

# Learning with Differentiable Algorithms

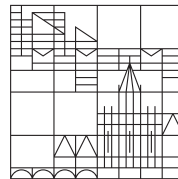
Doctoral Thesis for obtaining the Academic  
Degree Doctor of Natural Sciences (Dr. rer. nat.)

submitted by

Felix Petersen

at the University of Konstanz

Universität  
Konstanz



Faculty of Sciences  
Department of Computer and Information Science

Reviewers

Prof. Dr. Oliver Deussen  
Prof. Dr. Hilde Kuehne  
Prof. Dr. Tobias Sutter

August 22, 2022

**Submission & Defense**

This work was submitted on June 1, 2022, and successfully defended on August 22, 2022.

**Copyright**

©2022 Felix Petersen. All Rights Reserved.

**Distribution**

This work will be publicly accessible on arXiv and KOPS (the Institutional Repository of the University of Konstanz).

**Code**

The libraries accompanying this work will be publicly available under:

 **AlgoVision** <https://github.com/Felix-Petersen/algovision>

 **diffsort** <https://github.com/Felix-Petersen/diffsort>

 **difftopk** <https://github.com/Felix-Petersen/difftopk>

 **GenDR** <https://github.com/Felix-Petersen/gendr>

 **difflogic** <https://github.com/Felix-Petersen/difflogic> *(to be published soon)*

 **splitprop** <https://github.com/Felix-Petersen/splitprop> *(to be published soon)*

# Abstract

Classic algorithms and machine learning systems like neural networks are both abundant in everyday life. While classic computer science algorithms are suitable for precise execution of exactly defined tasks such as finding the shortest path in a large graph, neural networks allow learning from data to predict the most likely answer in more complex tasks such as image classification, which cannot be reduced to an exact algorithm. To get the best of both worlds, this thesis explores combining both concepts leading to more robust, better performing, more interpretable, more computationally efficient, and more data efficient architectures. The thesis formalizes the idea of algorithmic supervision, which allows a neural network to learn from or in conjunction with an algorithm. When integrating an algorithm into a neural architecture, it is important that the algorithm is differentiable such that the architecture can be trained end-to-end and gradients can be propagated back through the algorithm in a meaningful way. To make algorithms differentiable, this thesis proposes a general method for continuously relaxing algorithms by perturbing variables and approximating the expectation value in closed form, i.e., without sampling. In addition, this thesis proposes differentiable algorithms, such as differentiable sorting networks, differentiable renderers, and differentiable logic gate networks. Finally, this thesis presents alternative training strategies for learning with algorithms.



# Zusammenfassung

Klassische Algorithmen und maschinelle Lernsysteme wie neuronale Netze begegnen uns beide häufig im Alltag. Während sich klassische Informatik-Algorithmen für die präzise Ausführung genau definierter Aufgaben wie das Finden des kürzesten Weges in einem großen Graphen eignen, ermöglichen neuronale Netze das Lernen aus Daten, um bei komplexeren Aufgaben wie der Bildklassifizierung die wahrscheinlichste Antwort vorherzusagen. Um das Beste aus beiden Welten herauszuholen, wird die Kombination beider Konzepte untersucht, was zu robusteren, leistungsfähigeren, besser interpretierbaren, recheneffizienteren und dateneffizienteren Architekturen führt. Diese Dissertation formalisiert die Idee der algorithmischen Überwachung, welche es neuronalen Netzwerken ermöglicht, von einem Algorithmus oder in Verbindung mit einem Algorithmus zu lernen. Bei der Integration eines Algorithmus in eine neuronale Architektur ist es wichtig, dass der Algorithmus differenzierbar ist, sodass die Architektur Ende-zu-Ende trainiert werden kann und Gradienten auf sinnvolle Weise durch den Algorithmus zurück propagiert werden können. Um Algorithmen differenzierbar zu machen, präsentiert diese Arbeit ein allgemeines Verfahren zur stetigen Relaxierung von Algorithmen, indem wir Unsicherheit durch eine Verteilung einführen und den Erwartungswert in geschlossener Form approximieren. Überdies präsentiert diese Arbeit differenzierbare Algorithmen wie differenzierbare Sortier-Netzwerke, differenzierbare Renderer, und differenzierbare Logik-Gatter-Netzwerke. Abschließend werden alternative Trainingsstrategien für das Lernen mit Algorithmen vorgestellt.



# Acknowledgments

First and foremost, I want to thank my advisor, Oliver Deussen, for offering me a position in his research group. I am most grateful for Oliver's support and encouragement toward pursuing my own research interests, which slowly moved away from computer graphics to computer vision and core machine learning. While this direction is notably different from the core research direction of his lab, Oliver was a very effective and supportive advisor, providing invaluable guidance not only academically but also personally.

Moreover, I would like to give special thanks to my core collaborators Hilde Kuehne and Christian Borgelt. It was a great pleasure to collaborate on more than ten publications during the last years. I especially admire Hilde's compassion for simplifying complex matters and Christian's encyclopedic knowledge of classical machine learning concepts. I would also like to thank Tobias Sutter for collaborating on four papers, even though we have known each other for only half a year. I would like to thank my collaborators Mikhail Yurochkin and Yuekai Sun for introducing me to learning theory and reinforcing notation and clarity of theoretical results.

I would like to give special thanks to Bela Gipp for believing in me from the first moment, accepting me as a full member in his group, and offering me a position. I would like to extend my thanks to all members of Bela's lab. I would also like to thank Daniel Cohen-Or and his entire group for hosting me as a visiting researcher at the University of Tel Aviv for one semester.

Further, I would like to also give many thanks to all of my other collaborators, colleagues, and students. Especially, I want to mention Hendrik Strobelt, Bastian Goldluecke, Robert Denk, Debarghya Mukherjee, Nina Shvetsova, and Ruiheng Wu.

I would like to thank Oliver Seeck, Franz Kaertner, and Ingmar Hartl for hosting me for internships at DESY. Further, I would like to thank Tilman Irmscher for convincing me to submit my first research paper to *Jugend forscht*. I would like to give special thanks to Norina Procopan and Ulrike Leitner for enabling me to pursue my university studies during high school.

I want to thank my father for sparking my critical thinking and my early interest in science, as well as my grandmothers for their generosity and financial support during my early studies. I want to thank my best friend Nicolas Afritsch for our two decades-long close friendship. Further, I want to greatly thank the Welzel family for accepting me as if I were part of their family.



# Preface

In this thesis, we propose novel approaches for learning with differentiable algorithms and integrating algorithms into neural network architectures. Specifically, this thesis comprises the following contributions:

- ▶ We formalize the idea of algorithmic supervision, i.e., settings where an algorithm is applied to the predictions of a model and only the outputs of the algorithm are supervised.
- ▶ We propose analytical methods (contrasting stochastic methods) for relaxing algorithms and algorithmic concepts.
- ▶ We propose the idea of variable perturbation (contrasting input perturbations).
- ▶ We propose a general method for making arbitrary simple algorithms differentiable.
- ▶ We propose differentiable sorting networks, a continuous relaxation of sorting networks, and devise a theoretical characterization of monotonic and error-bounded differentiable sorting networks, leading to substantial empirical improvements on ranking supervision tasks.
- ▶ We propose differentiable top- $k$  classification learning, which relaxes the assumption of optimizing only for the top-1 or top-5 classification accuracy through the integration of differentiable sorting and ranking.
- ▶ We propose an array of differentiable renderers summarized as GenDR, the generalized differentiable renderer.
- ▶ We propose differentiable logic gate networks, a relaxation of logic gate networks, which achieve unprecedented inference speeds.
- ▶ We propose split backpropagation, which allows splitting the training of neural networks into multiple stages while maintaining end-to-end training capabilities.
- ▶ We propose Newton losses, an instance of split backpropagation, which incorporate second-order information of algorithmic loss functions into the training while training the actual neural network with fast first-order methods.
- ▶ We propose the Regularized Sampling Greedy Optimizer (RESGRO), another instance of split backpropagation, which is a very simple yet general method for learning end-to-end through hard-to-optimize functions, including non-differentiable functions.

This thesis is based on the following publications, all of which are collaborative works. For each publication, the contributions of each author are listed. In addition, the respective accompanying libraries, including respective implementations, are indicated by their logo.

- ▶ F. Petersen, C. Borgelt, H. Kuehne, and O. Deussen, *Learning with Algorithmic Supervision via Continuous Relaxations*, in Advances in Neural Information Processing Systems (NeurIPS), 2021. [1]  AlgoVision  
This work is covered in Chapter 2. The initial idea was conceived by the author. The algorithm, details, and implementation were developed by the author. All authors contributed to the text of the final publication.
- ▶ F. Petersen, C. Borgelt, H. Kuehne, and O. Deussen, *Differentiable Sorting Networks for Scalable Sorting and Ranking Supervision*, in Proceedings of the International Conference on Machine Learning (ICML), 2021. [2]  diffsort  
This work is covered in Chapter 3. The initial idea was conceived by the author. The algorithm, details, and implementation were developed by the author. All authors contributed to the text of the final publication.
- ▶ F. Petersen, C. Borgelt, H. Kuehne, and O. Deussen, *Monotonic Differentiable Sorting Networks*, in Proceedings of the International Conference on Learning Representations (ICLR), 2022. [3]  diffsort  
This work is covered in Chapter 3. The initial idea was conceived by the author. The algorithm, details, and implementation were developed by the author. All authors contributed to the text of the final publication.
- ▶ F. Petersen, H. Kuehne, C. Borgelt, and O. Deussen, *Differentiable Top-k Classification Learning*, in Proceedings of the International Conference on Machine Learning (ICML), 2022. [4]  difftopk

This work is covered in Chapter 4. The initial idea was conceived by the author. The algorithm, details, and implementation were developed by the author. All authors contributed to the text of the final publication.

- F. Petersen, B. Goldluecke, C. Borgelt, and O. Deussen, *GenDR: A Generalized Differentiable Renderer*, in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2022. [5] 

This work is covered in Chapter 5. The initial idea was conceived by the author. The algorithm, details, and implementation were developed by the author. All authors contributed to the text of the final publication.

- F. Petersen, C. Borgelt, H. Kuehne, and O. Deussen, *Deep Differentiable Logic Gate Networks*, under review, 2022. [6] 

This work is covered in Chapter 6. The initial idea was conceived by the author. The algorithm, details, and implementation were developed by the author. All authors contributed to the text of the final publication.

- F. Petersen, T. Sutter, C. Borgelt, H. Kuehne, and O. Deussen, *Newton Losses: Efficiently Including Second-Order Information into Gradient Descent*, under review, 2022. [7] 

This work is covered in Chapter 7. The initial idea was conceived by the author. The algorithm, details, and implementation were developed by the author. All authors contributed to the text of the final publication.

*We note that we have also prepared a substantially extended article based on this paper for submission to a journal. This extension adds RESGRO and a general formulation of splitting backpropagation, which are also included in this thesis.*

In addition to these publications, the author has also published other closely related works during the author's PhD research, which are not explicitly or only partially covered in this thesis. For each publication, the contributions of each author are listed. Also, the relation of each work to this thesis is discussed.

- F. Petersen, C. Borgelt, M. Yurochkin, H. Kuehne, and O. Deussen, *Propagating Distributions through Neural Networks*, under review, 2022. [8]

The initial idea was conceived by the author. The algorithm, details, and implementation were developed by the author. All authors contributed to the text of the final publication.

Making algorithms differentiable requires propagating distributions through them. To demonstrate the utility of propagating distributions, we propose propagating distributions also through neural networks. This leads to a variety of benefits, including uncertainty estimation and model robustness. The publication strengthens the proposed methods as it uses similar ideas for propagating distributions, and also supported the development of and gave insights about propagating distributions through algorithms.

- F. Petersen, D. Mukherjee, C. Borgelt, H. Kuehne, and O. Deussen, *APE-VAE: Training Variational Auto-Encoders with Approximate Evidence*, under review, 2022. [9]

The initial idea was conceived by the author. The algorithm, details, and implementation were developed by the author. The theoretical error-bound was conceived jointly by D. Mukherjee and the author. All authors contributed to the text of the final publication.

The paper proposes Approximate Evidence (APE), an alternative to the popular VAE training objective Evidence Lower Bound (ELBO). The proposed APE objective improves empirically over the ELBO objective and naturally supports sampling-free distribution propagation. Thus, the work applies propagating distributions through neural networks, specifically, through the decoder of a VAE, which leads to superior performance compared to sampling methods. APE is conceptually different from the ELBO as APE delivers an approximation to the evidence, which has a bounded approximation error, and the ELBO is a lower bound of the evidence, i.e., its optimization only indirectly improves the evidence.

- F. Petersen, B. Goldluecke, O. Deussen, and H. Kuehne, *Style Agnostic 3D Reconstruction via Adversarial Style Transfer*, in Proceedings of the IEEE Winter Conference on Applications of Computer Vision (WACV), 2022. [10]

The initial idea was conceived by the author. The algorithm, details, and implementation were developed by the author. All authors contributed to the text of the final publication.

As the output of differentiable relaxed algorithms usually does not distributionally match the outputs of their “hard” counterparts or real data, we propose to bridge this gap using domain adaptation. For this, we applied domain adaptation in the form of adversarial style transfer in the task of differentiable renderer supervised 3D geometry reconstruction.

- F. Petersen, A. H. Bermano, O. Deussen, and D. Cohen-Or, *Pix2Vex: Image-to-Geometry Reconstruction using a Smooth Differentiable Renderer*, in Arxiv, 2019. [11]

The initial idea was conceived by the author. The algorithm, details, and implementation were developed by the author. All authors contributed to the text of the final publication.

Pix2Vex was an early truly differentiable renderer, and is, to date, the only formally truly differentiable 3D mesh renderer. Later, this work inspired “Style Agnostic 3D Reconstruction via Adversarial Style Transfer” [10] as well as “GenDR: A Generalized Differentiable Renderer” [5].

- F. Petersen, C. Borgelt, and O. Deussen, *AlgoNet:  $C^\infty$  Smooth Algorithmic Neural Networks*, in Arxiv, 2019. [12]

The initial idea was conceived by the author. The algorithm, details, and implementation were developed by the author. All authors contributed to the text of the final publication.

AlgoNet was a largely theoretical work, describing the original ideas for “Learning with Algorithmic Supervision via Continuous Relaxations” [1] and “Differentiable Sorting Networks for Scalable Sorting and Ranking Supervision” [2]. The paper presented smooth WHILE-Programs, a minimalistic precursor of AlgoVision [1]. The work also discussed other concepts such as differentiable iterated function systems, weighted SoftMax, and SoftMedian.

- F. Petersen, C. Borgelt, and O. Deussen,  *$C^\infty$  Smooth Algorithmic Neural Networks for Solving Inverse Problems*, in the NeurIPS Deep Inverse Workshop, 2019. [13]

The initial idea was conceived by the author. The algorithm, details, and implementation were developed by the author. All authors contributed to the text of the final publication.

This workshop paper is largely based on the AlgoNet paper discussed above.

- F. Petersen, T. Sutter, C. Borgelt, D. Huh, H. Kuehne, Y. Sun, and O. Deussen, *ISAAC Newton: Input-based Approximate Curvature for Newton’s Method*, under review, 2022. [14]

The initial idea was conceived by the author. The algorithm, details, and implementation were developed by the author. All authors contributed to the text of the final publication.

ISAAC Newton is a method for more efficient training of neural networks by using selected second-order information. ISAAC Newton is a complement to Newton Losses. While Newton Losses incorporates second-order information of the loss into training, ISAAC Newton integrates second-order information derived from the input to each layer into training. Core implications of this work, in combination with other works presented in this thesis, are: Integrating second-order information using both ISAAC Newton and Newton Losses at the same time; and more efficient training of architectures such as differentiable logic gate networks.

- F. Petersen and T. Sutter, *Distributional Quantization*, under review, 2022. [15]

The initial idea was conceived by the author. The algorithm, details, and implementation were developed by the author. All authors contributed to the text of the final publication.

In this work, we present a simple and efficient scalar quantization algorithm based on the distribution of the input signal. We show the quantization algorithm to be error-optimal under some mild assumptions. This work was developed for differentiable logic gate networks and, more generally, for the goal of more efficient neural network training.

- F. Petersen\*, D. Mukherjee\*, Y. Sun, and M. Yurochkin, *Post-processing for Individual Fairness*, in Advances in Neural Information Processing Systems (NeurIPS), 2021. [16]

The initial ideas were conceived by all four authors jointly. D. Mukherjee and F. Petersen share joint first-authorship, distributed as follows: The theoretical properties were primarily devised and shown by D. Mukherjee. The algorithm, experiments, and implementation were developed by the author. All four authors contributed equally to the text of the final publication.

In the publication, we consider a setting where the learner only has access to limited information, i.e., the predictions of the original model and a similarity graph between individuals. We supervise the task of individual fairness through algorithmic knowledge about the similarity graph between individuals. For this, we cast the individual fairness problem as a graph smoothing problem corresponding to graph Laplacian regularization that preserves the desired “treat similar individuals similarly” interpretation. This work is an application of algorithmic supervision to fairness in a post-processing setting. The method is especially efficient compared to other individual fairness methods while achieving state-of-the-art on popular individual fairness benchmarks.

- D. Mukherjee\*, F. Petersen\*, M. Yurochkin, and Y. Sun, *Domain Adaptation meets Individual Fairness. And they get along.*, under review, 2022. [17]

The initial ideas were conceived by all four authors jointly. D. Mukherjee and F. Petersen share joint first-authorship, distributed as follows: The theoretical properties were primarily devised and shown by D. Mukherjee. The algorithm, experiments, and implementation were developed by the author. All four authors contributed equally to the text of the final publication.

In the publication, we show that domain adaptation methods can enforce individual fairness and vice versa.

- F. Petersen, M. Schubotz, and B. Gipp, *Towards Formula Translation using Recursive Neural Networks*, in Proceedings of the 11th Conference on Intelligent Computer Mathematics (CICM), Work-in-Progress Paper Track, 2018. [18]

The initial idea was conceived by the author. The algorithm, details, and implementation were developed by the author. All authors contributed to the text of the final publication.

In this work, we investigate recursive neural network architectures for machine translation of mathematical formulae from  $\text{\LaTeX}$  to the semantically-enhanced semantic  $\text{\LaTeX}$  markup language.

- F. Petersen, M. Schubotz, A. Greiner-Petter, and B. Gipp, *Neural Machine Translation for Mathematical Formulae*, under review, 2022. [19]

The initial idea was conceived by the author. The algorithm, details, and implementation were developed by the author. All authors contributed to the text of the final publication.

In this work, we apply convolutional sequence-to-sequence models and transformers to machine translation of mathematical formulae between different representations. Among others, we consider the translation of formulae from  $\text{\LaTeX}$  into the format of the computer algebra system Mathematica.

- N. Shvetsova, F. Petersen, R. Feris, and H. Kuehne, *Differentiable K-Nearest Neighbor Sorting for Self-supervised Learning*, under review, 2022. [20]

The initial idea was conceived by the authors jointly. All four authors contributed to the text of the final publication.

In this work, we apply differentiable sorting networks and differentiable top- $k$  to self-supervised learning on images. Specifically, we learn by enforcing that the  $k$  nearest neighbors of an image in the embedding space are the  $k$  image augmentations of the same image and not the negative examples based on other images.

# Contents

|                                                                         |             |
|-------------------------------------------------------------------------|-------------|
| <b>Abstract</b>                                                         | <b>iii</b>  |
| <b>Zusammenfassung</b>                                                  | <b>v</b>    |
| <b>Acknowledgments</b>                                                  | <b>vii</b>  |
| <b>Preface</b>                                                          | <b>ix</b>   |
| <b>Contents</b>                                                         | <b>xiii</b> |
| <b>1 Introduction</b>                                                   | <b>1</b>    |
| 1.1 Differentiable Algorithms . . . . .                                 | 3           |
| 1.2 Algorithmic Supervision . . . . .                                   | 5           |
| 1.3 Applications and Additional Related Work . . . . .                  | 6           |
| 1.4 Outline . . . . .                                                   | 9           |
| <b>2 Differentiable Algorithms</b>                                      | <b>11</b>   |
| 2.1 Related Work . . . . .                                              | 11          |
| 2.1.1 Stochastic Gradient Estimation . . . . .                          | 12          |
| 2.1.2 One-Step Linearization of Combinatorial Solvers . . . . .         | 13          |
| 2.1.3 Smooth Interpretation . . . . .                                   | 13          |
| 2.1.4 Neural Programs and Differentiable Program Interpreters . . . . . | 13          |
| 2.2 A General Method for Relaxing Algorithms . . . . .                  | 14          |
| 2.2.1 Relaxed Comparators . . . . .                                     | 16          |
| 2.2.2 Relaxed Indexing . . . . .                                        | 17          |
| 2.2.3 Complexity and Runtime Considerations . . . . .                   | 18          |
| 2.2.4 Alternative Distributions . . . . .                               | 18          |
| 2.2.5 The AlgoVision Library . . . . .                                  | 18          |
| 2.3 Sorting Supervision . . . . .                                       | 18          |
| 2.4 Shortest-Path Supervision . . . . .                                 | 21          |
| 2.5 Silhouette Supervision . . . . .                                    | 22          |
| 2.6 Levenshtein Distance Supervision . . . . .                          | 24          |
| <b>3 Differentiable Sorting and Ranking</b>                             | <b>27</b>   |
| 3.1 Related Work . . . . .                                              | 29          |
| 3.1.1 SoftSort and NeuralSort . . . . .                                 | 29          |
| 3.1.2 Optimal Transport / Sinkhorn Sort . . . . .                       | 30          |
| 3.1.3 FastSort . . . . .                                                | 30          |
| 3.1.4 Learning-to-Rank . . . . .                                        | 31          |
| 3.1.5 Neural Networks that Sort . . . . .                               | 31          |
| 3.1.6 Sorting Networks . . . . .                                        | 31          |
| 3.2 Sorting Networks . . . . .                                          | 32          |
| 3.2.1 Odd-Even Sorting Network . . . . .                                | 32          |
| 3.2.2 Bitonic Sorting Network . . . . .                                 | 32          |
| 3.3 Differentiable Sorting Networks . . . . .                           | 34          |
| 3.3.1 Differentiable Permutation Matrices . . . . .                     | 35          |
| 3.3.2 The Activation Replacement Trick $\varphi$ . . . . .              | 36          |
| 3.3.3 Monotonic Differentiable Sorting Networks . . . . .               | 38          |

|          |                                                                                  |           |
|----------|----------------------------------------------------------------------------------|-----------|
| 3.4      | Monotonicity and Error-Boundedness of Differentiable Sorting Operators . . . . . | 43        |
| 3.5      | Experiments . . . . .                                                            | 43        |
| 3.5.1    | The Activation Replacement Trick . . . . .                                       | 45        |
| 3.5.2    | Monotonic Differentiable Sorting Networks . . . . .                              | 47        |
| 3.5.3    | Runtime and Memory Analysis . . . . .                                            | 50        |
| <b>4</b> | <b>Differentiable Top-<math>k</math></b>                                         | <b>51</b> |
| 4.1      | Related Work . . . . .                                                           | 52        |
| 4.1.1    | Differentiable Top- $k$ Operators . . . . .                                      | 52        |
| 4.1.2    | Ranking and Top- $k$ Training Objectives . . . . .                               | 53        |
| 4.1.3    | Selection Networks . . . . .                                                     | 53        |
| 4.2      | Differentiable Top- $k$ . . . . .                                                | 53        |
| 4.2.1    | Differentiable Top- $k$ Networks . . . . .                                       | 54        |
| 4.3      | Top- $k$ Learning . . . . .                                                      | 55        |
| 4.3.1    | Implementation Details . . . . .                                                 | 56        |
| 4.4      | Experiments . . . . .                                                            | 57        |
| 4.4.1    | Training from Scratch . . . . .                                                  | 58        |
| 4.4.2    | Fine-Tuning . . . . .                                                            | 58        |
| 4.4.3    | Impact of the Distribution $P_K$ and Differentiable Sorting Methods . . . . .    | 59        |
| 4.4.4    | Differentiable Ranking Set Size $m$ . . . . .                                    | 61        |
| 4.4.5    | Comparison to the State-of-the-Art . . . . .                                     | 61        |
| <b>5</b> | <b>Differentiable Rendering</b>                                                  | <b>63</b> |
| 5.1      | Related Work . . . . .                                                           | 65        |
| 5.2      | The Generalized Differentiable Renderer . . . . .                                | 66        |
| 5.2.1    | Differentiable Occlusion Test . . . . .                                          | 66        |
| 5.2.2    | Aggregation . . . . .                                                            | 67        |
| 5.2.3    | Shading . . . . .                                                                | 68        |
| 5.3      | Instantiations of GenDR . . . . .                                                | 69        |
| 5.4      | Experiments . . . . .                                                            | 71        |
| 5.4.1    | Shape Optimization . . . . .                                                     | 71        |
| 5.4.2    | Camera Pose Optimization . . . . .                                               | 73        |
| 5.4.3    | Single-View 3D Reconstruction . . . . .                                          | 74        |
| <b>6</b> | <b>Differentiable Logic</b>                                                      | <b>77</b> |
| 6.1      | Related Work . . . . .                                                           | 78        |
| 6.1.1    | Differentiable Logics and Triangular Norms . . . . .                             | 78        |
| 6.1.2    | Learning Logic Gate Networks . . . . .                                           | 78        |
| 6.1.3    | Relaxed Connectivity in Networks . . . . .                                       | 79        |
| 6.1.4    | Evolutionary Learning of Networks . . . . .                                      | 79        |
| 6.1.5    | Learning of Decision Trees . . . . .                                             | 79        |
| 6.1.6    | Binary Neural Networks . . . . .                                                 | 80        |
| 6.1.7    | Sparse Neural Networks . . . . .                                                 | 80        |
| 6.2      | Logic Gate Networks . . . . .                                                    | 80        |
| 6.3      | Differentiable Logic Gate Networks . . . . .                                     | 81        |
| 6.3.1    | Training Considerations . . . . .                                                | 83        |
| 6.3.2    | Remarks . . . . .                                                                | 83        |
| 6.3.3    | Current Limitations and Opportunities . . . . .                                  | 84        |
| 6.4      | Experiments . . . . .                                                            | 85        |
| 6.4.1    | MONK's Problems . . . . .                                                        | 85        |
| 6.4.2    | Adult and Breast Cancer Data Sets . . . . .                                      | 86        |

|          |                                                                      |            |
|----------|----------------------------------------------------------------------|------------|
| 6.4.3    | MNIST . . . . .                                                      | 86         |
| 6.4.4    | CIFAR-10 . . . . .                                                   | 88         |
| 6.4.5    | Distribution of Logic Gates . . . . .                                | 90         |
| 6.4.6    | Hyperparameters and Model Architectures . . . . .                    | 90         |
| <b>7</b> | <b>Alternative Optimization Methods</b>                              | <b>93</b>  |
| 7.1      | Related Work . . . . .                                               | 94         |
| 7.2      | Splitting Backpropagation: A Two-Stage Optimization Method . . . . . | 94         |
| 7.3      | Newton Losses . . . . .                                              | 98         |
| 7.3.1    | Examples . . . . .                                                   | 100        |
| 7.4      | Regularized Sampling Greedy Optimizer (RESGRO) . . . . .             | 102        |
| 7.4.1    | Discussion . . . . .                                                 | 103        |
| 7.5      | Experiments . . . . .                                                | 103        |
| 7.5.1    | Classification . . . . .                                             | 103        |
| 7.5.2    | Sorting and Ranking Supervision . . . . .                            | 104        |
| 7.5.3    | Shortest-Path Supervision . . . . .                                  | 105        |
| <b>8</b> | <b>Conclusion</b>                                                    | <b>109</b> |
| 8.1      | Implications and Future Perspectives . . . . .                       | 109        |
| 8.2      | Extensions and Future Work . . . . .                                 | 111        |
|          | <b>SUPPLEMENTARY MATERIALS</b>                                       | <b>113</b> |
| <b>A</b> | <b>Distributions</b>                                                 | <b>115</b> |
| <b>B</b> | <b>T-Norms and T-Conorms</b>                                         | <b>119</b> |
| <b>C</b> | <b>Relaxed Minimum and Maximum</b>                                   | <b>123</b> |
| C.1      | Monotonicity of the Reciprocal Relaxed Min and Max . . . . .         | 125        |
| C.2      | Monotonicity of the Cauchy Relaxed Min and Max . . . . .             | 126        |
|          | <b>Bibliography</b>                                                  | <b>127</b> |



Four millennia ago, the Egyptians devised an algorithm for multiplying two numbers, which is the earliest record of any algorithm [21]. In 1843, Ada Lovelace published the first computer program of an algorithm and envisioned modern applications of computers such as art and music, at a time when such a computer was not even built [22, 23]. A century later, in 1943, McCulloch and Pitts [24] devised the first mathematical model of neural networks based on observations of the biological processes in the brain. Within the last decade, artificial neural network-based approaches gained a lot of attention in research. This resurgence can be attributed to advances in hardware [25], software [26–29], the development of convolutional networks [30, 31], and the supremacy of deep learning on many tasks, such as image classification [32, 33].

Today, classical algorithms and machine learning systems like neural networks are both abundant in everyday life. While classical computer science algorithms are suitable for precise execution of exactly defined tasks such as finding the shortest path in a large graph, neural networks allow learning from data to predict the most likely answer in more complex tasks such as image classification, which cannot be reduced to an exact algorithm. To get the best of both worlds, in this thesis, we explore combining *classical computer science algorithms* and *neural networks*, or, more generally, machine learning. This leads to more robust, better performing, more interpretable, more computationally efficient, and more data efficient architectures. Herein, model robustness can be achieved through a provably correct algorithm applied to embeddings. The model performance can be computationally improved when we can reduce the computational complexity of the neural network by replacing part of it by a fast algorithm. Also, with respect to accuracy, the performance can be improved as there is a smaller potential for errors, and the domain knowledge supports the network. Correspondingly, these models can also be more interpretable, as the inputs to algorithms are typically (by definition) interpretable. Finally, as algorithmic supervision is typically a kind of weakly-supervised learning, the level of supervision is reduced and the models are more data / label efficient.

Typically, neural networks are trained with stochastic gradient descent (SGD) or preconditioned SGD methods, such as the Adam optimizer [34]. These methods are based on computing the gradient (i.e., derivative) of a loss function with respect to the model’s parameters. This gradient indicates the direction of the steepest ascent of the loss. As minimizing the loss improves the model, we can optimize the model by going (in the model’s parameter space) in the opposite direction of the gradient, i.e., gradient descent. The derivative of the loss with respect to the model parameters can be efficiently computed using the backpropagation algorithm [35], which, in today’s deep learning frameworks [26, 29], is implemented as backward-mode automatic differentiation.

Gradient-based learning requires that all involved operations are differentiable; however, many interesting operations like sorting algorithms are

|                                                      |   |
|------------------------------------------------------|---|
| 1.1 Differentiable Algorithms                        | 3 |
| 1.2 Algorithmic Supervision                          | 5 |
| 1.3 Applications and Additional Related Work . . . . | 6 |
| 1.4 Outline . . . . .                                | 9 |

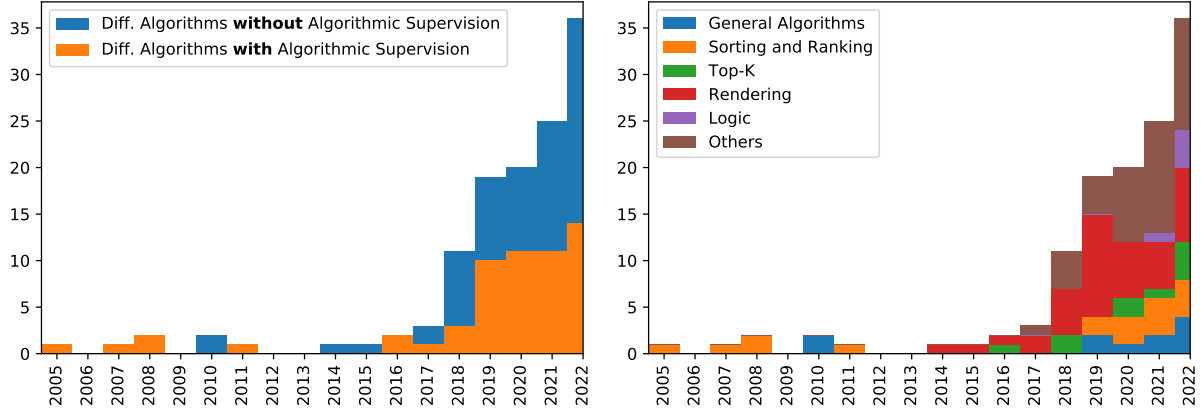


**Figure 1.1:** Trends in the field of learning with differentiable algorithms based on the references in this work. The histograms show the number of works in each year. The field started its formation in 2018 / 2019. *Left:* comparing works with and without algorithmic supervision. *Right:* categorizing works wrt. the kind of algorithm that was relaxed. Here, the first 5 categories correspond to Chapters 2–6 of this work. For 2022, as only 5 months passed at the time of writing, we doubled the height and halved the width of the bars, maintaining the area per paper.

non-differentiable. This is because conditional statements like `if` are piecewise constant, i.e., they have a derivative of 0, with the exception of the transitions (i.e., “jumps”) between `true` and `false`, at which their derivative is undefined. Accordingly, gradient-based learning with (non-differentiable) algorithms is generally not possible. Therefore, in this work, we focus on making algorithms differentiable through continuous relaxations. The primary idea of continuous relaxations is to introduce a level of uncertainty into algorithms, which, e.g., can lead to smooth transitions between `true` and `false` in `if` statements, making the algorithm fully differentiable. We note that when going beyond backpropagation, e.g., via RESGRO losses, as introduced in Chapter 7, differentiability and smoothness are not strictly necessary, but still desirable. We also note that gradient-free optimization on algorithms in conjunction with gradient-based learning of a neural network is introduced in this work, and that differentiable algorithms typically outperform gradient-free methods.

Learning with differentiable algorithms can be grouped into 2 disciplines:

- Differentiable Algorithms, i.e., the study of how to backpropagate through algorithms and obtain meaningful gradients.
- Algorithmic Supervision, i.e., incorporating algorithmic knowledge into the training of neural network models.

Learning with differentiable algorithms is a relatively new and recent field in machine learning. Specifically, apart from a few individual earlier works, the fields of differentiable algorithms and algorithmic supervision gained momentum in 2018. We have visualized this in the survey histograms in Figure 1.1. Here, we categorized all related works on differentiable algorithms into those which apply algorithmic supervision and those which do not (*left*). Further, we categorized them wrt. the kind of differentiable algorithm they propose or apply (*right*). We note that the author proposed and started his work on differentiable algorithms in 2018, i.e., at the beginning of the formation of the field. We also note that the field is growing and has only recently seen direct application to real-world problems.

