a reconstruction loss, whereas the second term can be seen as a complexity loss [88]. In [86] it is shown that the MDL loss and its gradient can be computed exactly for a single hidden layer neural network with nonlinear hidden activation functions and linear output activations. Their implementation utilized precomputed numerical integrals stored in a lookup table.

However, for DNNs with many layers, optimizing (3.80) is highly non-trivial and requires approximation techniques. It is worth inspecting (3.81) once again as it separates the intractable parts from the tractable parts of (3.80) more clearly. The third term of (3.81), the marginal likelihood or evidence term $\log p(\mathcal{D})$, is intractable but irrelevant for optimization. The second

term, the KL divergence between the approximation $q_{\boldsymbol{\nu}}(\mathbf{W})$ and the prior $p(\mathbf{W})$, is tractable for many common choices of the respective distributions, e.g., if both distributions are from the same exponential family. We typically also assume factorized distributions $q_{\boldsymbol{\nu}}(\mathbf{W})$ and $p(\mathbf{W})$ such that the KL divergence decomposes into a sum over the individual weights. For instance, the KL divergence between two one-dimensional Gaussians is given by

$$D_{\mathrm{KL}}(\mathcal{N}(\mu_1, \sigma_1^2) \,||\, \mathcal{N}(\mu_2, \sigma_2^2)) = \frac{1}{2}\left(\log \frac{\sigma_2^2}{\sigma_1^2} + \frac{\sigma_1^2 + (\mu_1 - \mu_2)^2}{\sigma_2^2} - 1 \right). \tag{3.82}$$

The relevant intractable part of (3.81) manifests itself in the first term, the expected negative log-likelihood. By recognizing that this term is closely related to the expected predictions (3.52) discussed in Section 3.3, we can readily take advantage of the techniques discussed there.

In the remainder of this section, we discuss two approaches to optimizing (3.80) with a special emphasis on the intractable expected log-likelihood. The first approach approximates the expected log-likelihood using the probabilistic forward pass to obtain a closed-form expression. The second approach computes Monte Carlo gradients in order to perform SGD. Unlike computing Monte Carlo estimates of the loss function itself, computing Monte Carlo estimates of its gradient requires more care to maintain compatibility with backpropagation.

3.4.1 A Closed-Form Approximation Using the Probabilistic Forward Pass

The expected predictions (3.52) differ from the expected log-likelihood in (3.81) only in the additional logarithm. Therefore, we can perform the probabilistic forward pass (see Section 3.3.2) to obtain a closed-form Gaussian approximation of the output activations $\mathbf{a}^L \sim \mathcal{N}(\boldsymbol{\mu}_{a^L}, \mathrm{diag}(\boldsymbol{\sigma}^2_{a^L}))$. The resulting approximation of the expected log-likelihood is given by

$$\mathbb{E}_{\mathbf{W}\sim q(\mathbf{W})}[\log p(\mathbf{y}\,|\,\mathbf{x}^0, \mathbf{W})] \approx \mathbb{E}_{\mathbf{a}^L \sim \mathcal{N}(\boldsymbol{\mu}_{a^L}, \mathrm{diag}(\boldsymbol{\sigma}^2_{a^L}))}[\log p(\mathbf{y}\,|\,\mathbf{a}^L)]. \tag{3.83}$$

For regression, the right hand side of (3.83) is an expectation of a quadratic form with respect to a Gaussian which can be computed exactly. Assuming $p(\mathbf{y}\,|\,\mathbf{a}^L) = \mathcal{N}(\mathbf{y}\,|\,\mathbf{a}^L, \mathbf{I}\beta^2)$, we obtain

$$\mathbb{E}_{\mathbf{a}^L \sim \mathcal{N}(\boldsymbol{\mu}_{a^L}, \mathrm{diag}(\boldsymbol{\sigma}^2_{a^L}))}[\log p(\mathbf{y}\,|\,\mathbf{a}^L)] = -\frac{d_L}{2}\log 2\pi\beta^2 - \frac{1}{2\beta^2}\sum_{i=1}^{d_L}\left[\sigma^2_{a_i^L} + (\mu_{a_i^L} - y_i)^2\right]. \tag{3.84}$$

We refer to Appendix B.2 for the derivation of an exact expression for general covariance matrices of the Gaussian distributions $p(\mathbf{y}\,|\,\mathbf{a}^L)$ and $\mathcal{N}(\mathbf{a}^L\,|\,\boldsymbol{\mu}_{a^L}, \boldsymbol{\Sigma}_{a^L})$.

For classification, the expectation on the right hand side of (3.83) does not admit an exact solution and, once again, we have to resort to approximations. We propose to approximate $\log p(\mathbf{y}\,|\,\mathbf{a}^L)$ using a second-order Taylor approximation around $\boldsymbol{\mu}_{a^L}$ as

$$\log p(\mathbf{y}\,|\,\mathbf{a}^L) \approx \log p(\mathbf{y}\,|\,\boldsymbol{\mu}_{a^L}) + \left(\mathbf{a}^L - \boldsymbol{\mu}_{a^L}\right)^\top \mathbf{g} + \frac{1}{2}\left(\mathbf{a}^L - \boldsymbol{\mu}_{a^L}\right)^\top \mathbf{H}\left(\mathbf{a}^L - \boldsymbol{\mu}_{a^L}\right) \tag{3.85}$$

with the respective gradient and Hessian matrix

$$\mathbf{g} = \nabla_{\mathbf{a}^L} \log p(\mathbf{y}\,|\,\mathbf{a}^L)\Big|_{\mathbf{a}^L = \boldsymbol{\mu}_{a^L}} \quad \text{and} \quad \mathbf{H} = \nabla^2_{\mathbf{a}^L} \log p(\mathbf{y}\,|\,\mathbf{a}^L)\Big|_{\mathbf{a}^L = \boldsymbol{\mu}_{a^L}}. \tag{3.86}$$

Using (B.18), the exact expectation of the second-order approximation (3.85) with respect to a Gaussian is given by

$$\mathbb{E}_{\mathbf{a}^L \sim \mathcal{N}(\boldsymbol{\mu}_{a^L}, \mathrm{diag}(\boldsymbol{\sigma}^2_{a^L}))}[\log p(\mathbf{y}\,|\,\mathbf{a}^L)] \approx \log p(\mathbf{y}\,|\,\boldsymbol{\mu}_{a^L}) + \frac{1}{2}\sum_{i=1}^{d_L}\sigma^2_{a_i^L} H_{i,i}. \tag{3.87}$$

Note that the first-order term involving the gradient $\mathbf{g}$ has vanished since the Taylor approximation was computed around the mean $\boldsymbol{\mu}_{a^L}$. Furthermore, since we assume a diagonal Gaussian $\mathcal{N}(\mathbf{a}^L \,|\, \boldsymbol{\mu}_{a^L}, \operatorname{diag}(\boldsymbol{\sigma}^2_{a^L}))$, we only require the diagonal entries of the Hessian $\mathbf{H}$.

It remains to compute the diagonal entries of the Hessian $\mathbf{H}$. For completeness, we also report the expressions for the gradient $\mathbf{g}$. For binary classification using a single output a^L, we have

$$\log p(y \,|\, a^L) = y \log(\operatorname{sigm}(a^L)) + (1-y)\log(1 - \operatorname{sigm}(a^L)). \tag{3.88}$$

The first and second partial derivatives of (3.88) evaluated at μ_{a^L} are given by

$$g = y - \operatorname{sigm}(\mu_{a^L}) \quad \text{and} \quad H = -\operatorname{sigm}(\mu_{a^L})(1 - \operatorname{sigm}(\mu_{a^L})). \tag{3.89}$$

For multiclass classification with targets $y \in \{1, \ldots, \mathcal{C}\}$, we have

$$\log p(y \,|\, \mathbf{a}^L) = \log(\operatorname{softmax}_y(\mathbf{a}^L)). \tag{3.90}$$

The first and second partial derivatives of (3.90) evaluated at $\boldsymbol{\mu}_{a^L}$ are given by

$$g_i = \mathbb{I}[y = i] - \operatorname{softmax}_i(\boldsymbol{\mu}_{a^L}) \quad \text{and} \quad H_{i,i} = -\operatorname{softmax}_i(\boldsymbol{\mu}_{a^L})(1 - \operatorname{softmax}_i(\boldsymbol{\mu}_{a^L})). \tag{3.91}$$

In practice, the second-order approximation often provides a good approximation to the true expectation. However, if the variance $\boldsymbol{\Sigma}_{a^L}$ is large such that the Gaussian puts significant mass in regions where the second-order approximation becomes inaccurate, the approximation quality might suffer. For alternative approximations we refer to [31] for binary classification and to [79] for multiclass classification.

From an optimization perspective, the approximated closed-form objective yields several benefits compared to stochastic methods relying on Monte Carlo gradients. Since the gradient of the loss can be computed in closed form, the optimization procedure does not suffer from a potentially high gradient variance impairing the convergence behavior. It is also easier to select an appropriate stopping criterion for iterative optimization procedures. Commonly used stopping criteria are based on the loss function or the gradient magnitude, both of which can be evaluated exactly. Furthermore, the closed-form objective can be optimized using more sophisticated second-order optimization techniques such as quasi-Newton methods. Second-order methods are often prone to numerical errors in the stochastic setting and, therefore, are rarely used to train DNNs. We conclude that especially in the small-data regime where it is feasible to compute gradients in batch mode using the whole dataset $\mathcal{D}$, the closed-form loss might provide a viable option. We note that the closed-form loss is still compatible with mini-batch optimization to inherit the advantages of stochastic optimization.

On the downside, we might also lose some favorable properties when optimizing an approximated objective. For instance, it is not guaranteed that minimizing the approximation can be seen as maximizing a lower bound to the evidence as in (3.29).

3.4.2 Optimization Using Monte Carlo Gradients

Although the expected log-likelihood term in (3.81) cannot be computed exactly, it can be approximated by Monte Carlo integration using samples from the variational approximation $q_{\boldsymbol{\nu}}(\mathbf{W})$. This raises the question whether it is possible to generate unbiased Monte Carlo samples of the gradient of the expected log-likelihood in order to perform SGD. The answer turns out to be positive, but it is not as straightforward as generating Monte Carlo samples of the expected log-likelihood itself.

The difficulty is best explained in terms of computation graphs (see Section 2.1.3). We consider

a general loss $\mathcal{L}(\mathbf{W})$ and a corresponding expected loss

$$\mathcal{L}_{\mathbb{E}}(\boldsymbol{\nu}) = \mathbb{E}_{\mathbf{W}\sim q_{\boldsymbol{\nu}}(\mathbf{W})}[\mathcal{L}(\mathbf{W})]. \tag{3.92}$$

Note that the expected log-likelihood term in (3.81) is a special case of (3.92), but we emphasize that our discussion is rather general and applies more widely. We can construct a computation graph (see Figure 3.5(a)) that generates a Monte Carlo sample of the loss $\mathcal{L}_{\mathbb{E}}(\boldsymbol{\nu})$ by introducing a stochastic operation node that depends on $\boldsymbol{\nu}$ and generates a sample $\mathbf{W} \sim q_{\boldsymbol{\nu}}(\mathbf{W})$. Subsequently, the given loss $\mathcal{L}(\mathbf{W})$ is evaluated for the sampled weights $\mathbf{W}$. However, when we invoke backpropagation and arrive at the stochastic operation node, we cannot backpropagate through this stochastic node to obtain a gradient of $\boldsymbol{\nu}$ since the sampling procedure is not a differentiable operation—in fact, it is not even a function.

Fortunately, there exist several reformulations of the gradient $\nabla_{\boldsymbol{\nu}}\mathcal{L}_{\mathbb{E}}$ that allow us to generate Monte Carlo gradients. For instance, in a first attempt to scale the work of Hinton and van Camp [86] to larger architectures, Graves [89] utilized a special property of Gaussian distributions [90] that allows us to obtain Monte Carlo gradients by sampling from $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, i.e.,

$$\nabla_{\boldsymbol{\mu}}\mathbb{E}_{\mathbf{W}\sim\mathcal{N}(\boldsymbol{\mu},\boldsymbol{\Sigma})}[\mathcal{L}(\mathbf{W})] = \mathbb{E}_{\mathbf{W}\sim\mathcal{N}(\boldsymbol{\mu},\boldsymbol{\Sigma})}[\nabla_{\mathbf{W}}\mathcal{L}(\mathbf{W})], \qquad \text{and} \tag{3.93}$$

$$\nabla_{\boldsymbol{\Sigma}}\mathbb{E}_{\mathbf{W}\sim\mathcal{N}(\boldsymbol{\mu},\boldsymbol{\Sigma})}[\mathcal{L}(\mathbf{W})] = \frac{1}{2}\mathbb{E}_{\mathbf{W}\sim\mathcal{N}(\boldsymbol{\mu},\boldsymbol{\Sigma})}[\nabla^2_{\mathbf{W}}\mathcal{L}(\mathbf{W})]. \tag{3.94}$$

However, computing the gradient with respect to the covariance matrix (3.94) is computationally expensive, even for diagonal covariance matrices $\boldsymbol{\Sigma}$, since it requires the computation of second-order derivatives. Therefore, for large DNNs for which computing the diagonal Hessian is impractical, an approximation based on the empirical Fisher matrix is proposed [89], resulting in biased gradient estimates.

In the following sections, we discuss the two most commonly used approaches to obtain unbiased Monte Carlo gradients in more general settings. The first one, called the log-derivative trick, is applicable for a wide range of distributions but it typically suffers from high variance. The second one, called the reparameterization trick, is only applicable to continuous differentiable distributions but its variance is typically much lower. Finally, we present the Gumbel-softmax approximation which enables us to compute biased gradients for discrete distributions using the reparameterization trick. We refer the interested reader to [91] for a comprehensive survey about Monte Carlo gradient estimation.

3.4.3 The Log-Derivative Trick

The log-derivative trick, also known as score function estimator [92], the likelihood ratio method [93], or the REINFORCE estimator [94], is based on the general property

$$\nabla_{\boldsymbol{\nu}} f(\boldsymbol{\nu}) = f(\boldsymbol{\nu})\nabla_{\boldsymbol{\nu}} \log f(\boldsymbol{\nu}). \tag{3.95}$$

Using (3.95), we can rewrite the gradient of the expected loss $\mathcal{L}_{\mathbb{E}}(\boldsymbol{\nu})$ as

$$\nabla_{\boldsymbol{\nu}}\mathbb{E}_{\mathbf{W}\sim q_{\boldsymbol{\nu}}(\mathbf{W})}[\mathcal{L}(\mathbf{W})] = \int \nabla_{\boldsymbol{\nu}} q_{\boldsymbol{\nu}}(\mathbf{W})\mathcal{L}(\mathbf{W})d\mathbf{W} \tag{3.96}$$

$$= \mathbb{E}_{\mathbf{W}\sim q_{\boldsymbol{\nu}}(\mathbf{W})}[\mathcal{L}(\mathbf{W})\nabla_{\boldsymbol{\nu}} \log q_{\boldsymbol{\nu}}(\mathbf{W})]. \tag{3.97}$$

In (3.96) we have assumed that differentiation and integration are interchangeable. This is valid under relatively mild conditions, e.g., by appealing to Leibniz's integral rule. Expression (3.97) follows from (3.95). Assuming that the terms inside the expectation (3.97) can be efficiently

evaluated, we can approximate the gradients by Monte Carlo averaging as

$$\nabla_{\boldsymbol{\nu}} \mathcal{L}_{\mathbb{E}}(\boldsymbol{\nu}) \approx \frac{1}{M} \sum_{i=1}^{M} \mathcal{L}(\mathbf{W}^i) \nabla_{\boldsymbol{\nu}} \log q_{\boldsymbol{\nu}}(\mathbf{W}^i) \qquad \text{with} \qquad \mathbf{W}^i \sim q_{\boldsymbol{\nu}}(\mathbf{W}). \tag{3.98}$$

The log-derivative trick provides a general framework to compute stochastic gradients of functions that are defined as expectations. Moreover, the method is widely applicable and can, for instance, also be used for discrete distributions $q_{\boldsymbol{\nu}}(\mathbf{W})$.

However, it turns out that the log-derivative trick typically exhibits high variance. The reason for this is that the direction of the gradient is guided by the uninformative log-density $\log q_{\boldsymbol{\nu}}(\mathbf{W})$ rather than the loss $\mathcal{L}(\mathbf{W})$. A better intuition for this behavior is obtained by considering an isotropic Gaussian distribution $q_{\boldsymbol{\nu}}(\mathbf{W})$. By generating samples $\mathbf{W} \sim q_{\boldsymbol{\nu}}(\mathbf{W})$ to compute the gradient $\nabla \log q_{\boldsymbol{\nu}}(\mathbf{W})$, the direction of the gradient will effectively be uniformly distributed. Consequently, on average we cannot even expect this direction to be a descent direction of the loss function $\mathcal{L}_{\mathbb{E}}(\boldsymbol{\nu})$. It is only due to the weighting with the loss function $\mathcal{L}(\mathbf{W})$ in (3.97) that causes the stochastic gradients to be equal to the true gradient in expectation.

Besides using more samples to reduce the variance of the gradient estimator, there exist a variety of variance reduction techniques such as control variates [58, 95] and Rao-Blackwellization [58, 96]. Since the log-derivative trick is rarely used without such variance reduction techniques, we briefly review these two techniques here. We note that there exist several other variance reduction techniques and we refer the reader to [91, 97] for more about the topic.

Control Variates

The control variate method reduces the variance of a Monte Carlo estimator by exploiting knowledge from a related expectation that can be computed in closed form. More specifically, assume that the expectation $\mathbb{E}_{\mathbf{u} \sim p(\mathbf{u})}[f(\mathbf{u})]$ does not admit a closed-form solution but we can approximate it by sampling $\mathbf{u} \sim p(\mathbf{u})$. Furthermore, let $g(\mathbf{u})$ be some function such that $\mathbb{E}_{\mathbf{u} \sim p(\mathbf{u})}[g(\mathbf{u})] = c$ is known. We can then rewrite the expectation as

$$\mathbb{E}_{\mathbf{u} \sim p(\mathbf{u})}[f(\mathbf{u})] = \mathbb{E}_{\mathbf{u} \sim p(\mathbf{u})}[f(\mathbf{u}) - \alpha(g(\mathbf{u}) - c)] \tag{3.99}$$

to obtain a different estimator for the expectation of $f(\mathbf{u})$ for any $\alpha \in \mathbb{R}$. The variance of the right hand side of (3.99) is

$$\mathbb{V}[f(\mathbf{u}) - \alpha(g(\mathbf{u}) - c)] = \mathbb{V}[f(\mathbf{u})] + \alpha^2 \mathbb{V}[g(\mathbf{u})] - 2\alpha \operatorname{cov}(f(\mathbf{u}), g(\mathbf{u})). \tag{3.100}$$

Consequently, if $\operatorname{cov}(f(\mathbf{u}), g(\mathbf{u}))$ is large and for a suitable choice of α, the variance of the new estimator can be reduced. In fact, the optimal α^*, obtained by setting the derivative of (3.100) with respect to α to zero, is given by

$$\alpha^* = \frac{\operatorname{cov}(f(\mathbf{u}), g(\mathbf{u}))}{\mathbb{V}[g(\mathbf{u})]}. \tag{3.101}$$

The optimal α^* is typically not available in closed form, but it can be approximated using the empirical covariance and variance using the generated samples of $f(\mathbf{u})$ and $g(\mathbf{u})$.

Rao-Blackwellization

Rao-Blackwellization is a variance reduction technique that reduces the sampling space by conditioning on certain dimensions. Assume that we want to estimate

$$\mathbb{E}_{(\mathbf{u}, \tilde{\mathbf{u}}) \sim p(\mathbf{u}, \tilde{\mathbf{u}})}[f(\mathbf{u}, \tilde{\mathbf{u}})] = \mathbb{E}_{\mathbf{u} \sim p(\mathbf{u})}[\underbrace{\mathbb{E}_{\tilde{\mathbf{u}} \sim p(\tilde{\mathbf{u}}|\mathbf{u})}[f(\mathbf{u}, \tilde{\mathbf{u}})]}_{\tilde{f}(\mathbf{u})}]. \tag{3.102}$$

If $\tilde{f}(\mathbf{u})$ can be computed efficiently for any $\mathbf{u}$ (either numerically or in closed form), we obtain an estimator for $f(\mathbf{u}, \tilde{\mathbf{u}})$ by sampling from the lower-dimensional space of $\mathbf{u}$ instead of the joint space of $(\mathbf{u}, \tilde{\mathbf{u}})$. Using the law of total variance, we can show that this estimator does not increase variance, i.e.,

$$\mathbb{V}_{(\mathbf{u},\tilde{\mathbf{u}})\sim p(\mathbf{u},\tilde{\mathbf{u}})}[f(\mathbf{u}, \tilde{\mathbf{u}})] = \underbrace{\mathbb{E}[\mathbb{V}[f(\mathbf{u}, \tilde{\mathbf{u}}) \,|\, \mathbf{u}]]}_{\geq 0} + \underbrace{\mathbb{V}[\mathbb{E}[f(\mathbf{u}, \tilde{\mathbf{u}}) \,|\, \mathbf{u}]]}_{\mathbb{V}[\tilde{f}(\mathbf{u})]} \geq \mathbb{V}_{\mathbf{u}\sim p(\mathbf{u})}[\tilde{f}(\mathbf{u})]. \tag{3.103}$$

In some cases it suffices to condition on a single dimension to achieve a substantial variance reduction [96]. In this case, even if the required integral or summation is not available in closed form, it can be computed numerically. A similar idea of computing parts of an expectation analytically is used in [31].

3.4.4 The Reparameterization Trick

Another approach to obtain Monte Carlo gradients of an expectation is the reparameterization trick. The main idea of this technique is to compute samples $\mathbf{W} \sim q_{\boldsymbol{\nu}}(\mathbf{W})$ by transforming samples $\boldsymbol{\varepsilon}$ from some fixed parameter-free distribution $p(\boldsymbol{\varepsilon})$ using the variational parameters $\boldsymbol{\nu}$ as $\mathbf{W} = g(\boldsymbol{\nu}, \boldsymbol{\varepsilon})$. This allows us to rewrite the expected loss $\mathcal{L}_{\mathbb{E}}(\boldsymbol{\nu})$ as

$$\mathbb{E}_{\mathbf{W}\sim q_{\boldsymbol{\nu}}(\mathbf{W})}[\mathcal{L}(\mathbf{W})] = \mathbb{E}_{\boldsymbol{\varepsilon}\sim p(\boldsymbol{\varepsilon})}[\mathcal{L}(g(\boldsymbol{\nu}, \boldsymbol{\varepsilon}))]. \tag{3.104}$$

Assuming that differentiation and integration are interchangeable, the gradient of (3.104) is given by

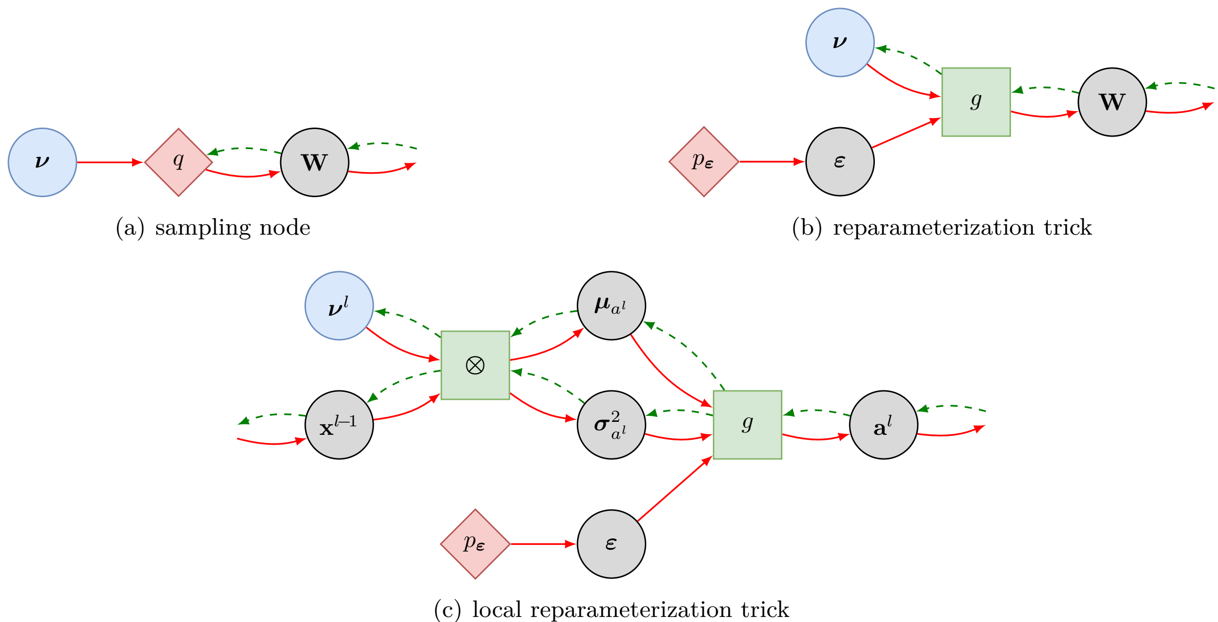
$$\nabla_{\boldsymbol{\nu}} \mathbb{E}_{\boldsymbol{\varepsilon}\sim p(\boldsymbol{\varepsilon})}[\mathcal{L}(g(\boldsymbol{\nu}, \boldsymbol{\varepsilon}))] = \mathbb{E}_{\boldsymbol{\varepsilon}\sim p(\boldsymbol{\varepsilon})}[\nabla_{\boldsymbol{\nu}} \mathcal{L}(g(\boldsymbol{\nu}, \boldsymbol{\varepsilon}))]. \tag{3.105}$$

It is then straightforward to estimate the gradient by sampling $\boldsymbol{\varepsilon} \sim p(\boldsymbol{\varepsilon})$. This is illustrated in Figure 3.5(b). The resulting estimator typically exhibits substantially lower variance than the log-derivative trick. However, the reparameterization trick is only applicable to continuous distribution $q_{\boldsymbol{\nu}}(\mathbf{W})$ for which a reparameterization $\mathbf{W} = g(\boldsymbol{\nu}, \boldsymbol{\varepsilon})$ exists.

Fortunately, this is the case for many distributions encountered in practice. Given that the inverse cdf $\Phi_{\boldsymbol{\nu}}^{-1}$ of a distribution is available in closed form, a sample from that distribution can be obtained by sampling $\varepsilon \sim \mathcal{U}([0, 1])$ and computing $\Phi_{\boldsymbol{\nu}}^{-1}(\varepsilon)$. More generally, we can often generate samples by applying a simple transformation to samples generated from a fixed base distribution. Many distributions belong to the location-scale family of distributions that are specified by a location parameter μ and a scale parameter σ (e.g., a Gaussian or a uniform distribution over an arbitrary interval). For these distributions, a sample can be generated by computing $g(\mu, \sigma, \varepsilon) = \mu + \sigma\varepsilon$ where ε is drawn from the base distribution specified by $\mu = 0$ and $\sigma = 1$. This also generalizes to the multivariate case, e.g., we can sample from a multivariate Gaussian $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ by computing $\boldsymbol{\mu} + \boldsymbol{\Sigma}^{\frac{1}{2}}\boldsymbol{\varepsilon}$ for $\boldsymbol{\varepsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. Other examples are sampling from a log-normal distribution by exponentiation of samples from a Gaussian or sampling from a gamma distribution by summing over several samples from exponential distributions.

The reparameterization trick is the predominant approach to optimize the variational inference objective (3.81) using SGD. In the continuous case, many works employ a Gaussian approximation $q_{\boldsymbol{\nu}}(\mathbf{W})$ for which the reparameterization trick is applicable. The reparameterization trick has been brought into the machine learning community concurrently by several works, e.g., to learn deep latent Gaussian models [98], to train variational autoencoders [62], and for various applications such as inferring the hyperparameters of Gaussian processes [99].

Based on these works, Blundell et al. [74] employed the reparameterization trick to train a Gaussian approximation $q_{\boldsymbol{\nu}}(\mathbf{W})$ for DNNs. They showed improved results for a scale mixture



(a) sampling node

(b) reparameterization trick

(c) local reparameterization trick

Figure 3.5: Computation graphs with sampling nodes (diamonds). Red arrows indicate the forward path. Green dashed arrows indicate the backward path. (a) Sampling directly from $q_{\boldsymbol{\nu}}(\mathbf{W})$ blocks the gradient flow during backpropagation. (b) The reparameterization trick moves the sampling node away from the path to the variational parameters $\boldsymbol{\nu}$. Therefore, we can compute the gradient with respect to $\boldsymbol{\nu}$ using backpropagation. (c) The local reparameterization trick computes the Gaussian activation distribution ($\boldsymbol{\mu}_{a^l}$ and $\boldsymbol{\sigma}^2_{a^l}$) from the inputs $\mathbf{x}^{l-1}$ and the variational parameters $\boldsymbol{\nu}$. Subsequently, the activations $\mathbf{a}^l$ are sampled using the reparameterization trick.

prior $p(\mathbf{W})$ modeled by a mixture of two zero-mean Gaussians with different variances, i.e.,

$$p(\mathbf{W}) = \alpha \mathcal{N}(0, \gamma_1^2) + (1 - \alpha)\mathcal{N}(0, \gamma_2^2). \tag{3.106}$$

This prior does not admit an analytic expression of the KL divergence term in (3.81), but it can be easily optimized using the reparameterization trick.

The Local Reparameterization Trick

Although the reparameterization trick already exhibits lower variance than the log-derivative trick, it is still possible to exploit structural properties of DNNs to reduce the variance even further. In particular, the *local reparameterization trick* [33]—being reminiscent of the ideas employed in the probabilistic forward pass—pushes the uncertainty from the approximate posterior $q_{\boldsymbol{\nu}}(\mathbf{W})$ to the activations $\mathbf{a}^l$ of the subsequent layer. Then the standard reparameterization trick is applied to sample from the induced activation distribution $q(\mathbf{a}^l)$ instead (see Figure 3.5(c)).

For this purpose, we assume a factorized Gaussian approximation $q_{\boldsymbol{\nu}}(\mathbf{W})$ and that the inputs from the previous layer $\mathbf{x}^{l-1}$ are deterministic values. Then the activations of $\mathbf{a}^l$, computed as sums over several random variables, follow a factorized Gaussian distribution $q(\mathbf{a}^L)$ with parameters

$$\mu_{a_i^l} = \sum_k \mathbb{E}[w_{i,k}^l]\, x_k^{l-1} \qquad \text{and} \qquad \sigma^2_{a_i^l} = \sum_k \mathbb{V}[w_{i,k}^l]\, (x_k^{l-1})^2. \tag{3.107}$$

As a consequence, the activation samples $\mathbf{a}^l$ are distributed according to $q(\mathbf{a}^l)$ regardless of whether we first sample the weights $\mathbf{W}^l$ and then compute the activations $\mathbf{a}^l$ or whether we sample directly from the induced activation distribution $q(\mathbf{a}^l)$.

However, the benefit of sampling from the activations is twofold. The first benefit is concerned with computational efficiency and is best seen by considering a practical implementation where data samples are processed concurrently as mini-batches to exploit parallel computation. In this case, we assume a loss $\mathcal{L}$ that can be written as a sum of per-sample losses ℓ_n as in (2.10). If the reparameterization trick is applied globally to sample the weights $\mathbf{W}$, these weights are reused for the entire mini-batch. As a consequence, the estimates of the individual per-sample losses ℓ_n might exhibit a significant covariance $\text{cov}(\ell_n, \ell_{n'})$ that increases the variance of their estimated sum $\mathcal{L}$. In principle, this can be avoided by sampling individual weights $\mathbf{W}$ for each per-sample loss ℓ_n, but this would be highly impractical from a computational perspective. Fortunately, this is not necessary since applying the local reparameterization trick to directly sample from the induced activation distribution $q(\mathbf{a}^l)$ has the same effect.

The second benefit is that the variance of the gradient is reduced even if we sample a single per-sample loss term ℓ_n. Intuitively, the reason for this is that there are many more weights than there are activations and each of these weights is subject to its individual noise. Consequently, the contribution of an individual weight to an activation becomes obscured due to the noise, which is harmful during backpropagation. If we alternatively perform the local reparameterization trick to sample the activations, there is a single point of noise injection and the contribution of each weight[16] to the sampled activations remains traceable. For a more rigorous treatment of this statement, we refer to [33].

Compared to the probabilistic forward pass, the local reparameterization trick can be computed more efficiently as it operates on deterministic inputs $\mathbf{x}^{l-1}$. This allows us to compute the activation variance according to (3.107) instead of (3.67) and to save two linear operations (matrix multiplications or convolutions) per layer. Moreover, for the local reparameterization trick the activations $\mathbf{a}^l$ are independent, whereas the probabilistic forward pass explicitly assumes independency since here the inputs $\mathbf{x}^{l-1}$ are stochastic. However, the issue of the probabilistic forward pass for shared weights persists. In particular, sampling the activations $\mathbf{a}^l$ is equivalent to sampling different weight values for each activation a_i^l although the underlying architecture assumes that the weights are equal.

So far, we have assumed that the variational distribution $q_{\boldsymbol{\nu}}(\mathbf{W})$ is a Gaussian such that the induced activation distribution $q(\mathbf{a}^l)$ is also Gaussian. However, even if the variational distribution $q_{\boldsymbol{\nu}}(\mathbf{W})$ is non-Gaussian, we can appeal to the central limit theorem and approximate the activation distribution $q(\mathbf{a}^l)$ well using a Gaussian. As we will see in Chapter 5, we can even apply the Gaussian activation approximation if the weight distributions $q_{\boldsymbol{\nu}}(\mathbf{W})$ are discrete. This is an important step of our method for training discrete-valued DNNs.

3.4.5 The Gumbel-Softmax Approximation

The major disadvantage of the reparameterization trick is that it is not applicable to discrete distributions. As a consequence, one has to apply different techniques such as the more general log-derivative trick that is known to suffer from high variance unless suitable variance reduction techniques are used. For many applications, the variance might be too high and optimization will not yield meaningful results in any reasonable amount of time. In these cases, it is often sensible to reduce the variance at the cost of introducing some bias into the sampling process.

This idea is pursued by the *Gumbel-softmax* approximation which allows us to apply the reparameterization trick to discrete distributions [100]. Note that the Gumbel-softmax approximation has been concurrently introduced by Maddison et al. [101] under the name *concrete distribution*, but we will stick to the Gumbel-softmax terminology throughout this thesis. Samples from a categorical distribution $p(v)$ are typically generated by sampling $\varepsilon \sim \mathcal{U}([0,1])$ and evaluating the inverse cdf $\Phi_p^{-1}(\varepsilon)$. The Gumbel-softmax approximation utilizes a different elegant technique for sampling from a categorical distribution known as the Gumbel-max trick.

[16] More precisely, its associated variational parameters $\boldsymbol{\nu}_i$

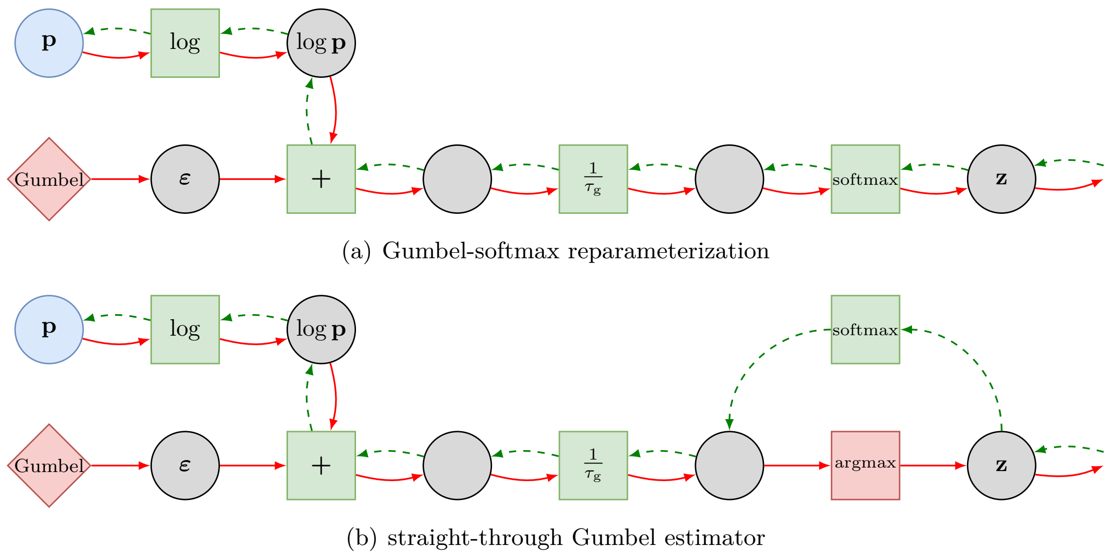


Figure 3.6: (a) Computation graph of the Gumbel-softmax reparameterization to obtain a sample $\mathbf{z}$*. Red arrows indicate the forward path. Green dashed arrows indicate the backward path. (b) The straight-through Gumbel estimator computes the* argmax *in the forward path to obtain a one-hot encoded sample* $\mathbf{z}$*. In the backward path, it computes the gradient of the* softmax *using the STE.*

Given a categorical random variable with K possible outcomes $\{v_1, \ldots, v_K\}$ and corresponding occurrence probabilities $\mathbf{p} = (p_1, \ldots, p_K)$, the Gumbel-max trick generates a sample $v_{\tilde{k}}$ according to $\mathbf{p}$ as

$$\tilde{k} = \operatorname*{argmax}_k \log(p_k) + \varepsilon_k \qquad \text{with} \qquad \varepsilon_k \sim \text{Gumbel}(0, 1). \tag{3.108}$$

We can generate samples $\varepsilon \sim \text{Gumbel}(0, 1)$ by transforming a uniform sample ε' according to

$$\varepsilon = -\log(-\log(\varepsilon')) \qquad \text{with} \qquad \varepsilon' \sim \mathcal{U}([0, 1]). \tag{3.109}$$

The Gumbel-softmax distribution is based on one-hot encodings of the indices k corresponding to the discrete values v_k. By viewing the argmax in (3.108) as a function mapping to one-hot encodings of the corresponding $\tilde{k}$ (see (2.20)), a continuous relaxation is obtained by applying the softmax function

$$z_k = \text{softmax}_k((\log(\mathbf{p}) + \boldsymbol{\varepsilon})/\tau_\text{g}) = \frac{\exp((\log(p_k) + \varepsilon_k)/\tau_\text{g})}{\sum_{k'}^K \exp((\log(p_{k'}) + \varepsilon_{k'})/\tau_\text{g})}, \tag{3.110}$$

where $\tau_\text{g} > 0$ is a temperature parameter. For $\tau_\text{g} \to 0$, we obtain a one-hot representation of the maximum argument. For $\tau_\text{g} \to \infty$, we recover a uniform distribution over the K-dimensional probability simplex. The Gumbel-softmax reparameterization is illustrated in Figure 3.6(a).

By carefully selecting the temperature τ_g, one can ensure that the continuous approximation (3.110) is sufficiently close to a one-hot vector while still allowing valuable gradient information to flow during backpropagation. In case the sampled values are required in form of their original discrete values v_k instead of a one-hot encoding thereof, these can be recovered by computing an expectation with respect to the softmax values $\mathbf{z}$ from (3.110) as

$$\mathbb{E}_\mathbf{z}[v] = \sum_{k=1}^K z_k v_k. \tag{3.111}$$

However, we emphasize that this does generally not yield any of the original discrete values v_k

and it depends on the application whether it is acceptable to operate on continuous relaxations thereof. In case the values v_k correspond to actual numerical values with a natural ordering, it might be acceptable to work on their continuous relaxations. However, if the values v_k are used to encode categories and mixing these categories does not have any meaningful interpretation, it might be required to operate on the discrete values v_k. In these cases, it is proposed to apply the STE by computing the argmax function in the forward pass and to apply the gradient of the softmax during backpropagation. This is illustrated in Figure 3.6(b). In [100] this is also called the *straight-through Gumbel estimator*.

The Gumbel-softmax approximation is used on two occasions in this thesis. In Chapter 5, we apply the Gumbel-softmax approximation to sample from the binary distribution obtained after the sign activation function. In this case, we have $v_1 = -1$ and $v_2 = 1$ and we obtain good results using the continuous approximation. In Chapter 7, we apply the Gumbel-softmax approximation to sample from gating variables that determine the structure of a BN. Here, applying the continuous approximation would eliminate the probabilistic interpretation of the BN and it is necessary to apply the STE.

3.5 Bayesian Neural Networks Using Sampling

Although HMC (see Section 3.2.4) provides excellent performance in many applications, its running time for posterior inference does not scale well to very large datasets containing tens of thousands of data samples or even more. The reasons for this are the same as those why batch gradient descent does not scale to large datasets and why one usually applies SGD instead (see Section 2.1.2). Similar to SGD, stochastic gradient MCMC methods have been developed that utilize gradients computed from a subset of the dataset to generate new samples. We briefly review two particular stochastic gradient MCMC methods, namely stochastic gradient Langevin dynamics (SGLD) [102] and stochastic gradient Hamiltonian Monte Carlo (SGHMC) [103].

For our discussion about stochastic gradient MCMC methods, we adopt the terminology from our discussion on SGD. In particular, we consider sampling of parameters $\boldsymbol{\theta}$ from a continuous parameter space $\boldsymbol{\Theta}$. For DNNs, $\boldsymbol{\theta}$ corresponds to the weights $\mathbf{W}$. Furthermore, we perform sampling from a posterior distribution governed by a dataset $\mathcal{D}$ whose log-density is given by

$$\log p(\boldsymbol{\theta} \,|\, \mathcal{D}) = \sum_{n=1}^{N} \log p(\mathbf{x}_n \,|\, \boldsymbol{\theta}) + \log p(\boldsymbol{\theta}) + \text{const}, \tag{3.112}$$

where the constant term corresponds to a normalization term that does not depend on the parameters $\boldsymbol{\theta}$.

3.5.1 Stochastic Gradient Langevin Dynamics

A special algorithm, known as Langevin Monte Carlo (LMC), is obtained by performing a single leapfrog iteration (i.e., $T = 1$) within HMC [104]. The name LMC stems from the fact that it performs a simulation of Langevin dynamics. LMC necessarily lacks the desirable property to make distant proposals in state space which, in HMC, is a direct consequence of simulating Hamiltonian dynamics for a sufficiently large number of steps T. However, LMC is appealing due to its algorithmic similarity to gradient descent (see Section 2.1.2). More specifically, an update of LMC takes the form

$$\boldsymbol{\theta}^t = \boldsymbol{\theta}^{t-1} + \frac{\eta}{2} \left(\sum_{n=1}^{N} \nabla_{\boldsymbol{\theta}} \log p(\mathbf{x}_n \,|\, \boldsymbol{\theta}^{t-1}) + \nabla_{\boldsymbol{\theta}} \log p(\boldsymbol{\theta}^{t-1}) \right) + \boldsymbol{\varepsilon}, \tag{3.113}$$

Algorithm 6 Stochastic Gradient Langevin Dynamics (SGLD)

1: **Input:** $\mathcal{D} = \{\mathbf{x}_1, \dots, \mathbf{x}_N\}$, initial parameters $\boldsymbol{\theta}^0$, step sizes $(\eta_t)_{t \geq 1}$ conforming (3.115)
2: **for** $t = 1$ **to** $\dots$ **do**
3: $\quad \mathcal{D}_t \leftarrow$ randomly select mini-batch of size N_B from $\mathcal{D}$
4: $\quad$ Draw $\boldsymbol{\varepsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}\eta_t)$
5: $\quad \mathbf{g} \leftarrow (N/N_\text{B}) \sum_{\mathbf{x} \in \mathcal{D}_t} \nabla_{\boldsymbol{\theta}} \log p(\mathbf{x} \,|\, \boldsymbol{\theta}^{t-1}) + \nabla_{\boldsymbol{\theta}} \log p(\boldsymbol{\theta}^{t-1})$
6: $\quad \boldsymbol{\theta}^t \leftarrow \boldsymbol{\theta}^{t-1} + (\eta_t/2)\mathbf{g} + \boldsymbol{\varepsilon}$
7: **end for**

where $\boldsymbol{\varepsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}\eta)$. Equation (3.113) differs from the update equation of gradient descent only in the additional noise term $\boldsymbol{\varepsilon}$. Considering the success of stochastic optimization methods such as SGD, it appears natural to also approximate the gradient in (3.113) using stochastic gradients obtained from subsets of the dataset $\mathcal{D}$. Indeed, after applying some modifications to (3.113), the SGLD algorithm [102] is obtained by repeatedly applying

$$\boldsymbol{\theta}^t = \boldsymbol{\theta}^{t-1} + \frac{\eta_t}{2} \left(\frac{N}{N_\text{B}} \sum_{\mathbf{x} \in \mathcal{D}_t} \nabla_{\boldsymbol{\theta}} \log p(\mathbf{x} \,|\, \boldsymbol{\theta}^{t-1}) + \nabla_{\boldsymbol{\theta}} \log p(\boldsymbol{\theta}^{t-1}) \right) + \boldsymbol{\varepsilon}_t, \tag{3.114}$$

where $\mathcal{D}_t$ is a randomly selected mini-batch of size N_B, $\boldsymbol{\varepsilon}_t \sim \mathcal{N}(\mathbf{0}, \mathbf{I}\eta_t)$, and $\eta_t > 0$ is a decaying step size parameter satisfying

$$\sum_{t=1}^{\infty} \eta_t = \infty \qquad \text{and} \qquad \sum_{t=1}^{\infty} \eta_t^2 < \infty. \tag{3.115}$$

Algorithm 6 shows a pseudocode of SGLD. Most importantly, the variance of the noise term $\boldsymbol{\varepsilon}_t$ is coupled to the decaying step size η_t, causing the algorithm to operate in two stages. In the first stage, we assume that the parameters $\boldsymbol{\theta}^t$ are not close to a local optimum such that the gradient exhibits a large magnitude. Consequently, the weight updates in (3.114) are largely determined by the stochastic gradients and, therefore, SGLD behaves much like an optimization algorithm.

In the second stage, the influence of the noise term $\boldsymbol{\varepsilon}_t$ increases since (i) the gradients become smaller and (ii) the step size η_t decreases faster than the standard deviation $\sqrt{\eta_t}$ of the noise term $\boldsymbol{\varepsilon}_t$. This results in the exploration of the distribution such that eventually samples from the true posterior $p(\boldsymbol{\theta} \,|\, \mathcal{D})$ are generated.

SGLD requires the choice of a suitable step size schedule that satisfies conditions (3.115). In [102] the step size is computed according to

$$\eta_t = \frac{\alpha_\eta}{(\beta_\eta + t)^{\gamma_\eta}} \tag{3.116}$$

where $\gamma_\eta \in (0.5, 1]$, resulting in three additional hyperparameters.

3.5.2 Stochastic Gradient Hamiltonian Monte Carlo

It is natural to ask whether HMC can be generalized to a stochastic version for $T > 1$, similarly as LMC has been generalized to SGLD. The answer turns out to be positive [103], but the derivation of a stochastic version of HMC requires more care. For the analysis in [103], the stochastic gradients obtained from mini-batches are assumed to be normally distributed by appealing to the central limit theorem. This allows us to write the true gradient $\nabla_{\boldsymbol{\theta}} \log p(\boldsymbol{\theta} \,|\, \mathcal{D})$ as a stochastic gradient plus zero-mean Gaussian noise $\boldsymbol{\varepsilon}$ whose variance depends on the current

Algorithm 7 Stochastic Gradient Hamiltonian Monte Carlo (SGHMC)

1: **Input:** $\mathcal{D} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$, initial parameters $\boldsymbol{\theta}^0$, fixed step size η, momentum ξ, T
2: **for** $t = 1$ **to** $\ldots$ **do**
3: Draw $\bar{\mathbf{v}}^0 \sim \mathcal{N}(\mathbf{0}, \mathbf{I}\eta)$
4: $\bar{\boldsymbol{\theta}}^0 \leftarrow \boldsymbol{\theta}^{t-1}$
5: **for** $k = 1$ **to** T **do**
6: $\mathcal{D}_{t,k} \leftarrow$ randomly select mini-batch of size N_B from $\mathcal{D}$
7: Draw $\boldsymbol{\varepsilon} \sim \mathcal{N}(\mathbf{0}, 2\eta\xi\mathbf{I})$
8: $\mathbf{g} \leftarrow (N/N_\mathrm{B}) \sum_{\mathbf{x} \in \mathcal{D}_{t,k}} \nabla_{\boldsymbol{\theta}} \log p(\mathbf{x} \,|\, \bar{\boldsymbol{\theta}}^{k-1}) + \nabla_{\boldsymbol{\theta}} \log p(\bar{\boldsymbol{\theta}}^{k-1})$
9: $\bar{\boldsymbol{\theta}}^k \leftarrow \bar{\boldsymbol{\theta}}^{k-1} + \bar{\mathbf{v}}^{k-1}$
10: $\bar{\mathbf{v}}^k \leftarrow (1 - \xi)\bar{\mathbf{v}}^{k-1} + \eta\mathbf{g} + \boldsymbol{\varepsilon}$
11: **end for**
12: $\boldsymbol{\theta}^t \leftarrow \bar{\boldsymbol{\theta}}^T$
13: **end for**

parameters $\boldsymbol{\theta}$ and the size of the mini-batches N_B, i.e.,

$$\nabla_{\boldsymbol{\theta}} \log p(\boldsymbol{\theta} \,|\, \mathcal{D}) = \frac{N}{N_\mathrm{B}} \sum_{\mathbf{x} \in \mathcal{D}_t} \nabla_{\boldsymbol{\theta}} \log p(\mathbf{x} \,|\, \boldsymbol{\theta}) + \nabla_{\boldsymbol{\theta}} \log p(\boldsymbol{\theta}) + \boldsymbol{\varepsilon}. \tag{3.117}$$

However, by simply adding Gaussian noise to the momentum updates of HMC, (3.48) and (3.50), the generated samples follow a distribution that exhibits increased entropy over time, and therefore, this distribution does not correspond to the desired stationary distribution anymore. Fortunately, we can correct for this behavior by adding a properly weighted friction term to the momentum updates such that the stationary distribution remains invariant under the resulting updates.

Similar as SGLD can be phrased as SGD with an additional noise term, SGHMC can be phrased as SGD with momentum and an additional noise term. The corresponding pseudocode is shown in Algorithm 7. The momentum formulation is appealing as experience about good settings of the SGD hyperparameters η and ξ_mom in (2.11) can be used to determine the corresponding SGHMC hyperparameters $\eta > 0$ and $\xi \in (0, 1)$.

So far, we have ignored the fact that both algorithms, SGLD and SGHMC, introduce errors that need to be corrected by a Metropolis-Hastings acceptance step. However, a Metropolis-Hastings acceptance step would require an evaluation of the whole dataset $\mathcal{D}$, the avoidance of which was the motivation for using stochastic gradients in the first place. Indeed, it is not even possible to compute the required Metropolis-Hastings acceptance probability since the probability of the reverse step $\hat{p}(\boldsymbol{\theta}^{t-1} \,|\, \boldsymbol{\theta}^t)$ cannot be computed. However, both SGLD and SGHMC rely on a decaying step size and it is argued that in the limit of $\eta_t \to 0$ the simulation becomes exact and the rejection probability of a Metropolis-Hastings acceptance step becomes negligible. Decaying step sizes, on the other hand, reduce the efficiency of the algorithm as it takes longer to explore the state space using a smaller step size η_t. In practice, we typically trade off some bias due to missing Metropolis-Hastings steps for more efficient exploration of the state space with non-zero η_t. Note that there exist methods that approximate Metropolis-Hastings steps using subsets of the data [105, 106].

4 Resource-Efficient Deep Neural Networks

While DNNs are the driving factor behind many recent success stories of machine learning, they are notoriously data and resource hungry—a property which has recently renewed significant research interest in resource-efficient approaches. This chapter provides an extensive overview of the current research directions, all of which are concerned with reducing the model size and/or improving run-time efficiency, while at the same time maintaining accuracy levels close to state-of-the-art models. We have identified three major research directions concerned with improving resource efficiency in DNNs (see Figure 1.1). In particular, these directions are

Quantized Neural Networks The weights of a DNN are typically stored as 32-bit floating-point values and computing predictions requires millions of floating-point operations. Quantization approaches reduce the number of bits required to store the weights and the activations of DNNs. While quantization approaches obviously reduce the memory footprint of a DNN, the selected numerical formats potentially also facilitate faster predictions using cheaper arithmetic operations. Even reducing precision down to binary or ternary values works reasonably well and essentially reduces DNNs to hardware-friendly logical circuits.

Network Pruning Starting from a fixed, potentially large DNN architecture, pruning approaches remove parts of the architecture during training or after training as a post-processing step. The parts being removed range from the very local scale of individual weights—which is called *unstructured pruning*—to a more global scale of neurons, channels, or even entire layers—which is called *structured pruning*. On the one hand, unstructured pruning is typically less sensitive to accuracy degradation, but special sparse matrix operations are required to obtain a computational benefit. On the other hand, structured pruning is more sensitive to accuracy degradation but the resulting data structures remain dense such that common highly optimized dense matrix operations available on most off-the-shelf hardware can be used.

Structural Efficiency This category comprises a diverse set of approaches that achieve resource efficiency at the structural level of DNNs: The idea of *knowledge distillation* is to train a small student DNN to mimic the behavior of a larger teacher DNN, which has been shown to yield improved results compared to training the small DNN directly. *Weight sharing* methods reduce the memory footprint by using only few weights that are shared among the connections. Several works have investigated *special matrix structures* that require fewer parameters and allow for faster matrix multiplications—the main workload in fully connected layers. Furthermore, there exist several *manually designed architectures* that introduce lightweight building blocks or modify existing building blocks to improve resource efficiency. Most recently, *NAS* methods have emerged that discover efficient DNN architectures automatically.

In the following overview, we will motivate each of these categories in turn and discuss representative approaches. Since not every approach can be attributed clearly to a single category, we present the individual approaches in the category where we think their contribution is most significant. Furthermore, we emphasize that many of the presented techniques are not mutually exclusive and that they can potentially be combined to further enhance resource efficiency. For instance, one can both sparsify a model *and* reduce arithmetic precision.

This chapter is largely based on our survey paper [107]. Besides some minor adaptations, this chapter extends the literature overview in our initial version of [107] by some references. Most importantly, we included literature concerned with *mixed-precision quantization*, i.e., methods that infer individual bit widths for different parts of a DNN (e.g., per layer) during the training procedure. Moreover, we extended the review on knowledge distillation methods by further references.

4.1 Quantized Neural Networks

Quantization in DNNs is concerned with reducing the number of bits used for the representation of the weights and the activations.[17] The reduction in memory requirements are obvious: Using fewer bits for the weights results in a lower memory overhead for storing the corresponding model, and using fewer bits for the activations results in a lower memory overhead for computing predictions. Furthermore, representations using fewer bits often facilitate faster computation. For instance, when quantization is driven to the extreme with binary weights $w \in \{-1, 1\}$ *and* binary activations $x \in \{-1, 1\}$, floating-point or fixed-point multiplications are replaced by hardware-friendly logical XNOR and bitcount operations. In this way, a sophisticated DNN is essentially reduced to a logical circuit.

However, training such discrete-valued DNNs[18] is difficult as they cannot be directly optimized using gradient-based methods. The challenge is to reduce the number of bits as much as possible while at the same time keeping the prediction accuracy close to that of a well-tuned full-precision DNN. In the following, we provide a literature overview of approaches that train reduced-precision DNNs, and, in a broader view, we also consider methods that use reduced-precision computations during backpropagation to facilitate low-resource training.

4.1.1 Early Quantization Approaches

Approaches for reduced-precision computations date back at least to the early 1990s. Höhfeld and Fahlman [108, 109] rounded the weights during training to fixed-point formats with different numbers of bits. They observed that training eventually stalls as small gradient updates are always rounded to zero. As a remedy, they proposed stochastic rounding, i.e., rounding values to the nearest value with a probability proportional to the distance to the nearest value. These quantized gradient updates are correct in expectation, do not cause training to stall, and yield good performance with substantially fewer bits than deterministic rounding. More recently, Gupta et al. [110] have shown that stochastic rounding can also be applied to modern deep architectures, as demonstrated on a hardware prototype.

Lin et al. [111] propose a method to reduce the number of multiplications required during training. At forward propagation, the weights are stochastically quantized to either binary weights $w \in \{-1, 1\}$ or ternary weights $w \in \{-1, 0, 1\}$ to remove the need for multiplications at all. During backpropagation, inputs and hidden neurons are quantized to powers of two, reducing multiplications to cheaper bit shift operations, and leaving only a negligible number of floating-point multiplications to be computed. However, the speed-up is limited to training since for testing the full-precision weights are required.

Courbariaux et al. [112] empirically studied the effect of different numeric formats (i.e., floating-point, fixed-point, and dynamic fixed-point) with varying bit widths on the performance

[17] In this context, activation quantization typically refers to quantizing the layer outputs $\mathbf{x}^l = h^l(\mathbf{a}^l)$ *after* the activation function h^l has been applied. Therefore, activation quantization is often modeled directly through the activation function h^l itself, e.g., by using a piecewise constant function.

[18] Due to finite precision of computer arithmetic, in fact any DNN is discrete-valued. However, we use this term here to emphasize the extremely small number of values.

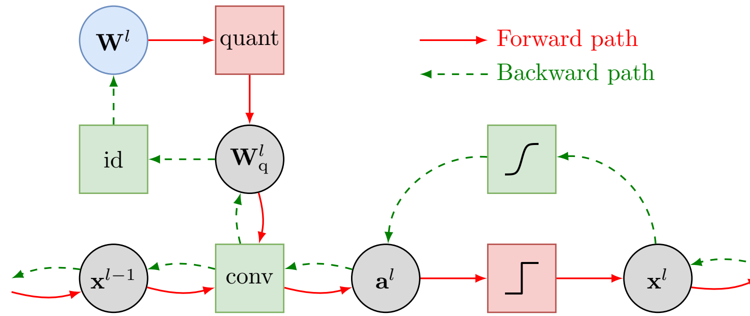


Figure 4.1: STE in a convolutional layer. The green boxes indicate differentiable functions. The red boxes indicate piecewise constant functions whose gradient is zero almost everywhere. The blue box indicates learnable weights. quant *denotes a quantization function, e.g., a rounding function, and* id *is the identity function. During forward propagation, the red path is followed, whereas during backpropagation, the dashed green path is followed to avoid the red zero-gradient boxes.*

of DNNs. Lin et al. [113] consider fixed-point quantization of pre-trained full-precision DNNs. They formulate a convex optimization problem to minimize the total number of bits required to store the weights and the activations under the constraint that the total output signal-to-quantization noise ratio is larger than a certain prespecified value. A closed-form solution of the convex objective yields layer-specific bit widths.

4.1.2 Quantization-Aware Training

Quantization operations, being piecewise constant functions with either undefined or zero derivatives, are not applicable to gradient-based learning using backpropagation. In recent years, the STE [14] (see Section 2.1.4) became the method of choice to compute an approximate gradient for training DNNs with weights that are represented using a very small number of bits. Such methods typically maintain a set of full-precision weights that are quantized during forward propagation. During backpropagation, the gradients are propagated through the quantization functions by assuming that their gradient equals one. In this way, the full-precision weights are updated using gradients computed at the quantized weights. At test-time, the full-precision weights are abandoned and only the quantized reduced-precision weights are kept. In a similar manner, many methods employ the STE to approximate the gradient for the quantization of activations. Figure 4.1 shows the computation graph of a typical DNN layer using quantized weights and activations. We refer to this scheme as *quantization-aware training* since quantization is an essential part during forward propagation, and it is intuitive to think of the real-valued weights becoming robust to quantization.

In [114], binary weight DNNs are trained using the STE to get rid of expensive floating-point multiplications. They consider deterministic rounding using the sign function and stochastic rounding using probabilities determined by the hard sigmoid function $\max(0, \min(1, (w+1)/2))$. During backpropagation, a set of auxiliary full-precision weights is updated based on the gradients of the quantized weights. Hubara et al. [115] extended this work by also quantizing the activations to a single bit using the sign activation function. This reduces the computational burden dramatically as floating-point multiplications and additions are reduced to hardware-friendly logical XNOR and bitcount operations, respectively.

Li et al. [116] trained ternary weights $w \in \{-\alpha, 0, \alpha\}$. Their quantizer sets weights whose magnitude is lower than a certain threshold Δ to zero, while the remaining weights are set to $\pm\alpha$ according to their sign. Their approach determines $\alpha > 0$ and Δ *during forward propagation* by approximately minimizing the squared quantization error of the real-valued weights. Zhu et al. [117] extended this work to ternary weights $w \in \{-\alpha, 0, \beta\}$ where $\alpha > 0$ and $\beta > 0$ are trainable parameters subject to gradient updates. They propose to select Δ^l based on the

maximum full-precision weight magnitude in each layer l, i.e., $\Delta^l = \gamma \cdot \max\{|w| : w \in \mathbf{W}^l\}$ with γ being a hyperparameter. These asymmetric weights considerably improve performance compared to symmetric weights as used in [116].

Rastegari et al. [118] approximate full-precision weight filters in CNNs as $\mathbf{W} = \alpha \mathbf{B}$ where α is a scalar and $\mathbf{B}$ is a binary weight matrix. This reduces the bulk of floating-point multiplications inside the convolutions to either additions or subtractions and only requires a single multiplication per output neuron with the scalar α. In a further step, the layer inputs $\mathbf{x}^{l-1}$ are quantized in a similar way to perform the convolution using only efficient XNOR operations and bitcount operations, followed by two floating-point multiplications per output neuron. Again, the STE is used during backpropagation. Lin et al. [119] generalized the ideas of [118] by approximating the full-precision weights using linear combinations of *multiple* binary weight filters for improved classification accuracy.

While most activation binarization methods use the sign function which can be seen as an approximation to the tanh function, Cai et al. [120] proposed a half-wave Gaussian quantization that more closely resembles the predominant ReLU activation function.

Motivated by the fact that weights and activations typically exhibit a non-uniform distribution, Miyashita et al. [121] proposed to quantize values to powers of two. Their representation allows getting rid of expensive multiplications, and they report higher robustness to quantization than linear rounding schemes using the same number of bits. Zhou et al. [122] proposed incremental network quantization where the weights of a pre-trained DNN are first partitioned into two sets. The weights in the first set are quantized to either zero or powers of two. The weights in the second set are kept at full precision and retrained to recover from the potential accuracy degradation due to quantization. They iterate partitioning, quantization, and retraining until all weights are quantized.

Jacob et al. [123] proposed a quantization scheme that accurately approximates floating-point operations using only integer arithmetic to speed up computation. During training, their forward pass simulates the quantization step to keep the performance of the quantized DNN close to the performance of using single-precision. At test-time, weights are represented as 8-bit integer values, reducing the memory footprint by a factor of four.

Liu et al. [124] introduced Bi-Real net, a ResNet-inspired architecture where the residual path is implemented with efficient binary convolutions while the shortcut path is kept real-valued to preserve the expressiveness of the DNN. The residual in each layer is computed by first transforming the input with the sign activation, followed by a binary convolution, and a final batch normalization step.

Instead of using a fixed quantizer, in *LQ-Net* [125] the quantizer is adapted during training. The proposed quantizer is inspired by the representation of integers as linear combinations $\mathbf{v}^\top \mathbf{b}$ with $\mathbf{v} = (2^0, \ldots, 2^{K-1})$ and $\mathbf{b} \in \{0,1\}^K$. The key idea is to consider a quantizer that assigns values to the nearest value representable as such a linear combination $\mathbf{v}^\top \mathbf{b}$ and to treat $\mathbf{v} \in \mathbb{R}^K$ as trainable parameters. It is shown that such a quantizer is compatible with efficient bit operations. The quantizer is optimized *during forward propagation* by minimizing the quantization error objective $\|\mathbf{B}\mathbf{v} - \mathbf{x}\|^2$ for $\mathbf{B} \in \{-1,1\}^{N \times K}$ and $\mathbf{v}$ by alternately fixing $\mathbf{B}$ and minimizing $\mathbf{v}$ and vice versa. It is proposed to use layerwise quantizers for the activations and channel-wise quantizers for the weights, i.e., an individual quantizer for each layer and channel, respectively.

Relaxed Quantization [126] introduces a stochastic differentiable soft rounding scheme. By injecting additive noise to the deterministic weights before rounding, one can compute probabilities of the weights being rounded to specific values in a predefined discrete set. Subsequently, these probabilities are used to differentiably round the weights using the Gumbel-softmax approximation [100]. Since this soft rounding scheme produces only values that are close to values from the discrete set but which are not exactly from this set, the authors also propose a hard variant using the STE.

Dong et al. [127] introduced Hessian-aware mixed-precision quantization for DNNs. Their

method quantifies the sensitivity of individual DNN blocks to weight quantization using the largest eigenvalue of the block-wise Hessian matrices which can be computed using the power iteration method. They compute two different orderings of the individual DNN blocks, both of which are based on these eigenvalues. The first ordering determines a relative ordering of the bit widths of individual blocks. This substantially reduces the exponential search space of layer-specific weights and allows them to manually set appropriate bit widths. The second ordering takes these bit widths into account and determines the sequence in which blocks are quantized and fine-tuned using quantization-aware training.

A linear quantizer has three characteristic properties, i.e., (i) a step size Q_d, (ii) a dynamic range $Q_{\max}$, and (iii) the number of bits Q_b. Since these quantities are interrelated according to

$$Q_{\max} = (2^{Q_b - 1} - 1) Q_d, \tag{4.1}$$

a linear quantizer is specified by knowing any two of them [128]. Given fixed layerwise bit widths Q_b^l, Esser et al. [129] incorporated layerwise step sizes Q_d^l as trainable parameters in the computation graph. By training the step sizes Q_d^l using the STE, they are adapted to the given objective. This is in contrast to previous work, such as XNOR-Net [118], that determine the step size Q_d^l using certain statistics obtained from the values to be quantized. Uhlich et al. [128] extended this idea to mixed-precision quantization. They investigated the three different possibilities to specify a linear quantizer (4.1) by only two of its characteristic properties and discovered substantial differences in the training behavior. They propose to parameterize the quantizers using the step size Q_d^l and the dynamic range $Q_{\max}^l$ and to train these values using backpropagation and the STE to obtain layerwise bit widths Q_b^l.

There also exist works that perform quantization *during* backpropagation to facilitate resource-efficient training. Zhou et al. [130] presented several quantization schemes for the weights and the activations that allow for flexible bit widths. Furthermore, they also propose a quantization scheme for backpropagation to facilitate low-resource training. In accordance with earlier work mentioned above, they note that stochastic quantization is essential for their approach. In [131], weights, activations, weight gradients, and activation gradients are subject to customized quantization schemes that allow for variable bit widths and facilitate integer arithmetic during training and testing. In contrast to [130], the work in [131] accumulates weight changes to low-precision weights instead of full-precision weights.

While most work on quantization based approaches is empirical, some works gained more theoretical insights [132, 133]. The recent work of Shekhovtsov et al. [16] has shown that for stochastic binary networks the STE arises from particular linearization approximations.

In Chapter 7, we show that quantization-aware training can be applied to other model classes beyond DNNs. In particular, we quantize the log-probability parameters of *BN classifiers* and contrast these models with quantized DNNs [134]. As opposed to other works that mostly consider DNN quantization in the context of large architectures and datasets, we find that quantization-aware training also performs well in the small-scale setting. Our work shows that BNs may provide a viable alternative to DNNs when it comes to trading off between prediction accuracy and computational aspects of the model.

4.1.3 Bayesian Approaches for Quantization

In this section, we review some quantization approaches, most of which are closely related to the Bayesian variational inference framework (see Section 3.4).

The work of Achterhold et al. [135] builds on the variational dropout based pruning approach of Louizos et al. [136] discussed in Section 4.2.3. They introduce a mixture of log-uniforms prior whose mixtures are centered at predefined quantization values. Consequently, the approximate posterior also concentrates at these values such that weights can be safely quantized without requiring a fine-tuning procedure.

The following works in this section directly operate on *discrete* weight distributions and, consequently, do not require a rounding procedure. Soudry et al. [83] approximate the true posterior $p(\mathbf{W} \,|\, \mathcal{D})$ over discrete weights using expectation propagation [54] with closed-form online updates. Starting with an uninformative approximation $q_{\boldsymbol{\nu}}(\mathbf{W})$, their approach combines the current approximation $q_{\boldsymbol{\nu}}(\mathbf{W})$ (serving as the prior in Bayes' rule (3.3)) with the likelihood for a single-sample dataset $\mathcal{D}_n = \{(\mathbf{x}_n, y_n)\}$ to obtain a refined posterior. To obtain a closed-form refinement step, they propose several approximations.

Although deviating from the Bayesian variational inference framework as no similarity measure to the true posterior is optimized, the approach of Shayer et al. [137] trains a distribution $q_{\boldsymbol{\nu}}(\mathbf{W})$ over either binary weights $w \in \{-1, 1\}$ or ternary weights $w \in \{-1, 0, 1\}$. They propose to minimize an expected loss $\mathbb{E}_{\mathbf{W} \sim q_{\boldsymbol{\nu}}(\mathbf{W})}[\mathcal{L}(\mathbf{W}; \mathcal{D})]$ for the variational parameters $\boldsymbol{\nu}$ with gradient-based optimization using the local reparameterization trick [33]. After training has finished, the discrete weights are obtained by either sampling or taking a mode from $q_{\boldsymbol{\nu}}(\mathbf{W})$. Since their approach is limited to the ReLU activation function, Peters and Welling [138] extended their work to the sign activation function. This involves several non-trivial changes since the sign activation, due to its zero derivative, requires that the local reparameterization trick must be performed *after* the sign function. Consequently, *distributions* need to be propagated through commonly used building blocks such as batch normalization and pooling operations.

Our work [139] presented in Chapter 5 directly fits into the line of research of Shayer et al. [137] and Peters and Welling [138]. We extend their works to beyond three distinct weights and, similarly as in [138], we apply the discrete sign activation function. Moreover, we introduce technical improvements such as a distribution-aware max pooling operation and a simpler initialization scheme for the variational distribution $q_{\boldsymbol{\nu}}(\mathbf{W})$.

Van Baalen et al. [88] propose a Bayesian mixed-precision quantization method for power-of-two bit widths. Their method is based on a recursive view of quantization where residual quantization errors are repeatedly quantized. They introduce gates that determine how many recursive quantization steps should be performed which in turn determines the number of used bits. While the quantization itself is subject to the STE, they propose to train gate probabilities using the Bayesian variational inference framework. The use of fewer bits for quantization is encouraged using a specific prior and, through an additional zero-bit gate, their framework simultaneously allows for weight pruning.

Havasi et al. [140] introduced a novel Bayesian compression technique that we present here in this section although it is rather a coding technique than a quantization technique. In a nutshell, their approach first computes a variational distribution $q_{\boldsymbol{\nu}}(\mathbf{W})$ over real-valued weights using mean field variational inference and then it encodes a sample $\mathbf{W}$ from $q_{\boldsymbol{\nu}}(\mathbf{W})$ in a smart way. They construct an approximation $\tilde{q}(\mathbf{W})$ to $q_{\boldsymbol{\nu}}(\mathbf{W})$ by importance sampling using the prior $p(\mathbf{W})$ as

$$q_{\boldsymbol{\nu}}(\mathbf{W}) \approx \tilde{q}(\mathbf{W}) = \sum_{i=1}^{2^K} \frac{q_{\boldsymbol{\nu}}(\mathbf{W}^i)}{p(\mathbf{W}^i)} \delta_{\mathbf{W}^i}(\mathbf{W}) \qquad \text{with} \qquad \mathbf{W}^i \sim p(\mathbf{W}), \tag{4.2}$$

where $\delta_{\mathbf{W}^i}$ denotes a point mass located at $\mathbf{W}^i$. In the next step, a sample $\mathbf{W}$ from $\tilde{q}(\mathbf{W})$ (or, equivalently, an approximate sample from $q_{\boldsymbol{\nu}}(\mathbf{W})$) is drawn which can be encoded by the corresponding number $k \in \{1, \ldots, 2^K\}$ using K bits. Using the same random number generator initialized with the same seed as in (4.2), the weights $\mathbf{W}$ can be recovered by sampling 2^K weights $\mathbf{W}^i$ from the prior $p(\mathbf{W})$ and selecting $\mathbf{W}^k$. Since the number of samples 2^K required to obtain a reasonable approximation to $q_{\boldsymbol{\nu}}(\mathbf{W})$ in (4.2) grows exponentially with the number of weights, this sampling based compression scheme is performed for smaller weight blocks such that each weight block can be encoded with K bits.

4.2 Network Pruning

Network pruning methods aim to achieve parameter sparsity by setting a substantial number of DNN weights to zero. Subsequently, the sparsity is exploited to improve resource efficiency of the DNN. On the one hand, there exist *unstructured* pruning approaches that set individual weights, regardless of their location in a weight tensor, to zero. Unstructured pruning approaches are typically less sensitive to accuracy degradation, but they require special sparse tensor data structures that in turn yield practical efficiency improvements only for very high sparsity. On the other hand, *structured* pruning methods aim to set whole weight structures to zero, e.g., by setting *all* weights of a matrix column to zero we would effectively prune an entire neuron. Conceptually, structured pruning is equivalent to removing tensor dimensions such that the reduced tensor remains compatible with highly optimized dense tensor operations.

In this section, we start with the unstructured case which includes many of the earlier approaches and continue with structured pruning that has been the focus of more recent works. Then we review approaches that relate to Bayesian principles before we discuss approaches that prune structures dynamically during forward propagation.

4.2.1 Unstructured Pruning

One of the earliest approaches to reduce the network size is the *optimal brain damage* algorithm of LeCun et al. [141]. Their main finding is that pruning based on weight magnitude is suboptimal, and they propose a pruning scheme based on the increase in loss function. Assuming a pre-trained network, a local second-order Taylor expansion with a diagonal Hessian approximation is employed that allows us to estimate the change in loss function caused by weight pruning without re-evaluating the costly network function. Removing parameters is alternated with retraining the pruned network. In this way, the model size can be reduced substantially without deteriorating its performance. Hassibi and Stork [142] found the diagonal Hessian approximation to be too restrictive, and their *optimal brain surgeon* algorithm uses an approximated full covariance matrix instead. While their method, similar as in [141], prunes weights that cause the least increase in loss function, the remaining weights are simultaneously adapted to compensate for the negative effect of weight pruning. This bypasses the need to alternate several times between pruning and retraining the pruned network.

However, it is not clear whether these approaches scale up to modern DNN architectures since computing the required (diagonal) Hessians is substantially more demanding (if not intractable) for millions of weights. Therefore, many of the more recently proposed techniques still resort to magnitude-based pruning. Han et al. [143] alternate between pruning connections below a certain magnitude threshold and retraining the pruned DNN. The results of this simple strategy are impressive, as the number of parameters in pruned DNNs is an order of magnitude smaller ($9\times$ for AlexNet and $13\times$ for VGG-16) than in the original networks. Hence, this work shows that DNNs are often heavily over-parameterized. In a follow-up paper, Han et al. [144] proposed *deep compression*, which extends the work in [143] by a parameter quantization and parameter sharing step, followed by Huffman coding to exploit the non-uniform weight distribution. This approach yields a reduction in memory footprint by a factor of 35–49 and, consequently, a reduction in energy consumption by a factor of 3–5.

Guo et al. [145] discovered that irreversible pruning decisions limit the achievable sparsity and that it is useful to reincorporate weights pruned in an earlier stage. In addition to each dense weight matrix $\mathbf{W} \in \mathbb{R}^{d_l \times d_{l-1}}$, they maintain a corresponding binary mask matrix $\mathbf{T} \in \{0,1\}^{d_l \times d_{l-1}}$ that determines whether a weight is currently pruned or not. In particular, the actual weights used during forward propagation are obtained as $\mathbf{W} \odot \mathbf{T}$ where $\odot$ denotes element-wise multiplication. Their method alternates between updating the weights $\mathbf{W}$ based on gradient

descent, and updating the weight masks $\mathbf{T}$ by thresholding the real-valued weights according to

$$T_{i,j}^{t+1} = \begin{cases} 0 & \text{if } |w_{i,j}^t| \in [0, \alpha) \\ T_{i,j}^t & \text{if } |w_{i,j}^t| \in [\alpha, \beta) \\ 1 & \text{if } |w_{i,j}^t| \in [\beta, \infty) \end{cases} , \tag{4.3}$$

where α and β are two thresholds and t refers to the iteration number. Most importantly, weight updates are also applied to the currently pruned weights according to $\mathbf{T}$ using the STE, such that pruned weights can reappear in (4.3). This reduces the number of parameters of AlexNet by a factor of 17.7 without deteriorating performance.

4.2.2 Structured Pruning

In [146], a determinantal point process (DPP) is used to find a group of neurons that are diverse and exhibit little redundancy. Conceptually, a DPP for a given ground set $\mathcal{S}$ defines a distribution over subsets $S \subseteq \mathcal{S}$ where subsets containing diverse elements have high probability. They consider $\mathcal{S}$ to be the set of N-dimensional vectors that individual neurons compute over the whole dataset. Their approach samples a diverse set of neurons $S \subseteq \mathcal{S}$ according to the DPP and then prunes the other neurons $\mathcal{S} \setminus S$. To compensate for the negative effect of pruning, the outgoing weights of the remaining neurons after pruning are adapted so as to minimize the activation change of the next layer.

Wen et al. [147] incorporated group lasso regularizers in the objective to obtain different kinds of sparsity in the course of training. They were able to remove filters, channels, and even entire layers in architectures containing shortcut connections. Liu et al. [148] proposed to introduce an ℓ^1-norm regularizer on the scale parameters γ_{bn} of batch normalization and to set $\gamma_{\text{bn}} = 0$ by thresholding. Since each batch normalization parameter γ_{bn} corresponds to a particular channel in the network, this results in channel pruning with minimal changes to existing training pipelines. In [149], the outputs of different structures are scaled with individual trainable scaling factors. By using a sparsity enforcing ℓ^1-norm regularizer on these scaling factors, the outputs of the corresponding structures are driven to zero and can be pruned.

Rather than pruning based on small parameter values, ThiNet [150] is a data-driven approach that prunes channels having the least impact on the subsequent layer. To prune channels in layer l, they propose to sample several activations $x_{i,w,h}^{l+1}$ at randomly selected spatial locations (w, h) and channels i of the following layer, and to greedily prune channels whose removal results in the least increase of squared error over these randomly selected activations. After pruning, they adapt the remaining filters to minimize the squared reconstruction error by minimizing a least squares problem.

Louizos et al. [151] propose to multiply weights with stochastic binary 0-1 gates associated with trainable probability parameters that effectively determine whether a weight should be pruned or not. They formulate an expected loss with respect to the distribution over the stochastic binary gates. By incorporating an expected ℓ^0-norm regularizer over the weights, the probability parameters associated with these gates are encouraged to be close to zero. To enable the use of the reparameterization trick, a continuous relaxation of the binary gates using a modified binary Gumbel-softmax distribution is used [100]. They show that their approach can be used for structured sparsity by associating the stochastic gates to entire structures such as channels. Li and Ji [152] extended this work by using the recently proposed unbiased ARM gradient estimator [153] instead of using the biased Gumbel-softmax approximation.

4.2.3 Bayesian Approaches for Network Pruning

In [74, 89], mean field variational inference is employed to obtain a factorized Gaussian approximation $q_{\boldsymbol{\nu}}(\mathbf{W})$, i.e., instead of learning a deterministic weight w per connection, they train for each connection a weight mean μ_w and weight variance σ_w^2. After training, weights are pruned by thresholding the "signal-to-noise ratio" $|\mu_w/\sigma_w|$.

Some pruning approaches are based on variational dropout [33] which interprets dropout as performing variational inference with specific prior and approximate posterior distributions. Within this framework, the otherwise fixed dropout rates α_{do}^l of Gaussian dropout appear as free parameters that can be optimized to improve a variational lower bound. Molchanov et al. [154] exploited this freedom to optimize individual weight dropout rates α_w such that weights w can be safely pruned if their dropout rate α_w is large. This idea has been extended in [136] by using sparsity enforcing priors and assigning dropout rates to groups of weights that are all connected to the same structure, which in turn allows for structured pruning. Furthermore, they show how their approach can be used to determine an appropriate bit width for each weight by exploiting the well-known connection between Bayesian inference and the MDL principle [87].

4.2.4 Dynamic Network Pruning

So far, we have presented methods that result in a fixed reduced architecture. In the following, we present methods that determine dynamically in the course of forward propagation which structures should be computed or, equivalently, which structures should be pruned. The intuition behind this idea is to vary the time spent for computing predictions based on the difficulty of the given input samples $\mathbf{x}^0$.

Lin et al. [155] proposed to train, in addition to the DNN, a RNN decision network which determines the channels to be computed using reinforcement learning. In each layer, the feature maps are compressed using global pooling and fed into the RNN which aggregates state information over the layers to compute its pruning decisions.

In [156], convolutional layers of a DNN are extended by a parallel low-cost convolution whose output after the ReLU function is used to scale the outputs of the potentially high-cost convolution. Due to the ReLU function, several outputs of the low-cost convolution will be exactly zero such that the computation of the corresponding output of the high-cost convolution can be omitted. For the low-cost convolution, they propose to use weight tensors $\mathbf{W} \in \mathbb{R}^{1 \times 1 \times d_{l-1} \times d_l}$ and $\mathbf{W} \in \mathbb{R}^{K \times K \times d_{l-1} \times 1}$. However, practical speed-ups are only reported for the $K \times K$ convolution where all channels at a given spatial location might get set to zero.

In a similar approach proposed by Gao at el. [157], the spatial dimensions of a feature map are reduced by global average pooling to a vector $\mathbf{u} \in \mathbb{R}^{d_{l-1}}$ which is linearly transformed to $\mathbf{v} \in \mathbb{R}^{d_l}$ using a single low-cost fully connected layer. To obtain a sparse vector $\mathbf{s} \in \mathbb{R}^{d_l}$, $\mathbf{v}$ is fed into the ReLU function, followed by a k-winner-takes-all function that sets all entries of a vector to zero that are not among the k largest entries in absolute value. By multiplying $\mathbf{s}$ in a channel-wise manner to the output of a high-cost convolution, at least $d_l - k$ channels will be zero and need not be computed. The number of channels k is derived from a predefined minimal pruning ratio hyperparameter.

4.3 Structural Efficiency in Deep Neural Networks

In this section, we review strategies that establish certain structural properties in DNNs to improve computational efficiency. Each of the proposed subcategories in this section follows rather different principles and the individual techniques might not be mutually exclusive.

4.3.1 Weight Sharing

Another technique to reduce the model size is weight sharing. Before we start, we note that weight sharing and quantization methods (see Section 4.1) are closely related: Quantization methods often have an inherent weight sharing property since the number of possible quantization values is often much smaller than the number of weights. However, the purpose of a method is typically different depending on which category it belongs to. On the one hand, the focus of weight quantization methods typically lies on the employed numerical formats. The purpose of these formats is to reduce the storage per weight and to facilitate more efficient computations. Furthermore, the number of distinct weight values is typically rather small and fixed, and the particular weight values are often constrained or even fixed in advance. On the other hand, the purpose of weight sharing is to reduce the memory by reducing the overall number of distinct weight values. For these methods, the particular weight values typically remain unconstrained. Note that some methods cannot be clearly attributed to either category, e.g., in deep compression [144] weight sharing and quantization are in part used synonymously.

In [76], a hashing function is used to randomly group network connections into "buckets", where the connections in each bucket share the same weight value. The advantage of their approach is that weight assignments need not be stored explicitly since they are given implicitly by the hashing function. The authors show a memory footprint reduction by a factor of 10 while keeping the predictive performance essentially unaffected.

Ullrich et al. [158] extended the soft weight sharing approach proposed in [75] to achieve both weight sharing and sparsity. The idea is to select a Gaussian mixture model prior over the weights and to train both the weights as well as the parameters of the mixture components. During training, the mixture components collapse to point measures and each weight gets attracted by a certain weight component. After training, weight sharing is obtained by assigning each weight to the mean of the component that best explains it, and weight pruning is obtained by assigning a relatively high mixture mass to a component with a fixed mean at zero.

Our work [159] presented in Chapter 6 utilizes weight sharing to reduce the memory footprint of a large Bayesian ensemble of DNNs. The weight sharing is enforced by introducing a DP prior over the weight prior distribution. We propose a sampling based inference scheme by alternately sampling weight assignments using Gibbs sampling and sampling weights using HMC [69, 70]. By using the same weight assignments for multiple weight samples, the memory overhead for the weight assignments becomes negligible and the total memory footprint of an ensemble is reduced.

4.3.2 Knowledge Distillation

Knowledge distillation is a method where the knowledge contained in a large *teacher* model is transferred to a smaller *student* model. In the first step, a large teacher model is obtained with conventional training methods on the given training data. Subsequently, the smaller student model is trained on data where the ground truth labels have been replaced by the soft labels obtained from the output of the teacher model, e.g., from the softmax output of a DNN. It has been shown that this substantially increases the accuracy of the student model compared to directly training on the given training data.

This general scheme is model agnostic, and early works applied knowledge distillation to compress *ensembles* of shallow neural networks [160] and other types of classifiers [161] into a single neural network. Zeng and Martinez [160] have shown that training on soft labels obtained from the teacher results in higher accuracy than training on the actual hard predictions. The work of Bucila [161] emphasizes the ability to train the student on unlabeled data to further reduce the accuracy gap between student and teacher. In addition, they presented a method to generate new synthetic inputs from the given training set, which might be useful if additional unlabeled data is limited or not available. They showed that the accuracy of the student can

improve substantially when trained on these synthetically generated inputs.

Ba and Caruana [162] applied these ideas to investigate the importance of depth in a DNN. They trained shallower (but not necessarily smaller) neural networks by mimicking the output activations $\mathbf{a}^L$ produced by a teacher DNN before applying the softmax function. The resulting shallow models perform similar as their deeper counterparts which was not achievable by training the shallow model on the ground truth targets directly. Therefore, the authors conclude that shallower models are as expressive as deeper models but they are more difficult to train.

The work of Li et al. [163] and Hinton et al. [164] applied knowledge distillation with the main focus on reducing model complexity of a large teacher DNN. In [164], it is proposed to obtain the soft labels $\hat{\mathbf{y}}$ from the teacher by scaling the output activations with a temperature $\tau > 0$ as

$$\hat{y}_i = \frac{\exp(a_i^L/\tau)}{\sum_j \exp(a_j^L/\tau)}. \tag{4.4}$$

For $\tau > 1$, the labels tend to become more uniform which has been reported to facilitate training. Furthermore, they propose to utilize the ground truth labels by minimizing a weighted average of the traditional cross-entropy loss based on the ground truth labels $\mathbf{y}$ and the knowledge distillation loss based on the soft targets $\hat{\mathbf{y}}$ in (4.4). Noteworthy, it was the work of Hinton et al. [164] that coined the term *knowledge distillation*.

FitNets [165] extend these ideas by also transferring knowledge from intermediate layers. They select an intermediate layer from the teacher DNN as the *hint layer* which they try to mimic in an intermediate *guide layer* of the student DNN. Since the hint layer and the guide layer are generally of different size, they introduce a regressor that predicts the hint layer from the guide layer. This ensures that the guide layer contains the same information as the hint layer. The proposed procedure operates in two stages. In the first stage, the student is trained up to the guide layer by minimizing the discrepancy between guide and hint layer. In the second stage, the whole student DNN is trained using conventional knowledge distillation as in [164].

Kim et al. [166] argue that matching the raw features of certain intermediate layers as in [165] is suboptimal since it is difficult to compare individual layers of different DNNs. Therefore, they propose a method to match more understandable *factors* extracted from the intermediate layers of the student and the teacher DNNs. Starting from a pre-trained teacher DNN, they first train an autoencoder which they call *paraphraser* to extract understandable factors from a selected intermediate layer of the teacher DNN. The student DNN is extended by a regressor which they call *translator* whose purpose is to predict the paraphraser factors from the features of a selected intermediate layer. The student DNN is then trained to simultaneously minimize the cross-entropy loss on the ground truth labels and the difference between paraphraser and translator output. They employ the paraphraser and the translator after the last convolutional layer in their DNNs.

In the context of quantization, knowledge distillation has been used to reduce the accuracy gap between real-valued DNNs and quantized DNNs [167, 168]. In particular, a real-valued teacher DNN is used to improve the accuracy of a quantized teacher DNN. Mishra and Marr [167] showed improved results using three different modes of knowledge distillation training, including a mode where the student and the teacher are trained simultaneously from scratch.

Phuong and Lampert [169] transferred knowledge between different parts of the *same* model. They employ multi-exit architectures which provide anytime predictions after certain intermediate layers; therefore, allowing for a trade-off between accuracy and prediction latency at run-time. The knowledge from the (most accurate) final layer is transferred to the earlier exits to improve their accuracy. Furthermore, they show that the earlier layers can be trained with unlabeled data in a semi-supervised setting.

In a Bayesian context, Korattikara et al. [170] applied knowledge distillation to condense a large ensemble of DNNs, for instance, obtained by sampling from the posterior distribution $p(\mathbf{W}|\mathcal{D})$. In this way, the predictive distribution (3.52) obtained by averaging the outputs of

the individual models can be transferred to a single DNN. Their method trains a single student DNN using the outputs of teacher DNNs that are generated on the fly using SGLD [102].

4.3.3 Special Matrix Structures

In this section, we review approaches that aim at reducing the model size by employing efficient matrix representations. There exist several methods using low-rank decompositions which represent a large matrix (or a large tensor) using only a fraction of the parameters. In most cases, the implicitly represented matrix is never computed explicitly such that also a computational speed-up is achieved. Furthermore, there exist approaches using special matrices that are specified by only few parameters and whose structure allows for extremely efficient matrix multiplications.

Denil et al. [171] proposed a method that is motivated by training only a subset of the weights and predicting the values of the other weights from this subset. In particular, they represent weight matrices $\mathbf{W} \in \mathbb{R}^{d_l \times d_{l-1}}$ using a low-rank approximation $\mathbf{UV}$ with $\mathbf{U} \in \mathbb{R}^{d_l \times d'}$, $\mathbf{V} \in \mathbb{R}^{d' \times d_{l-1}}$, and $d' < \min\{d_{l-1}, d_l\}$ to reduce the number of parameters. Instead of learning both factors $\mathbf{U}$ and $\mathbf{V}$, prior knowledge, such as smoothness of pixel intensities in an image, is incorporated to compute a fixed $\mathbf{V}$ using kernel techniques or autoencoders, and only the factor $\mathbf{U}$ is learned.

In [172], the tensor train matrix format is employed to substantially reduce the number of parameters required to represent large weight matrices of fully connected layers. Their approach enables the training of very large fully connected layers with relatively few parameters, and they achieve improved performance compared to simple low-rank approximations.

Denton et al. [173] propose specific low-rank approximations and clustering techniques for individual layers of pre-trained CNNs to reduce both memory footprint and computational overhead. Their approach yields substantial improvements for both the computational bottleneck in the convolutional layers and the memory bottleneck in the fully connected layers. By fine-tuning after applying their approximations, the performance degradation is kept at a decent level. Jaderberg et al. [174] propose two different methods to approximate pre-trained CNN filters as combinations of rank-1 basis filters to speed up computation. The rank-1 basis filters are obtained either by minimizing a reconstruction error of the original filters or by minimizing a reconstruction error of the outputs of the convolutional layers. Lebedev et al. [175] approximate the convolution tensor using the canonical polyadic (CP) decomposition—a generalization of low-rank matrix decompositions to tensors—using nonlinear least squares. Subsequently, the convolution using this low-rank approximation is performed by four consecutive convolutions, each with a smaller filter, to reduce the computation time substantially.

In [176], the weight matrices of fully connected layers are restricted to circulant matrices $\mathbf{W} \in \mathbb{R}^{d \times d}$, which are fully specified by only d parameters. While this dramatically reduces the memory footprint of fully connected layers, circulant matrices also facilitate faster computation as matrix-vector multiplication can be efficiently computed using the fast Fourier transform. In a similar vein, Yang et al. [177] reparameterize matrices $\mathbf{W} \in \mathbb{R}^{d \times d}$ of fully connected layers using the Fastfood transform as $\mathbf{W}=\mathbf{SHG\Pi HB}$, where $\mathbf{S}$, $\mathbf{G}$, and $\mathbf{B}$ are diagonal matrices, $\mathbf{\Pi}$ is a random permutation matrix, and $\mathbf{H}$ is the Walsh-Hadamard matrix. This reparameterization requires only a total of $4d$ parameters, and similar as in [176], the fast Hadamard transform enables an efficient computation of matrix-vector products.

4.3.4 Manual Architecture Design

Instead of modifying existing architectures to make them more efficient, manual architecture design is concerned with the development of new architectures that are inherently resource-efficient. Over the past years, several design principles and building blocks for DNN architectures have

emerged that exhibit favorable computational properties and sometimes also improve performance.

CNN architectures are typically designed to have a transition from convolutional layers to fully connected layers. At this transition, activations at all spatial locations of each channel are typically used as individual input features for the following fully connected layer. Since the number of these features is typically large, there is a memory bottleneck for storing the parameters of the weight matrix especially in the first fully connected layer.

Lin et al. [38] introduced two concepts that have been widely adopted by subsequent works. The first one, *global average pooling*, largely solves the above-mentioned memory issue at the transition to fully connected layers. Global average pooling reduces the spatial dimensions of each channel into a single feature by averaging over all values within a channel. This reduces the number of features at the transition drastically, and, by having the same number of channels as there are classes, it can also be used to completely get rid of fully connected layers. Second, they used 1×1 convolutions with weight kernels $\mathbf{W} \in \mathbb{R}^{1 \times 1 \times d_{l-1} \times d_l}$ which can be seen as performing the operation of a fully connected layer over each spatial location across all channels.

These 1×1 convolutions have been adopted by several popular architectures [23–25] and, due to their favorable computational properties compared to convolutions that take a spatial neighborhood into account, later have also been exploited to improve computational efficiency. For instance, InceptionNet [25] proposed to split standard $K \times K$ convolutions into two cheaper convolutions: (i) a 1×1 convolution to reduce the number of channels such that (ii) a subsequent $K \times K$ convolution is performed faster. Similar ideas are used in *SqueezeNet* [178] which employs 1×1 convolutions to reduce the number of input channels of subsequent parallel 1×1 and 3×3 convolutions. In addition, SqueezeNet uses the global average pooling output of per-class channels directly as input to the softmax in order to avoid fully connected layers that typically consume the most memory. Furthermore, by using deep compression [144] (see Section 4.2.1), the memory footprint was reduced to less than 0.5MB.

Szegedy et al. [39] extended the InceptionNet architecture by spatially separable convolutions to reduce the computational complexity, i.e., a $K \times K$ convolution is split into a $K \times 1$ convolution followed by a $1 \times K$ convolution. In *MobileNet* [179] depthwise separable convolutions are used to split a standard convolution in another way: (i) a depthwise convolution and (ii) a 1×1 convolution. The depthwise convolution applies a $K \times K$ filter to each channel separately without taking the other channels into account whereas the 1×1 convolution then aggregates information across channels. Although these two cheaper convolutions together are less expressive than a standard convolution, they can be used to trade off a small loss in prediction accuracy with a drastic reduction in computational overhead and memory requirements.

Sandler et al. [45] extended these ideas in their *MobileNetV2* to an architecture with residual connections. A typical residual block with bottleneck structure in ResNet [23] contains a 1×1 bottleneck convolution to reduce the number of channels, followed by a 3×3 convolution, followed by another 1×1 convolution to restore the original number of channels again. Contrary to that building block, MobileNetV2 introduces an *inverted* bottleneck structure where the shortcut path contains the bottleneck and the residual path performs computations in a high-dimensional space. In particular, the residual path performs a 1×1 convolution to *increase* the number of channels, followed by a cheap *depthwise* 3×3 convolution, followed by another 1×1 convolution to reduce the number of channels again. They show that their inverted structure is more memory efficient since the shortcut path, which needs to be kept in memory during computation of the residual path, is considerably smaller. Furthermore, they show improved performance compared to the standard bottleneck structure.

While it was more of a technical detail rather than a contribution on its own, AlexNet [6] used *grouped convolutions* with two groups to facilitate model parallelism for training on two GPUs with relatively low memory capacity. Instead of computing a convolution using a weight tensor $\mathbf{W} \in \mathbb{R}^{K \times K \times g d_{l-1} \times g d_l}$, a grouped convolution splits the input into g groups of d_{l-1} channels that are independently processed using weight tensors $\mathbf{W}_g \in \mathbb{R}^{K \times K \times d_{l-1} \times d_l}$. The outputs of these g

convolutions are then stacked again such that the same number of input and output channels are maintained while considerably reducing the computational overhead and memory footprint.

Although this reduces the expressiveness of the convolutional layer since there is no interaction between the different groups, Xie et al. [41] used grouped convolutions to enlarge the number of channels of a ResNet model which resulted in accuracy gains while keeping the computational complexity of the original ResNet model approximately the same. Zhang et al. [180] introduced a ResNet-inspired architecture called *ShuffleNet* which employs 1×1 *grouped* convolutions since 1×1 convolutions have been identified as computational bottlenecks in previous works, e.g., see [179]. To combine the computational efficiency of grouped convolutions with the expressiveness of a full convolution, ShuffleNet incorporates *channel shuffle* operations after grouped convolutions to partly recover the interaction between different groups.

4.3.5 Neural Architecture Search (NAS)

NAS is a recently emerging field concerned with the automatic discovery of good DNN architectures. This is achieved by designing a discrete space of possible architectures in which we subsequently search for an architecture that optimizes some objective—typically the validation error. By incorporating a measure of resource efficiency into this objective, this technique has recently attracted attention for the automatic discovery of resource-efficient architectures.

The task is very challenging: On the one hand, evaluating the validation error is time-consuming as it requires a full training run and typically only results in a noisy estimate thereof. On the other hand, the space of architectures is typically of exponential size in the number of layers. Hence, the space of architectures needs to be carefully designed in order to facilitate an efficient search within that space.

The influential work of Zoph and Le [43] introduced a scheme to encode DNN architectures of arbitrary depth as sequences of tokens which can be sampled from a controller RNN. This controller RNN is trained with reinforcement learning to generate well performing architectures using the validation error on a held-out validation set as a reward signal. However, the training effort is enormous since more than 10,000 training runs are required to achieve state-of-the-art performance on Cifar-10. This would be impractical on larger datasets such as ImageNet which was partly solved by subsequent NAS approaches, e.g., in [181]. In this review, we highlight methods that also consider resource efficiency constraints for NAS.

In MnasNet [44], a RNN controller is trained by also considering the latency of the sampled DNN architecture measured on a real mobile device. They achieve performance improvements under predefined latency constraints on a specific device. To run MnasNet on the large-scale ImageNet and COCO datasets [182], their algorithm is run on a *proxy task* by only training for five epochs, and only the most promising DNN architectures were trained using more epochs.

Wang et al. [183] determined the individual bit widths of mixed-precision quantization using a similar reinforcement learning framework. Their controller DNN generates for each layer two bit widths, one for the weights and one for the activations. A pre-trained full-precision DNN is then quantized using these bit widths and fine-tuned for one epoch to obtain a reward signal that is subsequently used to update the controller. Their method incorporates hardware-specific constraints, such as latency and energy consumption, that must be met by the controller.

Instead of generating architectures using a controller, ProxylessNAS [184] uses a heavily over-parameterized model where each layer contains several parallel paths, each computing a different architectural block with its individual parameters. For each layer, probability parameters for selecting a particular architectural block are introduced which are trained via backpropagation using the STE. After training, the most probable path determines the selected architecture. To favor resource-efficient architectures, a latency model is build using measurements done on a specific real device whose predicted latencies are used as a differentiable regularizer in the cost function. In their experiments, they show that different target devices prefer individual DNN

architectures to obtain a low latency.

Instead of using a different path for different operations in each layer, single-path NAS [185] combines all operations in a single *shared weight superblock* such that each operation uses a subset of this superblock. A weight-magnitude-based decision using trainable threshold parameters determines which operation should be performed, allowing for gradient-based training of both the weight parameters and the architecture. Again, the STE is employed to backpropagate through the threshold function.

Wu et al. [186] performed mixed-precision quantization using similar NAS concepts to those used in [187] and [184]. They introduce gates for every layer that determine the number of bits used for quantization, and they perform continuous stochastic optimization of probability parameters associated with each of these gates.

Liu et al. [188] have replicated several experiments of pruning approaches (see Section 4.2) and they observed that the typical workflow of training, pruning, and fine-tuning is often not necessary and only the discovered sparsity structure is important. In particular, they show for several pruning approaches that randomly initializing the weights after pruning and training the pruned structure from scratch results in most cases in a similar performance as performing fine-tuning after pruning. They conclude that network pruning can also be seen as a paradigm for architecture search.

Tan and Le [42] recently proposed EfficientNet which employs NAS for finding a resource-efficient architecture as a key component. In the first step, they perform NAS to discover a small resource-efficient model which is much cheaper than searching for a large model directly. In the next step, the discovered model is enlarged by a principled compound scaling approach which simultaneously increases the number of layers, the number of channels, and the spatial resolution. Although this approach is not targeting resource efficiency on its own, EfficientNet achieves state-of-the-art performance on ImageNet using a relatively small model.

In Chapter 7, we show that ideas from NAS for DNNs can be successfully transferred to a completely different model class, namely BNs. More specifically, we employ ideas from [187] and particularly from [184] to train the parameters of the BN and its TAN structure jointly using the same objective with gradient-based optimization techniques [189]. In [134], we also show that this objective can be extended to a model size aware objective. This allows us to trade off between accuracy and model size in a principled manner. These promising results suggest that NAS techniques from the deep learning community might be applicable to a broader class of models whose underlying structure is specified by some kind of graph.

5 Learning Discrete-Valued Neural Networks Using Weight Distributions

Quantization is concerned with mapping real-valued quantities to a discrete set. In recent years, much research effort has been spent on quantization techniques for DNNs to reduce their model complexity. This chapter is devoted to DNNs with both quantized weights and activations.

By assuming that the weights of a DNN are quantized to a discrete set, the task of training becomes a combinatorial optimization problem. However, there seems to be little hope that standard combinatorial optimization techniques will produce meaningful results. On the one hand, the number of discrete weights in modern DNNs is huge such that enumerating all weight combinations is not an option. On the other hand, successful combinatorial optimization techniques often traverse the solution space by following some carefully designed heuristic. Designing such a heuristic is difficult since DNNs are generally difficult to interpret. Considering that a large portion of the success of deep learning can be attributed to gradient-based learning techniques, it is not surprising that most quantization techniques also rely on gradient-based learning.

Most of the quantization techniques discussed in Section 4.1 employ the STE to approximate the zero gradient of piecewise constant quantization functions in a computation graph. However, the STE—although often providing convincing empirical results—is yet to be better understood theoretically. From a theoretical point of view, it is rather unsatisfactory that the gradient of a function known to be exactly zero is "approximated" by something non-zero.[19]

In this chapter, we introduce a method to train discrete-valued DNNs by continuous optimization of a well-defined objective. Notably, our method does not rely on the STE. For this purpose, we introduce a discrete distribution $q_{\boldsymbol{\nu}}(\mathbf{W})$ over the weights that is governed by continuous distribution parameters $\boldsymbol{\nu}$. Given some loss $\mathcal{L}(\mathbf{W}; \mathcal{D})$, we define a new loss over $\boldsymbol{\nu}$ as an expectation of $\mathcal{L}(\mathbf{W}; \mathcal{D})$ with respect to $q_{\boldsymbol{\nu}}(\mathbf{W})$. The resulting loss is differentiable in $\boldsymbol{\nu}$ and can be optimized with conventional gradient-based learning techniques. Importantly, this approach yields a differentiable loss irrespective of whether the employed activation function h^l is differentiable. We utilize this to also train DNNs using the sign activation function. After the distribution $q_{\boldsymbol{\nu}}(\mathbf{W})$ has been trained, we select either its most probable weights or some generated weight samples as our efficient discrete-valued DNNs. The whole procedure is outlined in Figure 5.1(a). Our method was originally inspired by the Bayesian variational inference framework for DNNs, and we show that the mean field variational inference framework is closely related to our method.

Compared to the most relevant previous works of Shayer et al. [137] and Peters and Welling [138], our work provides several extensions. These methods use a parameterization of $q_{\boldsymbol{\nu}}(\mathbf{W})$ that is tailored to binary and ternary weights and does not easily generalize to more than three weights. The same holds true for their method to initialize $q_{\boldsymbol{\nu}}(\mathbf{W})$ which is coupled to their parameterization. We introduce simpler parameterization and initialization schemes that naturally generalize to arbitrary discrete weights. Furthermore, we introduce a distribution-aware approximation for propagating distributions through max pooling. This is in contrast to the method of [138] where the distribution is only indirectly taken into account. In contrast to

[19] We stick to the vast literature that uses the term "approximate" in this context, but we want to highlight that this term is somewhat improperly used. In most meaningful scenarios, one typically uses approximation techniques to estimate values that are known to exist but are, for some reason, difficult to compute exactly.

several other works that require real-valued weights in the input and/or output layers [118, 130, 137], we employ discrete weights in *all* layers and still achieve state-of-the-art performance.

We empirically show the effectiveness of our method in an extensive experimental evaluation. Our method achieves state-of-the-art performance on several datasets. We show that using more discrete weight values typically results in higher prediction accuracy. Consequently, our method provides an effective means of trading off between computational requirements and accuracy. We conducted an extensive ablation study to gain insights into individual components of the proposed method. In particular, our experiments show that our parameterization facilitates training and results in higher prediction accuracy. We show that it is important for the max pooling approximation to take distributional properties more directly into account. We also conducted ensemble experiments by averaging the predictions of several models sampled from $q_{\boldsymbol{\nu}}(\mathbf{W})$. We find that only few samples are required to achieve a higher accuracy than the most probable model; therefore, providing us with another tool to trade off between computational requirements and accuracy.

This chapter is largely based on our paper "Training Discrete-Valued Neural Networks with Sign Activations Using Weight Distributions" that was presented at the ECML PKDD conference in 2019 [139]. The chapter extends the ECML paper mainly by additional experiments and an improved experimental setup. Each experiment is now conducted ten times (instead of five times), each using a different pre-trained model (instead of using the same pre-trained model five times). In our most notable additional experiments, we compare different max pooling approximations, we perform experiments using different training methods (e.g., using the probabilistic forward pass), and we perform model averaging experiments. Interestingly, we find that pre-trained models with ReLU activation yield substantially higher accuracies than pre-trained models using the tanh function. This is somewhat surprising since the discrete-valued DNNs employ the sign activation function which is closer in shape to the tanh function.

This chapter is outlined as follows. Section 5.1 introduces discrete-valued DNNs and the probabilistic loss with corresponding optimization techniques. Details about the model and individual building blocks are provided in Section 5.2. Our extensive experimental evaluation is presented in Section 5.3 and we discuss our findings in Section 5.4.

5.1 Training with Discrete Weight Distributions

We begin by defining the main subject of this chapter, namely *discrete-valued DNNs*. A discrete-valued DNN is a DNN where the weights, the activations[20], or both are from a discrete set $\mathbb{D}$. For the activations, we consider binary values $\mathbb{D}_2 = \{-1, 1\}$ which are conveniently obtained by the sign activation function. For the weights, let $\mathbb{D}_Q = \{v_1, \ldots, v_Q\}$ be a discrete set of weight values with $v_1 < \ldots < v_Q$. We consider discrete weights with $Q \in \{3, 4, 5\}$, i.e., ternary, quaternary, and quinary weights. The choice of the particular weights v_i is arbitrary and we restrict ourselves to evenly spaced weights with constant $\Delta_v = v_{i+1} - v_i$ that are symmetric around zero. In particular, we have $\mathbb{D}_3 = \{-1, 0, 1\}$, $\mathbb{D}_4 = \{-1, -\frac{1}{3}, \frac{1}{3}, 1\}$, and $\mathbb{D}_5 = \{-1, -\frac{1}{2}, 0, \frac{1}{2}, 1\}$. Note that the scale of the discrete weights is irrelevant as either we use the sign function that stays unaffected or batch normalization [34] compensates for any change in scale.

5.1.1 The Probabilistic Loss

To avoid solving an intractable combinatorial optimization problem over the discrete weights $w \in \mathbb{D}_Q$, we first introduce a discrete weight distribution $q_{\boldsymbol{\nu}}(\mathbf{W})$ governed by a set of continuous parameters $\boldsymbol{\nu}$. We adopt the common mean field assumption such that individual weights are

[20] We stick to the vast literature where *activation quantization* refers to quantization of the layer outputs $\mathbf{x}^l = h^l(\mathbf{a}^l)$ *after* the activation function h^l has been applied.

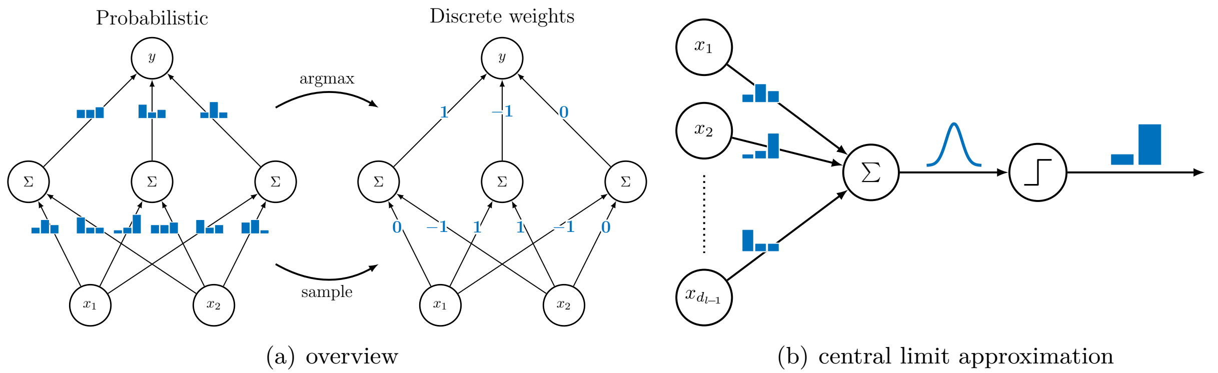


Figure 5.1: (a) Overview of our method. We train distributions over discrete weights (left). After training, a discrete-valued DNN (right) is obtained by selecting the most probable weights or sampling from these distributions. (b) The expectation in (5.1) *is approximated by invoking a central limit approximation at the neurons and propagating the resulting Gaussians through the sign activations. The binary activations are then either sampled using the Gumbel-softmax reparameterization or further propagated through the DNN using the probabilistic forward pass.*

independent under $q_{\boldsymbol{\nu}}(\mathbf{W})$. This implies that $q_{\boldsymbol{\nu}}(\mathbf{W})$ factorizes into a product of factors $q_w(w)$, one for each weight $w \in \mathbf{W}$. Each of these factors is a pmf over Q values governed by variational parameters $\boldsymbol{\nu}_w$. We elaborate more on the particular parameterization of the pmf over discrete weights in Section 5.2.4.

Let $\mathcal{L}(\mathbf{W}, \mathcal{D})$ be an arbitrary loss function defined over continuous weights $\mathbf{W}$. The discrete weight distribution $q_{\boldsymbol{\nu}}(\mathbf{W})$ allows us to formulate a probabilistic loss as

$$\mathcal{L}_{\text{prob}}(\boldsymbol{\nu}; \mathcal{D}) = \mathbb{E}_{\mathbf{W} \sim q_{\boldsymbol{\nu}}(\mathbf{W})}[\mathcal{L}(\mathbf{W}; \mathcal{D})] + \lambda \mathcal{R}(\boldsymbol{\nu}), \tag{5.1}$$

where $\mathcal{R}(\boldsymbol{\nu})$ is a differentiable regularizer over the distribution parameters $\boldsymbol{\nu}$. This loss exhibits certain favorable properties. Most importantly, $\mathcal{L}_{\text{prob}}$ is differentiable with respect to the distribution parameters $\boldsymbol{\nu}$ such that we can (in principle) perform gradient-based learning. To see why this holds, note that the expectation in (5.1) is essentially a sum over exponentially many terms

$$\mathbb{E}_{\mathbf{W} \sim q_{\boldsymbol{\nu}}(\mathbf{W})}[\mathcal{L}(\mathbf{W}; \mathcal{D})] = \sum_{\mathbf{W} \in \mathbb{D}^{|\mathbf{W}|}} q_{\boldsymbol{\nu}}(\mathbf{W}) \mathcal{L}(\mathbf{W}; \mathcal{D}). \tag{5.2}$$

In (5.2), the weights $\mathbf{W}$ do not appear as free variables such that the term $\mathcal{L}(\mathbf{W}; \mathcal{D})$ can be regarded as a constant. Therefore, the probabilistic loss $\mathcal{L}_{\text{prob}}$ can be seen as a weighted sum over exponentially many terms $q_{\boldsymbol{\nu}}(\mathbf{W})$ where the (constant) weights are specified by $\mathcal{L}(\mathbf{W}; \mathcal{D})$. Consequently, $\mathcal{L}_{\text{prob}}$ is differentiable whenever $q_{\boldsymbol{\nu}}(\mathbf{W})$ is differentiable. Note that this argument neither requires any explicit assumptions on $\mathcal{L}(\mathbf{W}; \mathcal{D})$ nor on the structure of the DNN. The argument holds regardless of the selected activation function h^l and, in particular, the loss function remains differentiable even for the sign activation function. In the remainder of this chapter, we restrict ourselves to the probabilistic loss (5.1) for $\mathcal{L}(\mathbf{W}; \mathcal{D})$ being the cross-entropy loss (2.8).

The probabilistic loss $\mathcal{L}_{\text{prob}}$ exhibits another favorable property if we only consider the expectation (5.2) and ignore the regularizer $\mathcal{R}(\boldsymbol{\nu})$ for the moment. An optimal distribution $q^*_{\boldsymbol{\nu}}(\mathbf{W})$ of (5.2) is obtained by assigning the whole mass to some discrete weights $\mathbf{W}^*$ that maximize $\mathcal{L}(\mathbf{W}, \mathcal{D})$. This can be easily seen since moving any positive mass from an optimal solution $\mathbf{W}^*$ of $\mathcal{L}(\mathbf{W}; \mathcal{D})$ to any suboptimal $\mathbf{W}$ increases (5.2). This also implies that the mean field assumption is sufficient in the sense that an optimal solution of (5.2) can be obtained. However, in practice, the regularizer $\mathcal{R}(\boldsymbol{\nu})$ of $\mathcal{L}_{\text{prob}}$ is still required for similar reasons as it is necessary for

continuous DNNs. As we will see, in our framework an entropy-increasing regularizer is useful to obtain a better Gaussian approximation when the central limit theorem is invoked.