The following two sections briefly cover differentiable algorithms and algorithmic supervision.

## 1.1 Differentiable Algorithms

In general, to allow for end-to-end training of neural architectures with integrated algorithms, the challenge is to estimate the gradients of a respective algorithm, e.g., via a differentiable approximation.

The core idea behind most differentiable algorithms is smoothing, i.e., perturbation by a probability distribution. Differentiable algorithms can be classified into two broad methodological schools:

- ▶ smoothing via stochastic perturbation by sampling;
- ▶ smoothing via analytical perturbation solved in closed form.

The primary focus of this work lies in analytical perturbations to be solved in closed form. However, we also introduce and discuss stochastic sampling methods in detail.

A seminal work in the domain of sampling-based stochastic perturbations is stochastic smoothing by Abernethy *et al.* [36]. Building on this work, Berthet *et al.* [37] propose extensions of stochastic smoothing for algorithmic supervision. In the field of reinforcement learning, stochastic smoothing and its derivatives are also known as the score function estimator [38, 39] or REINFORCE [40].

In the domain of analytical perturbations that are solved in closed form, many methods are based on analytical distribution propagation. A popular method of propagating distributions is moment matching [41–44]. However, in the case of correlated random variables, moment matching becomes intractable [45] and for distributions like the Cauchy distribution (which has infinite and undefined moments) moment matching is undefined. But the Cauchy distribution is an especially important distribution, e.g., in the case of differentiable sorting where it leads to monotonic differentiable sorting networks [3]. Therefore, propagating distributions via local linearization, as discussed by Petersen *et al.* [8, 9] and which also supports distributions like Cauchy, is a suitable alternative to moment matching and has been shown to be optimal in terms of total variation for ReLU neural networks [8]. These methods for propagating distributions are especially important for modeling correlations, which is necessary when modeling input perturbation.

For smoothing algorithms, there are two conceptually different mathematical concepts:

- ▶ perturbations of inputs to an algorithm;
- ▶ perturbations of variables / conditions in an algorithm.

Smoothing the inputs to an algorithm means relaxing an algorithm  $f(x)$  to  $\mathbb{E}[f(x + \epsilon)]$  where  $\epsilon$  is drawn from some distribution. Accordingly, smoothing an algorithm's inputs typically does not require assumptions about an algorithm and can be solved with stochastic smoothing: it is independent of the algorithm used, assuming that the algorithm is correct.

On the other hand, smoothing variables and conditions in an algorithm corresponds to relaxing statements like `if  $x > 0$`  to `if  $x + \epsilon > 0$` . In this case, the resulting function depends on the choice of algorithm for a given problem. This means that for a good choice of algorithm (e.g., Bellman-Ford), smoothing variables and conditions can lead to better relaxations than smoothing of inputs; however, for a poor or inadequate choice of algorithm

(e.g., Dijkstra), an uninformed smoothing of inputs would perform better. Further discussion of this can be found in Chapter 2.

While smoothing of inputs lends itself to stochastic smoothing and can be intractable for analytical propagation methods, smoothing variables can be difficult for stochastic methods and can be evaluated using analytical propagation methods. The contributions in this thesis focus on smoothing variables in an algorithm; however, we also consider smoothing of inputs.

|            | variables                     | inputs                    |
|------------|-------------------------------|---------------------------|
| analytical | [1–5],<br>[11], [46],<br>[47] | ([48])                    |
| stochastic | [49]                          | [36], [37],<br>[50], [51] |

**Figure 1.2:** Classification of a selection of differentiable algorithms into analytical vs. stochastic evaluation and modeling of variables vs. inputs. We note that [48] does not provide gradients.

In Figure 1.2, we classify a selection of perturbation-based differentiable algorithms with respect to methodology (analytical vs. stochastic perturbations) and assumptions (perturbation of inputs vs. perturbation of variables). We can observe that the majority of methods are either analytical perturbations of variables or stochastic perturbations of inputs, which is also expected according to the analysis above.

In the domain of differentiable rendering [11, 52, 53], the method of approximate analytical perturbations is popular [52] because methods like stochastic smoothing are typically intractable due to the high dimensionality of the involved spaces (image space and the space of 3D models). Accordingly, these methods model perturbations of variables or can be considered to model perturbations of variables (as most of them use an ad-hoc relaxation and do not identify their underlying probabilistic model). Modeling input perturbations using analytical methods is typically intractable due to the complexity of algorithms. But this thesis provides evidence that variable perturbations might indeed be better for high-quality gradients, given that an adequate algorithm is available for a given problem. There are also differentiable renderers based on Monte Carlo sampling [50, 54, 55]. Lidec *et al.* [49] model variable perturbations using the sampling-based stochastic smoothing technique.

In the domain of differentiable sorting, there are methods based on entropic regularization rather than modeling perturbations. Specifically, Cuturi *et al.* [56] reduced sorting to an optimal transport (OT) problem and induce an entropic regularization which relaxes the OT problem and thereby makes sorting smoothly differentiable. Entropic regularization, in the context of optimal transport, means that we enforce uncertainty in the solution of the OT problem, i.e., we regularize the entropy of the solution to be sufficiently large. In a seminal work, Cuturi [57] had shown that by introducing this entropic regularization OT can be solved via the iterative Sinkhorn algorithm, which was a milestone in optimal transport as it substantially simplified many OT problems and delivers a fast approximate OT method. As the entropic regularization enforces a smoothly differentiable distributional solution to an OT problem, it is suitable as a differentiable sorting and ranking algorithm [56].

Apart from smoothing-based differentiable algorithms, there are also heuristic relaxations of algorithms. These heuristics are usually specific to a certain setting and typically do not come with the benefits of smoothing, such as smooth optimization objectives. Examples of heuristic-based differentiable sorting algorithms are NeuralSort [58] and SoftSort [59]. In this work, we do not focus on heuristic approaches but include them for completeness.

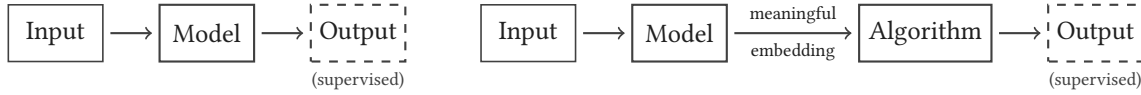


**Figure 1.3:** Direct supervision (on the left) in comparison to algorithmic supervision (on the right).

## 1.2 Algorithmic Supervision

Artificial neural networks have shown their ability to solve various problems, ranging from classical tasks in computer science, such as machine translation [60] and object detection [61], to many other topics in science, such as protein folding [62]. Simultaneously, classical algorithms exist, which typically solve predefined tasks based on a predefined control structure, such as sorting or shortest-path computation, and for which guarantees about their behavior can be deduced. Recently, research has started to combine both elements by integrating algorithmic concepts into neural network architectures. Those approaches allow training neural networks with alternative supervision strategies, such as learning 3D representations via a differentiable renderer [46, 63] or training neural networks with ordering information [56, 58]. We unify these alternative supervision strategies, which integrate algorithms into the training objective, as algorithmic supervision:

Algorithmic supervision is the idea of integrating an algorithm into the training objective of a neural network model. This lets us directly incorporate algorithmic domain knowledge of the problem, leading, i.a., to weaker supervision requirements and / or improved performance. In this work, we formally define algorithmic supervision as follows:

**Definition 1.1** (Algorithmic Supervision) *In algorithmic supervision, an algorithm is applied to the predictions of a model and the outputs of the algorithm are supervised. In contrast, in direct supervision, the predictions of a model are directly supervised. This is illustrated in Figure 1.3.*

Examples of algorithmic supervision that we cover in this thesis are:

- ▶ Sorting and Ranking Supervision
- ▶ Rendering and Silhouette Supervision
- ▶ Shortest-Path Supervision
- ▶ Editing Distance Supervision
- ▶ Top- $k$  Supervision

**Sorting and Ranking Supervision** Sorting and Ranking Supervision allow training based on ordering information. Specifically, it is the setting in which the relative order or ranking of a set of elements is given, while their absolute values remain unsupervised. This is achieved by using a differentiable ranking algorithm to rank model predictions for a set of elements and training by backpropagating the discrepancy between the resulting ranking and the ground truth ranking back to the neural network. For differentiable sorting, an array of methods has been proposed in recent years, which includes NeuralSort [58], SoftSort [59], Optimal Transport Sort [56], and FastSort [64], all of which appeared after we started working on differentiable sorting. This setting is covered in Chapter 2, Chapter 3, and Chapter 7.

**Rendering and Silhouette Supervision** Rendering and Silhouette Supervision allow learning 3D shapes from 2D information. The idea is to differentiably render a 3D model and then compare the rendering to a reference image or the rendered silhouette to a reference silhouette. By reducing the discrepancy between the rendering and a reference image, we can improve the 3D model, or the neural network that predicted the 3D model. In the domain of differentiable rendering, a body of recent research has considered rendering and silhouette supervision, including but not limited to: [1, 5, 10, 11, 46, 47, 50, 52, 63, 65]. This setting is covered in Chapter 2 and Chapter 5.

**Shortest-Path Supervision** Shortest-Path Supervision allows learning based on shortest-path information. By applying a differentiable shortest-path algorithm to a cost embedding, we obtain a relaxed shortest path. Given a reference shortest path, we can compute the discrepancy between the relaxed and the reference shortest paths as our loss. By minimizing the loss, we improve the cost embedding or the neural network that predicted it. Recently, a few related works have considered the setting of shortest-path supervision [1, 37, 66]. This setting is covered in Chapter 2 and Chapter 7.

**Top- $k$  Supervision** The idea of top- $k$  supervision is that we want to enforce that a target class is among the top- $k$  classes, e.g., that it is among the top-5 classes. An extension of this core idea is to relax the assumption that  $k$  is a constant to  $k$  being drawn from a distribution. For example, by setting  $k$  with a 50% probability to 1 and with a 50% probability to 5, we can optimize both the top-1 and top-5 objective at the same time and make the model more robust, which even allows improving top-1 accuracy. While the idea of top- $k$  supervision has been established for a long time [67–69], relaxing  $k$  is novel and is covered in Chapter 4.

For an illustration of the history of algorithmic supervision as a part of the field of differentiable algorithms, see Figure 1.1 (left).

## 1.3 Applications and Additional Related Work

There is a large variety of applications and use cases for integrating algorithmic components into machine learning. In addition to the methods and applications covered above, in the following, we cover additional applications and related works which use differentiable algorithms. Some of these build on the work of this thesis, and many others could also be based on the methods proposed in this work (e.g., methods building on top of an “arbitrary” differentiable sorting algorithm). For an illustration of trends in differentiable algorithms, see Figure 1.1 (right).

While some of these works cover single algorithmic concepts, e.g., sorting [56, 58, 64] or rendering [46, 47, 63], others have covered wider areas such as dynamic programming [70, 71] or gradient estimation for general optimizers [37, 66, 72, 73]. Paulus *et al.* [74] employ the Gumbel-Softmax [75] distribution to estimate gradients for concepts such as subset selection. Cuturi *et al.* [76] and Blondel *et al.* [77] present approaches for differentiable dynamic time warping. Dasgupta *et al.* [78] model box parameters with Gumbel distributions to obtain soft differentiable box embeddings.

In the domain of recommender systems [79, 80], early learning-to-rank works already appeared in the 2000s [81–84], but more recently Lee *et al.* [85] propose differentiable ranking metrics, Swezey *et al.* [86] and Pobrotyn *et al.* [87] propose PiRank and NeuralNDCG, respectively, which both rely on differentiable sorting, and Rolinek *et al.* [72] propose optimizing ranking metrics with their blackbox differentiation method [66]. Adams *et al.* [88] proposed a differentiable ranking method based on the Sinkhorn and Knopp algorithm [89, 90]. Kong *et al.* [91] propose Rankmax, an alternative to Softmax based on an adaptive projection, for learning-to-rank.

Plötz *et al.* [92] propose a continuous relaxation of the  $k$ -nearest-neighbor algorithm and propose integrating it into neural architectures. Grover *et al.* [58] apply differentiable sorting to  $k$ -nearest-neighbor learning in addition to ranking supervision. Based on ideas from differentiable sorting, Xie *et al.* [93] propose a differentiable top- $k$  operator based on optimal transport and the Sinkhorn algorithm [57]. They apply their method to  $k$ -nearest-neighbor learning ( $k$ NN), differential beam search with sorted soft top- $k$ , and top- $k$  attention for machine translation. Cordonnier *et al.* [51] use perturbed optimizers [37] to derive a differentiable top- $k$  operator, which they use for differentiable image patch selection. Goyal *et al.* [94] propose a continuous top- $k$  operator for differentiable beam search. Pietruszka *et al.* [95] propose the differentiable successive halving top- $k$  operator to approximate the normalized Chamfer Cosine Similarity ( $nCCS@k$ ). Patel *et al.* [96] propose a differentiable surrogate for recall@ $k$  for image retrieval.

In machine translation, Bahdanau *et al.* [97] investigate soft alignments, while Collobert *et al.* [98] present a fully differentiable beam searching algorithm. In speech recognition, Bahdanau *et al.* [99] propose differentiable surrogate task losses for sequence prediction tasks. Carr *et al.* [100] propose self-supervised learning of audio representation via differentiable sorting-based ranking supervision of shuffled patches of spectrograms of the audio. Huang *et al.* [101] propose learning relational surrogate losses supervised through differentiable sorting networks [2].

Algorithmic supervision also has similarities to neuro-symbolic systems [102, 103] as they both incorporate algorithmic domain knowledge of the problem. Yang *et al.* [104] combine neuro-symbolic learning with differentiable symbolic execution for safety losses in the context of safe learning.

Over the last years, many differentiable renders for different types of 3D representations have been proposed. These include renderers for 3D meshes [46, 47, 63, 65], voxels [105], point clouds [106], surfels [107], signed distance functions [108, 109], and other implicit representations [110–113]. A notable mention is the work by Sitzmann *et al.* [111], who propose a differentiable ray marching algorithm for learning scene representation networks. Deng *et al.* [114] propose a method for learning convex decompositions of shapes via a differentiable convex set indicator. Peng *et al.* [115] propose a differentiable Poisson solver, which they apply to various 3D tasks. Rakotosaona *et al.* [116] propose a differentiable surface triangulation algorithm for triangle mesh optimization.

Cho *et al.* [117] proposed differentiable spline approximations (including differentiable non-uniform rational basis splines (NURBS)) for tasks such as image segmentation, 3D point cloud reconstruction, and finite element analysis.

In another line of work, drawing strokes and sketching have been made differentiable [118] for learning to draw [119] and, more generally, also rendering of vector graphics has been made differentiable [120]. In a different but related line of work, Petersen *et al.* [12] and Scott [121] present differentiable iterated function systems.

There is a line of work on learning with differentiable physics simulations ([122–124] among many others). Holl *et al.* [125] proposed PhiFlow, a differentiable partial differential equation (PDE) solver for physics simulations like fluid dynamics, which can be used for learning [126]. Sitzmann *et al.* [127] propose a differentiable simulation of optics and apply it to achromatic extended depth of field as well as super-resolution. Ingraham *et al.* [128] propose learning protein structure with a differentiable simulator. Fu *et al.* [129] propose differentiable scaffolding trees for molecular optimization. Another line of works considered learning with combinatorial solvers for NP-hard problems [66, 130].

Bangaru *et al.* [131] propose a method for making programs that solve integrals (or may be expressed as solving an integral) differentiable: instead of discretizing the algorithm from an integral to a sum and then differentiating it, they propose to automatically differentiate the symbolic form of the integral and then discretize it.

Domke [132] proposes a finite difference implicit differentiation method. Vlastelica *et al.* [66] build on this one-step linearization method and apply it to a range of algorithmic supervision settings. Niepert *et al.* [133] propose a method for backpropagating through discrete structures via sampling from exponential family distributions and implicit differentiation. Blondel *et al.* [134] present an overview of implicit differentiation methods and present an efficient, modular, and general framework for implicit differentiation.

Amos *et al.* [135] and Agrawal *et al.* [136] consider including optimization problems as layers into neural networks; by making the solution of the optimization problem differentiable, these layers can be trained.

Djolonga *et al.* [137] integrate submodular minimization into neural architectures, and specifically, by using a graph cut algorithm, they achieve improved image segmentation performance. Rolinek *et al.* [138] apply their blackbox differentiation method [66] to graph matching for keypoint correspondence tasks. Charpentier *et al.* [139] propose a method for differentiable sampling of directed acyclic graphs via differentiable sorting and ranking methods. Zheng *et al.* [140] and Lorch *et al.* [141] each propose a relaxation of the constraint of a directed graph being acyclic to a differentiable loss, which they use for structure learning.

Riad *et al.* [142] build on spectral pooling [143] and make the stride of a convolution differentiable by expressing it as a masking on the Fourier domain, which allows learning the optimal stride during training. Simultaneously, Romero *et al.* [144], propose convolution with differentiable kernel sizes by parameterizing the kernel as an implicit neural representation and masking the kernel using a Gaussian distribution to enforce local support. Thandiackal *et al.* [145] propose a differentiable zooming method for multiple instance learning.

Chen *et al.* [146] propose a differentiable Voronoi tessellation for semi-discrete normalizing flows. Corenflos *et al.* [147] propose a differentiable particle filtering algorithm via optimal transport with entropic regularization.

Engel *et al.* [148] propose DDSP, a differentiable digital signal processing framework.

## 1.4 Outline

This thesis is organized into 8 chapters: The chapters build chronologically upon each other, specifically, Chapter 2 introduces core ideas and methods on which each of the Chapters 3–6 build. Chapter 7 introduces alternative optimization methods and is therefore largely independent of the chronological flow of Chapters 2–6, but knowledge from Chapters 2 and 3 is recommended for an in-depth understanding of the applications.

**Chapter 1** introduces the core ideas of learning with differentiable algorithms and covers important related work and applications.

**Chapter 2** details general approaches for differentiable algorithms and algorithmic supervision. For that, the chapter gives a general overview of differentiable algorithms and can be seen as an extension to the introduction. The following chapters build on the ideas presented in this chapter, and each of them considers a specific class of differentiable algorithms in greater depth. This chapter is accompanied by the  **AlgoVision** library.

**Chapter 3** investigates differentiable sorting and ranking approaches, with a focus on differentiable sorting networks. We start by presenting differentiable sorting approaches and, through careful theoretical analysis, we derive improved differentiable sorting operators. This chapter is accompanied by the  **diffsort** library.

**Chapter 4** presents differentiable top- $k$  approaches, conceptually building on differentiable sorting and ranking methods. Specifically, we introduce differentiable top- $k$  networks, an improvement over differentiable sorting networks for the top- $k$  operator. Building on differentiable top- $k$ , we present top- $k$  classification learning and achieve competitive performance on the ImageNet classification task. This chapter is accompanied by the  **difftopk** library.

**Chapter 5** covers differentiable rendering. We cover various approaches in differentiable rendering and present GenDR, the generalized differentiable renderer, which (at least approximately) includes most of the existing differentiable renderers and also generalizes to new differentiable renderers. This chapter is accompanied by the  **GenDR** library.

**Chapter 6** proposes differentiable logic gate networks, which are a relaxation of logic gate networks and can, therefore, be trained. This allows extremely fast inference speeds as the resulting logic gate networks can be executed natively on common hardware as such hardware primarily operates on logic gates in the first place. This is an example of a differentiable algorithm that can be trained and is not necessarily connected to algorithmic supervision. This chapter is accompanied by the  **difflogic** library.

**Chapter 7** considers alternative optimization strategies. Specifically, it discusses splitting backpropagation, a general two-stage optimization algorithm based on regularization, which allows optimizing algorithmic losses with a different optimizer than used for optimizing the neural network. Split

backpropagation also allows splitting a neural network itself into multiple sections and can be extended to multiple splits, which can then all be trained end-to-end, even when subsections are trained with alternative optimizers or are even non-differentiable. This chapter is accompanied by the  **splitprop** library.

We summarize the major contributions of this thesis and discuss directions for future research in **Chapter 8**.

## Differentiable Algorithms

This chapter gives a general overview of differentiable algorithms and is an extension to the introduction by covering general methods for differentiable algorithms. In the following chapters, we build on these ideas and consider specific differentiable algorithms in greater depth.

In general, to allow for end-to-end training of neural architectures with integrated algorithms, the challenge is to estimate the gradients of a respective algorithm, e.g., by a differentiable approximation. For this, many existing solutions are tailored to specific problems like, e.g., differentiable sorting or rendering. In contrast to these approaches, in this chapter, we cover general methods for differentiable algorithms, which are not limited to specific problems.

In this chapter, we propose a general approach for making algorithms differentiable and estimating their gradients. Specifically, we propose continuous relaxations of different algorithmic concepts such as comparators, conditional statements, bounded and unbounded loops, and indexing. For this, we perturb those variables in a discrete algorithm, for which we want to compute a gradient, by logistic distributions. This allows estimating the expected value of an algorithm's output sampling-free and in closed form, e.g., compared to methods approximating the distributions via Monte-Carlo sampling (e.g., [37]). To keep the computation feasible, we approximate the expectation value by merging computation paths after each conditional block in sequences of conditional blocks. For nested conditional blocks, we compute the exact expectation value without merging conditional cases. This trade-off allows merging paths in regular intervals and thus alleviates the need to keep track of all possible path combinations. As we model perturbations of variables when they are accessed, all distributions are independent, which contrasts the case of modeling input perturbations, where all computation paths would have to be handled separately.

To demonstrate the practical aspects, we apply the proposed approach in the context of four tasks that make use of algorithmic supervision to train a neural network, namely sorting supervision [56, 58, 149], shortest-path supervision [37, 66], silhouette supervision (differentiable rendering) [46, 63, 105], and, finally, Levenshtein distance supervision. We show that the proposed method outperforms state-of-the-art methods on sorting supervision and shortest-path supervision, and performs comparably on silhouette supervision. Later in this work, we will cover sorting supervision and silhouette supervision in greater detail in Chapters 3 and 5, respectively.

|     |                                              |    |
|-----|----------------------------------------------|----|
| 2.1 | Related Work . . . . .                       | 11 |
| 2.2 | A General Method for Relaxing Algorithms . . | 14 |
| 2.3 | Sorting Supervision . . .                    | 18 |
| 2.4 | Shortest-Path Supervision . . . . .          | 21 |
| 2.5 | Silhouette Supervision .                     | 22 |
| 2.6 | Levenshtein Distance Supervision . . . . .   | 24 |

### 2.1 Related Work

In the following, we cover related works in the domain of general differentiable algorithms. While Sections 2.1.1 and 2.1.2 cover methods that are strongly related and applicable to algorithmic supervision, Sections 2.1.3 and 2.1.4 cover methods that are not directly applicable to algorithmic supervision and which were developed in a different context. For additional

“Smooth WHILE-Programs”, which is a minimalistic (yet Turing-complete) precursor of the presented method, was previously proposed by Petersen *et al.* [12].

applications of differentiable algorithms, see Section 1.3, and for specific differentiable algorithms, see the respective subsections in each experimental section of this chapter.

### 2.1.1 Stochastic Gradient Estimation

Stochastic gradient estimation is a popular method for differentiating algorithms. These methods focus on modeling perturbations of the inputs to an algorithm via stochastic sampling. Relevant works in this direction include stochastic smoothing [36], perturbed optimizers [37], and stochastic softmax tricks [74].

**Stochastic Smoothing** In this paragraph, we cover the method of stochastic smoothing [36]. This method regularizes a non-differentiable and discontinuous loss function  $\ell(y)$  by randomly perturbing its input with random noise  $\epsilon$  (i.e.,  $\ell(y + \epsilon)$ ). The loss function is then approximated as  $\ell(y) \approx \ell_\epsilon(y) = \mathbb{E}[\ell(y + \epsilon)]$ . While  $\ell$  is not differentiable, its smoothed stochastic counterpart  $\ell_\epsilon$  is differentiable and the corresponding gradients can be estimated via the following result.

**Lemma 2.1** (Exponential Family Smoothing, adapted from [36, Lemma 1.5]) *Given a distribution over  $\mathbb{R}^m$  with a probability density function  $\mu$  of the form  $\mu(\epsilon) = \exp(-\nu(\epsilon))$  for any twice-differentiable  $\nu$ , then*

$$\begin{aligned}\nabla_y \ell_\epsilon(y) &= \nabla_y \mathbb{E}[\ell(y + \epsilon)] = \mathbb{E}[\ell(y + \epsilon) \nabla_\epsilon \nu(\epsilon)], \\ \nabla_y^2 \ell_\epsilon(y) &= \nabla_y^2 \mathbb{E}[\ell(y + \epsilon)] = \mathbb{E}\left[\ell(y + \epsilon) \left( \nabla_\epsilon \nu(\epsilon) \nabla_\epsilon \nu(\epsilon)^\top - \nabla_\epsilon^2 \nu(\epsilon) \right)\right].\end{aligned}\tag{2.1}$$

A variance-reduced form of these gradient estimators for symmetric distributions (via the control variates method) is

$$\begin{aligned}\nabla_y \mathbb{E}[\ell(y + \epsilon)] &= \mathbb{E}[(\ell(y + \epsilon) - \ell(y)) \nabla_\epsilon \nu(\epsilon)], \\ \nabla_y^2 \mathbb{E}[\ell(y + \epsilon)] &= \mathbb{E}\left[(\ell(y + \epsilon) - \ell(y)) \left( \nabla_\epsilon \nu(\epsilon) \nabla_\epsilon \nu(\epsilon)^\top - \nabla_\epsilon^2 \nu(\epsilon) \right)\right].\end{aligned}\tag{2.2}$$

In the field of reinforcement learning, stochastic smoothing and its derivatives are also known as the score function estimator [38, 39] or REINFORCE [40]. In addition to the above method of control variates, there are also other variance-reduction methods: RELAX [150], which trains a model that predicts the control variate, and REBAR [151], which constructs the control variate based on the difference between the REINFORCE gradient estimator for the relaxed model and the gradient estimator from the reparameterization trick.

**Perturbed Optimizers with Fenchel-Young Losses** Berthet *et al.* [37] build on stochastic smoothing and Fenchel-Young losses [152] to propose perturbed optimizers with Fenchel-Young losses. For this, they use algorithms, like Dijkstra’s shortest-path algorithm, to solve optimization problems of the type  $\max_{w \in \mathcal{C}} \langle y, w \rangle$ , where  $\mathcal{C}$  denotes the feasible set, e.g., the set of valid paths. Berthet *et al.* [37] identify the argmax to be the differential of max, which allows a simplification of stochastic smoothing. By identifying

similarities to Fenchel-Young losses, they find that the gradient of their loss is

$$\nabla_y \ell(y) = \mathbb{E}_\epsilon \left[ \arg \max_{w \in \mathcal{C}} \langle y + \epsilon, w \rangle \right] - w^* \quad (2.3)$$

where  $w^*$  is the ground truth solution of the optimization problem (e.g., the shortest path). This formulation allows optimizing the model without the need for computing the actual value of the loss function.

### 2.1.2 One-Step Linearization of Combinatorial Solvers

The blackbox differentiation method by Vlastelica *et al.* [66] uses a one-step finite differences linearization in the direction of the backpropagated gradient to approximate the first derivative. The method works for estimating the gradients of combinatorial solvers of linear problems. If  $g : \mathbb{R}^m \rightarrow \{0, 1\}^m / \mathbb{R}^m$  is a linear combinatorial solver,  $z = g(y)$ , and  $\nabla_z \ell(y) = \frac{\partial \ell(y)}{\partial g(y)}$  is given, the derivative wrt. the input of the solver may be approximated as

$$\nabla_y \ell = \frac{\partial \ell(y)}{\partial y} \approx \frac{1}{\lambda} \left( g \left( y + \lambda \frac{\partial \ell(y)}{\partial g(y)} \right) - g(y) \right) \quad (2.4)$$

for some  $\lambda > 0$ .

### 2.1.3 Smooth Interpretation

In another line of work, in the field of computer-aided verification, Chaudhuri *et al.* [48, 153] propose a program smoothing method based on randomly perturbing the inputs of a program by a Gaussian distribution. Here, an initial Gaussian perturbation is propagated through program transformations, and a final distribution over perturbed outputs is approximated via a mixture of Gaussians. The smooth function is then optimized using the gradient-free Nelder-Mead optimization method. The main differences to our method are that we perturb all relevant variables (and not the inputs) with logistic distributions and use this for gradient-based optimization.

### 2.1.4 Neural Programs and Differentiable Program Interpreters

Another line of work deals with the problem of inductive program synthesis by relaxing the discrete space of programs into a continuous one [154–157]. Here, neural networks are viewed as continuous relaxations over the space of programs, which can be used to complete partial programs. Their relaxation of the space of programs allows optimization of the source code or a neural representation of the source code. That is, their goal is to make the program interpreter differentiable and not the program itself. Compared to that, for algorithmic supervision, an algorithm is already given and does not need to be learned. Algorithmic supervision supervises an upstream neural network by an algorithm and thus requires the algorithm to be relaxed with respect to its inputs. In summary, our work focuses on making an algorithm differentiable with respect to its input, while neural programs and differentiable program interpreters make a program differentiable with respect to the formulation of the program itself.

## 2.2 A General Method for Relaxing Algorithms

To continuously relax algorithms and, thus, make them differentiable, we relax all values with respect to which we want to differentiate into logistic distributions. We choose the logistic distribution as it provides two distinctive properties that make it especially suited for the task at hand: (1) logistic distributions have heavier tails than normal distributions, which allows for larger probability mass and thus larger gradients when two compared values are further away from each other. (2) the cumulative density function (CDF) of the logistic distribution is the logistic sigmoid function, which can be computed analytically, and its gradient is easily computable. This contrasts the CDF of the normal distribution, which has no closed form and is commonly approximated via a polynomial [158]. However, this should not be seen as a restriction of the proposed method as it is also directly applicable to other distributions, as we discuss in Section 2.2.4.

Specifically, we relax any value  $x$ , for which we want to compute gradients, by perturbing it into a logistic random variable  $\tilde{x} \sim \text{Logistic}(x, 1/\beta)$ , where  $\beta$  is the inverse temperature parameter such that for  $\beta \rightarrow \infty : \tilde{x} = x$ . Based on this, we can relax a discrete conditional statement, e.g., based on the condition  $x < c$  with constant  $c \in \mathbb{R}$ , as follows:

$$[y \text{ if } \tilde{x} < c \text{ else } z] \quad (2.5)$$

$$\equiv \int_{-\infty}^c f_{\log}(t; x, 1/\beta) \cdot y \, dt + \int_c^{\infty} f_{\log}(t; x, 1/\beta) \cdot z \, dt \quad (2.6)$$

$$= F_{\log}(c; x, 1/\beta) \cdot y + (1 - F_{\log}(c; x, 1/\beta)) \cdot z \quad (2.7)$$

$$= \sigma(c - x) \cdot y + (1 - \sigma(c - x)) \cdot z \quad (2.8)$$

where  $\sigma$  is the logistic (sigmoid) function  $\sigma(x) = 1/(1 + e^{-x\beta})$ . In this example, as  $x$  increases, the result smoothly transitions from  $y$  to  $z$ . Thus, the derivative of the result wrt.  $x$  is defined as

$$\begin{aligned} \frac{\partial}{\partial x} [y \text{ if } \tilde{x} < c \text{ else } z] &= \frac{\partial}{\partial x} (y \cdot \sigma(c - x) + z \cdot (1 - \sigma(c - x))) \\ &= (z - y) \cdot \sigma(c - x) \cdot (1 - \sigma(c - x)). \end{aligned} \quad (2.9)$$

Hence, the gradient descent method can influence the condition ( $\tilde{x} < c$ ) to hold if the `if` case reduces the loss, or influence the condition to fail if the `else` case reduces the loss function.

In this example,  $y$  and  $z$  may not only be scalar values but also results of algorithms or parts of an algorithm themselves. This introduces a recursive formalism of relaxed program flow, where parts of an algorithm are combined via a convex combination:

$$[f(s) \text{ if } a < b \text{ else } g(s)] \equiv \sigma(b - a) \cdot f(s) + (1 - \sigma(b - a)) \cdot g(s) \quad (2.10)$$

where  $f$  and  $g$  denote functions, algorithms, or sequences of statements that operate on the set of all variables  $s$  via call-by-value and return the set of all variables  $s$ . The result of this may either overwrite the set of all variables ( $s := [\dots]$ ) or be used in a nested conditional statement.

After introducing `if-else` statements above, we extend the idea to loops, which extends the formalism of relaxed program flow to Turing-completeness.

In fixed loops, i.e., loops with a predefined number of iterations, since there is only a single computation path, no relaxation is necessary, and, thus, fixed loops can be handled by unrolling.

The more complex case is conditional unbounded loops, i.e., **While** loops, which are executed as long as a condition holds. For that, let  $(s_i)_{i \in \mathbb{N}}$  be the sequence of all variables after applying  $i$  times the content of a loop. That is,  $s_0 = s$  for an initial set of all variables  $s$ , and  $s_i = f(s_{i-1})$ , where  $f$  is the content of a loop, i.e., a function, an algorithm, or sequence of statements. Let  $a, b$  denote accessing variables of  $s$ , i.e.,  $s[a], s[b]$ , respectively. By recursively applying the rule for **if-else** statements, we obtain the following rule for unbounded loops:

$$[\text{while } a < b \text{ do } s := f(s)] \quad (2.11)$$

$$\equiv \sum_{i=0}^{\infty} \underbrace{\prod_{j=0}^i (\sigma(b_j - a_j))}_{(a)} \cdot \underbrace{(1 - \sigma(b_{i+1} - a_{i+1}))}_{(b)} \cdot s_i \quad (2.12)$$

Here, (a) is the probability that the  $i$ th iteration is reached and (b) is the probability that there are no more than  $i$  iterations. Together, (a) and (b) is the probability that there are exactly  $i$  iterations weighing the state of all variables after applying  $i$  times  $f$ , which is  $s_i$ . Computationally, we evaluate the infinite series until the probability of execution (a) becomes numerically negligible or a predefined maximum number of iterations has been reached. Again, the result may either overwrite the set of all variables ( $s := \dots$ ) or be used in a nested conditional statement.

**Complexity and Merging of Paths** To compute the exact expectation value of an algorithm under logistic perturbation of its variables, all computation paths would have to be evaluated separately to account for dependencies. However, this would result in an exponential complexity. Therefore, we compute the exact expectation value for nested conditional blocks, but for sequential conditional blocks we merge the computation paths at the end of each block. This allows for a linear complexity in the number of sequential conditional blocks and an exponential complexity only in the largest depth of nested conditional blocks. Note that the number of sequential conditional blocks is usually much larger than the depth of conditional blocks, e.g., hundreds or thousands of sequential blocks and a maximum depth of just 2 – 5 in our experiments. An example of a dependency is the expression  $[a := (f(x) \text{ if } i < 0 \text{ else } g(x)); b := (0 \text{ if } a < 0 \text{ else } a^2)]$ , which contains a dependence between the two sequential conditional blocks, which introduces the error in our approximation. In general, our formalism also supports modeling dependencies between sequential conditional blocks; however, practically, it might become intractable for entire algorithms. Also, it is possible to consider relevant dependencies explicitly if an algorithm relies on specific dependencies.

**Perturbations of Variables vs. Perturbations of Inputs** Note that modeling the perturbation of variables is different from modeling the perturbation of the inputs. A condition, where the difference becomes clear, is, e.g.,  $(\tilde{x} < x)$ . When modeling input perturbations, the condition would have a strong implicit conditional dependency and evaluate to a 0% probability. However, in this chapter, we do *not* model perturbations of the inputs but

instead model perturbations of variables each time they are accessed such that accessing a variable twice is independently identically distributed (iid). Therefore,  $(\tilde{x} < x)$  evaluates to a 50% probability. To minimize the approximation error of the relaxation, only those variables should be relaxed for which a gradient is required.

### 2.2.1 Relaxed Comparators

So far, we have only considered the comparator  $<$ .  $>$  follows by swapping the arguments:

$$\mathbb{P}[a < b] \equiv \sigma(b - a) \quad \mathbb{P}[a > b] \equiv \sigma(a - b) \quad (2.13)$$

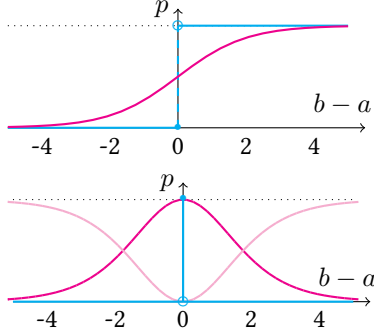


**Figure 2.1:** Top: hard decision boundary (cyan) and probability under logistic perturbation (magenta). Bottom: hard equality (cyan), relaxed equality (magenta), and relaxed inequality (light magenta).

**Relaxed Equality** For the equality operator  $=$ , we consider two distributions  $\tilde{a} \sim \text{Logistic}(a, 1/\beta)$  and  $\tilde{b} \sim \text{Logistic}(b, 1/\beta)$ , which we want to check for similarity / equality. Given value  $a$ , we compute the likelihood that  $a$  is a sample from  $\tilde{b}$  rather than  $\tilde{a}$ . If  $a$  is equally likely to be drawn from  $\tilde{a}$  and  $\tilde{b}$ ,  $\tilde{a}$  and  $\tilde{b}$  are equal. If  $a$  is unlikely to be drawn from  $\tilde{b}$ ,  $\tilde{a}$  and  $\tilde{b}$  are unequal. To compute whether it is equally likely for  $a$  to be drawn from  $\tilde{a}$  and  $\tilde{b}$ , we take the ratio between the likelihood that  $a$  is from  $\tilde{b}$  ( $f_{\log}(a; b, 1/\beta)$ ) and the likelihood that  $a$  is from  $\tilde{a}$  ( $f_{\log}(a; a, 1/\beta)$ ):

$$\mathbb{P}[a = b] \equiv \frac{f_{\log}(a; b, 1/\beta)}{f_{\log}(a; a, 1/\beta)} = \frac{f_{\log}(b; a, 1/\beta)}{f_{\log}(b; b, 1/\beta)} = \text{sech}^2\left(\frac{b - a}{2/\beta}\right) \quad (2.14)$$

These relaxed comparators are displayed in Figure 2.1. An alternative derivation for (2.14) is the normalized conjunction of  $\neg(a < b)$  and  $\neg(a > b)$ .

**Relaxed Logics** To compute probabilities of conjunction (i.e., and) or of disjunction (i.e., or), we use the product or probabilistic sum, respectively. This corresponds to an intersection / union of independent events. Alternatives to and extensions of this are discussed throughout this thesis, and a summary of alternatives can be found in Supplementary Material B.

**Relaxed Maximum** To compare more than two elements and relax the arg max function, we use the multinomial logistic distribution, which is also known as the softmax distribution.

$$\mathbb{P}(i = \arg \max_j X_j) = \frac{e^{X_i \beta}}{\sum_j e^{X_j \beta}} \quad (2.15)$$

To relax the max operation, we use the product of arg max / softmax and the respective vector. The softmax distribution corresponds to an arg max under perturbation with a Gumbel distribution. Alternatives to this are discussed throughout this thesis, and an extensive discussion of min / max between two elements can be found in Chapter 3 and Supplementary Material C.

**Comparing Categorical Variables** To compare a categorical probability distribution  $X \in [0, 1]^n$  with a categorical probability distribution  $Y$ , we consider two scenarios: If  $Y \in \{0, 1\}^n$ , i.e.,  $Y$  is one-hot, we can use the inner product of  $X$  and  $Y$  to obtain their joint probability. However, if

$Y \notin \{0, 1\}^n$ , i.e.,  $Y$  is not one-hot, even if  $X = Y$  the inner product can not be 1, but a probability of 1 would be desirable if  $X = Y$ . Therefore, we ( $L_2$ ) normalize  $X$  and  $Y$  before taking the inner product, which corresponds to the cosine similarity. An example of the application of comparing categorical probability distributions is shown in the Levenshtein distance experiments in Section 2.6.

### 2.2.2 Relaxed Indexing

As vectors, arrays, and tensors are essential for algorithms and machine learning, we also formalize relaxed indexing. For this, we introduce real-valued indexing and categorical indexing.

**Real-Valued Indexing** In a relaxed algorithm, indices may be drawn from the set of real numbers as they may be a convex combination of computations or a real-valued input. This poses a challenge since it requires interpolating between multiple values. The direct approach would be grid sampling with bilinear or bicubic interpolation to interpolate values. For example, Neural Turing Machines use linear interpolation for real-valued indexing [159]. However, in bilinear or bicubic interpolation, relationships exceeding the direct (or next) neighbors in the array are not modeled, and they also do not model logistic perturbations. Therefore, we index an  $n$ -dimensional tensor  $\mathbf{A}$  with values  $i \in \mathbb{R}^n$  via logistic perturbation by applying a convolution with a logistic filter  $g$  and obtain the result as  $(g * \mathbf{A})(i)$ . The convolution of a tensor  $\mathbf{A}$  with a logistic filter  $g$  (not to be confused with discrete-discrete convolution in neural networks) yields a function that is evaluated at point  $i \in \mathbb{R}^n$ . We choose the logistic filter over bilinear and bicubic filters because we model logistic perturbation and additionally because bilinear and bicubic filters only have compact support, whereas the logistic filter provides infinite support. This allows modeling relationships beyond the next neighbors and is more flexible as the inverse temperature  $\beta$  can be tuned for the respective application. For stability, we normalize the coefficients used to interpolate the respective indexed values such that they sum up to one: instead of computing  $(g * \mathbf{A})(i) = \sum_j g(j - i) \mathbf{A}_j$ , we compute  $\sum_j g(j - i) \mathbf{A}_j / (\sum_j g(j - i))$  where  $j$  are all valid indices for the tensor  $\mathbf{A}$ . To prevent optimization algorithms from exploiting aliasing effects, we divide the coefficients by their sum only in the forward pass, but ignore this during the backward pass (computation of the gradient). Real-valued indexing is displayed in Figures 2.2 and 2.3, comparing it to hard indexing for 1D and 2D arrays.

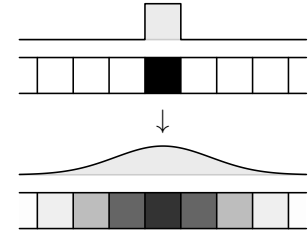


**Figure 2.2:** 1D real-valued indexing: the gray-value represents the extent to which each value is used for indexing.

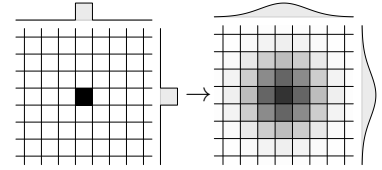


**Figure 2.3:** 2D real-valued indexing: the gray-value represents the extent to which each value is used for indexing.

**Relaxed Categorical Indexing** If a categorical probability distribution over indices is given, e.g., computed by  $\text{argmax}$  or its relaxation  $\text{softmax}$ , categorical indexing can be used. Here, the marginal categorical distribution is used as weights for indexing a tensor.

Note that real-valued indexing assumes that the indexed array follows a semantic order such as time series, an image, or the position in a grid. If, in contrast, the array contains categorical information such as the nodes of graphs, values should be indexed with categorical indexing as their neighborhood is arbitrary.

### 2.2.3 Complexity and Runtime Considerations

In terms of runtime, one has to note that runtime optimized algorithms (e.g., Dijkstra) usually do not improve the runtime for the relaxation because for the proposed continuous relaxations all cases in an algorithm have to be executed. Thus, an additional condition to reduce the computational cost does not improve runtime because both (all) cases are executed. Instead, it becomes beneficial if an algorithm solves a problem in a rather fixed execution order. On the other hand, optimizing an algorithm with respect to runtime leads to interpolations between the fastest execution paths. This optimization cannot improve the gradients but rather degrades them as it is an additional approximation and produces gradients with respect to runtime heuristics. For example, when relaxing the Dijkstra shortest-path algorithm, there is an interpolation between different orders of visiting nodes, which is the heuristic that makes Dijkstra fast. However, if we have to follow all paths anyway (to compute the relaxation), it can lead to a combinatorial explosion. In addition, by interpolating between those alternative orders, a large amount of uncertainty is introduced, and the gradients will also depend on the orders of visiting nodes, both of which are undesirable. Further, algorithms with rather strict execution can be executed in parallel on GPUs such that they can be faster than runtime-optimized sequential algorithms on CPUs. Therefore, we prefer simple algorithms with a largely fixed execution structure and without runtime optimizations from both a runtime and gradient quality perspective.

### 2.2.4 Alternative Distributions

Above, we focused on perturbing by logistic distributions; however, the formulation is not limited to this setting. Specifically, in Chapters 3 and 5, we cover a variety of alternative distributions. For a given problem setting, the optimal choice of distribution depends on the specific used algorithm as well as the data, which we specifically discuss for the setting of differentiable renderers in Chapter 5. A collection of possible distributions is discussed in Supplementary Material A.

### 2.2.5 The AlgoVision Library

We published the proposed method in the form of the  **AlgoVision** library to facilitate easy application and further research in this direction. The library is based on Python and PyTorch [26]. It is publicly available under <https://github.com/Felix-Petersen/algovision>. The accompanying documentation is published under <https://felix-petersen.github.io/algovision-docs/> and also archived in the Internet Archive. Examples for algorithms can be found in Sections 2.3 and 2.6.

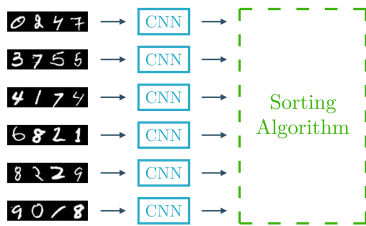


Figure 2.4: Illustration of Sorting Supervision

## 2.3 Sorting Supervision

In this section, we cover the first application of algorithmic supervision, namely sorting and ranking supervision. Here, the idea is to train a model based on relative ordering information instead of absolute values or labels.

This can be done using a differentiable sorting algorithm that enforces the model to return values corresponding to the ground truth ordering, as illustrated in Figure 2.4.

**Related Work** The four-digit MNIST sorting benchmark task for sorting supervision has been first proposed by Grover *et al.* [58]. Grover *et al.* [58] address the task by relaxing the permutation matrices to double stochastic matrices. Cuturi *et al.* [56] pick up on this benchmark and propose a differentiable proxy by approximating the sorting problem with a regularizing optimal transport algorithm. These methods are discussed in greater detail in the differentiable sorting-focused Chapter 3.

In the sorting supervision four-digit MNIST experiment, a set of four-digit numbers based on concatenated MNIST digits [160] is given, and the task is to find an order-preserving mapping from the images to scalars. A CNN learns to predict a scalar for each of  $n$  four-digit numbers such that their order is preserved among the predicted scalars. For this, only sorting supervision in the form of the ground truth order of input images is given, while their absolute values remain unsupervised. This follows the protocol of Grover *et al.* [58] and Cuturi *et al.* [56]. An example of a concatenated MNIST image is .

Using our method, we relax the well established stable sorting algorithm Bubble sort [161], which works by iteratively going through a list and swapping any two adjacent elements if they are not in the correct order until there are no more swap operations in one iteration. In the following, we display the annotated AlgoVision code of the relaxed algorithm. The library provides a set of modules, a subset of which is imported at the beginning. These modules can then be used to define the algorithm, which is then automatically relaxed and supports backpropagation, parallelization, and CPU as well as GPU execution.

```
from algovision import (
    Algorithm, Input, Output, Var, VarInt,                                # core
    Let, LetInt, Print,                                                  # instructions
    Eq, NEq, LT, LEq, GT, GEq, CatProbEq, CosineSimilarity, IsTrue, IsFalse, # conditions
    If, While, For,                                                      # control_structures
    Min, ArgMin, Max, ArgMax                                             # functions
)
import torch

bubble_sort = Algorithm(
    # Define the variables the input corresponds to
    Input('array'),
    # Declare and initialize all differentiable variables
    Var('a', torch.tensor(0.)),
    Var('b', torch.tensor(0.)),
    Var('swapped', torch.tensor(1.)),
    Var('loss', torch.tensor(0.)),
    # Declare and initialize a hard integer variable (VarInt) for the control flow.
    # It can be defined in terms of a lambda expression. The required variables
    # are automatically inferred from the signature of the lambda expression.
    VarInt('n', lambda array: array.shape[1] - 1),
```

```

# Start a relaxed While loop:
While(IsTrue('swapped'),
  # Set `swapped` to 0 / False
  Let('swapped', 0),
  # Start an unrolled For loop. Corresponds to `for i in range(n):`
  For('i', 'n',
    # Set `a` to the `i`th element of `array`
    Let('a', 'array', ['i']),
    # Using an inplace lambda expression, we can include computations
    # based on variables to obtain the element at position i+1.
    Let('b', 'array', [lambda i: i+1]),
    # An If-Else statement with the condition a > b
    If(GT('a', 'b'),
      if_true=[
        # Set the i+1 th element of array to a
        Let('array', [lambda i: i + 1], 'a'),
        # Set the i th element of array to b
        Let('array', ['i'], 'b'),
        # Set swapped to 1 / True
        Let('swapped', 1.),
        # Increment the loss by 1 using a lambda expression
        Let('loss', lambda loss: loss + 1.),
      ],
    ),
  ),
  # Decrement the hard integer variable n by 1
  LetInt('n', lambda n: n-1),
),
# Define what the algorithm should return
Output('array'),
Output('loss'),
# Set the inverse temperature beta
beta=5,
)

```

We include a variable (swapped) that keeps track of whether the input sequence is in the correct order by setting it to true if a swap operation occurs. Due to the relaxation, this variable is a floating-point number between 0 and 1 corresponding to the probability that the predictions are sorted correctly (under perturbation of variables). We use this probability as the loss function. This variable equals the probability that no swap operation was necessary, and thus  $\mathcal{L} = 1 - \prod_{p \in P} (1 - p)$  for probabilities  $p$  of each potential swap  $p \in P$ . By that, the training objective enforces the input sequence to be sorted and, thus, enforces that the scores predicted by the neural network correspond to the supervised partial order. In fact, the number of swaps in bubble sort corresponds to the Kendall's  $\tau$  coefficient, which indicates to which degree a sequence is sorted. Note that, e.g., QuickSort does not have this property.

We emphasize that the task of the trained neural network is *not* to sort a sequence but instead to predict a score for each element such that the ordering / ranking corresponds to the supervised ordering / ranking. While the relaxed algorithm can sort the inputs correctly, at evaluation time, following the setup of [58] and [56] we use an argsort method to test whether the

| Method                 | $n = 3$              | $n = 5$              | $n = 7$              |
|------------------------|----------------------|----------------------|----------------------|
| Stoch. NeuralSort [58] | 0.920 (0.946)        | 0.790 (0.907)        | 0.636 (0.873)        |
| Det. NeuralSort [58]   | 0.919 (0.945)        | 0.777 (0.901)        | 0.610 (0.862)        |
| Optimal Transport [56] | 0.928 (0.950)        | 0.811 (0.917)        | 0.656 (0.882)        |
| Relaxed Bubble Sort    | <b>0.944 (0.961)</b> | <b>0.842 (0.930)</b> | <b>0.707 (0.898)</b> |

**Table 2.1:** Results for the 4-digit MNIST sorting task, averaged over 10 runs. Baselines as reported by Cuturi *et al.* [56]. Trained and evaluated on sets of  $n$  elements, the displayed metrics are exact matches (and element-wise correct ranks).

outputs produced by the neural network are in accordance with the ground truth partial order. We use the same network architecture as [58] and [56]. Here, we only optimize the inverse temperature for  $n = 5$ , resulting in  $\beta = 8$ , and fix this for all other  $n$ . For training, we use the Adam optimizer [34] with a learning rate of  $10^{-4}$  for a number of iterations between  $1.5 \cdot 10^5$  and  $1.5 \cdot 10^6$ . For comparability to Grover *et al.* [58] and Cuturi *et al.* [56], we use the same network architecture. That is, two convolutional layers with a kernel size of  $5 \times 5$ , 32 and 64 channels respectively, each followed by a ReLU and MaxPool layer; after flattening, it is followed by a linear layer with a size of 64, a ReLU layer, and a linear output layer mapping to a scalar.

We evaluate our method against state-of-the-art hand-crafted relaxations of the sorting operation using the same network architecture and evaluation metrics as Grover *et al.* [58] and Cuturi *et al.* [56]. As displayed in Table 2.1, our general formulation outperforms state-of-the-art hand-crafted relaxations of the sorting operation for sorting supervision. In Chapter 3, we show that differentiable sorting networks improve upon this result.

## 2.4 Shortest-Path Supervision

The setting of shortest-path supervision is illustrated in Figure 2.5. Here, an image is given, a cost embedding is predicted by a neural network, and the algorithm yields a differentiable shortest path. After comparing the predicted shortest path to a ground truth shortest path and backpropagating the error through the algorithm, we can train the neural network.



**Figure 2.5:** Illustration of Shortest-Path Supervision

For the 2D Warcraft terrains shortest-path supervision benchmark, we follow the setup by Vlastelica *et al.* [66] and Berthet *et al.* [37] and use the data set of 10 000 patches of Warcraft terrains of size  $96 \times 96$  representing terrain grids of size  $12 \times 12$ . Given an image of a terrain (e.g., Figure 2.6 first), the goal is to predict the shortest path from the top-left to the bottom-right (e.g., Figure 2.6 third) according to a hidden cost matrix (e.g., Figure 2.6 second). For this,  $12 \times 12$  binary matrices of the shortest path are supervised, while the hidden cost matrix is used only to determine the shortest path. Integrating a shortest-path algorithm into a neural architecture lets the neural network produce a cost embedding of the terrain, which the shortest-path algorithm uses for predicting the shortest path.

**Related Work** Vlastelica *et al.* [66] tackle this task by finding a linearization of the Dijkstra algorithm [162], which they can differentiate. Berthet *et*

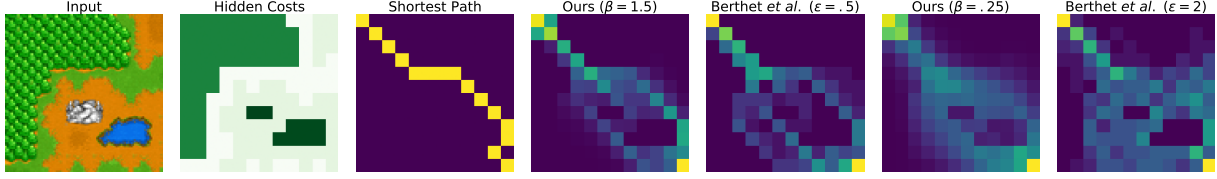


**Figure 2.6:** (From left to right.) Example input image of the Warcraft terrain data set with the hidden cost matrix and the resulting shortest path. Shortest paths relaxed with the proposed method for  $\beta \in \{1.5, 0.25\}$ , which correspond to the perturbed paths by Berthet *et al.* [37] with  $\epsilon \in \{0.5, 2.0\}$ .

**Table 2.2:** Results for the Warcraft shortest-path task using shortest-path supervision, averaged over 10 runs. Reported is the exact match accuracy (EM). For additional and more extensive results, see Chapter 7.

| Method          | EM           |
|-----------------|--------------|
| ResNet Baseline | 40.2%        |
| Black-Box Loss  | 86.6%        |
| Perturbed Opt.  | 80.6%        |
| AlgoVision      | <b>95.8%</b> |

*al.* [37] take up this problem and produce gradient estimates for the Dijkstra algorithm by stochastically perturbing the inputs to the shortest-path optimization problem. In their works, Vlastelica *et al.* [66] and Berthet *et al.* [37] show that integrating and differentiating a shortest-path algorithm can improve the results by allowing the neural network to predict a cost matrix from which the shortest path can be computed via the algorithm. This performs significantly better than a ResNet baseline, where the shortest paths have to be predicted by the neural network alone (see Table 2.2).

For this task, we relax the Bellman-Ford algorithm [163] with 8-neighborhood, node weights, and path reconstruction. For the loss between the supervised shortest paths and the paths produced by the relaxed Bellman-Ford algorithm, we use the  $\ell^2$  loss. To illustrate the shortest paths created by our method and to compare them to those created through the Perturbed Optimizers by Berthet *et al.* [37], we display examples of back-traced shortest paths for two inverse temperatures in Figure 2.6 (center to right).

We use the same ResNet network architecture as Vlastelica *et al.* [66] and Berthet *et al.* [37]. That is, the first five layers of ResNet18 followed by an adaptive max pooling to the size of  $12 \times 12$  and an averaging over all features. As in previous works, we train for 50 epochs with batch size 70 and decay the learning rate by a factor of 0.1 after 60% as well as after 80% of training.

As shown in Table 2.2, our relaxation outperforms all baselines.

## 2.5 Silhouette Supervision

The silhouette supervision experiment in this chapter is to demonstrate the flexibility and applicability of the proposed approach even to complex tasks like silhouette supervision. In Chapter 5, we discuss a natively implemented differentiable renderer, a special case of which is equivalent to the differentiable renderer proposed here.

Reconstructing a 3D model from a single 2D image is an important task in computer vision. This task is frequently solved using silhouette supervision as illustrated in Figure 2.7. Here, a single image is processed by a neural network, which returns a 3D mesh; this mesh is then rendered back into the image space and the predicted image can be compared to the silhouette of the input, facilitating training.

**Figure 2.7:** Illustration of Silhouette Supervision



**Related Work** Recent works [46, 63, 105] have benchmarked their differentiable renderers on a data set of 13 object classes from ShapeNet [164] that have been rendered from 24 azimuths at a resolution of  $64 \times 64$  [63]. Kato *et*

*al.* [63] propose a renderer where surrogate gradients of rasterization are approximated to perform 3D mesh reconstruction via silhouette supervision as well as 3D style transfer. Liu *et al.* [46] propose a differentiable renderer without surrogate gradients by using a differentiable aggregating process and apply it to 3D mesh reconstruction as well as pose / shape optimization. More details can be found in the differentiable rendering-focused Chapter 5.

For training, the silhouette of the predicted mesh is rendered from two view-points by a differentiable renderer and the intersection-over-union between the rendered and predicted meshes is used as a training objective to update the neural network [46, 63]. For training, we also use the same neural network architecture as Kato *et al.* [63] as well as Liu *et al.* [46]. While some differentiable renderers also render RGB images, in these experiments, only the silhouette is used for supervision. Specifically, public implementations of [46, 63] only use silhouette supervision.

For this task, we relax two silhouette rendering algorithms. The algorithms rasterize a 3D mesh as follows: For each pixel and for each triangle of the mesh, if a pixel lies inside a triangle, the value of the pixel in the output image is set to 1. The condition of whether a pixel lies inside a triangle is checked in two alternative fashions: (1) by three nested `if` conditions that check on which side of each edge the pixel lies. (2) by checking whether the directed euclidean distance between a pixel and a triangle is positive. Note that, by relaxing these algorithms using our framework, we obtain differentiable renderers equivalent to Pix2Vex [11] for (1) and a specific instance of GenDR for (2). A greater discussion of relations between differentiable renderers can be found in Chapter 5. Examples of relaxed silhouette renderings and an example image from the data set are displayed in Figure 2.8.

As the simple silhouette renderer does not have any optimizations, such as discarding pixels that are far away from a triangle or triangles that are occluded by others, it is not very efficient. Thus, due to limited resources, we are only able to train with a maximum batch size of 2 while previous works used a batch size of 64. Therefore, we reproduce the recent best performing work by Liu *et al.* [46] on a batch size of only 2 to allow for a fair comparison. For directed Euclidean distance, we use an inverse temperature of  $\beta = 2\,000$ ; for three edges,  $\beta = 10\,000$ . For comparability to Liu *et al.* [46], we use the same network architecture. That is, three convolutional layers with a kernel size of  $5 \times 5$ , 64, 128, and 256 channels respectively, each followed by a ReLU; after flattening, this is followed by 6 ReLU-activated fully connected layers with the following output dimensions: 1024, 1024, 512, 1024, 1024,  $642 \times 3$ . The  $642 \times 3$  elements are interpreted as three-dimensional vectors that displace the vertices of a sphere with 642 vertices. We train the Three Edges

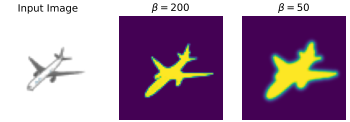


**Figure 2.8:** An input image from the data set (left). Silhouette of a prediction rendered with directed Euclidean distance approach for two different inverse temperatures  $\beta = 200$  (center) and  $\beta = 50$  (right).

**Table 2.3:** Single-view 3D reconstruction results using silhouette supervision. Reported is the 3D IoU.

| Method                              | Airplane      | Bench         | Dresser       | Car           | Chair         | Display       | Lamp          | Speaker       | Rifle         | Sofa          | Table         | Phone         | Vessel        | Mean          |
|-------------------------------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|
| <b>With a batch size of 64</b>      |               |               |               |               |               |               |               |               |               |               |               |               |               |               |
| Yan <i>et al.</i> [105] (retrieval) | 0.5564        | 0.4875        | 0.5713        | 0.6519        | 0.3512        | 0.3958        | 0.2905        | 0.4600        | 0.5133        | 0.5314        | 0.3097        | 0.6696        | 0.4078        | 0.4766        |
| Yan <i>et al.</i> [105] (voxel)     | 0.5556        | 0.4924        | 0.6823        | 0.7123        | 0.4494        | 0.5395        | 0.4223        | 0.5868        | 0.5987        | 0.6221        | 0.4938        | 0.7504        | 0.5507        | 0.5736        |
| Kato <i>et al.</i> [63] (NMR)       | 0.6172        | 0.4998        | 0.7143        | 0.7095        | 0.4990        | 0.5831        | 0.4126        | 0.6536        | 0.6322        | 0.6735        | 0.4829        | 0.7777        | 0.5645        | 0.6015        |
| Liu <i>et al.</i> [46] (SoftRas)    | 0.6419        | 0.5080        | 0.7116        | 0.7697        | 0.5270        | 0.6156        | 0.4628        | 0.6654        | 0.6811        | 0.6878        | 0.4487        | 0.7895        | 0.5953        | 0.6234        |
| <b>With a batch size of 2</b>       |               |               |               |               |               |               |               |               |               |               |               |               |               |               |
| Liu <i>et al.</i> [46] (SoftRas)    | <b>0.5741</b> | <b>0.3746</b> | 0.6373        | <b>0.6939</b> | <b>0.4220</b> | 0.5168        | 0.4001        | 0.6068        | <b>0.6026</b> | <b>0.5922</b> | 0.3712        | <b>0.7464</b> | <b>0.5534</b> | <b>0.5455</b> |
| Relaxed Sil. via Three Edges        | 0.5418        | 0.3667        | <b>0.6626</b> | 0.6546        | 0.3899        | 0.5229        | 0.4105        | <b>0.6232</b> | 0.5497        | 0.5639        | 0.3580        | 0.6609        | 0.5279        | 0.5256        |
| Relaxed Sil. via Euclid. Dist       | 0.5399        | 0.3698        | 0.6503        | 0.6524        | 0.4044        | <b>0.5261</b> | <b>0.4247</b> | 0.6225        | 0.5723        | 0.5643        | <b>0.3829</b> | 0.7265        | 0.5180        | 0.5349        |



```

levenshtein = Algorithm(
    # Define the variables the input corresponds to.
    Input('array_s'),
    Input('array_t'),
    # Declare and initialize all differentiable variables.
    Var('d', lambda array_s, array_t:
        torch.zeros(array_s.shape[1] + 1, array_t.shape[1] + 1)
    ),
    Var('subs_cost', torch.tensor(0.)),
    Var('return_d', torch.tensor(0.)),
    # Initialize the borders of the cost matrix.
    For('i', lambda d: d.shape[1]-1,
        Let('d', [lambda i: i+1, 0], lambda i: i+1)
    ),
    For('j', lambda d: d.shape[2]-1,
        Let('d', [0, lambda j: j+1], lambda j: j+1)
    ),
    # Run the dynamic programming algorithm.
    For('i', lambda d: d.shape[1]-1,
        For('j', lambda d: d.shape[2]-1,
            # Differentiable check for equality of two categorical embeddings.
            If(CosineSimilarity(lambda array_s, i: array_s[:, i],
                                lambda array_t, j: array_t[:, j]),
                if_true=Let('subs_cost', 0),
                if_false=Let('subs_cost', 1),
            ),
            # Correspond to
            # `d[i+1, j+1] = min(d[i, j+1]+1, d[i+1, j]+1, d[i+1, j+1]+subs_cost`.
            Let('d', [lambda i: i+1, lambda j: j+1],
                lambda d, i, j, subs_cost: Min(beta=5)(d[:, i, j+1]+1,
                                                         d[:, i+1, j]+1,
                                                         d[:, i, j]+subs_cost)
            )
        ),
    ),
    # Return the distance and the cost matrix.
    Let('return_d', 'd', [lambda d: d.shape[1]-1, lambda d: d.shape[2]-1]),
    Output('return_d'),
    Output('d'),
    beta=5,
)

```

An example Levenshtein distance matrix and its relaxation are displayed in Figure 2.10.

For learning, pairs of strings of images of 32 handwritten characters  $a, b$  as well as the ground truth Levenshtein distance  $LD(y_a, y_b)$  are given. We sample pairs of strings  $a, b$  from an alphabet of 2 or 4 characters. For sampling the second string given the first one, we uniformly choose between two and four insertion and deletion operations. Thus, the average editing distance for strings, which use two different characters, is 4.25 and for four characters is 5. We process each letter, using a CNN, which returns a categorical distribution over letters, which is then fed to the algorithm. An example of a pair of strings based on  $\{A, C, G, T\}$  is

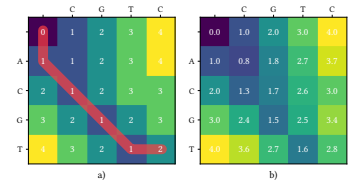


**Figure 2.10:** Levenshtein distance matrix. *Left:* hard matrix and alignment path. *Right:* relaxed matrix with inverse temperature  $\beta = 1.5$ .

**Table 2.4:** EMNIST classification results with Levenshtein distance supervision averaged over 10 runs.

| Method     |            | AB          | BC          | CD          | DE          | EF          | IL          | OX          | ACGT        | OSXL        |
|------------|------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|
| Baseline   | Top-1 acc. | .616        | .651        | .768        | .739        | .701        | .550        | .893        | .403        | .448        |
|            | F1-score   | .581        | .629        | .759        | .711        | .674        | .490        | .890        | .336        | .384        |
| Relaxed LD | Top-1 acc. | <b>.671</b> | <b>.807</b> | <b>.816</b> | <b>.833</b> | <b>.847</b> | <b>.570</b> | <b>.960</b> | <b>.437</b> | <b>.487</b> |
|            | F1-score   | <b>.666</b> | <b>.805</b> | <b>.815</b> | <b>.831</b> | <b>.845</b> | <b>.539</b> | <b>.960</b> | <b>.367</b> | <b>.404</b> |

**6TACC69a2a2CG+49e4a7TQeC%99CLTAAQ** and  
**8T0e6CG6aG2C\$A6CAA+7TACC9a9CLtQA**.

Our training objective is minimizing the  $\ell^2$  loss between the predicted distance and the ground truth distance:

$$\mathcal{L} = \left\| \text{LD} \left( (\text{CNN}(a_i))_{i \in \{1..32\}}, (\text{CNN}(b_i))_{i \in \{1..32\}} \right) - \text{LD}(\mathbf{y}_a, \mathbf{y}_b) \right\|_2. \quad (2.17)$$

For training, we use an inverse temperature of  $\beta = 9$  and Adam ( $\eta = 10^{-4}$ ) for 128 – 512 iterations. The CNN consists of two convolutional layers with a kernel size of 5 and hidden sizes of 32 and 64, each followed by a ReLU, and a max-pooling layer. The convolutional layers are followed by two fully connected layers with a hidden size of 64 and a ReLU activation.

For evaluation, we use Hungarian matching with Top-1 accuracy as well as the F1-score. We compare it against a baseline, which uses the  $\ell_1$  distance between encodings instead of the editing distance:

$$\mathcal{L} = \left\| \left\| (\text{CNN}(a_i))_{i \in \{1..32\}} - (\text{CNN}(b_i))_{i \in \{1..32\}} \right\|_1 - \text{LD}(\mathbf{y}_a, \mathbf{y}_b) \right\|_2. \quad (2.18)$$

Table 2.4 shows that our method consistently outperforms the baseline on both metrics in all cases. The character combinations AB, BC, CD, DE, and EF are a canonical choice for random combinations. The characters IL are the hardest combination of two letters as they even get frequently confused by supervised neural networks [165] and can also be indistinguishable for humans. The characters OX represent the easiest case as supervised classifiers can perfectly distinguish them on the test dataset [165]. For two letter combinations, we achieve Top-1 accuracies between 57% (IL) and 96% (OX). Even for four letter combinations (ACGT and OSXL), we achieve Top-1 accuracies of up to 48.7%. Note that, as we use strings of length 32, in the Levenshtein algorithm, more than 1 000 statements are relaxed.

## Conclusion

In this chapter, we covered and proposed general approaches for continuous relaxations of algorithms that allow their integration into end-to-end trainable neural network architectures. For that, we use convex combinations of execution paths of algorithms that are parameterized by smooth functions. We found that the proposed general framework can compete with continuous relaxations of specific algorithms as well as gradient estimation methods on a variety of algorithmic supervision tasks. While this chapter covered a general approach, in the following chapters, we focus on relaxing specific algorithms.

# Differentiable Sorting and Ranking

# 3

After presenting general methods for relaxing algorithms, in this chapter, we focus on differentiable sorting and ranking, which allows us to explore it in much greater detail.