When the optimization procedure finishes and a suitable distribution $q_{\boldsymbol{\nu}}(\mathbf{W})$ has been obtained, it remains to select a proper discrete-valued DNN. We propose to either take the most probable weights $\operatorname{argmax}_{\mathbf{W}} q_{\boldsymbol{\nu}}(\mathbf{W})$ or to generate samples $\mathbf{W} \sim q_{\boldsymbol{\nu}}(\mathbf{W})$. Note that whenever we mention *discrete weight training*, we actually refer to the training of the discrete distribution $q_{\boldsymbol{\nu}}(\mathbf{W})$ from which we subsequently infer the discrete weights. Before we discuss how to optimize (5.1) using gradient-based techniques, we briefly highlight some similarities between our method and Bayesian DNNs using variational inference as discussed in Section 3.4.

5.1.2 Relation to Variational Inference

The presented work is closely related to the Bayesian variational inference framework. Recall from Chapter 3 that, for a Bayesian treatment of DNNs, we assume a prior distribution $p(\mathbf{W})$ over the weights and interpret the softmax output of the DNN as likelihood $p(\mathcal{D}\,|\,\mathbf{W})$ to obtain a posterior $p(\mathbf{W}\,|\,\mathcal{D}) \propto p(\mathcal{D}\,|\,\mathbf{W})p(\mathbf{W})$ over the weights. As the induced posterior $p(\mathbf{W}|\mathcal{D})$ is generally intractable, the aim of variational inference is to approximate it by a simpler distribution $q_{\boldsymbol{\nu}}(\mathbf{W})$ by minimizing the negative evidence lower bound

$$\mathcal{L}_{\text{elbo}}(\boldsymbol{\nu};\mathcal{D}) = -\mathbb{E}_{\mathbf{W}\sim q_{\boldsymbol{\nu}}(\mathbf{W})}[\log p(\mathcal{D}\,|\,\mathbf{W})] + D_{\text{KL}}(q_{\boldsymbol{\nu}}(\mathbf{W})\,||\,p(\mathbf{W})). \tag{5.3}$$

Equation (5.3) is proportional to (5.1) for $\mathcal{L}(\mathbf{W};\mathcal{D})$ being the cross-entropy loss (2.8), $\mathcal{R}(\boldsymbol{\nu})$ being the KL divergence, and $\lambda = 1/N$. The main difference of our work to variational inference is our motivation to use distributions in order to obtain a gradient-based learning scheme for discrete-valued DNNs with discrete activation functions. Variational inference is typically used to approximate expectations over the posterior $p(\mathbf{W}\,|\,\mathcal{D})$ and to obtain well calibrated uncertainty estimates for DNN predictions.

5.1.3 Optimizing the Probabilistic Loss

We assume that the regularizer $\mathcal{R}(\boldsymbol{\nu})$ is given in closed form and that we can evaluate its gradient using backpropagation. It remains to compute the gradient of the intractable expected loss (5.2). To do so, we employ two different techniques for variational inference in Bayesian DNNs, namely the local reparameterization trick and the probabilistic forward pass. Note that both of these methods have already been discussed in detail; see Section 3.4.4 for the (local) reparameterization trick and Sections 3.3.2 and 3.4.1 for the probabilistic forward pass. In the following, we briefly recap the basic ideas of these methods and highlight the necessary adaptations required for discrete-valued DNNs. Figure 5.1(b) provides an overview of the basic concepts of both methods.

The first method is based on the *local reparameterization trick* and computes Monte Carlo estimates of the gradient in order to perform SGD. Let $\mathbf{x}^{l-1}$ be the deterministic inputs of layer l and $q_{\boldsymbol{\nu}}(\mathbf{W}^l)$ be a distribution over the weights in layer l. By appealing to the central limit theorem, the induced marginal distributions over the activations $q(a_i^l)$ can be approximated by a Gaussian $\mathcal{N}(\mu_{a_i^l}, \sigma^2_{a_i^l})$ where

$$\mu_{a_i^l} = \sum_k \mathbb{E}[w_{i,k}^l]\,x_k^{l-1} \qquad \text{and} \qquad \sigma^2_{a_i^l} = \sum_k \mathbb{V}[w_{i,k}^l]\,(x_k^{l-1})^2. \tag{5.4}$$

Most importantly, the Gaussian approximation is justified even for discrete weight distributions $q_{\boldsymbol{\nu}}(\mathbf{W})$. The local reparameterization trick is typically applied to sample from the induced activation marginals with parameters (5.4). This is, however, not applicable in our case since we

subsequently apply the sign activation function which prevents the gradient flow during back-propagation. Therefore, we propagate the Gaussian approximation through the sign activation function and apply the reparameterization trick afterwards. The resulting binary distribution after the sign function is given by

$$q(X_i^l = 1) = \Phi(\mu_{a_i^l} / \sigma_{a_i^l}), \tag{5.5}$$

where Φ denotes the cdf of a standard normal distribution. The binary distributions $q(\mathbf{x}^l)$ vary smoothly as a function of the parameters (5.4) and, therefore, backpropagation remains possible.

Since the reparameterization trick is restricted to continuous distributions, we cannot directly apply it to sample from $q(\mathbf{x}^l)$. Therefore, we apply the Gumbel-softmax approximation [100, 101] to sample from $q(\mathbf{x}^l)$ (see Section 3.4.5). Given the binary distribution (5.5), two samples $\varepsilon_1, \varepsilon_2 \sim \text{Gumbel}(0,1)$ generated according to (3.109), and the Gumbel-softmax temperature τ_g, an approximate binary sample is obtained as

$$x_i^l = \tanh\left(\frac{\log q(X_i^l = 1) + \varepsilon_1 - \log q(X_i^l = -1) - \varepsilon_2}{2\tau_\text{g}} \right). \tag{5.6}$$

We refer to Appendix B.8 for a derivation of (5.6). Note that one can apply the *straight-through* Gumbel-softmax estimator [100] to obtain actual binary values x_i^l. In this case, (5.6) serves as a STE during backpropagation. In our experiments (see Section 5.3.3), we show that this does not yield improved results.

These steps are iterated up to the output layer where we eventually obtain a Gaussian approximation $q(a_i^L)$ over the output activations. The loss function is finally evaluated by generating a sample from $q(\mathbf{a}^L)$ using the reparameterization trick.

The second method, the *probabilistic forward pass*, is equivalent to the previous method up to the point where the binary activations are sampled from $q(\mathbf{x}^l)$. Instead of sampling from $q(\mathbf{x}^l)$, the probabilistic forward pass computes the Gaussian activation marginals $q(a_i^l)$ based on an input distribution $q(\mathbf{x}^{l-1})$. Assuming independence of the inputs $\mathbf{x}^{l-1}$ and the weights $\mathbf{W}^l$, the corresponding expressions are given by

$$\mu_{a_i^l} = \sum_k \mathbb{E}[w_{i,k}^l] \mathbb{E}[x_k^{l-1}] \tag{5.7}$$

and

$$\sigma_{a_i^l}^2 = \sum_k \mathbb{E}[w_{i,k}^l]^2 \mathbb{V}[x_k^{l-1}] + \mathbb{V}[w_{i,k}^l] \mathbb{E}[x_k^{l-1}]^2 + \mathbb{V}[w_{i,k}^l] \mathbb{V}[x_k^{l-1}]. \tag{5.8}$$

Note that for the first layer where the inputs are deterministic, these expressions reduce to (5.4). To obtain a sampling-free procedure, the loss function in the output layer is approximated by the second-order approximation (3.87) for the cross-entropy loss (i.e., for the negative of (3.90)).

For both discussed methods, we are required to pass the Gaussian activation distribution $q(\mathbf{a}^l)$ through the sign function. So far, we have ignored intermediate building blocks that commonly appear before the sign activation function. In particular, many architectures employ batch normalization and pooling operations, and there might be building blocks that are yet to be discovered in the future. It is often not straightforward how such building blocks can be generalized to distribution-valued inputs. We will discuss generalizations of batch normalization and max pooling to Gaussian distributions in Section 5.2.2 and Section 5.2.3, respectively. Note that such generalization are not required for the local reparameterization trick in combination with traditional continuous activation functions such as ReLU and tanh.

To cover another aspect, we want to mention that, under certain conditions, the activation distribution $q(\mathbf{a}^l)$ can be computed exactly. Consider a sum over K independent discrete ran-

dom variables $Z_1, \ldots, Z_K$. If the random variables Z_k take values from arbitrary discrete sets, computing the pmf of their sum requires exponential time since the sum can take exponentially many values. However, if the random variables take values from the same evenly spaced set, the pmf of their sum can be computed in subexponential time. This is best illustrated by considering two independent random variables Z_1 and Z_2 over the integers. Clearly their sum will also be an integer and we can express its pmf by a convolution as

$$p(Z_1 + Z_2 = z) = \sum_{k=-\infty}^{\infty} p(Z_1 = k) p(Z_2 = z - k). \tag{5.9}$$

For finite discrete random variables Z_1 and Z_2, the sum (5.9) simplifies to a finite sum that can be computed in subexponential time. Note that we can apply (5.9) repeatedly for sums over several independent discrete random variables. For discrete-valued DNNs with sign activations, the summands $w_{i,k}^l x_k^l$ of an activation a_i^l are from an evenly spaced discrete set such that, in principle, the marginal $q(a_i^l)$ can be computed exactly in subexponential time. However, due to efficiency reasons, the exact computation is not used in practice.

We conclude this section with a brief justification of the expected loss (5.2). Since our goal is to obtain a single discrete-valued DNN achieving a good performance, the question arises whether we can expect the *most probable* discrete weights to perform well if we perform well in expectation. For both methods, the local reparameterization trick and the probabilistic forward pass, the loss only depends on the means $\mathbb{E}[w]$ and the variances $\mathbb{V}[w]$. Using discrete weights with $v_1 = -1$ and $v_Q = 1$, we can represent any mean in the interval $[-1, 1]$. However, we can only achieve low variance if the mean $\mathbb{E}[w]$ is close to a weight in $\mathbb{D}$. Therefore, our approach can be seen as a particular way of parameterizing means $\mathbb{E}[w]$ and constrained variances $\mathbb{V}[w]$ through the probabilities $q_{\boldsymbol{\nu}}(\mathbf{W})$. Since we require small variances to obtain a small expected loss—in fact a point mass is optimal for the expected log-likelihood (5.2) as discussed above—optimization favors means $\mathbb{E}[w]$ that are close to values in $\mathbb{D}$. Consequently, also the most probable weights in $q_{\boldsymbol{\nu}}(\mathbf{W})$ are expected to perform well.

5.2 Model Details

In this section, we introduce details about our model and the adaptations of typical building blocks required for compatibility with our method. We start with the basic layout of our DNNs. Then we proceed with adaptations required for batch normalization and max pooling. Finally, we introduce our parameterization and initialization schemes for the weight distributions $q_{\boldsymbol{\nu}}(\mathbf{W})$.

5.2.1 Model Layout

The basic convolutional block for the sign activation function is depicted in Figure 5.2(a). We typically start with dropout, followed by a convolution. Motivated by Rastegari et al. [118], we perform the pooling operation (if present) right after the convolution to avoid information loss. Afterwards we perform batch normalization as described in Section 5.2.2, followed by computing the pmf after the sign function. Unless the probabilistic forward pass is used, we finally perform the local reparameterization trick using the Gumbel-softmax approximation. For continuous activation functions (ReLU or tanh; see Figure 5.2(b)), a Gaussian reparameterization is performed right after the convolution.

We do not perform batch normalization in the final layer. Instead we introduce a real-valued bias and divide the output activations by the square root of the number of incoming neurons. This normalization step counteracts the relatively large values of the discrete weights and the binary inputs from the previous layer. We found this to be crucial for training as otherwise

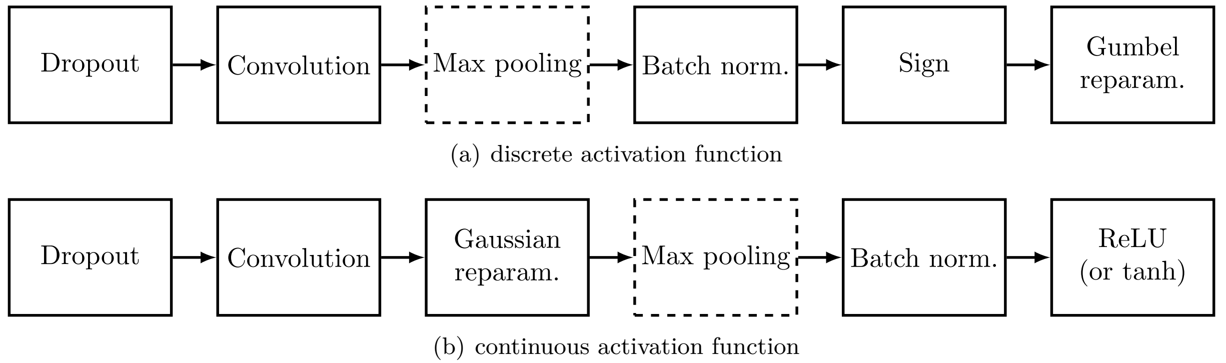


Figure 5.2: Convolutional blocks. Max pooling is not always present. For the probabilistic forward pass, the reparameterization block is omitted.

the output softmax would be too saturated in most cases. Moreover, we found dropout in our experiments to be particularly helpful as it improved performance considerably. Dropout was performed by randomly setting both the neuron's mean and its variance to zero in order to completely remove its influence.

5.2.2 Batch Normalization for Gaussian Distributions

As briefly mentioned in Section 5.1.3, we are required to generalize batch normalization to distributions. Batch normalization is particularly important when using sign activations to avoid excessive information loss [118]. We use the method proposed in [138] to normalize distributions to approximately have zero mean and unit variance. The mini-batch statistics for N_B samples are computed as

$$\mu_{\mathrm{bn},i} = \frac{1}{N_\mathrm{B}} \sum_{n=1}^{N_\mathrm{B}} \mu_{a_{n,i}} \quad \text{and} \quad \sigma^2_{\mathrm{bn},i} = \frac{1}{N_\mathrm{B}-1} \sum_{n=1}^{N_\mathrm{B}} \sigma^2_{a_{n,i}} + (\mu_{a_{n,i}} - \mu_{\mathrm{bn},i})^2. \tag{5.10}$$

Subsequently, batch normalization is computed as

$$\mu_{a_i} \leftarrow \frac{\mu_{a_i} - \mu_{\mathrm{bn},i}}{\sigma_{\mathrm{bn},i}} \cdot \gamma_{\mathrm{bn},i} + \beta_{\mathrm{bn},i} \quad \text{and} \quad \sigma^2_{a_i} \leftarrow \frac{\sigma^2_{a_i}}{\sigma^2_{\mathrm{bn},i}} \cdot \gamma^2_{\mathrm{bn},i}, \tag{5.11}$$

where $\beta_{\mathrm{bn},i}$ and $\gamma_{\mathrm{bn},i}$ are the learnable batch normalization parameters.

For predictions at test-time, batch normalization requires us to compute these statistics over the entire training set. Since this is typically too expensive, it is common practice to estimate these statistics by an exponential moving average over the mini-batches during training, i.e.,

$$\mu^{\mathrm{new}}_{\mathrm{tr},i} \leftarrow \xi_\mathrm{bn}\, \mu_{\mathrm{bn},i} + (1 - \xi_\mathrm{bn}) \mu^{\mathrm{old}}_{\mathrm{tr},i}, \tag{5.12}$$

where $\xi_\mathrm{bn} \in (0, 1)$ is a hyperparameter, and we proceed similarly for $\sigma_{\mathrm{tr},i}$. However, as already noted in [138], the statistics for the most probable discrete-valued DNN might differ significantly from those observed during training. Therefore, it is necessary to compute the batch statistics separately using the *most probable discrete-valued DNN*. This is accomplished by computing an exponential moving average over 100 random mini-batches of the training set *after* each epoch and right before evaluating the validation and test errors. In our experiments (see Section 5.3.7), we observed stronger fluctuations of the test errors and inferior overall performance when the statistics are computed during training according to (5.12). The computational overhead for this re-estimation was negligible compared to the overall training time. Note that batch normalization, although introducing real-valued variables, requires only a marginal computational

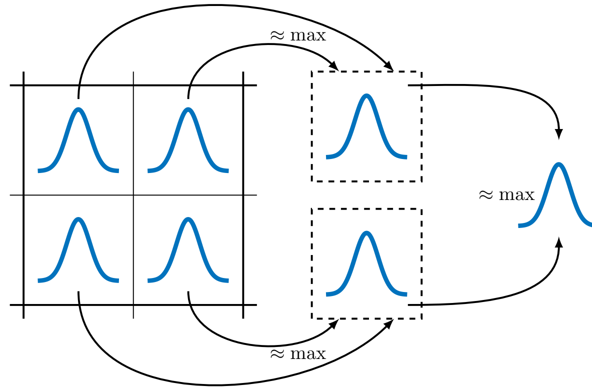


Figure 5.3: Max pooling approximation. In the first stage, the maximum of the two upper and the two lower Gaussians are approximated. In the next stage, the overall maximum is approximated.

overhead at test-time [190].

5.2.3 Max Pooling for Gaussian Distributions

Many CNN architectures involve max pooling operations where feature maps are downscaled by only passing the maximum of several spatially neighboring activations to the next layer. To generalize max pooling to distributions, we approximate the maximum of two Gaussians by another Gaussian using moment matching. Let μ_1, μ_2 and σ_1^2, σ_2^2 be the means and the variances of two *independent* Gaussians, respectively. According to [191], the mean $\mu_{\max}$ and the variance $\sigma_{\max}^2$ of the maximum of these Gaussians are given by

$$\mu_{\max} = \mu_1 \Phi(\beta) + \mu_2 \Phi(-\beta) + \alpha \phi(\beta) \qquad \text{and} \tag{5.13}$$

$$\sigma_{\max}^2 = (\sigma_1^2 + \mu_1^2)\,\Phi(\beta) + (\sigma_2^2 + \mu_2^2)\,\Phi(-\beta) + (\mu_1 + \mu_2)\,\alpha\,\phi(\beta) - \mu_{\max}^2, \tag{5.14}$$

where ϕ and Φ are the pdf and the cdf of a standard normal distribution, respectively, and

$$\alpha = \sqrt{\sigma_1^2 + \sigma_2^2} \quad \text{and} \quad \beta = \frac{\mu_1 - \mu_2}{\alpha}. \tag{5.15}$$

This scheme can be iteratively applied to approximate the maximum of several Gaussians. As long as the number of Gaussians is relatively small—CNNs typically involve 2×2 max pooling—this scheme results in a fairly efficient approximation for max pooling. Our scheme is illustrated in Figure 5.3. We first approximate the maximum of the two upper and the two lower activations by a Gaussian, and then we approximate the maximum of these two Gaussians by another Gaussian.

This is in contrast to the max pooling approximation proposed by Peters and Welling [138]. Their method generates samples from the Gaussian inputs and selects the mean and the variance of the input that generated the largest sample. Although the samples are generated from the input distributions, the properties of these distributions are effectively ignored afterwards. To see this more clearly, assume that we want to approximate the maximum over several standard normal distributions. In this case, the true mean grows with the number of Gaussians, but their method outputs a standard normal distribution irrespective of the number of inputs.

A more sophisticated approach has been proposed by Shekhovtsov and Flach [79]. In a first step, their method approximates the distribution of the argmax of a set of Gaussians. They present two versions for the argmax approximation, i.e., a quadratic-time and a faster linear-time approximation. Based on this approximation, their method subsequently approximates the mean and the variance of the maximum itself.

In our experiments (see Section 5.3.4), we also explore the possibility of computing a Monte Carlo estimate of the mean and the variance of the maximum. For this purpose, several samples of the maximum are generated by sampling multiple times from the Gaussian inputs using the reparameterization trick. Subsequently, the empirical mean and variance over these maxima is computed. Note that this approach requires at least two samples to obtain an unbiased non-zero estimate of the variance, and the approximation quality grows with the number of samples.

5.2.4 Parameterization and Initialization of Weight Distributions

Shayer et al. [137] introduced a parameterization for ternary distributions based on two probabilities, $q_w(W = 0)$ and $q_w(W = 1 | W \neq 0)$, which is not easily generalizable to distributions over more than three weights. In this work, we parameterize distributions over Q values using unconstrained unnormalized log-probabilities (logits) $\nu_{w,i}$ for $i \in \{1, \ldots, Q\}$. The normalized probabilities $q_w(w)$ can be recovered by applying the softmax function to the logits $\boldsymbol{\nu}_w$, i.e.,

$$q_w(W = v_i) = \mathrm{softmax}_i(\boldsymbol{\nu}_w). \tag{5.16}$$

This straightforward parameterization allows us to select the ith weight by setting $\nu_{w,i} > \nu_{w,i'}$ for $i' \neq i$. Due to the sum-to-one constraint of probabilities, we can reduce the number of parameters to $Q - 1$ by fixing an arbitrary logit, e.g., $\nu_{w,1} = 0$. However, we refrain to do so as it is more natural to increase a probability explicitly by increasing its corresponding logit rather than indirectly by reducing all other logits.

Moreover, Shayer et al. [137] introduced an initialization method for the distribution parameters $\boldsymbol{\nu}$ by matching the mean $\mathbb{E}[w]$ to the real-valued weights $\tilde{w}$ of a pre-trained network. We note that matching the mean $\mathbb{E}[w] = \tilde{w}$ is non-trivial since it is an underconstrained problem for $Q \geq 3$. In our experiments (see Section 5.3.5), we found a proper initialization scheme to be crucial since for randomly initialized logits one usually gets stuck in a bad local minimum. However, their initialization method also does not generalize easily to more than three weights, especially since it is coupled to their parameterization.

Therefore, we propose to use the following initialization scheme to approximately match the means which we found to be at least as effective as Shayer et al.'s approach for ternary weights (see Section 5.3.5). Let $v_1 < \ldots < v_Q$ be the set of discrete weight values. Furthermore, let $q_{\min}$ be a minimum probability that is required to avoid zero probabilities. The maximal probability is then given by $q_{\max} = 1 - (Q - 1)q_{\min}$ and we define $\Delta_q = q_{\max} - q_{\min}$. Given a real-valued weight $\tilde{w}$, we initialize the parameters $\boldsymbol{\nu}_w$ to obtain

$$q_w(W = v_i) = \begin{cases} q_{\min} + \Delta_q \frac{\tilde{w} - v_{i-1}}{v_i - v_{i-1}} & v_{i-1} < \tilde{w} \leq v_i \\ q_{\min} + \Delta_q \frac{v_{i+1} - \tilde{w}}{v_{i+1} - v_i} & v_i < \tilde{w} \leq v_{i+1} \\ q_{\max} & (i = 1 \wedge \tilde{w} < v_1) \vee (i = Q \wedge \tilde{w} > v_Q) \\ q_{\min} & \text{otherwise.} \end{cases} \tag{5.17}$$

Figure 5.4(a) illustrates this scheme for quinary weights. In practice, weight magnitudes might differ significantly across layers. Shayer et al. [137] addressed this by dividing the pre-trained weights $\tilde{w}$ in each layer by their layerwise standard deviation before applying (5.17). We propose the following scheme which distributes probabilities more uniformly across the discrete weight values in order to benefit from the increased expressiveness when using a larger Q. Let

$$\Phi_e^l(w) = 1/|\mathbf{W}^l| \sum_{\tilde{w} \in \mathbf{W}^l} \mathbb{I}[\tilde{w} \leq w] \tag{5.18}$$

be the empirical cdf of the weights in layer l. To achieve scale invariance, we compute $\tilde{w}^l \leftarrow \Phi_e^l(\tilde{w}^l)$ such that the weights $\tilde{w}^l$ cover the unit interval with equal spacing while preserving

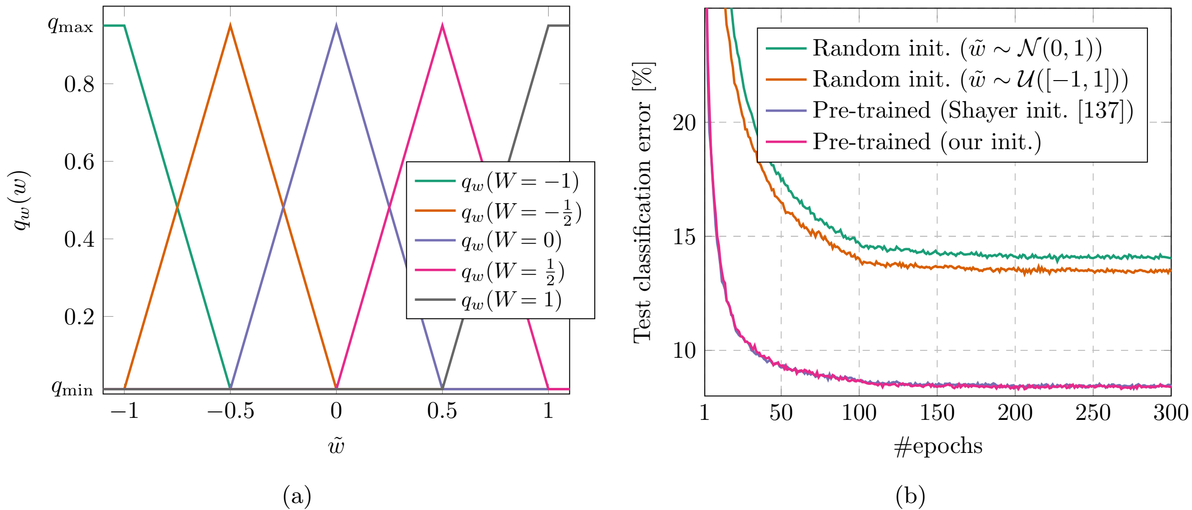


Figure 5.4: (a) Our initialization method for quinary weight distributions $q_w(w)$ based on pre-trained real-valued weights $\tilde{w}$. (b) Test classification error [%] over number of epochs (averaged over ten runs) on Cifar-10 for ternary weights using different initialization methods for $q_{\boldsymbol{\nu}}(\mathbf{W})$. For randomly initialized $q_{\boldsymbol{\nu}}(\mathbf{W})$, we sampled real-valued weights $\tilde{w}$ either from $\mathcal{N}(0, 1)$ or from $\mathcal{U}([-1, 1])$ before applying the method illustrated in (a) for ternary weights.

the relative order of the weights. Then we shift and scale the weights $\tilde{w}$ to cover the interval $[v_1 - \Delta_v/2, v_Q + \Delta_v/2]$. Finally, the probabilities $q_w(w)$ are initialized according to (5.17). This ensures that each discrete weight value v_i is initially selected equally often as the most probable weight in $q_w(w)$. We propose to use this scheme for the positive and the negative weights separately such that the signs of the weights are preserved.

5.3 Experiments

We performed classification experiments using the following general setup (dataset dependent settings are provided in Section 5.3.1). We optimized the probabilistic loss $\mathcal{L}_{\text{prob}}$ (5.1) for the cross-entropy loss $\mathcal{L}_{\text{CE}}$ (2.8) using Adam [11], and we report the test classification error of the epoch resulting in the best validation classification error. All experiments employ the local reparameterization trick using the Gumbel-softmax approximation, except those in Section 5.3.3 where we compare with other methods. We selected the regularizer $\mathcal{R}(\boldsymbol{\nu})$ to be the squared ℓ^2-norm over the logits [137] and set $\lambda = 10^{-10}$. Penalizing large logits can be seen as enforcing a uniform pmf $q_{\boldsymbol{\nu}}(\mathbf{W})$ and, therefore, increasing entropy and variance. As stated in [137], this rather helps to obtain better Gaussian approximations using the central limit theorem rather than to reduce overfitting. After each gradient update, we clip the logits to the range $[-5, 5]$. We set the initial learning rate to 10^{-2} for the logits and to 10^{-3} for all other parameters (batch normalization, bias in the final layer). The learning rate is reduced by a factor of α_{lr} after every τ_{lr} epochs, where τ_{lr} is a dataset dependent parameter that can be found in Section 5.3.1. We selected the Gumbel-softmax temperature $\tau_{\text{g}} = 1$ and the exponential moving average parameter of batch normalization $\xi_{\text{bn}} = 0.1$. For batch normalization, the activation statistics of the training set were estimated *after* each epoch as an exponential moving average over 100 mini-batches using the most probable discrete-valued DNN. The DNN architectures, the number of epochs, the mini-batch size, and the dropout rates are dataset dependent hyperparameters that are provided in Section 5.3.1.

The discrete-valued DNNs were initialized with pre-trained real-valued models for $q_{\max} = 0.95$ using the method described in Section 5.2.4. Pre-training was performed for DNNs with both

	ReLU					tanh				
Dataset	#epochs	learning rate	α_{lr}	τ_{lr}	λ_{wd}	#epochs	learning rate	α_{lr}	τ_{lr}	λ_{wd}
MNIST (PI)	600	$3 \cdot 10^{-4}$	0.1	150	10^{-5}	1,000	10^{-3}	0.5	150	10^{-4}
MNIST	300	$3 \cdot 10^{-3}$	0.1	150	10^{-6}	600	$3 \cdot 10^{-3}$	0.9885	1	10^{-6}
Cifar-10	300	10^{-4}	0.1	100	10^{-3}	300	10^{-3}	0.1	100	10^{-4}
Cifar-100	300	$3 \cdot 10^{-4}$	0.1	100	10^{-3}	600	$3 \cdot 10^{-4}$	0.4	100	10^{-3}
SVHN	300	10^{-3}	0.1	100	10^{-5}	300	10^{-3}	0.1	100	10^{-4}

Table 5.1: Dataset specific parameters used for real-valued pre-training.

ReLU and tanh activation functions. We applied weight decay regularization (i.e., a squared ℓ^2-norm penalty on the weights) with dataset dependent trade-off parameter λ_{wd}, and we selected $\xi_{\text{bn}} = 0.01$. For pre-training, the training set statistics for batch normalization were estimated by an exponential moving average *during* training.

Each experiment was conducted ten times and we report the average test errors and standard deviations over these ten runs. To be more precise, we pre-trained ten real-valued DNNs and always selected the model of the epoch achieving the best validation error to initialize $q_{\boldsymbol{\nu}}(\mathbf{W})$. For each of these ten pre-trained models, a single discrete-valued DNN was trained. Whenever we report results for real-valued DNNs, these results correspond to the pre-trained models. Unless stated otherwise, all results for discrete-valued DNNs are reported for the most probable model from $q_{\boldsymbol{\nu}}(\mathbf{W})$.

5.3.1 Dataset Setups

This section provides the experimental settings that are specific to the individual datasets. For a detailed description of the datasets we refer to Appendix A. Furthermore, the following experimental setups are specific to the training of the discrete weight distributions $q_{\boldsymbol{\nu}}(\mathbf{W})$. The settings specific to real-valued pre-training are summarized in Table 5.1. Note that real-valued pre-training and training of the weight distributions were performed using the same batch sizes and dropout rates.

MNIST (PI): The permutation-invariant setting of the MNIST dataset (see Appendix A.1) where each pixel is treated as independent feature. In particular, we are not allowed to take pixel locality into account, i.e., we do not use a CNN. For this setting we use the fully connected architecture

$$\text{FC1200} - \text{FC1200} - \text{FC10},$$

where FC1200 denotes a fully connected layer with 1,200 output neurons. We trained for 500 epochs using mini-batches of 100 samples with $\tau_{\text{lr}} = 100$ and $\alpha_{\text{lr}} = 0.5$. We used dropout probabilities (0.2, 0.4, 0.4) where the first entry corresponds to the input layer and the following entries correspond to the subsequent layers.

MNIST: The unconstrained setting of the MNIST dataset where we exploit the image structure. We use CNNs with the architecture

$$\text{32C5} - \text{P2} - \text{64C5} - \text{P2} - \text{FC512} - \text{FC10},$$

where 32C5 denotes that 5×5 filter kernels are applied and 32 output feature maps are generated, and P2 denotes that 2×2 max pooling is applied. We trained for 500 epochs using mini-batches of 100 samples with $\tau_{\text{lr}} = 100$ and $\alpha_{\text{lr}} = 0.5$. We used dropout probabilities (0, 0.2, 0.3, 0).

Dataset	Real+ReLU	Ternary+Sign	Quaternary+Sign	Quinary+Sign
MNIST (PI)	0.961 ± 0.041	1.235 ± 0.064	1.221 ± 0.052	1.207 ± 0.050
MNIST	0.474 ± 0.049	0.571 ± 0.053	0.566 ± 0.039	0.550 ± 0.038
Cifar-10	6.114 ± 0.250	8.532 ± 0.172	8.239 ± 0.135	8.168 ± 0.175
Cifar-100	24.528 ± 0.315	30.226 ± 0.277	29.816 ± 0.277	29.563 ± 0.313
SVHN	1.829 ± 0.062	2.533 ± 0.064	2.535 ± 0.053	2.522 ± 0.063

Table 5.2: Classification errors [%] for various DNN settings. We report means and standard deviations over ten experiments. Real+ReLU is the baseline that was used to initialize the discrete DNNs.

Dataset	Real+Tanh	Ternary+Sign	Quaternary+Sign	Quinary+Sign
MNIST (PI)	1.021 ± 0.049	1.372 ± 0.031	1.328 ± 0.068	1.306 ± 0.061
MNIST	0.585 ± 0.044	0.612 ± 0.038	0.598 ± 0.048	0.616 ± 0.035
Cifar-10	7.906 ± 0.157	9.491 ± 0.195	9.413 ± 0.154	9.234 ± 0.183
Cifar-100	30.384 ± 0.246	32.848 ± 0.206	32.745 ± 0.182	32.451 ± 0.257
SVHN	2.154 ± 0.038	2.639 ± 0.083	2.591 ± 0.075	2.626 ± 0.073

Table 5.3: Classification errors [%] for various DNN settings. We report means and standard deviations over ten experiments. Real+Tanh is the baseline that was used to initialize the discrete DNNs.

Cifar-10 and Cifar-100: Since both datasets, Cifar-10 and Cifar-100 (see Appendix A.3), are very similar and they mainly differ in the number of object categories, we use the same experimental setup and pre-processing steps for both datasets. For training, we perform data augmentation by shifting the images randomly by up to four pixels in each direction, and we randomly flip images along the vertical axis similar as in [23, 137]. We use the VGG-inspired [37] CNN architecture

$$2{\times}128\text{C3} - \text{P2} - 2{\times}256\text{C3} - \text{P2} - 2{\times}512\text{C3} - \text{P2} - \text{FC1024} - \text{FC10/100},$$

where 2×128C3 denotes two consecutive 128C3 blocks. We trained for 300 epochs using mini-batches of 100 samples with $\tau_{\text{lr}} = 100$ and $\alpha_{\text{lr}} = 0.1$. We used dropout probabilities (0, 0.2, 0.2, 0.3, 0.3, 0.3, 0.4, 0).

SVHN: Since the SVHN dataset is quite large (see Appendix A.4), we do not perform data augmentation. We use the same CNN architecture as for the Cifar datasets except that we use only half the number of feature maps in each convolutional layer, i.e.,

$$2{\times}64\text{C3} - \text{P2} - 2{\times}128\text{C3} - \text{P2} - 2{\times}256\text{C3} - \text{P2} - \text{FC1024} - \text{FC10}.$$

We trained for 100 epochs using mini-batches of 250 samples with $\tau_{\text{lr}} = 35$ and $\alpha_{\text{lr}} = 0.1$. We used the same dropout probabilities as for the Cifar datasets.

5.3.2 Classification Results

We start by investigating the influence of the activation function used for pre-training (ReLU and tanh) on the discrete DNNs. The corresponding test errors for ReLU pre-training and tanh pre-training are shown in Table 5.2 and Table 5.3, respectively. The pre-trained real-valued models using the ReLU activation clearly outperform their tanh counterpart. This also translates to better results when using the ReLU models to initialize the discrete-valued models using sign activations. This is rather surprising since the tanh activation provides a much better approximation to the sign function than the ReLU. We observed this clear advantage of the ReLU activation over tanh in every experiment and, therefore, we mostly focus on results for ReLU in the upcoming sections.

Next, we compare the results when using different types of discrete weights, namely ternary, quaternary, and quinary weights. Using ReLU pre-training, the results improve on all datasets

Dataset	Ternary+ReLU	Quaternary+ReLU	Quinary+ReLU	Ternary+Sign	Quaternary+Sign	Quinary+Sign
MNIST (PI)	1.035 ± 0.031	1.011 ± 0.047	1.022 ± 0.049	1.181 ± 0.041	1.194 ± 0.044	1.181 ± 0.053
MNIST	0.460 ± 0.034	0.485 ± 0.026	0.459 ± 0.036	0.577 ± 0.049	0.559 ± 0.031	0.556 ± 0.045
Cifar-10	6.170 ± 0.207	6.028 ± 0.152	5.873 ± 0.156	8.403 ± 0.097	8.116 ± 0.150	8.104 ± 0.197
Cifar-100	25.221 ± 0.221	25.088 ± 0.302	24.861 ± 0.276	30.281 ± 0.418	29.966 ± 0.357	29.787 ± 0.204
SVHN	1.865 ± 0.052	1.876 ± 0.055	1.858 ± 0.045	2.506 ± 0.043	2.488 ± 0.040	2.502 ± 0.069

Table 5.4: Classification errors [%] for various DNN settings. We report means and standard deviations over ten experiments. The models in the first three columns were initialized with Real+ReLU from Table 5.2. The models in the last three columns were initialized with the corresponding discrete weight model with ReLU activation.

Dataset	Ternary+Tanh	Quaternary+Tanh	Quinary+Tanh	Ternary+Sign	Quaternary+Sign	Quinary+Sign
MNIST (PI)	1.287 ± 0.061	1.249 ± 0.062	1.212 ± 0.067	1.292 ± 0.086	1.251 ± 0.090	1.240 ± 0.054
MNIST	0.578 ± 0.043	0.587 ± 0.024	0.568 ± 0.049	0.624 ± 0.058	0.615 ± 0.064	0.611 ± 0.062
Cifar-10	7.950 ± 0.165	7.826 ± 0.137	7.742 ± 0.133	9.175 ± 0.169	8.962 ± 0.178	8.866 ± 0.156
Cifar-100	29.572 ± 0.183	29.429 ± 0.280	29.117 ± 0.278	32.756 ± 0.394	32.630 ± 0.387	32.279 ± 0.305
SVHN	2.276 ± 0.054	2.247 ± 0.056	2.260 ± 0.051	2.561 ± 0.060	2.518 ± 0.067	2.517 ± 0.070

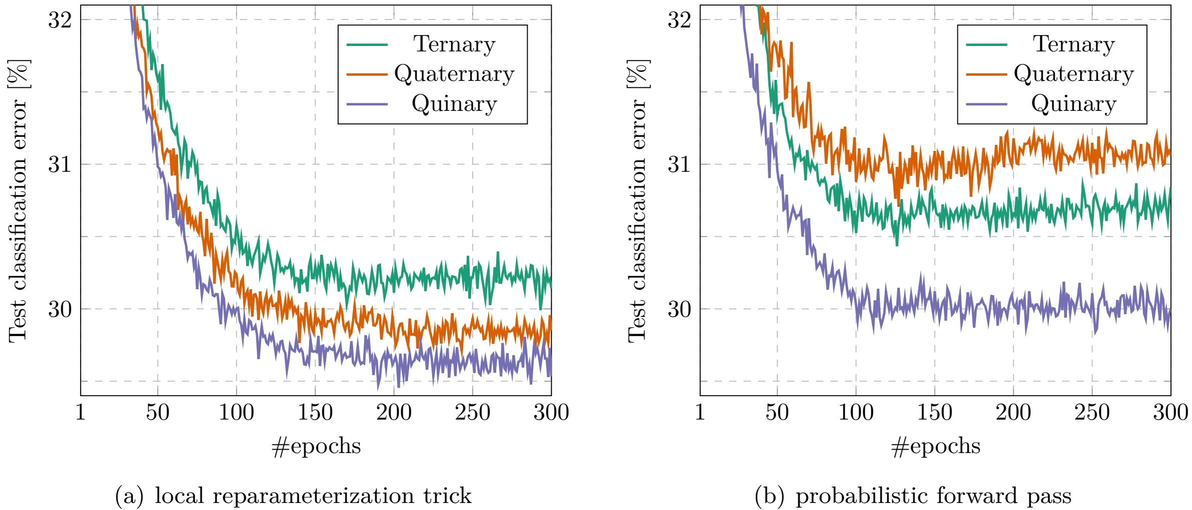
Table 5.5: Classification errors [%] for various DNN settings. We report means and standard deviations over ten experiments. The models in the first three columns were initialized with Real+Tanh from Table 5.3. The models in the last three columns were initialized with the corresponding discrete weight model with tanh *activation.*

when more discrete weight values are used, except on SVHN where results are similar for all discrete weight types. Using tanh pre-training, the results improve on the three datasets MNIST (PI), Cifar-10, and Cifar-100; on the remaining two datasets MNIST and SVHN the test error differences are marginal. Figure 5.5(a) shows some exemplary learning curves for the three discrete weight types on Cifar-100. These results show that—if model expressiveness is an issue—we can expect the accuracy to improve as more discrete weight values are used. Note that these results also indicate the effectiveness of the proposed initialization and parameterization methods.

In the next experiment, we performed a two-stage procedure using an intermediate training run where we only discretized the weights and kept the real-valued activation functions. In a subsequent training run, we used these DNNs to initialize the training with discrete weights *and* sign activation. For both of these training runs, we applied the same experimental setup as used for standard discrete weight training with sign activation from above.

The results for ReLU and tanh activations of these experiments are shown in Table 5.4 and Table 5.5, respectively. For both activation functions, the accuracy degradation is less severe when only the weights are quantized and the real-valued activation function is kept. A notable exception is observable on MNIST (PI) for the tanh activation where the accuracy degradation is similar as when in addition the sign activation is used. However, interestingly enough, the test error for weight-only quantization decreases on several datasets compared to their initializing models from Table 5.2 and Table 5.3 (see quinary weights on MNIST and Cifar-10 for both activation functions, and on Cifar-100 for tanh). This might either indicate a regularizing effect as has also been noted in [117], or weight quantization does not harm the expressiveness of the model and we benefit from longer training. After all, these findings are in line with other papers that have shown little performance degradation when the real-valued activation function is kept and only the weights are discretized [117, 137].

Next, we compare the corresponding test errors of discrete-valued DNNs with sign activation once pre-trained with real-valued models (Table 5.2 and Table 5.3) and once using the two-stage procedure (Table 5.4 and Table 5.5). The accuracy of the two-stage procedure always improves over the former on MNIST (PI), Cifar-10, and SVHN, and also on Cifar-100 for tanh. The results are mixed on the remaining datasets. In total, the accuracy improves in 10 out of 15 experiments for ReLU and in 13 out of 15 experiments for tanh. These results suggest that pre-training using a two-stage procedure is mostly beneficial. However, we note that it remains unclear whether the improved results are merely an artifact of the longer total training time.



(a) local reparameterization trick

(b) probabilistic forward pass

Figure 5.5: Test classification error [%] over number of epochs (averaged over ten runs) for different weight types on Cifar-100. Initialization was performed using ReLU pre-training. (a) Results for training with the reparameterization trick using the Gumbel-softmax approximation. (b) Results for training with the probabilistic forward pass.

	straight-through Gumbel approximation			probabilistic forward pass		
Dataset	Ternary+Sign	Quaternary+Sign	Quinary+Sign	Ternary+Sign	Quaternary+Sign	Quinary+Sign
MNIST (PI)	1.302 ± 0.043	1.259 ± 0.065	1.225 ± 0.048	1.211 ± 0.054	1.180 ± 0.076	1.183 ± 0.069
MNIST	0.552 ± 0.052	0.552 ± 0.046	0.544 ± 0.045	0.550 ± 0.036	0.565 ± 0.033	0.566 ± 0.039
Cifar-10	8.500 ± 0.120	8.412 ± 0.197	8.240 ± 0.131	8.686 ± 0.216	8.638 ± 0.345	8.454 ± 0.255
Cifar-100	30.703 ± 0.320	30.706 ± 0.223	30.153 ± 0.328	30.509 ± 0.339	30.895 ± 0.441	29.889 ± 0.265
SVHN	2.573 ± 0.055	2.505 ± 0.044	2.556 ± 0.034	2.600 ± 0.066	2.558 ± 0.067	2.556 ± 0.070

Table 5.6: Classification errors [%] for various DNN settings. We report means and standard deviations over ten experiments. The models in the first three columns were trained using the local reparameterization trick with the straight-through Gumbel estimator. The models in the last three columns were trained using the probabilistic forward pass. Initialization was performed using ReLU pre-training.

We also compare our method with other work. We select [115, 118, 130, 138] since their quantization is similar to ours. Hubara et al. [115] use binary weights and sign activations, albeit using larger architectures. They report two results and achieve on average 1.18±0.22% on MNIST (PI), 10.775±0.625% on Cifar-10, and 2.66±0.135 on SVHN. XNOR-Net [118] uses real-valued *data-dependent* scale factors to perform a binary convolution. Using the same structure as [115], they achieve 10.17% on Cifar-10. DoReFa-Net [130] achieves 2.9% on SVHN using binary weights and binary 0-1 activations. The work in [138] is closest to ours and achieves 0.74% on MNIST and 10.30% on Cifar-10 using ternary weights and sign activations.

5.3.3 Straight-Through Gumbel Estimator and Probabilistic Forward Pass

In this section, we compare training using the standard setting (i.e., local reparameterization trick in conjunction with the Gumbel-softmax approximation) with two different training methods. The first method replaces the Gumbel-softmax approximation with the straight-through Gumbel estimator. The second method applies the probabilistic forward pass.

The results for both methods are summarized in Table 5.6. The results are slightly in favor of the standard setting (Table 5.2). This indicates that it is less important to operate on truly discrete activations during training as it is the case for the straight-through Gumbel estimator. When comparing with the probabilistic forward pass, we observe that the tendency of obtaining better results when using more discrete weight values is less pronounced. For instance, in Fig-

ure 5.5(b) quaternary weights perform worst whereas more discrete weight values result in lower test errors for the standard setting in Figure 5.5(a).

We note that the training behavior of both compared methods might be different to the standard setting and, therefore, adapted hyperparameters might yield improved results. However, we do not expect substantial accuracy gains compared to the standard setting by further tuning the hyperparameters.

5.3.4 Different Max Pooling Methods

In this section, we compare different methods for max pooling applied to Gaussian inputs. In particular, we compare the following five methods for 2×2 max pooling:

Maximum mean For each 2×2 region, we select the mean and the variance of the location with the largest mean.

Maximum sample For each spatial location, a sample from the corresponding Gaussian is generated. Then we select for each 2×2 region the mean and the variance of the Gaussian that generated the largest sample. This method was proposed by Peters and Welling [138].

Reparameterization Multiple samples of the maximum of a 2×2 region are computed by generating multiple samples from the Gaussian activations. Importantly, the Gaussian samples are generated using the reparameterization trick to remain compatible with backpropagation. We then compute the empirical mean and the empirical variance of the sampled maxima for each 2×2 region. We conducted experiments for different numbers of generated samples.

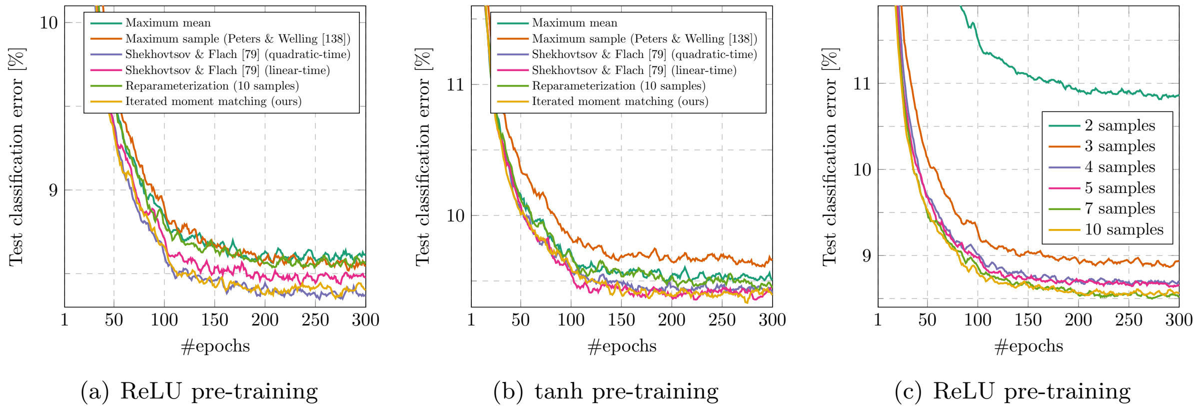
Shekhovtsov & Flach The method from Shekhovtsov and Flach [79] for approximating the mean and the variance of the maximum of several Gaussians. We conduct experiments for their two proposed methods, namely a quadratic-time approximation and a faster linear-time approximation.

Iterated moment matching Our proposed method (see Section 5.2.3) that first approximates the mean and the variance of the two upper and the two lower spatial locations. Then we combine these two estimates to obtain an overall approximation of the maximum of the 2×2 region.

Figure 5.6(a) shows the test error over the number of epochs on Cifar-10 for each of these methods. The discrete-valued DNNs employ ternary weights and were initialized using ReLU pre-training. Our iterated moment matching method and the quadratic-time approximation from Shekhovtsov and Flach perform best. The linear-time method from Shekhovtsov and Flach performs slightly worse, but is faster for 2×2 max pooling compared to their quadratic-time approximation. The remaining three methods (maximum mean, maximum sample, and reparameterization) achieve a similar accuracy which is worse compared to the other methods.

Figure 5.6(b) shows the same results for discrete-valued DNNs initialized using tanh pre-training. The results are qualitatively similar compared to the ReLU experiments. Notably, their is no accuracy gap between the linear-time and the quadratic-time approximations from Shekhovtsov and Flach. Furthermore, the maximum sample method from Peters and Welling gets clearly outperformed by all the other methods.

Next, we investigate the influence of the number of samples used for the reparameterization sampling method. The results are shown in Figure 5.6(c). At least two samples are required in order to estimate the variance of the maximum. However, using only two samples performs rather poor. The accuracy improves considerably for three samples and it continues to improve as more samples are used. We did not conduct experiments for more than ten samples, but it appears that the test error already saturates.



(a) ReLU pre-training (b) tanh pre-training (c) ReLU pre-training

Figure 5.6: Test error [%] over number of epochs (averaged over ten runs) for different max pooling approximations on Cifar-10 using ternary weights (see main text for a detailed description of the methods). For better visualization of the results, the depicted traces were smoothed using a simple moving average over five epochs. (a) Initialization was performed using ReLU pre-training. (b) Initialization was performed using tanh *pre-training. (c) Various numbers of samples for reparameterization sampling. Initialization was performed using ReLU pre-training.*

We emphasize that this method incurs a substantial computational overhead if many samples are used. Furthermore, note that for each drawn sample, only a single input of the 2×2 region receives a gradient during backpropagation (the location whose sample is maximal). On the contrary, in our method and the methods from Shekhovtsov and Flach, every input location contributes to the maximum and, therefore, receives a gradient. We believe that this might also be a reason for these methods to perform better. In summary, our results show that the quality of the max pooling approximation is important and that distribution-aware max pooling can improve the accuracy.

5.3.5 The Influence of Parameter Initialization and Dropout

The following experiments demonstrate the influence of parameter initialization on the prediction accuracy. We conducted several experiments on Cifar-10 using ternary weights.

In the first experiment, we compare our initialization method for $q_{\boldsymbol{\nu}}(\mathbf{W})$ using ReLU pre-training with two random initialization methods. Both of these methods assign random values to the real-valued weights $\tilde{w}$ and then initialize the probabilities $q_w(w)$ according to (5.17). The first method samples real-valued weights $\tilde{w} \sim \mathcal{N}(0,1)$. The second method samples real-valued weights $\tilde{w} \sim \mathcal{U}([-1,1])$.

The results are shown in Figure 5.4(b). Our method converges faster than the random strategies and achieves 4.964% lower absolute test error than the uniform random strategy, which in turn achieves approximately 0.5% lower test error than the Gaussian strategy. The random initialization schemes seem to get stuck in bad local minima. This indicates that the loss surface is substantially more delicate to optimize compared to training a conventional real-valued DNN.

We also compare our method to the initialization method from Shayer et al. [137]. To do so, we compute the probabilities that would have been assigned by their method and initialize our parameterization of $q_{\boldsymbol{\nu}}(\mathbf{W})$ using the corresponding normalized logits. Both methods perform similar and the two traces are almost indistinguishable. We emphasize, however, that our initialization method was designed to generalize to more than three weights, and we did not expect to outperform the method from [137] using ternary weights. In summary, this highlights that a proper initialization strategy is crucial for the training of weight distributions.

This raises the question whether it pays off to put more effort into pre-training to eventually obtain a better discrete-valued DNN. To answer this question, we optimized several dropout rates

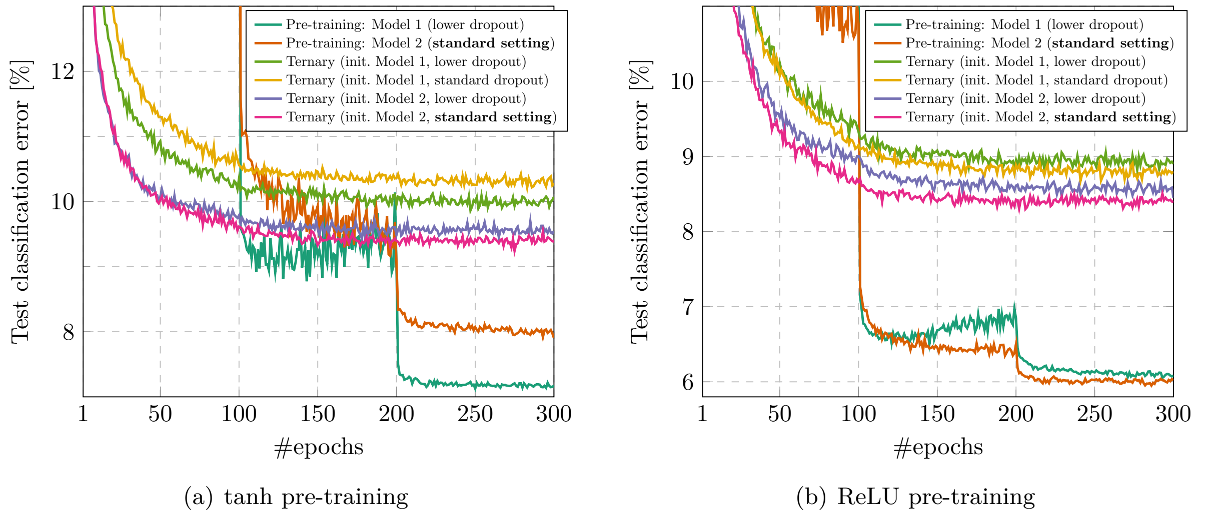


Figure 5.7: Test classification error [%] over number of epochs (averaged over ten experiments) on Cifar-10 for different dropout settings. The standard setting *traces correspond to the results reported in Table 5.3. The* standard dropout *setting uses the same dropout rates but a different pre-trained model. The* lower dropout *setting uses lower dropout rates (see text for details). (a) Results for* tanh *pre-training. (a) Results for ReLU pre-training.*

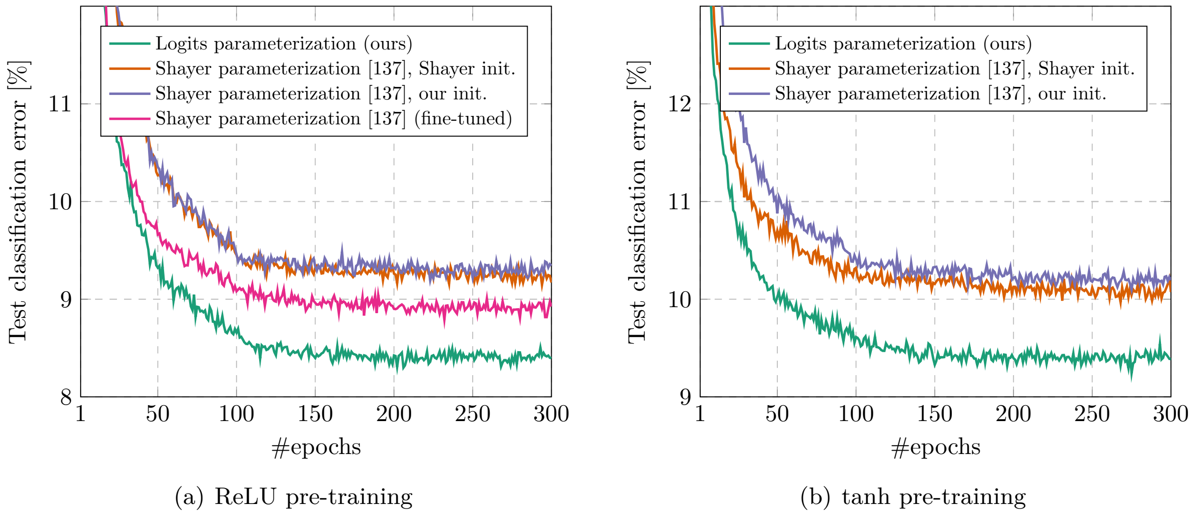
for the initial real-valued DNN with tanh activation, while keeping all the other hyperparameters the same. This resulted in dropout rates (0, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1). We refer to this dropout setting as *lower dropout* in the remainder of this section.