Sorting and ranking, i.e., the ability to score elements by their relevance, is essential in various applications. It can be used for choosing the best results to display by a search engine or organizing data in memory, among many others. As sorting a sequence of values requires finding the respective ranking order, we use the terms “sorting” and “ranking” interchangeably.

Recently, the idea of end-to-end training of neural networks with sorting and ranking supervision by a continuous relaxation of the sorting and ranking functions has been presented by Grover *et al.* [58]. The idea of ordering supervision is that the ground truth order of some samples is known while their absolute values remain unsupervised. This is achieved by integrating a sorting algorithm in the neural architecture. As for training with a sorting algorithm in the architecture, the error needs to be propagated in a meaningful way back to the neural network, it is necessary to use a differentiable sorting function. Several such differentiable relaxations of the sorting and ranking functions have been introduced, e.g., by Adams *et al.* [88], Grover *et al.* [58], Cuturi *et al.* [56], and Blondel *et al.* [64]. These methods enable training a neural network based on ordering and ranking information instead of absolute ground truth values.

Starting in the 1950s, sorting networks have been presented to address the sorting task [167]. Sorting networks are sorting algorithms with a fixed execution structure, which makes them suitable for hardware implementations, e.g., as part of circuit designs. They are oblivious to the input, i.e., their execution structure is independent of the data to be sorted. They allow for fast hardware-implementation, e.g., in application-specific integrated circuits (ASICs), as well as on highly parallelized general-purpose hardware like GPUs. As such hardware implementations are significantly faster than conventional multipurpose hardware, they are of interest for sorting in high-performance computing applications [168]. This motivated the optimization of sorting networks toward faster networks with fewer layers, which is a still-standing problem [169]. Note that, although the name is similar, sorting networks are *not* neural networks that sort.

In this chapter, we propose to combine traditional sorting networks and differentiable sorting functions by presenting smooth differentiable sorting networks. Later in this chapter, we focus on analyzing differentiable sorting functions [56, 58, 64] and demonstrate how monotonicity improves differentiable sorting networks.

Sorting networks are a family of sorting algorithms that consist of two basic components: so-called “wires” (or “lanes”) carrying values and conditional swap operations that connect pairs of wires [167]. An example of such a sorting network is shown in the center of Figure 3.1. The conditional swap operations swap the values carried by these wires if they are not in the desired order. As sorting networks are data-oblivious, i.e., the program

|     |                                                                                 |    |
|-----|---------------------------------------------------------------------------------|----|
| 3.1 | Related Work . . . . .                                                          | 29 |
| 3.2 | Sorting Networks . . . . .                                                      | 32 |
| 3.3 | Differentiable Sorting<br>Networks . . . . .                                    | 34 |
| 3.4 | Monotonicity and Error-<br>Boundedness of Differen-<br>tiable Sorting Operators | 43 |
| 3.5 | Experiments . . . . .                                                           | 43 |

flow of the algorithm is independent of the input data, they are especially suitable for continuous relaxation. Sorting networks are conventionally non-differentiable as they use min and max operators for conditionally swapping elements. Thus, we relax these operators by perturbation with a probability distribution, e.g., with the logistic distribution.

One problem that arises in this context is that using a logistic sigmoid function does not preserve the monotonicity of the relaxed sorting operation, which can cause gradients with the wrong sign. In Section 3.3.3, we present a family of sigmoid functions that preserve the monotonicity of differentiable sorting networks. These include the CDF of the Cauchy distribution, as well as a function that minimizes the error-bound and thus induces the smallest possible approximation error. For all sigmoid functions, we prove and visualize the respective properties and validate their advantages empirically. In fact, by making the sorting function monotonic, it also becomes quasiconvex, which has been shown to produce favorable convergence rates [170]. In Figure 3.5, we demonstrate monotonicity for different choices of sigmoid functions. As can be seen in Figure 3.7, existing differentiable sorting operators are either non-monotonic or have an unbounded error. We show that sigmoid functions with specific characteristics produce monotonic and error-bounded differentiable sorting networks. We provide theoretical guarantees for these functions and also give the monotonic function that minimizes the approximation error, and demonstrate that the proposed functions improve empirical performance.

To validate the proposed idea and to show its generalization, we evaluate it for two sorting network architectures, the odd-even as well as the bitonic sorting network. The idea of odd-even sort is to iteratively compare adjacent elements and swap pairs that are in the wrong order. The method alternately compares all elements at odd and even indices with their successors. To make sure that the smallest (or greatest) element will be propagated to its final position for any possible input of length  $n$ , we need  $n$  exchange layers. An odd-even network is displayed in Figure 3.1 (center). Odd-even networks can be seen as the most generic architectures, and are mainly suitable for small input sets as their number of layers directly depends on the number of elements to be sorted.

Bitonic sorting networks [171] use bitonic sequences to sort based on the Divide-and-Conquer principle and allow sorting in only  $\mathcal{O}(\log^2 n)$  parallel time. Bitonic sequences are twice monotonic sequences, i.e., they consist of a monotonically increasing and a monotonically decreasing sequence. Bitonic sorting networks recursively combine pairs of monotonic sequences into bitonic sequences and then merge them into single monotonic sequences. Starting at single elements, they eventually end up with one sorted monotonic sequence. With the bitonic architecture, we can sort large numbers of input values as we only need  $\log_2 n \cdot ((\log_2 n) + 1)/2$  layers to sort  $n$  inputs. As a consequence, the proposed architecture provides good accuracy even for large input sets and allows scaling up sorting and ranking supervision to large input sets of up to 1024 elements.

Following the experiments in the previous chapter as well as Grover *et al.* [58] and Cuturi *et al.* [56], we benchmark our continuous relaxation of the sorting function on the four-digit MNIST [160] sorting supervision benchmark. To evaluate the performance in the context of a real-world application, we apply our continuous relaxation to the multi-digit images of the Street View House

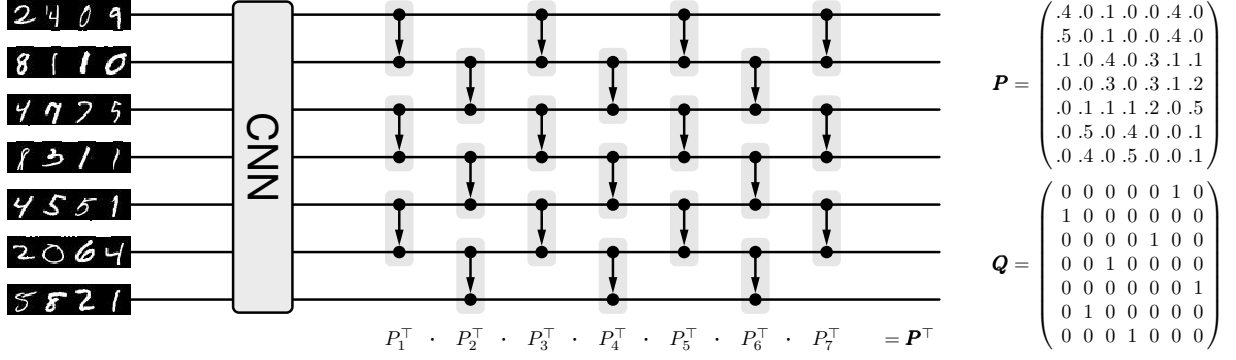


**Figure 3.1:** Overview of the system for training with sorting supervision. *Left:* input images are fed separately / independently into a Convolutional Neural Network (CNN) that maps them to scalar values. *Center:* the odd-even sorting network sorts the scalars by parallel conditional swap operations. *Right:* the sorting network produces a differentiable permutation matrix  $P$  which can then be compared to the ground truth permutation matrix  $Q$  using binary cross-entropy to produce the training loss. By propagating this error backward through the sorting network, we can train the CNN.

Number (SVHN) data set. We compare the performance of both sorting network architectures and evaluate their characteristics under different conditions. We show that both differentiable sorting network architectures outperform existing continuous relaxations of the sorting function on the four-digit MNIST sorting benchmark and also perform well on the more realistic SVHN benchmark. Further, we show that our model scales and achieves performance gains on larger sets of ordered elements and confirm this up to  $n = 1024$  elements. An overview of the overall architecture is shown in Figure 3.1.

In the next chapter, we apply differentiable sorting and ranking to top- $k$  classification learning.

## 3.1 Related Work

In this section, we discuss related work wrt. sorting and ranking. We begin by discussing differentiable sorting and ranking approaches. We continue by discussing learning-to-rank, an application in the domain of recommender systems. After that, we cover neural networks that sort. Finally, we discuss sorting networks, which are the basis for the differentiable sorting networks that we propose in this chapter.

### 3.1.1 SoftSort and NeuralSort

A (hard) permutation matrix is a square matrix with entries 0 and 1 such that every row and every column sums up to 1, which defines the permutation necessary to sort a sequence. To make the sorting operation differentiable, Grover *et al.* [58] proposed NeuralSort, which relaxes permutation matrices to unimodal row-stochastic matrices. For this, they use the softmax of pairwise differences of (cumulative) sums of the top elements. This relaxation allows for gradient-based stochastic optimization. They prove that this, for the temperature parameter approaching 0, is the correct permutation matrix, and propose multiple deep learning differentiable sorting benchmark tasks. Note that NeuralSort is not based on sorting networks. On various tasks, including the four-digit MNIST sorting benchmark, they evaluate their

relaxation against the Sinkhorn and Gumbel-Sinkhorn approaches proposed by Mena *et al.* [149].

Prillo *et al.* [59] build on this idea but simplify the formulation and provide SoftSort, a faster alternative to NeuralSort. They show that it is sufficient to build on pairwise differences of elements of the vectors to be sorted instead of the cumulative sums. Specifically, given a set of  $n$  inputs, i.e.,  $y \in \mathbb{R}^n$ , SoftSort is defined as

$$P(y) = \text{softmax}(-|y^\top \ominus \text{sort}(y)|/\tau) = \text{softmax}(-|y^\top \ominus Sy|/\tau) \quad (3.1)$$

where  $\tau$  is a temperature parameter, “sort” sorts the entries of a vector in non-ascending order,  $\ominus$  is the element-wise broadcasting subtraction,  $|\cdot|$  is the element-wise absolute value, and “softmax” is the row-wise softmax operator. NeuralSort is defined similarly and omitted for the sake of brevity. In the limit of  $\tau \rightarrow 0$ , SoftSort and NeuralSort converge to the exact ranking permutation matrix [58, 59]. Prillo *et al.* [59] find that SoftSort achieves approximately the same accuracy as NeuralSort, while being faster to compute.

### 3.1.2 Optimal Transport / Sinkhorn Sort

Cuturi *et al.* [56] propose an entropy regularized optimal transport (OT) formulation of the sorting operation. They use the idea that sorting can be achieved by minimizing the matching cost between elements and an auxiliary target of increasing values. That is, the smallest element is matched to the first value, the second smallest to the second value, etc. They make this differentiable by regularizing the OT problem with an entropic penalty, solve it by applying Sinkhorn iterations [57] and compute the gradients via automatic differentiation rather than the implicit function theorem, which resolves the need for solving a linear equation system. The Sinkhorn algorithm produces a transport matrix, which can be used as a relaxed permutation matrix.

### 3.1.3 FastSort

Blondel *et al.* [64] propose a differentiable sorting and ranking operator by casting sorting and ranking as linear programs over the permutahedron and regularizing them, which turns them into projections onto the permutahedron. They solve this by reducing it to isotonic optimization and make it differentiable by considering the Jacobians of the isotonic optimization and the projection. While FastSort is asymptotically (to date) the fastest differentiable sorting operator with a complexity of  $\mathcal{O}(n \log n)$ , it does not produce a differentiable permutation matrix. However, in sorting and ranking supervision, we are typically primarily interested in the differentiable permutation matrix, which makes FastSort rather unsuitable for the applications explored in this chapter. Accordingly, they evaluate the proposed approach in the context of top- $k$  classification, and label ranking accuracy via a soft Spearman’s rank correlation coefficient.

### 3.1.4 Learning-to-Rank

RankNet was one of the first approaches to use a neural network for learning to rank [81]. RankNet uses ordered pairs of elements and optimizes a neural network to predict their relevance scores. This was followed shortly after by LambdaRank [82] which incorporates the influence of swapping the considered elements on an information retrieval metric. In 2008, SoftRank [83] was introduced, which uses pairwise distances between the scores of elements to elevate this method to supervising the order of arbitrary many elements. Adams *et al.* [88] propose relaxing permutation matrices to doubly-stochastic matrices based on marginals of distributions over permutation matrices and apply their method to the LETOR learning-to-rank benchmark [172]. We note that these works on neural-network based information retrieval typically used two-layer or similar fully connected networks, as such shallow networks were common at the time of publishing. Recently, Lee *et al.* [85] propose differentiable ranking metrics, and Swezey *et al.* [86] propose Pi-Rank, a learning-to-rank method which is based on the differentiable sorting method NeuralSort.

### 3.1.5 Neural Networks that Sort

In the past, neural networks that sort have been proposed, e.g., by Ceterchi *et al.* [173], who proposed simulating sorting networks with spiking neural P systems. Spiking neural P systems are predecessors of current spiking networks, a form of computational models inspired by biological neurons. This was later adapted by Metta *et al.* [174] for a spiking neural P system with anti-spikes and rules on synapses.

Graves *et al.* [159] raised the idea of integrating sorting capabilities into neural networks in the context of Neural Turing Machines (NTM). The NTM architecture contains two basic components: a neural network controller based on a long short-term memory network (LSTM) and a memory bank with an attention mechanism, both of which are differentiable. The authors use this architecture to sort sequences of binary vectors according to given priorities. Vinyals *et al.* [175] address the problem of the order of input and output elements in LSTM sequence-to-sequence models by content-based attention. To show the effect of the proposed model, they apply it to the task of sorting numbers and formulate the task of sorting as an instance of the set2seq problem. Mena *et al.* [149] introduce the Gumbel-Sinkhorn, a Sinkhorn-operator-based analog of the Gumbel-Softmax distribution for permutations. They evaluate the proposed approach, i.e., on the task of sorting up to 120 numbers. Note that these architectures learn to sort, while sorting networks and differentiable sorting functions sort provably correct. These methods allow sorting input values, as an alternative to classical sorting algorithms, but not training with sorting supervision because they are not meaningfully differentiable.

### 3.1.6 Sorting Networks

The goal of research on sorting networks is to find optimal sorting networks, i.e., networks that can sort an input of  $n$  elements in as few layers of parallel swap operations as possible. Initial attempts to sorting networks required

$\mathcal{O}(n)$  layers, each of which requires  $\mathcal{O}(n)$  operations (examples are bubble and insertion sort [167]). With parallel hardware, these sorting algorithms can be executed in  $\mathcal{O}(n)$  time. Further research led to the discovery of the bitonic sorting network (aka. bitonic sorter), which requires only  $\mathcal{O}(\log^2 n)$  layers [167, 171]. Using genetic and evolutionary algorithms, slightly better optimal sorting networks were found for specific  $n$  [169, 176]. However, these networks do not exhibit a simple, regular structure. Ajtai, Komlós, and Szemerédi [177] presented the AKS sorting network which can sort in  $\mathcal{O}(\log n)$  parallel time, i.e., using only  $\mathcal{O}(n \log n)$  operations. However, the complexity constants for the AKS algorithm are to date unknown and optimistic approximations assume that it is faster than bitonic sort if and only if  $n \gg 10^{80}$ . Today, sorting networks are still in use, e.g., for fast sorting implementations on GPU accelerated hardware as described by Govindaraju *et al.* [168] and in hybrid systems as described by Gowanlock *et al.* [178].

## 3.2 Sorting Networks

In this section, we introduce two common sorting networks: the simple odd-even sorting network as well as the more complex but also more efficient bitonic sorting network.

### 3.2.1 Odd-Even Sorting Network

One of the simplest sorting networks is the odd-even sorting network [179]. Here, neighboring elements are swapped if they are in the wrong order. As the name implies, this is done in a fashion alternating between comparing odd and even indexed elements with their successors. In detail, for sorting an input sequence  $a_1 a_2 \dots a_n$ , each layer updates the elements such that  $a'_i = \min(a_i, a_{i+1})$  and  $a'_{i+1} = \max(a_i, a_{i+1})$  for all odd or even indices  $i$ , respectively. Using  $n$  of such layers, a sequence of  $n$  elements is sorted as displayed in Figure 3.1 (center).

### 3.2.2 Bitonic Sorting Network

Second, we review the bitonic sorting network for sorting  $n = 2^k$  elements where  $k \in \mathbb{N}_+$ . The sorting network can be extended to  $n \in \mathbb{N}_+$  [167], which is also supported in the  **diffsort** library. The bitonic sorting network builds on bitonic sequences: a sequence  $(a_i)_{1 \leq i \leq n}$  is called bitonic if (after an appropriate circular shift)  $a_1 \leq \dots \leq a_j \geq \dots \geq a_n$  for some  $j$ .

Following the Divide-and-Conquer principle, in analogy to merge sort, bitonic sort recursively splits the task of sorting a sequence into the tasks of sorting two subsequences of equal length, which are then combined into a bitonic sequence. Like merge sort, bitonic sort starts by merging individual elements, to obtain sorted lists of length 2 (first gray block in Figure 3.2). Pairs of these are then combined into bitonic sequences and then merged into monotonic sequences (second gray block in Figure 3.2). This proceeds, doubling the length of the sorted sequences with each (gray) block, until the entire sequence is sorted. The difference to merge sort lies in the bitonic

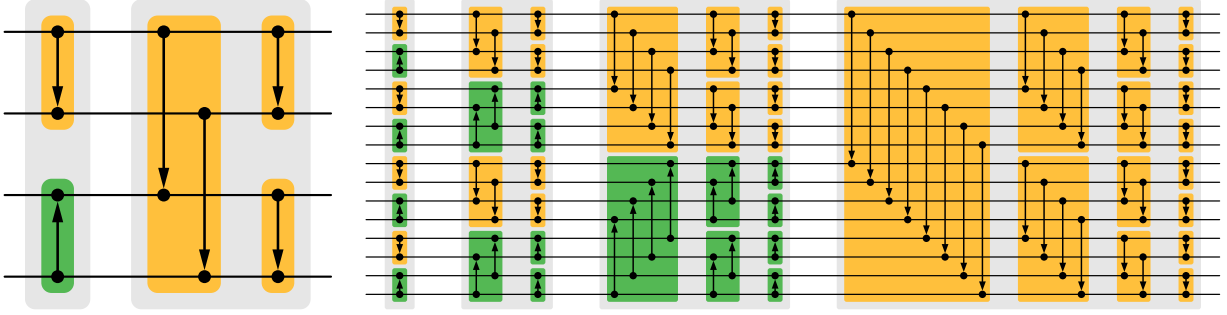


**Figure 3.2:** Bitonic sorting networks for 4 and 16 lanes, consisting of bitonic merge blocks (colored). Arrows pointing toward the maximum.

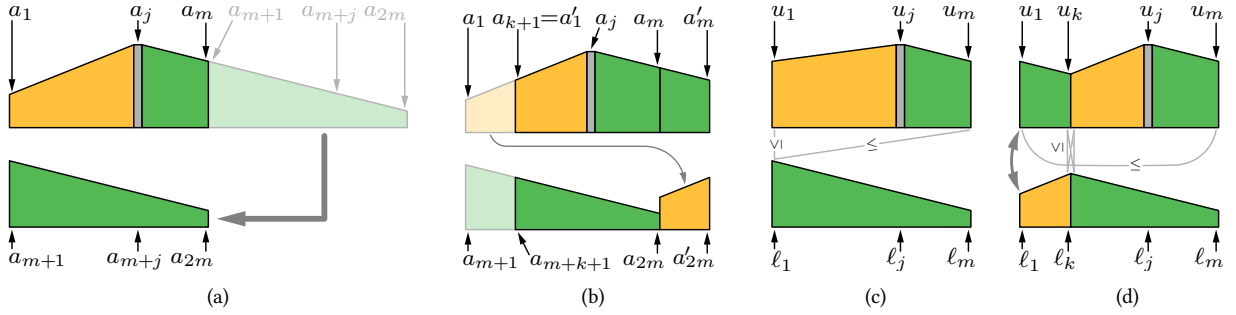


**Figure 3.3:** Bitonic merge turns a bitonic input sequence into two bitonic output sequences, with all elements in the one (upper,  $u_i$ ) sequence larger than all elements in the other (lower,  $\ell_i$ ) sequence. The diagrams show the vertical alignment of elements to compare (a) and the invariance to cyclic permutations (b). Depending on the values in (a), no exchanges (c) or exchanges (d) are executed.

merge operation, which merges two sequences sorted in opposite order (i.e., a single bitonic sequence) into a single sorted (monotonic) sequence.

**Proof Sketch** In the following, we detail the bitonic sorting network and sketch a proof of why the bitonic sorting networks sorts:

A bitonic sequence is sorted by several bitonic merge blocks, shown in orange and green in Figure 3.2. Each block takes a bitonic input sequence  $a_1 a_2 \dots a_{2m}$  of length  $2m$  and turns it into two bitonic output sequences  $\ell_1 \ell_2 \dots \ell_m$  and  $u_1 u_2 \dots u_m$  of length  $m$  that satisfy  $\max_{i=1}^m \ell_i \leq \min_{i=1}^m u_i$ . These subsequences are recursively processed by bitonic merge blocks, until the output sequences are of length 1. At this point, the initial bitonic sequence has been turned into a monotonic sequence due to the minimum/maximum conditions that hold between the output sequences (and thus elements).

A bitonic merge block computes its output as  $\ell_i = \min(a_i, a_{m+i})$  and  $u_i = \max(a_i, a_{m+i})$ . This is depicted in Figure 3.2 by the arrows pointing from the minimum to the maximum. To demonstrate that bitonic merge works, we show that this operation indeed produces two bitonic output sequences for which the relationship  $\max_{i=1}^m \ell_i \leq \min_{i=1}^m u_i$  holds.

Note that neither a cyclic permutation of the sequence (Figure 3.3 (b),  $a'_i = a_{(i+k-1 \bmod 2m)+1}$  for some  $k$ ), nor a reversal, change the bitonic character of the sequence. As can be seen in Figure 3.3 (b), even under cyclic permutation, still the same pairs of elements are considered for a potential swap. Thus, as a cyclic permutation or a reversal only causes the output sequences to be analogously cyclically permuted or reversed, this changes neither the bitonic character of these sequences nor the relationship between them. Therefore, it suffices to consider the special case shown in Figure 3.3a, with a

monotonically increasing sequence (orange) followed by a monotonically decreasing sequence (green) and the maximum element  $a_j$  (gray) in the first half. Note that in this case  $\forall i; j \leq i \leq m : a_i \geq a_{m+i} \wedge u_i = a_i \wedge \ell_i = a_{m+i}$ .

For this case, we distinguish two subcases:  $a_1 \geq a_{m+1}$  and  $a_1 < a_{m+1}$ .

If, on one hand,  $a_1 \geq a_{m+1}$ , we have the situation shown in Figure 3.3 (c): the output sequence  $u_1 u_2 \dots u_m$  is simply the first half of the sequence, the output sequence  $\ell_1 \ell_2 \dots \ell_m$  is the second half. Thus, both output sequences are bitonic (since they are subsequences of a bitonic input sequence) and  $\min_{i=1}^m u_i = \min(u_1, u_m) \geq \ell_1 = \max_{i=1}^m \ell_i$ .

If, on the other hand,  $a_1 < a_{m+1}$ , we can infer  $\exists k; 1 \leq k < j : a_k > a_{m+k} \wedge a_{k+1} \leq a_{m+k+1}$ . This situation is depicted in Figure 3.3 (d). Thus,  $\forall i; 1 \leq i \leq k : u_i = a_{m+i} \wedge \ell_i = a_i$  and  $\forall i; k < i \leq m : u_i = a_i \wedge \ell_i = a_{m+i}$ . Since  $u_k = a_{m+k} > a_k = \ell_k$ ,  $u_k = a_{m+k} \geq a_{m+k+1} = \ell_{k+1}$ ,  $u_{k+1} = a_{k+1} \geq a_{m+k+1} = \ell_{k+1}$ ,  $u_{k+1} = a_{k+1} \geq a_k = \ell_k$ , we obtain  $\max_{i=1}^m \ell_i \leq \min_{i=1}^m u_i$ . Figure 3.3 (d) shows that the two output sequences are bitonic and that all elements of the upper output sequence are greater than or equal to all elements of the lower output sequence.

### 3.3 Differentiable Sorting Networks

To relax sorting networks, we need to relax the min and max operators, which are used as a basis for the swap operations in sorting networks. By perturbing the difference between the two inputs with a distribution, min and max can be relaxed to differentiable *min* and *max*. Note that we denote the differentiable relaxations in *italic* font and their hard counterparts in roman font. The differentiable relaxations *min* and *max* differ from the commonly used softmin and softmax, which are relaxations of argmin and argmax [68].

One example of such a relaxation of min and max is the logistic relaxation

$$\min_{\sigma}(a, b) = a \cdot \sigma(b - a) + b \cdot \sigma(a - b), \quad (3.2)$$

$$\max_{\sigma}(a, b) = a \cdot \sigma(a - b) + b \cdot \sigma(b - a) \quad (3.3)$$

where  $\sigma$  is the logistic sigmoid function with inverse temperature  $\beta > 0$ :

$$\sigma : x \mapsto \frac{1}{1 + e^{-\beta x}}. \quad (3.4)$$

In the limit, for  $\beta \rightarrow \infty$ ,  $\sigma$  converges to the Heaviside function and the smooth operators converge to the discrete operators. We formalize the notion of a sigmoid function and generalize it beyond the example of the logistic distribution.

**Definition 3.1** (Sigmoid Function) *We define a (unipolar) sigmoid (i.e., s-shaped) function as a function  $f$  that is continuous, monotonically non-decreasing, and odd-symmetric (around  $\frac{1}{2}$ ) with*

$$f : \mathbb{R} \rightarrow [0, 1] \quad \text{with} \quad \lim_{x \rightarrow -\infty} f(x) = 0 \quad \text{and} \quad \lim_{x \rightarrow \infty} f(x) = 1.$$

Based on a sigmoid function  $f$ , we can define a continuous conditional swap as follows.

**Definition 3.2** (Continuous Conditional Swaps) *Following [2], we define a continuous conditional swap in terms of a sigmoid function  $f : \mathbb{R} \rightarrow [0, 1]$  as*

$$\min_f(a, b) = a \cdot f(b - a) + b \cdot f(a - b) \quad (3.5)$$

$$\max_f(a, b) = a \cdot f(a - b) + b \cdot f(b - a) \quad (3.6)$$

$$\operatorname{argmin}_f(a, b) = (f(b - a), f(a - b)) \quad (3.7)$$

$$\operatorname{argmax}_f(a, b) = (f(a - b), f(b - a)). \quad (3.8)$$

We require a continuous odd-symmetric sigmoid function to preserve most of the properties of min and max, while also making  $\operatorname{argmin}$  and  $\operatorname{argmax}$  continuous as discussed in Supplementary Material C.

### 3.3.1 Differentiable Permutation Matrices

For sorting and ranking supervision, i.e., training a neural network to predict scalars, where only the order of these scalars is known, we use the ground truth permutation matrix as supervision. Thus, to train an underlying neural network end-to-end through the differentiable sorting network, we need to return the underlying permutation matrix rather than the actual sorted scalar values. For that, we compute the permutation matrices for the swap operations for each layer as shown in Figure 3.1. Here, for all swap operations between any elements  $a_i$  and  $a_j$  that are to be ordered in non-descending order, the layer-wise permutation matrix is

$$P_{l,ii} = P_{l,jj} = \alpha_{ij} = f(a_j - a_i), \quad (3.9)$$

$$P_{l,ij} = P_{l,ji} = 1 - \alpha_{ij} = 1 - f(a_j - a_i) \quad (3.10)$$

where all other entries of  $P_l$  are set to 0. By multiplication, we compute the complete relaxed permutation matrix  $\mathbf{P}$  as

$$\mathbf{P} = P_n \cdot \dots \cdot P_2 \cdot P_1 = \left( \prod_{l=1}^n P_l^\top \right)^\top. \quad (3.11)$$

A column in the relaxed permutation matrix can be seen as a distribution over possible ranks for the corresponding input value. Multiplying  $\mathbf{P}$  with an input  $x$  yields the differentiably sorted vector  $\hat{x} = \mathbf{P}x$ , which is also the output of the differentiable sorting network. Note that computing  $\mathbf{P}$  is optional, as we can compute  $\hat{x}$  faster without computing it via  $\mathbf{P}$  because  $\hat{x}$  is just the output of the differentiable sorting network. Whether it is necessary to compute  $\mathbf{P}$ , or whether  $\hat{x}$  suffices, depends on the specific application. For example, for a cross-entropy ranking / sorting loss as used in the experiments in Section 3.5,  $\mathbf{P}$  can be used to compute the cross-entropy to a ground truth permutation matrix  $\mathbf{Q}$  as

$$\mathcal{L} := \sum_{c=1}^n \left( \frac{1}{n} \operatorname{CE}(\mathbf{P}_c, \mathbf{Q}_c) \right) \quad (3.12)$$

where  $\mathbf{P}_c$  and  $\mathbf{Q}_c$  denote the  $c$ th columns of  $\mathbf{P}$  and  $\mathbf{Q}$ , respectively. Note that, as the cross-entropy loss is, by definition, computed element-wise, the column-wise cross-entropy is equivalent to the row-wise cross-entropy.

In the following, we establish doubly-stochasticity and differentiability of  $\mathbf{P}$ , which are important properties for differentiable sorting and ranking operators.

**Lemma 3.1** (Doubly-Stochasticity and Differentiability of  $\mathbf{P}$ ) (i) *The relaxed permutation matrix  $\mathbf{P}$ , produced by a differentiable sorting network, is doubly-stochastic.* (ii)  *$\mathbf{P}$  has the same differentiability as  $f$ , e.g., if  $f$  is continuously differentiable in the input,  $\mathbf{P}$  will be continuously differentiable in the input to the sorting network. If  $f$  is differentiable almost everywhere (a.e.),  $\mathbf{P}$  will be differentiable a.e.*

*Proof.* (i) For each conditional swap between two elements  $i, j$ , the relaxed permutation matrix is 1 at the diagonal except for rows  $i$  and  $j$ : at points  $i, i$  and  $j, j$  the value is  $v \in [0, 1]$ , at points  $i, j$  and  $j, i$  the value is  $1 - v$  and all other entries are 0. This is doubly-stochastic as all rows and columns add up to 1 by construction. As the product of doubly-stochastic matrices is doubly-stochastic, the relaxed permutation matrix  $\mathbf{P}$ , produced by a differentiable sorting network, is doubly-stochastic.

(ii) The composition of differentiable functions is differentiable and the addition and multiplication of differentiable functions is also differentiable. Thus, a sorting network is differentiable if the employed sigmoid function is differentiable. “Differentiable” may be replaced with any other form of differentiability, such as “differentiable a.e.”  $\square$

### 3.3.2 The Activation Replacement Trick $\varphi$

In this subsection, we present the activation replacement trick (ART), which can greatly enhance the performance of differentiable sorting networks in the case of using the logistic sigmoid function / distribution. In the next subsection, we present monotonic differentiable sorting networks, an alternative to the ART and derived from a different viewpoint. As we will see in the experimental evaluation, monotonic differentiable sorting networks typically outperform differentiable sorting networks with the ART; however, there are also cases in which the ART performs better, specifically for large  $n$ .

Due to the nature of the sorting network, values with large as well as very small differences are compared in each layer. Comparing values with large differences causes vanishing gradients, while comparing values with very small differences can modify, i.e., blur, values as they are only partially swapped. Specifically, in the case of the logistic function, it is saturated for large inputs but also returns a value close to the mean for inputs that are close to each other. Based on these observations, we propose an activation replacement trick, which avoids vanishing gradients as well as blurring. That is, we modify the distribution of the differences between compared values to avoid small differences close to 0 as well as large differences.

Assuming that the inputs to a sorting network are normally distributed, there are many cases in which the differences of two values  $|a_j - a_i|$  are

very small as well as many cases in which the differences are very large. For the relaxation of sorting networks, this poses two problems:

If  $|a_j - a_i|$  is close to 0, while we obtain large gradients, this also blurs the two values to a great extent, modifying them considerably. Thus, it is desirable to avoid  $|a_j - a_i| \approx 0$ .

On the other hand, if  $|a_j - a_i|$  is large, vanishing gradients occur, which hinders training.

To counter these two problems at the same time, we propose the activation replacement trick. We transform the differences between two values to be potentially swapped (e.g.,  $x = (a_j - a_i)$ ) from a unimodal Gaussian distribution into a bimodal distribution, which has a low probability density around 0. To this end, we apply the transformation

$$\varphi : x \mapsto \frac{x}{|x|^\lambda + \epsilon} \quad (3.13)$$

to the differences  $x$ , where  $\lambda \in [0, 1]$  and  $\epsilon \approx 10^{-10}$ .  $\varphi$  pushes all input values (depending on the sign) toward  $-1$  and  $+1$ , respectively. Thus, by applying  $\varphi$  before  $\sigma$ , we move the input values outside  $[-1, +1]$  to positions at which they have a larger gradient, thus mitigating the problem of vanishing gradients. Simultaneously, we achieve a probability density of 0 at  $|a_j - a_i| = 0$  (i.e., here  $p(\varphi(0)) = 0$ ) as all values close to zero are mapped toward  $-1$  and  $+1$ , respectively. This is displayed in Figure 3.4.

As we multiply by the inverse temperature parameter  $\beta$ , we map the input to the sigmoid function toward  $-\beta$  and  $+\beta$ , respectively. Thus, when replacing  $\sigma(x)$  by  $\sigma(\varphi(x))$ , we push the output values toward  $\frac{1}{1+e^{-1 \cdot \beta}}$  or  $\frac{1}{1+e^{1 \cdot \beta}}$ . This increases the gradient  $\frac{\partial \sigma(\varphi(x))}{\partial x}$  for large  $\text{abs}(x)$  which are those values causing the vanishing gradients, addressing the problem of vanishing gradients. Further, for all  $x \in (-1, +1)$  this pushes the output values away from  $1/2$ , addressing the problem of blurring of values.

This leads to a sigmoid function of  $\sigma \circ \varphi$ . Empirically, the activation replacement trick accelerates the training through our sorting network. We observe that, while sorting networks up to 21 layers (i.e., bitonic networks with  $n \leq 64$ ) can operate with moderate inverse temperature (i.e.,  $\beta \leq 15$ ) and without the activation replacement trick (i.e.,  $\lambda = 0$ ), for more layers, the activation replacement trick becomes necessary for good performance. Notably, the activation replacement trick also improves the performance for sorting networks with fewer layers. Further, the activation replacement trick allows training with smaller inverse temperature  $\beta$ , which makes training more stable specifically for long sequences as it avoids exploding gradients.