The corresponding learning curves for pre-training and the training of discrete-valued DNNs are shown in Figure 5.7(a). Using *lower dropout*, we achieve an average test error of 7.133% for the pre-trained DNNs—an absolute improvement of 0.773% compared to the standard setting reported in Table 5.3. However, when using these models to initialize $q_{\boldsymbol{\nu}}(\mathbf{W})$, we obtain inferior accuracy for the discrete-valued DNNs with sign activation. Importantly, this does not depend on whether we train the discrete-valued DNNs using the original dropout setting or *lower dropout*.

The results look similar but less pronounced for the ReLU activation function (Figure 5.7(b)). Here, using *lower dropout*, we obtain a slightly worse accuracy during pre-training. However, the accuracy gap for the discrete-valued DNNs is relatively large compared to the smaller gap after pre-training.

We can also see that the original dropout rates perform best when they are used both during pre-training *and* during training of the discrete-valued DNNs. However, we also observed that different dropout rates in both stages can improve accuracy. More specifically, we conducted an experiment on MNIST (PI) using different dropout rates of (0.1, 0.2, 0.3) for training of the discrete-valued DNN, but we kept the original dropout rates of (0.2, 0.4, 0.4) during pre-training. For initialization using ReLU pre-training, this yields improved test errors of 1.210%, 1.175%, and 1.165% for ternary, quaternary, and quinary weights, respectively. For initialization using tanh pre-training, the respective test errors improved to 1.289%, 1.261%, and 1.264%. When using these lower dropout rates during pre-training, we obtained worse results for discrete weights using both dropout settings. Note that we do not report these improved results in Table 5.2 and Table 5.3 to maintain the clear experimental setup of using equal dropout rates for pre-training and training of the weight distributions $q_{\boldsymbol{\nu}}(\mathbf{W})$.

These results suggest that dropout might play an important role for the overall accuracy, especially during pre-training. For practical applications, tuning the dropout rates might be a key factor for obtaining a high accuracy. However, it is not well understood what properties render a real-valued DNN a good candidate to initialize the weight distributions $q_{\boldsymbol{\nu}}(\mathbf{W})$.



(a) ReLU pre-training

(b) tanh pre-training

Figure 5.8: Test classification error [%] over number of epochs on Cifar-10 for ternary weights. We compare different parameterizations and initialization methods for $q_{\boldsymbol{\nu}}(\mathbf{W})$. (a) Results for initialization using ReLU pre-training. For the fine-tuned *setting, an extended hyperparameter optimization was conducted. (b) Results for initialization using* tanh *pre-training.*

5.3.6 The Influence of the Distribution Parameterization

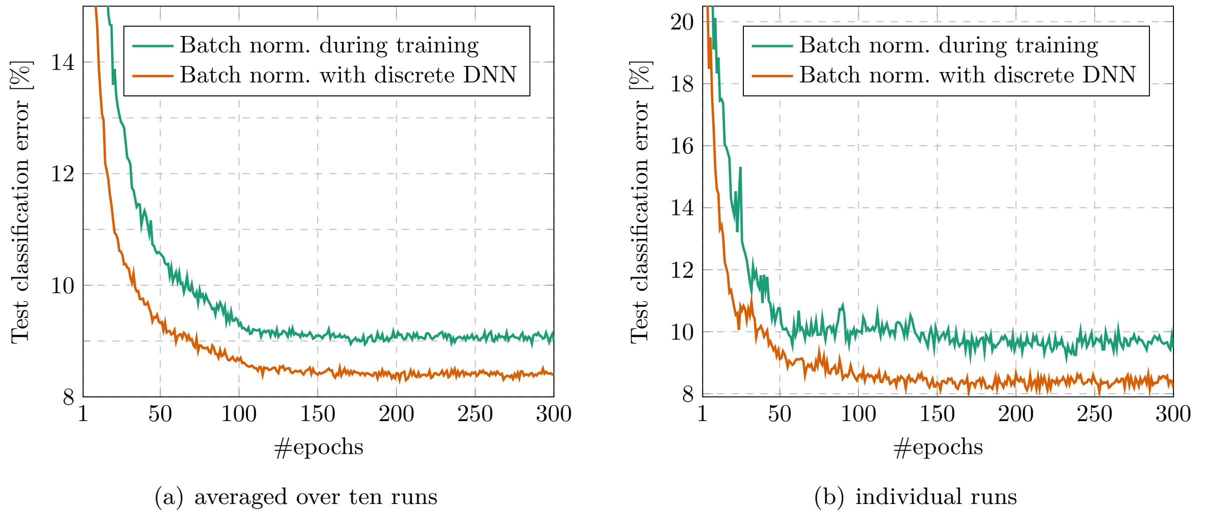
In the next experiments, we compare our parameterization of $q_{\boldsymbol{\nu}}(\mathbf{W})$ for ternary weights with the parameterization introduced by Shayer et al. [137]. Figure 5.8 shows learning curves for the two parameterizations on Cifar-10. To ensure that the test error discrepancy between the two methods can be primarily attributed to the parameterization and not to other experimental settings, we trained the parameterization from Shayer et al. using both our and their proposed initialization method for $q_{\boldsymbol{\nu}}(\mathbf{W})$. Both initialization methods yield similar results, showing that our initialization method also performs well using their parameterization.

Considering that their parameterization results in a different loss surface and a potentially different training behavior, we fine-tuned the learning rate and the regularization parameter λ for their parameterization. The corresponding results for initialization using ReLU pre-training are shown in Figure 5.8(a). We obtained a mean test error of $8.881\% \pm 0.134$ for a learning rate of $2 \cdot 10^{-2}$ and $\lambda = 10^{-11}$. Even for the fine-tuned setting, there is still a substantial test error gap between the two parameterizations. Although our parameterization requires more parameters, these results strongly indicate that our parameterization results in a loss surface that facilitates the optimization procedure.

5.3.7 The Influence of Batch Normalization

In the next experiment, we verify the importance of estimating the training set statistics required for batch normalization using the most probable discrete-valued DNN from $q_{\boldsymbol{\nu}}(\mathbf{W})$. Therefore, we compare the results of two experiments that only differ in the estimation of the training set statistics: The first method computes an exponential moving average over the statistics (5.10) obtained during training. The second method employs the standard setting, i.e., computing an exponential moving average over 100 randomly selected mini-batches using the most probable discrete-valued DNN after every epoch.

The corresponding learning curves on Cifar-10 for ternary weights and initialization using ReLU pre-training are shown in Figure 5.9(a). The accuracy degrades heavily when the statistics are estimated during training. Moreover, we observe stronger fluctuations in the learning curves of individual experiments, especially in the early training phase. Since this fluctuation behavior



(a) averaged over ten runs

(b) individual runs

Figure 5.9: Test classification error [%] over number of epochs on Cifar-10 for ternary weights. Initialization was performed using ReLU pre-training. Results were obtained by estimating the training set statistics for batch normalization using an exponential moving average (i) during training and (ii) after every epoch using the most probable discrete-valued DNN over 100 mini-batches. (a) Average test errors over ten runs. (b) Test errors of two individual runs initialized using the same pre-trained DNN.

is lost in the averaged results of Figure 5.9(a), two exemplary learning curves of individual experiments using the same pre-trained DNN for initialization are shown in Figure 5.9(b).

Note that the estimation of the training set statistics does not influence the training procedure for the remaining parameters. Furthermore, we emphasize that both experiments were initialized using the same pre-trained models. As a consequence, the accuracy difference can be clearly attributed to the different methods of estimating the batch normalization statistics.

5.3.8 Model Averaging

So far, our experimental evaluation was restricted to analyzing the most probable discrete-valued DNN from $q_{\boldsymbol{\nu}}(\mathbf{W})$. To evaluate the information present in the learned discrete weight distribution $q_{\boldsymbol{\nu}}(\mathbf{W})$, we conducted model averaging experiments by sampling DNNs from $q_{\boldsymbol{\nu}}(\mathbf{W})$.

All reported results are obtained by averaging the unnormalized logits $\mathbf{a}^L$ right before the softmax activation is applied. We also conducted additional experiments (results not reported) where we evaluated (i) averaging of the softmax predictions $\mathbf{x}^L$ and (ii) predictions by majority vote. In essence, all averaging methods achieve better results as more samples are averaged. Averaging the logits and the softmax outputs performs similarly (but slightly in favor of logit averaging) which both consistently outperform majority vote. For softmax averaging, it is important to perform output activation normalization; otherwise the softmax outputs are closer to one-hot and, therefore, a similar behavior to majority vote is obtained.

Furthermore, for each DNN sampled from $q_{\boldsymbol{\nu}}(\mathbf{W})$, we re-estimate the batch normalization parameters by computing an exponential moving average over 100 mini-batches from the training set. This mostly improves the accuracy compared to using the batch normalization statistics from the most probable DNN of $q_{\boldsymbol{\nu}}(\mathbf{W})$ (results not reported), especially when only few samples are averaged. However, we emphasize that this statement does not hold true in general, e.g., on Cifar-10 and SVHN the accuracy deteriorates when the batch normalization parameters are re-estimated. We believe that the reason for this lies in a reduction of model diversity due to the re-estimation. The results show that at least individual samples from $q_{\boldsymbol{\nu}}(\mathbf{W})$ achieve a higher accuracy on every dataset when their batch normalization parameters are re-estimated.

Dataset	1 sample	3 samples	5 samples	10 samples	20 samples	100 samples
MNIST (PI)	1.239 ± 0.045	1.215 ± 0.047	1.210 ± 0.049	1.205 ± 0.049	1.199 ± 0.047	1.196 ± 0.047
	1.211 ± 0.047	1.177 ± 0.043	1.174 ± 0.046	1.168 ± 0.047	1.168 ± 0.050	1.161 ± 0.048
	1.200 ± 0.054	1.175 ± 0.049	1.174 ± 0.052	1.169 ± 0.052	1.169 ± 0.050	1.169 ± 0.046
MNIST	0.559 ± 0.053	0.530 ± 0.052	0.527 ± 0.051	0.525 ± 0.051	0.522 ± 0.051	0.522 ± 0.050
	0.551 ± 0.048	0.532 ± 0.038	0.527 ± 0.038	0.527 ± 0.037	0.525 ± 0.036	0.523 ± 0.037
	0.560 ± 0.047	0.537 ± 0.039	0.534 ± 0.038	0.531 ± 0.040	0.530 ± 0.040	0.531 ± 0.041
Cifar-10	9.211 ± 0.246	8.071 ± 0.203	7.837 ± 0.209	7.660 ± 0.186	7.568 ± 0.180	7.517 ± 0.179
	9.274 ± 0.297	8.036 ± 0.186	7.778 ± 0.163	7.582 ± 0.162	7.491 ± 0.159	7.394 ± 0.161
	9.014 ± 0.262	7.916 ± 0.200	7.684 ± 0.175	7.506 ± 0.165	7.426 ± 0.167	7.361 ± 0.161
Cifar-100	30.981 ± 0.379	28.647 ± 0.271	28.206 ± 0.277	27.863 ± 0.239	27.706 ± 0.261	27.542 ± 0.280
	30.978 ± 0.391	28.500 ± 0.234	28.015 ± 0.222	27.576 ± 0.183	27.383 ± 0.181	27.223 ± 0.194
	30.400 ± 0.369	28.215 ± 0.283	27.769 ± 0.244	27.450 ± 0.246	27.300 ± 0.242	27.143 ± 0.264
SVHN	2.566 ± 0.074	2.344 ± 0.056	2.301 ± 0.053	2.268 ± 0.050	2.253 ± 0.051	2.241 ± 0.051
	2.525 ± 0.058	2.304 ± 0.054	2.262 ± 0.050	2.230 ± 0.048	2.211 ± 0.047	2.194 ± 0.047
	2.534 ± 0.065	2.321 ± 0.058	2.289 ± 0.066	2.257 ± 0.064	2.246 ± 0.060	2.233 ± 0.060

Table 5.7: Test classification errors [%] of model averaging for selected numbers of samples from $q_{\boldsymbol{\nu}}(\mathbf{W})$. The first, second, and third row of each dataset correspond to ternary, quaternary, and quinary weights, respectively. For each of ten weight distributions $q_{\boldsymbol{\nu}}(\mathbf{W})$, ten experiments of averaging up to 100 predictions were conducted. We report the mean and the standard deviation over these 100 experiments. Initialization was performed using ReLU pre-training.

Dataset	1 sample	3 samples	5 samples	10 samples	20 samples	100 samples
MNIST (PI)	1.352 ± 0.050	1.316 ± 0.049	1.310 ± 0.046	1.311 ± 0.040	1.306 ± 0.044	1.305 ± 0.041
	1.308 ± 0.061	1.273 ± 0.061	1.265 ± 0.063	1.263 ± 0.060	1.260 ± 0.064	1.258 ± 0.064
	1.298 ± 0.070	1.270 ± 0.066	1.267 ± 0.065	1.263 ± 0.063	1.261 ± 0.061	1.252 ± 0.054
MNIST	0.612 ± 0.042	0.572 ± 0.032	0.565 ± 0.032	0.561 ± 0.027	0.554 ± 0.026	0.550 ± 0.024
	0.623 ± 0.036	0.592 ± 0.036	0.593 ± 0.034	0.592 ± 0.035	0.588 ± 0.039	0.585 ± 0.037
	0.618 ± 0.044	0.586 ± 0.037	0.579 ± 0.033	0.570 ± 0.025	0.566 ± 0.025	0.564 ± 0.020
Cifar-10	9.965 ± 0.242	8.866 ± 0.152	8.640 ± 0.144	8.468 ± 0.131	8.386 ± 0.125	8.324 ± 0.123
	10.152 ± 0.301	8.902 ± 0.200	8.677 ± 0.195	8.499 ± 0.152	8.398 ± 0.149	8.307 ± 0.134
	9.729 ± 0.207	8.704 ± 0.161	8.504 ± 0.136	8.360 ± 0.130	8.299 ± 0.131	8.255 ± 0.137
Cifar-100	34.121 ± 0.419	31.506 ± 0.299	30.975 ± 0.240	30.562 ± 0.225	30.325 ± 0.219	30.170 ± 0.216
	35.113 ± 0.816	31.793 ± 0.325	31.111 ± 0.314	30.600 ± 0.259	30.326 ± 0.212	30.138 ± 0.227
	33.733 ± 0.482	31.257 ± 0.360	30.719 ± 0.320	30.355 ± 0.318	30.176 ± 0.299	30.023 ± 0.280
SVHN	2.702 ± 0.134	2.446 ± 0.094	2.394 ± 0.087	2.355 ± 0.082	2.337 ± 0.078	2.325 ± 0.076
	2.690 ± 0.084	2.424 ± 0.062	2.374 ± 0.056	2.336 ± 0.051	2.311 ± 0.054	2.299 ± 0.050
	2.638 ± 0.078	2.408 ± 0.052	2.367 ± 0.053	2.330 ± 0.052	2.318 ± 0.048	2.302 ± 0.048

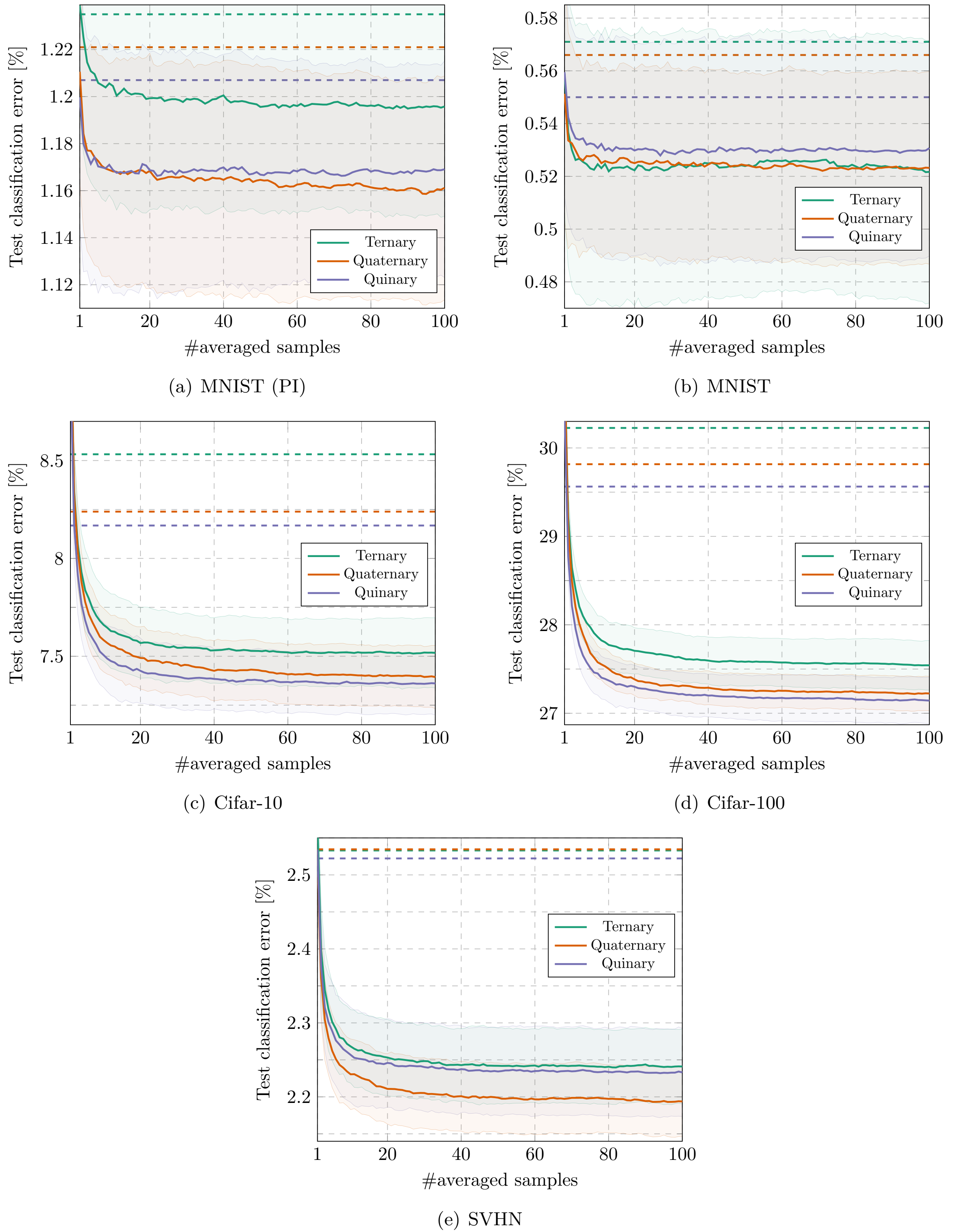
Table 5.8: Test classification errors [%] of model averaging for selected numbers of samples from $q_{\boldsymbol{\nu}}(\mathbf{W})$. The first, second, and third row of each dataset correspond to ternary, quaternary, and quinary weights, respectively. For each of ten weight distributions $q_{\boldsymbol{\nu}}(\mathbf{W})$, ten experiments of averaging up to 100 predictions were conducted. We report the mean and the standard deviation over these 100 experiments. Initialization was performed using tanh *pre-training.*

However, in some experiments the ensemble seems to benefit from including weaker but more diverse DNNs. We conclude that whether to apply such a re-estimation or not remains a tunable parameter of the method, whose choice depends on the dataset and on the number of DNNs one is willing to average over.

The results for logit averaging with batch normalization re-estimation and initialization using ReLU pre-training are shown in Figure 5.10. Table 5.7 lists concrete test errors for selected numbers of averaged models. Table 5.8 lists the same results for initialization using tanh pre-training. In all our experiments, three samples were sufficient for model averaging to surpass the accuracy of the most probable model; in most cases two samples were already sufficient.

In Figures 5.10(a), 5.10(b), and 5.10(e), we can see that the accuracy order of different weight types (ternary, quaternary, and quinary) is different for model averaging and the most probable DNN. However, this is not a major concern as the weight distribution $q_{\boldsymbol{\nu}}(\mathbf{W})$ and ensembles inferred from it can be regarded as a different, much more expressive model class than the single most probable DNN. This becomes evident when measuring the memory overhead for storing the model parameters. The real-valued variational parameters $\boldsymbol{\nu}$ require substantially more memory compared to the weights of a *real-valued* DNN with the same architecture.

In summary, only few samples are required to substantially outperform the corresponding most probable model from $q_{\boldsymbol{\nu}}(\mathbf{W})$. We note that an ensemble can potentially be used to obtain prediction uncertainties to further guide any applications depending on the ensemble's output.



(a) MNIST (PI)

(b) MNIST

(c) Cifar-10

(d) Cifar-100

(e) SVHN

Figure 5.10: Test classification error [%] over number of averaged predictions obtained from samples of $q_{\boldsymbol{\nu}}(\mathbf{W})$. We report the mean and one standard deviation (shaded region) over ten experiments for ten different weight distributions $q_{\boldsymbol{\nu}}(\mathbf{W})$ (i.e., 100 experiments in total). The test errors of the single most probable DNNs from $q_{\boldsymbol{\nu}}(\mathbf{W})$ are shown as dashed lines. Initialization was performed using ReLU pre-training.

However, since our study focuses on the single most probable discrete-valued DNN, an experimental evaluation of uncertainties obtained from an ensemble is beyond the scope of this work.

5.4 Discussion

We have introduced a method to train discrete-valued DNNs by means of an indirect procedure: In the first step, a discrete distribution over the weights $q_{\boldsymbol{\nu}}(\mathbf{W})$ is trained by gradient-based optimization. In the second step, a discrete-valued DNN is inferred from $q_{\boldsymbol{\nu}}(\mathbf{W})$ either by taking its most probable weights or by sampling from it.

Whereas previous works (see [137, 138]) are tailored to binary and ternary weights, our method is applicable to arbitrary discrete weights. This is accomplished by simpler parameterization and initialization schemes for the discrete weight distribution $q_{\boldsymbol{\nu}}(\mathbf{W})$. Furthermore, we have introduced a distribution-aware approximation for max pooling. This is in contrast to the method proposed in [138] where the input distributions are used to randomly select a maximum component, but not to approximate the mean and the variance of the maximum itself. As opposed to the majority of the literature where the weights of the input and output layers are not quantized, we also quantize the weights in these layers.

In our experiments, we achieved state-of-the-art performance on several image classification datasets. We have shown that both the local reparameterization trick and the probabilistic forward pass are effective methods to train $q_{\boldsymbol{\nu}}(\mathbf{W})$. Our method does not rely on the STE. Compared to the standard continuous Gumbel-softmax approximation, we did not observe improved accuracies when the straight-through Gumbel estimator is employed.

Our experiments show that using more discrete weight values mostly results in higher accuracy. This allows us to effectively trade off between computational requirements and accuracy. Nevertheless, we note that we cannot always expect accuracies to improve if more weights are used. The optimal number of discrete weight values depends on the dataset and the architecture, and using fewer weights might actually have a regularizing effect.

When only the weights are quantized and the activations remain real-valued, we observed almost no accuracy degradation compared to using real-valued weights and activations. This is in line with the majority of the literature stating that activation quantization has a stronger impact on the accuracy than weight quantization. Using a two-stage procedure where a first training run quantizes only the weights and a subsequent second training run additionally quantizes the activations, we mostly observed higher accuracies than by quantizing both weights and activations immediately. However, we did not investigate whether this is a mere artifact of a longer total training time.

Our model averaging experiments using samples from $q_{\boldsymbol{\nu}}(\mathbf{W})$ have shown that only few (at most three) samples are sufficient to outperform the most probable DNN. By selecting an appropriate number of samples, we can trade off between computational costs and accuracy. We found that re-estimating the batch normalization statistics for the individual DNNs resulted in higher accuracy when only few samples are averaged. However, in some cases we observed the opposite when many samples were averaged. We conclude that whether to apply such a re-estimation or not remains a tunable hyperparameter of the proposed method.

We have shown that our initialization method for $q_{\boldsymbol{\nu}}(\mathbf{W})$ is at least as effective as the method proposed in [137] for ternary weights. Our experiments show that a proper initialization is crucial to obtain a reasonable accuracy and extracting knowledge from a pre-trained real-valued DNN is a very effective method. We believe that the final accuracy of a discrete-valued DNN does not crucially depend on how this knowledge is extracted exactly and there might be many different ways of successfully achieving this. Therefore, a simple solution suitable for arbitrary discrete weights as proposed in this chapter might be an optimal choice.

We found that initialization using pre-trained DNNs using ReLU activations consistently results in higher accuracies compared to using the tanh activation. As expected, the ReLU activation outperforms tanh for real-valued DNNs. However, it is rather surprising that these results translate to discrete-valued DNNs considering that the tanh is much closer in shape to the ultimately used sign function.

Regarding the parameterization of $q_{\boldsymbol{\nu}}(\mathbf{W})$, we have observed that our logit parameterization

facilitates training and outperforms the conditional parameterization from [137] for ternary weights. This verifies our intuition that the logit parameterization results in a well-behaved loss surface that is more amenable to optimization. For instance, an arbitrary weight probability can be increased by increasing the corresponding logit. This is more difficult to achieve for other parameterizations where the influence of the parameters on the corresponding probabilities is more entangled.

Furthermore, we improved on the stochastic max pooling approximation from Peters and Welling [138] by taking distributional properties explicitly into account. In our experiments, we compared several max pooling approximations and verified that it is important to also take distributional properties into account.

5.4.1 Limitations and Future Work

When varying the number of discrete weight values, we sometimes observed mixed results, especially for quaternary weights. We believe that this might be due to the missing but potentially important zero weight. Future work should address the role of the zero weight and different (non-symmetric) quantization levels for quaternary weights. Furthermore, discrete weights with trainable quantization levels might be promising to improve expressiveness and accuracy of the model.

As briefly mentioned in Section 5.1.3, our method indirectly parameterizes the means $\mathbb{E}[w]$ and the variances $\mathbb{V}[w]$. The number of parameters of the logit parameterization scales linearly in the number of discrete weight values Q. This might be prohibitive for many discrete weight values. Future work should explore the possibility of directly parameterizing $\mathbb{E}[w]$ and $\mathbb{V}[w]$. For instance, our reasoning in Section 5.1.3 suggests that only parameterizing the mean and selecting the smallest possible variance for that mean might be promising.

Although pre-training using a real-valued DNN is crucial, it is not well understood which properties distinguish a good pre-trained model. For instance, we observed that pre-trained models achieving a higher accuracy do not necessarily result in a better performing discrete-valued DNN. In our experiments, we obtained mixed results by varying the dropout rates. We believe that dropout plays an important role in this context. It also appears that dropout affects distribution training differently than training conventional real-valued DNNs. Furthermore, we did not investigate why initialization using ReLU pre-training outperforms initialization using tanh pre-training although ReLU exhibits a rather different functional shape than sign. We hypothesize that batch normalization might be an important factor for the successful transfer of ReLU features to sign features. A thorough experimental evaluation of our conjectures and open questions regarding initialization is left to future work.

Since our method crucially relies on a pre-trained model, our method is also related to transfer learning and knowledge distillation. A promising direction for future research is to investigate whether established techniques from these fields can be used to further improve the accuracy.

Our sampling experiments have shown that the distribution $q_{\boldsymbol{\nu}}(\mathbf{W})$ contains more information than its most probable weights. However, we did not evaluate predictions uncertainties obtained by averaging many DNNs sampled from $q_{\boldsymbol{\nu}}(\mathbf{W})$. We leave this to future work.

Applying our method to a wider range of architectures and datasets is an important direction of future research. In particular, since our method introduces a non-negligible computational overhead during training, scaling it to very large architectures and datasets such as ImageNet would be an important contribution.

Weight Sharing Using Dirichlet Processes

Besides variational inference, the second pillar of approximate Bayesian inference are sampling based methods. These methods generate samples from the induced posterior distribution $p(\mathbf{W} | \mathcal{D})$ after observing some dataset $\mathcal{D}$. The generated samples then represent a discrete approximation to the posterior distribution $p(\mathbf{W} | \mathcal{D})$. Subsequently, these samples are used to estimate expectations with respect to the posterior—a process also known as Monte Carlo integration or Monte Carlo averaging.

However, when applying sampling techniques to DNNs, we are facing several challenges. Generating weakly correlated or even independent samples for DNNs is difficult since the posterior $p(\mathbf{W} | \mathcal{D})$ inherits all the nonlinearity and multimodality properties from DNNs. The de facto standard methods to perform sampling for weight posteriors of DNNs are HMC and variants thereof [69, 70]. HMC is generally known to produce good samples and often serves as a golden standard for assessing the performance of novel Bayesian methods.

However, HMC is a time-consuming procedure that requires several gradient computations to generate a single sample. Moreover, the standard HMC algorithm operates in batch mode, i.e., the gradients must be computed using the whole dataset $\mathcal{D}$. Because of this it is often not possible to generate samples on the fly when they are needed or it is simply wasteful to discard these samples after generating them in a time-consuming procedure. Note that as discussed in Section 3.5, there exist some promising stochastic MCMC algorithms that operate on mini-batches [102, 103, 192, 193], but we focus on the standard HMC algorithm as it is less prone to random walk behavior. As a consequence, we are often required to precompute these samples offline and store them. This, however, might introduce large memory requirements, especially since a single DNN might be already quite large.

In this chapter, we set out the goal to reduce the memory consumption of an ensemble of fully connected DNN samples. For this purpose, we adopt the technique of weight sharing, i.e., different connections of the DNN share the exact same weight values. We refer the reader to Section 4.3.1 for a review of DNN methods relying on weight sharing. Sharing weights of a single DNN does not provide much benefit in terms of memory requirements if additionally the assignments of weights to their corresponding connections must be stored. Nevertheless, for an ensemble we can distribute the memory requirements for the weight assignments over many DNNs by using the same weight assignment multiple times and only varying the shared weights.

To achieve the weight sharing, we incorporate a DP prior [194] into our model. A DP is essentially a distribution over distributions. A DP has certain properties that make it appealing to achieve parameter sharing in various settings. We propose to introduce a DP prior over the weight distribution $p(\mathbf{W})$ which results in a weight sharing. On the downside, exact posterior inference using DPs is typically intractable and one is forced to use approximations which are often slow.

The proposed model maintains a set of weights $\mathbf{w}$ and weight assignment matrices $\mathbf{Z}$ that assign each connection of the DNN to a particular weight $w \in \mathbf{w}$. We propose to use a block Gibbs sampling scheme for inference in our model, i.e., we alternate between sampling the weights $\mathbf{w}$ conditioned on the weight assignments $\mathbf{Z}$ and sampling the weight assignments $\mathbf{Z}$ conditioned on the weights $\mathbf{w}$. We introduce algorithmic techniques and approximations that utilize the structure of DNNs to make posterior inference computationally tractable. Before

sampling new weight assignments $\mathbf{Z}$, we sample an *ensemble* of weights in order to distribute the memory requirements for the weight assignments $\mathbf{Z}$ over several weight samples.

In our experiments, we demonstrate the feasibility of our approach in various classification and regression experiments. Our model maintains a good prediction performance compared to Bayesian DNNs without weight sharing and DNNs trained with backpropagation while using only a fraction of the weights. The proposed method outperforms DNNs with random weight sharing on most datasets and on some datasets it even outperforms Bayesian DNNs without weight sharing. This indicates that our method has a regularizing effect and helps sampling based algorithms that typically scale poorly with the size of the sampling space, i.e., a large DNN can be used while operating in a low-dimensional weight space.

There exist different possibilities to achieve weight sharing in a DNN. For instance, Chen et al. [76] proposed to use a random weight sharing and to store the weight assignments implicitly using a hashing function. Other methods rely on a heuristic weight sharing, for instance, obtained by clustering the weights using the k-means algorithm [144]. Many models also employ a fixed weight sharing based on prior knowledge. Examples are CNNs where the weights are shared among different spatial locations, RNNs where the weights are shared among different time steps, and autoencoders where the weights of the decoder are often transposed versions of the encoder weights [22].

Our method is different in that the weight sharing is an inherent property of the model. As a consequence, the weight sharing emerges naturally as part of the Bayesian inference procedure and is adapted to the given data. Furthermore, our method does not require to specify the number of different weight values in advance. The number of weight values is only weakly determined by the choice of a concentration parameter α_{dp} governing the DP prior.

A related approach to reduce the memory for storing an ensemble of DNNs is presented by Korattikara et al. [170]. Their approach distills the knowledge contained in a sequence of MCMC samples in an online fashion into a single DNN. This avoids the need to store an ensemble of DNNs in the first place. However, their approach relies on stochastic MCMC methods that generate samples rapidly in order to perform SGD.

This chapter is largely based on our paper "Bayesian Neural Networks with Weight Sharing Using Dirichlet Processes" that has been published in the IEEE journal "Transactions on Pattern Analysis and Machine Intelligence" [159]. We begin our discussion in Section 6.1 with a brief review of DPs and one of their most common applications, i.e., mixture models with an unbounded number of components. In Section 6.2, we introduce our Bayesian DNN model using DPs and present our sampling based inference method for this model. We present our extensive experiments in Section 6.3 before we discuss our findings in Section 6.4.

6.1 Dirichlet Processes: A Distribution over Distributions

A DP is a distribution over distributions parameterized by a concentration parameter $\alpha_{\mathrm{dp}} > 0$ and a base distribution G_0. Ferguson [194] defined the DP in the following nonconstructive way. Given an arbitrary finite partition $(\mathcal{R}_1, \ldots, \mathcal{R}_M)$ of the space $\mathcal{R}$ on which G_0 is defined. G is drawn from a DP with parameters G_0 and α_{dp}, denoted as $G \sim \mathcal{DP}(G_0, \alpha_{\mathrm{dp}})$, if the vector of masses that G assigns to the subsets $\mathcal{R}_m$ follows a Dirichlet distribution with parameters $(\alpha_{\mathrm{dp}} G_0(\mathcal{R}_1), \ldots, \alpha_{\mathrm{dp}} G_0(\mathcal{R}_M))$.

By investigation of the following simple hierarchical Bayesian model, we can gain more insights into the practical utility of the DP. Consider a distribution G that is drawn from a DP and a dataset $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ that is obtained by drawing samples from the random distribution G, i.e.,

$$\begin{aligned} G &\sim \mathcal{DP}(G_0, \alpha_{\mathrm{dp}}) \\ \mathbf{x}_n &\sim G. \end{aligned} \tag{6.1}$$

By marginalizing out the random distribution G, the distribution over the samples $\mathbf{x}_n$ can be written as

$$\mathbf{x}_n \sim \frac{1}{n-1+\alpha_{\text{dp}}} \sum_{n'=1}^{n-1} \delta_{\mathbf{x}_{n'}} + \frac{\alpha_{\text{dp}}}{n-1+\alpha_{\text{dp}}} G_0, \tag{6.2}$$

where $\delta_{\mathbf{x}_{n'}}$ denotes a point mass located at $\mathbf{x}_{n'}$ [195]. Equation (6.2) shows that samples $\mathbf{x}_n$ drawn from G have a positive probability of being exactly equal to some previously drawn samples $\mathbf{x}_{n'}$ for $n' < n$. Note that this is true even if the base distribution G_0 is continuous, whereas drawing the exact sample multiple times from a continuous distribution has zero probability. As we will see in the remainder, this property of DPs enables the desired parameter sharing.

The distribution (6.2) has an intuitive interpretation in terms of a *Pólya urn* scheme [196]. Assume an urn that initially contains α_{dp} black balls. When a black ball is drawn, both the black ball and a new ball with a randomly generated color (corresponding to a draw from G_0) are returned to the urn. If a non-black ball is drawn, both the currently drawn ball and an additional ball of the same color are returned to the urn. This scheme implies a rich-get-richer principle where colors being drawn more often tend to be drawn more often in the future.

There exist other equivalent definitions of the DP that shed more light on the properties of the random distribution G drawn from a DP. The constructive *stick-breaking* definition of Sethuraman [197] shows that G can be represented as an infinite mixture of point masses. More specifically, a sample G from $\mathcal{DP}(G_0, \alpha_{\text{dp}})$ has the form $\sum_{k=1}^{\infty} \pi_k \delta_{\boldsymbol{\theta}_k}$ where

$$\boldsymbol{\theta}_k \sim G_0 \tag{6.3}$$

$$\pi_k = \xi_k \prod_{j=1}^{k-1} (1 - \xi_j), \quad \xi_k \sim \text{Beta}(1, \alpha_{\text{dp}}). \tag{6.4}$$

We denote that $\boldsymbol{\pi} = \{\pi_k\}_{k=1}^{\infty}$ is drawn according to (6.4) as $\boldsymbol{\pi} \sim \text{GEM}(\alpha_{\text{dp}})$.[21] The term "stick-breaking" originates from viewing (6.4) as successively breaking off a random proportion (determined by a draw from $\text{Beta}(1, \alpha_{\text{dp}})$) from a stick and assigning it to the mixture probabilities π_k. By starting from a stick of unit length, the mixture probabilities π_k will sum to one and a valid mixture distribution is obtained. The stick-breaking definition makes the discreteness of G explicit which is utilized to achieve parameter sharing in various settings.

However, storing an infinite number of mixture components as required by the stick-breaking definition is impossible. Fortunately, when dealing with a finite amount of data, it suffices to store only those components that are assigned to at least one data sample. Before showing how DPs are useful in obtaining shared weights in DNNs, we illustrate practical inference algorithms for one of their most prominent use cases: DP mixtures, i.e., mixture models with an unbounded number of components.

6.1.1 Dirichlet Process Mixtures

We consider mixture models of distributions of the form $F(\boldsymbol{\theta})$ to model the distribution over some observations $\mathcal{D} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$, where $\boldsymbol{\theta}$ are the free parameters governing the distribution F. For instance, in the case of GMMs we have $F(\boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ and $\boldsymbol{\theta} = (\boldsymbol{\mu}, \boldsymbol{\Sigma})$. The DP mixture model is obtained by adding an additional level to the hierarchy of the model specified by (6.1) to obtain

$$\begin{aligned} G &\sim \mathcal{DP}(G_0, \alpha_{\text{dp}}) \\ \boldsymbol{\theta}_n &\sim G \\ \mathbf{x}_n &\sim F(\boldsymbol{\theta}_n). \end{aligned} \tag{6.5}$$

[21] The term GEM was introduced in [198] and stands for Griffiths, Engen, and McCloskey.

The corresponding graphical model is shown in Figure 6.1(a). In this model each observation $\mathbf{x}_n$ is associated with its *individual* component parameters $\boldsymbol{\theta}_n$. Nevertheless, since the component parameters are drawn from a random distribution G drawn from a DP, several of the component parameters $\boldsymbol{\theta}_n$ will be shared with positive probability. It is then natural to define a component or a cluster as those samples $\mathbf{x}_n$ that are assigned equal component parameters $\boldsymbol{\theta}_n$.

However, it would be wasteful to store the exact same parameters $\boldsymbol{\theta}_n$ multiple times and most algorithms are based on a different equivalent view of DP mixtures. To develop a practical algorithm, it is convenient to consider the limit of $K \to \infty$ for finite mixture models with K components [195, 199]. We consider a Bayesian formulation of mixtures of K distributions of the form $F(\boldsymbol{\theta})$. Each component is associated with its own component parameters $\boldsymbol{\theta}_k$, and each observation $\mathbf{x}_n$ is associated with an indicator $z_n \in \{1, \ldots, K\}$ that specifies the component from which it was generated, i.e., $\mathbf{x}_n \sim F(\boldsymbol{\theta}_{z_n})$. The indicators z_n are drawn from a discrete distribution with mixture probabilities $\boldsymbol{\pi}$. We assume a prior distribution $\boldsymbol{\theta}_k \sim G_0$ and a symmetric Dirichlet prior $\boldsymbol{\pi} \sim \text{Dirichlet}(\alpha_{\text{dp}}/K, \ldots, \alpha_{\text{dp}}/K)$ over the mixture probabilities. Here, we assume that G_0 and α_{dp} are fixed, but we note that hyperpriors over them are considered in [199]. The model can be summarized as

$$\boldsymbol{\pi} \sim \text{Dirichlet}(\alpha_{\text{dp}}/K, \ldots, \alpha_{\text{dp}}/K) \tag{6.6}$$

$$z_n \sim \text{Discrete}(\boldsymbol{\pi}) \tag{6.7}$$

$$\boldsymbol{\theta}_k \sim G_0 \tag{6.8}$$

$$\mathbf{x}_n \sim F(\boldsymbol{\theta}_{z_n}). \tag{6.9}$$

In this model we can analytically integrate out the mixture probabilities $\boldsymbol{\pi}$ to obtain a prior over the indicators z_n as a product of conditional priors

$$p(\mathbf{z}) = \prod_{n=1}^{N} p(z_n \,|\, z_1, \ldots, z_{n-1}), \tag{6.10}$$

where the individual factors are given by

$$p(Z_n = k \,|\, z_1, \ldots, z_{n-1}) = \frac{N_{n,k} + \alpha_{\text{dp}}/K}{n - 1 + \alpha_{\text{dp}}}. \tag{6.11}$$

Here we have defined $N_{n,k} := |\{n' : z_{n'} = k, n' < n\}|$ to be the number of indicators $z_{n'}$ for $n' < n$ that are assigned to component k.

The DP mixture model is obtained by considering the limit $K \to \infty$. In this case the Bayesian model is updated by replacing (6.6) with

$$\boldsymbol{\pi} \sim \text{GEM}(\alpha_{\text{dp}}). \tag{6.12}$$

The corresponding graphical model is shown in Figure 6.1(b). The conditional prior (6.11) obtained by integrating out the mixture probabilities $\boldsymbol{\pi}$ is given by

$$p(Z_n = k \,|\, z_1, \ldots, z_{n-1}) = \begin{cases} \frac{N_{n,k}}{n-1+\alpha_{\text{dp}}} & N_{n,k} \geq 1 \\ \frac{\alpha_{\text{dp}}}{n-1+\alpha_{\text{dp}}} & N_{n,k} = 0. \end{cases} \tag{6.13}$$

From (6.13) we can see that the probability of assigning $z_n = k$ is proportional to the number of $z_{n'}$ for $n' < n$ that are already assigned to k. With probability proportional to α_{dp}, z_n will be assigned to a new value that is different from all the currently assigned $z_{n'}$. Note that by comparing (6.13) and the Pólya urn scheme (6.2), we can see that our original view of DP mixtures (6.5) is equivalent to considering the limit $K \to \infty$ for finite mixture models.

The particular form of the conditional prior (6.13) is called a Chinese restaurant process

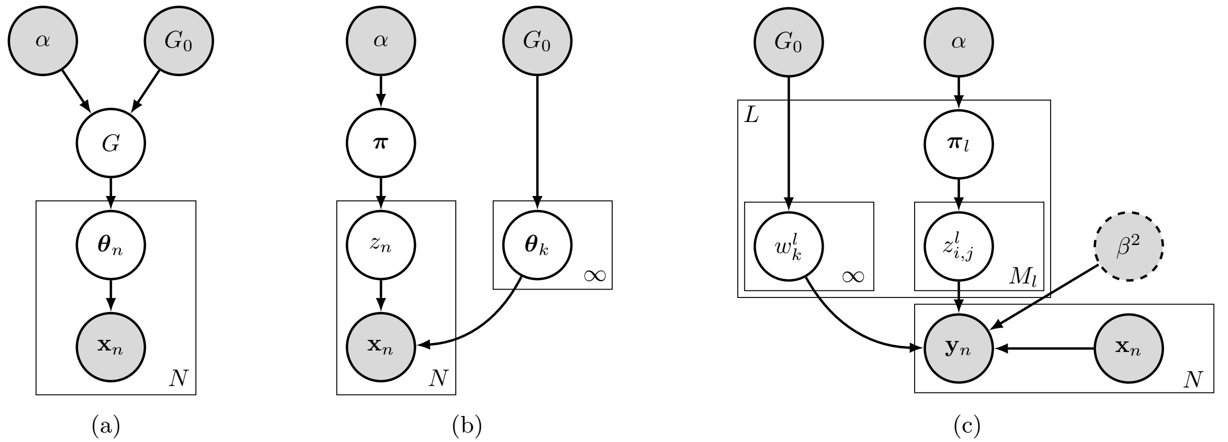


Figure 6.1: BN illustration of various models. Observed quantities are represented as shaded circles. (a) Per-sample parameter view of DP mixtures. Each sample is associated with its own component parameters $\boldsymbol{\theta}_n$. Due to the discreteness of DPs, some of the parameters $\boldsymbol{\theta}_n$ will be shared. (b) Explicit parameter view of DP mixtures. The distinct component parameters $\boldsymbol{\theta}_k$ are explicitly modeled. Each sample is associated with an indicator z_n that links to a particular component k. (c) DP DNN model with layerwise weight sharing. The dashed circle indicates that β^2 is only relevant for regression. The prior weight variance γ^2 has been absorbed into G_0 and is not shown explicitly. For global weight sharing, the dependency on l is dropped.

(CRP) due to an analogy with Chinese restaurants that seemingly have an infinite supply of tables. The analogy is typically illustrated in terms of a restaurant with an infinite number of tables and customers that enter the restaurant in sequence [200]. The first customer enters the restaurant and occupies a table. The following customers enter the restaurant and either sit at a table with probability proportional to the number of customers already sitting at that table or they occupy a new table with probability proportional to α_{dp}. This process specifies a distribution over partitions of the customers, i.e., who is sitting together with whom at a table.

A useful property of the CRP—also known as exchangeability—is that the resulting partition does not depend on the particular order in which the customers arrive. Many practical algorithms exploit this property as it allows us to assume an arbitrary order in which the customers arrive. Exchangeability can be easily verified by noting that the right hand side of (6.13) only depends on the numbers of z_n assigned to the same components k and not on the particular values of z_n.

It is easily seen that the number of components as specified by (6.13) is unbounded as there is always a positive probability of creating a new component. The overall number of clusters is governed by the concentration parameter α_{dp}, i.e., larger values of α_{dp} induce a higher probability of creating a new component. It can be shown that the asymptotic growth of the number of components is logarithmic in the number of customers [200]. Note that due to the unbounded number of components K also the numbers of component parameters $\boldsymbol{\theta}_k$ becomes unbounded. It turns out that this does not imply any technical difficulties for practical algorithms since it suffices to store only those $\boldsymbol{\theta}_k$ that are currently assigned by some indicator z_n.

6.1.2 Bayesian Inference for Dirichlet Process Mixtures

Performing exact Bayesian inference in a DP mixture requires summing over all possible assignments of z_n that result in different partitions of the data samples $\mathbf{x}_n$. This summation, however, becomes intractable quickly and we have to resort to approximations for any moderate number of samples N. In this section, we discuss three widely applicable approximate Bayesian inference algorithms based on MCMC that are also discussed by Neal in [195]. The presented methods are Gibbs sampling algorithms that update one indicator z_n at a time while keeping the remaining

variables fixed. We refer to Section 3.2.4 for a more detailed discussion on sampling methods and Gibbs sampling in particular.

The first and the second method are restricted to conjugate models, i.e., the base distribution G_0 is conjugate to the likelihood $F(\boldsymbol{\theta})$ (see Section 3.1.1). The third method is also applicable to non-conjugate models. The discussed methods operate on the unbounded mixture model discussed in the previous section where the mixture probabilities $\boldsymbol{\pi}$ have been integrated out to obtain the conditional prior (6.13). The state of the model is determined by the set of indicator variables $\mathbf{z} = \{z_1, \ldots, z_N\}$ and the mixture parameters $\boldsymbol{\Theta} = \{\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_K\}$.

We emphasize that in this model the specific values of the indicators z_n are not relevant and they are only significant in that they determine whether z_n and $z_{n'}$ for $n \neq n'$ are equal or not. For simplicity, we assume that the indicators z_n take values from $\{1, \ldots, K\}$ where K denotes the number of currently assigned components, i.e., for each k with $1 \leq k \leq K$ there exists at least one $z_n = k$. An indicator z_n is called a *singleton* if there exists no other indicator $z_{n'} = z_n$ for $n' \neq n$. As we will see, singletons often require special treatment in inference algorithms.

For the first and the second method, we assume a conjugate model that permits computation of the required integrals and posterior distributions in closed form. Furthermore, we assume that G_0 is from a family from which it is easy to draw samples from. This implies that we can easily draw samples from the induced posterior distributions after observing data.

Method 1: The first method proceeds by alternately sampling the indicators $\mathbf{z}$ and the mixture parameters $\boldsymbol{\Theta}$. To sample from the distribution of a single indicator z_n conditioned on the remaining variables of the current state, we can exploit exchangeability of the CRP and assume that z_n was the last observation. This allows us to express the prior of z_n conditioned on all the other $z_{n'}$ for $n' \neq n$ in the form of the last factor $n = N$ of (6.13). Let $\mathbf{z}_{-n} = \mathbf{z} \setminus \{z_n\}$ be the set of indicators without z_n, and let $N_{-n,k} = |\{n' : z_{n'} = k, n' \neq n\}|$ be the number of indicators without z_n that are currently assigned to component k. Then the required conditional probability is proportional to

$$p(Z_n = k \,|\, \mathbf{z}_{-n}, \boldsymbol{\Theta}) \propto \begin{cases} N_{-n,k} F(\mathbf{x}_n \,|\, \boldsymbol{\theta}_k) & k \leq K \\ \alpha_{\mathrm{dp}} \int F(\mathbf{x}_n \,|\, \boldsymbol{\theta}) \, G_0(\boldsymbol{\theta}) \, d\boldsymbol{\theta} & k \text{ unassigned.} \end{cases} \tag{6.14}$$

Given our conjugacy assumption, we can compute the integral for unassigned k in (6.14) analytically.

By repeatedly sampling indicator variables z_n according to (6.14), we expect to see new mixture components arise from time to time. Moreover, we also expect to see that existing components vanish whenever a singleton indicator is assigned to a different already existing component. In fact, if z_n is a singleton it will always be assigned to a different component since $N_{-n,z_n} = 0$. In both cases, emerging components and vanishing components, we additionally have to take care of the corresponding component parameters $\boldsymbol{\theta}_k$. If a singleton component vanishes, it suffices to discard the corresponding $\boldsymbol{\theta}_k$. If a new component emerges, a corresponding component parameter $\boldsymbol{\theta}_k$ is created by sampling from the posterior distribution obtained by observing only $\mathbf{x}_n$, i.e.,

$$\boldsymbol{\theta}_k \sim p(\boldsymbol{\theta}_k \,|\, \mathbf{x}_n) \propto G_0(\boldsymbol{\theta}_k) \, F(\mathbf{x}_n | \boldsymbol{\theta}_k). \tag{6.15}$$

Note that due to our conjugacy assumption above, evaluating the distribution (6.15) and generating samples from it is tractable.

Once the indicators $\mathbf{z}$ have been sampled, it remains to sample the component parameters $\boldsymbol{\Theta}$. The conditional distribution of a component parameter $\boldsymbol{\theta}_k$ depends only on those $\mathbf{x}_n$ that are currently assigned to component k by their corresponding indicators z_n, i.e.,

$$p(\boldsymbol{\theta}_k \,|\, \mathbf{z}) \propto G_0(\boldsymbol{\theta}_k) \prod_{n: z_n = k} F(\mathbf{x}_n \,|\, \boldsymbol{\theta}_k). \tag{6.16}$$

Again due to our conjugacy assumption, the left hand side of (6.16) will be of the same family as G_0 and sampling $\boldsymbol{\theta}_k$ is tractable. By iterating these two steps, sampling the indicators $\mathbf{z}$ conditioned on the component parameters $\boldsymbol{\Theta}$ and sampling the component parameters $\boldsymbol{\Theta}$ conditioned on the indicators $\mathbf{z}$, we recover what is called *Algorithm 2* in Neal's paper [195].

Method 2: The second method is an improved version of *Method 1*, called *Algorithm 3* in [195], and is obtained by marginalizing out the component parameters $\boldsymbol{\Theta}$ completely. This results in a special instance of a collapsed Gibbs sampling algorithm where certain dimensions are marginalized out to enable faster progress in state space during sampling.[22] The resulting state space contains only the indicator variables $\mathbf{z}$ and, therefore, only determines which of the z_n are grouped together without explicitly specifying the component parameters $\boldsymbol{\Theta}$. In this case, the required conditional distribution for Gibbs sampling is given by

$$p(Z_n = k \,|\, \mathbf{z}_{-n}) \propto \begin{cases} N_{-n,k} \int F(\mathbf{x}_n \,|\, \boldsymbol{\theta}) p(\boldsymbol{\theta} \,|\, \{\mathbf{x}_{n'} \,|\, z_{n'} = k, n' \neq n\})\, d\boldsymbol{\theta} & k \leq K \\ \alpha_{\mathrm{dp}} \int F(\mathbf{x}_n \,|\, \boldsymbol{\theta}) G_0(\boldsymbol{\theta})\, d\boldsymbol{\theta} & k \text{ unassigned.} \end{cases} \tag{6.17}$$

Again by our conjugacy assumption, the resampling probabilities in (6.17) are available in closed form. Note that if the corresponding component parameters $\boldsymbol{\theta}_k$ are required explicitly, they can at any time be obtained by sampling $\boldsymbol{\theta}_k$ from the posterior (6.16).

Method 3: The two methods discussed so far are only applicable to conjugate models. The third method, called *Algorithm 8* in [195], is also applicable to the non-conjugate case and belongs to the category of auxiliary variable samplers. More specifically, the algorithm performs sampling in a temporarily augmented state space to avoid the computation of the required integrals and posterior distributions that are generally intractable in the non-conjugate case. After sampling has finished, the auxiliary variables of the augmented state space are discarded.

The algorithm is provided with a user selected number of auxiliary variables $R \geq 1$ and only requires that we can efficiently generate samples from G_0. The first stage of the algorithm sequentially samples each indicator z_n according to the following procedure. Let $K_{-n} = |\{z_{n'} : n' \neq n\}|$ be the number of components obtained by ignoring sample n and assume that these components take values in $\{1, \ldots, K_{-n}\}$. Note that this potentially requires a relabeling step if z_n is a singleton. The state space is augmented by R auxiliary component parameters $\tilde{\boldsymbol{\Theta}} = \{\boldsymbol{\theta}_{K_{-n}+1}, \ldots, \boldsymbol{\theta}_{K_{-n}+R}\}$ that are currently unassigned. If z_n is *not* a singleton, the auxiliary component parameters $\{\boldsymbol{\theta}_{K_{-n}+1}, \ldots, \boldsymbol{\theta}_{K_{-n}+R}\}$ are generated by sampling from G_0. If z_n is a singleton, the current component parameters $\boldsymbol{\theta}_{z_n}$ are assigned to the first auxiliary component parameters $\boldsymbol{\theta}_{K_{-n}+1}$ and only the remaining $R-1$ component parameters $\{\boldsymbol{\theta}_{K_{-n}+2}, \ldots, \boldsymbol{\theta}_{K_{-n}+R}\}$ are sampled from G_0. The prior probability mass for unassigned components, α_{dp}, is split evenly among the R auxiliary parameters.

The new value of z_n is then sampled according to

$$p(Z_n = k \,|\, \mathbf{z}_{-n}, \boldsymbol{\Theta}, \tilde{\boldsymbol{\Theta}}) = \begin{cases} N_{-n,k} F(\mathbf{x}_n \,|\, \boldsymbol{\theta}_k) & 1 \leq k \leq K_{-n} \\ (\alpha_{\mathrm{dp}}/R) F(\mathbf{x}_n \,|\, \boldsymbol{\theta}_k) & K_{-n} < k \leq H, \end{cases} \tag{6.18}$$

where we have defined $H = K_{-n} + R$. After a new value of z_n has been sampled, the unassigned $\boldsymbol{\theta}_k$ are discarded again. As in the algorithms for conjugate models, this procedure allows for the emergence and disappearance of components in a principled manner.

The second stage of the algorithm samples the component parameters $\boldsymbol{\Theta}$ conditioned on the indicators $\mathbf{z}$. To do so, we sample from the posterior distributions (6.16) based on those samples $\mathbf{x}_n$ that are assigned to the same component. However, for the non-conjugate case,

[22] Strictly speaking, *Algorithm 2* from [195] is also a collapsed Gibbs sampler since the mixture probabilities $\boldsymbol{\pi}$ have been marginalized out.

these posterior distributions are generally not of the same form as the base distribution G_0. Therefore, one often must resort to more generally applicable sampling methods such as slice sampling [65], HMC [69, 70], or any other application-specific update scheme that leaves the stationary distribution invariant.

The three discussed methods are generally applicable but suffer from some problems that have been addressed in follow-up work. One major drawback of Gibbs sampling based algorithms that update only a single indicator z_n at a time is that they are prone to getting trapped in bad local modes if, for instance, different components exhibit similar component parameters. To merge or split several such components, the incremental nature of Gibbs sampling requires a sequence of update steps traversing through a highly unlikely region in state space. This issue has been explicitly addressed by approaches employing split-merge proposals in a Metropolis-Hastings scheme to make faster progress in state space [201]. However, these methods rely on the exact computation of specific integrals and, therefore, are typically restricted to the conjugate case.

The DP has been generalized to the *hierarchical DP* by adding an additional layer to the hierarchical Bayesian model [202]. Given a set of datasets, the hierarchical DP allows us to model each of these datasets with its individual unbounded mixture model while allowing some of the component parameters to be shared among mixture models of different datasets.

Besides sampling based methods, there also exist various approximate inference methods based on variational inference. For instance, Blei and Jordan [203] proposed a variational inference algorithm based on the explicit stick-breaking construction. Their approach relies on a truncated stick-breaking representation of the variational approximation using a fixed maximum number of components.

In the next Section, we adapt *Method 3* (Neal's *Algorithm 8* [195] for non-conjugate models) to sample assignments of weights to individual connections in a DNN. We also highlight some specific properties of our DNN model that make the computational complexity of posterior inference substantially more challenging compared to the case of DP mixtures.