Note that, in the case of bitonic sorting networks, in the first layer of the last merge block,  $n/2$  elements in non-descending order are element-wise compared to  $n/2$  elements in non-ascending order. Thus, in this layer, we compare the minimum of the first sequence to the maximum of the second sequence and vice versa. At the same time, we also compare the median of both sequences as well as values close to the median to each other. While we consider very large differences as well as very small differences in the same layer, the activation replacement trick achieves an equalization of the mixing behavior, reducing blurring and vanishing gradients.

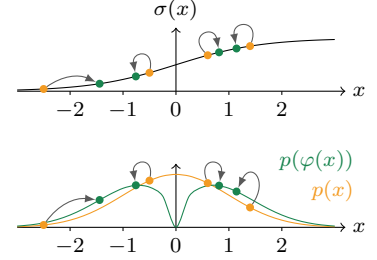


**Figure 3.4:** The Activation Replacement Trick. *Top:* on the logistic sigmoid function, the input values  $x$  (orange) are mapped to  $\varphi(x)$  (green) and are thus closer to  $-1$  and  $+1$ . *Bottom:* probability density functions of Gaussian distributed input values  $x$  (orange) and the distribution of replaced input values  $\varphi(x)$  (green).

We note that the activation replacement trick and monotonicity are mutually exclusive. They both have independent theoretical motivations, justifications, and advantages. While the activation replacement trick has its merits in sorting large numbers of elements  $n$ , in general, monotonic differentiable sorting networks perform better and convey more stable behavior.

### 3.3.3 Monotonic Differentiable Sorting Networks

In this subsection, we formalize the notion of monotonic and error-bounded differentiable sorting networks and characterize families of monotonic and error-bounded differentiable sorting networks. For this, we start by defining monotonicity in continuous conditional swap operations.

**Definition 3.3** (Monotonic Continuous Conditional Swaps) *We say  $f$  produces monotonic conditional swaps if  $\min_f(x, 0)$  is non-decreasingly monotonic in  $x$ , i.e.,  $\min'_f(x, 0) \geq 0$  for all  $x$ .*

It is sufficient to define it w.l.o.g. in terms of  $\min_f(x, 0)$  due to its commutativity, stability, and odd-symmetry of the operators (cf. Supplementary Material C).

**Theorem 3.2** (Monotonicity of Continuous Conditional Swaps) *A continuous conditional swap (in terms of a differentiable sigmoid function  $f$ ) being non-decreasingly monotonic in all arguments and outputs requires that the derivative of  $f$  decays no faster than  $1/x^2$ , i.e.,*

$$f'(x) \in \Omega\left(\frac{1}{x^2}\right). \quad (3.14)$$

*Proof.* We show that Equation (3.14) is a necessary criterion for monotonicity of the conditional swap. Because  $f$  is a continuous sigmoid function with  $f : \mathbb{R} \rightarrow [0, 1]$ ,  $\min_f(x, 0) = f(-x) \cdot x > 0$  for some  $x > 0$ . Thus, monotonicity of  $\min_f(x, 0)$  implies  $\limsup_{x \rightarrow \infty} \min_f(x, 0) > 0$  (otherwise the value would decrease again from a value  $> 0$ .) Thus,

$$\lim_{x \rightarrow \infty} \min_f(x, 0) = \lim_{x \rightarrow \infty} f(-x) \cdot x = \lim_{x \rightarrow \infty} \frac{f(-x)}{1/x} \quad (3.15)$$

$$\stackrel{\text{(L'Hôpital's rule)}}{=} \lim_{x \rightarrow \infty} \frac{-f'(-x)}{-1/x^2} = \lim_{x \rightarrow \infty} \frac{f'(-x)}{1/x^2} = \lim_{x \rightarrow \infty} \frac{f'(x)}{1/x^2} \quad (3.16)$$

$$= \limsup_{x \rightarrow \infty} \frac{f'(x)}{1/x^2} > 0 \iff f'(x) \in \Omega\left(\frac{1}{x^2}\right). \quad (3.17)$$

assuming  $\lim_{x \rightarrow \infty} \frac{f'(x)}{1/x^2}$  exists. Otherwise, it can be proven analogously via a proof by contradiction.  $\square$

After defining the monotonicity of conditional swap operators, we can extend this to differentiable sorting networks.

**Corollary 3.3** (Monotonic Sorting Networks) *If the individual conditional swaps of a sorting network are monotonic, the sorting network is also monotonic.*

*Proof.* If single layers  $g, h$  are non-decreasingly monotonic in all arguments and outputs, their composition  $h \circ g$  is also non-decreasingly monotonic in all arguments and outputs. Thus, a network of arbitrarily many layers is non-decreasingly monotonic.  $\square$

Above, we formalized the property of monotonicity. Another important aspect is whether the error of the differentiable sorting network is bounded. It is very desirable to have a bounded error because without bounded errors, the error, i.e., the difference between the result of the differentiable sorting network and the result of the hard sorting function, diverges to infinity. Minimizing this error is desirable.

**Definition 3.4** (Error-Bounded Continuous Conditional Swaps) *A continuous conditional swap has a bounded error if and only if  $\sup_x \min_f(x, 0) = c$  is finite. The continuous conditional swap is therefore said to have an error bounded by  $c$ .*

It is sufficient to define it w.l.o.g. in terms of  $\min_f(x, 0)$  due to its commutativity, stability, and odd-symmetry of the operators (cf. Supplementary Material C). In general, for better comparability between functions, we assume a Lipschitz continuous function  $f$  with Lipschitz constant 1.

**Theorem 3.4** (Error-Bounds of Continuous Conditional Swaps)

(i) *A differentiable continuous conditional swap has a bounded error if*

$$f'(x) \in \mathcal{O}\left(\frac{1}{x^2}\right). \quad (3.18)$$

(ii) *If it is additionally monotonic, the error-bound can be found as  $\lim_{x \rightarrow \infty} \min_f(x, 0)$  and additionally the error is bound only if Equation (3.18) holds.*

*Proof.* (i) W.l.o.g. we consider  $x > 0$ . Let  $g(z) := f(-1/z)$ ,  $g(0) = 0$ . Thus,  $g'(z) = 1/z^2 \cdot f'(-1/z) \leq c$  according to Equation (3.18). Thus,  $g(z) = g(0) + \int_0^z g'(t)dt \leq c \cdot z$ . Therefore,  $f(-1/z) \leq c \cdot z \implies 1/z \cdot f(-1/z) \leq c$  and with  $x = 1/z \implies x \cdot f(-x) = \min_f(x, 0) \leq c$ .

(ii) Let  $\min_f(x, 0)$  be monotonic and bound by  $\min_f(x, 0) \leq c$ . For  $x > 0$  and  $h(x) := \min_f(x, 0)$ ,

$$\begin{aligned} h'(x) &= -x \cdot f'(-x) + f(-x) \implies x^2 f'(-x) = \underbrace{-x h'(x)}_{\leq 0} + x \cdot f(-x) \\ &\leq x \cdot f(-x) \leq c. \end{aligned} \quad (3.19)$$

Thus,  $f'(x) \in \mathcal{O}\left(\frac{1}{x^2}\right)$ .  $\square$

After characterizing a family of distributions leading to error-bounded conditional swap operators, we can also extend this property to differentiable sorting networks.

**Theorem 3.5** (Error-Bounds of Diff. Sorting Networks) *If the error of individual conditional swaps of a sorting network is bounded by  $\epsilon$  and the network has  $\ell$  layers, the total error is bounded by  $\epsilon \cdot \ell$ .*

*Proof.* Induction over number  $k$  of executed layers. Let  $x^{(k)}$  be input  $x$  differentially sorted for  $k$  layers and  $\mathbf{x}^{(k)}$  be input  $x$  hard sorted for  $k$  layers as an anchor. We require this anchor, as it is possible that  $\mathbf{x}_i^{(k)} < \mathbf{x}_j^{(k)}$  but  $x_i^{(k)} > x_j^{(k)}$  for some  $i, j, k$ .

Begin of induction:  $k = 0$ . Input vector  $x$  equals the vector  $x^{(0)}$  after 0 layers. Thus, the error is equal to  $0 \cdot \epsilon$ .

Step of induction: Given that after  $k - 1$  layers the error is smaller than or equal to  $(k - 1)\epsilon$ , we need to show that the error after  $k$  layers is smaller than or equal to  $k\epsilon$ .

The layer consists of comparator pairs  $i, j$ . W.l.o.g. we assume  $\mathbf{x}_i^{(k-1)} \leq \mathbf{x}_j^{(k-1)}$ . W.l.o.g. we assume that wire  $i$  will be the min and that wire  $j$  will be the max, therefore  $\mathbf{x}_i^{(k)} \leq \mathbf{x}_j^{(k)}$ . This implies  $\mathbf{x}_i^{(k-1)} = \mathbf{x}_i^{(k)}$  and  $\mathbf{x}_j^{(k-1)} = \mathbf{x}_j^{(k)}$ . We distinguish two cases:

- $(\mathbf{x}_i^{(k-1)} \leq \mathbf{x}_j^{(k-1)} \text{ and } x_i^{(k-1)} \leq x_j^{(k-1)})$  According to the assumption,  $|x_i^{(k-1)} - \mathbf{x}_i^{(k)}| \leq \epsilon$  and  $|x_j^{(k-1)} - \mathbf{x}_j^{(k)}| \leq \epsilon$ . Thus,  $|x_i^{(k)} - \mathbf{x}_i^{(k)}| \leq |x_i^{(k-1)} - \mathbf{x}_i^{(k-1)}| + |x_i^{(k-1)} - x_i^{(k)}| \leq (k - 1)\epsilon + \epsilon = k\epsilon$ .
- $(\mathbf{x}_i^{(k-1)} \leq \mathbf{x}_j^{(k-1)} \text{ but } x_i^{(k-1)} > x_j^{(k-1)})$  This case can only occur if  $|x_j^{(k-1)} - \mathbf{x}_i^{(k-1)}| \leq (k - 1)\epsilon$  and  $|x_i^{(k-1)} - \mathbf{x}_j^{(k-1)}| \leq (k - 1)\epsilon$  because  $\mathbf{x}_i^{(k-1)}$  and  $\mathbf{x}_j^{(k-1)}$  have to be so close that within margin of error such a reversed order is possible. According to the assumption,  $|x_j^{(k-1)} - x_i^{(k)}| \leq \epsilon$  and  $|x_i^{(k-1)} - x_j^{(k)}| \leq \epsilon$ . Thus,  $|x_i^{(k)} - \mathbf{x}_i^{(k)}| \leq |x_j^{(k-1)} - \mathbf{x}_i^{(k-1)}| + |x_j^{(k-1)} - x_i^{(k)}| \leq (k - 1)\epsilon + \epsilon = k\epsilon$ .  $\square$

**Discussion** Monotonicity is highly desirable because, otherwise, adverse effects can occur, e.g., an input requiring to be decreased in order to increase the output. In gradient-based training, non-monotonicity is problematic as it produces gradients with the opposite sign. In addition, as monotonicity is also given in hard sorting networks, it is desirable to preserve this property in the relaxation. Further, monotonic differentiable sorting networks are quasiconvex and quasiconcave as any monotonic function is both quasiconvex and quasiconcave, which leads to favorable convergence rates [170]. Bounding and reducing the deviation from its hard counterpart reduces the relaxation error, and is therefore desirable.

**Corollary 3.6** (Monotonic and Error-Bounded Diff. Sorting Networks) *We can approximately characterize the sigmoid function that we are interested in, i.e., those which lead to monotonic and error-bounded differentiable sorting networks as*

$$f'(x) \in \Theta\left(\frac{1}{x^2}\right). \quad (3.20)$$

*This characterization is approximate as for Theorem 3.2 an asymptotic characterization can only be a requirement for monotonicity but cannot be sufficient. This is because a local non-monotonic behavior cannot be prevented through an asymptotic requirement.*

*Proof.* Equation (3.20) ( $\Theta$ ) is the intersection of (3.14) ( $\Omega$ ) and (3.18) ( $\mathcal{O}$ ).  $\square$

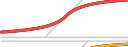
| Function               | $f$ (CDF)                                                                         | $f'$ (PDF)                                                                        | Eq.          | Mono. | Bounded Error                |
|------------------------|-----------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|--------------|-------|------------------------------|
| $\sigma$               |  |  | (3.21)       | ✗     | ✓ ( $\approx .0696/\alpha$ ) |
| $\sigma \circ \varphi$ |  |  | (3.21, 3.13) | ✗     | ✓ ( $\approx .0302^*$ )      |
| $f_{\mathcal{R}}$      |  |  | (3.22)       | ✓     | ✓ ( $1/4/\alpha$ )           |
| $f_{\mathcal{C}}$      |  |  | (3.23)       | ✓     | ✓ ( $1/\pi^2/\alpha$ )       |
| $f_{\mathcal{O}}$      |  |  | (3.24)       | ✓     | ✓ ( $1/16/\alpha$ )          |

**Table 3.1:** For each function, we display the *function*, its *derivative*, and indicate whether the respective relaxed sorting network is *monotonic* and has a *bounded error*.

### Examples

Above, we have specified the space of functions for the differentiable swap operation, as well as their desirable properties. In the following, we discuss four notable candidates as well as their properties. The properties of these functions are visualized in Figures 3.5 and 3.6 and an overview of their properties is given in Table 3.1.

**Logistic Distributions** The first candidate is the logistic sigmoid function (the CDF of a logistic distribution) as proposed earlier:

$$\sigma(x) = \text{CDF}_{\mathcal{L}}(\beta x) = \frac{1}{1 + e^{-\beta x}} \quad (3.21)$$

This function is the de-facto default sigmoid function in machine learning. It provides a continuous, error-bounded, and Lipschitz continuous conditional swap. However, for the logistic function, monotonicity is not given, as displayed in Figure 3.5.

**Reciprocal Sigmoid Function** To obtain a function that yields a monotonic as well as error-bound differentiable sorting network, a necessary criterion is  $f'(x) \in \Theta(1/x^2)$  (the intersection of Equations (3.14) and (3.18).) A natural choice is, therefore,  $f'_{\mathcal{R}}(x) = \frac{1}{(2|x|+1)^2}$ , which produces

$$f_{\mathcal{R}}(x) = \int_{-\infty}^x \frac{1}{(2\beta|t|+1)^2} dt = \frac{1}{2} \frac{2\beta x}{1+2\beta|x|} + \frac{1}{2}. \quad (3.22)$$

$f_{\mathcal{R}}$  fulfills all criteria, i.e., it is an adequate sigmoid function and produces monotonic and error-bound conditional swaps. It has an  $\epsilon$ -bounded-error of  $\epsilon = 0.25$ . It is also an affine transformation of the elementary bipolar sigmoid function  $x \mapsto \frac{x}{|x|+1}$ . Properties of this function are visualized in Table 3.1 and Figure 3.5. Proofs for monotonicity can be found in Supplementary Material C.

**Cauchy Distributions** By using the CDF of the Cauchy distribution, we maintain monotonicity while reducing the error-bound to  $\epsilon = 1/\pi^2 \approx 0.101$ . It is defined as

$$f_{\mathcal{C}}(x) = \text{CDF}_{\mathcal{C}}(\beta x) = \frac{1}{\pi} \int_{-\infty}^x \frac{\beta}{1 + (\beta t)^2} dt = \frac{1}{\pi} \arctan(\beta x) + \frac{1}{2} \quad (3.23)$$

In the experimental evaluation, we find that tightening the error improves the performance.

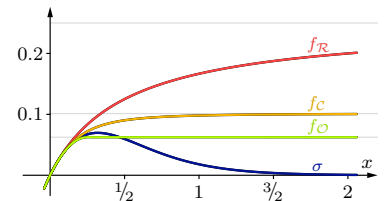


**Figure 3.5:**  $\min_f(x, 0)$  for different sigmoid functions  $f$ ; color coding as in Table 3.1.

**Optimal Monotonic Sigmoid Function** At this point, we are interested in the monotonic swap operation that minimizes the error-bound. Here, we set 1-Lipschitz continuity again as a requirement to make different relaxations of conditional swaps comparable. We show that  $f_{\mathcal{O}}$  is the best possible sigmoid function achieving an error-bound of only  $\epsilon = 1/16$

**Theorem 3.7** (Optimal Sigmoid Function) *The optimal sigmoid function minimizing the error-bound, while producing a monotonic and 1-Lipschitz continuous (with  $\beta = 1$ ) conditional swap operation, is*

$$f_{\mathcal{O}}(x) = \begin{cases} -\frac{1}{16\beta x} & \text{if } \beta x < -\frac{1}{4}, \\ 1 - \frac{1}{16\beta x} & \text{if } \beta x > +\frac{1}{4}, \\ \beta x + \frac{1}{2} & \text{otherwise.} \end{cases} \quad (3.24)$$

*Proof.* Given the above conditions, the optimal sigmoid function is uniquely determined and can easily be derived as follows: Due to stability, it suffices to consider  $\min_f(x, 0) = x \cdot f(-x)$  or  $\max_f(0, x) = -x \cdot f(x)$ . Due to symmetry and inversion, it suffices to consider  $\min_f(x, 0) = x \cdot f(-x)$  for  $x > 0$ .

Since  $\min(x, 0) = 0$  for  $x > 0$ , we have to choose  $f$  in such a way as to make  $\min_f(x, 0) = x \cdot f(-x)$  as small as possible, but not negative. For this,  $f(-x)$  must be made as small as possible. Since we know that  $f(0) = \frac{1}{2}$  and we are limited to functions  $f$  that are Lipschitz continuous with  $\alpha = 1$ ,  $f(-x)$  cannot be made smaller than  $\frac{1}{2} - x$ , and hence  $\min_f(x, 0)$  cannot be made smaller than  $x \cdot (\frac{1}{2} - x)$ . To make  $\min_f(x, 0)$  as small as possible, we have to follow  $x \cdot (\frac{1}{2} - x)$  as far as possible (i.e., to values  $x$  as large as possible). Monotonicity requires that this function can be followed only up to  $x = \frac{1}{4}$ , at which point we have  $\min_f(\frac{1}{4}, 0) = \frac{1}{4} \cdot (\frac{1}{2} - \frac{1}{4}) = \frac{1}{16}$ . For larger  $x$ , that is, for  $x > \frac{1}{4}$ , the value of  $x \cdot (\frac{1}{2} - x)$  decreases again and hence the functional form of the sigmoid function  $f$  has to change at  $x = \frac{1}{4}$  to remain monotonic.

The best that can be achieved for  $x > \frac{1}{4}$  is to make it constant, as it must not decrease (due to monotonicity) and should not increase (to minimize the deviation from the crisp / hard version). That is,  $\min_f(x, 0) = \frac{1}{16}$  for  $x > \frac{1}{4}$ . It follows  $x \cdot f(-x) = \frac{1}{16}$  and hence  $f(-x) = \frac{1}{16x}$  for  $x > \frac{1}{4}$ . Note that, if the transition from the linear part to the hyperbolic part were at  $|x| < \frac{1}{4}$ , the function would not be Lipschitz continuous with  $\alpha = 1$ .  $\square$

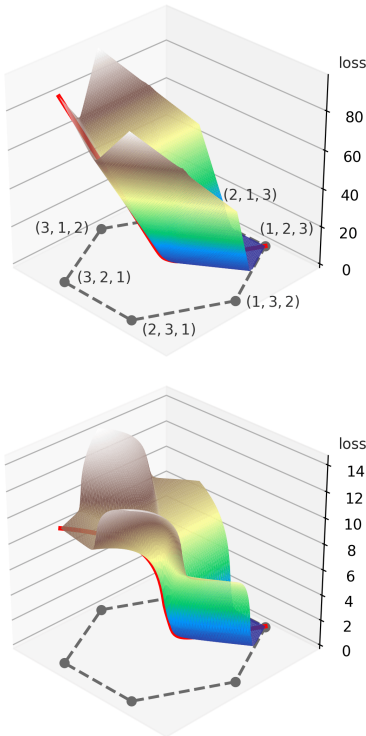


**Figure 3.6:** Loss for a 3-wire odd-even sorting network, drawn over a permutahedron projected onto the  $x$ - $y$ -plane. For logistic sigmoid (top) and optimal sigmoid (bottom).

An overview of the selection of sigmoid functions we consider is shown in Table 3.1. Note how  $f_{\mathcal{R}}$ ,  $f_{\mathcal{C}}$  and  $f_{\mathcal{O}}$  in this order get closer to  $x + \frac{1}{2}$  (the light gray diagonal line) and hence steeper in their middle part. This is reflected by a widening region of values of the derivatives that are close to or even equal to 1.

Table 3.1 also indicates whether a sigmoid function yields a *monotonic* swap operation or not, which is visualized in Figure 3.5: clearly  $\sigma$ -based sorting networks are not monotonic, while all others are. It also states whether the *error is bounded*, which for a monotonic swap operation means  $\lim_{x \rightarrow \infty} \min_f(x, 0) < \infty$ , and gives their bound relative to the Lipschitz constant  $\alpha$ .

Figure 3.6 displays the loss for a sorting network with  $n = 3$  inputs. We project the hexagon-shaped 3-value permutahedron onto the  $x$ - $y$ -plane, while the  $z$ -axis indicates the loss. Note that, at the rightmost point  $(1, 2, 3)$ , the loss is 0 because all elements are in the correct order, while at the left front  $(2, 3, 1)$  and rear  $(3, 1, 2)$  the loss is at its maximum because all elements are at the wrong positions. Along the red center line, the loss rises logarithmic for the optimal sigmoid function on the right. Note that the monotonic sigmoid functions produce a loss that is larger when more elements are in the wrong order. For the logistic function,  $(3, 2, 1)$  has the same loss as  $(2, 3, 1)$  even though one of the ranks is correct at  $(3, 2, 1)$ , while for  $(2, 3, 1)$  all three ranks are incorrect.

### 3.4 Monotonicity and Error-Boundedness of Differentiable Sorting Operators

The *Relaxed Bubble sort*, as proposed in the previous chapter, uses logistic distributions, which indicates a lack of monotonicity. In fact, for  $n = 2$ , Relaxed Bubble sort is equivalent to differentiable sorting networks with the logistic distribution, i.e., it is non-monotonic.

For *differentiable sorting networks with the activation replacement trick*, the asymptotic character of  $\sigma \circ \varphi$  does not fulfill the requirement set by Theorem 3.2, and is, therefore, non-monotonic as also displayed in Figure 3.7 (purple).

For the special case of  $n = 2$ , i.e., for sorting two elements, *NeuralSort* [58] is equivalent to differentiable sorting networks with the logistic sigmoid function. Thus, it is non-monotonic as displayed in Figure 3.7. The same also applies to *SoftSort* [59].

For *SinkhornSort*, we can simply construct an example of non-monotonicity by keeping one value fixed, e.g., at zero, and varying the second value ( $x$ ) as in Figure 3.7 and displaying the minimum. Interestingly, for the case of  $n = 2$ , this function is numerically equal to *NeuralSort* and differentiable sorting networks with the logistic function.

For *FastSort*, we follow the same principle and find that it is indeed monotonic (in this example); however, the error is unbounded, which is undesirable.

We summarize monotonicity and error-boundedness for all differentiable sorting functions in Table 3.2.

### 3.5 Experiments

The experimental section is structured as follows: first, we give all details on the overall training setting; second, we evaluate differentiable sorting networks with the ART, which allows us to do large-scale experiments; third, we benchmark all methods and focus on the benefits of monotonic differentiable sorting networks; finally, we perform a runtime and memory analysis. We evaluate the proposed differentiable sorting networks on the four-digit MNIST sorting benchmark [56, 58] as well as on the real-world SVHN data set.

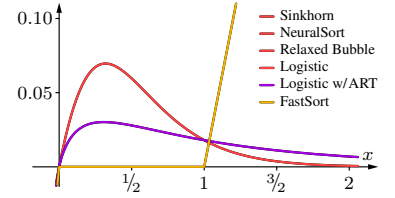


**Figure 3.7:**  $\min(x, 0)$  for SinkhornSort (red), NeuralSort (red), Relaxed Bubble sort (red), diffsort with logistic sigmoid (red), diffsort with activation replacement trick (purple), and FastSort (orange).

**Table 3.2:** For each differentiable sorting operator, whether it is monotonic (M), and whether it has a bounded error (BE).

| Method                                        | M | BE |
|-----------------------------------------------|---|----|
| NeuralSort                                    | ✗ | –  |
| SoftSort                                      | ✗ | –  |
| SinkhornSort                                  | ✗ | –  |
| FastSort                                      | ✓ | ✗  |
| Relaxed Bubble Sort                           | ✗ | –  |
| Diff. Sorting Networks $\sigma$               | ✗ | ✓  |
| Diff. Sorting Networks $\sigma \circ \varphi$ | ✗ | ✓  |
| Diff. Sorting Networks $f_{\mathcal{R}}$      | ✓ | ✓  |
| Diff. Sorting Networks $f_{\mathcal{C}}$      | ✓ | ✓  |
| Diff. Sorting Networks $f_{\mathcal{O}}$      | ✓ | ✓  |

**MNIST** For the four-digit MNIST sorting benchmark, MNIST digits are concatenated to four-digit numbers, e.g., . A CNN then predicts a scalar value corresponding to the value displayed in the four-digit image. For training,  $n$  of those four-digit images are separately processed by the CNN and then sorted by the relaxed sorting network as shown in Figure 3.1. Based on the permutation matrix produced by the sorting network and the ground truth ranking, the training objective is computed (Equation (3.12)) and the CNN is updated. At test time, we forward single images of four-digit numbers from the test data set. For evaluation, the discrete rankings of the predicted values are compared to the rankings of their ground truth. Note that the  $n$  used for testing and evaluation can be independent of the  $n$  used for training because the  $n$  images are processed independently.

**SVHN** Since the multi-digit MNIST data set is an artificial data set, we also evaluate our technique on the SVHN data set [180]. This data set comprises house numbers collected from Google Street View and provides a larger variety wrt. different fonts and formats than the MNIST data set. We use the published “Format 1” and preprocess it as described by Goodfellow *et al.* [181], cropping the centered multi-digit numbers with a boundary of 30%, resizing it to a resolution of  $64 \times 64$ , and then selecting  $54 \times 54$  pixels at a random location. As SVHN contains 1 – 5 digit numbers, we can avoid the concatenation and use the original images directly. Example images are . Otherwise, the experimental setup is as for the four-digit MNIST data set.

**Network Architecture** For the MNIST sorting task, we use the same convolutional neural network (CNN) architecture as Grover *et al.* [58] and Cuturi *et al.* [56] to allow for comparability. This architecture consists of two convolutional layers with a kernel size of  $5 \times 5$ , 32 and 64 channels respectively, each followed by a ReLU and MaxPool layer; this is (after flattening) followed by a fully connected layer with a size of 64, a ReLU layer, and a fully connected output layer mapping to a scalar. For the SVHN task, we use a network with four convolutional layers with a kernel size of  $5 \times 5$  and (32, 64, 128, 256) filters, each followed by a ReLU and a max-pooling layer with stride  $2 \times 2$ ; followed by a fully connected layer with size 64, a ReLU, and a layer with output size 1.

**Evaluation Metrics** For evaluation, discrete rankings based on the scalar predictions are computed and compared to the discrete ground truth rankings. As in previous works, we use the evaluation metrics of exact match (EM) of the predicted ranking, and fraction of element-wise correct ranks (EW) in the predicted ranking. For EM and EW, we follow Grover *et al.* [58] and Cuturi *et al.* [56], and use the same  $n$  for training and evaluation. However, this can be a problem in the context of large input sets as these evaluation metrics become unreliable as  $n$  increases. For example, the difficulty of exact matches rises with the factorial of  $n$ , which is why they become too sparse to allow for valid conclusions for large  $n$ . To allow for a comparison of the performance independent of the number of elements  $n$  used for training, for some experiments, we also evaluate the models based on the EM accuracy for  $n = 5$  (EM5). That is, the network can be trained with an arbitrary  $n$ , but the evaluation is done for  $n = 5$ .

**Training Settings** We use the Adam optimizer [34] with a learning rate of  $10^{-3.5}$ . We use a constant batch size of 100 as in previous works unless denoted otherwise. Note that, although  $\lambda$  is chosen as a constant value for all  $n$ , a higher accuracy is possible when optimizing  $\lambda$  for each  $n$  separately. The fast sorting and ranking method by Blondel *et al.* [64] does not produce permutation matrices; thus, we cannot use the cross-entropy loss, and therefore, we use the mean-squared-error loss between predicted and ground truth ranks.

### 3.5.1 The Activation Replacement Trick

We start by evaluating the performance of differentiable sorting networks with the ART. For this, we train each model for up to  $10^6$  steps. Furthermore, we set  $\lambda = 0.25$  and use an inverse temperature of two times the number of layers ( $\beta = 2n$  for odd-even and  $\beta = (\log_2 n)(1 + \log_2 n)$  for bitonic.)

**Comparison to State-of-the-Art (MNIST)** We first compare our approach to the methods proposed by Grover *et al.* [58] and Cuturi *et al.* [56]. Here, we follow the setting that the  $n$  used for evaluation is the same as the  $n$  used for training. The evaluation is shown in Table 3.3. We report results for exact match, correct ranks, and EM5, respectively. For the odd-even architecture, we compare results for the original  $n \in \{3, 5, 7, 9, 15\}$ . Our approach outperforms current methods on all metrics and input set sizes. In addition, we extend the original benchmark set sizes by  $n \in \{2, 4, 8, 16, 32\}$ , allowing for the canonical version of the bitonic sorting network which requires input size of powers of 2. We apply  $n \in \{2, 4, 8, 16, 32\}$  to the odd-even as well as the bitonic sorting network. In this direct comparison, we can see that the bitonic and the odd-even architectures perform similarly. Notably, the EM and EW accuracies do not always correlate as can be seen for  $n = 32$ . Here, the EM accuracy is greater for the bitonic network and the EW accuracy is greater for the odd-even network. We attribute this to the odd-even network’s gradients causing swaps of neighbors while the bitonic network’s gradients provide a holistic approach favoring exact matches.

**SVHN** The results in Table 3.4 show that the real-world SVHN task is significantly harder than the MNIST task. On this data set, differentiable sorting networks are also better than current methods on all metrics and input set sizes. Here, the performance of odd-even and bitonic are similar. Notably, the EM5 accuracy is largest for the bitonic sorting network at  $n = 32$ , which demonstrates that the method benefits from longer input sets. Further, for  $n \in \{8, 16, 32\}$ , the bitonic sorting network marginally outperforms the odd-even sorting network on all metrics.

### Large-Scale Sorting and Ranking Supervision

We are interested in the effect of training with larger input set sizes  $n$ . As the bitonic sorting network requires significantly fewer layers than odd-even and is (thus) faster, we use the bitonic sorting network for the scalability experiments. Here, we evaluate for  $n = 2^k, k \in \{5, 6, 7, 8, 9, 10\}$  on the MNIST sorting benchmark, comparing the EM5 accuracy as shown in Table 3.5.

**Table 3.3:** Results for comparing the ART to state-of-the-art [56, 58] using the same network architectures averaged over 5 runs. The first three rows are duplicated from Cuturi *et al.* [56]. The metrics are (EM | EW | EM5). The models are trained for up to  $10^6$  steps.

| MNIST                                   | $n = 3$     |             |             | $n = 5$     |             |             | $n = 7$     |             |             | $n = 9$     |             |             | $n = 15$    |             |             |
|-----------------------------------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|
| Stoch. NeuralSort                       | 92.0        | 94.6        |             | 79.0        | 90.7        | 79.0        | 63.6        | 87.3        |             | 45.2        | 82.9        |             | 12.2        | 73.4        |             |
| Det. NeuralSort                         | 91.9        | 94.5        |             | 77.7        | 90.1        | 77.7        | 61.0        | 86.2        |             | 43.4        | 82.4        |             | 9.7         | 71.6        |             |
| Optimal Transport                       | 92.8        | 95.0        |             | 81.1        | 91.7        | 81.1        | 65.6        | 88.2        |             | 49.7        | 84.7        |             | 12.6        | 74.2        |             |
| Fast Sort & Rank                        | 90.6        | 93.5        | 73.5        | 71.5        | 87.2        | 71.5        | 49.7        | 81.3        | 70.5        | 29.0        | 75.2        | 69.2        | 2.8         | 60.9        | 67.4        |
| $\sigma \circ \varphi$ : ART (Odd-Even) | <b>95.2</b> | <b>96.7</b> | <b>86.1</b> | <b>86.3</b> | <b>93.8</b> | <b>86.3</b> | <b>75.4</b> | <b>91.2</b> | <b>86.4</b> | <b>64.3</b> | <b>89.0</b> | <b>86.7</b> | <b>35.4</b> | <b>83.7</b> | <b>87.6</b> |

| MNIST                                   | $n = 2$ |      |      | $n = 4$ |      |      | $n = 8$ |      |      | $n = 16$ |      |      | $n = 32$ |      |      |
|-----------------------------------------|---------|------|------|---------|------|------|---------|------|------|----------|------|------|----------|------|------|
| $\sigma \circ \varphi$ : ART (Odd-Even) | 98.1    | 98.1 | 84.3 | 90.5    | 94.9 | 85.5 | 63.6    | 87.9 | 83.6 | 31.7     | 82.8 | 87.3 | 1.7      | 69.1 | 86.7 |
| $\sigma \circ \varphi$ : ART (Bitonic)  | 98.1    | 98.1 | 84.0 | 91.4    | 95.3 | 86.7 | 70.6    | 90.3 | 86.9 | 30.5     | 81.7 | 86.6 | 2.7      | 67.3 | 85.4 |

**Table 3.4:** Results for training on the SVHN data set averaged over 5 runs. The metrics are (EM | EW | EM5). The models are trained for up to  $10^6$  steps.

| SVHN                                    | $n = 2$     |             |             | $n = 4$     |             |             | $n = 8$     |             |             | $n = 16$   |             |             | $n = 32$ |             |             |
|-----------------------------------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|------------|-------------|-------------|----------|-------------|-------------|
| Det. NeuralSort                         | 90.1        | 90.1        | 39.9        | 61.4        | 78.1        | 45.4        | 15.7        | 62.3        | 48.5        | 0.1        | 45.7        | 51.0        | 0.0      | 29.9        | 52.7        |
| Optimal Transport                       | 85.5        | 85.5        | 25.9        | 57.6        | 75.6        | 41.6        | 19.9        | 64.5        | 51.7        | 0.3        | 47.7        | 53.8        | 0.0      | 29.4        | 53.3        |
| Fast Sort & Rank                        | 93.4        | 93.4        | 57.6        | 58.0        | 75.8        | 41.5        | 8.6         | 52.7        | 34.4        | 0.3        | 36.5        | 41.6        | 0.0      | 14.0        | 27.5        |
| $\sigma \circ \varphi$ : ART (Odd-Even) | 93.4        | 93.4        | 58.0        | <b>74.8</b> | <b>85.5</b> | <b>62.6</b> | 35.2        | 73.5        | 63.9        | 1.8        | 54.4        | 62.3        | 0.0      | 36.6        | 62.6        |
| $\sigma \circ \varphi$ : ART (Bitonic)  | <b>93.8</b> | <b>93.8</b> | <b>58.6</b> | 74.4        | 85.3        | 62.1        | <b>38.3</b> | <b>75.1</b> | <b>66.8</b> | <b>3.9</b> | <b>59.6</b> | <b>66.8</b> | 0.0      | <b>42.4</b> | <b>67.7</b> |

For this experiment, we consider inverse temperature values of  $\beta \in \{30, 32.5, 35, 37.5, 40\}$  and report the mean, best, and worst over all inverse temperature values for each  $n$ . We set  $\lambda$  to 0.4 as this allows for stable training with  $n > 128$ . To keep the evaluation feasible, we reduce the number of steps during training to  $10^4$ , compared to the  $10^6$  iteration in Table 3.3. Again, we use the Adam optimizer with a learning rate of  $10^{-3.5}$ .