6.2 Dirichlet Process Neural Networks

We assume a Bayesian treatment of DNNs as introduced in Chapter 3. In particular, we assume a prior distribution over the weights, $p(\mathbf{W})$. The likelihood is specified by the DNN function and the given task (regression or classification). We refer to Section 3.2.1 for a detailed discussion on the likelihood defined by a DNN. We note that regression introduces an additional hyperparameter β^2 specifying the output variance. We restrict ourselves to fully connected layers and we assume that biases are included in the weight matrices. Furthermore, we only consider vanilla architectures as introduced in Section 2.2.1, i.e., we do not use batch normalization or any other more sophisticated techniques.

To obtain weight sharing, we assume that the weights of the DNN are independently drawn from a distribution $G := p(\mathbf{W})$ which is itself drawn from a DP with concentration parameter α_{dp} and base distribution G_0. The employed inference algorithm is based on Neal's *Algorithm 8* [195] for non-conjugate models and, therefore, only requires that we can efficiently draw samples from G_0. Throughout this chapter, we assume that G_0 is a zero-mean Gaussian with variance γ^2. The extension to arbitrary base distributions is straightforward.

There are essentially two ways to use the DP prior. First, we can assign a DP prior over a single global weight prior $p(\mathbf{W})$ for the weights in all layers of the DNN. This allows weights to be shared between different layers of the DNN. Second, we can assign a separate DP prior over individual layerwise weight priors $p(\mathbf{W}^l)$ such that weights are only shared within a layer. We refer to the former as *global* sharing and to the latter as *layerwise* sharing. Throughout

this chapter we employ layerwise sharing since we empirically observed better results using this approach.

For layerwise sharing, the model is defined as follows. The weights are given by $\mathbf{w} = \{\mathbf{w}^1, \dots, \mathbf{w}^L\}$ where $\mathbf{w}^l \in \mathbb{R}^{K_l}$ stores the K_l currently assigned weights in layer l. The weight indicators are given by $\mathbf{Z} = \{\mathbf{Z}^1, \dots, \mathbf{Z}^L\}$ where $\mathbf{Z}^l \in \{1, \dots, K_l\}^{d_l \times d_{l-1}}$ contains the indicators of all connections in layer l. Note that K_l depends on the indicators $\mathbf{Z}^l$ and may change over time. The full weight matrices $\tilde{\mathbf{W}}^l$ of the DNN are recovered as $\tilde{w}^l_{i,j} = w^l_{z^l_{i,j}}$. The number of connections in layer l is denoted as $M_l := |\mathbf{Z}^l|$ and the overall number of connections in the DNN is denoted as $M := \sum_{l=1}^{L} M_l$. In the context of DNNs, we also call a concrete instantiation of the weight indicators $\mathbf{Z}$ a *configuration*. The full model with layerwise sharing is summarized as

$$\boldsymbol{\pi}_l \sim \mathrm{GEM}(\alpha_{\mathrm{dp}}) \tag{6.19}$$

$$z^l_{i,j} \sim \mathrm{Discrete}(\boldsymbol{\pi}_l) \tag{6.20}$$

$$w^l_k \sim G_0 := \mathcal{N}(0, \gamma^2) \tag{6.21}$$

$$\mathbf{y}_n \sim p(\cdot \,|\, \mathbf{x}_n, \mathbf{Z}, \mathbf{w}\ [, \beta^2]), \tag{6.22}$$

where the square brackets in (6.22) indicate that β^2 is only relevant for regression. The model is illustrated in Figure 6.1(c). In this chapter, we assume that the hyperparameters α_{dp}, β^2 and γ^2 are fixed, but the model can be extended to include priors over them.

6.2.1 Posterior Inference in Dirichlet Process Neural Networks

Posterior inference using the joint distribution of the weights and the configuration $p(\mathbf{w}, \mathbf{Z} | \mathcal{D})$ is a challenging task.[23] On the one hand, given a fixed configuration, the conditional posterior over the weights $p(\mathbf{w} | \mathbf{Z}, \mathcal{D})$ is typically complicated and highly multimodal. On the other hand, given a fixed set of weights $\mathbf{w}$, there is an intractable number of configurations $\mathbf{Z}$ to consider.

Rather than searching for a single assignment of the weights $\mathbf{w}$ and the configuration $\mathbf{Z}$, we propose to use sampling techniques for inference. Our sampling based inference scheme is a block Gibbs scheme where we alternate between sampling from the posterior of the configuration given the weights $p(\mathbf{Z} | \mathbf{w}, \mathcal{D})$ and sampling from the posterior of the weights given the configuration $p(\mathbf{w} | \mathbf{Z}, \mathcal{D})$

Sampling from the Configuration Posterior

To sample from the configuration posterior $p(\mathbf{Z} | \mathbf{w}, \mathcal{D})$, we adapt Neal's auxiliary variable Gibbs sampling scheme for DP mixtures (*Algorithm 8* in [195]) that we have discussed in detail as *Method 3* in Section 6.1.2. Note that due to the highly nonlinear likelihood defined by the output of a DNN, we are restricted to algorithms suitable for non-conjugate models.

To adapt Neal's algorithm for DP mixtures to our DNN model, we must address several differences between DP mixtures and our Bayesian DNN model. In a DP mixture model, each indicator z_n is associated to a single sample $\mathbf{x}_n$ and the overall number of indicators grows with the number of samples N. The likelihood for each sample $\mathbf{x}_n$ is independent from the indicators of the remaining samples $\mathbf{z}_{-n}$. As a result, the conditional distribution over z_n (6.18) for Gibbs sampling only requires the evaluation of the likelihood for the associated sample $\mathbf{x}_n$ and the remaining samples can be ignored.

The situation is different for DNNs. Here each indicator z_m is associated to a connection of the DNN. As a result, the overall number of indicators can be considered fixed as it depends

[23] Here and in the remainder we will sometimes omit the dependency of distributions on the hyperparameters α_{dp}, β^2, and γ^2 for brevity.

Algorithm 8 Sampling the weight indicator z_m^l (based on Neal's *Algorithm 8* [195])

1: **Input:** $\mathcal{D}, \mathbf{z}, \mathbf{w}, R, \alpha_{\text{dp}}, \gamma^2$ $[, \beta^2]$
2: $K_{-m} \leftarrow |\{z_{m'}^l : m' \neq m\}|$ # number of components without z_m^l
3: $H \leftarrow K_{-m} + R$
4: **if** $|\{m' : z_{m'}^l = z_m^l, m' \neq m\}| = 0$ **then** # is z_m^l currently a singleton?
5: Rearrange $\mathbf{z}^l$ and $\mathbf{w}^l$ such that $z_m^l = K_{-m} + 1$
6: Draw $w_k \sim G_0(\gamma^2)$ for $K_{-m} + 1 < k \leq H$
7: **else**
8: Draw $w_k \sim G_0(\gamma^2)$ for $K_{-m} < k \leq H$
9: **end if**
10: **for** $k = 1$ **to** H **do**
11: **if** $k \leq K_{-m}$ **then**
12: $\rho \leftarrow |\{m' : z_{m'}^l = k, m' \neq m\}|$
13: **else**
14: $\rho \leftarrow \alpha_{\text{dp}}/R$
15: **end if**
16: $p_k \leftarrow \rho \prod_{n=1}^N p(\mathbf{y}_n | \mathbf{x}_n, \mathbf{z}_{-m}, z_m^l = k, \mathbf{w}[, \beta^2])$
17: **end for**
18: $\boldsymbol{p} \leftarrow (p_1, \ldots, p_H) / \sum_{k=1}^H p_k$
19: Draw $z_m^l \sim \text{Discrete}(\boldsymbol{p})$

on the number of connections M in the given DNN architecture. More importantly, the likelihood for each sample $(\mathbf{x}_n, \mathbf{y}_n)$ depends on *every* indicator z_m. Therefore, to perform Gibbs sampling for a particular indicator z_m, we must evaluate the likelihood for the entire dataset $\mathcal{D}$. Although this requires only minor conceptual adaptations to Neal's algorithm, the implied practical consequences are severe. In particular, for each connection m and for each currently assigned weight $w \in \mathbf{w}$, the output of the DNN for the whole dataset has to be computed. This makes the sampling process computationally expensive. As we will see in the remainder, a careful implementation that exploits structural properties of DNNs is required to obtain a practical algorithm.

The pseudocode for sampling a single weight indicator z_m^l for $m = (i, j)$ is shown in Algorithm 8. The weight of connection m is replaced by all of the K_{-m} currently assigned weights $w \in \mathbf{w}$ and $R \geq 1$ additional auxiliary weights drawn from the base distribution G_0 (line 8). In case connection m is currently assigned a singleton weight, one of the R auxiliary variables is assigned the current weight $w_{z_m^l}$ and only $R - 1$ of them are drawn from G_0 (lines 5–6). For each of the $H \coloneqq K_{-m} + R$ weight replacements, a conditional probability p_k is computed which is subsequently used to sample the new weight indicator z_m^l (lines 10–19).

Our algorithm cycles through all connections of the DNN and updates their weight indicators with Gibbs sampling according to Algorithm 8. In Section 6.2.2 we show approximations and approaches to avoid many redundant computations.

Sampling from the Weight Posterior

After updating the weight indicators $\mathbf{Z}$ for all connections of the DNN, we propose to use HMC [69, 70] to sample from the conditional distribution of the weights $p(\mathbf{w} | \mathbf{Z}, \mathcal{D})$. HMC comes with two advantages. (i) It updates all variables simultaneously rather than sampling the individual weights in turn as in Gibbs sampling. (ii) It uses gradient information of the log-density to explore the state space more systematically.

A drawback of plain HMC is that it requires the selection of two parameters, T and η, that are critical in achieving good performance. This shortcoming is addressed by the AHMC algorithm

which finds suitable values for T and η automatically [72]. To do so, AHMC performs Bayesian optimization [204] to maximize the normalized expected squared jumping distance between consecutive samples.

The conditional density of the weights in our model is proportional to

$$p(\mathbf{w}|\mathbf{Z}, \mathcal{D}) \propto \prod_{n}^{N} p(\mathbf{y}_n|\mathbf{x}_n, \mathbf{Z}, \mathbf{w}[, \beta^2]) \prod_{l}^{L} \prod_{k}^{K_l} G_0(w_k^l|\gamma^2). \tag{6.23}$$

The gradient of the logarithm of (6.23) required for HMC is easily computed using automatic differentiation. For a detailed discussion on HMC we refer to Section 3.2.4.

6.2.2 Computational Tricks and Inference Complexity

As already mentioned, the computational cost of Algorithm 8 is high. Nevertheless, changing the weight of a single connection is only a local change to the DNN. In the following, we introduce techniques to avoid computing a full forward pass each time a weight indicator z_m is replaced.

Likelihood Interpolation at Neurons

We introduce an approximation that reduces the number of full DNN evaluations drastically. The main computational cost of Algorithm 8 arises in line 16 where for each of the H possible weights a forward pass of the whole dataset is computed. Since a single weight $w_{i,j}^l$ only influences the output x_i^l of the neuron to which it is connected, we can view the log-likelihood for the nth data sample as a function $f_n(x_i^l(w_{i,j}^l))$. We can precompute the log-likelihood $f_n(x_i^l)$ at several predetermined values $x_i^l \in \mathcal{X}_{\text{int}}$ and approximate $f_n(x_i^l(w_{i,j}^l))$ with interpolation rather than performing a computationally expensive forward pass each time $w_{i,j}^l$ is replaced. For brevity, we will omit the subscripts and superscripts in the remainder and only write $f(x(w))$.

Bounded activation functions: Assume that the activation function of the DNN has a bounded output with lower bound l and upper bound u. This is true for many commonly used sigmoidal activation functions such as tanh and sigm. Given a discretization parameter $s \geq 1$, we define a set $\mathcal{X}_{\text{int}} = \{l + k(u-l)/s : k = 0, \ldots, s\}$ of $s+1$ evenly spaced values between l and u at which we evaluate the log-likelihood terms $f_n(x)$ for all data samples. The parameter s controls the quality of the approximation; for $s \to \infty$ the approximation becomes exact.

ReLU activation function: Dividing the output range of the activation function into evenly spaced intervals is not possible for common *unbounded* activation functions such as the ReLU $h(a) = \max(0, a)$. However, we can assume that activations cluster around zero and do not grow arbitrarily large, assuming that the inputs and the weights are relatively small. For a given base parameter $b > 1$, an exponent $u \in \mathbb{Z}$, and a discretization parameter $s \geq 1$, we define a set $\mathcal{X}_{\text{int}} = \{0, b^u, \ldots, b^{u+s-1}\}$ of $s+1$ logarithmically spaced values at which the log-likelihood terms $f_n(x)$ are evaluated. For $b=2$ and $u=-3$, we obtain a fine-grained interpolation scheme in the vicinity of zero, resulting in small interpolation errors in most cases, while still allowing to capture a large scale of values. Furthermore, this scheme is exact for negative activations a. In the rare case an activation is larger than $\max \mathcal{X}_{\text{int}}$, we propose to use *extrapolation*.

The number of forward passes with likelihood interpolation reduces from $N \cdot H$ to $N \cdot (s+1)$ to precompute the interpolation coefficients. When sampling a weight indicator $z_{i,j}^l$, we interpolate N values for each of the H weight replacements. For nearest neighbor interpolation, this scheme has the interpretation as approximating a single activation function by a piecewise constant function (see Figure 6.2). However, in practice nearest neighbor interpolation is inferior to more

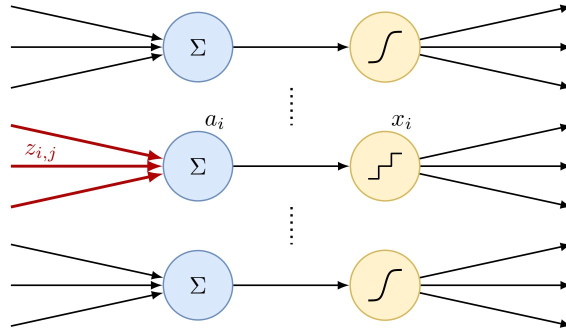


Figure 6.2: Likelihood interpolation using nearest neighbor interpolation can be seen as approximating a single activation function by a piecewise constant function. The DNN function is precomputed for every possible output x_i of the piecewise constant function and the results are stored in a lookup table. The lookup table is then reused to sample all (red) indicators $z_{i,j}$ feeding into neuron i, avoiding many expensive forward passes. Note, however, that nearest neighbor interpolation yields inferior results compared with linear and cubic interpolation.

sophisticated interpolation schemes. We propose to use cubic interpolation [205] which requires a memory overhead of $\mathcal{O}(N \cdot s)$ to store interpolation coefficients.

Gibbs Cycling Order

By using a specific order when cycling through the weight indicators $z^l_{i,j}$, we can keep the computational overhead manageable. First, we propose to sample one layer at a time, beginning with the first layer and progressing towards the last layer. Sampling a layer at a time has the advantage that the computation of the DNN up to previous layers stays unaffected and only needs to be computed once. Next, we propose to iterate through neurons i of the subsequent layer and to sample all connections (i, j) feeding into neuron i before progressing to another neuron $i' \neq i$. This allows us to reuse the precomputed interpolation coefficients for all weight indicators $z^l_{i,j}$ feeding into neuron i and to avoid many unnecessary forward passes.

Using likelihood interpolation and an appropriate Gibbs cycling order is crucial in order to make sampling from the posterior $p(\mathbf{Z} \mid \mathbf{w}, \mathcal{D})$ tractable. However, the need to use approximations also makes it hard to assess the impact of likelihood interpolation on the quality of the samples. In our experiments we found $s = 10$ to yield good results for both tanh and ReLU.

Incremental Activation Updates

The following method does not affect the number of forward passes but is worth considering for an efficient implementation of the configuration sampling algorithm. Consider updating weight indicator $z^l_{i,j}$ where neuron j is an input and neuron i is an output of layer l. Let $a^l_{i,\text{old}}$ be the activation of neuron i before applying the activation function h. When the current weight w_{old} is replaced by a new weight w_{new}, the activation can be computed as $a^l_{i,\text{new}} = a^l_{i,\text{old}} + x^{l-1}_j (w_{\text{new}} - w_{\text{old}})$. The activation function can then be applied to the updated value $a^l_{i,\text{new}}$ of the neuron. This avoids recomputing the sum of all inputs to neuron i and produces the new output $x^l_{i,\text{new}}$ in constant time.

Next, assume that $l < L$ and there exists another layer $l + 1$ to which neuron i is an input. An obvious, yet important, observation is that the outputs $a^l_{i'}$ of other neurons $i' \neq i$ in layer l are unaffected by changing the weight of connection (i, j). This implies that the activations in the next layer can be updated as $a^{l+1}_{k,\text{new}} = a^{l+1}_{k,\text{old}} + w^{l+1}_{k,i} (x^l_{i,\text{new}} - x^l_{i,\text{old}})$ for all neurons k in the next layer. This reduces the computational effort of computing a forward pass from layer l to layer $l + 1$ from a matrix-vector multiplication to a cheaper incremental update. However, from

layer $l+1$ to layer L, a full forward pass is necessary.

Running Time of Posterior Sampling

Since the running time of Algorithm 8 is largely determined by computing forward passes in the DNN, our analysis is restricted to counting the number of forward passes. Let $\bar{H}$ be the average number of weight replacements per connection. When implementing Algorithm 8 naïvely, the number of forward passes to sample all connections in layer l is $\mathcal{O}(N \cdot d_{l-1} \cdot d_l \cdot \bar{H})$. Using likelihood interpolation reduces the number of forward passes per connection from $\bar{H}$ to constant $s+1$. Moreover, using the proposed Gibbs cycling order we can reuse the interpolation coefficients for all d_{l-1} connections feeding into a neuron. This results in $\mathcal{O}(N \cdot d_l \cdot s)$ forward passes in total.

6.3 Experiments

We compare the performance of several models. We performed MAP training for plain feed-forward DNNs using the L-BFGS[24] quasi-Newton algorithm [49]. We did not perform stochastic mini-batch optimization since for sampling our main focus is on HMC which does not use mini-batches either. We used tanh (BFGS Tanh) and ReLU (BFGS ReLU) activation functions. We evaluated one and two hidden layers with $d_l \in \{50, 100, 250, 500, 1000\}$ hidden neurons. For two hidden layers we set $d_1 = d_2$. We optimized the weight variance $\gamma^2 \in \{10^{-2}, 10^{-1}, \dots, 10^2\}$ and selected random initial weights drawn from $\mathcal{N}(0, 10^{-2})$.

We evaluated Bayesian DNNs without weight sharing (BNN). We generated 5,000 sets of weights (i.e., 5,000 DNNs) with AHMC after discarding the first 200 DNNs as burn-in. We evaluated one and two hidden layers for $d_l \in \{50, 100\}$ and set $d_1 = d_2$ in case of two hidden layers. We optimized the weight variance $\gamma^2 \in \{10^{-2}, 10^{-1}, 10^0\}$ and initialized AHMC with a mode from the posterior distribution. The upper and lower bounds of AHMC used for Bayesian optimization were set to $b_l^T = 1$ and $b_u^T = 250$ for T and $b_l^\eta = 10^{-6}$ and $b_u^\eta = 10^{-1}$ for η. For more details on these parameters the interested reader is referred to [72].

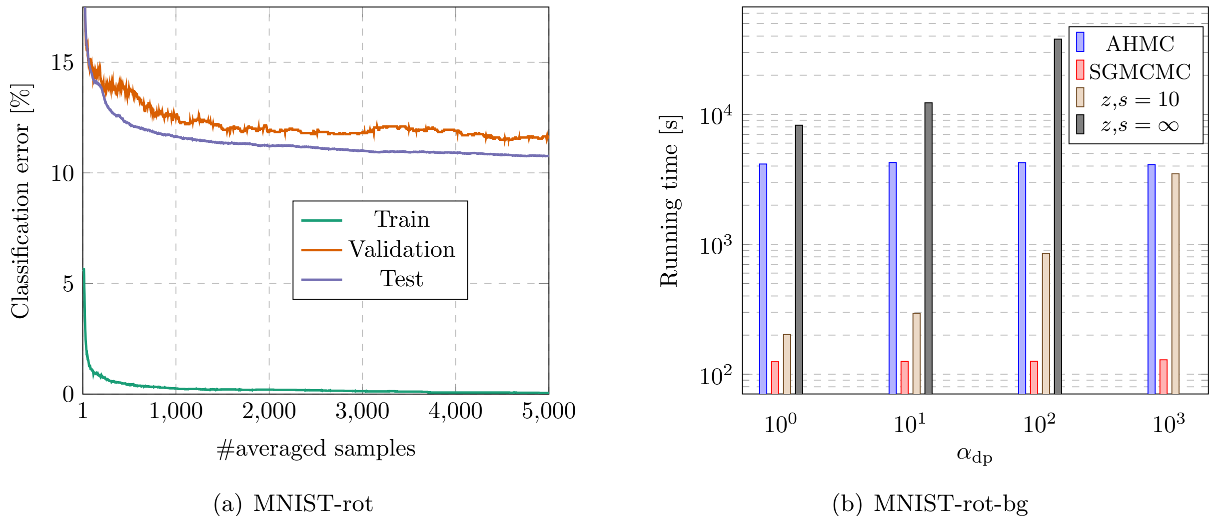
For our model (DP BNN), we evaluated the same DNN structures and the same prior variance γ^2 as for BNNs. We evaluated the DP parameter $\alpha_{\text{dp}} \in \{10^0, 10^1, 10^2, 10^3\}$. We randomly initialized the weight configuration using a CRP with parameter α_{dp} and performed 100 iterations of configuration sampling as burn-in. This adapts the configuration to the given data and results in better performance than starting from a random configuration. The weights were randomly initialized by sampling from $\mathcal{N}(0, \gamma^2)$. Then we generated 200 DNNs with AHMC followed by 24 iterations of alternating between sampling all weight indicators according to Algorithm 8 and sampling 200 sets of weights with AHMC.[25] This setting requires us to store only 25 configurations and for each configuration the 200 different sets of weights, resulting in an ensemble of 5,000 DNNs. We fixed the number of auxiliary variables $R = 100$ and the discretization parameter $s = 10$ for all experiments.

We also performed experiments with randomly shared weights (RND BNN). For each DNN structure, we generated 25 random configurations with the same number of weights as in the best performing DP BNN experiment. For each of the 25 configurations, we performed 400 iterations of AHMC and discarded the first 200 DNNs as burn-in, resulting in an overall number of 5,000 DNNs.

For regression, we also trained DNNs with dropout [9]. We used the same DNN structures as for BNNs. We evaluated the dropout probabilities $p_{\text{do}} \in \{0.005, 0.01, 0.05, 0.1\}$, the weight

[24] The BFGS method is named after its inventors Broyden, Fletcher, Goldfarb, and Shanno, and the L stands for *limited-memory*.

[25] AHMC actually generated 400 sets of weights but the first 200 were used as burn-in and for finding suitable HMC parameters, T and η, with Bayesian optimization.



(a) MNIST-rot

(b) MNIST-rot-bg

Figure 6.3: (a) Classification error [%] over number of averaged samples for DP BNN ReLU on MNIST-rot. (b) Average running time over α_{dp} on MNIST-rot-bg. We evaluated weight sampling (200 weight sets) using AHMC and SGMCMC. The running time differences between SGLD and SGHMC (shown as SGMCMC) are negligible. For configuration sampling (z), we evaluated a single Gibbs cycle for two values of s ($s = \infty$ corresponds to likelihood interpolation not *being used).*

decay $\gamma_{\mathrm{do}} \in \{10^{-5}, 10^{-4}, \ldots, 10^{1}\}$, and the output variance $\beta^2 \in \{10^{-2}, 10^{-1}\}$. The models were optimized using ADAM [11] with mini-batches of 100 samples. The step size is set to 10^{-3} and multiplied by 0.999 after every epoch. For the dropout experiments, the likelihood term of the objective is normalized by the number of samples N. As a result, the weight decay γ_{do} has a different interpretation compared to the prior weight variance γ^2 of BFGS optimized models, i.e., $1/\gamma^2 = N\gamma_{\mathrm{do}}$. As mentioned in [30], we can obtain predictive uncertainties for the DNN output by sampling several DNNs according to the dropout probabilities p_{do}. In our experiments, we sampled 5,000 DNNs.

6.3.1 Classification Results

For classification, we performed experiments on the MNIST dataset (see Appendix A.1) and variants of the MNIST dataset (see Appendix A.2). We whitened the data by transforming the input features with principal component analysis (PCA) to 50 dimensions as in [206] and normalizing the PCA features to zero mean and unit variance.

The best mean classification errors over five runs for each model are summarized in Table 6.1. The results are clearly in favor of the ReLU activation function. Nevertheless, we emphasize that DP BNN outperforms their RND BNN counterparts for both tanh and ReLU on most datasets. BFGS optimized DNNs achieve the best results only on MNIST-basic. We emphasize that the BFGS experiments favored larger models which we did not evaluate for the BNN models. In particular, on all datasets at least either BFGS Tanh or BFGS ReLU used the largest structure with two hidden layers and 1,000 neurons.

Interestingly, for tanh, DP BNN even outperforms BNN on four out of six datasets. This indicates that weight sharing has a regularizing effect on the DNNs, preventing them from over-fitting. Moreover, due to weight sharing, AHMC can sample in a lower-dimensional space which might improve the quality of the generated samples. Figure 6.3(a) shows how the classification error of a DP BNN ReLU with two hidden layers of 100 neurons evolves as the number of averaged DNN samples grows. The steep decrease at the beginning indicates a good decorrelation between consecutive samples.

	Tanh				ReLU			
Dataset	BFGS	BNN	RND BNN	DP BNN	BFGS	BNN	RND BNN	DP BNN
MNIST	1.74 ± 0.03	1.70 ± 0.02	1.78 ± 0.04	1.63 ± 0.03	1.55 ± 0.04	**1.44 ± 0.04**	1.70 ± 0.03	1.46 ± 0.06
MNIST-basic	4.13 ± 0.09	4.43 ± 0.07	4.82 ± 0.02	4.15 ± 0.05	**3.50 ± 0.07**	3.75 ± 0.02	3.74 ± 0.03	3.83 ± 0.06
MNIST-bg	21.42 ± 0.25	17.60 ± 0.05	18.28 ± 0.03	18.03 ± 0.10	18.69 ± 0.17	**16.55 ± 0.05**	17.51 ± 0.03	17.08 ± 0.03
MNIST-bg-rnd	9.25 ± 0.04	8.88 ± 0.03	9.27 ± 0.02	8.28 ± 0.03	8.31 ± 0.08	**7.64 ± 0.01**	7.89 ± 0.04	7.94 ± 0.07
MNIST-rot	11.74 ± 0.14	11.49 ± 0.12	12.19 ± 0.05	11.06 ± 0.07	13.93 ± 0.10	**10.41 ± 0.38**	13.74 ± 0.07	10.84 ± 0.14
MNIST-rot-bg	48.47 ± 0.24	41.41 ± 0.06	43.43 ± 0.09	42.98 ± 0.14	48.15 ± 0.41	**39.44 ± 0.16**	45.78 ± 0.10	42.33 ± 0.36

Table 6.1: Average test classification errors [%] and standard deviations over five runs on MNIST and variants thereof.

	Tanh			ReLU			
Dataset	BNN	RND BNN	DP BNN	Dropout	BNN	RND BNN	DP BNN
Abalone	-2.223 ± 0.021	-2.212 ± 0.024	**-2.199 ± 0.018**	-2.238 ± 0.031	-2.734 ± 0.048	-2.652 ± 0.038	-2.226 ± 0.020
Housing	-2.276 ± 0.031	-2.412 ± 0.027	**-2.271 ± 0.032**	-2.449 ± 0.045	-2.442 ± 0.024	-2.437 ± 0.025	-2.363 ± 0.089
Concrete	-2.727 ± 0.121	-2.828 ± 0.104	**-2.723 ± 0.131**	-2.978 ± 0.020	-2.903 ± 0.096	-2.881 ± 0.074	-2.731 ± 0.137
Power Plant	-2.847 ± 0.007	-2.859 ± 0.007	-2.848 ± 0.009	**-2.837 ± 0.005**	-2.859 ± 0.007	-2.865 ± 0.008	-2.860 ± 0.008
WineQ-red	-0.682 ± 0.020	**-0.577 ± 0.094**	-0.650 ± 0.075	-1.156 ± 0.045	-0.984 ± 0.035	-0.942 ± 0.029	-0.833 ± 0.021
WineQ-white	-0.786 ± 0.021	-0.909 ± 0.017	**-0.780 ± 0.016**	-1.154 ± 0.030	-1.544 ± 0.059	-1.273 ± 0.034	-0.947 ± 0.027
Abalone	2.397 ± 0.034	2.204 ± 0.018	2.326 ± 0.020	2.157 ± 0.051	2.654 ± 0.364	**2.123 ± 0.025**	2.446 ± 0.057
Housing	2.940 ± 0.158	3.108 ± 0.192	2.946 ± 0.163	3.019 ± 0.158	2.902 ± 0.133	2.906 ± 0.136	**2.792 ± 0.137**
Concrete	**3.998 ± 0.106**	4.267 ± 0.144	4.003 ± 0.117	4.787 ± 0.164	4.487 ± 0.175	4.521 ± 0.182	4.076 ± 0.214
Power Plant	**3.647 ± 0.052**	3.757 ± 0.057	3.659 ± 0.070	3.676 ± 0.061	3.764 ± 0.055	3.811 ± 0.057	3.771 ± 0.063
WineQ-red	**0.613 ± 0.011**	0.617 ± 0.012	0.619 ± 0.015	0.616 ± 0.014	0.698 ± 0.018	0.675 ± 0.021	0.661 ± 0.017
WineQ-white	**0.623 ± 0.011**	0.658 ± 0.012	**0.623 ± 0.009**	0.648 ± 0.009	0.732 ± 0.015	0.671 ± 0.011	0.681 ± 0.012

Table 6.2: Average test log-likelihoods and standard deviations (top) and average test root mean squared errors and standard deviations (bottom) on various UCI regression datasets obtained using 5-fold cross-validation.

Dataset	DP BNN SGLD ReLU	DP BNN SGHMC ReLU
MNIST	1.49 ± 0.05	1.48 ± 0.05
MNIST-basic	3.85 ± 0.11	3.82 ± 0.07
MNIST-bg	17.22 ± 0.10	17.17 ± 0.12
MNIST-bg-rnd	8.00 ± 0.06	7.99 ± 0.05
MNIST-rot	10.92 ± 0.11	10.98 ± 0.23
MNIST-rot-bg	41.43 ± 0.10	41.61 ± 0.12

Table 6.3: Average test classification errors [%] and standard deviations over five runs of stochastic MCMC methods for weight sampling. We evaluated SGLD and SGHMC.

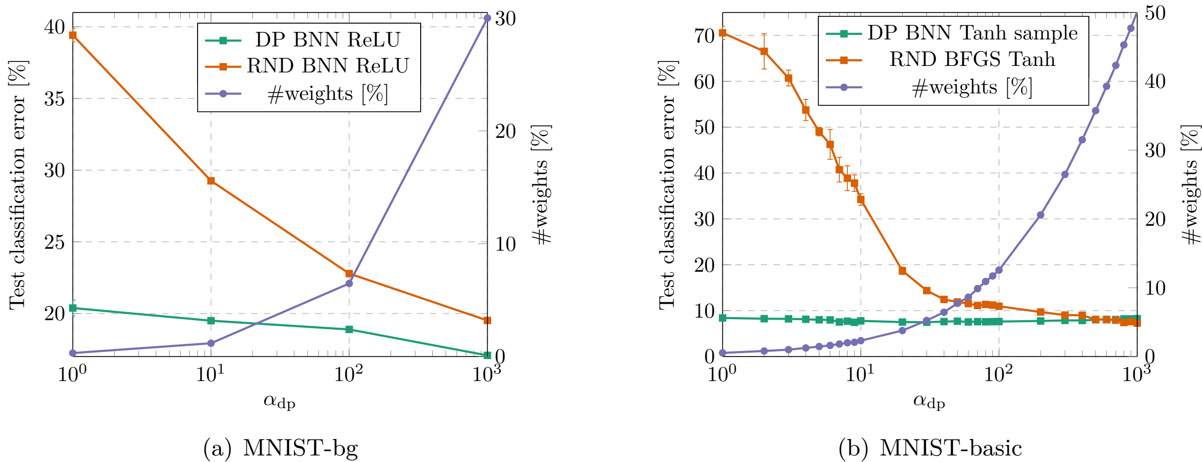
Dataset	N	d_0
Abalone	4,177	8
Housing	506	13
Concrete	1,030	8
Power Plant	9,568	4
WineQ-red	1,599	11
WineQ-white	4,898	11

Table 6.4: Regression datasets: Number of data samples N and number of input features d_0.

6.3.2 Classification Results with Stochastic Gradient MCMC

We replicated the experiments for DP BNN from Section 6.3.1 but replaced AHMC for weight sampling with SGLD [102] and SGHMC [103]. We refer to Section 3.5 for details about stochastic sampling methods and their corresponding parameters. We used mini-batches of size $N_\text{B} = 100$. For SGLD, we updated the step size η_t according to (3.116) after every epoch where t is the current epoch. For the hyperparameters of (3.116), we evaluated $\beta_\eta \in \{10^1, 10^2, 10^3\}$ and fixed $\alpha_\eta = 10^{-3}$ and $\gamma_\eta = 0.55$. For SGHMC, we evaluated the step size $\eta \in \{10^{-7}, 10^{-6}\}$ and fixed the momentum term $\xi = 0.1$. To keep the computational overhead of both methods equal, we generated a weight sample after every epoch. This implies that parameter T of SGHMC equals the number of update steps per epoch.

The results are shown in Table 6.3. We only report results for ReLU as it outperformed tanh. Both methods, SGLD and SGHMC, achieve a similar performance and are on par with AHMC (see Table 6.1). On some datasets, AHMC benefits from the automatic search for good leapfrog parameters whereas on other datasets the stochastic methods benefit from the stochasticity of the gradients. Nevertheless, stochastic gradient sampling methods are crucial for a scalable algorithm, but this is not the main focus of this work.



(a) MNIST-bg (b) MNIST-basic

Figure 6.4: (a) Test classification error (left y-axis) and fraction of used weights compared to DNNs without sharing (right y-axis) over α_{dp} for DP BNN ReLU and RND BNN ReLU with two hidden layers of 100 neurons. (b) Test classification errors and fraction of used weights of BFGS optimized DNNs with random weight sharing and posterior samples of DP BNNs using tanh.

6.3.3 Regression Results

For regression, we used several datasets from the UCI repository (see Appendix A.7). The number of data samples N and the number of input features d_0 for each dataset are summarized in Table 6.4. The task of each dataset is to predict a scalar target value. For all DNN models, we evaluated $\beta^2 \in \{10^{-2}, 10^{-1}\}$. All features and target values were normalized to zero mean and unit variance. We performed 5-fold cross-validation and report the test log-likelihood and the test root mean squared error (RMSE) *without* target normalization. Note that we report both the log-likelihood and the RMSE of the same model resulting in the best test log-likelihood.

The results are shown in Table 6.2. For dropout, we only report results for the ReLU activation function (Dropout ReLU) as it outperformed tanh on average. For the BNN models, tanh performed better on these tasks. DP BNN Tanh achieved the best log-likelihood on four out of six datasets. Furthermore, DP BNN for both Tanh and ReLU consistently outperform their RND counterparts except for WineQ-red where RND BNN Tanh performs best. This is due to outliers that are by chance well captured by 1–2 DNNs of the ensemble of 5,000 DNNs for RND BNN Tanh, whereas these outliers are poorly captured by the other methods. For the RMSE, BNN Tanh performed best on four out of six datasets. The DP models also outperform Dropout and their RND counterparts in terms of RMSE on most datasets. We emphasize that the log-likelihood was used to select the models and we report the RMSE for the same models.

6.3.4 Reducing the Number of Weights

On the MNIST datasets, at most 50% of the weights were used for $\alpha_{\mathrm{dp}} = 10^3$. For the largest evaluated structure with two hidden layers and $d_1 = d_2 = 100$, at most 30% of the weights were used. The savings grow with larger structures as the number of weights depends logarithmically on the number of connections [200]. The additional memory needed to store 25 sharing configurations is relatively small compared to the total number of 5,000 DNN samples.

The concentration parameter α_{dp} can be used to trade off between the number of weights and the classification error. Figure 6.4(a) shows the influence of α_{dp} on both the classification error and the number of weights for the ReLU activation with two hidden layers having 100 neurons each on MNIST-bg. By setting $\alpha_{\mathrm{dp}} = 1$, only about 0.3% of the weights are used and the classification error increases by approximately 3% (absolute). Using our setup of 200 weight

samples per configuration, a total overhead of 0.5% to store the configuration is added and thus the overall memory requirement per single DNN is 0.8% compared to BNNs without sharing.

6.3.5 Benefit over Random Weight Sharing

The next experiment compares configurations identified by our sampling method with randomly shared weights on MNIST-basic. We used DNNs with tanh activation and two hidden layers with 50 neurons each. For several values of α_{dp}, we performed 200 iterations of configuration sampling, dropped the first 100 samples as burn-in, and averaged the errors of the individual samples. This experiment was performed five times, resulting in 500 samples. To show that configuration sampling by itself finds suitable configurations, we did not perform any weight sampling. For comparison, we initialized for each α_{dp} five DNNs with random sharing using the same number of weights as obtained by configuration sampling and trained them using BFGS.

The average test error is shown in Figure 6.4(b). The average error of a *single* DP BNN posterior sample is almost constant for all α_{dp} values whereas the error of DNNs with random weight sharing drops consistently as more and more weights are used. Especially when only few weights are used, DP BNNs outperform DNNs with random weight sharing. Without weight sharing, the best error with BFGS is 5.7%. This is about 1.5% better than the error of a single DP BNN posterior sample using only few weights.

Next we compare the full ensemble performance of 5,000 DNNs on MNIST-bg. In particular, we use two hidden layers with 100 neurons each and compare our standard setup (DP BNN ReLU) with randomly shared weights (RND BNN ReLU). The results for several α_{dp} are shown in Figure 6.4(a). Especially when α_{dp} is small and only few weights are used, random weight sharing achieves a poor performance.

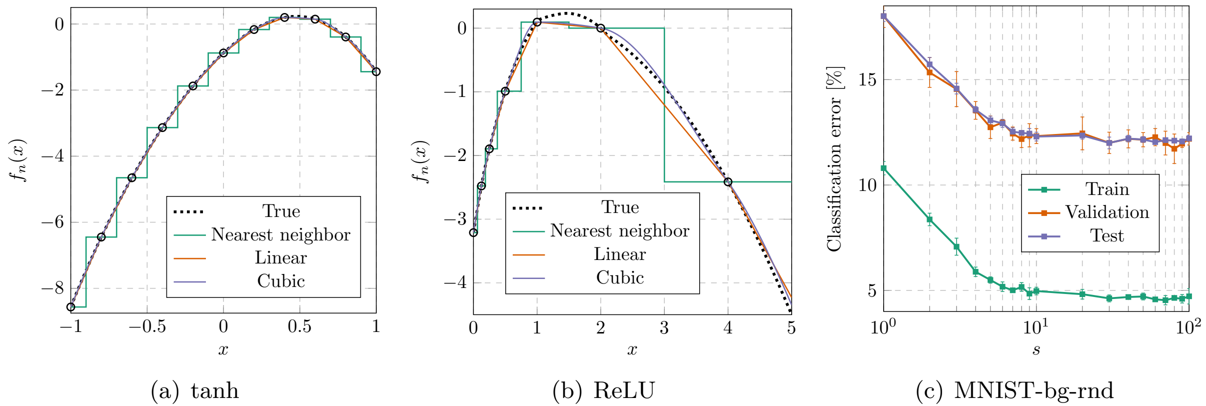
6.3.6 Running Time Experiments

We compared the running time of configuration sampling and weight sampling for different α_{dp} values on MNIST-rot-bg. We used two layers with 100 hidden neurons each. Since the number of weights depends largely on α_{dp} and the network structure and the variants of MNIST are all of equal size, the running time varies only slightly among these datasets.

The results are shown in Figure 6.3(b). The running times of SGLD and SGHMC are approximately equal and we therefore report their running times as SGMCMC. The running time for weight sampling with AHMC and SGMCMC is largely unaffected by α_{dp} since computing gradients with backpropagation is mainly affected by the DNN structure and not by the number of used weights. However, since configuration sampling requires each connection to be replaced by all existing weights, its running time grows with larger α_{dp}. Without likelihood interpolation the algorithm takes up to two orders of magnitudes longer which is impractical. The SGMCMC methods provide a speed-up of about one order of magnitude compared to AHMC, shifting the computational bottleneck for all α_{dp} values to configuration sampling.

6.3.7 Different Interpolation Methods

Figure 6.5(a) and Figure 6.5(b) show qualitative examples of likelihood interpolation for tanh and ReLU, respectively. For both activation functions, linear and cubic interpolation [205] provide a much better fit to the ground truth than simple nearest neighbor interpolation. For tanh, the smooth cubic interpolation scheme provides a slightly better fit than linear interpolation since the ground truth $f_n(x)$ is also smooth. For the non-smooth ReLU activation, $f_n(x)$ is non-smooth whenever activations in the following layers change their sign. This behavior occurs especially for large x where our grid of ground truth values is relatively coarse such that interpolation becomes difficult.



(a) tanh (b) ReLU (c) MNIST-bg-rnd

Figure 6.5: Likelihood interpolation. (a) Exemplary log-likelihood $f_n(x)$ for a single sample $(\mathbf{x}_n, \mathbf{y}_n)$ as a function of a specific neuron's output x with tanh *activation. The likelihood is evaluated exactly at the black circles which are subsequently used for interpolation. The true function (black dotted) is shown together with three interpolation schemes. (b) Same as in (a) for the ReLU activation. (c) Classification errors (averaged over five runs) for several discretization parameters s. The results were produced by averaging an ensemble of ten DNNs obtained by configuration sampling using nearest neighbor likelihood interpolation (no weight sampling).*

We note that for regression a continuous interpolation scheme (linear or cubic) is crucial, and nearest neighbor interpolation produced inferior results (results not reported). This behavior was hardly observable for classification. This is explained by the fact that regression requires a precise prediction of a numerical target value which makes it more susceptible to approximation errors. Classification, on the other hand, is more robust since it only requires the correct prediction of the maximum class.

6.3.8 Influence of the Discretization Parameter

To assess the impact of likelihood interpolation on the quality of the generated samples, we evaluated several values of the discretization parameter s on MNIST-bg-rnd for two hidden layers with 50 neurons each. We performed 20 iterations of configuration sampling with nearest neighbor likelihood interpolation, discarded the first ten samples as burn-in, and averaged the outputs of the last ten DNNs. The average classification error and the standard deviation of performing this experiment five times is shown in Figure 6.5(c). The error drops consistently until $s = 10$ and then stays almost constant until $s = 100$. We note that running the experiment without likelihood interpolation takes too long but $s = 100$ gives already a good approximation. More accurate approximations using larger s do not seem to pay off for the increased computational effort.

For linear and cubic interpolation or when performing the same experiment with intermediate runs of weight sampling using AHMC, the error is approximately constant over the whole range of s showing that the influence of s is even less severe.

6.4 Discussion

In this chapter, we introduced a DP prior over the weight prior distribution $p(\mathbf{W})$. This results in a weight sharing which we subsequently exploit to reduce the memory footprint of storing an ensemble of DNNs. As opposed to common clustering techniques that operate solely on the weight values, our method results in a weight sharing that is adopted to the given data.

We infer the weight sharing using block Gibbs sampling, i.e., we alternate between sampling from the conditional posterior over the weights, $p(\mathbf{w}|\mathbf{Z}, \mathcal{D})$, and sampling from the conditional posterior over the configurations, $p(\mathbf{Z}|\mathbf{w}, \mathcal{D})$. Weight sampling is performed using AHMC and configuration sampling is performed using Gibbs sampling. By using a faster stochastic gradient MCMC method for weight sampling, configuration sampling becomes the computational bottleneck. For each configuration $\mathbf{Z}$, we generate multiple weight samples $\mathbf{w}$ in order to distribute the memory overhead for the configurations over many samples.

Since sampling from $p(\mathbf{Z}|\mathbf{w}, \mathcal{D})$ naïvely is intractable, we developed an interpolation based approximation that takes the structure of DNNs into account. This approximation avoids many redundant computations and makes posterior sampling feasible. A relatively small discretization parameter of $s = 10$ was sufficient for interpolation to perform well. Cubic interpolation produced good results for both classification and regression as well as for both tanh and ReLU activation functions.

In our experiments, we demonstrated the ability of our model to reduce the total number of parameters substantially. Our model mostly outperforms Bayesian DNNs with random weight sharing. On some datasets (especially for regression), our model even outperforms DNNs without sharing. This indicates a regularizing effect of weight sharing. We have shown that by varying the concentration parameter α_{dp} of the DP, we can effectively trade off between model size and accuracy. However, note that selecting a larger α_{dp} considerably increases the running time of configuration sampling. Especially in the small α_{dp} regime, our method substantially outperforms randomly shared weights.

Experiments using layerwise weight sharing consistently outperformed global weight sharing. One reason could be that different layers need to exhibit different weight scales, and sharing weights among different layers deteriorates performance. For instance, in a different context the *He initialization* (2.32) and (2.33) uses different layerwise weight scales for initialization to substantially improve convergence of optimization [20].

6.4.1 Limitations and Future Work

Although the proposed method provides an elegant framework to obtain weight sharing in DNNs, it suffers from several drawbacks that limit its practical applicability. First and foremost, the proposed method is restricted to smaller architectures and small to medium sized datasets. We note that the current work was developed with the constraint of introducing as little bias as possible. In particular, the interpolation based approximation is the only component introducing bias. However, this constraint is too restrictive and trading bias for increased computational efficiency might be key to obtain a scalable method. We believe that low order Taylor approximations of the likelihood similar as in [83] are promising to achieve this.

Furthermore, extensions to other DNN architectures such as CNNs and RNNs need to be considered. However, the proposed interpolation method cannot be applied to these architectures since changing individual weights of CNNs and RNNs affects several neurons. Again, low order Taylor approximations might be promising solutions. Nevertheless, we note that the proposed method can be readily used for the fully connected layers of CNNs where commonly most of the weights are located.

In its present form, our method is tailored to reducing the number of parameters of an ensemble of DNNs. Exploring different ways to reduce the memory footprint of individual DNNs by means of DP based weight sharing is worthy of investigation. We believe that, once the computational challenges have been solved and a scalable algorithm has been developed, the proposed method might be a viable alternative to other clustering based sharing techniques. For instance, a core component of the work of Han et al. [144] is k-means. It would be interesting to see whether a clustering that also takes the given data into account is beneficial. It is also conceivable that DP based weight sharing could serve to fine-tune a previously determined weight sharing.

In the current version of our method, the model hyperparameters α_{dp}, β^2, and γ^2 are fixed. We leave a Bayesian treatment considering hyperpriors over them to future work. This could be an important step towards reducing the influence of the tunable hyperparameters.

7 Resource-Efficient Bayesian Network Classifiers

So far, this thesis was mostly devoted to DNNs and techniques to improve their computational efficiency. Most of the literature (see Section 4) considers medium to large-scale datasets that require a moderate architecture size to achieve a decent accuracy. As a consequence, also the resulting DNNs after applying the respective methods are still too large for resource-constrained devices. For some applications, it is necessary that the underlying architecture is extraordinarily small or a different, more resource-efficient model class is employed in the first place.

Because of this, we aim to answer the question whether it is possible to transfer techniques from the recent DNN literature to such a different model class. In particular, we consider BN classifiers with naïve Bayes or TAN structure for the case of discrete input features. BN classifiers are inherently efficient in that they require very few operations to compute predictions. For datasets with $\mathcal{C}$ classes and D discrete input features, a BN classifier efficiently computes predictions by accumulating $(D + 1) \cdot \mathcal{C}$ log-probabilities and determining the most probable class. Notably, no other operations, such as multiplications, are required.

In this chapter, we adapt two particular techniques from the deep learning literature to BN classifiers. The first technique is a structure learning method to discover small BNs that achieve high accuracy. While tuning DNN structures has long relied on manual efforts in order to achieve high accuracy, the field of NAS for the automatic discovery of DNN architectures has become an active research area just very recently. Importantly, many NAS techniques for DNNs incorporate a mechanism that allows us to take resource efficiency into account. As a result, these methods find structures that do not only perform well in terms of accuracy, but which are also efficient in terms of various resource efficiency metrics. We refer to Section 4.3.5 for a thorough overview of NAS methods for DNNs.

At the core of most NAS methods, the architecture of a DNN is viewed as a graph and the task is to find how the nodes in this graph are connected to each other. This suggests that NAS techniques might be applicable to BNs since their structure is also a graph. The method we are going to present is based on *differentiable* NAS methods [184, 187]. Differentiable methods are appealing as they enable us to train the parameters and the structure of a DNN according to the same discriminative criterion through gradient-based optimization without requiring combinatorial search heuristics.

To be more precise, we propose a differentiable approach for TAN structure learning of BN classifiers. Our method assumes a fixed variable ordering and is based on a relaxation of the discrete graph structure to a *discrete distribution* over graph structures. Given a differentiable loss $\mathcal{L}$ over the BN parameters, we formulate a new structure learning (SL) loss $\mathcal{L}_{\text{SL}} = \mathbb{E}[\mathcal{L}]$ as an expected loss with respect to this distribution over graph structures. Subsequently, the structure loss $\mathcal{L}_{\text{SL}}$ is jointly optimized for the BN parameters and the *continuous* distribution parameters for the graph structures using gradient-based learning. After learning, we select the most probable structure. This allows us to perform structure learning using commonly used discriminative criteria such as the conditional likelihood or the probabilistic margin [207]. In fact, the proposed method is agnostic to the specific loss and only requires that it is differentiable. To also accommodate for the model size, we introduce an additional loss term that penalizes the number of parameters of a specific TAN structure. This allows us to effectively trade off between accuracy and model size.

Our second method performs quantization-aware training of the BN parameters using the STE. During training, our quantization method maintains a set of real-valued auxiliary parameters $\boldsymbol{\theta}$ that are quantized to few bits during forward propagation to obtain $\boldsymbol{\theta}_\mathrm{q}$. During backpropagation, the gradient is computed with respect to the quantized parameters $\boldsymbol{\theta}_\mathrm{q}$ which is then passed "straight-through" to update the real-valued parameters $\boldsymbol{\theta}$. This procedure is typically more effective than performing quantization as a post-processing step, since the real-valued parameters $\boldsymbol{\theta}$ become robust to quantization during training. After training, the model is deployed using the quantized parameters $\boldsymbol{\theta}_\mathrm{q}$. This paradigm has been widely used for quantization in the deep learning literature and we refer to Section 4.1.2 for a comprehensive review of these methods.

We perform extensive experiments using a hybrid generative-discriminative loss based on the probabilistic margin [207]. Our structure learning method consistently outperforms random TAN structures and likelihood optimized Chow-Liu TAN structures [208] on all evaluated datasets by a large margin. We show that a heuristic variable ordering for image data based on pixel locality further improves performance. We demonstrate that by incorporating a model size penalty into our objective, we can generate a trajectory of Pareto optimal BN classifiers with respect to accuracy and model size.

Moreover, we show in extensive experiments that quantization-aware training is an effective and simple method for quantization in BN classifiers. We compare our quantization method with a specialized branch-and-bound approach that directly operates on the discrete parameter space of BNs [209]. Our quantization method does not only achieve higher accuracy, but it also takes much less training time than the computationally intensive branch-and-bound algorithm.

We also contrast quantized BN classifiers with small-scale quantized DNNs with respect to (i) model size, (ii) number of operations required for predictions, and (iii) the prediction accuracy. We investigate Pareto optimal models with respect to these three dimensions and find that no model class can be excluded a priori. Quite importantly, our analysis also shows that quantization-aware training of DNNs performs well for small architectures and small datasets. This is notable since most work on DNN quantization is mainly focusing on the large-scale setting and the small-scale setting has been rarely investigated in the literature.

This chapter is largely based on our papers "Differentiable TAN Structure Learning for Bayesian Network Classifiers" presented at the PGM conference in 2020 [189] and "On Resource-Efficient Bayesian Network Classifiers and Deep Neural Networks" presented at the ICPR conference in 2021 [134].[26]

The outline of this chapter is as follows. Section 7.1 introduces the necessary background. This includes BNs with a particular focus on classification, a brief overview of generative and discriminative training for probabilistic classifiers, and related structure learning work. Section 7.2 presents our differentiable TAN structure learning approach. Quantization-aware parameter learning for BNs is introduced in Section 7.3. The experiments for structure learning and quantization are presented in Section 7.4 and Section 7.5, respectively. We discuss our findings in Section 7.6.

7.1 Bayesian Network Classifiers

Note that BNs have already been introduced in Section 3.1.2, but for this chapter to be more self-contained, we review the basic concepts again here. Let $\boldsymbol{X} = \{X_1, \ldots, X_D\}$ be a multivariate random variable. A BN is a graphical representation of a probability distribution $p(\boldsymbol{X})$ as a directed acyclic graph $\mathcal{G}$ whose nodes correspond to the random variables X_i. More specifically, the graph $\mathcal{G}$ determines a factorization of $p(\boldsymbol{X})$ according to

$$p(\boldsymbol{X}) = \prod_{i=1}^{D} p\left(X_i \,|\, \mathrm{pa}(X_i)\right), \tag{7.1}$$

[26] The conference is actually termed *ICPR 2020* but was postponed to 2021 due to the COVID-19 pandemic.

where $\text{pa}(X_i)$ is the set of parents of X_i in $\mathcal{G}$. This factorization allows us to specify the full joint distribution $p(\boldsymbol{X})$ by the individual factors $p(X_i \,|\, \text{pa}(X_i))$. We consider distributions over *discrete* random variables such that each conditional distribution $p(X_i \,|\, \text{pa}(X_i))$ can be represented as a conditional probability table (CPT) $\boldsymbol{\theta}_{i|J_i}$, where $J_i = \{j : X_j \in \text{pa}(X_i)\}$ are the indices of X_i's parents. The joint distribution $p(\boldsymbol{X})$ is then specified by the collection of CPTs $\boldsymbol{\theta}_{\mathcal{G}} = \{\boldsymbol{\theta}_{1|J_1}, \ldots, \boldsymbol{\theta}_{D|J_D}\}$ of all random variables $\boldsymbol{X}$.

When considering the task of classification in particular, we are given an additional class random variable Y and assume that a BN is used to model the joint distribution $p(\boldsymbol{X}, Y)$. In this context, the variables $\boldsymbol{X}$ are called input features. We can then construct a probabilistic classifier by assigning an input $\mathbf{x}$ to the conditionally most probable class

$$\hat{y} = \operatorname*{argmax}_y p(y \,|\, \mathbf{x}) = \operatorname*{argmax}_y p(\mathbf{x}, y) = \operatorname*{argmax}_y \log p(\mathbf{x}, y). \tag{7.2}$$

Assuming that the individual factors of (7.1) can be computed efficiently, classification according to (7.2) is particularly convenient by accumulating only $D+1$ log-probabilities

$$\log p(\boldsymbol{X}, Y) = \log p\left(Y \,|\, \text{pa}(Y)\right) + \sum_{i=1}^{D} \log p\left(X_i \,|\, \text{pa}(X_i)\right) \tag{7.3}$$

for each class $y \in \{1, \ldots, \mathcal{C}\}$ and reporting the most probable class $\hat{y}$.

However, the situation becomes problematic concerning the size of the CPTs $\boldsymbol{\theta}_{i|J_i}$ which is determined by the number of values that X_i and $\text{pa}(X_i)$ can take jointly. Consequently, assuming that each random variable X_i can take at least two values, the size of X_i's CPT grows exponentially with the number of parents $|\,\text{pa}(X_i)|$. This can become a computational bottleneck even for few parents. Therefore, it is desirable to maintain graph structures where each node has few parents such that inference tasks remain feasible. In this paper, we consider two commonly used types of structures for BN classifiers, namely the naïve Bayes structure and TAN structures. These structures restrict the number of conditioning parents and, therefore, do not suffer from the exponential growth.

7.1.1 Naïve Bayes and Tree-Augmented Naïve Bayes (TAN) Structures

The naïve Bayes structure is illustrated in Figure 7.1(a). The graph $\mathcal{G}$ contains a single root node Y which is the sole parent of each feature node X_i. The factorization induced by the naïve Bayes assumption is given by

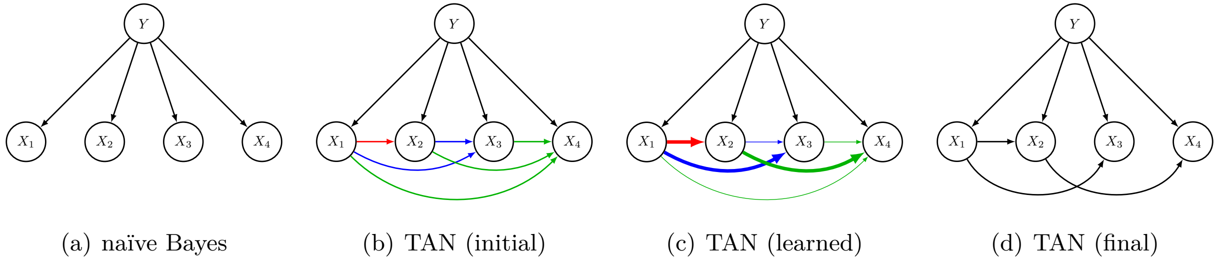
$$p(\boldsymbol{X}, Y) = p(Y) \prod_{i=1}^{D} p\left(X_i | Y\right). \tag{7.4}$$

The naïve Bayes model assumes that all inputs $\boldsymbol{X}$ are conditionally independent given the class Y. Although this independence assumption rarely holds in practice, naïve Bayes models often perform reasonably well while requiring only few parameters and enabling fast inference.