In the first two columns of Table 3.5, we show a head-to-head comparison with the setting in Table 3.3 with  $\lambda = 0.25$  and  $\lambda = 0.4$  for  $n = 32$ . Trained for  $10^6$  steps, the EM5 accuracy is 85.4%, while it is 78.2% after  $10^4$  steps. Increasing  $\lambda$  from 0.25 to 0.4 improves the EM5 accuracy from 78.2% to 80.97%.

This also demonstrates that already at this scale, a larger  $\lambda$ , i.e., a stronger activation replacement trick, can improve the overall accuracy of a bitonic sorting network compared to training with  $\lambda = 0.25$ .

As the size  $n$  of training tuples increases, this also increases the overall number of observed images during training. Therefore, in the left half of Table 3.5, we consider the accuracy for a constant total of observed images per iteration, i.e., for  $n \times \text{batch size} = 4096$  (e.g., for  $n = 32$  this results in a batch size of 128, while for  $n = 1024$ , the batch size is only 4). In the right half of Table 3.5, we consider a constant batch size of 4.

With increasing  $n$ , the accuracy of our model increases for a constant number of observed images, even though it has to operate on very small batch sizes. This shows that training with larger ordered sets results in better accuracy. This suggests that, if possible, larger  $n$  should be prioritized over larger batch sizes and that good results can be achieved by using the largest possible  $n$  for the available data to learn from all available information.

**Table 3.5:** Results the ART for large  $n$  measured using the EM5 metric with fixed number of samples as well as a fixed batch size. Independent of the batch size, the model always performs better for larger  $n$ . Trained for  $10^4$  steps & averaged over 10 runs.

| $\lambda$      | 0.25         | 0.4          | 0.4          | 0.4          | 0.4          | 0.4          | 0.4          | 0.4          | 0.4          | 0.4          | 0.4          | 0.4          | 0.4          |
|----------------|--------------|--------------|--------------|--------------|--------------|--------------|--------------|--------------|--------------|--------------|--------------|--------------|--------------|
| $n$            | 32           | 32           | 64           | 128          | 256          | 512          | 1024         | 32           | 64           | 128          | 256          | 512          | 1024         |
| batch size     | 128          | 128          | 64           | 32           | 16           | 8            | 4            | 4            | 4            | 4            | 4            | 4            | 4            |
| $\beta = 30$   | <b>78.20</b> | 79.89        | 81.25        | <b>82.50</b> | <b>82.05</b> | 82.50        | <b>82.80</b> | 71.08        | <b>75.88</b> | 79.43        | <b>81.46</b> | 82.98        | <b>82.80</b> |
| $\beta = 32.5$ | 76.98        | 79.62        | <b>81.66</b> | 80.15        | 81.87        | 82.64        | 81.63        | <b>72.31</b> | 75.59        | <b>79.71</b> | 81.36        | <b>82.99</b> | 81.63        |
| $\beta = 35$   | 77.45        | 80.93        | 81.26        | 80.72        | 81.42        | 81.51        | 81.15        | 71.15        | 75.73        | 78.81        | 79.32        | 82.30        | 81.15        |
| $\beta = 37.5$ | 76.40        | 80.02        | 80.05        | 81.50        | 80.05        | <b>82.67</b> | 80.07        | 70.69        | 75.80        | 79.11        | 80.64        | 82.70        | 80.07        |
| $\beta = 40$   | 77.69        | <b>80.97</b> | 80.23        | 81.55        | 79.75        | 81.89        | 81.15        | 70.20        | 74.67        | 78.14        | 80.06        | 81.39        | 81.15        |
| mean           | 77.35        | 80.29        | 80.89        | 81.28        | 81.03        | 82.24        | 81.36        | 71.09        | 75.53        | 79.04        | 80.57        | 82.47        | 81.36        |
| best $\beta$   | <b>78.20</b> | <b>80.97</b> | <b>81.66</b> | <b>82.50</b> | <b>82.05</b> | <b>82.67</b> | <b>82.80</b> | <b>72.31</b> | <b>75.88</b> | <b>79.71</b> | <b>81.46</b> | <b>82.99</b> | <b>82.80</b> |
| worst $\beta$  | 76.40        | 79.62        | 80.05        | 80.15        | 79.75        | 81.51        | 80.07        | 70.20        | 74.67        | 78.14        | 79.32        | 81.39        | 80.07        |

### Ablation Study and Hyperparameter Sensitivity

To assess the impact of the proposed activation replacement trick (ART), we evaluate both architectures with and without ART at  $\lambda = 0.25$  in Table 3.6. The accuracy improves by using the ART for small as well as for large  $n$ . For large  $n$ , the activation replacement trick has a greater impact on the performance of both architectures. For the bitonic sorting network at  $n = 32$ , even a stronger ART with  $\lambda = 0.4$  is beneficial as demonstrated in Table 3.5. In Figure 3.8, we evaluate both differentiable sorting networks for varying ART intensities  $\lambda$ . Here, performance increases with larger  $\lambda$ s (i.e., with a stronger ART). For  $\lambda > 0.5$ , the performance drops as  $\varphi$  converges to a discrete step function for  $\lambda \rightarrow 1$ .

**Table 3.6:** Ablation Study: Evaluation of the ART ( $\lambda = 0$  vs.  $\lambda = 0.25$ ) for  $n = 4$  and  $n = 32$  on the MNIST and the SVHN data set using odd-even and bitonic sorting networks. The displayed metric is EW.

| Setting / $\lambda$ | $n = 4$ |             | $n = 32$ |             |
|---------------------|---------|-------------|----------|-------------|
|                     | 0       | 0.25        | 0        | 0.25        |
| Odd-E (MNIST)       | 94.5    | <b>94.9</b> | 61.5     | <b>69.1</b> |
| Bitonic (MNIST)     | 93.6    | <b>95.3</b> | 62.8     | <b>67.3</b> |
| Odd-E (SVHN)        | 77.3    | <b>85.5</b> | 28.5     | <b>36.6</b> |
| Bitonic (SVHN)      | 78.1    | <b>85.3</b> | 35.0     | <b>42.4</b> |

### 3.5.2 Monotonic Differentiable Sorting Networks

In this section, we evaluate the performance of monotonic differentiable sorting networks in comparison to existing non-monotonic differentiable sorting methods as well as non-monotonic differentiable sorting networks like the ART variant.

To understand the behavior of the proposed monotonic functions compared to the logistic sigmoid function, we evaluate all sigmoid functions for different inverse temperatures  $\beta$  during training. We investigate eight settings: odd-even networks for  $n \in \{3, 5, 7, 9, 15, 32\}$  and a bitonic sorting network with  $n \in \{15, 32\}$  on the MNIST data set. Notably, there are 15 layers in the bitonic sorting networks with  $n = 32$ , while the odd-even networks for  $n = 15$  also has 15 layers. We display the results of this evaluation in Figure 3.9. Note that, here, we train for  $10^5$  training steps to reduce the computational cost.

We observe that the optimal inverse temperature depends on the number of layers, rather than the overall number of samples  $n$ . This can be seen when comparing the peak accuracy of each function for the odd-even sorting network for different  $n$  and thus for different numbers of layers. The bitonic network for  $n = 32$  (last row, right) has the same number of layers as  $n = 15$  in the odd-even network (third row, left). Here, the peak performances for each sigmoid function fall within the same range, whereas the peak performances for the odd-even network for  $n = 32$  (bottom left) are shifted almost an order of magnitude to the right. For all configurations, the

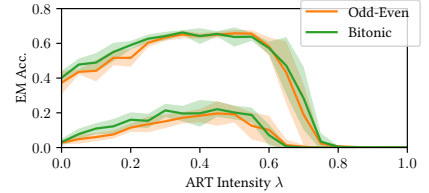
**Figure 3.8:** Comparing different ART strengths  $\lambda$  for  $n = 8$  (top) and  $n = 16$  (bottom). Training with  $\lambda \leq 0.5$  is stable.

**Table 3.7:** Results on the four-digit MNIST and SVHN tasks using the same architecture as previous works [2, 56, 58, 64]. The metric is the proportion of rankings correctly identified (EM), and the value in parentheses is the proportion of individual element ranks correctly identified (EW). All results are averaged over 5 runs. SVHN w/  $n = 32$  is omitted to reduce the carbon impact of the evaluation. We train each differentiable sorting network-based model for up to  $2 \cdot 10^5$  steps for MNIST, and for up to  $10^6$  steps for SVHN.

| MNIST                                  | $n = 3$     | $n = 5$     | $n = 7$     | $n = 9$     | $n = 15$    | $n = 32$   | $n = 16$<br>(bitonic) | $n = 32$<br>(bitonic) |
|----------------------------------------|-------------|-------------|-------------|-------------|-------------|------------|-----------------------|-----------------------|
| NeuralSort                             | 91.9 (94.5) | 77.7 (90.1) | 61.0 (86.2) | 43.4 (82.4) | 9.7 (71.6)  | 0.0 (38.8) | —                     | —                     |
| Sinkhorn Sort                          | 92.8 (95.0) | 81.1 (91.7) | 65.6 (88.2) | 49.7 (84.7) | 12.6 (74.2) | 0.0 (41.2) | —                     | —                     |
| Fast Sort & Rank                       | 90.6 (93.5) | 71.5 (87.2) | 49.7 (81.3) | 29.0 (75.2) | 2.8 (60.9)  | —          | —                     | —                     |
| Diffsort (Logistic)                    | 92.0 (94.5) | 77.2 (89.8) | 54.8 (83.6) | 37.2 (79.4) | 4.7 (62.3)  | 0.0 (56.3) | 10.8 (72.6)           | 0.3 (63.2)            |
| Diffsort (Log. w/ ART)                 | 94.3 (96.1) | 83.4 (92.6) | 71.6 (90.0) | 56.3 (86.7) | 23.5 (79.4) | 0.5 (64.9) | 19.0 (77.5)           | 0.8 (63.0)            |
| $f_{\mathcal{R}}$ : Reciprocal Sigmoid | 94.4 (96.1) | 85.0 (93.3) | 73.4 (90.7) | 60.8 (88.1) | 30.2 (81.9) | 1.0 (66.8) | 28.7 (82.1)           | 1.3 (68.0)            |
| $f_{\mathcal{C}}$ : Cauchy CDF         | 94.2 (96.0) | 84.9 (93.2) | 73.3 (90.5) | 63.8 (89.1) | 31.1 (82.2) | 0.8 (63.3) | 29.0 (82.1)           | 1.6 (68.1)            |
| $f_{\mathcal{O}}$ : Optimal Sigmoid    | 94.6 (96.3) | 85.0 (93.3) | 73.6 (90.7) | 62.2 (88.5) | 31.8 (82.3) | 1.4 (67.9) | 28.4 (81.9)           | 1.4 (67.7)            |
| SVHN                                   | $n = 3$     | $n = 5$     | $n = 7$     | $n = 9$     | $n = 15$    | —          | $n = 16$<br>(bitonic) | —                     |
| Diffsort (Logistic)                    | 76.3 (83.2) | 46.0 (72.7) | 21.8 (63.9) | 13.5 (61.7) | 0.3 (45.9)  | —          | 1.2 (50.6)            | —                     |
| Diffsort (Log. w/ ART)                 | 83.2 (88.1) | 64.1 (82.1) | 43.8 (76.5) | 24.2 (69.6) | 2.4 (56.8)  | —          | 3.4 (59.2)            | —                     |
| $f_{\mathcal{R}}$ : Reciprocal Sigmoid | 85.7 (89.8) | 68.8 (84.2) | 53.3 (80.0) | 40.0 (76.3) | 13.2 (66.0) | —          | 11.5 (64.9)           | —                     |
| $f_{\mathcal{C}}$ : Cauchy CDF         | 85.5 (89.6) | 68.5 (84.1) | 52.9 (79.8) | 39.9 (75.8) | 13.7 (66.0) | —          | 12.2 (65.6)           | —                     |
| $f_{\mathcal{O}}$ : Optimal Sigmoid    | 86.0 (90.0) | 67.5 (83.5) | 53.1 (80.0) | 39.1 (76.0) | 13.2 (66.3) | —          | 10.6 (66.8)           | —                     |

**Table 3.8:** Inverse temperatures  $\beta$  used for Table 3.7, which correspond to the optima in Figure 3.9.

| Method                 | $n$                | 3                  | 5                 | 7                 | 9                | 15                | 32                 | 16               | 32                  |
|------------------------|--------------------|--------------------|-------------------|-------------------|------------------|-------------------|--------------------|------------------|---------------------|
|                        | odd-even / bitonic | oe                 | oe                | oe                | oe               | oe                | oe                 | bi               | bi                  |
| $\sigma$               | Logistic           | 79                 | 30                | 33                | 54               | 32                | 128                | 43               | 8                   |
| $\sigma \circ \varphi$ | Log. w/ ART        | 15                 | 20                | 13                | 34               | 16                | 29                 | 28               | 26                  |
| $f_{\mathcal{R}}$      | Reciprocal Sigmoid | 14                 | 60                | 69                | 44               | 120               | 1140               | 124              | 76                  |
| $f_{\mathcal{C}}$      | Cauchy CDF         | 14.5 $\pi$<br>45.6 | 51 $\pi$<br>160.2 | 71 $\pi$<br>223.1 | 15 $\pi$<br>47.1 | 40 $\pi$<br>125.7 | 169 $\pi$<br>531.0 | 12 $\pi$<br>37.7 | 48.5 $\pi$<br>152.4 |
| $f_{\mathcal{O}}$      | Optimal Sigmoid    | 6                  | 20                | 29                | 32               | 25                | 124                | 17               | 25                  |

proposed sigmoid functions for monotonic sorting networks improve over the standard logistic sigmoid function, as well as the ART.

**Comparison to Other Differentiable Sorting Methods** We compare the proposed functions to other state-of-the-art approaches using the same network architecture and training setup as used in previous works, as well as among themselves. We report the results in Table 3.7. Here, for each setting, we use the inverse temperature, which performed best in Figure 3.9; the concrete choices of  $\beta$  can be found in Table 3.8. The proposed monotonic differentiable sorting networks outperform current state-of-the-art methods by a considerable margin. Especially for those cases where more samples needed to be sorted, the gap between monotonic sorting nets and other techniques grows with larger  $n$ .

Comparing the three proposed functions among themselves, we observe that for odd-even networks on MNIST, the error-optimal function  $f_{\mathcal{O}}$  performs best. This is because here the approximation error is small. However, for the more complex bitonic sorting networks,  $f_{\mathcal{C}}$  (Cauchy) performs better than  $f_{\mathcal{O}}$ . This is because  $f_{\mathcal{O}}$  does not provide a higher-order smoothness and is only  $C^1$  smooth, while the Cauchy function  $f_{\mathcal{C}}$  is analytic and  $C^\infty$  smooth.

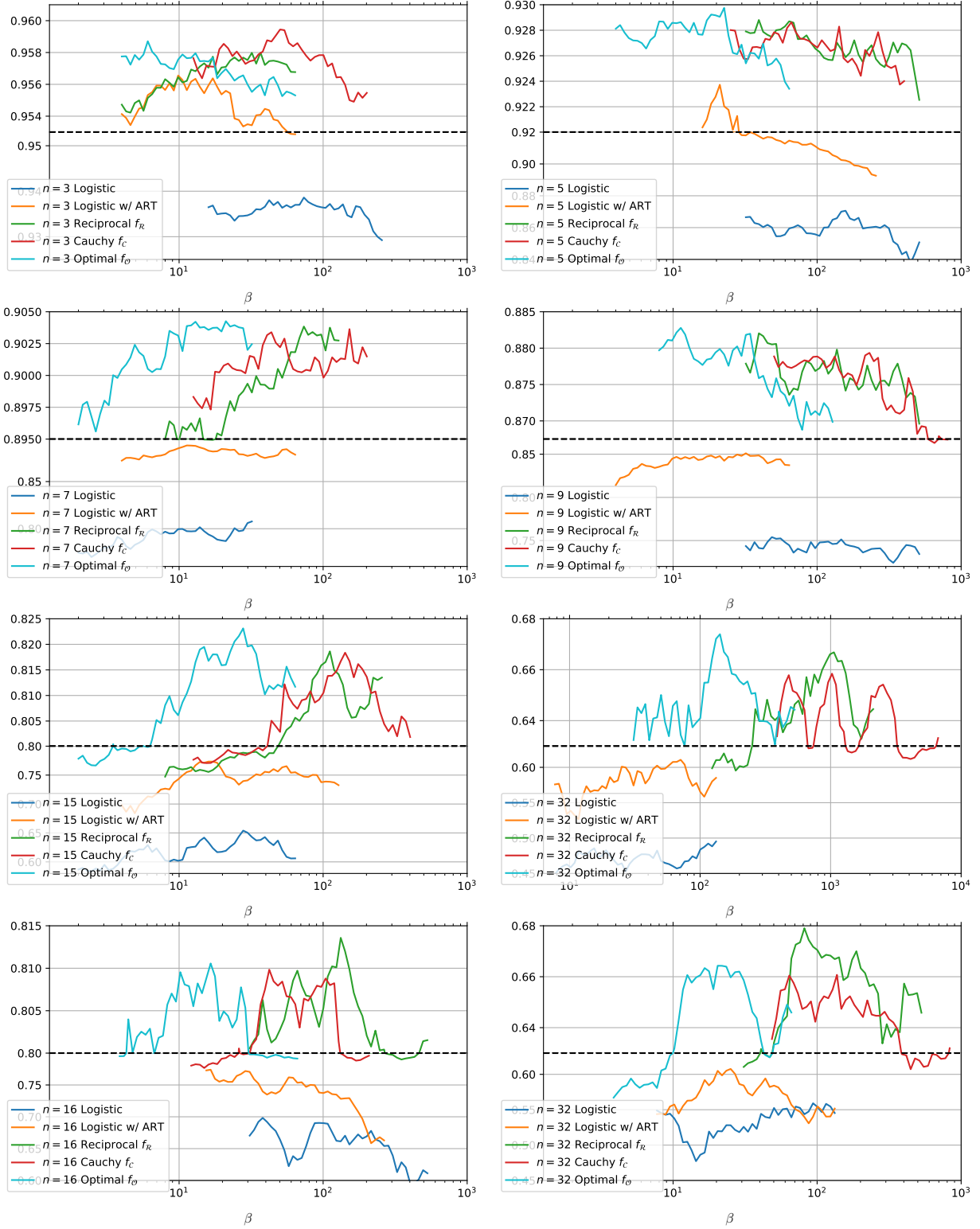


**Figure 3.9:** Evaluating different sigmoid functions on the sorting MNIST task for ranges of different inverse temperatures  $\beta$ . The metric is the proportion of individual element ranks correctly identified (EW). In all settings, the monotonic sorting networks clearly outperform the non-monotonic ones. *First 3 rows:* Odd-Even sorting networks with  $n \in \{3, 5, 7, 9, 15, 32\}$ . *Last row:* Bitonic network with  $n \in \{16, 32\}$ . For small  $n$  such as 3, Cauchy performs best because it has a low error but is smooth at the same time. For larger  $n$  such as 15 and 32, the optimal sigmoid function (wrt. error)  $f_O$  performs better because it, while not being smooth, has the smallest possible approximation error which is more important for deeper networks. For the bitonic network with its more complex structure at  $n = 16$  and  $n = 32$  (last row), the reciprocal sigmoid  $f_R$  performs best.

**Table 3.9:** Runtimes, memory requirements, and number of layers for sorting  $n$  elements. Runtimes reported for an Nvidia GTX 1070. We include NeuralSort [58], FastRank [64], and OT Sort [56].

| $n$   | Differentiable Odd-Even Sort |             |        |          | Differentiable Bitonic Sort  |             |        |          | NeuralSort                  |             | FastRank                     | OT Sort       |
|-------|------------------------------|-------------|--------|----------|------------------------------|-------------|--------|----------|-----------------------------|-------------|------------------------------|---------------|
|       | GPU                          | CPU         | Memory | # Layers | GPU                          | CPU         | Memory | # Layers | GPU                         | CPU         | CPU                          | CPU           |
| 4     | <b>69 ns</b>                 | 1.9 $\mu$ s | 1KB    | 4        | <b>52 ns</b>                 | 1.3 $\mu$ s | 840B   | 3        | <b>145 ns</b>               | 7.1 $\mu$ s | <b>189 <math>\mu</math>s</b> | <b>1.0 ms</b> |
| 16    | <b>1.2 <math>\mu</math>s</b> | 54 $\mu$ s  | 42KB   | 16       | <b>759 ns</b>                | 40 $\mu$ s  | 28KB   | 10       | <b>396 ns</b>               | 11 $\mu$ s  | <b>215 <math>\mu</math>s</b> | <b>7.5 ms</b> |
| 32    | <b>7.4 <math>\mu</math>s</b> | 309 $\mu$ s | 315KB  | 32       | <b>3.5 <math>\mu</math>s</b> | 159 $\mu$ s | 152KB  | 15       | <b>969 ns</b>               | 13 $\mu$ s  | <b>303 <math>\mu</math>s</b> | <b>17 ms</b>  |
| 128   | <b>493 <math>\mu</math>s</b> | 19 ms       | 20.2MB | 128      | <b>97 <math>\mu</math>s</b>  | 5 ms        | 4.1MB  | 28       | <b>12 <math>\mu</math>s</b> | 177 $\mu$ s | <b>834 <math>\mu</math>s</b> | <b>55 ms</b>  |
| 1 024 | <b>660 ms</b>                | 31 s        | 4.9GB  | 1 024    | <b>15 ms</b>                 | 1.7 s       | 549MB  | 55       | <b>1.2 ms</b>               | 11 ms       | <b>4.8 ms</b>                | <b>754 ms</b> |

### 3.5.3 Runtime and Memory Analysis

Finally, we report the runtime and memory consumption of differentiable sorting networks in Table 3.9. For GPU runtimes, we use a native CUDA implementation and measure the time and memory for sorting  $n$  input elements including forward and backward pass. For CPU runtimes, we use a PyTorch [26] implementation. For a small number of input elements, the odd-even and bitonic sorting networks have around the same time and memory requirements, while for larger numbers of input elements, bitonic is much faster than odd-even.

The asymptotic runtime of differentiable odd-even sort is in  $\mathcal{O}(n^3)$  and for bitonic sort the runtime is in  $\mathcal{O}(n^2(\log n)^2)$ . Note that, for this, the matrix multiplication in Equation (3.11) is a sparse matrix multiplication. We also report runtimes for other differentiable sorting and ranking methods. For large  $n$ , we empirically confirm that FastRank [64] is the fastest method, i.e., because it produces only output ranks / sorted output values and not differentiable permutation matrices. Note that differentiable sorting networks also produce sorted output values. Computing only sorted output values is significantly faster than computing the full differentiable permutation matrices; however, for the effective cross-entropy training objective, differentiable permutation matrices are necessary.

## Conclusion

In this chapter, we presented differentiable sorting networks for training based on sorting and ranking supervision. To this end, we relaxed the discrete min and max operators necessary for pairwise swapping in traditional sorting network architectures via perturbation. We proposed an activation replacement trick to avoid the problems of vanishing gradients as well as blurred values for deep sorting networks and showed that it is possible to robustly sort and rank even long sequences on large input sets of up to at least 1024 elements. Further, we addressed and analyzed monotonicity and error-boundedness in differentiable sorting and ranking operators. Specifically, we focused on differentiable sorting networks and presented a family of sigmoid functions that preserve monotonicity and bound approximation errors in differentiable sorting networks. This makes the sorting functions quasiconvex, and we empirically observe that the resulting method outperforms the state-of-the-art in differentiable sorting supervision. In the next chapter, we build on these ideas to present differentiable top- $k$  learning.

## Differentiable Top- $k$

After discussing differentiable sorting and ranking in the previous chapter, this chapter covers the related top- $k$  operator and proposes novel relaxations of it. We use differentiable top- $k$  in a new learning setting, which we term differentiable top- $k$  classification learning.

Classification is one of the core disciplines in machine learning and computer vision. The advent of classification problems with hundreds or even thousands of classes let the top- $k$  classification accuracy (i.e., one of the top- $k$  classes has to be the correct class) be established as an important metric. Usually, models are trained to optimize the top-1 accuracy; and top-5 etc. are used for evaluation only. Some works [69, 182] have challenged this idea and proposed top- $k$  losses, such as a smooth top-5 margin loss. These methods have demonstrated superior robustness over the established top-1 softmax cross-entropy in the presence of additional label noise [69]. In standard classification settings, however, these methods have so far not shown improvements over the established top-1 softmax cross-entropy.

In this chapter, we challenge the idea of selecting a single top- $k$  metric such as top-1 or top-5 for defining the loss. Instead, we propose to specify  $k$  to be drawn from a distribution  $P_K$ , which may or may not depend on the confidence of specific data points or on the class label. Examples for distributions  $P_K$  are  $[.5, 0, 0, 0, .5]$  (50% top-1 and 50% top-5),  $[.1, 0, 0, 0, .9]$  (10% top-1 and 90% top-5), and  $[.2, .2, .2, .2, .2]$  (20% top- $k$  for each  $k$  from 1 to 5). Note that, when  $k$  is drawn from a distribution, this is done sampling-free as we can compute the expectation value in closed form.

Conventionally, given scores returned by a neural network, softmax produces a probability distribution over the top-1 rank. Recent advances in differentiable sorting and ranking, as discussed in the previous chapter, provide methods for generalizing this to probability distributions over all ranks represented by a matrix  $\mathbf{P}$ . Based on differentiable ranking, multiple differentiable top- $k$  operators have recently been proposed. They found applications in differentiable  $k$ -nearest neighbor algorithms [20, 58], differentiable beam search [93], attention mechanisms [93], and differentiable image patch selection [51]. In these areas, integrating differentiable top- $k$  improved results considerably by creating a more natural end-to-end learning setting. However, to date, none of the differentiable top- $k$  operators have been employed as neural network losses for top- $k$  classification learning with  $k > 1$ .

Building on differentiable sorting and ranking methods, we propose a new family of differentiable top- $k$  classification losses where  $k$  is drawn from a probability distribution. We find that our top- $k$  losses improve not only top- $k$  accuracies, but also top-1 accuracy on multiple learning tasks.

We empirically evaluate our method using four differentiable sorting and ranking methods on the CIFAR-100 [183], ImageNet-1K [184], and the ImageNet-21K-P [185] data sets. Using CIFAR-100, we demonstrate the capabilities of our losses to train models from scratch. On ImageNet-1K, we demonstrate that our losses are capable of fine-tuning existing models

|     |                                   |    |
|-----|-----------------------------------|----|
| 4.1 | Related Work . . . . .            | 52 |
| 4.2 | Differentiable Top- $k$ . . . . . | 53 |
| 4.3 | Top- $k$ Learning . . . . .       | 55 |
| 4.4 | Experiments . . . . .             | 57 |

and achieve a new state-of-the-art for publicly available models on both top-1 and top-5 accuracy. We benchmark our method on multiple recent models and demonstrate that our proposed method consistently outperforms the baselines for the best two differentiable sorting and ranking methods. With ImageNet-21K-P, where many classes overlap (but only one is the ground truth), we demonstrate that our losses are scalable to more than 10 000 classes and achieve improvements of over 1% with only last layer fine-tuning.

Overall, while the performance improvements on fine-tuning are rather limited (because we retrain only the classification head), they are consistent and can be achieved without the large costs of training from scratch. The absolute 0.2% improvement that we achieve on the ResNeXt-101 32x48d WSL top-5 accuracy corresponds to an error reduction by approximately 10%, and can be achieved at much less than the computational cost of (re-)training the full model in the first place.

## 4.1 Related Work

We structure the related work into three broad sections: works that derive and apply differentiable top- $k$  operators, works that use ranking and top- $k$  training objectives in general, and works that present classic selection networks.

### 4.1.1 Differentiable Top- $k$ Operators

As top- $k$  can be reduced to sorting and ranking, the methods presented in the previous chapter can be used to obtain a differentiable top- $k$  operator. However, in this chapter, we discuss how to improve over the vanilla top- $k$  operators derived from differentiable sorting and ranking.

Grover *et al.* [58] include an experiment where they use the NeuralSort differentiable top- $k$  operator for  $k$ -nearest-neighbor ( $k$ NN) learning. Cuturi *et al.* [56], Blondel *et al.* [64], and Petersen *et al.* [2] each apply their differentiable sorting and ranking methods to top- $k$  supervision with  $k = 1$ .

Xie *et al.* [93] propose a differentiable top- $k$  operator based on optimal transport and the Sinkhorn algorithm [57]. They apply their method to  $k$ NN learning, differential beam search with sorted soft top- $k$ , and top- $k$  attention for machine translation. Cordonnier *et al.* [51] use perturbed optimizers [37] to derive a differentiable top- $k$  operator, which they use for differentiable image patch selection. Lee *et al.* [85] propose using NeuralSort for a differentiable top- $k$  operator to produce differentiable ranking metrics for recommender systems. Goyal *et al.* [94] propose a continuous top- $k$  operator for differentiable beam search. Pietruszka *et al.* [95] propose the differentiable successive halving top- $k$  operator to approximate the normalized Chamfer Cosine Similarity ( $nCCS@k$ ).

### 4.1.2 Ranking and Top- $k$ Training Objectives

Fan *et al.* [186] propose the “average top- $k$ ” loss, an aggregate loss that averages over the  $k$  largest individual losses of a training data set. They apply this aggregate loss to SVMs for classification tasks. Note that this is not a differentiable top- $k$  loss. Here, the top- $k$  operator is not differentiable, and instead used for deciding which data points’ losses are aggregated into the loss.

Lapin *et al.* [67, 182] propose relaxed top- $k$  surrogate error functions for multiclass SVMs. Inspired by learning-to-rank losses, they propose top- $k$  calibration, a top- $k$  hinge loss, a top- $k$  entropy loss, as well as a truncated top- $k$  entropy loss. They apply their method to multiclass SVMs and learn via stochastic dual coordinate ascent (SDCA).

Berrada *et al.* [69] build on these ideas and propose smooth loss functions for deep top- $k$  classification. Their differentiable surrogate top- $k$  loss achieves good performance on the CIFAR-100 and ImageNet1K tasks. While their method does not improve performance on the raw data sets in comparison to the strong Softmax Cross-Entropy baseline, in settings of label noise and data set subsets, they improve classification accuracy. Specifically, with label noise of 20% or more on CIFAR-100, they improve top-1 and top-5 accuracy and for subsets of ImageNet1K of up to 50% they improve top-5 accuracy. In contrast to [69], our method improves classification accuracy in unmodified settings. In our experiments, for the special case of  $k$  being a concrete integer and not being drawn from a distribution, we provide comparisons to the smooth top- $k$  surrogate loss.

Yang *et al.* [187] provide a theoretical analysis of top- $k$  surrogate losses as well as produce a new surrogate top- $k$  loss, which they evaluate in synthetic data experiments.

A related idea is set-valued classification, where a set of labels is predicted. We refer to Chzhen *et al.* [188] for an extensive overview. We note that our goal is not to predict a set of labels, but instead we return a score for each class corresponding to a ranking, where only one class can correspond to the ground truth.

### 4.1.3 Selection Networks

Previous selection networks have been proposed by Wah *et al.* [189], Zazon-Ivry *et al.* [190], and Karpiński *et al.* [191], among others. All of these are based on classic divide-and-conquer sorting networks, which recursively sort subsequences and merge them. In selection networks, during merging, only the top- $k$  elements are merged instead of the full (sorted) subsequences. In comparison to those earlier works, we propose a new class of selection networks, which achieve tighter bounds (for  $k \ll n$ ), and relax them.

## 4.2 Differentiable Top- $k$

The differentiable sorting algorithms discussed in Chapter 3 produce relaxed permutation matrices of size  $n \times n$ . However, for top- $k$  classification learning, we require only the top  $k$  rows for the number  $k$  of top-ranked classes to

consider. Here,  $k$  is the largest  $k$  that is considered for the objective, i.e., where  $P_K(k) > 0$ . As  $n \gg k$ , producing a  $k \times n$  matrix instead of a  $n \times n$  matrix is much faster.

**NeuralSort and SoftSort** For NeuralSort and SoftSort, it is possible to simply compute only the top rows, as the algorithm is defined row-wise.

**Optimal Transport / Sinkhorn Sort** For optimal transport / Sinkhorn sort [56], it is not directly possible to improve the runtime, as the full matrix is required in each Sinkhorn iteration. However, Xie *et al.* [93] proposed an optimal transport and Sinkhorn-based differentiable top- $k$  operator, which computes a  $2 \times n$  matrix where the first row corresponds to the top- $k$  elements and the second row correspond to the remaining elements. As this formulation does not directly support distinguishing between the placements of the top- $k$  elements among each other, which is necessary for the proposed top- $k$  learning objective, we use the SinkhornSort algorithm by Cuturi *et al.* [56], which we also used in the previous chapter.

**Differentiable Sorting Networks** For differentiable sorting networks, it is possible to reduce the cost from  $\mathcal{O}(n^2 \log^2(n))$  to  $\mathcal{O}(nk \log^2(n))$  via a bi-directional evaluation. Here, it is important to note the shape and order of multiplications for obtaining  $P$ . As we only need those elements, which are (after the last layer of the sorting network) at the top  $k$  ranks that we want to consider, we can omit all remaining rows of the permutation matrix of the last layer (layer  $t$ ) and thus it is only of size  $(k \times n)$ .

$$\underbrace{(k \times n)}_P = \underbrace{(k \times n)}_{\text{layer } t} \underbrace{(n \times n)}_{\text{layer } t-1} \dots \underbrace{(n \times n)}_{\text{layer } 1} \quad (4.1)$$

Note that during the execution of the sorting network,  $P$  is conventionally computed from layer 1 to layer  $t$ , i.e., from right to left. If we computed it in this order, we would only save a tiny fraction of the computational cost and only during the last layer. Thus, we propose to execute the differentiable sorting network, save the values that populate the (sparse)  $n \times n$  layer-wise permutation matrices, and compute  $P$  in a second pass from the back to the front, i.e., from layer  $t$  to layer 1, or from left to right in Equation (4.1). This allows executing  $t$  dense-sparse matrix multiplications with dense  $k \times n$  matrices and sparse  $n \times n$  matrices instead of dense  $n \times n$  and sparse  $n \times n$  matrices. With this, we reduce the asymptotic complexity from  $\mathcal{O}(n^2 \log^2(n))$  to  $\mathcal{O}(nk \log^2(n))$ .

#### 4.2.1 Differentiable Top- $k$ Networks

As only the top- $k$  rows of a relaxed permutation matrix are required for top- $k$  classification learning, it is possible to further improve the efficiency of computing the top- $k$  probability distribution via differentiable sorting networks by improving the underlying sorting network or (in general) comparator network architecture. Thus, we propose differentiable top- $k$  networks, which relax selection networks in analogy to how differentiable sorting networks relax sorting networks. Selection networks are networks that select only the top- $k$  out of  $n$  elements [167].

## Splitter Selection Networks

In this context, we propose splitter selection networks (SSN), a novel class of selection networks that requires only  $\mathcal{O}(\log n)$  layers (instead of the  $\mathcal{O}(\log^2 n)$  layers for sorting networks), which makes top- $k$  supervision with differentiable top- $k$  networks more efficient and reduces the error (which is introduced in each layer). SSNs follow the idea that the input is split into locally sorted sublists and then all wires that are not candidates to be among the global top- $k$  can be eliminated. For example, for  $n = 1024, k = 5$ , SSNs require only 22 layers, while the best previous selection network requires 34 layers and full sorting (with a bitonic network) even requires 55 layers. For  $n = 10450, k = 5$  (i.e., for ImageNet-21K-P), SSNs require 27 layers, the best previous selection network requires 50 layers, and full sorting requires 105 layers. In addition, the layers of SSNs are less computationally expensive than those of the bitonic sorting network. Further details on SSNs, as well as their full construction, can be found in [4].

Concluding, the contribution of differentiable top- $k$  networks is two-fold: first, we propose a novel kind of selection networks that needs fewer layers, and second, we relax those similarly to differentiable sorting networks.

## 4.3 Top- $k$ Learning

In this section, we start by introducing our objective, elaborate its exact formulation, and then build on differentiable sorting principles to efficiently approximate the objective. A visual overview of the loss architecture is given in Figure 4.1.

The goal of top- $k$  learning is to extend the learning criterion from only accepting exact (top-1) predictions to accepting  $k$  predictions among which the correct class has to be. In its general form, for top- $k$  learning,  $k$  may differ for each application, class, data point, or a combination thereof. For example, in one case, one may want to rank 5 predictions and assign a score that depends on the rank of the true class among these ranked predictions, while, in another case, one may want to obtain 5 predictions but not care about their order. In yet another case, such as image classification, one may want to enforce a top-1 accuracy on images from the “person” super-class, but resign to a top-3 accuracy for the “animal” super-class, as it may have more ambiguities in class-labels. We model this by a random variable  $K$ , following a distribution  $P_K$  that describes the relative importance of different values  $k$ . The discrete distribution  $P_K$  is either a marginalized distribution for a given setting (such as the discrete uniform distribution), or a conditional distribution for each class, data point, etc. This allows specifying a marginalized or conditional distribution  $k \sim P_K$ . This generalizes the ideas of conventional top-1 supervision (usually softmax cross-entropy) and top- $k$  supervision for a  $k$  like  $k = 5$  (usually based on surrogate top- $k$  margin/hinge losses like [69, 182]) and unifies them.

The objective of top- $k$  learning is maximizing the probability of accepted predictions of the model  $f_\theta$  on data  $X, y \sim \mathcal{D}$  given marginal distribution  $P_K$  (or conditional  $P_{K|X,y}$  if it depends on the class  $y$  and/or data point  $X$ ).

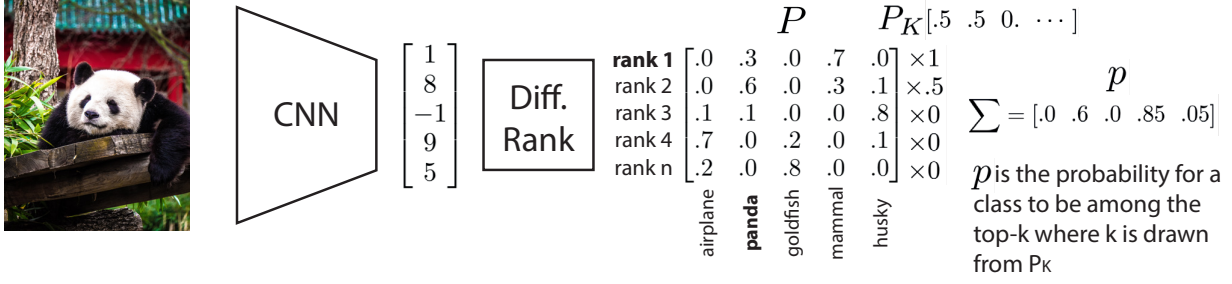


**Figure 4.1:** Overview of the proposed architecture: A CNN predicts scores for an image, which are then ranked by a differentiable ranking algorithm returning the probability distribution for each rank in matrix  $P$ . The rows of this distribution correspond to ranks, and the columns correspond to the respective classes. In the example, we use a 50% top-1 and 50% top-2 loss, i.e.,  $P_K = [.5, .5, 0, 0, 0]$ . Here, the  $k$ th value refers to the top- $k$  component, which is satisfied if the prediction is at *any* of rank-1 to rank- $k$ . Thus, the weights for the different ranks can be computed via a cumulative sum and are  $[1, .5, 0, 0, 0]$ . The correspondingly weighted sum of rows of  $P$  yields the probability distribution  $p$ , which can then be used in a cross-entropy loss.

In the following,  $P_{k,y}$  is the predicted probability of  $y$  being the  $k$ th-best prediction for data point  $X$ .

$$\arg \max_{\theta} \mathbb{E}_{X,y \sim \mathcal{D}} \left[ \log \left( \mathbb{E}_{k \sim P_K} \left[ \sum_{m=1}^k P_{m,y}(f_{\theta}(X)) \right] \right) \right] \quad (4.2)$$

To evaluate the probability of  $y$  to be the top-1 prediction, we can simply use  $\text{softmax}_y(f_{\theta}(X))$ . However,  $k > 1$  requires more consideration. Here, we require probability scores  $P_{k,c}$  for the  $k$ th prediction over classes  $c \in \mathbb{C}$ , where  $\sum_{c=1}^n P_{k,c} = 1$  (i.e.,  $P$  is row-stochastic) and ideally additionally  $\sum_{k=1}^n P_{k,c} = 1$  (i.e.,  $P$  is also column stochastic and thus doubly-stochastic.) With this, we can optimize our model by minimizing the following loss

$$\mathcal{L}(X, y) = -\log \left( \sum_{k=1}^n P_K(k) \left( \sum_{m=1}^k P_{m,y}(f_{\theta}(X)) \right) \right), \quad (4.3)$$

which is the cross-entropy over the probabilities that the true class is among the top- $k$  class for each possible  $k$ . Note that  $\sum_{k=1}^n P_K(k) = 1$ .

To compute  $P_{k,c}$ , we require a function mapping from a vector of real-valued scores to an (ideally) doubly-stochastic matrix  $P$ . The most suitable for this are the differentiable relaxations of the sorting and ranking functions, which produce differentiable permutation matrices  $P$ , which we introduced in Chapter 3. We build on these approximations to propose instances of top- $k$  learning losses and extend differentiable sorting networks to differentiable top- $k$  networks, as just finding the top- $k$  scores is computationally cheaper than sorting all elements and reduces the approximation error.

### 4.3.1 Implementation Details

Despite those performance improvements, evaluating the differentiable ranking operators still requires a considerable amount of computational effort for large numbers of classes. Especially if the number  $n$  of elements to be ranked is  $n = 1\,000$  (ImageNet-1K) or even  $n > 10\,000$  (ImageNet-21K-P), the differentiable ranking operators can dominate the overall computational costs. In addition, for large numbers  $n$  of elements to be ranked, the performance of differentiable ranking operators decreases as differentially ranking more elements naturally introduces larger errors [2, 56, 58, 59]. Thus, we reduce the number of outputs to be ranked differentially by only considering

those classes (for each input) that have a score among the top- $m$  scores. For this, we make sure that the ground truth class is among those top- $m$  scores, by replacing the lowest of the top- $m$  scores by the ground truth class, if necessary. For  $n = 1000$ , we choose  $m = 16$ , and for  $n > 10\,000$ , we choose  $m = 50$ . We find that this greatly improves training performance.

Because the differentiable ranking operators are (by their nature of being differentiable) only approximations to the hard ranking operator, they each have their characteristics and inconsistencies. Thus, for training models from scratch, we replace the top-1 component of the loss by the regular softmax, which has a better and more consistent behavior. This guides the other loss if the differentiable ranking operator behaves inconsistently. To avoid the top- $k$  components affecting the guiding softmax component and avoid probabilities greater than 1 in  $p$ , we can separate the cross-entropy into a mixture of the softmax cross-entropy (for the top-1 component) and the top- $k$  cross-entropy (for the top- $k \geq 2$  components) as follows:

$$\begin{aligned} \mathcal{L}(X, y) = & P_K(1) \cdot \text{SoftmaxCELoss}(f_\theta(X), y) \\ \text{sm+top-k} & \\ & - (1 - P_K(1)) \cdot \log \left( \sum_{k=2}^n P_K(k) \left( \sum_{m=1}^k P_{m,y}(f_\theta(X)) \right) \right) \end{aligned} \quad (4.4)$$

## 4.4 Experiments

We evaluate the proposed top- $k$  classification loss for four differentiable ranking operators on CIFAR-100 [183], ImageNet-1K [184], as well as the winter 2021 edition of ImageNet-21K-P [185]. We use CIFAR-100, which can be considered a small-scale data set with only 100 classes, to train a ResNet18 model [192] from scratch and show the impact of the proposed loss function on the top-1 and top-5 accuracy. In comparison, ImageNet-1K and ImageNet-21K-P provide rather large-scale data sets with 1 000 and 10 450 classes, respectively. To avoid the unreasonable carbon-footprint of training many models from scratch, we decided to exclusively use publicly available backbones for all ImageNet experiments. This has the additional benefit of allowing more settings, making our work easily reproducible, and allowing to perform multiple runs on different seeds to improve the statistical significance of the results. For ImageNet-1K, we use two publicly available state-of-the-art architectures as backbones: First, the (four) ResNeXt-101 WSL architectures by Mahajan *et al.* [193], which were pretrained in a weakly-supervised fashion on a billion-scale data set from Instagram. Second, the Noisy Student EfficientNet-L2 [194], which was pretrained on the unlabeled JFT-300M data set [195]. For ResNeXt-101 WSL, we extract 2 048-dimensional embeddings and for the Noisy Student EfficientNet-L2, we extract 5 504-dimensional embeddings of ImageNet-1K and fine-tune on them.

We apply the proposed loss in combination with various available differentiable sorting and ranking approaches, namely NeuralSort, SoftSort, SinkhornSort, and DiffSortNets. To determine the optimal (inverse) temperature for each differentiable sorting method, we perform a grid search at a resolution of factor 2. For training, we use the Adam optimizer [34]. For training on CIFAR-100 from scratch, we train for up to 200 epochs with a batch size of 100 at a learning rate of  $10^{-3}$ . For ImageNet-1K, we train for up to 100 epochs at a batch size of 500 and a learning rate of  $10^{-4.5}$ .

**Table 4.1:** CIFAR-100 results for training a ResNet18 from scratch. The metrics are Top-1 | Top-5 accuracy averaged over 2 seeds.

| Method                    | $P_K$                  | CIFAR-100    |              |
|---------------------------|------------------------|--------------|--------------|
| <i>Baselines</i>          |                        |              |              |
| Softmax                   | $([1, 0, 0, 0, 0])$    | 61.27        | 85.31        |
| Smooth top- $k$ loss [69] | $([0, 0, 0, 0, 1])$    | 53.07        | 85.23        |
| Top-5 NeuralSort          | $[0, 0, 0, 0, 1]$      | 22.58        | 84.41        |
| Top-5 SoftSort            | $[0, 0, 0, 0, 1]$      | 1.01         | 5.09         |
| Top-5 SinkhornSort        | $[0, 0, 0, 0, 1]$      | 55.62        | <b>87.04</b> |
| Top-5 DiffSortNets        | $[0, 0, 0, 0, 1]$      | 52.81        | 84.21        |
| <i>Ours</i>               |                        |              |              |
| Top- $k$ NeuralSort       | $[.2, .2, .2, .2, .2]$ | 61.46        | 86.03        |
| Top- $k$ SoftSort         | $[.2, .2, .2, .2, .2]$ | 61.53        | 82.39        |
| Top- $k$ SinkhornSort     | $[.2, .2, .2, .2, .2]$ | 61.89        | <b>86.94</b> |
| Top- $k$ DiffSortNets     | $[.2, .2, .2, .2, .2]$ | <b>62.00</b> | 86.73        |

For ImageNet-21K-P, we train for up to 40 epochs at a batch size of 500 and a learning rate of  $10^{-4}$ . We use early stopping and found that these settings lead to convergence in all settings (except those that also diverge with other settings). As baselines, we use the respective original models, softmax cross-entropy, as well as learning with the smooth surrogate top- $k$  loss [69].

#### 4.4.1 Training from Scratch

We start by demonstrating that the proposed loss can be used to train a network from scratch. As a reference baseline, we train a ResNet18 from scratch on CIFAR-100. In Table 4.1, we compare the baselines (i.e., top-1 softmax, the smooth top-5 loss [69], as well as “pure” top-5 losses using four differentiable sorting and ranking methods) with our top- $k$  loss with  $k \sim [.2, .2, .2, .2, .2]$ .

We find that training with top-5 alone—in some cases—slightly improves the top-5 accuracy but has a substantially worse top-1 accuracy. Here, we note that the smooth top-5 loss [69], top-5 Sinkhorn [56], and top-5 DiffSort [2] achieve moderate performance. Notably, Sinkhorn [56] outperforms the softmax baseline on the top-5 metric, while NeuralSort and SoftSort are less stable and yield worse results especially on top-1 accuracy.

By using our loss that corresponds to drawing  $k$  from  $[.2, .2, .2, .2, .2]$ , we can achieve substantially improved results, especially also on the top-1 accuracy metric. Using the DiffSortNets yields the best results on the top-1 accuracy and Sinkhorn yields the best results on the top-5 accuracy. Note that, here, also NeuralSort and SoftSort achieve good results in this setting, which can be attributed to our loss with  $k \sim [.2, .2, .2, .2, .2]$  being more robust to inconsistencies and outliers in the used differentiable sorting method. Interestingly, top-5 SinkhornSort achieves the best performance on the top-5 metric, which suggests that SinkhornSort is a very robust differentiable sorting method as it does not require additional top- $k$  components. Nevertheless, it is advisable to include other top- $k$  components as the model trained purely on top-5 exhibits poor top-1 performance.

#### 4.4.2 Fine-Tuning

In this section, we discuss the results for fine-tuning on ImageNet-1K and ImageNet-21K-P. In Table 4.2, we find a very similar behavior to training

| Method                    | $P_K$               | ImgNet-1K                    | ImgNet-21K-P                |
|---------------------------|---------------------|------------------------------|-----------------------------|
| <i>Baselines</i>          |                     |                              |                             |
| Softmax                   | $([1, 0, 0, 0, 0])$ | 86.06   97.795               | 39.29   69.63               |
| Smooth top- $k$ loss [69] | $([0, 0, 0, 0, 1])$ | 85.15   97.540               | 34.03   65.56               |
| Top-5 NeuralSort          | $[0, 0, 0, 0, 1]$   | 33.37   94.748               | 15.87   33.81               |
| Top-5 SoftSort            | $[0, 0, 0, 0, 1]$   | 18.23   94.965               | 33.61   69.82               |
| Top-5 SinkhornSort        | $[0, 0, 0, 0, 1]$   | 85.65   <b>97.991</b>        | 36.93   69.80               |
| Top-5 DiffSortNets        | $[0, 0, 0, 0, 1]$   | 69.05   97.389               | 35.96   69.76               |
| <i>Ours</i>               |                     |                              |                             |
| Top- $k$ NeuralSort       | $[.5, 0, 0, 0, .5]$ | 86.30   97.896               | 37.85   68.08               |
| Top- $k$ SoftSort         | $[.5, 0, 0, 0, .5]$ | 86.26   97.963               | 39.93   70.63               |
| Top- $k$ SinkhornSort     | $[.5, 0, 0, 0, .5]$ | <b>86.29</b>   <b>97.971</b> | 39.85   70.56               |
| Top- $k$ DiffSortNets     | $[.5, 0, 0, 0, .5]$ | 86.24   97.937               | <b>40.22</b>   <b>70.88</b> |

**Table 4.2:** ImageNet-1K and ImageNet-21K-P results for fine-tuning the head of ResNeXt-101 32x48d WSL [193]. The metrics are Top-1 | Top-5 accuracy averaged over 10 seeds for ImageNet-1K and 2 seeds for ImageNet-21K-P.

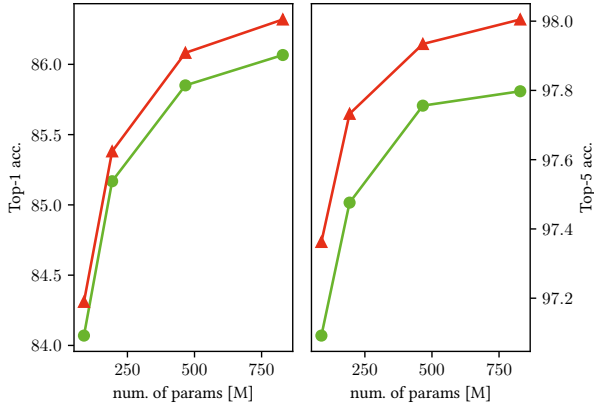


**Figure 4.2:** ImageNet-1K accuracy improvements for all ResNeXt-101 WSL model sizes (32x8d, 32x16d, 32x32d, 32x48d). Green (●) is the original model and red (▲) is with top- $k$  fine-tuning.

from scratch on CIFAR-100. Specifically, we find that the accuracy improves by drawing  $k$  from a distribution for training. An exception is (again) SinkhornSort, where focusing only on top-5 yields the best top-5 accuracy on ImageNet-1K, but the respective model exhibits poor top-1 accuracy. Overall, we find that drawing  $k$  from a distribution improves performance in all cases.

To demonstrate that the improvements also translate to different backbones, we show the improvements on all four model sizes of ResNeXt-101 WSL (32x8d, 32x16d, 32x32d, 32x48d) in Figure 4.2. Also, here, our method improves the model in all settings.

#### 4.4.3 Impact of the Distribution $P_K$ and Differentiable Sorting Methods

We start by demonstrating the impact of  $P_K$ , which is the distribution from which we draw  $k$ . Let us first consider the case where  $k$  is 5 with probability  $\alpha$  and 1 with probability  $1-\alpha$ , i.e.,  $P_K = [1-\alpha, 0, 0, 0, \alpha]$ . In Figure 4.3 (left), we demonstrate the impact that arises from changing  $\alpha$ , i.e., transitioning from a pure top-1 loss to a pure top-5 loss, in the case of fine-tuning ResNeXt-101 WSL with our loss using the SinkhornSort algorithm. Increasing the weight of the top-5 component does not only increase the top-5 accuracy but also improves the top-1 accuracy up to around 60% top-5; when using only  $k = 5$ , the top-1 accuracy drastically decays as the incentive for the true class to be at the top-1 position vanishes (or is only indirectly given by being among the top-5.) While the top-5 accuracy in this plot is best for a pure top-5 loss, this generally only applies to the Sinkhorn algorithm and

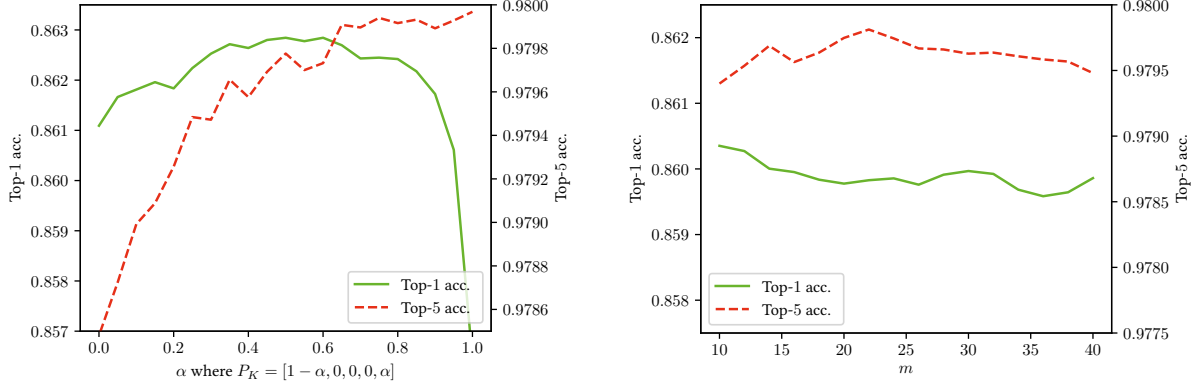


**Figure 4.3:** Effects of varying the ratio between top-1 and top-5 (left) and varying the size of differentially ranked subset  $m$  (right). Both experiments build on the differentiable Sinkhorn ranking algorithm [56]. On the left,  $m = 16$ , on the right,  $\alpha = 0.75$ . Averaged over 5 runs.

overall training is more stable if a pure top-5 is avoided. This can also be seen in Tables 4.1 and 4.2.

In Tables 4.3 and 4.4, we consider additional settings with all differentiable ranking methods. Specifically, we compare four notable settings:  $[.5, 0, 0, 0, .5]$ , i.e., equally weighted top-1 and top-5;  $[.25, 0, 0, 0, .75]$  and  $[.1, 0, 0, 0, .9]$ , i.e., top-5 has larger weights;  $[.2, .2, .2, .2, .2]$ , i.e., the case of having an equal weight of 0.2 for top-1 to top-5. The  $[.5, 0, 0, 0, .5]$  setting is a rather canonical setting which usually performs well on both metrics, while the others tend to favor top-5. In the  $[.5, 0, 0, 0, .5]$  setting, all sorting methods improve upon the softmax baseline on both top-1 and top-5 accuracy. When increasing the weight of the top-5 component, the top-5 generally improves while top-1 decays.

Here we find a core insight of this chapter: the best performance cannot be achieved by optimizing top- $k$  for only a single  $k$ , but instead, drawing  $k$  from a distribution improves performance on all metrics.

Comparing the differentiable ranking methods, we can find the overall trend that SoftSort outperforms NeuralSort, and that SinkhornSort as well as DiffSortNets perform best. We can see that some sorting algorithms are more sensitive to the overall  $P_K$  than others: Whereas SinkhornSort [56] and DiffSortNets [2] continuously outperform the softmax baseline, NeuralSort [58] and SoftSort [59] tend to collapse when over-weighting the top-5 components.

Comparing the performance on the medium-scale ImageNet-1K to the larger ImageNet-21K-P in Table 4.2, we observe a similar pattern. Here, again, using the top- $k$  component alone is not enough to significantly increase accuracy, but combining top-1 and top- $k$  components helps to improve accuracy on both reported metrics. While NeuralSort struggles in this large-scale ranking problem and stays below the softmax baseline, DiffSortNets [2] provide the best top-1 and top-5 accuracy with 40.22% and 70.88%, respectively.

We note that we do not claim that all settings (especially all differentiable sorting methods) improve the classification performance on all metrics. Instead, we include all methods and also additional settings to demonstrate the capabilities and limitations of each differentiable sorting method.

Overall, it is notable that SinkhornSort achieves the overall most robust training behavior, while also being by far the slowest sorting method and

| Method / $P_K$        | [.5, 0, 0, 0, .5]           | [.25, 0, 0, 0, .75] | [.1, 0, 0, 0, .9]    | [.2, .2, .2, .2, .2]  |
|-----------------------|-----------------------------|---------------------|----------------------|-----------------------|
| <i>CIFAR-100</i>      |                             |                     |                      |                       |
| NeuralSort            | 61.12   86.47               | 61.07   87.23       | 52.57   85.76        | 61.46   86.03         |
| SoftSort              | 61.17   83.95               | 61.05   83.10       | 58.16   79.26        | 61.53   82.39         |
| SinkhornSort          | 61.34   86.38               | 61.50   86.68       | 57.35   86.34        | 61.89   86.94         |
| DiffSortNets          | 60.07   86.44               | 61.57   86.51       | 61.74   <b>87.22</b> | <b>62.00</b>   86.73  |
| <i>ImageNet-1K</i>    |                             |                     |                      |                       |
| NeuralSort            | 86.30   97.896              | 84.26   95.410      | 84.32   94.889       | 85.75   97.865        |
| SoftSort              | 86.26   97.963              | 86.16   97.954      | 87.30   95.915       | 86.18   97.979        |
| SinkhornSort          | <b>86.29</b>   97.971       | 86.24   97.989      | 86.18   97.987       | 86.22   97.989        |
| DiffSortNets          | 86.24   97.937              | 86.15   97.936      | 86.04   97.980       | 86.21   <b>98.003</b> |
| <i>ImageNet-21K-P</i> |                             |                     |                      |                       |
| NeuralSort            | 37.85   68.08               | 36.16   67.60       | 33.02   67.29        | 37.09   67.90         |
| SoftSort              | 39.93   70.63               | 39.08   70.27       | 37.78   70.07        | 39.68   70.57         |
| SinkhornSort          | 39.85   70.56               | 39.21   70.41       | 38.42   70.12        | 39.22   70.49         |
| DiffSortNets          | <b>40.22</b>   <b>70.88</b> | 39.56   70.58       | 38.48   70.25        | 39.69   70.69         |

**Table 4.3:** CIFAR-100 results for different distributions  $P_K$  for training a ResNet18 from scratch. The metrics are Top-1 | Top-5 accuracy averaged over 2 seeds.

**Table 4.4:** ImageNet-1K and ImageNet-21K-P results for different distributions  $P_K$  for fine-tuning the head of ResNeXt-101 32x48d WSL [193]. The metrics are Top-1 | Top-5 accuracy averaged over 10 seeds for ImageNet-1K and 2 seeds for ImageNet-21K-P.

thus potentially slowing down training drastically, especially when the task is only fine-tuning. SinkhornSort tends to require more Sinkhorn iterations towards the end of training. DiffSortNets are considerably faster; especially, it is possible to only compute the top- $k$  probability matrices and because of our advances for more efficient selection networks.

#### 4.4.4 Differentiable Ranking Set Size $m$

We consider how accuracy is affected by varying the number of scores  $m$  to be differentially ranked. Generally, the runtime of differentiable top- $k$  operators depends between linearly and cubic on  $m$ ; thus it is important to choose an adequate value for  $m$ . The choice of  $m$  between 10 and 40 has only a moderate impact on the accuracy, as can be seen in Figure 4.3 (right). However, when setting  $m$  to large values such as 1 000 or larger, we observe that the differentiable sorting methods tend to become unstable. We note that we did not specifically tune  $m$ , and that better performance can be achieved by fine-tuning  $m$ , as displayed in the plot.

#### 4.4.5 Comparison to the State-of-the-Art

We compare the proposed method to current state-of-the-art methods in Table 4.5. We focus on methods that are publicly available and build upon two of the best-performing models, namely Noisy Student EfficientNet-L2 [194], and ResNeXt-101 32x48d WSL [193]. Using both backbones, we achieve improvements on both metrics, and when fine-tuning on the Noisy Student EfficientNet-L2, we achieve a new state-of-the-art for publicly available models.

**Significance Tests** To evaluate the significance of the results, we perform a t-test (with a significance level of 0.01). We find that our model is significantly better than the original model on both top-1 and top-5 accuracy metrics. Comparing to the observed accuracies of the baseline (88.33 | 98.65), DiffSortNets are significantly better ( $p=0.00001$  | 0.00005). Comparing to

**Table 4.5:** ImageNet-1K result comparison to state-of-the-art. Among the overall best-performing differentiable sorting / ranking methods, almost all results in reasonable settings outperform their respective baseline on Top-1 and Top-5 accuracy. For publicly available models / backbones, we achieve a new state-of-the-art for top-1 and top-5 accuracy. Our results are averaged over 10 runs.

| Method                               | Public  | Top-1        | Top-5        |
|--------------------------------------|---------|--------------|--------------|
| ResNet50                             | [192] ✓ | 79.26        | 94.75        |
| ResNet152                            | [192] ✓ | 80.62        | 95.51        |
| ResNeXt-101 32x48d WSL               | [193] ✓ | 85.43        | 97.57        |
| ViT-L/16                             | [196] ✓ | 87.76        | —            |
| Noisy Student EfficientNet-L2        | [194] ✓ | <b>88.35</b> | <b>98.65</b> |
| BiT-L                                | [197] ✗ | 87.54        | 98.46        |
| CLIP (w/ Noisy Student EffNet-L2)    | [198] ✗ | ≈ 88.4       | —            |
| ViT-H/14                             | [196] ✗ | 88.55        | —            |
| ALIGN (EfficientNet-L2)              | [199] ✗ | 88.64        | <b>98.67</b> |
| Meta Pseudo Labels (EfficientNet-L2) | [200] ✗ | 90.20        | ≈ 98.8       |
| ViT-G/14                             | [201] ✗ | 90.45        | —            |
| CoAtNet-7                            | [202] ✗ | <b>90.88</b> | —            |
| ResNeXt-101 32x48d WSL               |         | 86.06        | 97.80        |
| Top- $k$ SinkhornSort                |         | <b>86.22</b> | 97.99        |
| Top- $k$ DiffSortNets                |         | 86.21        | <b>98.00</b> |
| Noisy Student EfficientNet-L2        |         | 88.33        | 98.65        |
| Top- $k$ SinkhornSort                |         | 88.32        | 98.66        |
| Top- $k$ DiffSortNets                |         | <b>88.37</b> | <b>98.68</b> |

the reported accuracies of the baseline (88.35 | 98.65), DiffSortNets are also significantly better ( $p=0.00087$  |  $0.00005$ ).

## Conclusion

We discussed differentiable top- $k$  operators and proposed differentiable top- $k$  networks, which are more efficient than differentiable sorting networks through a computational trick and by using the proposed splitter selection network architecture. Further, we presented a novel loss, which relaxes the assumption of using a fixed  $k$  for top- $k$  classification learning. We performed an array of experiments to explore different top- $k$  classification learning settings and achieved a state-of-the-art on ImageNet for publicly available models.

# Differentiable Rendering

After covering differentiable sorting, ranking, and top- $k$ , this chapter studies differentiable rendering and presents a generalized family of differentiable renderers. We continue to build on the foundation presented in Chapter 2, especially Section 2.5. We discuss from scratch which components are necessary for differentiable rendering and formalize the requirements for each component. We instantiate our general differentiable renderer, which generalizes existing differentiable renderers like SoftRas and DIB-R, with an array of different smoothing distributions to cover a large spectrum of reasonable settings. We evaluate an array of differentiable renderer instantiations on the popular ShapeNet 3D reconstruction benchmark and analyze the implications of our results. Surprisingly, the simple uniform distribution yields the best overall results when averaged over 13 classes; however, in general, the optimal choice of distribution heavily depends on the task.

In the past years, many differentiable renderers have been published. These include the seminal differentiable mesh renderer OpenDR [65], the Neural 3D Mesh Renderer [63], and SoftRas [46] among many others. Using a differentiable renderer enables a multitude of computer vision applications, such as human pose estimation [203], camera intrinsics estimation [204], 3D shape optimization [63], 3D reconstruction [46, 47, 63], and 3D style transfer [63].

A fundamental difference between different classes of differentiable renderers is the choice of the underlying 3D representation. In this chapter, we focus on differentiable 3D mesh renderers [46, 47, 63, 65]; however, the aspects that we investigate could also be applied to other differentiable rendering concepts, such as rendering voxels [105], point clouds [106], surfels [107], signed distance functions [108, 109], and other implicit representations [110, 111].

Differentiable mesh renderers can be constructed in different ways: either using an exact and hard renderer with approximate surrogate gradients or using an approximate renderer with natural gradients. Loper *et al.* [65] and Kato *et al.* [63] produce approximate surrogate gradients for their differentiable renderer, while their forward rendering is hard. In contrast, other differentiable renderers approximate the forward rendering in such a way that they produce a natural gradient. This can be achieved by modeling or approximating a renderer under a probabilistic perturbation, which is continuous and makes the renderer differentiable. For that, in Chapter 2, we used the logistic distribution. Rhodin *et al.* [205] model it with a Gaussian distribution, while Liu *et al.* [46] model it with the square root of a logistic distribution, and Chen *et al.* [47] with the exponential distribution. While this variational interpretation of perturbing by a respective distribution is not stressed in some of these papers [46, 47], we believe it is important because it explicitly allows comparing the characteristics of the differentiable renderers. Moreover, the methods that only approximate gradients can also be seen as approximately modelling a perturbation: the gradient computed for the Neural 3D Mesh Renderer [63] is approximately a perturbation by a uniform distribution. Note that, here, the solutions for rendering under

|     |                                                   |    |
|-----|---------------------------------------------------|----|
| 5.1 | Related Work . . . . .                            | 65 |
| 5.2 | The Generalized Differentiable Renderer . . . . . | 66 |
| 5.3 | Instantiations of GenDR                           | 69 |
| 5.4 | Experiments . . . . .                             | 71 |

perturbations are obtained analytically in closed form and without sampling.

In this chapter, we introduce a generalized differentiable renderer (GenDR). By choosing an appropriate probability distribution, we can (at least approximately) recover the above differentiable mesh renderers, which shows that a core distinguishing aspect of differentiable renderers is the type of distributions that they model. The choice of probability distribution herein is directly linked to the sigmoid (i.e., S-shaped) function used for rasterization. For example, a Heaviside sigmoid function corresponding to the Dirac delta distribution yields a conventional non-differentiable renderer, while a logistic sigmoid function of squared distances corresponds to the square root of a logistic distribution. Herein, the sigmoid function is the CDF of the corresponding distribution. In this chapter, we select and present an array of distributions and examine their theoretical properties.