The TAN structure generalizes the naïve Bayes structure by allowing each feature X_i—in addition to the class variable Y—to directly depend on at most one other feature X_j. An example TAN structure is illustrated in Figure 7.1(d). The factorization of a TAN BN is given by

$$p(\boldsymbol{X}, Y) = p(Y) \prod_{i=1}^{D} p\left(X_i \,|\, \text{pa}(X_i)\right), \tag{7.5}$$

subject to $|\,\text{pa}(X_i)| \leq 2$ and $Y \in \text{pa}(X_i)$. As we will see in Section 7.4, this relaxation of



(a) naïve Bayes (b) TAN (initial) (c) TAN (learned) (d) TAN (final)

Figure 7.1: (a) The naïve Bayes model as a BN. (b)–(d) TAN structure learning: We consider every left-to-right edge of the ordered variables as a candidate for the conditioning parents. The TAN structure allows each feature node X_i to have one additional parent besides Y. Therefore, each edge is associated with a selection probability indicated by its thickness. (b) Initially, all edges are equally probable. (c) After learning, those edges leading to the best objective value are more probable. (d) The most probable incoming edge of each node is selected as the conditioning parent.

the graph structure typically improves the predictive performance, but it also introduces a substantial memory overhead due to larger CPTs.

We will use the following refined notation for the parameters of a TAN structure. The set of all CPTs is given by $\boldsymbol{\theta}_{\mathcal{G}} = \{\boldsymbol{\theta}_y\} \cup \{\boldsymbol{\theta}_{1|j_1}, \ldots, \boldsymbol{\theta}_{D|j_D}\}$ for $j_i \in \{0, \ldots, D\}$. Here, $\boldsymbol{\theta}_y$ denotes the CPT of $p(Y)$ and $\boldsymbol{\theta}_{i|j}$ denotes the CPT of $p(X_i \,|\, X_j, Y)$. We define $X_0 = \emptyset$ such that $\boldsymbol{\theta}_{i|0}$ denotes the CPT of $p(X_i \,|\, Y)$, i.e., X_i has no additional parent X_j. For this notation, the naïve Bayes structure appears as a special case of the TAN structure where $j_i = 0$ for all i.

As opposed to the naïve Bayes model, the TAN structure is not fixed. We can utilize this freedom to perform structure learning in order to balance accuracy and model size. However, this is not straightforward as the number of possible TAN structures is exponential in the number of input features D. In Section 7.2, we present a differentiable method for jointly training the graph structure and the CPTs that favors smaller models.

7.1.2 Hybrid Generative-Discriminative Training

Given a dataset $\mathcal{D} = \{(\mathbf{x}_n, y_n)\}_{n=1}^N$ comprising N samples of input-target pairs, a model of the joint distribution $p(\boldsymbol{X}, Y)$ over inputs and outputs such as BN classifiers can be trained using a generative or a discriminative approach. In contrast, DNNs as introduced in Section 2.2 are inherently discriminative models designed to model the conditional distribution $p(Y \,|\, \boldsymbol{X})$.

A *generative* approach is concerned with modeling the joint distribution $p(\boldsymbol{X}, Y)$ well. This is typically accomplished by minimizing the negative log-likelihood (NLL) loss

$$\mathcal{L}_{\text{NLL}}(\boldsymbol{\theta}_{\mathcal{G}}; \mathcal{D}) = -\sum_{n=1}^{N} \log p(\mathbf{x}_n, y_n). \tag{7.6}$$

A *discriminative* approach is concerned with modeling the conditional distribution $p(Y \,|\, \boldsymbol{X})$ directly. Note that, in this case, the model under consideration still represents a joint distribution $p(\boldsymbol{X}, Y)$, but it is not important anymore that it reflects the given data well. In this work, we consider a discriminative loss based on the notion of a probabilistic margin [207, 210]. In particular, we minimize the large margin (LM) loss

$$\mathcal{L}_{\text{LM}}(\boldsymbol{\theta}_{\mathcal{G}}; \mathcal{D}) = \sum_{n=1}^{N} \max(0, \gamma_{\text{LM}} - \beta_n(\boldsymbol{\theta}_{\mathcal{G}})), \tag{7.7}$$

where $\gamma_{\text{LM}} > 0$ is a desired log-margin hyperparameter and β_n is the probabilistic log-margin of

the nth sample defined as

$$\beta_n(\boldsymbol{\theta}_{\mathcal{G}}) = \log\left(\frac{p(y_n\,|\,\mathbf{x}_n)}{\max_{y\neq y_n} p(y\,|\,\mathbf{x}_n)}\right) = \log p(\mathbf{x}_n, y_n) - \max_{y\neq y_n} \log p(\mathbf{x}_n, y). \tag{7.8}$$

In our implementation of (7.8), we employ the softened version of the maximum from [207], i.e.,

$$\max(v_1, \dots, v_M) \approx \frac{1}{\xi_{\text{LM}}} \log \sum_{i=1}^{M} \exp(\xi_{\text{LM}}\, v_i), \tag{7.9}$$

where $\xi_{\text{LM}} > 1$ is a hyperparameter.

In practice, the discriminative approach is often superior in terms of classification performance, but often discards most of the probabilistic semantics of the model. This motivates a hybrid generative-discriminative loss to combine the advantages of both approaches as

$$\mathcal{L}_{\text{HYB}}(\boldsymbol{\theta}_{\mathcal{G}}; \mathcal{D}) = \mathcal{L}_{\text{NLL}}(\boldsymbol{\theta}_{\mathcal{G}}; \mathcal{D}) + \lambda_{\text{HYB}}\, \mathcal{L}_{\text{LM}}(\boldsymbol{\theta}_{\mathcal{G}}; \mathcal{D}), \tag{7.10}$$

where $\lambda_{\text{HYB}} > 0$ is a hyperparameter. Previous work has shown that by carefully trading off between the generative and the discriminative loss in (7.10), most of the probabilistic semantics can be maintained while achieving good accuracy. In many cases a hybrid classifier even outperforms a pure discriminative classifier since the generative term can be seen as a regularizer [211].

7.1.3 Structure Learning for Bayesian Networks

Structure learning for BNs is concerned with determining a directed acyclic BN graph $\mathcal{G}$ that is optimal in some sense. This is a highly non-trivial task since the number of graphs $\mathcal{G}$ is superexponential in the number of variables.

There exist basically two different classes of structure learning algorithms for BNs. On the one hand, *constraint-based approaches* apply statistical tests to identify conditional independencies among the variables X_i and aim to find a graph $\mathcal{G}$ that best reflects these conditional independencies [212]. On the other hand, *score-based approaches* perform classical combinatorial optimization over the space of BN graphs $\mathcal{G}$ to maximize a given score function. Score maximization is known to be NP-hard, even for the favorable case of decomposable scores, i.e., scores that decompose into terms that only depend on individual nodes and their parents [213].

Our work is closely related to the score-based approach and, therefore, our discussion will be restricted to this class of algorithms. Most score-based approaches apply some form of greedy hill climbing heuristic to maximize the given score. Starting from some arbitrary graph $\mathcal{G}$, hill climbing approaches successively apply local changes to the structure (e.g., edge removals and reversals) that result in a valid acyclic graph $\mathcal{G}'$. If such a local change improves the score, the resulting graph $\mathcal{G}'$ is adopted; otherwise the local change is discarded. This procedure typically continues until there exists no local change that improves the score. Hill climbing is most effective when the score is decomposable. In this case, the new score after applying a local change can be computed efficiently by an incremental score update [214].

We note that score functions depend, besides the given dataset $\mathcal{D}$, only on the graph structure $\mathcal{G}$ and *not* on the particular parameters $\boldsymbol{\theta}_{\mathcal{G}}$. However, many common scores are still implicitly defined in terms of some optimal parameters $\boldsymbol{\theta}^*_{\mathcal{G}}$ that are available in closed form. For instance, the likelihood score is defined in terms of the closed-form ML parameters $\boldsymbol{\theta}_{\mathcal{G},\text{ML}}$. As a consequence, most decomposable scores are generative since discriminative scores, such as the conditional log-likelihood score, do not admit closed-form expressions for the corresponding optimal parameters $\boldsymbol{\theta}^*_{\mathcal{G}}$. Instead, iterative optimization procedures are required in this case [208, 215].

Nevertheless, according to our discussion in Section 7.1.2, a generative score might limit the

performance of the resulting models on classification tasks and it is desirable to employ a discriminative score. Grossman and Domingos [215] have shown that a discriminative conditional likelihood score based on ML parameters $\boldsymbol{\theta}_{\mathcal{G},\mathrm{ML}}$ yields good results. They also performed experiments using the full conditional likelihood score using a time-consuming iterative optimization procedure within the hill climbing loop. However, they did not report improved results on the small datasets where this was feasible. A rather different approach is pursued by Peharz and Pernkopf [216] who performed margin based structure learning using a general purpose branch-and-bound algorithm.

There also exist various methods restricted to TAN structure learning—the setting considered in this chapter. In this case, there exists a polynomial time algorithm to find an optimal structure with respect to the likelihood score [208]. This algorithm is an extension of the algorithm proposed by Chow and Liu [217] to compute tree structured ML BNs. The algorithm first computes a maximum spanning tree in a complete graph whose nodes correspond to $\boldsymbol{X}$ and whose undirected edge weights are determined by the conditional mutual information $I(X_i; X_j \mid Y)$. Subsequently, the resulting undirected tree is transformed into a directed tree by directing the edges away from an arbitrarily selected root node X_i. We refer to structures discovered by this algorithm as Chow-Liu structures.

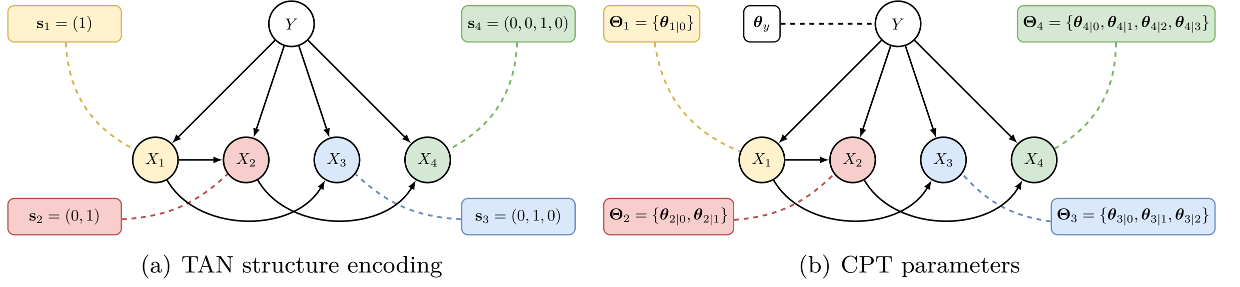
Discriminative training for TAN structures has been considered by Pernkopf et al. [218] who applied greedy hill climbing to a probabilistic margin score. Pernkopf and Wohlmayr [219] reported improved results for the probabilistic margin when optimized with simulated annealing. However, all of these "discriminative" methods have in common that their discriminative score is based on generative ML parameters $\boldsymbol{\theta}_{\mathcal{G},\mathrm{ML}}$ or, as in [215], they require a time-consuming iterative optimization procedure within the hill climbing loop. Our differentiable method presented in Section 7.2 allows for the joint training of the structure $\mathcal{G}$ and the parameters $\boldsymbol{\theta}_{\mathcal{G}}$ according to the same discriminative criterion by means of gradient-based optimization. The proposed method implicitly utilizes properties from stochastic mini-batch optimization to avoid local minima. This is a well-known problem of combinatorial search heuristics that has been addressed by several works, such as more elaborate search spaces [220] and perturbation methods [221].

7.1.4 Relation between Bayesian Network Classifiers and Deep Neural Networks

In this section, we highlight a connection between DNNs and BNs that supports the transfer of well-established methods from the deep learning literature to BN classifiers. Recall that the output layer of a DNN performs an affine transformation $\mathbf{W}^L \mathbf{x}^{L-1} + \mathbf{w}_0^L$ and reports the most probable class. This computation is equivalent to the computation performed by a logistic regression model. A connection between DNNs and BNs can then be established by appealing to the well-known fact that the naïve Bayes model is the generative counterpart to the discriminative logistic regression model [222].

Indeed, for discrete input features X_i, the following reasoning shows that logistic regression models and BN classifiers with naïve Bayes structure compute their predictions in the same way. For a BN classifier, the logarithm of (7.4) required for computing predictions is obtained by computing for each target value y a sum over log-probabilities for each feature X_i. By encoding all discrete input features x_i as one-hot vectors $\bar{\mathbf{x}}_i$ and stacking them into a single sparse vector $\bar{\mathbf{x}}$, we can cast the same computation as an affine transformation $\bar{\mathbf{W}}\bar{\mathbf{x}} + \bar{\mathbf{w}}_0$. Here, $\bar{\mathbf{W}}$ contains entries from the CPTs $\boldsymbol{\theta}_{i|0}$ for $i \in \{1, \ldots, D\}$ and $\bar{\mathbf{w}}_0$ corresponds to $\boldsymbol{\theta}_y$.

This line of reasoning can also be extended to TAN structures. In this case, predictions can be written as an affine transformation by using one-hot encodings $\bar{\mathbf{x}}_i$ of the values that X_i and its additional parent X_j take jointly. In this sense, our BN classifiers can be viewed as shallow neural networks.



(a) TAN structure encoding (b) CPT parameters

Figure 7.2: TAN structure learning. (a) The parent of each node X_i is encoded by a one-hot vector $\mathbf{s}_i$. The first entry of $\mathbf{s}_i$ encodes that no additional parent is used. (b) Each feature node X_i maintains a set of CPTs $\mathbf{\Theta}_i$ for each possible parent.

7.2 Differentiable TAN Structure Learning

In this section, we introduce a loss that admits the joint training of the graph $\mathcal{G}$ and the CPTs $\boldsymbol{\theta}_\mathcal{G}$. We show how this loss can be trained by means of gradient-based optimization using the reparameterization trick [62, 98, 99] based on the Gumbel-softmax approximation [100, 101], and the STE [14], all of which are popular methods in the deep learning community. For this chapter to be self-contained, a brief summary of these methods is provided. For a more detailed discussion on the employed methods we refer to Section 3.4.4 for the reparameterization trick, to Section 3.4.5 for the Gumbel-softmax approximation, and to Section 2.1.4 for the STE.

7.2.1 The Structure Learning Loss

Let $X_1, \dots, X_D$ be a fixed ordering of the input features. Our differentiable structure learning approach considers for each feature node X_i every X_j with $j < i$ as a possible parent. This is illustrated in Figure 7.1(b). Although the search space depends on the particular ordering of $\boldsymbol{X}$ and does not cover all possible TAN structures, it is convenient as the resulting graph $\mathcal{G}$ is guaranteed to be acyclic.

To enable a joint treatment of the graph structure $\mathcal{G}$ and the CPTs $\boldsymbol{\theta}_\mathcal{G}$, we reformulate $\log p(\boldsymbol{X}, Y)$ for TAN BNs by introducing new discrete parameters $\mathbf{s} = \{\mathbf{s}_1, \dots, \mathbf{s}_D\}$ that specify the graph structure $\mathcal{G}$. In particular, let $\mathbf{s}_i = (s_{i|0}, \dots, s_{i|i-1})$ be a one-hot encoding of X_i's parents such that $s_{i|j} = 1$ iff $\mathrm{pa}(X_i) = \{X_j, Y\}$ and $s_{i|0} = 1$ iff $\mathrm{pa}(X_i) = \{Y\}$ (i.e., no additional parent). This is illustrated in Figure 7.2(a). This encoding of the TAN structure allows us to talk about $\mathcal{G}$ and $\mathbf{s}$ interchangeably. Furthermore, let $\mathbf{\Theta}_i = \{\boldsymbol{\theta}_{i|0}, \dots, \boldsymbol{\theta}_{i|i-1}\}$ be the CPTs of all possible parents of X_i, and let $\mathbf{\Theta} = \{\mathbf{\Theta}_1, \dots, \mathbf{\Theta}_D\} \cup \{\boldsymbol{\theta}_y\}$ be the collection of all possible CPTs. This is illustrated in Figure 7.2(b). Then the log joint probability for a given TAN structure $\mathbf{s}$ can be expressed as

$$\log p(\boldsymbol{X}, Y) = \log p_{\boldsymbol{\theta}_y}(Y) + \sum_{i=1}^{D} \sum_{j=0}^{i-1} s_{i|j} \log p_{\boldsymbol{\theta}_{i|j}}(X_i \,|\, X_j, Y), \tag{7.11}$$

where the subscripts of p are to emphasize the dependency on the CPTs $\mathbf{\Theta}$, and we define $X_0 = \emptyset$. We can then use (7.11) to generalize an arbitrary log-likelihood based loss $\mathcal{L}_\mathcal{G}(\boldsymbol{\theta}_\mathcal{G})$ for a specific graph $\mathcal{G}$ to a loss $\mathcal{L}(\mathbf{\Theta}, \mathbf{s})$ where the graph structure $\mathbf{s}$ appears as free parameter. This gives us a conceptual framework to optimize the structure parameters $\mathbf{s}$ and the CPTs $\mathbf{\Theta}$ jointly.

However, minimizing the combinatorial loss $\mathcal{L}(\mathbf{\Theta}, \mathbf{s})$ is difficult as the number of different structures $\mathbf{s}$ scales exponentially in the number of features D. To circumvent the combinatorial nature of $\mathcal{L}(\mathbf{\Theta}, \mathbf{s})$, we first introduce *continuous* distribution parameters $\boldsymbol{\phi} = (\boldsymbol{\phi}_1, \dots, \boldsymbol{\phi}_D)$ where $\boldsymbol{\phi}_i = (\phi_{i|0}, \dots, \phi_{i|i-1})$ is a probability vector with $\sum_{j=0}^{i-1} \phi_{i|j} = 1, \phi_{i|j} \geq 0$. The parameters $\boldsymbol{\phi}$

induce a probability distribution over the one-hot vectors $\mathbf{s}$ and, consequently, also over the graph structures $\mathcal{G}$. We can then express a differentiable structure learning loss $\mathcal{L}_{\mathrm{SL}}$ as an expectation with respect to the distribution over graph structures as

$$\mathcal{L}_{\mathrm{SL}}(\boldsymbol{\Theta}, \boldsymbol{\phi}) = \mathbb{E}_{\mathbf{s}\sim p_{\boldsymbol{\phi}}}\left[\mathcal{L}(\boldsymbol{\Theta}, \mathbf{s})\right] = \mathbb{E}_{\mathcal{G}\sim p_{\boldsymbol{\phi}}}\left[\mathcal{L}_{\mathcal{G}}(\boldsymbol{\theta}_{\mathcal{G}})\right]. \tag{7.12}$$

Here $\boldsymbol{\theta}_{\mathcal{G}}$ are the CPTs of $\boldsymbol{\Theta}$ required by the graph structure $\mathcal{G}$. Given an optimal structure $\mathbf{s}^*$ with respect to $\mathcal{L}(\boldsymbol{\Theta}, \mathbf{s})$, an optimal solution to (7.12) is given by the distribution $\boldsymbol{\phi}^*$ that assigns all mass on this particular structure, i.e., $\boldsymbol{\phi}^* = \mathbf{s}^*$. Note that the distribution $\boldsymbol{\phi}$ has no particular interpretation and merely serves the purpose of obtaining a differentiable loss $\mathcal{L}_{\mathrm{SL}}$.

7.2.2 Minimizing the Structure Learning Loss

The structure loss (7.12) becomes intractable for a moderate number of features D. However, it can be optimized with stochastic gradient descent using Monte Carlo estimates of the gradient of (7.12). The Monte Carlo estimates of the gradient are obtained via the reparameterization trick [62, 98, 99], which has recently become a popular method for optimizing intractable expectations. The idea of the reparameterization trick is to generate a sample $\mathbf{s}$ by transforming the distribution parameters $\boldsymbol{\phi}$ along with a random sample $\boldsymbol{\varepsilon}$ drawn from a *fixed parameter-free* distribution $p(\boldsymbol{\varepsilon})$ to obtain $\mathbf{s} = g(\boldsymbol{\phi}, \boldsymbol{\varepsilon})$. This allows us to compute gradient samples of $\mathcal{L}_{\mathrm{SL}}$ with respect to $\boldsymbol{\phi}$ using the backpropagation algorithm.

For a categorical distribution with probabilities $\boldsymbol{\phi}_i$, we can sample a one-hot encoded vector $\mathbf{s}_i$ by means of a reparameterization using the Gumbel-max trick [100, 101] according to

$$\mathbf{s}_i = \operatorname*{argmax}_j \{\log(\phi_{i|j}) + \varepsilon_{i,j} : j < i\} \qquad \text{with} \qquad \varepsilon_{i,j} \sim \mathrm{Gumbel}(0,1), \tag{7.13}$$

where we assume that argmax computes a *one-hot* encoding of the maximum argument j.

However, the gradient of $\mathrm{argmax}(\boldsymbol{\phi}_i + \boldsymbol{\varepsilon}_i)$ is zero almost everywhere and cannot be used for backpropagation. To overcome this, we employ the STE [14]. The STE approximates the gradient of a zero-derivative function $f(u)$ during backpropagation with the non-zero gradient of a similar function $\tilde{f}(u) \approx f(u)$, i.e.,

$$\frac{\partial \mathcal{L}}{\partial f}\frac{\partial f}{\partial u} \approx \frac{\partial \mathcal{L}}{\partial f}\frac{\partial \tilde{f}}{\partial u}. \tag{7.14}$$

In our case, we approximate the gradient of the argmax in (7.13) during backpropagation by the gradient of a softmax function

$$\mathrm{softmax}_j((\log \boldsymbol{\phi}_i + \boldsymbol{\varepsilon}_i)/\tau_{\mathrm{g}}) = \frac{\exp((\log \phi_{i|j} + \varepsilon_{i,j})/\tau_{\mathrm{g}})}{\sum_{j'} \exp((\log \phi_{i|j'} + \varepsilon_{i,j'})/\tau_{\mathrm{g}})}, \tag{7.15}$$

where $\tau_{\mathrm{g}} > 0$ is a temperature hyperparameter. For $\tau_{\mathrm{g}} \to \infty$ we obtain a uniform distribution and $\tau_{\mathrm{g}} \to 0$ recovers the one-hot encoding. This particular sampling procedure is also known as straight-through Gumbel-softmax approximation [100].

We note that for some applications it is sufficient to compute the forward pass using the "softened" one-hot vectors obtained by the softmax (7.15) such that the STE is not required. However, in our case this would have severe consequences as the resulting log-likelihoods in (7.11) would be computed as a blend of several conditioning parents. While the resulting model still can be used as a classifier, its probabilistic interpretation as a BN with TAN structure would be corrupted. In practice, we have observed that by using the softened one-hot vectors in the forward pass, many probabilities $\boldsymbol{\phi}_i$ tend to be uniform since mixing the probabilities of several parents improves the expressiveness of the classifier. In this case, we cannot expect the

classifier induced by the most probable structure $\mathcal{G}$ to perform well.

In our approach, the number of possible conditioning parents—and, therefore, also the number of CPTs $\boldsymbol{\theta}_{i|j}$—is $\mathcal{O}(D^2)$. Note that although most of the structure parameters $s_{i|j}$ in (7.11) equal zero, we still need to consider every conditional log-probability $\log p_{\boldsymbol{\theta}_{i|j}}(X_i \,|\, X_j, Y)$ as they are required to compute the gradient of $\boldsymbol{\phi}$ using the STE. Since this is prohibitive for large D, we propose to consider for each feature node X_i only a fixed randomly selected parent subset $\{X_j : j < i\}$ of maximum size $K \ll D$.[27] This results in a linear dependence on the number of features D as $\mathcal{O}(KD)$.

We have now established a means of optimizing (7.12) for $\boldsymbol{\Theta}$ and $\boldsymbol{\phi}$ using SGD. After training has finished, we select the single most probable structure $\mathcal{G}$ and the corresponding CPTs $\boldsymbol{\theta}_{\mathcal{G}}$. We emphasize that the presented method for structure learning is agnostic to the particular loss $\mathcal{L}_{\mathcal{G}}$ and that it can be used in conjunction with any differentiable loss.

7.2.3 Model-Size-Aware TAN Structure Learning

To also take the model size into account, we extend the structure learning loss (7.12) with an additional expected model size (MS) term to obtain

$$\mathcal{L}_{\mathrm{SL}}^{\mathrm{MS}}(\boldsymbol{\Theta}, \boldsymbol{\phi}) = \mathcal{L}_{\mathrm{SL}}(\boldsymbol{\Theta}, \boldsymbol{\phi}) + \lambda_{\mathrm{MS}} \mathbb{E}_{\mathbf{s} \sim p_{\boldsymbol{\phi}}} \left[\mathcal{L}_{\mathrm{MS}}(\mathbf{s}) \right], \tag{7.16}$$

where $\mathcal{L}_{\mathrm{MS}}(\mathbf{s})$ returns the number of parameters in the CPTs for structure $\mathbf{s}$, and $\lambda_{\mathrm{MS}} > 0$ is a trade-off hyperparameter. The second term on the right hand side of (7.16) is given by

$$\mathbb{E}_{\mathbf{s} \sim p_{\boldsymbol{\phi}}} \left[\mathcal{L}_{\mathrm{MS}}(\mathbf{s}) \right] = |\boldsymbol{\theta}_y| + \sum_{i=1}^{D} \sum_{j=0}^{i-1} \phi_{i|j} \cdot |\boldsymbol{\theta}_{i|j}|, \tag{7.17}$$

where $|\boldsymbol{\theta}|$ denotes the number of parameters of $\boldsymbol{\theta}$. Objective (7.16) allows us to achieve different trade-offs between accuracy and model size by careful selection of λ_{MS} while learning the CPTs $\boldsymbol{\Theta}$ and the TAN structure $\mathcal{G}$ simultaneously.

7.3 Parameter Quantization for Bayesian Network Classifiers

In this section, we introduce quantization-aware training for BN classifiers. We refer to Section 4.1.2 for a thorough discussion on quantization-aware training of DNNs. Compared to DNNs, the main difference of quantization for BNs is that their parameters $\boldsymbol{\theta}_{\mathcal{G}}$ are probabilities subject to a normalization constraint. Moreover, when operating in log-space, the normalized log-probabilities are non-positive, allowing for an unsigned numeric representation. We also introduce the quantization scheme for DNNs used for our extensive empirical evaluation in Section 7.5.

7.3.1 Quantization-Aware Bayesian Network Classifiers

Quantization-aware training of BN classifiers is illustrated in Figure 7.3. As briefly discussed in Section 7.1, it is convenient to store the CPTs of BNs as log-probabilities $\boldsymbol{\theta}$. During training, we store the parameters as *unnormalized* log-probabilities $\boldsymbol{\rho}$. At forward propagation, we compute the *normalized* log-probabilities $\boldsymbol{\theta}$ as

$$\theta_{i|j}^{u,\mathbf{v}} = \rho_{i|j}^{u,\mathbf{v}} - \log \sum_{u'} \exp \rho_{i|j}^{u',\mathbf{v}}, \tag{7.18}$$

[27] Since we also allow no additional parent (i.e., $s_{i|0} = 1$), this results in $K + 1$ choices for the parents of X_i.

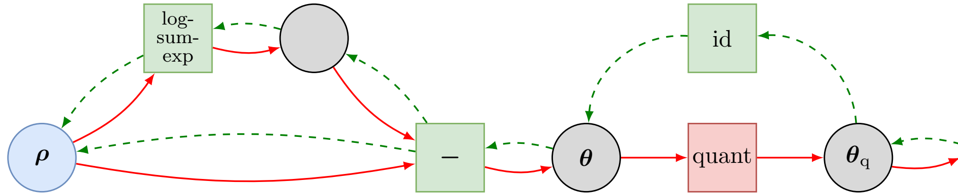


Figure 7.3: Computation graph for quantization-aware training of BN classifiers. During the forward path (red arrows), the unnormalized log-probabilities $\boldsymbol{\rho}$ *are first normalized by subtracting the* logsumexp*. Subsequently, the normalized log-probabilities* $\boldsymbol{\theta}$ *are quantized (*quant*). In the backward path (green dashed arrows), the derivative of the identity function (id) is computed.*

where $\theta_{i|j}^{u,\mathbf{v}} = \log p\,(X_i = u\,|\,(X_j, Y) = \mathbf{v})$. Note that the normalization is necessary as otherwise the log-likelihood could be made arbitrarily large.

In [209], it is proposed to represent the *non-positive* normalized log-probabilities $\boldsymbol{\theta}$ as

$$\theta_{\mathrm{q}} = -\sum_{k=1}^{B_I+B_F} b_k \cdot 2^{-B_F-1+k}, \tag{7.19}$$

where $\mathbf{b} \in \{0,1\}^{B_I+B_F}$ is a bitmask, B_F denotes the number of fractional bits, and B_I denotes the number of integer bits. To quantize $\theta \leq 0$ to the set of possible values representable by (7.19), we apply the quantizer

$$\mathrm{quant}_{\mathrm{BN}}(\theta) = \mathrm{clip}(\mathrm{round}(\theta \cdot 2^{B_F}) \cdot 2^{-B_F},\ -U,\ 0), \tag{7.20}$$

where $\mathrm{clip}(v,l,u) := \min(\max(v,l),u)$ is the clipping function and $U := 2^{B_I} - 2^{-B_F}$ is the largest magnitude representable by (7.19). During backpropagation, we apply the derivative of the identity function for the STE. Note that after quantization the log-probabilities are in general not normalized anymore.

7.3.2 Quantization-Aware Deep Neural Networks

Quantization-aware training of DNNs is illustrated in Figure 4.1. We quantize the DNN weights according to

$$\mathrm{quant}_{\mathrm{DNN}}(w) = \mathrm{quant}\left(\frac{\mathrm{clip}(w,\ -1,\ 1)+1}{2}; B\right) \cdot 2 - 1, \tag{7.21}$$

where $\mathrm{quant}(v;B)$ is the quantization scheme proposed in [130] which quantizes $v \in [0,1]$ to a B-bit number as

$$\mathrm{quant}(v;B) = \frac{1}{2^B - 1} \cdot \mathrm{round}((2^B-1)\cdot v). \tag{7.22}$$

Again, we employ the identity function for the STE.

In case the sign activation function is used, we use a stochastic sign function during training according to

$$\mathrm{sign}_{\mathrm{stoch}}(a) = \begin{cases} 1 & \text{if } \varepsilon \leq (1+a)/2 \\ -1 & \text{otherwise,} \end{cases} \tag{7.23}$$

where $\varepsilon \sim \mathcal{U}([0,1])$ is drawn from a uniform distribution. During backpropagation, we employ the derivative of tanh for the STE.

7.4 Structure Learning Experiments

We conducted experiments on two UCI classification datasets (Letter and Satimage, see Appendix A.6), the USPS dataset (see Appendix A.5), and the MNIST dataset (see Appendix A.1). For MNIST, the original images of size 28×28 were linearly downscaled to 14×14 pixels, resulting in 196 features. The features were discretized using the approach from [223]. The resulting average numbers of discrete values per feature are 9.1, 11.5, 3.4, and 13.2 for Letter, Satimage, USPS, and MNIST, respectively. Except for Satimage where we use 5-fold cross-validation, we split each dataset into two thirds of training samples and one third of test samples. For MNIST, this results in 46,669 training images and 23,331 test images, which deviates from the default setting described in Appendix A.1.

Unless stated otherwise, BN classifiers were trained according to the hybrid generative-discriminative loss (7.10), i.e., $\mathcal{L}_{\mathcal{G}} = \mathcal{L}_{\mathrm{HYB}}$ in (7.12). All experiments were performed using the stochastic optimizer Adam [11] for 500 epochs. We used mini-batches of size 50 on Satimage, 100 on Letter and USPS, and 250 on MNIST. Each experiment is performed using the two learning rates $\{3 \cdot 10^{-3}, 3 \cdot 10^{-2}\}$ for the CPTs $\boldsymbol{\Theta}$, and we report the superior result of the two optimization runs. The learning rate is decayed exponentially after each epoch, such that it decreases by a factor of 10^{-3} over the training run. We used a fixed learning rate of 10^{-3} for the structure parameters $\boldsymbol{\phi}$ that is not decayed.

The CPTs $\boldsymbol{\Theta}$ and the structure parameters $\boldsymbol{\phi}$ are stored as unnormalized log-probabilities. We initialize $\boldsymbol{\Theta}$ randomly using a uniform distribution $\mathcal{U}([-0.1, 0.1])$, and we set $\boldsymbol{\phi}$ initially to zero, resulting in a uniform distribution over graphs $\mathcal{G}$.

We anneal τ_{g} in (7.15) exponentially from 10^1 to 10^{-1} over the training run. This results in a more uniform distribution over structures $\mathbf{s}$ at the beginning of training, facilitating exploration of different structures, whereas the distribution becomes more concentrated at particular structures towards the end of training.

We compare the following BN structures.

Naïve Bayes (NB) The naïve Bayes structure without additional parents.

TAN Random TAN structure obtained by a random variable ordering and selecting a random parent X_j for each X_i with $j < i$. We evaluated ten parent selections for five variable orderings, resulting in 50 structures in total.

Chow-Liu TAN structure obtained using the procedure from [208].

TAN Subset (ours) Fixed random variable ordering and randomly selected parent subset of maximum size K satisfying the $j < i$ constraint. We evaluated five parent subsets for five variable orderings, resulting in 25 configurations in total. We selected $K \in \{2, 5, 8\}$.

TAN All (ours) Fixed random variable ordering and considering *all* parents satisfying $j < i$. We evaluated five variable orderings. This setting is only evaluated for the datasets Letter and Satimage that have fewer features D.

TAN Heuristic (ours) Heuristically determined feature ordering and respective parent subsets based on pixel locality. This setting is evaluated for the image datasets USPS and MNIST. We evaluated three ordering heuristics (for details see Section 7.4.2) and selected $K \in \{1, 2, 5, 8\}$.

We evaluated the same five random variable orderings for TAN Random, TAN Subset, and TAN All. To assess the impact of different K, the parent subsets for smaller K are strict subsets of parent subsets for larger K. Note that we allow X_i to have no additional parent such that for TAN Subset and TAN Heuristic there are effectively up to $K + 1$ choices for the conditioning parents of X_i.

loss $\mathcal{L}$	$\mathcal{L}_{\text{NLL}}$ (ML)		$\mathcal{L}_{\text{HYB}}$			$\mathcal{L}_{\text{SL}}$ with $\mathcal{L}_{\text{HYB}}$ (ours)		
structure	NB	Chow-Liu	NB	TAN Random	Chow-Liu	TAN Subset	TAN All	TAN Heuristic
Letter	25.89	15.23	12.93	10.66	9.37	8.73	8.76	/
Satimage	17.95	11.90	10.83	9.83	9.91	9.31	9.39	/
USPS	13.11	8.74	4.29	2.65	3.42	2.10	/	2.27
MNIST	17.34	7.03	4.44	4.38	3.65	3.53	/	3.29

Table 7.1: Classification errors [%] of various BN structures on several datasets. See text for details.

We tuned the hyperparameters of $\mathcal{L}_{\text{HYB}}$ using random search in two different settings. In setting (I), we selected 500 hyperparameter configurations according to $\log_{10}\lambda_{\text{HYB}} \sim \mathcal{U}([0,3])$, $\log_{10}\gamma_{\text{LM}} \sim \mathcal{U}([-1,2])$, and $\xi_{\text{LM}} \sim \mathcal{U}([1,20])$. In setting (II), we selected 100 hyperparameter configurations according to $\log_{10}\lambda_{\text{HYB}} \sim \mathcal{U}([1,3])$, $\log_{10}\gamma_{\text{LM}} \sim \mathcal{U}([-1,2])$, and used a fixed $\xi_{\text{LM}} = 10$. We applied (II) to experiments where we evaluated more structural settings, namely TAN Random, TAN Subset, and TAN Heuristic, and we applied setting (I) to all other experiments. Note that these hyperparameters are tuned individually for each experiment, i.e., each setting of the remaining hyperparameters is evaluated 500 times for setting (I) and 100 times for setting (II).

7.4.1 Classification Results

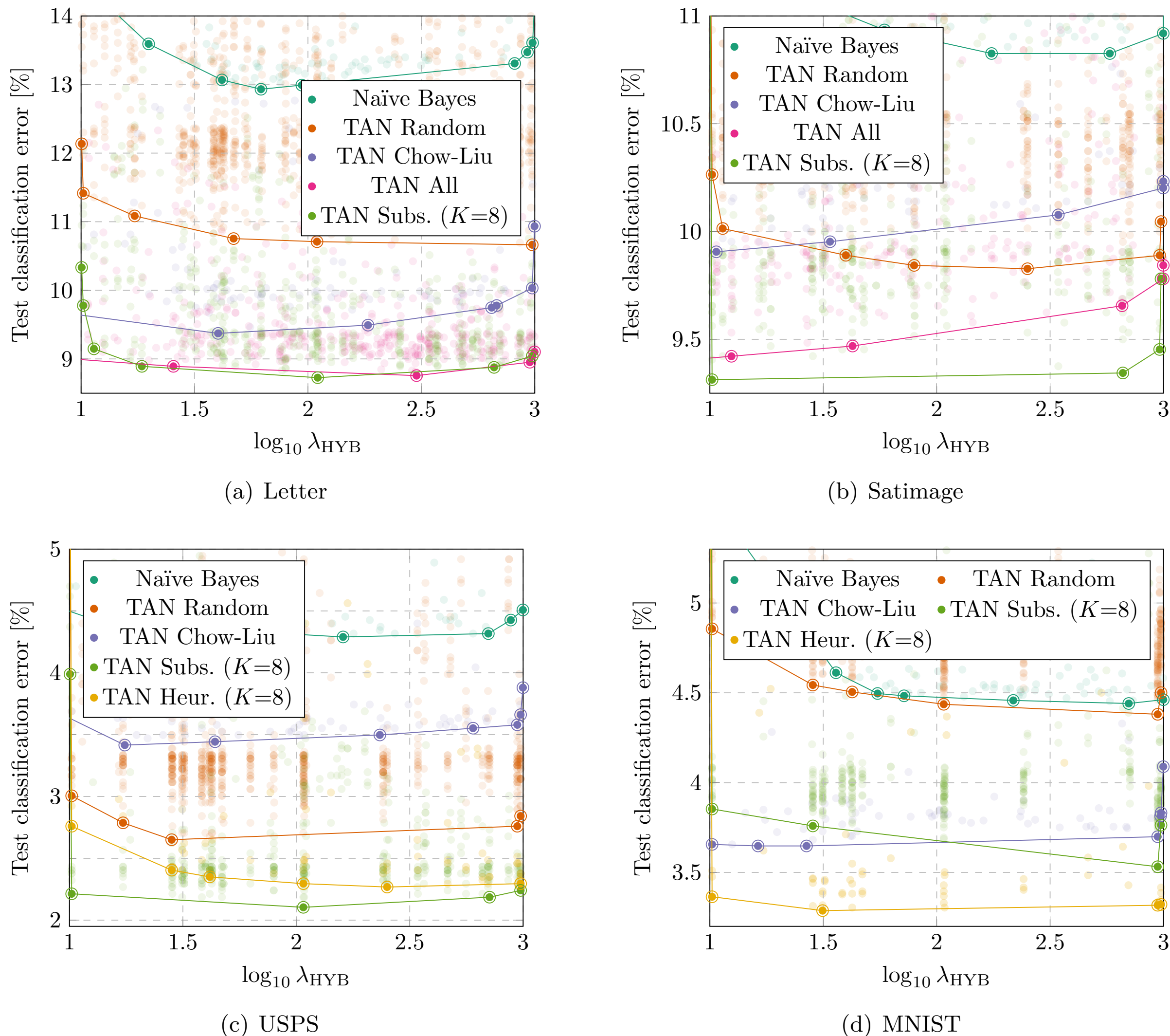
We report the test errors *after* 500 epochs of training. Results for ML parameters are obtained in closed form. The best classification errors [%] over all parameter settings are shown in Table 7.1. Results of individual experiments are shown in Figure 7.4.

Models with generative parameters (ML, i.e., $\lambda_{\text{HYB}} = 0$) perform poorly, showing that discriminative training is beneficial. The TAN structures outperform Naïve Bayes (NB) by a large margin. This highlights the benefit of introducing simple TAN interactions, even when they are selected randomly (TAN Random). Note that the data-driven Chow-Liu structure does not outperform TAN Random on all datasets, and there is even a large test error gap on USPS where we observed overfitting.

Our method (TAN Subset) outperforms TAN Random and Chow-Liu on all datasets. We emphasize that our method outperforms these baseline models on a wide range of settings (see Figure 7.4), and not just using the best setting. TAN Subset achieved its best performance using the largest parent subsets with $K = 8$ on all datasets. Interestingly, TAN All does not benefit when considering *all* conditioning parents compared to TAN Subset with $K = 8$. We have two possible explanations for this. First, Letter and Satimage have relatively low numbers of features ($D = 16$ and $D = 36$, respectively), and $K = 8$ suffices to cover a good structure with high probability using randomly selected parent subsets. Second, using a larger K results in a sparser gradient since only the gradient with respect to a single CPT $\boldsymbol{\theta}_{i|j}$ for each i is non-zero (i.e., the CPT whose corresponding parent is being sampled). This reduces the number of gradient updates per CPT which affects the learning behavior. Consequently, a different choice of number of epochs or learning rate might yield improved results.

7.4.2 Heuristic Structures for Image Data

We evaluated three different heuristic feature orderings for quadratically sized images (see Figure 7.5). Feature ordering A considers the pixels row-wise from top to bottom in a left-to-right fashion. Feature ordering B proceeds along the main diagonal by traversing the lower triangular image matrix (including the diagonal) in a row-wise fashion, and including after each pixel the corresponding transposed pixel from the upper triangular matrix. Feature ordering C proceeds from the center of the image outwards. Assuming that a center region of $H \times H$ is already ordered, we add the H pixels directly above and the H pixels directly below in a left-to-right



(a) Letter

(b) Satimage

(c) USPS

(d) MNIST

Figure 7.4: Test classification errors [%] of different methods over $\log_{10} \lambda_{\text{HYB}}$. Each point corresponds to a different experiment. For better visualization, the lines show the lower part of the convex hull of equally colored points, i.e., no points of this color are below this line.

fashion, and then we add the H pixels directly to the left and the H pixels directly to the right in a top-to-bottom fashion. Finally, we also include the four corner pixels in the order top-left, top-right, bottom-left, and bottom-right. For each ordering A, B, and C, the subset of at most K conditioning parents of X_i is obtained by selecting $\{X_j : j < i\}$ as the K closest features with respect to Euclidean distance of the corresponding pixel locations.

The heuristic structures show clear improvements on MNIST (see Table 7.1 and Figure 7.4(d)), and the best result was obtained using ordering C. Overall, when comparing different runs over several random hyperparameter settings in Figure 7.6(a), we found that ordering C slightly outperforms ordering B which in turn slightly outperforms ordering A. Note that any of the proposed heuristic orderings outperform the random orderings of TAN Subset.

Figure 7.6(b) compares distinct values of K. Overall, using larger parent subsets improves the performance. When comparing $K = 1$ and $K = 2$, we can see that choosing between two neighboring pixels and not just selecting a fixed one improves performance. When going from $K = 5$ to $K = 8$, the accuracy gains are marginal, showing that our method works well for smaller K if the feature ordering and parent subsets are carefully selected. We emphasize that this latter behavior is specific to the heuristic structures, and we generally observed larger gains

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

(a) heuristic ordering A

1	3	6	11	18	27
2	4	8	13	20	29
5	7	9	15	22	31
10	12	14	16	24	33
17	19	21	23	25	35
26	28	30	32	34	36

(b) heuristic ordering B

33	17	18	19	20	34
25	13	5	6	14	29
26	9	1	2	11	30
27	10	3	4	12	31
28	15	7	8	16	32
35	21	22	23	24	36

(c) heuristic ordering C

Figure 7.5: Heuristic feature orderings for image data of size 6×6*. (a) Feature ordering A proceeds in a row-wise fashion. (b) Feature ordering B proceeds along the main diagonal. (c) Feature ordering C starts in the center and proceeds outwards. For details see the main text.*

when going from $K = 5$ to $K = 8$ for TAN Subset.

We did not observe benefits of the heuristic structures on USPS, which we attribute to overfitting issues similar as to why Chow-Liu performs worse than TAN Random on this dataset.

7.4.3 Influence of the Feature Ordering and Parent Subsets

The influence of the feature ordering of TAN Subset on Satimage for $K = 8$ is shown in Figure 7.6(c). We can see that some random orderings clearly outperform others over a wide range of parameters, showing that the selected ordering might have a large impact on the overall performance.

To further assess the influence of fixing the feature ordering, we conducted an experiment by allowing conditioning parents that violate the $j < i$ constraint. This results in pseudo TAN structure classifiers which potentially contain cycles and, therefore, are not BNs anymore. The classification errors remained similar on all datasets except Letter where we achieved 7.98% (0.75% absolute improvement). These findings suggest that more elaborate techniques considering different orderings as in [224] are worthy of investigation.

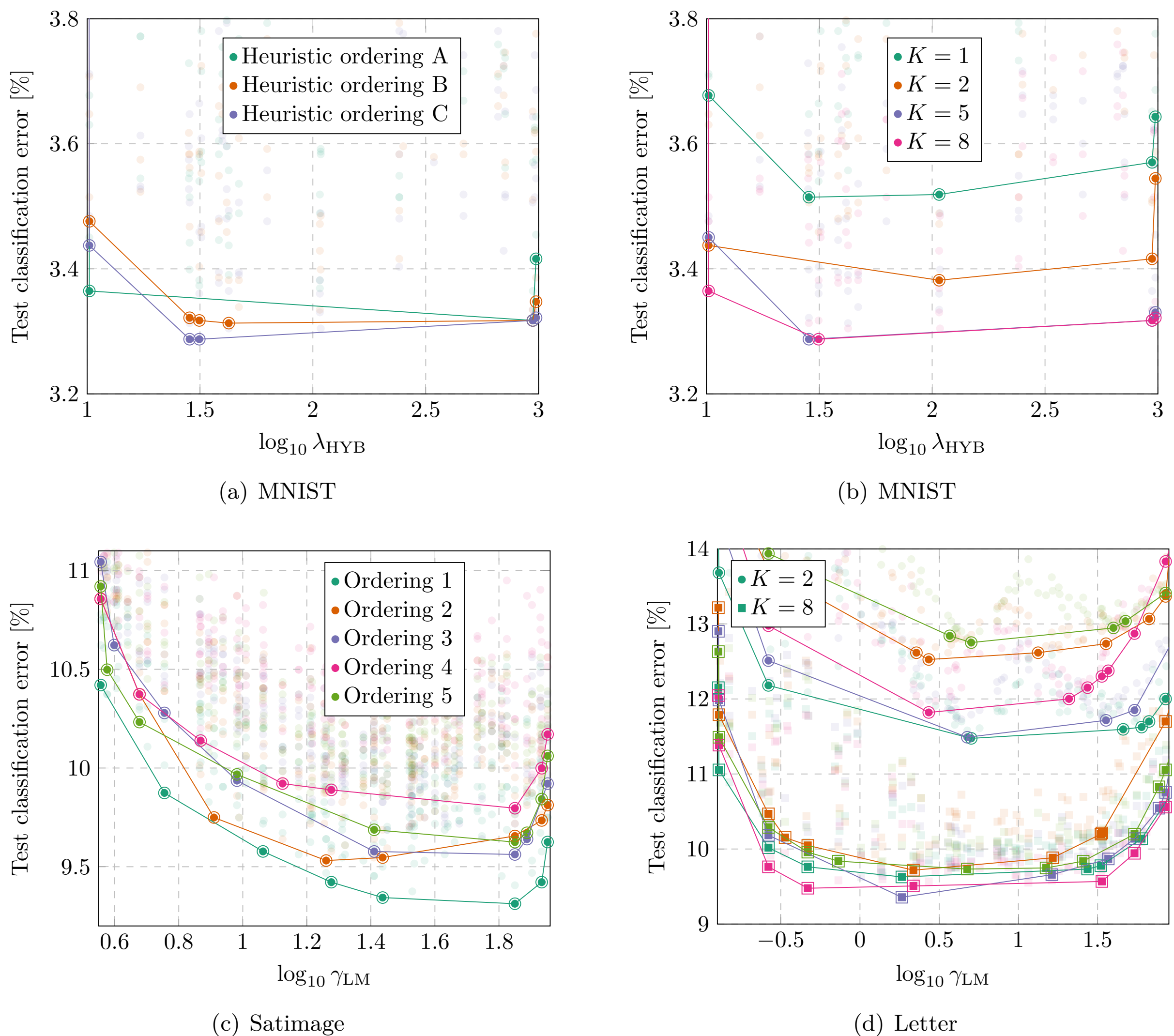
Next, we investigated the influence of particular parent subsets. Figure 7.6(d) shows different parent subsets of TAN Subset on Letter for $K \in \{2, 8\}$. Again, using larger $K = 8$ clearly outperforms the smaller $K = 2$. Similar to feature orderings, some parent subsets clearly outperform others over a wide range of parameters, but here increasing K reduces the gap.

7.4.4 Recovering the Chow-Liu Structure

We conducted an experiment using the generative loss $\mathcal{L}_{\text{NLL}}$ (i.e., $\lambda_{\text{HYB}} = 0$) to see whether our approach is capable of recovering the "ground truth" Chow-Liu structure. Therefore, we computed a Chow-Liu structure and a corresponding ordering such that the Chow-Liu structure is contained in the search space. Our method was able to recover the Chow-Liu structure consistently. Note that this is a minimal requirement of our structure learning approach as the structure learning loss $\mathcal{L}_{\text{SL}}$ decomposes to local terms similar as the generative loss $\mathcal{L}_{\text{NLL}}$, substantially simplifying the optimization problem.

7.4.5 Model-Size-Aware TAN Structure Learning

We performed model-size-aware TAN structure learning according to the method described in Section 7.2.3. The experiments were performed using the TAN Subset setting. The hyperpa-



(a) MNIST

(b) MNIST

(c) Satimage

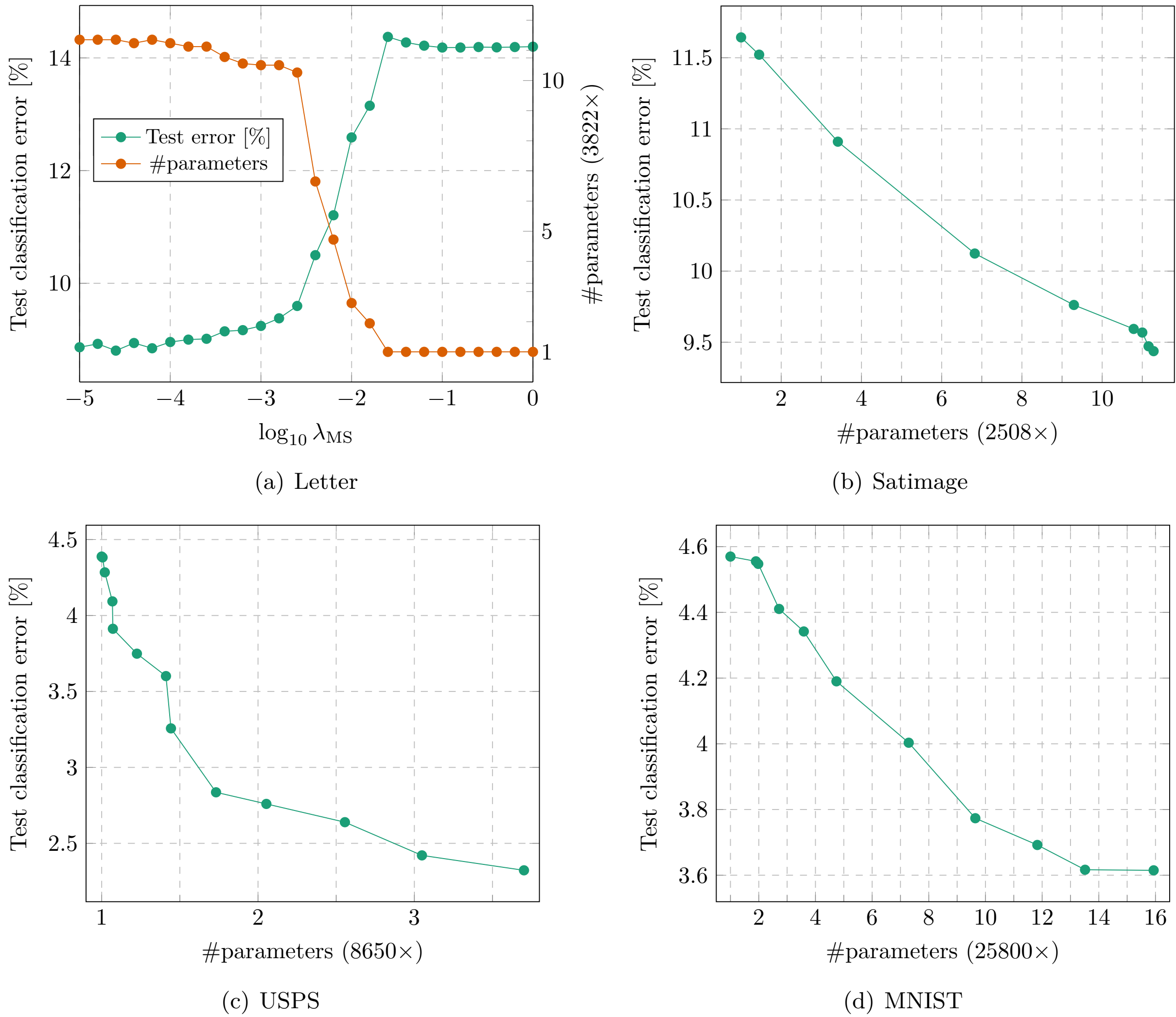
(d) Letter

Figure 7.6: Test classification errors [%] over a hyperparameter of $\mathcal{L}_{\text{HYB}}$. Each point corresponds to a different experiment. For better visualization, the lines show the lower part of the convex hull of equally colored points, i.e., no points of this color are below this line. (a) Results of TAN Heuristic on MNIST with an emphasis on different heuristic feature orderings. (b) Same results as in (a), but with an emphasis on the maximum number of conditioning parents $K \in \{1, 2, 5, 8\}$. (c) Results of TAN Subset on Satimage for several feature orderings. (d) Results of TAN Subset on Letter for several parent subsets (encoded as colors) and $K \in \{2, 8\}$ evaluated for a fixed feature ordering.

rameters of $\mathcal{L}_{\text{HYB}}$, the feature ordering, and the subsets of possible parents are obtained from the best TAN Subset experiment of Table 7.1. We evaluated several trade-off parameters λ_{MS} to obtain different model sizes and accuracies.

Figure 7.7(a) shows how the test error and the number of parameters vary with λ_{MS} on Letter (results are qualitatively similar on the other datasets). For small λ_{MS}, we obtain an unconstrained TAN structure, whereas for large λ_{MS}, we recover the naïve Bayes structure. For intermediate λ_{MS}, we observe increasing test errors and decreasing model sizes as λ_{MS} increases.

Figures 7.7(b)–7.7(d) show the Pareto frontier with respect to model size and accuracy by varying λ_{MS} on Satimage, USPS, and MNIST, respectively. The leftmost point in each figure corresponds to the naïve Bayes model discovered for large λ_{MS}. The rightmost points correspond to unconstrained TAN structures. For intermediate points, we obtain different trade-offs between model size and accuracy. Especially on USPS, a negligible increase in model size is sufficient to



(a) Letter

(b) Satimage

(c) USPS

(d) MNIST

Figure 7.7: Model-size-aware TAN structure learning for BN classifiers. The number of parameters are shown as multiples of those required by the naïve Bayes structure. (a) Test classification error [%] (left y-axis) and number of parameters (right y-axis) over model size trade-off parameter λ_{MS} on Letter. (b)–(d) Pareto optimal models with respect to model size and test classification error obtained by evaluating several λ_{MS} on the remaining datasets.

achieve substantial gains in accuracy compared to the naïve Bayes structure.

Since the CPTs grow by a factor of the number of possible parent values, the granularity of the achievable trade-offs is dataset dependent. On USPS, the average number of values per feature is relatively low (i.e., 3.4), and we can trade off smoothly between model size and accuracy. On Letter, Satimage, and MNIST, the corresponding numbers of values per feature are larger (i.e., 9.1, 11.5 and 13.2, respectively). This translates into larger model size differences between neighboring points (note the axis scale).

7.5 Quantization Experiments

For the BN quantization experiments, we essentially use the same setup as described in Section 7.4. The hyperparameters of $\mathcal{L}_{\mathrm{HYB}}$ were tuned according to setting (II). For parameter quantization using (7.20), we evaluated the total number of bits $B_I + B_F \in \{1, \ldots, 8\}$. We varied the number of integer bits $B_I \in \{1, \ldots, 6\}$ and report for each total number of bits the

result of the best performing B_I. Note that B_F becomes negative for some configurations. In these cases, not every integer value is a possible outcome after quantization.

DNNs were trained according to the cross-entropy loss. We performed stochastic optimization using Adam [62] for 500 epochs using the same mini-batch sizes as in the BN experiments. Each experiment is performed using the three learning rates $\{3 \cdot 10^{-4}, 3 \cdot 10^{-3}, 3 \cdot 10^{-2}\}$, and we report the best result at the end of the three optimization runs. The learning rate is decayed exponentially after each epoch, such that it decreases by a factor of 10^{-4} over the training run. The initial weights are drawn from a uniform distribution whose variance is determined according to [19]. CNN experiments were only conducted on the image datasets USPS and MNIST. For DNNs we normalized the discretized input features to zero mean and unit variance and treat the resulting values as real-valued quantities.

7.5.1 Fixed Parameter Memory Budget

In the first quantization experiment, we investigate the classification performance of several models with a fixed memory budget for their parameters. We compare BN classifiers using a naïve Bayes structure (BNC NB) with fully connected DNNs (FC NNs) and CNNs. The target memory is selected as the number of bits required by BNC NB for a given bit width $B_I + B_F$. We designed DNNs that require approximately the same memory.

For fully connected DNNs, we constrained the number of hidden units d_l in each layer to be equal. We evaluated the bit width $B \in \{1, \ldots, 8\}$, the number of layers $L \in \{2, 3, 4, 5\}$, and performed each experiment once with and once without batch normalization. In case batch normalization is employed, we count the batch normalization parameters as 32 bits, resulting in 64 bits per hidden unit. Batch normalization is not performed in the output layer, where we use biases that are counted as 32 bits per output. We do not use biases in the hidden layers, even when batch normalization is not employed. With this specification, the total number of bits only depends on the number of hidden units. We select the number of hidden units by rounding the real-valued number that would exactly match the target memory.

We proceed similarly for CNNs where we select the number of channels d_l. We consider CNNs with one or two convolutional layers, followed by a fully connected output layer. After each convolutional layer, we downscale the image by a factor of two using max pooling. In case of two convolutional layers, the number of channels of the second layer is constrained to be twice the number of channels of the first layer (i.e., $d_2 = 2d_1$). Again, the batch normalization parameters (if used) incur 64 bits for each channel, and we employ 32 bit biases in the output layer. The resulting real-valued number of channels is rounded separately for the first and the second hidden layer.

If not stated otherwise, DNNs treat the (normalized) discrete input features as real values. We also perform experiments using one-hot encoded input features as outlined in Section 7.1.4 such that BNs and DNNs treat the inputs equally. Note that this increases the number of weights in the first layer for a given number of hidden units.

The best results for a given target number of bits are shown in Figure 7.8. The optimal number of bits per weight is highly dataset dependent. For instance, our BN classifiers with one bit per weight perform reasonably well on USPS, while the performance still improves up to 6–7 bits on Letter.

We confirm that CNNs are extremely memory efficient. Even for the smallest memory budget, CNNs with ReLU activation are more accurate than all other models using the largest memory budget. The activation function is crucial as the accuracy degrades considerably for the sign function.

Fully connected DNNs outperform BN classifiers consistently. This is due to DNNs treating the inputs as real values, which allows them to be more memory efficient by only maintaining one weight per feature rather than one weight per feature value. By spending the gained memory

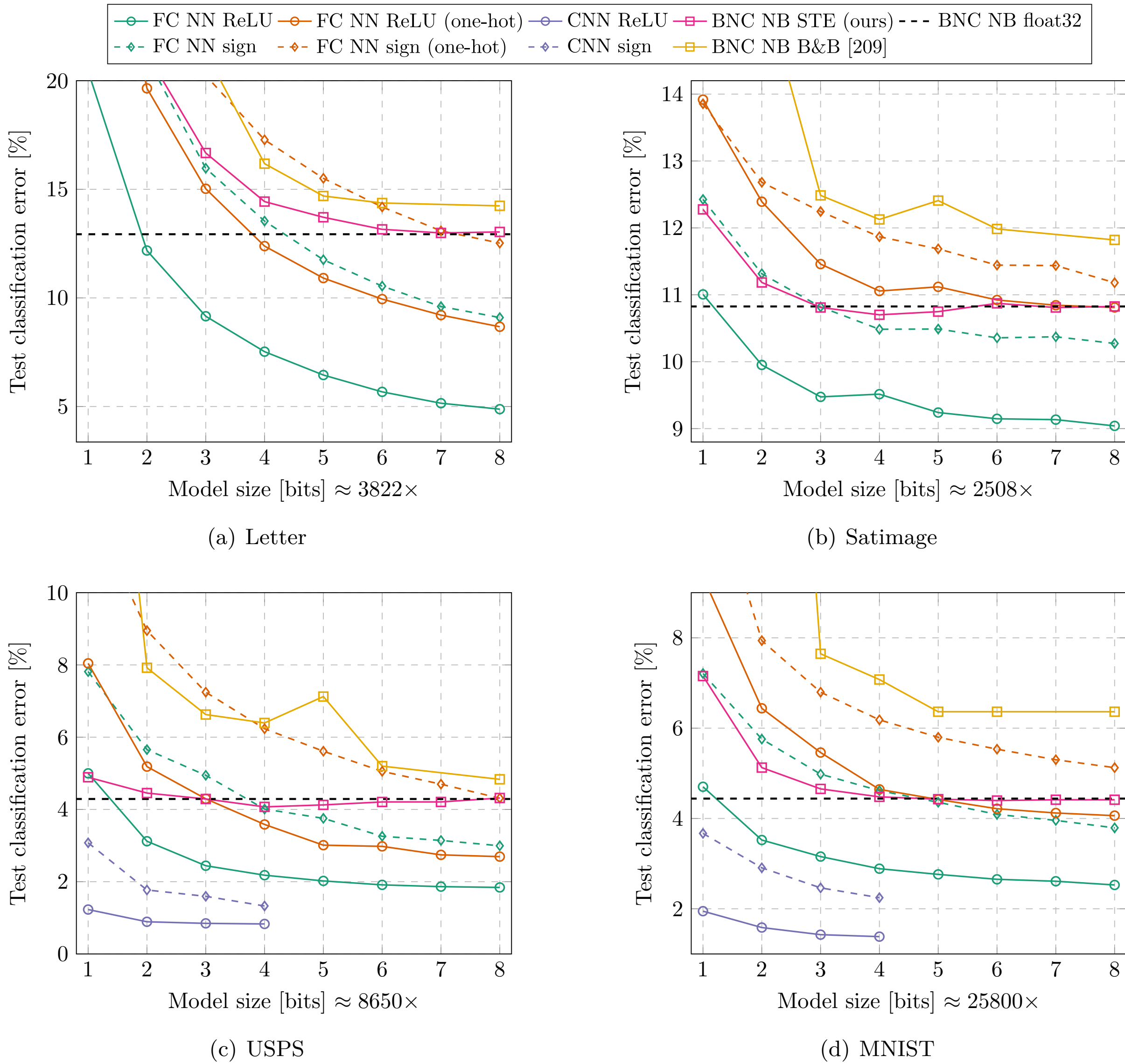


Figure 7.8: Test classification errors [%] over model size budgets in bits. The x-axis shows the model size of BN classifiers with naïve Bayes structure (BNC NB) for given bit widths $B_I + B_F$. Fully connected DNNs (FC NNs) and CNNs are designed to have approximately (due to rounding) the same model size.

into additional layers computing intermediate representations, the performance improves. We verified that the improvements can be attributed to the intermediate representations of DNNs, as logistic regression (i.e., a single layer neural network) with float32 weights performs poorly on the real-valued inputs.

A fairer comparison is obtained by using one-hot encoded inputs for DNNs. Note that for one-hot encoded inputs, it is still possible for DNNs to achieve a lower memory overhead than BN classifiers by (i) employing a hidden layer with fewer units than the number of classes and by (ii) using fewer bits per weight. Especially the case of using fewer bits per weight highlights the importance of a DNN's capability to compute intermediate representations. For instance, on USPS, the performance of BN classifiers with three or more bits can be obtained by a fully connected DNN using fewer bits and by spending the gained memory in an additional layer.

Our quantized BN classifier outperforms the specialized branch-and-bound method (B&B) from [209] by a large margin. From a practical perspective, quantization-aware training fits seamlessly into existing gradient-based learning frameworks and incurs only a negligible com-

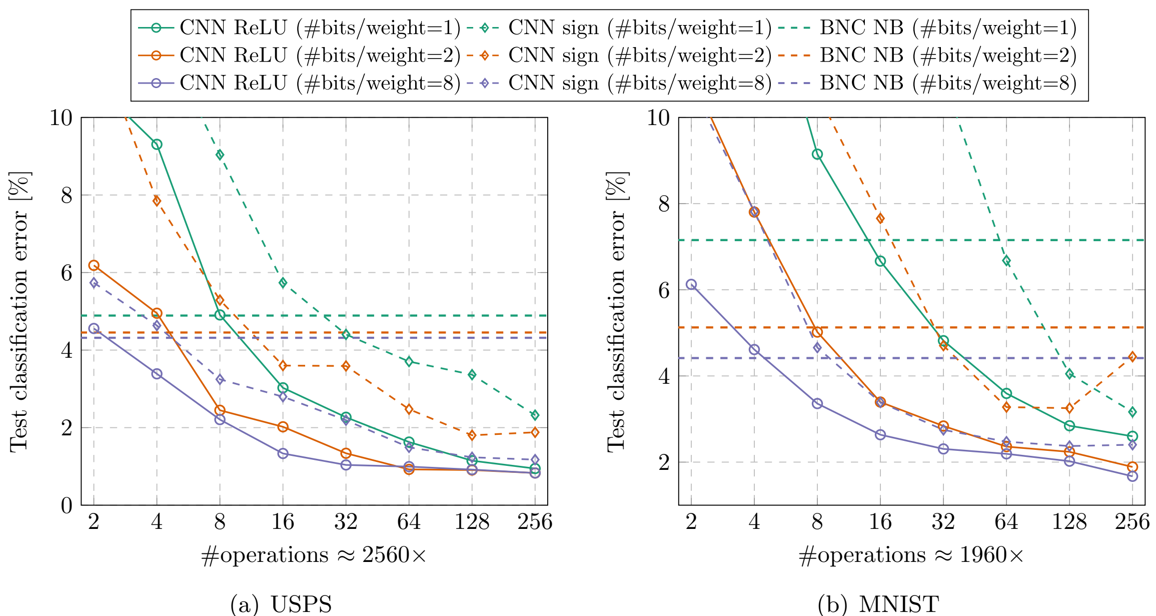


Figure 7.9: Test classification errors [%] over fixed budgets for the number of operations. The x-axis corresponds to multiples of the number of operations required by a BN classifier (BNC).

putational overhead. Branch-and-bound algorithms, on the other hand, are computationally intensive and often rely on carefully selected heuristics to reduce the running time. We also observed that different hyperparameters are optimal for different bit widths $B_I + B_F$. This is in contrast to [209] where the running time of the branch-and-bound algorithm did not allow for an extensive hyperparameter search.

7.5.2 Fixed Number of Operations Budget

We compare BN classifiers using a naïve Bayes structure (BNC NB) with CNNs using a fixed budget for the number of operations. Since BNs require very few operations, we design CNNs that require multiples of that number of operations. Similar to how the CNN architecture is obtained in Section 7.5.1, we select the number of channels to match a given target number of operations. We treat both addition and multiply-accumulate as single operations. Batch normalization and adding biases incur one operation per hidden unit.

Figure 7.9 shows the best results for fixed operation budgets. On USPS and MNIST, CNNs with ReLU activation require at least 2–4× and 4–8× as many operations as a BN, respectively, to achieve a better performance. For the sign activation, an even larger number of operations is required to match the accuracy of the BN. Moreover, CNNs require many operations to achieve their full potential. On USPS, CNNs require at least 64× the operations, and on MNIST, they even require 256× the operations to achieve their best performance.

7.5.3 Quantization for BN Classifiers

Figure 7.10 shows test errors of quantized BN classifiers with naïve Bayes and TAN structures. For each dataset, we used the TAN structure discovered in the best TAN Subset experiment of Table 7.1. Consequently, the test errors achieved by the float32 TAN BN classifiers are rather optimistic, which explains the consistent test error gap to the quantized models on Satimage and MNIST.

Our quantization approach allows us to effectively trade off between accuracy and model size for both BN architectures. The number of bits at which the test error saturates depends, in

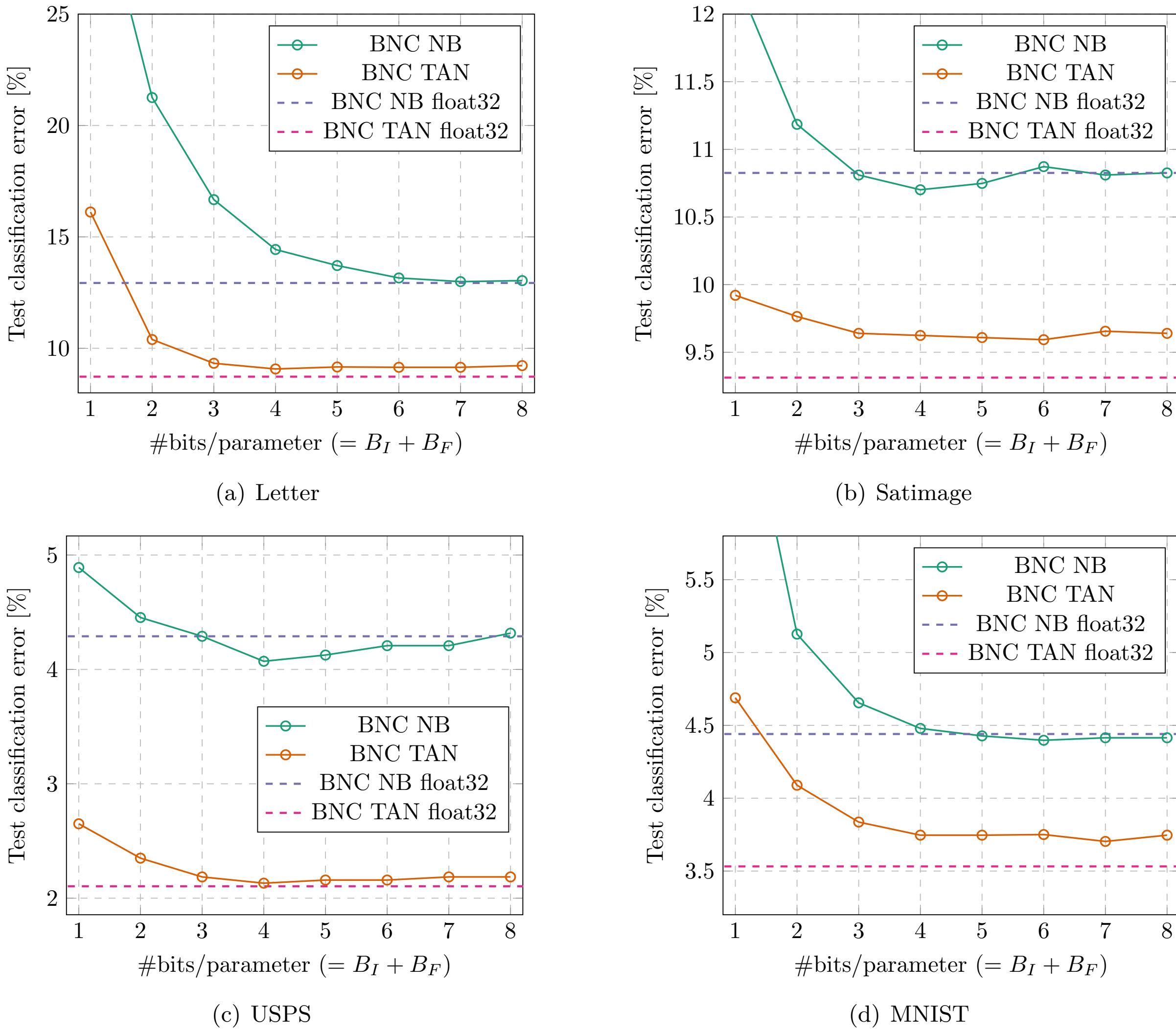


Figure 7.10: Test classification errors [%] over numbers of bits per parameter $B_I + B_F$ for quantized BN classifiers (BNC) with naïve Bayes (NB) and TAN structure. The horizontal lines show the respective test errors for float32 parameters.

addition to the dataset, also on the architecture. The naïve Bayes model is already prone to underfitting such that it suffers more severely than the more expressive TAN structure when using only one or two bits. For instance, when comparing the test error for two bits and eight bits, the difference is rather small for the TAN structure on all datasets, but it is quite substantial for the naïve Bayes model on some datasets.

7.5.4 Comparing Bayesian Network Classifiers and Deep Neural Networks

Finally, we contrast DNNs and BN classifiers with respect to (i) number of bits to store the parameters, (ii) number of operations, and (iii) test error. Figure 7.11 shows Pareto optimal models with respect to these three dimensions, i.e., we cannot improve on these models in one dimension without degrading some other dimension.[28] The models were obtained from the previous quantization experiments in Sections 7.5.1, 7.5.2, and 7.5.3. We do not include results for DNNs operating on one-hot encoded inputs.

BN classifiers require very few operations and achieve a moderate test error. Among BNs, we

[28] Of course, this statement is only true within the set of models discovered in our experiments.

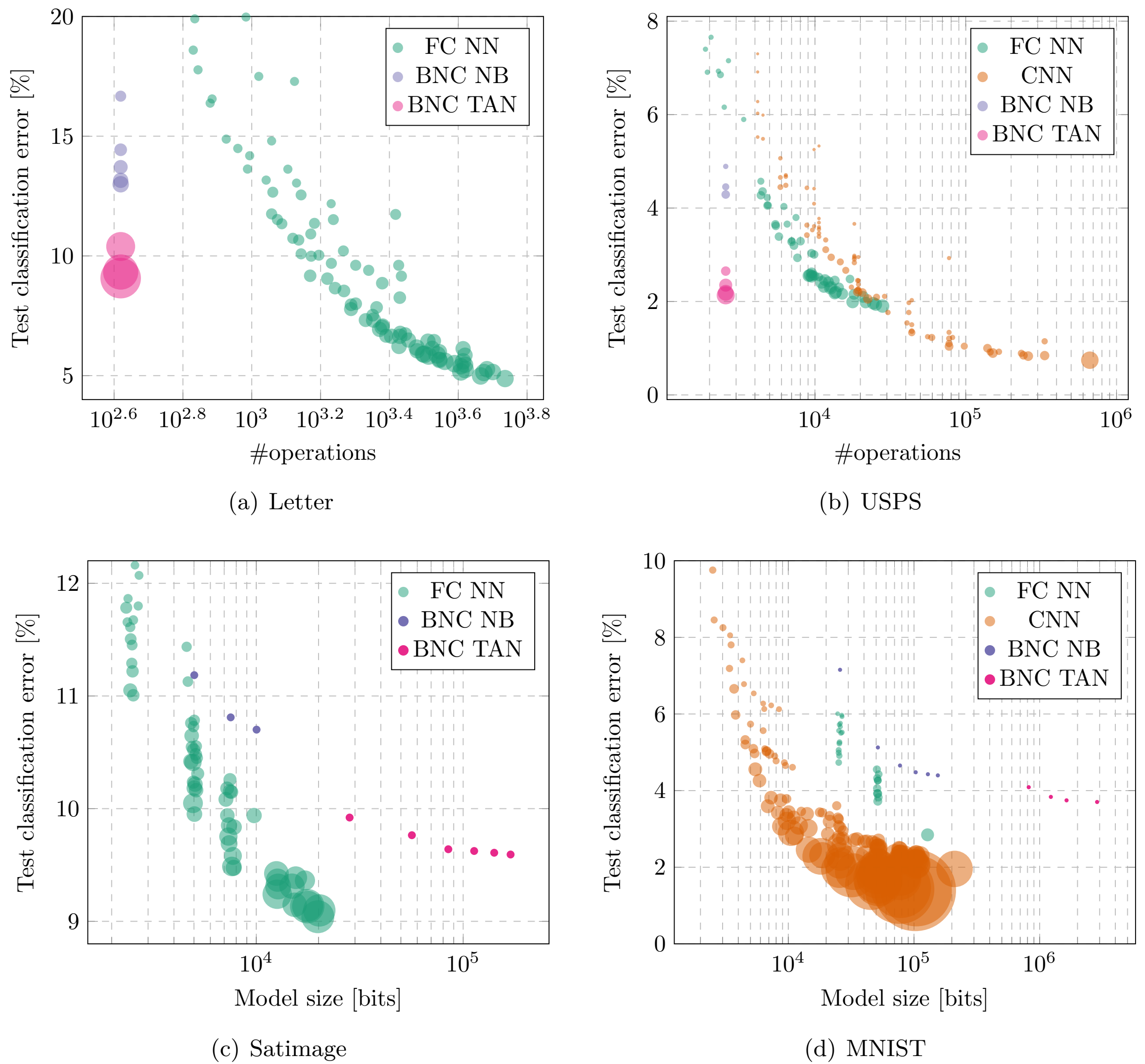


Figure 7.11: Comparison of BN classifiers (BNCs) and DNNs. Each disk corresponds to a Pareto optimal model with respect to test error, number of operations, and memory usage for the parameters. (a), (b): Test classification errors [%] over number of operations required to compute predictions. The area of the disks is proportional to the model size in bits. (c), (d): Pareto optimal models with model size on the x-axis and number of operations encoded as the area of the disks.

can improve the performance by selecting a TAN structure instead of a naïve Bayes structure, but this typically incurs a considerable memory overhead. For instance, on MNIST where the average number of values per feature is relatively high (i.e., 13.2), it is questionable whether the performance gain can be justified, considering that the memory increases by an order of magnitude.

At the same time, DNNs outperform BNs on every dataset in terms of accuracy, but they require substantially more operations to do so. Fully connected DNNs allow for a fine-grained trade-off between accuracy, memory, and operations due to their flexible structure. However, as discussed in Section 7.5.1, the memory efficiency of fully connected DNNs can partly be explained by the fact that they consider the inputs as real-valued quantities. Interestingly, by introducing a *bottleneck* layer exhibiting fewer units than there are output classes, DNNs might even require fewer operations than BNs. This can be seen, for instance, in Figure 7.11(b) on USPS. However, the accuracy degradation in this case is also quite substantial.

Once again, we can see that CNNs are extremely memory efficient, but they require many

operations. For instance, on MNIST, CNNs require up to three orders of magnitude more operations than BNs to achieve their best accuracy.

7.6 Discussion

This chapter is dedicated to answering the question whether other fields of machine learning can benefit from recent advances in (resource-efficient) deep learning. For this purpose, we selected BN classifiers with naïve Bayes and TAN structure as our model of study.

Our first main contribution is closely related to recently developed differentiable NAS techniques such as [184, 187]. In particular, we presented an approach to jointly train the parameters of a BN along with its graph structure through gradient-based optimization. This is accomplished by introducing a discrete distribution over TAN structures and minimizing an expected loss with respect to this distribution. After training, the most probable TAN structure is selected.[29] The method can be easily implemented using modern automatic differentiation frameworks and does not require combinatorial optimization techniques such as greedy hill climbing. The presented method is agnostic to the specific loss and only requires that it is differentiable. In this work, we used a hybrid generative-discriminative loss based on a probabilistic margin criterion. By incorporating a model size penalty into the loss, we are able to effectively trade off between accuracy and model size.

We conducted extensive experiments showing that our method consistently outperforms random TAN structures and Chow-Liu TAN structures. We observed that the selected variable ordering and parent subsets affect the final accuracy of the model. For image data, we proposed a heuristic variable ordering and parent subset selection to further improve performance. The results show that selecting larger parent subsets results in higher accuracy. However, the choice of parent subsets affects the learning behavior such that optimal hyperparameters (e.g., number of epochs, learning rate) might actually depend on the particular parent subsets. This could partly explain why TAN All—which actually considers all possible parents—did not outperform TAN Subset.

Using the model size loss, we obtained accuracy and model size trade-offs ranging from the naïve Bayes model to unconstrained TAN structures. In some cases, introducing few TAN connections was sufficient to considerably increase the accuracy compared to the naïve Bayes structure. The granularity of the achieved trade-offs depends largely on the number of possible values per feature. Consequently, the proposed method is most effective when individual features can take only few values.

Our second main contribution considers quantization-aware parameter learning for BN classifiers using the STE . We highlighted the effectiveness of our approach in extensive experiments. Our method outperforms a specialized branch-and-bound algorithm for learning discrete BN classifiers by a large margin. Moreover, we contrasted quantized BN classifiers with quantized DNNs and identified regimes of model size, number of operations, and test error in which each model class performs best. In particular, BN classifiers require few operations and achieve decent accuracy. CNNs are memory efficient and achieve the highest accuracy, but they require many operations to do so. Fully connected DNNs provide flexible trade-offs. We emphasize that our results also show that quantized DNNs perform well in the small-scale setting which has been hardly addressed in the literature.

In summary, our work shows that recently developed techniques from the deep learning community are transferable to other model classes. We believe that several other techniques, especially among those discussed in Chapter 4, can be successfully transferred to other model classes and, in particular, to BNs.

[29] This is similar to our method presented in Chapter 5 where we introduce discrete distributions over weights.

7.6.1 Limitations and Future Work

The performance of our structure learning method depends on two choices that must be made before training, i.e., (i) a fixed variable ordering and (ii) a particular parent subset of maximum size K. While we have shown that a heuristic variable ordering for image data is beneficial, we did not investigate heuristics for arbitrary data. Heuristic variable orderings based on information theoretic criteria, such as the method proposed in [225], could be promising. Furthermore, methods that exclude cycles by means of continuous optimization (e.g., [224]) might fit naturally into our framework.

Since our method requires a separate CPT per possible parent, considering all possible parents is only feasible for datasets with a moderate number of features. As a remedy, we proposed to select subsets of the possible parents if the number of features is high. Similar to the variable orderings, heuristic selection strategies based on information theoretic criteria could be promising. In our experiments, we have also shown that the influence of the parent subsets can be reduced by increasing K. While this is currently only possible up to a very limited point, using more compact representations of the CPTs might be a solution. We refer to [226] for a concise overview of different CPT representations. In particular, functional representations using DNNs (e.g., [227]) are promising to combine several CPTs in a single shared representation.

Shared representations might also resolve the issue of sparse gradients, i.e., non-zero gradients are obtained only for those CPTs whose corresponding parent is being sampled. As a consequence, the gradients become increasingly sparse for nodes having many possible parents, resulting in severe imbalances between low and high ranked variables in the selected ordering. For a shared representation, the same parameters are updated during training irrespective of the sampled parents, potentially improving the learning behavior.

We believe that the underlying principles of our approach are not restricted to TAN structures. Again, shared parameter representations might be key to apply the proposed method to more flexible structures; in particular, if individual nodes are allowed to have larger numbers of parents. For instance, extending TAN structures by allowing for two additional parents would be a natural next step.

Furthermore, other NAS techniques, such as reinforcement learning based approaches, are worthy of investigation. For DNNs, reinforcement learning based NAS is typically limited by long training times. Since BNs are usually applied to smaller datasets and training takes less time, the original limiting factors of these NAS approaches become less severe.

Future work for quantization includes mixed-precision quantization with individual bit widths at various levels (e.g., per CPT). Furthermore, optimizing the numbers of integer bits B_I and fractional bits B_F using the STE similar as in [128] would be an important contribution. Currently, these hyperparameters are tuned using an extensive grid search.

8 Conclusions and Outlook

In this thesis, we investigated probabilistic methods to obtain resource-efficient machine learning models. The main work horse of this thesis are DNNs which are currently the predominantly used models in the context of machine learning. After starting with a basic introduction of DNNs and methods for training them, we have shown how the Bayesian framework can be used to equip DNNs with a proper probabilistic interpretation. Whereas the resulting Bayesian DNNs are commonly appreciated for their capability of producing prediction uncertainties—i.e., they know what they (don't) know—, we slightly deviated from the conventional path and developed probabilistic methods to obtain resource-efficient models. This resulted in three particular contributions whose origins can, to varying degrees, be traced back to Bayesian modeling. The first two contributions are specific to DNNs. The third contribution investigated whether recent advances in deep learning are transferable to other model classes.

To fit our contributions into the rapidly growing literature, we provided an extensive overview of the recent trends and methods of achieving resource efficiency in deep learning. We identified three major research directions in this scientific field, i.e., quantized DNNs, network pruning, and improving computational efficiency through structural properties. Quantization and pruning approaches are mostly model agnostic and promising tools to reduce the complexity of any given architecture. However, in the long run, it is most likely that structural considerations are key to obtaining impressive accuracies with significantly smaller models. This is at least suggested by a recurring pattern in the evolution of modern DNN architectures: Initially, some incredibly large architectures pave the way towards new state-of-the-art performances. Subsequently, these performances are also achieved by much smaller DNNs at whose core we can find simple but well-conceived new building blocks. While until recently such novel building blocks were manually designed, the emerging field of NAS appears to be a promising candidate to automate this process. We believe that NAS will be of high practical relevance in the future especially due to its capability of discovering application and hardware-specific structures.

We contributed to the resource efficiency literature with our methods presented in Chapters 5, 6, and 7. In the remainder, we review the most important findings of our contributions before we provide (what we believe to be) promising directions of future research. For an in-depth discussion of the individual contributions, we refer to the discussion sections at the end of the individual chapters.

Chapter 5: In Chapter 5, we presented a method that is closely related to variational inference—a commonly used technique for approximate Bayesian inference. In particular, we introduced a method to train discrete distributions over the weights of a DNN. Once training of the discrete weight distributions has finished, we infer concrete weights from those distributions either by taking their most probable weights or by sampling from them. Training discrete distributions has the advantage that we can perform gradient-based optimization. Importantly, this holds true even for discrete activations functions. We utilized this freedom to train DNNs with the binary sign activation function, and we evaluated ternary, quaternary, and quinary weights.

Our method supports arbitrary numbers of discrete weight values which extends previous works that are tailored to binary and ternary weights. This is accomplished by introducing new

parameterization and initialization schemes for the weight distributions. Moreover, by introducing a distribution-aware max pooling approximation, we extend previous works that take the distribution only indirectly into account. Our experiments show that our new components, parameterization and max pooling approximation, facilitate training and result in higher accuracy.

By increasing the number of discrete weight values, we can increase the expressiveness of a discrete-valued DNN. In this way, we are able to trade off between computational costs and accuracy. On the evaluated datasets, our method improved in most cases when more discrete weight values were used. By averaging predictions of a varying number of DNNs sampled from the weight distributions, our method provides another means of trading off between computational costs and accuracy.

We observed that a proper initialization method is crucial to obtain state-of-the-art performances. Our initialization method (similar as in previous works) is based on a pre-trained real-valued DNN. We found that the ReLU activation resulted in higher accuracies than tanh when used during pre-training. This is somewhat surprising since the functional shapes of ReLU and the ultimately used sign function are rather different.

Chapter 6: Our second method proposed in Chapter 6 builds upon sampling methods—the second pillar of approximate Bayesian inference besides variational inference. In this work, we introduced a DP on top of the weight prior of Bayesian DNNs. This results in a weight sharing that is subsequently exploited to drastically reduce the number of parameters of an ensemble of DNNs. Although being a theoretically elegant solution to achieve weight sharing in DNNs, several computational challenges arise in practice. To make sampling based posterior inference feasible, we introduced several approximations and algorithmic techniques.

In our experiments, we demonstrated that our method is capable of substantially reducing the number of parameters of an ensemble. Our method outperforms DNNs with randomly shared weights in most experiments. In some cases our method even outperforms DNNs without weight sharing which indicates a regularizing effect. By varying the DP parameter α_{dp}, our method allows us to trade off between the number of parameters and the accuracy. Especially for very few parameters our method substantially outperforms randomly shared weights.

Chapter 7: Our last contribution presented in Chapter 7 addresses the question whether recent advances in (resource-efficient) deep learning are transferable to other model classes. We positively answered this question for BN classifiers with naïve Bayes and TAN structures. For this purpose, we considered two particular methods, i.e., (i) structure learning and (ii) parameter quantization.

Our structure learning approach is closely related to recently proposed differentiable NAS techniques. The proposed method allows us to jointly train the BN parameters and its TAN structure by means of gradient-based optimization. This is accomplished by introducing a discrete distribution over TAN structures and minimizing an expected loss with respect to this distribution. Once the distribution has been trained, the most probable TAN structure is selected. The method can be easily implemented using modern automatic differentiation frameworks and does not require combinatorial optimization techniques such as greedy hill climbing. By incorporating a model size penalty to the loss, various trade offs between accuracy and model size can be obtained.

For parameter quantization, we perform quantization-aware training using the STE. The STE replaces the zero gradient of quantization functions by the non-zero gradient of a different function of similar shape. In this way, the continuous parameters can be updated using gradient-based learning. Once training has finished, only the quantized parameters are kept.

We demonstrated the effectiveness of our approaches in extensive experiments. The discovered TAN structures consistently outperform random TAN structures and generatively trained Chow-Liu TAN structures. Using a heuristic variable ordering and parent subset selection for image

data, we were able to further improve the performance. We showed that the model size penalty allows us to obtain various trade-offs between model size and accuracy, ranging from the naïve Bayes structure to unconstrained TAN structures. We observed that slightly increasing the model size can yield substantial gains in accuracy.

In extensive quantization experiments, we showed that quantization-aware training is a highly effective method to trade off between parameter bit width and accuracy. Our method outperforms a specialized branch-and-bound algorithm tailored to parameter quantization. We conducted a comprehensive comparison of quantized BNs with quantized small-scale DNNs. Our experiments show that both model classes offer benefits in different regimes of computational efficiency and model accuracy. In particular, BN classifiers require few operations and achieve decent accuracy, CNNs are memory efficient and achieve the lowest error at the cost of many operations, and fully connected DNNs provide flexible trade-offs. We emphasize that our experiments demonstrate the effectiveness of quantization-aware training for *small-scale* DNNs. Therefore, our work also closes a gap in the vast DNN quantization literature which is mostly concerned with large architectures and datasets. In summary, our work shows that other areas of machine learning might benefit as well from recent advances in deep learning.

8.1 Limitations and Future Work

In the following, we discuss limitations of our contributions and (what we believe to be) promising directions of future research. Again, we refer to the discussion sections of the individual chapters for a thorough treatment of the limitations of the individual topics.

Chapter 5: Our method for learning discrete-valued DNNs in Chapter 5 relies crucially on the initialization with a pre-trained real-valued DNN. However, there are several open questions that should be addressed by future work. Several experiments using different pre-training conditions produced mixed results and it remains mostly unclear what properties of a pre-trained DNN are responsible for the final accuracy. Furthermore, it is not fully understood why the ReLU activation is suitable to initialize a DNN with sign activations. Related to this is understanding the role of batch normalization and dropout for successful pre-training and initialization. Due to the necessity of pre-training, our method can, to a certain extent, also be seen as an instance of transfer learning or knowledge distillation. Future work should investigate whether more sophisticated techniques from these areas are applicable to facilitate training.

Our method introduces non-negligible computational overhead during training. This might be prohibitive for very large architectures and datasets such as ImageNet. Therefore, increasing scalability of the proposed method would be an important contribution. Exploring other parameterizations of the weight distributions, especially by directly parameterizing the weight mean and the weight variance, might result in a better behaved loss surface. At the same time, such parameterizations might also reduce the computational overhead during training.

In some of our experiments, we observed mixed behavior for different discrete weight types. For instance, there is no explicit zero weight for quaternary weights which might be a limitation. Future work should investigating other types of (non-symmetric) discrete weights and weights with trainable quantization levels. Finally, we did not evaluate the quality of prediction uncertainties obtained from the trained distributions.

Chapter 6: The major drawback of our DP based weight sharing scheme presented in Chapter 6 is its limitation to small architectures and small to medium sized datasets. Furthermore, the method does not easily generalize to other architectures such as CNNs and RNNs. We believe that more global approximation schemes based on low order Taylor approximations (similar as in [83]) are promising to achieve this. While our method is tailored to ensembles of DNNs, it is

worthy to explore how individual DNNs can benefit from a DP based weight sharing. Assuming that the computational challenges have been solved, a variation of the proposed method might be a viable alternative to other clustering based weight sharing schemes such as k-means (e.g., see [144]). Finally, a Bayesian treatment of the hyperparameters by introducing another level to the Bayesian hierarchy could reduce the influence of the fixed hyperparameters.

Chapter 7: The proposed structure learning approach assumes a fixed variable ordering. Furthermore, if the number of variables is large, our method is restricted to selecting a subset of possible parents. Both of these choices restrict the space of TAN structures under consideration, and our experiments show that these choices affect the final accuracy. For the variable ordering, we identified heuristic orderings and continuous optimization techniques to avoid cycles as potential solutions. For the parent subset selection, we identified heuristic selection strategies and shared parameter representations as promising solutions. Shared parameter representations might also serve to eliminate sparse gradients, potentially resulting in improved learning behavior. We believe that the presented structure learning techniques are not limited to TAN structures, and extending our method to other BN structures is an interesting direction of future research.

For our quantization approach, the extension to mixed-precision quantization is a natural next step. Furthermore, continuous optimization of the numbers of integer and fractional bits using the STE would eliminate the need to perform an extensive hyperparameter search.

9 List of Publications

J. Rock, **W. Roth**, M. Toth, P. Meissner, and F. Pernkopf; *Resource-Efficient Deep Neural Networks for Automotive Radar Interference Mitigation*; In: IEEE Journal of Selected Topics in Signal Processing; vol. 15 (4), pp. 927–940, 2021

W. Roth, G. Schindler, H. Fröning, and F. Pernkopf; *On Resource-Efficient Bayesian Network Classifiers and Deep Neural Networks*; In: International Conference on Pattern Recognition; pp. 10297–10304, 2020

D. Peter, **W. Roth**, and F. Pernkopf; *Resource-efficient DNNs for Keyword Spotting using Neural Architecture Search and Quantization*; In: International Conference on Pattern Recognition; pp. 9273–9279, 2020

M. Huber, G. Schindler, **W. Roth**, H. Fröning, C. Schörkhuber, and F. Pernkopf; *Towards Real-Time Single-Channel Singing-Voice Separation with Pruned Multi-Scaled DenseNets*; In: IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP); pp. 806–810, 2020

L. Pfeifenberger, M. Zöhrer, **W. Roth**, G. Schindler, H. Fröning, and F. Pernkopf; *Resource-Efficient Speech Mask Estimation for Multi-Channel Speech Enhancement*; arXiv: abs/2007.11477, 2020

J. Rock, **W. Roth**, P. Meissner, and F. Pernkopf; *Quantized Deep Neural Networks for Radar Interference Mitigation*; In: European Conference on Machine Learning - ITEM Workshop; 2020

W. Roth and F. Pernkopf; *Differentiable TAN Structure Learning for Bayesian Network Classifiers*; In: International Conference on Probabilistic Graphical Models (PGM); pp. 389–400, 2020

G. Schindler, **W. Roth**, F. Pernkopf, and H. Fröning; *Parameterized Structured Pruning for Deep Neural Networks*; In: International Conference on Machine Learning, Optimization, and Data Science (LOD); pp. 16–27, 2020

W. Roth, G. Schindler, M. Zöhrer, L. Pfeifenberger, R. Peharz, S. Tschiatschek, H. Fröning, F. Pernkopf, and Z. Ghahramani, *Resource-Efficient Neural Networks for Embedded Systems*; arXiv: abs/2001.03048, 2020

W. Roth and F. Pernkopf; *Bayesian Neural Networks with Weight Sharing Using Dirichlet Processes*; In: IEEE Transactions on Pattern Analysis and Machine Intelligence; vol. 42 (1), pp. 246–252, 2020

W. Roth, G. Schindler, H. Fröning, and F. Pernkopf; *Training Discrete-Valued Neural Networks with Sign Activations Using Weight Distributions*; In: European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD);

pp. 382–398, 2019

B. K. Aichernig, R. Bloem, M. Ebrahimi, M. Horn, F. Pernkopf, **W. Roth**, A. Rupp, M. Tappler, and M. Tranninger; *Learning a Behavior Model of Hybrid Systems Through Combining Model-Based Testing and Machine Learning*; In: International Conference on Testing Software and Systems (ICTSS); pp. 3–21, 2019

W. Roth, R. Peharz, S. Tschiatschek, and F. Pernkopf; *Hybrid Generative-Discriminative Training of Gaussian Mixture Models*; In: Pattern Recognition Letters; vol. 112, pp. 131–137, 2018

W. Roth and F. Pernkopf; *Variational Inference in Neural Networks using an Approximate Closed-Form Objective*; In: Conference on Neural Information Processing Systems, Workshop on Bayesian Deep Learning; 2016

F. B. Pokorny, R. Peharz, **W. Roth**, M. Zöhrer, F. Pernkopf, P. B. Marschik, and B. W. Schuller; *Manual Versus Automated: The Challenging Routine of Infant Vocalisation Segmentation in Home Videos to Study Neuro(mal)development*; In: Interspeech - International Conference on Spoken Language Processing; pp. 2997–3001, 2016

A Datasets

The following datasets were used for our own experiments throughout the thesis. We provide descriptions of the datasets in their default settings. Whenever we deviate from this default setting, we state this explicitly in the relevant experimental context.

A.1 MNIST

The MNIST dataset [228] contains grayscale images of size 28×28 pixels showing handwritten digits from 0–9. The training set contains 60,000 images and the test set contains 10,000 images. We split the training set into a training set of 50,000 training images and 10,000 validation images. We normalize the pixels to be in the range $[-1, 1]$.

We consider two settings for the MNIST dataset. (i) In the *permutation-invariant (PI)* setting each pixel is treated as independent feature. In this setting it is not allowed to exploit the image structure and to take pixel locality into account, i.e., CNNs are not allowed. (ii) In the *unconstrained* setting we keep the image structure and use CNNs.

Some examples of the dataset are shown in Figure A.1(a). The MNIST dataset is used in Chapters 5, 6, and 7. In Chapter 5 and Chapter 7, we conduct experiments using both settings (i) and (ii). In Chapter 6, we only conduct experiments for the permutation-invariant setting. To distinguish between the two settings in Chapter 5, we refer to the permutation-invariant setting as MNIST (PI). In Chapter 7, we do not make the setting explicit and it should be clear from the context which setting is being used.

A.2 Variants of MNIST

To obtain more challenging datasets, the MNIST dataset has been transformed by various operations [229]. In particular, there are the following variants:

- MNIST-basic: This dataset has not been transformed. The dataset is merely split differently into training, validation, and test set (see below).
- MNIST-bg (background): The background pixels of the images are replaced by random image patches.
- MNIST-bg-rnd (background random): The background pixels of the images are set to uniformly distributed random pixel values.
- MNIST-rot (rotated): The images are randomly rotated.
- MNIST-rot-bg (rotated background): The transformations of MNIST-rot and MNIST-bg are combined.

Compared to the original MNIST dataset, the variants of MNIST exhibit a different splitting into training, validation, and test set. In particular, there are 10,000 training samples, 2,000

validation samples, and 50,000 test samples. The remaining attributes (image size, normalization) are equivalent to the original MNIST dataset. We only perform experiments using the permutation-invariant setting on these datasets.

Some examples of these datasets are shown in Figures A.1(a)–A.1(e). The variants of the MNIST dataset are used in Chapter 6.

A.3 Cifar-10 and Cifar-100

The Cifar-10 dataset [230] contains 32×32 pixel RGB images showing objects from ten different categories. The dataset contains 50,000 training images and 10,000 test images. We split the training set into 45,000 training images and 5,000 validation images. The pixels are normalized to be in the range $[-1, 1]$. Cifar-100 is similar to Cifar-10 except that the task is to assign an image to one of 100 object categories.

Some examples of the Cifar-10 and the Cifar-100 datasets are shown in Figure A.1(f) and Figure A.1(g), respectively. We conduct experiments on the Cifar-10 and the Cifar-100 dataset in Chapter 5.

A.4 SVHN

The SVHN dataset [231] contains 32×32 pixel RGB images showing picture sections of house numbers that need to be classified to the digits 0–9. The dataset is split into 604,388 training images and 26,032 test images. We follow the procedure of [232] to split the training set into 598,388 training images and 6,000 validation images. Once again, we normalize pixels to be in the range $[-1, 1]$.

Some examples of the SVHN dataset are shown in Figure A.1(h). We conduct experiments on the SVHN dataset in Chapter 5.

A.5 USPS

The USPS dataset contains 16×16 grayscale images showing handwritten digits from 0–9 [233, 234]. The images were extracted from zip codes of mail envelopes. We consider the dataset in the permutation-invariant and the unconstrained setting (see also Appendix A.1). It should be clear from the context which of these settings is used.

We use the particular dataset from [209] containing 11,000 images. In their version of the dataset, the features have been discretized according to the method from [223]. We split the data into two thirds of training samples (i.e., 7,340) and one third of test samples (i.e., 3,660).

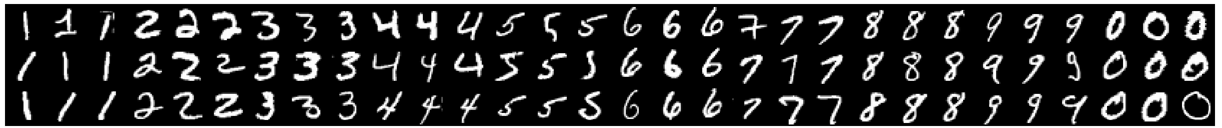
Some examples of the USPS dataset are shown in Figure A.1(i). We conduct experiments on the USPS dataset in Chapter 7.

A.6 UCI Datasets for Classification

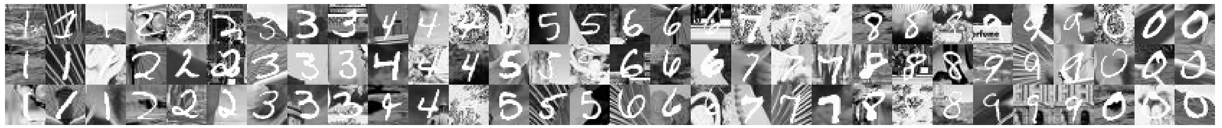
We used the following classification datasets from the UCI repository [235] for our experiments in Chapter 7.

A.6.1 Letter

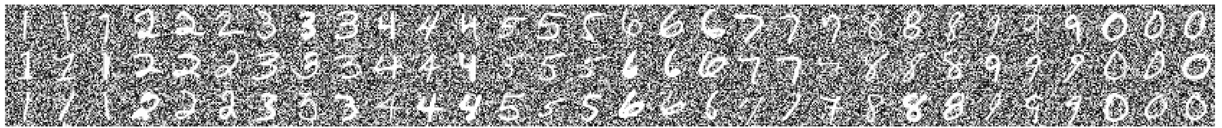
The Letter dataset [236] contains 20,000 samples, describing one of 26 English letters using 16 numerical features (statistical moments and edge counts) extracted from images. We split the data into two thirds of training samples (i.e., 13,342) and one third of test samples (i.e., 6,658).



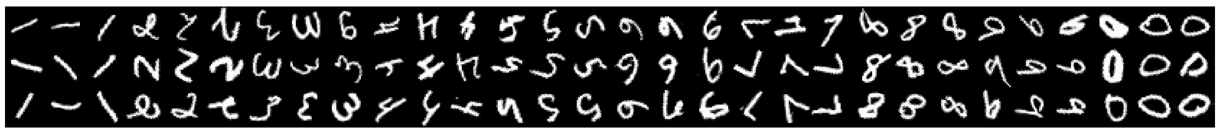
(a) MNIST / MNIST-basic



(b) MNIST-bg (background)

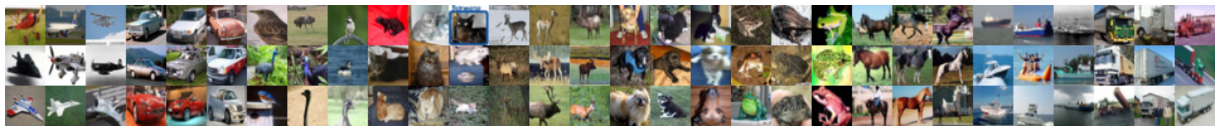


(c) MNIST-bg-rnd (background random)

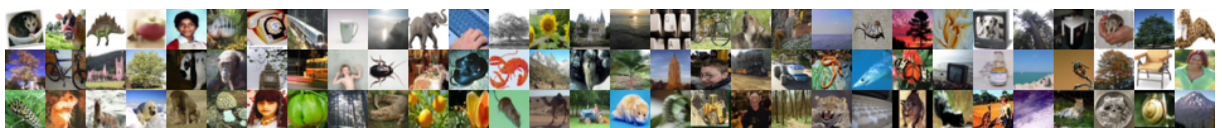


(d) MNIST-rot (rotated)

(e) MNIST-rot-bg (rotated background)



(f) Cifar-10



(g) Cifar-100

(h) SVHN

(i) USPS

Figure A.1: Examples of the benchmark image classification datasets used in this thesis.

A.6.2 Satimage

The Satimage dataset consists of 6,435 samples containing multispectral values of 3×3 pixel neighborhoods in satellite images, resulting in a total of 36 features. The task is to classify the central pixel of these image patches to one of the six categories red soil, cotton crop, gray soil, damp gray soil, soil with vegetation stubble, and very damp gray soil. Since this dataset is rather small, we perform 5-fold cross-validation to estimate the test error.

A.7 UCI Datasets for Regression

We used several regression datasets from the UCI repository [235] for our experiments in Chapter 6. The task of each dataset is to predict a continuous scalar target variable. Since these datasets are rather small, the test errors and test log-likelihoods are estimated using 5-fold cross-validation.

A.7.1 Abalone

The Abalone dataset [237] contains 4,177 samples. The task is to predict the age of an abalone using eight measured attributes, i.e., the sex, three length attributes, and four weight attributes. The actual age is obtained by cutting the shell and counting rings through a microscope.

A.7.2 Boston Housing

The Boston Housing dataset [238] (or, in short, Housing) contains 506 samples. The task is to predict the median value of owner-occupied homes in suburbs of Boston. Each sample consists of 13 features containing information about a particular suburb, e.g., per capita crime rate, nitric oxides concentration, and pupil-teacher ratio.

A.7.3 Concrete Compressive Strength

The Concrete Compressive Strength dataset [239] (or, in short, Concrete) contains 1,030 samples. The task is to predict the concrete compressive strength from eight features, i.e., the age and seven attributes describing the mass of certain ingredients per cubic meter. The actual concrete compressive strength was determined in a laboratory.

A.7.4 Combined Cycle Power Plant

The Combined Cycle Power Plant dataset [240] (or, in short, Power Plant) contains 9,568 samples. The task is to predict the net hourly electrical energy output of the power plant from four average hourly measurements, i.e., ambient temperature, atmospheric pressure, relative humidity, and vacuum (exhaust steam pressure).

A.7.5 Wine Quality

There are two versions of the Wine Quality dataset [241], both of which exhibit the same data format. (i) The red wine version (referred to as WineQ-red) contains 1,599 samples and (ii) the white wine version (referred to as WineQ-white) contains 4,898 samples. The task is to predict the wine quality obtained by expert ratings based on eleven measurements, e.g., alcohol, pH value, and sulfates.

B

Useful Calculations

B.1 Full Covariance Gaussian Approximation of the Activation Distribution

We derive a Gaussian approximation for the activations $\mathbf{a}^l$ of layer l given the inputs $\mathbf{x}^{l-1}$ from the previous layer and a weight matrix $\mathbf{W}^l$. We assume that the inputs $\mathbf{x}^{l-1}$ are independent from the weights $\mathbf{W}^l$. Furthermore, we assume that the individual weights $w_{i,k}$ are independent. The activation a_i^l is computed as

$$a_i^l = \sum_k w_{i,k}^l x_k^{l-1}. \tag{B.1}$$

The mean $\mathbb{E}[a_i^l]$ of the Gaussian approximation is given by

$$\mathbb{E}[a_i^l] = \sum_k \mathbb{E}[w_{i,k}^l] \mathbb{E}[x_k^{l-1}]. \tag{B.2}$$

To compute the covariance $\operatorname{cov}(a_i^l, a_j^l)$, we first determine the covariance of individual summands. These are given as

$$\begin{aligned}
\operatorname{cov}(w_{i,k}^l x_k^{l-1}, w_{j,k'}^l x_{k'}^{l-1}) &= \mathbb{E}[(w_{i,k}^l x_k^{l-1} - \mathbb{E}[w_{i,k}^l x_k^{l-1}])(w_{j,k'}^l x_{k'}^{l-1} - \mathbb{E}[w_{j,k'}^l x_{k'}^{l-1}])] && \text{(B.3)}\\
&= \mathbb{E}[w_{i,k}^l x_k^{l-1} w_{j,k'}^l x_{k'}^{l-1}] - \mathbb{E}[w_{i,k}^l]\mathbb{E}[x_k^{l-1}]\mathbb{E}[w_{j,k'}^l]\mathbb{E}[x_{k'}^{l-1}] && \text{(B.4)}\\
&= \mathbb{E}[w_{i,k}^l w_{j,k'}^l]\mathbb{E}[x_k^{l-1} x_{k'}^{l-1}] - \mathbb{E}[w_{i,k}^l]\mathbb{E}[x_k^{l-1}]\mathbb{E}[w_{j,k'}^l]\mathbb{E}[x_{k'}^{l-1}]. && \text{(B.5)}
\end{aligned}$$

For $i \neq j$ or $k \neq k'$, the term $\mathbb{E}[w_{i,k}^l w_{j,k'}^l]$ factorizes into $\mathbb{E}[w_{i,k}^l]\mathbb{E}[w_{j,k'}^l]$. For $i = j$ and $k = k'$, we obtain $\mathbb{E}[(w_{i,k}^l)^2] = \mathbb{E}[w_{i,k}^l]\mathbb{E}[w_{j,k'}^l] + \mathbb{V}[w_{i,k}^l]$, in which case we have to take care of the additional variance term $\mathbb{V}[w_{i,k}^l]$. Using this observation and by factorizing out common terms, we obtain

$$\begin{aligned}
&\operatorname{cov}(w_{i,k}^l x_k^{l-1}, w_{j,k'}^l x_{k'}^{l-1}) && \text{(B.6)}\\
&= \mathbb{E}[w_{i,k}^l]\mathbb{E}[w_{j,k'}^l]\left(\operatorname{cov}(x_k^{l-1}, x_{k'}^{l-1})\right) + \mathbb{I}[i=j]\mathbb{I}[k=k']\mathbb{V}[w_{i,k}^l]\mathbb{E}[(x_k^{l-1})^2]. && \text{(B.7)}
\end{aligned}$$

We can now compute $\operatorname{cov}(a_i^l, a_j^l)$ as

$$\begin{aligned}
\operatorname{cov}\Big(a_i^l, a_j^l\Big) &= \operatorname{cov}\left(\sum_k w_{i,k}^l x_k^{l-1}, \sum_{k'} w_{j,k'}^l x_{k'}^{l-1}\right) = \sum_k \sum_{k'} \operatorname{cov}\Big(w_{i,k}^l x_k^{l-1}, w_{j,k'}^l x_{k'}^{l-1}\Big) && \text{(B.8)}\\
&= \sum_k \sum_{k'} \mathbb{E}[w_{i,k}^l]\mathbb{E}[w_{j,k'}^l]\operatorname{cov}(x_k^{l-1}, x_{k'}^{l-1}) + \mathbb{I}[i=j]\sum_k \mathbb{V}[w_{i,k}^l]\mathbb{E}[(x_k^{l-1})^2]. && \text{(B.9)}
\end{aligned}$$

B.2 Expectation of a Quadratic Form with respect to a Gaussian

We derive an exact solution for the expectation of a quadratic form with respect to a Gaussian in D dimensions. Let $\mathbf{H} \in \mathbb{R}^{D \times D}$. We have

$$\mathbb{E}_{\mathbf{u}\sim\mathcal{N}(\boldsymbol{\mu},\boldsymbol{\Sigma})}[(\mathbf{u}-\tilde{\mathbf{u}})^\top \mathbf{H}(\mathbf{u}-\tilde{\mathbf{u}})] = \sum_{i=1}^{D}\sum_{j=1}^{D} H_{i,j}\,\mathbb{E}_{\mathbf{u}\sim\mathcal{N}(\boldsymbol{\mu},\boldsymbol{\Sigma})}[(u_i-\tilde{u}_i)(u_j-\tilde{u}_j)] \tag{B.10}$$

$$= \sum_{i=1}^{D}\sum_{j=1}^{D} H_{i,j}\,\mathbb{E}_{\mathbf{u}\sim\mathcal{N}(\boldsymbol{\mu},\boldsymbol{\Sigma})}[(u_iu_j - u_i\tilde{u}_j - \tilde{u}_iu_j + \tilde{u}_i\tilde{u}_j)] \tag{B.11}$$

$$= \sum_{i=1}^{D}\sum_{j=1}^{D} H_{i,j}\,[(\Sigma_{i,j} + \mu_i\mu_j - \mu_i\tilde{u}_j - \tilde{u}_i\mu_j + \tilde{u}_i\tilde{u}_j)] \tag{B.12}$$

$$= \sum_{i=1}^{D}\sum_{j=1}^{D} H_{i,j}\Sigma_{i,j} + H_{i,j}(\mu_i-\tilde{u}_i)(\mu_j-\tilde{u}_j) \tag{B.13}$$

$$= \mathrm{tr}(\mathbf{H}^\top\boldsymbol{\Sigma}) + (\boldsymbol{\mu}-\tilde{\mathbf{u}})^\top\mathbf{H}(\boldsymbol{\mu}-\tilde{\mathbf{u}}), \tag{B.14}$$

where $\mathrm{tr}(\cdot)$ denotes the trace of a square matrix, i.e., the sum of its diagonal entries, and we used the identity $\mathbb{E}[u_iu_j] = \mathrm{cov}(u_i,u_j) + \mathbb{E}[u_i]\mathbb{E}[u_j]$ in (B.12).

We can also use (B.14) to approximate the expectation of non-quadratic functions $f(\mathbf{u})$. For this purpose, we first approximate the non-quadratic function $f(\mathbf{u})$ by its second-order Taylor approximation around the mean $\boldsymbol{\mu}$ as

$$f(\mathbf{u}) \approx \tilde{f}(\mathbf{u}) = f(\boldsymbol{\mu}) + (\mathbf{u}-\boldsymbol{\mu})^\top\mathbf{g} + \frac{1}{2}(\mathbf{u}-\boldsymbol{\mu})^\top\mathbf{H}(\mathbf{u}-\boldsymbol{\mu}) \tag{B.15}$$

with the respective gradient and Hessian matrix

$$\mathbf{g} = \nabla_{\mathbf{u}} f(\mathbf{u})\big|_{\mathbf{u}=\boldsymbol{\mu}} \quad \text{and} \quad \mathbf{H} = \nabla^2_{\mathbf{u}} f(\mathbf{u})\Big|_{\mathbf{u}=\boldsymbol{\mu}}. \tag{B.16}$$

The expectation $\mathbb{E}[f(\mathbf{u})]$ is then approximated by $\mathbb{E}[\tilde{f}(\mathbf{u})]$ using (B.14). This yields

$$\mathbb{E}_{\mathbf{u}\sim\mathcal{N}(\boldsymbol{\mu},\boldsymbol{\Sigma})}[\tilde{f}(\mathbf{u})] = f(\boldsymbol{\mu}) + \frac{1}{2}\mathrm{tr}(\mathbf{H}^\top\boldsymbol{\Sigma}). \tag{B.17}$$

Note that the linear term in (B.15) and the second term in (B.14) have vanished due to the particular choice of performing the Taylor expansion around $\boldsymbol{\mu}$. A simpler approximation is obtained by assuming that any of $\mathbf{H}$ or $\boldsymbol{\Sigma}$ is diagonal. In this case, (B.17) simplifies to

$$\mathbb{E}_{\mathbf{u}\sim\mathcal{N}(\boldsymbol{\mu},\boldsymbol{\Sigma})}[\tilde{f}(\mathbf{u})] = f(\boldsymbol{\mu}) + \frac{1}{2}\sum_{i=1}^{D} H_{i,i}\Sigma_{i,i}. \tag{B.18}$$

We emphasize that a second-order Taylor approximation $\tilde{f}(\mathbf{u})$ might produce values that are not in the image of the original function $f(\mathbf{u})$. For instance, a second-order Taylor approximation of $\mathrm{sigm}(u)$ produces values outside the interval $(0,1)$ and, therefore, the approximated expectation according to (B.17) might not be meaningful.

B.3 Approximating the Logistic Sigmoid by a Gaussian CDF

We can approximate

$$\mathrm{sigm}(u) \approx \Phi(\tilde{c}u) \tag{B.19}$$

by matching the derivatives at $u = 0$. The derivatives are given by

$$\frac{d}{du}\,\mathrm{sigm}(u)|_{u=0} = \mathrm{sigm}(u)\,(1 - \mathrm{sigm}(u))|_{u=0} = \frac{1}{4} \tag{B.20}$$

and

$$\begin{aligned}\frac{d}{du}\Phi(\tilde{c}u)|_{u=0} &= \tilde{c}\left[\frac{d}{du}\Phi(u)\right]_{u=0} = \tilde{c}\left[\frac{d}{du}\int_{-\infty}^{u}\frac{1}{\sqrt{2\pi}}\exp\left(-\frac{z^2}{2}\right)dz\right]_{u=0} \\ &= \frac{\tilde{c}}{\sqrt{2\pi}}\exp\left(-\frac{u^2}{2}\right)\bigg|_{u=0} = \frac{\tilde{c}}{\sqrt{2\pi}}.\end{aligned} \tag{B.21}$$

Equating (B.20) and (B.21) yields $\tilde{c} = \sqrt{\pi/8}$.

B.4 Approximating the Squared Logistic Sigmoid by a Logistic Sigmoid

The squared logistic sigmoid has a sigmoidal shape and, therefore, can be approximated by a logistic sigmoid whose argument is linearly transformed [31] as

$$\mathrm{sigm}(u)^2 \approx \mathrm{sigm}(\tilde{\alpha}(u - \tilde{b})). \tag{B.22}$$

One particular accurate way to perform the approximation is by matching the values and the derivatives of $\mathrm{sigm}(u)^2$ and $\mathrm{sigm}(\tilde{\alpha}(u - \tilde{b}))$ at $u_0 = -\log(\sqrt{2} - 1)$ satisfying $\mathrm{sigm}(u_0)^2 = 1/2$. By matching the values $\mathrm{sigm}(u_0)^2$ and $\mathrm{sigm}(\tilde{\alpha}(u_0 - \tilde{b}))$, we immediately obtain $\tilde{b} = u_0$ since $\mathrm{sigm}(0) = 1/2$.

In the next step, we match the derivatives at u_0. Using $\mathrm{sigm}(u_0) = 1/\sqrt{2}$, we have

$$\frac{d}{du}\,\mathrm{sigm}(u)^2|_{u=u_0} = 2\,\mathrm{sigm}(u)\,\mathrm{sigm}(u)\,(1 - \mathrm{sigm}(u))|_{u=u_0} = 1 - \frac{1}{\sqrt{2}}. \tag{B.23}$$

The derivative of $\mathrm{sigm}(\tilde{\alpha}(u - \tilde{b}))$ for $\tilde{b} = u_0$ is given by

$$\frac{d}{du}\,\mathrm{sigm}(\tilde{\alpha}(u - u_0))|_{u=u_0} = \tilde{\alpha}\,\mathrm{sigm}(\tilde{\alpha}(u - u_0))\,(1 - \mathrm{sigm}(\tilde{\alpha}(u - u_0)))|_{u=u_0} = \frac{\tilde{\alpha}}{4}. \tag{B.24}$$

Equating (B.23) and (B.24) yields $\tilde{\alpha} = 4 - 2\sqrt{2}$.

B.5 Convolving the Logistic Sigmoid with a Gaussian

The integral

$$\mathbb{E}_{u\sim\mathcal{N}(\mu,\sigma)}[\mathrm{sigm}(u)] = \int_{-\infty}^{\infty}\mathrm{sigm}(u)\mathcal{N}(u\,|\,\mu,\sigma^2)du \tag{B.25}$$

does not admit an analytic solution. However, by replacing $\mathrm{sigm}(u)$ with the functionally similar Gaussian cdf $\Phi(u)$, the integral has the analytic solution

$$\int_{-\infty}^{\infty}\Phi(\tilde{c}u)\mathcal{N}(u\,|\,\mu,\sigma^2)du = \Phi\left(\frac{\mu}{\sqrt{\tilde{c}^{-2} + \sigma^2}}\right). \tag{B.26}$$

By applying approximation (B.19) on both sides of (B.26), we obtain the approximation

$$\int_{-\infty}^{\infty} \mathrm{sigm}(u)\mathcal{N}(u\,|\,\mu,\sigma^2)du \approx \mathrm{sigm}\left(\frac{\mu}{\sqrt{\tilde{c}^{-2}+\sigma^2}}\cdot\frac{1}{\tilde{c}}\right) = \mathrm{sigm}\left(\frac{\mu}{\sqrt{1+\tilde{c}^2\sigma^2}}\right) \tag{B.27}$$

for $\tilde{c} = \sqrt{\pi/8}$. This approximation has been used, for instance, in [31, 242].

B.6 Convolving the Squared Logistic Sigmoid with a Gaussian

For the probabilistic forward pass, we require the computation of the raw second moment

$$\mathbb{E}_{u\sim\mathcal{N}(\mu,\sigma^2)}[\mathrm{sigm}(u)^2] = \int_{-\infty}^{\infty} \mathrm{sigm}(u)^2\mathcal{N}(u\,|\,\mu,\sigma^2)du. \tag{B.28}$$

Again, the resulting integral does not admit an analytic solution, but we can apply approximation (B.22) to obtain

$$\begin{aligned}
\mathbb{E}_{u\sim\mathcal{N}(\mu,\sigma^2)}[\mathrm{sigm}(u)^2] &\approx \mathbb{E}_{u\sim\mathcal{N}(\mu,\sigma^2)}[\mathrm{sigm}(\tilde{\alpha}(u-\tilde{b}))] &\text{(B.29)}\\
&= \mathbb{E}_{z\sim\mathcal{N}(0,1)}[\mathrm{sigm}(\tilde{\alpha}((z\sigma+\mu)-\tilde{b}))] &\text{(B.30)}\\
&= \mathbb{E}_{z\sim\mathcal{N}(0,1)}[\mathrm{sigm}(\tilde{\alpha}\sigma z+\tilde{\alpha}(\mu-\tilde{b}))] &\text{(B.31)}\\
&= \mathbb{E}_{\tilde{u}\sim\mathcal{N}(\tilde{\alpha}(\mu-\tilde{b}),\tilde{\alpha}^2\sigma^2)}[\mathrm{sigm}(\tilde{u})]. &\text{(B.32)}
\end{aligned}$$

Using (B.27), we obtain

$$\mathbb{E}_{u\sim\mathcal{N}(\mu,\sigma^2)}[\mathrm{sigm}(u)^2] \approx \mathrm{sigm}\left(\frac{\tilde{\alpha}(\mu-\tilde{b})}{\sqrt{1+\tilde{\alpha}^2\tilde{c}^2\sigma^2}}\right) \tag{B.33}$$

for $\tilde{\alpha} = 4-2\sqrt{2}$, $\tilde{b} = -\log(\sqrt{2}-1)$, and $\tilde{c} = \sqrt{\pi/8}$. This approximation has been used in [31].

B.7 Convolving the (Squared) Hyperbolic Tangent with a Gaussian

The integral

$$\mathbb{E}_{u\sim\mathcal{N}(\mu,\sigma^2)}[\tanh(u)] = \int_{-\infty}^{\infty} \tanh(u)\mathcal{N}(u|\mu,\sigma^2)du \tag{B.34}$$

does not admit an analytic solution, but we can resort to approximation (B.27) derived for the logistic sigmoid by noting that $\tanh(u) = 2\,\mathrm{sigm}(2u)-1$. This yields

$$\begin{aligned}
\mathbb{E}_{u\sim\mathcal{N}(\mu,\sigma^2)}[\tanh(u)] &= 2\mathbb{E}_{u\sim\mathcal{N}(\mu,\sigma^2)}[\mathrm{sigm}(2u)]-1 &\text{(B.35)}\\
&= 2\mathbb{E}_{\tilde{u}\sim\mathcal{N}(2\mu,4\sigma^2)}[\mathrm{sigm}(\tilde{u})]-1 &\text{(B.36)}\\
&\approx 2\,\mathrm{sigm}\left(\frac{2\mu}{\sqrt{1+4\tilde{c}^2\sigma^2}}\right)-1 = \tanh\left(\frac{\mu}{\sqrt{1+4\tilde{c}^2\sigma^2}}\right). &\text{(B.37)}
\end{aligned}$$

The squared hyperbolic tangent $\tanh(u)^2$ is not of a sigmoidal shape as it was the case for $\mathrm{sigm}(u)^2$ and can thus not be modelled using a hyperbolic tangent itself. We can still resort to results obtained for the logistic sigmoid by noting that $\tanh(u)^2 = 4\,\mathrm{sigm}(2u)^2 - 4\,\mathrm{sigm}(2u)+1$.

Using (B.33) and (B.27), we obtain

$$\mathbb{E}_{u\sim\mathcal{N}(\mu,\sigma^2)}[\tanh(u)^2] = \mathbb{E}_{u\sim\mathcal{N}(\mu,\sigma^2)}[4\,\mathrm{sigm}(2u)^2 - 4\,\mathrm{sigm}(2u) + 1] \tag{B.38}$$

$$= \mathbb{E}_{u\sim\mathcal{N}(2\mu,4\sigma^2)}[4\,\mathrm{sigm}(u)^2 - 4\,\mathrm{sigm}(u) + 1] \tag{B.39}$$

$$= 4\,\mathrm{sigm}\left(\frac{\tilde{\alpha}(2\mu - \tilde{b})}{\sqrt{1 + 4\tilde{\alpha}^2\tilde{c}^2\sigma^2}}\right) - 4\,\mathrm{sigm}\left(\frac{2\mu}{\sqrt{1 + 4\tilde{c}^2\sigma^2}}\right) + 1. \tag{B.40}$$

B.8 Sampling from a Binary Gumbel-Softmax Distribution

We consider the Gumbel-softmax approximation for a binary distribution over the values $\{-1, 1\}$ defined by the probabilities $p_{+1} = p(Z = 1)$ and $p_{-1} = p(Z = -1)$. Expressions for other binary values (e.g., $\{0, 1\}$) are obtained similarly. We are given a temperature τ_{g} and two samples $\varepsilon_1, \varepsilon_2 \sim \mathrm{Gumbel}(0, 1)$. For convenience, we define

$$\rho_{+1} = \frac{\log(p_{+1}) + \varepsilon_1}{\tau_{\mathrm{g}}} \qquad \text{and} \qquad \rho_{-1} = \frac{\log(p_{-1}) + \varepsilon_2}{\tau_{\mathrm{g}}}. \tag{B.41}$$

A Gumbel-softmax sample $\mathbf{z} = (z_{+1}, z_{-1})$ is then obtained by

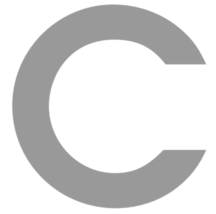
$$z_{+1} = \frac{\exp(\rho_{+1})}{\exp(\rho_{+1}) + \exp(\rho_{-1})} \qquad \text{and} \qquad z_{-1} = \frac{\exp(\rho_{-1})}{\exp(\rho_{+1}) + \exp(\rho_{-1})}. \tag{B.42}$$

A scalar-valued Gumbel-softmax sample is obtained by $v = z_{+1} - z_{-1}$. This yields

$$\frac{\exp(\rho_{+1}) - \exp(\rho_{-1})}{\exp(\rho_{+1}) + \exp(\rho_{-1})} = \frac{\exp(\frac{\rho_{+1}+\rho_{-1}}{2})\left(\exp(\frac{\rho_{+1}-\rho_{-1}}{2}) - \exp(\frac{\rho_{-1}-\rho_{+1}}{2})\right)}{\exp(\frac{\rho_{+1}+\rho_{-1}}{2})\left(\exp(\frac{\rho_{+1}-\rho_{-1}}{2}) + \exp(\frac{\rho_{-1}-\rho_{+1}}{2})\right)} \tag{B.43}$$

$$= \tanh\left(\frac{\rho_{+1} - \rho_{-1}}{2}\right) \tag{B.44}$$

$$= \tanh\left(\frac{\log(p_{+1}) + \varepsilon_1 - \log(p_{-1}) - \varepsilon_2}{2\tau_{\mathrm{g}}}\right). \tag{B.45}$$



List of Acronyms

AHMC	adaptive Hamiltonian Monte Carlo
BN	Bayesian network
BNC	Bayesian network classifier
BNN	Bayesian neural network
cdf	cumulative distribution function
CNN	convolutional neural network
CPT	conditional probability table
CPU	central processing unit
CRP	Chinese restaurant process
DNN	deep neural network
DP	Dirichlet process
DPP	determinantal point process
GMM	Gaussian mixture model
GPU	graphics processing unit
HMC	Hamiltonian Monte Carlo
KL	Kullback-Leibler
LM	large margin
LMC	Langevin Monte Carlo
MAP	maximum a posteriori
MCMC	Markov chain Monte Carlo
MDL	minimum description length
ML	maximum likelihood
MS	model size
MSE	mean squared error
NAS	neural architecture search
NB	naïve Bayes
NLL	negative log-likelihood
PCA	principal component analysis
pdf	probability density function
pmf	probability mass function
RMSE	root mean squared error
RNN	recurrent neural network
SGD	stochastic gradient descent
SGHMC	stochastic gradient Hamiltonian Monte Carlo
SGLD	stochastic gradient Langevin dynamics
SL	structure learning

STE	straight-through gradient estimator
TAN	tree-augmented naïve Bayes
TPU	tensor processing unit

Bibliography

[1] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. P. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[2] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. P. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[3] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[4] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2020, pp. 1877–1901.

[5] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[6] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems (NIPS)*, 2012, pp. 1106–1114.

[7] U. von Luxburg and B. Schölkopf, "Statistical learning theory: Models, concepts, and results," in *Inductive Logic*, ser. Handbook of the History of Logic, vol. 10, 2011, pp. 651–706.

[8] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, no. 5, pp. 359–366, 1989.

[9] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research (JMLR)*, vol. 15, no. 56, pp. 1929–1958, 2014.

[10] L. Bottou, F. E. Curtis, and J. Nocedal, "Optimization methods for large-scale machine learning," *SIAM Review*, vol. 60, no. 2, pp. 223–311, 2018.

[11] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *International Conference on Learning Representations (ICLR)*, 2015.

[12] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, "Automatic differentiation in machine learning: A survey," *Journal of Machine Learning Research (JMLR)*, vol. 18, no. 153, pp. 1–43, 2017.

[13] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[14] Y. Bengio, N. Léonard, and A. C. Courville, "Estimating or propagating gradients through stochastic neurons for conditional computation," 2013. arXiv: 1308.3432.

[15] R. Salakhutdinov and G. Hinton, "Semantic hashing," *International Journal of Approximate Reasoning*, vol. 50, no. 7, pp. 969–978, 2009.

[16] A. Shekhovtsov, V. Yanush, and B. Flach, "Path sample-analytic gradient estimators for stochastic binary networks," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2020, pp. 12 884–12 894.

[17] A. M. Saxe, J. L. McClelland, and S. Ganguli, "Exact solutions to the nonlinear dynamics of learning in deep linear neural networks," in *International Conference on Learning Representations (ICLR)*, 2014.

[18] D. Mishkin and J. Matas, "All you need is a good init," in *International Conference on Learning Representations (ICLR)*, 2016.

[19] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *International Conference on Artificial Intelligence and Statistics (AISTATS)*, vol. 9, 2010, pp. 249–256.

[20] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification," in *IEEE International Conference on Computer Vision (ICCV)*, 2015, pp. 1026–1034.

[21] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and F.-F. Li, "ImageNet: A large-scale hierarchical image database," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2009, pp. 248–255.

[22] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science*, vol. 313, no. 5786, pp. 504–507, 2006.

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

[24] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 2261–2269.

[25] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 1–9.

[26] S. Sabour, N. Frosst, and G. E. Hinton, "Dynamic routing between capsules," in *Advances in Neural Information Processing Systems (NIPS)*, 2017, pp. 3856–3866.

[27] G. Desjardins, K. Simonyan, R. Pascanu, and K. Kavukcuoglu, "Natural neural networks," in *Advances in Neural Information Processing Systems (NIPS)*, 2015, pp. 2071–2079.

[28] L. Huang, D. Yang, B. Lang, and J. Deng, "Decorrelated batch normalization," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 791–800.

[29] P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol, "Extracting and composing robust features with denoising autoencoders," in *International Conference on Machine Learning (ICML)*, 2008, pp. 1096–1103.

[30] Y. Gal and Z. Ghahramani, "Dropout as a Bayesian approximation: Representing model uncertainty in deep learning," in *International Conference on Machine Learning (ICML)*, vol. 48, 2016, pp. 1050–1059.

[31] S. I. Wang and C. D. Manning, "Fast dropout training," in *International Conference on Machine Learning (ICML)*, vol. 28, 2013, pp. 118–126.

[32] S. Wager, S. I. Wang, and P. Liang, "Dropout training as adaptive regularization," in *Advances in Neural Information Processing Systems (NIPS)*, 2013, pp. 351–359.

[33] D. P. Kingma, T. Salimans, and M. Welling, "Variational dropout and the local reparameterization trick," in *Advances in Neural Information Processing Systems (NIPS)*, 2015, pp. 2575–2583.

[34] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International Conference on Machine Learning (ICML)*, vol. 37, 2015, pp. 448–456.

[35] L. Wan, M. D. Zeiler, S. Zhang, Y. LeCun, and R. Fergus, "Regularization of neural networks using DropConnect," in *International Conference on Machine Learning (ICML)*, vol. 28, 2013, pp. 1058–1066.

[36] G. Ghiasi, T.-Y. Lin, and Q. V. Le, "DropBlock: A regularization method for convolutional networks," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2018, pp. 10 750–10 760.

[37] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representations (ICLR)*, 2015.

[38] M. Lin, Q. Chen, and S. Yan, "Network in network," in *International Conference on Learning Representations (ICLR)*, 2014.

[39] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the Inception architecture for computer vision," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 2818–2826.

[40] K. He, X. Zhang, S. Ren, and J. Sun, "Identity mappings in deep residual networks," in *European Conference on Computer Vision (ECCV)*, 2016, pp. 630–645.

[41] S. Xie, R. B. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 5987–5995.

[42] M. Tan and Q. V. Le, "EfficientNet: Rethinking model scaling for convolutional neural networks," in *International Conference on Machine Learning (ICML)*, vol. 97, 2019, pp. 6105–6114.

[43] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," in *International Conference on Learning Representations (ICLR)*, 2017.

[44] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le, "MnasNet: Platform-aware neural architecture search for mobile," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 2820–2828.

[45] M. Sandler, A. G. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted residuals and linear bottlenecks," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 4510–4520.

[46] D. Hernandez and T. B. Brown, "Measuring the algorithmic efficiency of neural networks," 2020. arXiv: 2005.04305.

[47] M. J. Wainwright and M. I. Jordan, "Graphical models, exponential families, and variational inference," *Foundations and Trends® in Machine Learning*, vol. 1, no. 1–2, pp. 1–305, 2008.

[48] D. Koller and N. Friedman, *Probabilistic Graphical Models: Principles and Techniques*, ser. Adaptive Computation and Machine Learning. MIT Press, 2009.

[49] J. Nocedal and S. Wright, *Numerical Optimization*, 2nd ed. Springer New York, 2006.

[50] T. Minka, "Divergence measures and message passing," Tech. Rep. MSR-TR-2005-173, 2005.

[51] Y. Li and R. E. Turner, "Rényi divergence variational inference," in *Advances in Neural Information Processing Systems (NIPS)*, 2016, pp. 1073–1081.

[52] D. Wang, H. Liu, and Q. Liu, "Variational inference with tail-adaptive f-divergence," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2018, pp. 5742–5752.

[53] K. P. Murphy, *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.

[54] T. P. Minka, "Expectation propagation for approximate Bayesian inference," in *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2001, pp. 362–369.

[55] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2006.

[56] Z. Ghahramani and M. J. Beal, "Propagation algorithms for variational Bayesian learning," in *Advances in Neural Information Processing Systems (NIPS)*, 2000, pp. 507–513.

[57] M. D. Hoffman, D. M. Blei, C. Wang, and J. Paisley, "Stochastic variational inference," *Journal of Machine Learning Research (JMLR)*, vol. 14, no. 4, pp. 1303–1347, 2013.

[58] R. Ranganath, S. Gerrish, and D. M. Blei, "Black box variational inference," in *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2014, pp. 814–822.

[59] C. Zhang, J. Bütepage, H. Kjellström, and S. Mandt, "Advances in variational inference," *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, vol. 41, no. 8, pp. 2008–2026, 2019.

[60] D. J. Rezende and S. Mohamed, "Variational inference with normalizing flows," in *International Conference on Machine Learning (ICML)*, 2015, pp. 1530–1538.

[61] G. Papamakarios, E. T. Nalisnick, D. J. Rezende, S. Mohamed, and B. Lakshminarayanan, "Normalizing flows for probabilistic modeling and inference," 2019. arXiv: 1912.02762.

[62] D. P. Kingma and M. Welling, "Auto-encoding variational Bayes," in *International Conference on Learning Representations (ICLR)*, 2014.

[63] R. M. Neal, "Probabilistic inference using Markov chain Monte Carlo methods," Department of Computer Science, University of Toronto, Tech. Rep. CRG-TR-93-1, 1993.

[64] W. K. Hastings, "Monte Carlo sampling methods using Markov chains and their applications," *Biometrika*, vol. 57, no. 1, pp. 97–109, 1970.

[65] R. M. Neal, "Slice sampling," *The Annals of Statistics*, vol. 31, no. 3, pp. 705–741, 2003.

[66] G. E. Hinton, "Training products of experts by minimizing contrastive divergence," *Neural Computation*, vol. 14, no. 8, pp. 1771–1800, 2002.

[67] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent Dirichlet allocation," *Journal of Machine Learning Research (JMLR)*, vol. 3, pp. 993–1022, 2003.

[68] T. L. Griffiths and M. Steyvers, "Finding scientific topics," *Proceedings of the National Academy of Sciences (PNAS)*, vol. 101, no. suppl 1, pp. 5228–5235, 2004.

[69] S. Duane, A. Kennedy, B. J. Pendleton, and D. Roweth, "Hybrid Monte Carlo," *Physics Letters B*, vol. 195, no. 2, pp. 216–222, 1987.

[70] R. M. Neal, "Bayesian training of backpropagation networks by the hybrid Monte Carlo method," Department of Computer Science, University of Toronto, Tech. Rep. CRG-TR-92-1, 1992.

[71] R. H. Swendsen and J.-S. Wang, "Nonuniversal critical dynamics in Monte Carlo simulations," *Physical Review Letters*, vol. 58, pp. 86–88, 2 1987.

[72] Z. Wang, S. Mohamed, and N. de Freitas, "Adaptive Hamiltonian and Riemann manifold Monte Carlo," in *International Conference on Machine Learning (ICML)*, vol. 28, 2013, pp. 1462–1470.

[73] M. D. Hoffman and A. Gelman, “The No-U-Turn sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo,” *Journal of Machine Learning Research (JMLR)*, vol. 15, no. 47, pp. 1593–1623, 2014.

[74] C. Blundell, J. Cornebise, K. Kavukcuoglu, and D. Wierstra, “Weight uncertainty in neural networks,” in *International Conference on Machine Learning (ICML)*, 2015, pp. 1613–1622.

[75] S. J. Nowlan and G. E. Hinton, “Simplifying neural networks by soft weight-sharing,” *Neural Computation*, vol. 4, no. 4, pp. 473–493, 1992.

[76] W. Chen, J. T. Wilson, S. Tyree, K. Q. Weinberger, and Y. Chen, “Compressing neural networks with the hashing trick,” in *International Conference on Machine Learning (ICML)*, 2015, pp. 2285–2294.

[77] A. Y. K. Foong, Y. Li, J. M. Hernández-Lobato, and R. E. Turner, “’In-between’ uncertainty in Bayesian neural networks,” in *Workshop on Uncertainty & Robustness in Deep Learning @ ICML*, 2019.

[78] M. N. Gibbs, “Bayesian Gaussian processes for regression and classification,” PhD thesis, University of Cambridge, 1997.

[79] A. Shekhovtsov and B. Flach, “Feed-forward propagation in probabilistic neural networks with categorical and max layers,” in *International Conference on Learning Representations (ICLR)*, 2019.

[80] W. Roth and F. Pernkopf, “Variational inference in neural networks using an approximate closed-form objective,” in *Workshop on Bayesian Deep Learning @ NIPS*, 2016.

[81] Y. W. Teh, D. Newman, and M. Welling, “A collapsed variational Bayesian inference algorithm for latent Dirichlet allocation,” in *Advances in Neural Information Processing Systems (NIPS)*, 2006, pp. 1353–1360.

[82] F. Ribeiro and M. Opper, “Expectation propagation with factorizing distributions: A Gaussian approximation and performance results for simple models,” *Neural Computation*, vol. 23, no. 4, pp. 1047–1069, 2011.

[83] D. Soudry, I. Hubara, and R. Meir, “Expectation backpropagation: Parameter-free training of multilayer neural networks with continuous or discrete weights,” in *Advances in Neural Information Processing Systems (NIPS)*, 2014, pp. 963–971.

[84] J. M. Hernández-Lobato and R. P. Adams, “Probabilistic backpropagation for scalable learning of Bayesian neural networks,” in *International Conference on Machine Learning (ICML)*, 2015, pp. 1861–1869.

[85] A. Wu, S. Nowozin, E. Meeds, R. E. Turner, J. M. Hernández-Lobato, and A. L. Gaunt, “Deterministic variational inference for robust Bayesian neural networks,” in *International Conference on Learning Representations (ICLR)*, 2019.

[86] G. E. Hinton and D. van Camp, “Keeping the neural networks simple by minimizing the description length of the weights,” in *ACM Conference on Computational Learning Theory (COLT)*, 1993, pp. 5–13.

[87] P. D. Grünwald, *The minimum description length principle*. MIT press, 2007.

[88] M. van Baalen, C. Louizos, M. Nagel, R. A. Amjad, Y. Wang, T. Blankevoort, and M. Welling, “Bayesian bits: Unifying quantization and pruning,” in *Advances in Neural Information Processing Systems (NeurIPS)*, 2020, pp. 5741–5752.

[89] A. Graves, “Practical variational inference for neural networks,” in *Advances in Neural Information Processing Systems (NIPS)*, 2011, pp. 2348–2356.

[90] M. Opper and C. Archambeau, “The variational Gaussian approximation revisited,” *Neural Computation*, vol. 21, no. 3, pp. 786–792, 2009.

[91] S. Mohamed, M. Rosca, M. Figurnov, and A. Mnih, "Monte Carlo gradient estimation in machine learning," *Journal of Machine Learning Research (JMLR)*, vol. 21, no. 132, pp. 1–62, 2020.

[92] J. P. Kleijnen and R. Y. Rubinstein, "Optimization and sensitivity analysis of computer simulation models by the score function method," *European Journal of Operational Research*, vol. 88, no. 3, pp. 413–427, 1996.

[93] P. W. Glynn, "Likelihood ratio gradient estimation for stochastic systems," *Communications of the ACM*, vol. 33, no. 10, pp. 75–84, 1990.

[94] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine Learning*, vol. 8, pp. 229–256, 1992.

[95] J. W. Paisley, D. M. Blei, and M. I. Jordan, "Variational Bayesian inference with stochastic search," in *International Conference on Machine Learning (ICML)*, 2012.

[96] M. K. Titsias and M. Lázaro-Gredilla, "Local expectation gradients for black box variational inference," in *Advances in Neural Information Processing Systems (NIPS)*, 2015, pp. 2638–2646.

[97] A. B. Owen, *Monte Carlo theory, methods and examples.* 2013.

[98] D. J. Rezende, S. Mohamed, and D. Wierstra, "Stochastic backpropagation and approximate inference in deep generative models," in *International Conference on Machine Learning (ICML)*, 2014, pp. 1278–1286.

[99] M. K. Titsias and M. Lázaro-Gredilla, "Doubly stochastic variational Bayes for non-conjugate inference," in *International Conference on Machine Learning (ICML)*, 2014, pp. 1971–1979.

[100] E. Jang, S. Gu, and B. Poole, "Categorical reparameterization with Gumbel-softmax," in *International Conference on Learning Representations (ICLR)*, 2017.

[101] C. J. Maddison, A. Mnih, and Y. W. Teh, "The Concrete distribution: A continuous relaxation of discrete random variables," in *International Conference on Learning Representations (ICLR)*, 2017.

[102] M. Welling and Y. W. Teh, "Bayesian learning via stochastic gradient Langevin dynamics," in *International Conference on Machine Learning (ICML)*, 2011, pp. 681–688.

[103] T. Chen, E. B. Fox, and C. Guestrin, "Stochastic gradient Hamiltonian Monte Carlo," in *International Conference on Machine Learning (ICML)*, 2014, pp. 1683–1691.

[104] R. M. Neal, "MCMC using Hamiltonian dynamics," in *Handbook of Markov Chain Monte Carlo*. CRC Press, 2011, ch. 5.

[105] R. Bardenet, A. Doucet, and C. C. Holmes, "Towards scaling up Markov chain Monte Carlo: an adaptive subsampling approach," in *International Conference on Machine Learning (ICML)*, vol. 32, 2014, pp. 405–413.

[106] A. Korattikara, Y. Chen, and M. Welling, "Austerity in MCMC land: Cutting the Metropolis-Hastings budget," in *International Conference on Machine Learning (ICML)*, vol. 32, 2014, pp. 181–189.

[107] W. Roth, G. Schindler, M. Zöhrer, L. Pfeifenberger, R. Peharz, S. Tschiatschek, H. Fröning, F. Pernkopf, and Z. Ghahramani, "Resource-efficient neural networks for embedded systems," 2020. arXiv: 2001.03048.

[108] M. Höhfeld and S. E. Fahlman, "Learning with limited numerical precision using the cascade-correlation algorithm," *IEEE Transactions on Neural Networks*, vol. 3, no. 4, pp. 602–611, 1992.

[109] ——, “Probabilistic rounding in neural network learning with limited precision,” *Neurocomputing*, vol. 4, no. 6, pp. 291–299, 1992.

[110] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, “Deep learning with limited numerical precision,” in *International Conference on Machine Learning (ICML)*, 2015, pp. 1737–1746.

[111] Z. Lin, M. Courbariaux, R. Memisevic, and Y. Bengio, “Neural networks with few multiplications,” in *International Conference on Learning Representations (ICLR)*, 2016.

[112] M. Courbariaux, Y. Bengio, and J.-P. David, “Training deep neural networks with low precision multiplications,” in *Workshop @ International Conference on Learning Representations (ICLR)*, 2015.

[113] D. D. Lin, S. S. Talathi, and V. S. Annapureddy, “Fixed point quantization of deep convolutional networks,” in *International Conference on Machine Learning (ICML)*, 2016, pp. 2849–2858.

[114] M. Courbariaux, Y. Bengio, and J.-P. David, “BinaryConnect: Training deep neural networks with binary weights during propagations,” in *Advances in Neural Information Processing Systems (NIPS)*, 2015, pp. 3123–3131.

[115] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, “Binarized neural networks,” in *Advances in Neural Information Processing Systems (NIPS)*, 2016, pp. 4107–4115.

[116] F. Li, B. Zhang, and B. Liu, “Ternary weight networks,” 2016. arXiv: 1605.04711.

[117] C. Zhu, S. Han, H. Mao, and W. J. Dally, “Trained ternary quantization,” in *International Conference on Learning Representations (ICLR)*, 2017.

[118] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, “XNOR-Net: ImageNet classification using binary convolutional neural networks,” in *European Conference on Computer Vision (ECCV)*, 2016, pp. 525–542.

[119] X. Lin, C. Zhao, and W. Pan, “Towards accurate binary convolutional neural network,” in *Advances in Neural Information Processing Systems (NIPS)*, 2017, pp. 345–353.

[120] Z. Cai, X. He, J. Sun, and N. Vasconcelos, “Deep learning with low precision by half-wave Gaussian quantization,” in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 5406–5414.

[121] D. Miyashita, E. H. Lee, and B. Murmann, “Convolutional neural networks using logarithmic data representation,” 2016. arXiv: 1603.01025.

[122] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen, “Incremental network quantization: Towards lossless CNNs with low-precision weights,” in *International Conference on Learning Representations (ICLR)*, 2017.

[123] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. G. Howard, H. Adam, and D. Kalenichenko, “Quantization and training of neural networks for efficient integer-arithmetic-only inference,” in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 2704–2713.

[124] Z. Liu, B. Wu, W. Luo, X. Yang, W. Liu, and K.-T. Cheng, “Bi-Real net: Enhancing the performance of 1-bit CNNs with improved representational capability and advanced training algorithm,” in *European Conference on Computer Vision (ECCV)*, 2018, pp. 747–763.

[125] D. Zhang, J. Yang, D. Ye, and G. Hua, “LQ-Nets: Learned quantization for highly accurate and compact deep neural networks,” in *European Conference on Computer Vision (ECCV)*, 2018, pp. 373–390.

[126] C. Louizos, M. Reisser, T. Blankevoort, E. Gavves, and M. Welling, "Relaxed quantization for discretized neural networks," in *International Conference on Learning Representations (ICLR)*, 2019.

[127] Z. Dong, Z. Yao, A. Gholami, M. W. Mahoney, and K. Keutzer, "HAWQ: Hessian aware quantization of neural networks with mixed-precision," in *IEEE International Conference on Computer Vision (ICCV)*, 2019, pp. 293–302.

[128] S. Uhlich, L. Mauch, F. Cardinaux, K. Yoshiyama, J. A. Garcıa, S. Tiedemann, T. Kemp, and A. Nakamura, "Mixed precision DNNs: All you need is a good parametrization," in *International Conference on Learning Representations (ICLR)*, 2020.

[129] S. K. Esser, J. L. McKinstry, D. Bablani, R. Appuswamy, and D. S. Modha, "Learned step size quantization," in *International Conference on Learning Representations (ICLR)*, 2020.

[130] S. Zhou, Z. Ni, X. Zhou, H. Wen, Y. Wu, and Y. Zou, "DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients," 2016. arXiv: `1606.06160`.

[131] S. Wu, G. Li, F. Chen, and L. Shi, "Training and inference with integers in deep neural networks," in *International Conference on Learning Representations (ICLR)*, 2018.

[132] H. Li, S. De, Z. Xu, C. Studer, H. Samet, and T. Goldstein, "Training quantized nets: A deeper understanding," in *Advances in Neural Information Processing Systems (NIPS)*, 2017, pp. 5811–5821.

[133] A. G. Anderson and C. P. Berg, "The high-dimensional geometry of binary neural networks," in *International Conference on Learning Representations (ICLR)*, 2018.

[134] W. Roth, G. Schindler, H. Fröning, and F. Pernkopf, "On resource-efficient Bayesian network classifiers and deep neural networks," in *International Conference on Pattern Recognition (ICPR)*, 2020, pp. 10 297–10 304.

[135] J. Achterhold, J. M. Köhler, A. Schmeink, and T. Genewein, "Variational network quantization," in *International Conference on Learning Representations (ICLR)*, 2018.

[136] C. Louizos, K. Ullrich, and M. Welling, "Bayesian compression for deep learning," in *Advances in Neural Information Processing Systems (NIPS)*, 2017, pp. 3288–3298.

[137] O. Shayer, D. Levi, and E. Fetaya, "Learning discrete weights using the local reparameterization trick," in *International Conference on Learning Representations (ICLR)*, 2018.

[138] J. W. T. Peters and M. Welling, "Probabilistic binary neural networks," 2018. arXiv: `1809.03368`.

[139] W. Roth, G. Schindler, H. Fröning, and F. Pernkopf, "Training discrete-valued neural networks with sign activations using weight distributions," in *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD), Part II*, 2019, pp. 382–398.

[140] M. Havasi, R. Peharz, and J. M. Hernández-Lobato, "Minimal random code learning: Getting bits back from compressed model parameters," in *International Conference on Learning Representations (ICLR)*, 2019.

[141] Y. LeCun, J. S. Denker, and S. A. Solla, "Optimal brain damage," in *Advances in Neural Information Processing Systems (NIPS)*, 1989, pp. 598–605.

[142] B. Hassibi and D. G. Stork, "Second order derivatives for network pruning: Optimal brain surgeon," in *Advances in Neural Information Processing Systems (NIPS)*, 1992, pp. 164–171.

[143] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," in *Advances in Neural Information Processing Systems (NIPS)*, 2015, pp. 1135–1143.

[144] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and Huffman coding," in *International Conference on Learning Representations (ICLR)*, 2016.

[145] Y. Guo, A. Yao, and Y. Chen, "Dynamic network surgery for efficient DNNs," in *Advances in Neural Information Processing Systems (NIPS)*, 2016, pp. 1379–1387.

[146] Z. Mariet and S. Sra, "Diversity networks: Neural network compression using determinantal point processes," in *International Conference on Learning Representations (ICLR)*, 2016.

[147] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *Advances in Neural Information Processing Systems (NIPS)*, 2016, pp. 2074–2082.

[148] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang, "Learning efficient convolutional networks through network slimming," in *IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 2755–2763.

[149] Z. Huang and N. Wang, "Data-driven sparse structure selection for deep neural networks," in *European Conference on Computer Vision (ECCV)*, 2018, pp. 317–334.

[150] J.-H. Luo, J. Wu, and W. Lin, "ThiNet: A filter level pruning method for deep neural network compression," in *IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 5068–5076.

[151] C. Louizos, M. Welling, and D. P. Kingma, "Learning sparse neural networks through L_0 regularization," in *International Conference on Learning Representations (ICLR)*, 2018.

[152] Y. Li and S. Ji, "L_0-ARM: Network sparsification via stochastic binary optimization," in *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD), Part II*, 2019, pp. 432–448.

[153] M. Yin and M. Zhou, "ARM: Augment-REINFORCE-merge gradient for stochastic binary networks," in *International Conference on Learning Representations (ICLR)*, 2019.

[154] D. Molchanov, A. Ashukha, and D. P. Vetrov, "Variational dropout sparsifies deep neural networks," in *International Conference on Machine Learning (ICML)*, vol. 70, 2017, pp. 2498–2507.

[155] J. Lin, Y. Rao, J. Lu, and J. Zhou, "Runtime neural pruning," in *Advances in Neural Information Processing Systems (NIPS)*, 2017, pp. 2181–2191.

[156] X. Dong, J. Huang, Y. Yang, and S. Yan, "More is less: A more complicated network with less inference complexity," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 1895–1903.

[157] X. Gao, Y. Zhao, L. Dudziak, R. D. Mullins, and C.-Z. Xu, "Dynamic channel pruning: Feature boosting and suppression," in *International Conference on Learning Representations (ICLR)*, 2019.

[158] K. Ullrich, E. Meeds, and M. Welling, "Soft weight-sharing for neural network compression," in *International Conference on Learning Representations (ICLR)*, 2017.

[159] W. Roth and F. Pernkopf, "Bayesian neural networks with weight sharing using Dirichlet processes," *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, vol. 42, no. 1, pp. 246–252, 2020.

[160] X. Zeng and T. R. Martinez, "Using a neural network to approximate an ensemble of classifiers," *Neural Processing Letters*, vol. 12, no. 3, pp. 225–237, 2000.

[161] C. Bucila, R. Caruana, and A. Niculescu-Mizil, "Model compression," in *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2006, pp. 535–541.

[162] J. Ba and R. Caruana, “Do deep nets really need to be deep?” In *Advances in Neural Information Processing Systems (NIPS)*, 2014, pp. 2654–2662.

[163] J. Li, R. Zhao, J.-T. Huang, and Y. Gong, “Learning small-size DNN with output-distribution-based criteria,” in *INTERSPEECH: Conference of the International Speech Communication Association*, 2014, pp. 1910–1914.

[164] G. Hinton, O. Vinyals, and J. Dean, “Distilling the knowledge in a neural network,” in *Deep Learning and Representation Learning Workshop @ NIPS*, 2015.

[165] A. Romero, N. Ballas, S. E. Kahou, A. Chassang, C. Gatta, and Y. Bengio, “FitNets: Hints for thin deep nets,” in *International Conference on Learning Representations (ICLR)*, 2015.

[166] J. Kim, S. Park, and N. Kwak, “Paraphrasing complex network: Network compression via factor transfer,” in *Advances in Neural Information Processing Systems (NeurIPS)*, 2018, pp. 2765–2774.

[167] A. K. Mishra and D. Marr, “Apprentice: Using knowledge distillation techniques to improve low-precision network accuracy,” in *International Conference on Learning Representations (ICLR)*, 2018.

[168] A. Polino, R. Pascanu, and D. Alistarh, “Model compression via distillation and quantization,” in *International Conference on Learning Representations (ICLR)*, 2018.

[169] M. Phuong and C. Lampert, “Distillation-based training for multi-exit architectures,” in *IEEE International Conference on Computer Vision (ICCV)*, 2019, pp. 1355–1364.

[170] A. Korattikara, V. Rathod, K. P. Murphy, and M. Welling, “Bayesian dark knowledge,” in *Advances in Neural Information Processing Systems (NIPS)*, 2015, pp. 3438–3446.

[171] M. Denil, B. Shakibi, L. Dinh, M. Ranzato, and N. de Freitas, “Predicting parameters in deep learning,” in *Advances in Neural Information Processing Systems (NIPS)*, 2013, pp. 2148–2156.

[172] A. Novikov, D. Podoprikhin, A. Osokin, and D. P. Vetrov, “Tensorizing neural networks,” in *Advances in Neural Information Processing Systems (NIPS)*, 2015, pp. 442–450.

[173] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus, “Exploiting linear structure within convolutional networks for efficient evaluation,” in *Advances in Neural Information Processing Systems (NIPS)*, 2014, pp. 1269–1277.

[174] M. Jaderberg, A. Vedaldi, and A. Zisserman, “Speeding up convolutional neural networks with low rank expansions,” in *British Machine Vision Conference (BMVC)*, 2014.

[175] V. Lebedev, Y. Ganin, M. Rakhuba, I. V. Oseledets, and V. S. Lempitsky, “Speeding-up convolutional neural networks using fine-tuned CP-decomposition,” in *International Conference on Learning Representations (ICLR)*, 2015.

[176] Y. Cheng, F. X. Yu, R. S. Feris, S. Kumar, A. N. Choudhary, and S.-F. Chang, “An exploration of parameter redundancy in deep networks with circulant projections,” in *IEEE International Conference on Computer Vision (ICCV)*, 2015, pp. 2857–2865.

[177] Z. Yang, M. Moczulski, M. Denil, N. de Freitas, A. J. Smola, L. Song, and Z. Wang, “Deep fried convnets,” in *IEEE International Conference on Computer Vision (ICCV)*, 2015, pp. 1476–1483.

[178] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer, “SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size,” 2016. arXiv: 1602.07360.

[179] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, “MobileNets: Efficient convolutional neural networks for mobile vision applications,” 2017. arXiv: 1704.04861.

[180] X. Zhang, X. Zhou, M. Lin, and J. Sun, "ShuffleNet: An extremely efficient convolutional neural network for mobile devices," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 6848–6856.

[181] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 8697–8710.

[182] T.-Y. Lin, M. Maire, S. J. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: Common objects in context," in *European Conference on Computer Vision (ECCV)*, 2014, pp. 740–755.

[183] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "HAQ: Hardware-aware automated quantization with mixed precision," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 8612–8620.

[184] H. Cai, L. Zhu, and S. Han, "ProxylessNAS: Direct neural architecture search on target task and hardware," in *International Conference on Learning Representations (ICLR)*, 2019.

[185] D. Stamoulis, R. Ding, D. Wang, D. Lymberopoulos, B. Priyantha, J. Liu, and D. Marculescu, "Single-path NAS: Designing hardware-efficient convnets in less than 4 hours," in *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD), Part II*, 2019, pp. 481–497.

[186] B. Wu, Y. Wang, P. Zhang, Y. Tian, P. Vajda, and K. Keutzer, "Mixed precision quantization of convnets via differentiable neural architecture search," 2018. arXiv: `1812.00090`.

[187] H. Liu, K. Simonyan, and Y. Yang, "DARTS: Differentiable architecture search," in *International Conference on Learning Representations (ICLR)*, 2019.

[188] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell, "Rethinking the value of network pruning," in *International Conference on Learning Representations (ICLR)*, 2019.

[189] W. Roth and F. Pernkopf, "Differentiable TAN structure learning for Bayesian network classifiers," in *International Conference on Probabilistic Graphical Models (PGM)*, 2020, pp. 389–400.

[190] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. H. W. Leong, M. Jahre, and K. A. Vissers, "FINN: A framework for fast, scalable binarized neural network inference," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2017, pp. 65–74.

[191] D. Sinha, H. Zhou, and N. V. Shenoy, "Advances in computation of the maximum of a set of Gaussian random variables," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 8, pp. 1522–1533, 2007.

[192] S. Ahn, A. Korattikara, and M. Welling, "Bayesian posterior sampling via stochastic gradient Fisher scoring," in *International Conference on Machine Learning (ICML)*, 2012, pp. 1591–1598.

[193] C. Li, C. Chen, D. E. Carlson, and L. Carin, "Preconditioned stochastic gradient Langevin dynamics for deep neural networks," in *AAAI Conference on Artificial Intelligence*, 2016, pp. 1788–1794.

[194] T. S. Ferguson, "A Bayesian analysis of some nonparametric problems," *Annals of Statistics*, vol. 1, no. 2, pp. 209–230, 1973.

[195] R. M. Neal, "Markov chain sampling methods for Dirichlet process mixture models," *Journal of Computational and Graphical Statistics*, vol. 9, no. 2, pp. 249–265, 2000.

[196] D. Blackwell and J. B. MacQueen, "Ferguson distributions via Pólya urn schemes," *Annals of Statistics*, vol. 1, no. 2, pp. 353–355, 1973.

[197] J. Sethuraman, “A constructive definition of Dirichlet priors,” *Statistica Sinica*, vol. 4, no. 2, pp. 639–650, 1994.

[198] W. J. Ewens, “Population genetics theory - the past and the future,” in *Mathematical and Statistical Developments of Evolutionary Theory*, S. Lessard, Ed. Springer Netherlands, 1990, pp. 177–227.

[199] C. E. Rasmussen, “The infinite Gaussian mixture model,” in *Advances in Neural Information Processing Systems (NIPS)*, 1999, pp. 554–560.

[200] S. J. Gershman and D. M. Blei, “A tutorial on Bayesian nonparametric models,” *Journal of Mathematical Psychology*, vol. 56, no. 1, pp. 1–12, 2012.

[201] S. Jain and R. M. Neal, “A split-merge Markov chain Monte Carlo procedure for the Dirichlet process mixture model,” *Journal of Computational and Graphical Statistics*, vol. 13, no. 1, pp. 158–182, 2004.

[202] Y. W. Teh, M. I. Jordan, M. J. Beal, and D. M. Blei, “Hierarchical Dirichlet processes,” *Journal of the American Statistical Association*, vol. 101, no. 476, pp. 1566–1581, 2006.

[203] D. M. Blei and M. I. Jordan, “Variational inference for Dirichlet process mixtures,” *Bayesian Analysis*, vol. 1, no. 1, pp. 121–143, 2006.

[204] J. Snoek, H. Larochelle, and R. P. Adams, “Practical Bayesian optimization of machine learning algorithms,” in *Advances in Neural Information Processing Systems (NIPS)*, 2012, pp. 2960–2968.

[205] F. N. Fritsch and R. E. Carlson, “Monotone piecewise cubic interpolation,” *SIAM Journal on Numerical Analysis*, vol. 17, no. 2, pp. 238–246, 1980.

[206] J. Chang and J. W. F. III, “Parallel sampling of DP mixture models using sub-cluster splits,” in *Advances in Neural Information Processing Systems (NIPS)*, 2013, pp. 620–628.

[207] W. Roth, R. Peharz, S. Tschiatschek, and F. Pernkopf, “Hybrid generative-discriminative training of Gaussian mixture models,” *Pattern Recognition Letters*, vol. 112, pp. 131–137, 2018.

[208] N. Friedman, D. Geiger, and M. Goldszmidt, “Bayesian network classifiers,” *Machine Learning*, vol. 29, no. 2–3, pp. 131–163, 1997.

[209] S. Tschiatschek, K. Paul, and F. Pernkopf, “Integer Bayesian network classifiers,” in *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD), Part III*, 2014, pp. 209–224.

[210] F. Pernkopf, M. Wohlmayr, and S. Tschiatschek, “Maximum margin Bayesian network classifiers,” *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, vol. 34, no. 3, pp. 521–532, 2012.

[211] R. Peharz, S. Tschiatschek, and F. Pernkopf, “The most generative maximum margin Bayesian networks,” in *International Conference on Machine Learning (ICML)*, vol. 28, 2013, pp. 235–243.

[212] D. Colombo and M. H. Maathuis, “Order-independent constraint-based causal structure learning,” *Journal of Machine Learning Research (JMLR)*, vol. 15, no. 116, pp. 3921–3962, 2014.

[213] D. M. Chickering, D. Heckerman, and C. Meek, “Large-sample learning of Bayesian networks is NP-hard,” *Journal of Machine Learning Research (JMLR)*, vol. 5, pp. 1287–1330, 2004.

[214] D. Heckerman, D. Geiger, and D. M. Chickering, “Learning Bayesian networks: The combination of knowledge and statistical data,” *Machine Learning*, vol. 20, pp. 197–243, 1995.

[215] D. Grossman and P. M. Domingos, "Learning Bayesian network classifiers by maximizing conditional likelihood," in *International Conference on Machine Learning (ICML)*, vol. 69, 2004.

[216] R. Peharz and F. Pernkopf, "Exact maximum margin structure learning of Bayesian networks," in *International Conference on Machine Learning (ICML)*, 2012, pp. 1047–1054.

[217] C. K. Chow and C. N. Liu, "Approximating discrete probability distributions with dependence trees," *IEEE Transactions on Information Theory*, vol. 14, no. 3, pp. 462–467, 1968.

[218] F. Pernkopf, M. Wohlmayr, and M. Mücke, "Maximum margin structure learning of Bayesian network classifiers," in *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, 2011, pp. 2076–2079.

[219] F. Pernkopf and M. Wohlmayr, "Stochastic margin-based structure learning of Bayesian network classifiers," *Pattern Recognition*, vol. 46, no. 2, pp. 464–471, 2013.

[220] M. Teyssier and D. Koller, "Ordering-based search: A simple and effective algorithm for learning Bayesian networks," in *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2005, pp. 584–590.

[221] G. Elidan, M. Ninio, N. Friedman, and D. Schuurmans, "Data perturbation for escaping local maxima in learning," in *National Conference on Artificial Intelligence (AAAI)*, 2002, pp. 132–139.

[222] A. Y. Ng and M. I. Jordan, "On discriminative vs. generative classifiers: A comparison of logistic regression and naive Bayes," in *Advances in Neural Information Processing Systems (NIPS)*, 2001, pp. 841–848.

[223] U. Fayyad and K. Irani, "Multi-interval discretization of continuous-valued attributes for classification learning," in *International Joint Conference on Artificial Intelligence (IJCAI)*, 1993, pp. 1022–1027.

[224] X. Zheng, B. Aragam, P. Ravikumar, and E. P. Xing, "DAGs with NO TEARS: Continuous optimization for structure learning," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2018, pp. 9472–9483.

[225] F. Pernkopf and J. A. Bilmes, "Efficient heuristics for discriminative structure learning of Bayesian network classifiers," *Journal of Machine Learning Research (JMLR)*, vol. 11, no. 81, pp. 2323–2360, 2010.

[226] Y. Shen, A. Choi, and A. Darwiche, "A new perspective on learning context-specific independence," in *International Conference on Probabilistic Graphical Models (PGM)*, 2020, pp. 425–436.

[227] S. Bengio and Y. Bengio, "Taking on the curse of dimensionality in joint distributions using neural networks," *IEEE Transactions on Neural Networks*, vol. 11, no. 3, pp. 550–557, 2000.

[228] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[229] H. Larochelle, D. Erhan, A. C. Courville, J. Bergstra, and Y. Bengio, "An empirical evaluation of deep architectures on problems with many factors of variation," in *International Conference on Machine Learning (ICML)*, vol. 227, 2007, pp. 473–480.

[230] A. Krizhevsky, "Learning multiple layers of features from tiny images," University of Toronto, Tech. Rep., 2009.

[231] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng, "Reading digits in natural images with unsupervised feature learning," in *Workshop on Deep Learning and Unsupervised Feature Learning @ NIPS*, 2011.

[232] P. Sermanet, S. Chintala, and Y. LeCun, "Convolutional neural networks applied to house numbers digit classification," in *International Conference on Pattern Recognition (ICPR)*, 2012, pp. 3288–3291.

[233] T. Hastie, R. Tibshirani, and J. H. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, 2nd Edition*, ser. Springer Series in Statistics. Springer, 2009.

[234] Y. LeCun, B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, and L. D. Jackel, "Handwritten digit recognition with a back-propagation network," in *Advances in Neural Information Processing Systems (NIPS)*, 1989, pp. 396–404.

[235] D. Dua and C. Graff, *UCI machine learning repository*, `http://archive.ics.uci.edu/ml`, 2017.

[236] P. W. Frey and D. J. Slate, "Letter recognition using Holland-style adaptive classifiers," *Machine Learning*, vol. 6, pp. 161–182, 1991.

[237] S. G. Waugh, "Extending and benchmarking cascade-correlation: Extensions to the cascade-correlation architecture and benchmarking of feed-forward supervised artificial neural networks," PhD thesis, University of Tasmania, 1995.

[238] D. Harrison and D. L. Rubinfeld, "Hedonic housing prices and the demand for clean air," *Journal of Environmental Economics and Management*, vol. 5, no. 1, pp. 81–102, 1978.

[239] I.-C. Yeh, "Modeling of strength of high-performance concrete using artificial neural networks," *Cement and Concrete Research*, vol. 28, no. 12, pp. 1797–1808, 1998.

[240] P. Tüfekci, "Prediction of full load electrical power output of a base load operated combined cycle power plant using machine learning methods," *International Journal of Electrical Power & Energy Systems*, vol. 60, pp. 126–140, 2014.

[241] P. Cortez, A. Cerdeira, F. Almeida, T. Matos, and J. Reis, "Modeling wine preferences by data mining from physicochemical properties," *Decision Support Systems*, vol. 47, no. 4, pp. 547–553, 2009.

[242] D. J. C. MacKay, "The evidence framework applied to classification networks," *Neural Computation*, vol. 4, no. 5, pp. 720–736, 1992.