Another aspect of approximate differentiable renderers is their aggregation function, i.e., the function that aggregates the occupancy probabilities of all faces for each pixel. Existing differentiable renderers commonly aggregate the probabilities via the probabilistic sum ( $\perp^P(a, b) = a + b - ab$  or  $1 - \prod_{t \in T} 1 - p_t$ ), which corresponds to the probability that at least one face covers the pixel assuming that probabilities  $p_t$  for each triangle  $t$  are stochastically independent (cf. Eq. 4 in [46] or Eq. 6 in [47]). In the field of real-valued logics and adjacent fields, this is well-known as a T-conorm, a relaxed form of the logical ‘or’. Two examples of other T-conorms are the maximum T-conorm  $\perp^M(a, b) = \max(a, b)$  and the Einstein sum  $\perp^E(a, b) = (a + b) / (1 + ab)$ , which models the relativistic addition of velocities. We generalize our differentiable renderer to use any continuous T-conorm and present a variety of suitable T-conorms.

In total, the set of resulting concrete instances arising from our generalized differentiable renderer and the proposed choices amounts to 1 242 concrete differentiable renderers. We extensively benchmark all of them on a shape optimization task and a camera pose estimation task. Further, we evaluate the best performing and most interesting instances on the popular ShapeNet [164] 13 class single-view 3D reconstruction experiment [63]. Here, we also include those instances that approximate other existing differentiable renderers. We note that we do not introduce an alternative shading technique, and rely on existing blended shaders instead.

In the domain of differentiable rendering, different from the work discussed in this chapter, we have also published two papers. Specifically, the differentiable renderer Pix2Vex [11] and style-agnostic differentiable renderer-based 3D reconstruction via domain adaptation [10].

Summarizing the contributions of this chapter, (i) we propose a generalized differentiable mesh renderer; (ii) we identify existing differentiable renderers (approximately) as instances of our generalized renderer; (iii) we propose a variety of suitable sigmoid functions and T-conorms and group them by their characteristics; (iv) we extensively benchmark 1 242 concrete differentiable renderers, analyze which characteristics and families of functions lead to a good performance, and find that the best choice heavily depends on the task, class, or characteristics of the data.

## 5.1 Related Work

The related work can be classified into those works that present differentiable renderers and those which apply them, although there is naturally also significant overlap. For additional details on differentiable rendering approaches, cf. the survey by Kato *et al.* [52].

**Analytical Differentiable Renderers** The first large category of differentiable renderers are those which produce approximate gradients in an analytical and sampling-free way. This can either happen by surrogate gradients during backpropagation, as in [63], or by making the forward computation naturally differentiable by perturbing the distances between pixels and triangles analytically in closed form [11, 47, 206]. GenDR falls into this category and is of the second case. Existing works each present their renderer for a specific distribution or sigmoid function. We formally characterize the necessary functions for a differentiable renderer and present an array of options.

**Monte-Carlo Differentiable Renderers** An alternative to analytical differentiable renderers are those which are based on Monte-Carlo sampling techniques. The first example of this is the “redner” path tracer by Li *et al.* [54], who use edge sampling to approximate the gradients of their renderer. Loubet *et al.* [207] build on these ideas and reparameterize the involved discontinuous integrands yielding improved gradient estimates. Zhang *et al.* [208] extend these ideas by differentiating the full path integrals which makes the method more efficient and effective. Lidec *et al.* [49] approach Monte-Carlo differentiable rendering by estimating the gradients of a differentiable renderer via the perturbed optimizers method [37].

**Applications** Popular applications for differentiable renderers are pose [46, 47, 49, 63, 65, 204], shape [11, 63, 208], material [209, 210], texture [46, 47, 207], and lighting [208] estimation. Here, the parameters of an initial scene are optimized to match the scene in a reference image or a set of reference images. Another interesting application is single-view 3D shape prediction without 3D supervision. Here, a neural network predicts a 3D representation from a single image, and the rendering of the image is compared to the original input image. This learning process is primarily guided by supervision of the object silhouette. It is possible to omit this supervision via adversarial style transfer as demonstrated by Petersen *et al.* [10]. Other applications are generating new 3D shapes that match a data set [211, 212] as well as adversarial examples in the real world [213].

In our experiments, we use optimization for pose and shape to benchmark *all* proposed differentiable renderers. As the single-view 3D mesh reconstruction is a complex experiment requiring training a neural network, we benchmark our method on this task only for a selected subset of differentiable renderers.

**T-norms and T-conorms** T-norms and T-conorms (triangular norms and conorms) are binary functions that generalize the logical conjunction (‘and’) and disjunction (‘or’), respectively, to real-valued logics or probability spaces [214, 215]. A generalization of ‘or’ is necessary for a differentiable renderer to aggregate the occlusion caused by faces. The existing analytical differentiable renderers all use the probabilistic T-conorm.

## 5.2 The Generalized Differentiable Renderer

In this section, we present our generalized differentiable mesh renderer. With a differentiable renderer, we refer to a renderer that is continuous everywhere and differentiable almost everywhere (a.e.). Note that, in this context, continuity is a stricter criterion than differentiable a.e. because formally (i) conventional renderers are already differentiable a.e. (which does not mean that they can provide any meaningful gradients), and (ii) most existing “differentiable” renderers are not actually differentiable everywhere.

We start by introducing how a classic hard rendering algorithm operates. The first step is to bring all objects into image space, which is typically a sequence of affine transformations followed by the camera projection. This step is already differentiable. The second step is rasterization: For each pixel, we need to compute the set of faces (typically triangles) which cover it. If the pixel is covered by at least one face, the face that is closest to the camera is displayed.

### 5.2.1 Differentiable Occlusion Test

To make the test whether a pixel  $p$  is occluded by a face  $t$  differentiable, we start by computing the signed Euclidean distance  $d(p, t)$  between pixel and face boundary. By convention, pixels inside the triangle have a positive distance, pixels outside the triangle have a negative distance. For pixels exactly on the boundary, the distance to the face is 0.

For a hard occlusion test, we would just check whether  $d(p, t)$  is non-negative. In a differentiable renderer, we instead introduce a perturbation in the form of a probability distribution with density  $f$  together with a temperature or scale parameter  $\tau > 0$ . We then evaluate the probability that the perturbed distance  $d(p, t) - \tau\epsilon$  is non-negative, where  $\epsilon$  is distributed according to  $f$ . Thus, we compute the probability that  $t$  occludes  $p$  as

$$\begin{aligned} \mathbb{P}_{\epsilon \sim f}(d(p, t) - \tau\epsilon \geq 0) &= \mathbb{P}_{\epsilon \sim f}(\epsilon \leq d(p, t)/\tau) \\ &= \int_{-\infty}^{d(p, t)/\tau} f(x) dx = F\left(\frac{d(p, t)}{\tau}\right) \end{aligned} \quad (5.1)$$

where  $F$  is the CDF of the distribution  $f$  and thus yields a closed-form solution for the desired probability (provided that  $F$  has a closed-form solution or can be appropriately approximated). In a differentiable renderer, we require  $F$  being continuous. Typically,  $F$  has the S-shape of a sigmoid function, see Table 5.1. Therefore, in this chapter, we refer to CDFs as sigmoid functions and use both terms interchangeably.

Most existing differentiable renderers use sigmoid functions or transformations thereof, see Section 5.3, to softly evaluate whether a pixel lies inside a triangle. This accords with the probabilistic interpretation in Equation (5.1) where the probability distribution is defined via the sigmoid function used in each case. Here, the logistic sigmoid function is a popular choice of such a sigmoid function. Note that, recently, it has frequently been referred to as “the” sigmoid in the literature, which is not to be confused with the original and more general terminology.

**Example 5.1** (Logistic Sigmoid)  $F_L(x) = 1/(1 + \exp(-x))$  is the logistic sigmoid function, which corresponds to the logistic distribution.

### 5.2.2 Aggregation

The second step to be made differentiable is the aggregation of multiple faces. While this is conventionally done via a logical ‘or’, the differentiable real-valued counterpart is a T-conorm. T-conorms are formally defined as follows.

**Definition 5.1** (T-conorm) A T-conorm is a binary operation  $\perp : [0, 1] \times [0, 1] \rightarrow [0, 1]$ , which satisfies

- ▶ associativity:  $\perp(a, \perp(b, c)) = \perp(\perp(a, b), c)$ ,
- ▶ commutativity:  $\perp(a, b) = \perp(b, a)$ ,
- ▶ monotonicity:  $(a \leq c) \wedge (b \leq d) \Rightarrow \perp(a, b) \leq \perp(c, d)$ ,
- ▶ 0 is a neutral element  $\perp(a, 0) = a$ .

**Remark 5.1** (T-conorms and T-norms) While T-conorms  $\perp$  are the real-valued equivalents of the logical ‘or’, so-called T-norms  $\top$  are the real-valued equivalents of the logical ‘and’. Certain T-conorms and T-norms are dual in the sense that one can derive one from the other using a complement (typically  $1 - x$ ) and De Morgan’s laws ( $\top(a, b) = 1 - \perp(1 - a, 1 - b)$ ).

We proceed by stating the T-conorm that is used in all applicable previous approximate differentiable renderers with natural gradients.

**Example 5.2** (Probabilistic Sum) The *probabilistic sum* is a T-conorm that corresponds to the probability that at least one out of two independent events occurs. It is defined as

$$\perp^P(a, b) = a + b - ab. \quad (5.2)$$

An alternative to this is the Einstein sum, which is based on the relativistic addition of velocities.

**Example 5.3** (Einstein Sum) The *Einstein sum* is a T-conorm that corresponds to the velocity addition under special relativity:

$$\perp^P(a, b) = \frac{a + b}{1 + ab}. \quad (5.3)$$

Combining the above concepts, we can compute the occupancy or coverage of a pixel  $p$  given a set of faces  $T$  as

$$\mathcal{A}_O(p, T) = \bigvee_{t \in T} F(d(p, t)/\tau). \quad (5.4)$$

In this chapter, we do not vary the shading aggregation and remain with the popular softmax choice. This is equivalent to a Gumbel-Min perturbation of the distances between the camera and the faces. We note that there are no closed-form solutions for adequate alternatives to the  $n$ -ary softmax in the literature that correspond to a probabilistic interpretation and could be expected to perform adequately. In fact, this is an open research problem.

**Figure 5.1:** Taxonomy of probability distributions corresponding to sigmoid functions. The subdivisions are chosen wrt. properties that have a categorically different influence on the behavior of the corresponding renderer. The order of splits when going down in the tree (which could be chosen differently, e.g., symmetric/asymmetric could be the first split) reflects the importance of the properties.

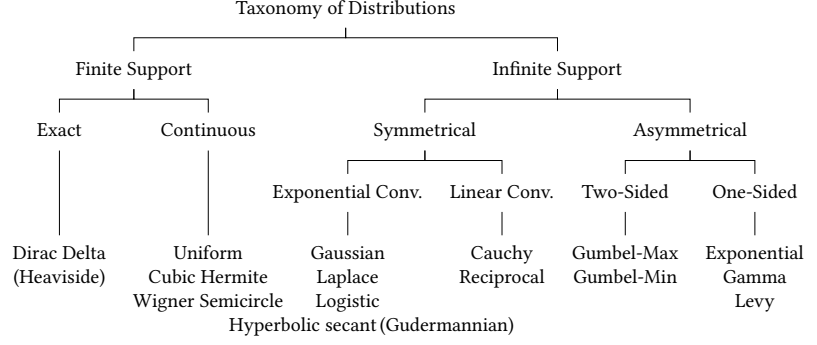
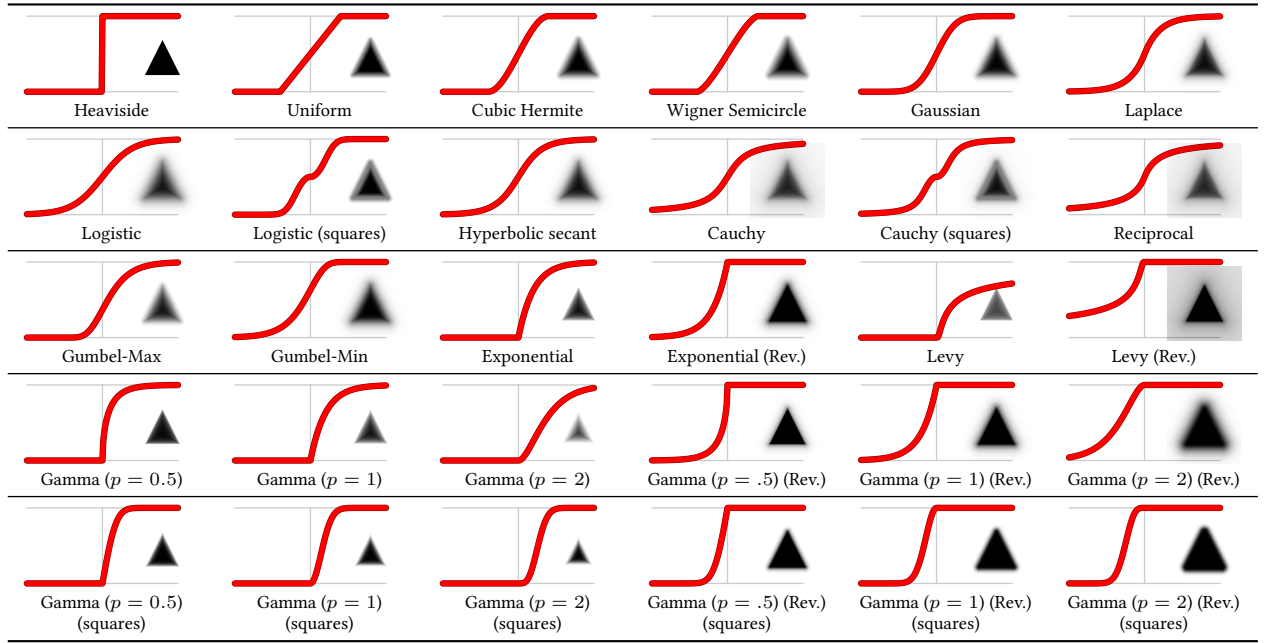


**Table 5.1:** Visualization of a selection of sigmoid functions, which are the CDFs of probability distributions. For each distribution, we display a single rendered triangle to demonstrate their different effects.



Therefore, we focus on varying the two components above and for shading we rely on Petersen *et al.* [11] and Liu *et al.* [46].

### 5.2.3 Shading

The coloring of faces is handled via the Phong model or any other shading model, which is already differentiable. In the literature, Chen *et al.* [47] compare different choices. Finally, to aggregate the coloring of each pixel depending on the distance of the face to the camera (depth), there are two popular choices in the literature: no depth perturbations and taking the closest triangle (like [47, 63, 65]) and Gumbel depth perturbations (like [11, 46]). Only the latter choice is truly continuous, and the closed-form solution for Gumbel depth perturbations is the well-known softmin. As there are (i) no closed-form solutions for adequate alternatives to Gumbel perturbations in the literature, and (ii) these two options have been extensively studied in the literature [11, 46, 47, 49, 63, 65], in this chapter, we do not modify this component and focus on the differentiable silhouette computation and aggregation. While we implement both options in GenDR, in our evaluation, we perform all experiments agnostic to the choice of shading aggregation as the experiments rely solely on the silhouette.

## 5.3 Instantiations of GenDR

We proceed by discussing instantiations of the generalized differentiable renderer (GenDR).

**Distributions** Figure 5.1 provides a taxonomy of the distributions and sigmoid functions that are visualized in Table 5.1. We classify the distributions into those with finite support as well as others with infinite support, where the support is the set of points for which the PDF is greater than zero. Note that the CDFs are constant outside the support region. Among the distributions with *finite support*, there is the *exact* Dirac delta distribution corresponding to the Heaviside function, which yields a discrete renderer, i.e., not a differentiable renderer. There are also *continuous* distributions allowing meaningful gradients, but (due to finite support) only in a limited proximity to each face. Here, we have, among others, the uniform distribution, which corresponds to a piecewise linear step function. The derivative of the uniform distribution is equivalent or very similar (due to minor implementation aspects) to the surrogate gradient of the Neural 3D Mesh Renderer [63]. The distributions with *infinite support* can be categorized into symmetrical and asymmetrical. Among the symmetrical distributions, the Gaussian, the Laplace, the logistic, and the hyperbolic secant have an *exponential convergence* behavior or exponential decay of probability density. On the other hand, there is also the Cauchy distribution which has a *linear convergence*. This yields a significantly different behavior. We include the algebraic function  $x \mapsto x/(2 + 2|x|) + 1/2$ , called reciprocal sigmoid, which we introduced in Chapter 3. This also has a *linear convergence*. Finally, we consider *asymmetrical* distributions with infinite support. The Gumbel-Max and Gumbel-Min are extreme value distributions [216] and *two-sided*, which means that their support covers both positive and negative arguments. The exponential, Gamma, and Levy distributions are one-sided distributions. Here, it is important to not only consider the original distributions but also their mirrored or reversed variants, as well as shifted variations as can be seen in the last three rows of Table 5.1.

SoftRas [46] squares the absolute part of the distance before applying the logistic sigmoid function and thus models the square roots of logistic perturbations. Instead of modifying the argument of  $F$ , we instead interpret it as applying a transformed counterpart CDF  $F_{\text{sq}}$ , which is more in line with the probabilistic interpretation in Equation (5.1). More precisely, we compute the occlusion probability as

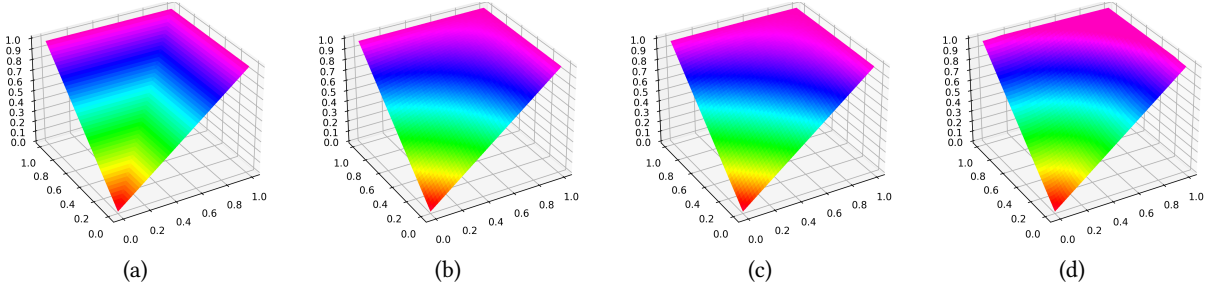
$$F_{\text{sq}}(d(p, t)/\tau) := F(|d(p, t)| \cdot d(p, t)/\tau). \quad (5.5)$$

That means that for each choice of  $F$ , we obtain a counterpart  $F_{\text{sq}}$ . A selection of these for different CDFs  $F$  is visualized in Table 5.1 denoted by “(squares)”. For a mathematical definition of each sigmoid function, see Supplementary Material A.

**Aggregations** Table 5.2 provides an overview of selected T-conorms and displays their properties. The logical ‘or’ is not a T-conorm but the discrete and discontinuous equivalent, which is why we include it here. While there are also discontinuous T-conorms such as the drastic T-conorm, these are

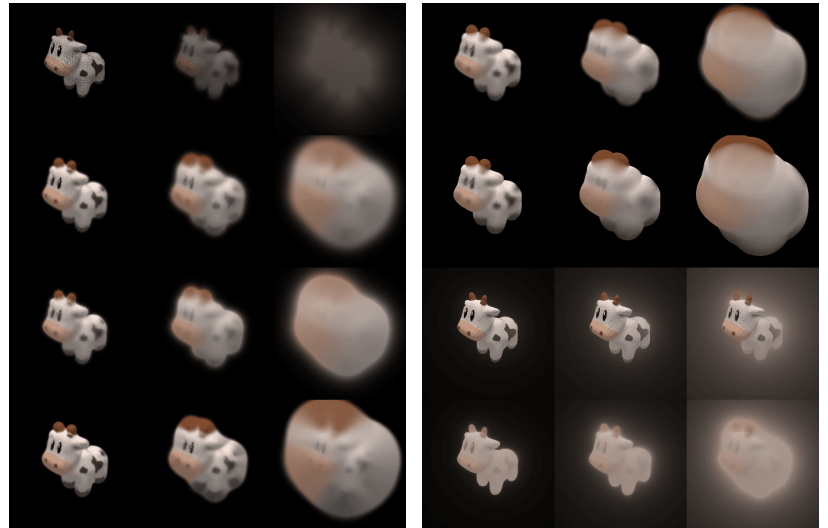
**Table 5.2:** Overview of a selection of suitable T-conorms, which we also benchmark.

| T-conorm        |                | equal to / where          | continuous | contin. diff. | strict | idempotent | nilpotent | Archimedean | $\uparrow / \downarrow$ wrt. $p$ |
|-----------------|----------------|---------------------------|------------|---------------|--------|------------|-----------|-------------|----------------------------------|
| (Logical ‘or’)  | $\vee$         |                           | (X)        | (X)           | —      | (✓)        | —         | —           | —                                |
| Maximum         | $\perp^M$      |                           | ✓          | X             | X      | ✓          | X         | X           | —                                |
| Probabilistic   | $\perp^P$      | $= \perp_1^H = \perp_1^A$ | ✓          | ✓             | ✓      | X          | X         | ✓           | —                                |
| Einstein        | $\perp^E$      | $= \perp_0^H$             | ✓          | ✓             | ✓      | X          | X         | ✓           | —                                |
| Hamacher        | $\perp_p^H$    | $p \in (0, \infty)$       | ✓          | ✓             | ✓      | X          | X         | ✓           | $\downarrow$                     |
| Frank           | $\perp_p^F$    | $p \in (0, \infty)$       | ✓          | ✓             | ✓      | X          | X         | ✓           | $\downarrow$                     |
| Yager           | $\perp_p^Y$    | $p \in (0, \infty)$       | ✓          | X             | X      | X          | ✓         | ✓           | $\uparrow$                       |
| Aczél-Alsina    | $\perp_p^A$    | $p \in (0, \infty)$       | ✓          | ✓             | ✓      | X          | X         | ✓           | $\uparrow$                       |
| Dombi           | $\perp_p^D$    | $p \in (0, \infty)$       | ✓          | ✓             | ✓      | X          | X         | ✓           | $\uparrow$                       |
| Schweizer-Sklar | $\perp_p^{SS}$ | $p \in (-\infty, 0)$      | ✓          | ✓             | ✓      | X          | X         | ✓           | —                                |

**Figure 5.2:** Plot of four selected T-conorms. From left to right: Maximum, Probabilistic, Einstein, and Yager (w/  $p = 2$ ). While (b) and (c) are smooth, the Yager T-conorm (d) is non-smooth, it plateaus and the value is constant outside the unit circle.

naturally not suitable for a differentiable renderer, which is why we exclude them. All except for Max and Yager are continuously differentiable.

The top four rows in Table 5.2 contain individual T-conorms, and the remainder are families of T-conorms. Here, we selected only suitable ranges for the parameter  $p$ . Note that there are some cases in which the T-conorms coincide, e.g.  $\perp^P = \perp_1^H = \perp_1^A$ . A discussion of the remaining properties and a mathematical definition of each T-conorm can be found in Supplementary Material B. Figure 5.2 displays some of the T-conorms and illustrates different properties. In Figure 5.3, we display example renderings with different settings and provide a visual comparison of how the aggregation function affects rendering.

**Figure 5.3:** Visual comparison of different instances of GenDR. In each image, moving from left to right increases the temperature or scale  $\tau$  of the distribution. *Left:* we use a logistic distribution to perturb the triangles and use different T-norms for aggregation (top to bottom:  $\perp^M$ ,  $\perp^P$ ,  $\perp_2^Y$ ,  $\perp_{0.5}^A$ ). *Right:* for the two first rows, we use a uniform distribution and use  $\perp_2^Y$  and  $\perp_{0.5}^A$ . For the last two rows, we use a Cauchy distribution and use  $\perp^P$  and  $\perp_2^Y$ .

| Renderer                   | Distribution            | T-conorm  |
|----------------------------|-------------------------|-----------|
| OpenDR [65]                | Uniform (backward)      | —         |
| N3MR [63]                  | Uniform (backward)      | —         |
| Rhodin <i>et al.</i> [205] | Gaussian                | $\perp^P$ |
| SoftRas [63]               | Square-root of Logistic | $\perp^P$ |
| Log. Relax (Ch. 2)         | Logistic                | $\perp^P$ |
| DIB-R [47]                 | Exponential             | $\perp^P$ |

**Table 5.3:** Differentiable renderers that are (approximately) special cases of GenDR. OpenDR and N3MR do not use a specific T-conorm as their forward computation is hard.

**Existing Special Cases of GenDR** In Table 5.3, we list which existing differentiable renderers are conceptually instances of GenDR. These renderers do each have some other differences, but one key difference lies in the type of distribution employed. Differences regarding shading were discussed in Section 5.2.3.

## 5.4 Experiments

### 5.4.1 Shape Optimization

Our first experiment is a shape optimization task. Here, we use the mesh of an *airplane* and render it from 24 azimuths using a hard renderer. The task is to optimize a mesh (initialized as a sphere) to fit the silhouette of the airplane within 100 optimization steps. Limiting the task to 100 optimization steps is critical for two reasons: (i) The task can be considered to be solved perfectly with any differentiable renderer that produces the correct gradient sign within a large number of steps, but we are interested in the quality of the gradients for the optimization task and how efficient each renderer is. (ii) The total evaluation is computationally expensive because we evaluate a total of 1 242 renderers and perform a grid search over the distribution parameters for each one to provide a fair and reliable comparison.

**Setup** For optimization, we use the Adam optimizer [34] with parameters  $\beta_1 = 0.5, \beta_2 = 0.95$ . For each setting, we perform a grid search over three learning rates ( $\lambda \in \{10^{-1.25}, 10^{-1.5}, 10^{-1.75}\}$ ) and temperatures  $\tau \in \{10^{-0.1 \cdot n} \mid n \in \mathbb{N}, 0 \leq n \leq 80\}$ . Here,  $\lambda = 10^{-1.5} \approx 0.03$  performs best in almost all cases. As for the scale hyperparameter, it is important to use a fine-grained as well as large grid because this behaves differently for each distribution. Here, we intentionally chose the grid larger than the range of reasonable values to ensure that the best choice is used for each setting; the extreme values for the scale were never optimal. We perform this evaluation from five different elevation angles  $\{-60^\circ, -30^\circ, 0^\circ, 30^\circ, 60^\circ\}$  as independent runs and average the final results for each renderer instance.

**Results** In Figure 5.4, we display the results of our evaluation. We can observe that the regular distributions  $F$  typically perform better than the counterpart  $F_{\text{sq}}$ , except for the case of Cauchy and reciprocal sigmoid, which are those with a linear convergence rate. We explain this by the fact that by squaring the distance before applying the sigmoid function, the function has a quadratic convergence rate instead. As the linearly converging functions also perform poorly in comparison to the exponentially converging functions (Gaussian, Laplace, Logistic, Gudermannian), we conclude that linear convergence is inferior to quadratic and exponential convergence. Columns 1 – 3 contain the distributions with finite support, and these do not perform very well on this task. The block of exponentially decaying distributions (columns 4 – 7) performs well. The block of linearly decaying

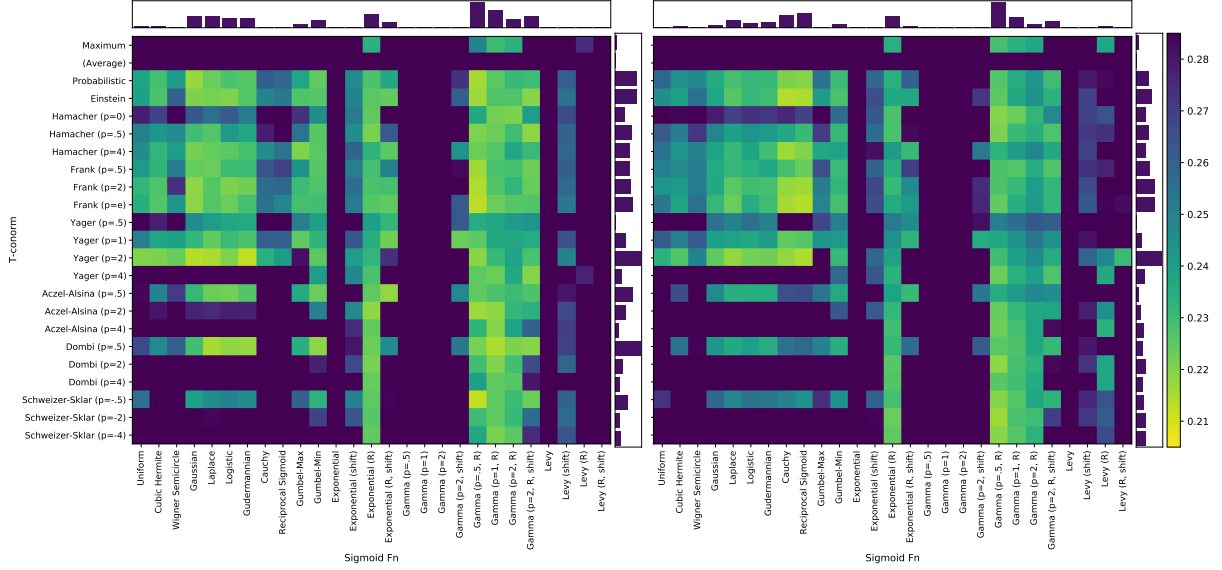


**Figure 5.4:** Results for the 24-view airplane shape optimization task. The optimization is done within a tight budget of 100 steps and the metric is the loss, i.e., lower (=yellow) is better. *Left:* original set of distributions  $F$ . *Right:* the respective counter-parts  $F_{sq}$  in the same location. The marginal histograms display participation in the top 10% combinations.

We note that this result also demonstrates that there is no one optimal distribution for all problems and algorithms, e.g., consider it in relation to differentiable sorting networks where linear convergence is highly desirable.

distributions (columns 8 – 9) performs badly, as discussed above. The block of Levy distributions (last 4 columns) performs even worse because it has an even slower convergence. Here, it also becomes slightly better in the squared setting, but it still exhibits worse performance than for linear convergence.

**Comparison of Distributions** Gumbel, exponential, and gamma distributions do not all perform equally well, but Gumbel-Min, the reversed exponential, and the reversed gamma are all competitive. Confer Table 5.1 where it becomes clear that this is because Gumbel-Max, exponential and gamma have all of their mass inside the triangle, i.e., they yield smaller faces. This is problematic because in this case, it can cause gaps between neighboring triangles, which hinders optimization. These gaps can also be seen in Figure 5.3 (top left-most rendering). As the reverse counterparts yield larger faces and do not suffer from this problem, they perform better. Note that, in this respect, the asymmetrical distributions have an advantage over the symmetrical distributions because symmetrical distributions always have an accumulated density of 0.5 at the edge, and thus the size of the face stays the same. We can see that, among the asymmetrical distributions, Gamma performs best.

**Comparison of T-conorms** We find that  $\perp^M$  and “average” (which is not a T-conorm but was used as a baseline in [46]) perform poorly. Also,  $\perp_4^Y$ ,  $\perp_2^A$ ,  $\perp_4^A$ ,  $\perp_2^D$ ,  $\perp_4^D$ ,  $\perp_{-2}^{SS}$ , and  $\perp_{-4}^{SS}$  perform poorly overall. This can be explained as they are rather extreme members of their respective T-conorm families; in all of them, the  $p$ th power is involved, which can become a problematic component, e.g.,  $x^4$  is vanishingly small for  $x = 0.5$ . Interestingly, the gamma and exponential distributions still perform well with these, likely since they are not symmetric and have an accumulated probability of 1 on the edge. Notably, the Yager T-conorm ( $p = 2$ ) performs very well, although having a plateau and thus zero gradients outside the unit disc (Figure 5.2 (d)).

Finally, we compute histograms of how many times each respective distribution and T-conorm is involved in the best 10% of overall results. This is independent for the left and right plots. We can observe that Gamma

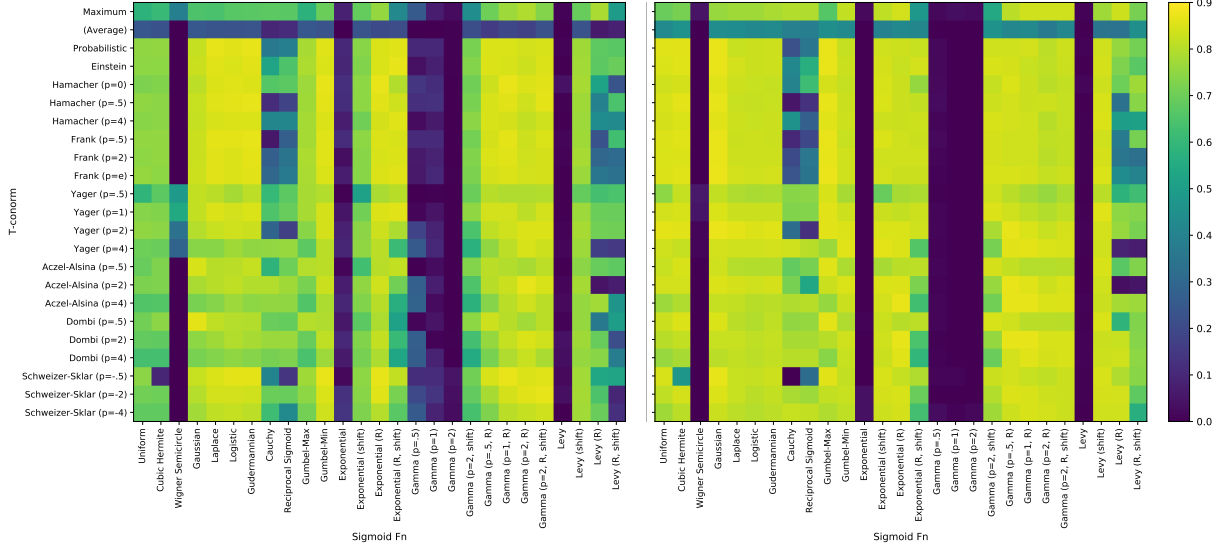


**Figure 5.5:** Results for the teapot camera pose optimization task. The optimization is done with a temperature  $\tau$  that is scheduled to decay. The metric is the fraction of camera poses recovered, while the initialization angle errors are uniformly sampled from  $[15^\circ, 75^\circ]$ . The figure shows the original set of distributions  $F$  (left) and the respective square-root distribution  $F_{sq}$  (right).

( $p = 0.5$ , Reversed) performs the best overall (because it is more robust to the choice of T-conorm). Among the T-conorms, we find that  $\perp_2^Y$  and  $\perp_{0.5}^D$  perform best. The probabilistic and Einstein sums perform equally, and share the next place.

### 5.4.2 Camera Pose Optimization

In our second experiment, the goal is to find the camera pose for a model of a *teapot* from a reference image. The angle is randomly modified by an angle uniformly drawn from  $[15^\circ, 75^\circ]$ , and the distance and camera view angle are also randomized. We sample 600 pairs of a reference image and an initialization and use this set of settings for each method. For optimization, we use Adam with a learning rate of either 0.1 or 0.3 (via grid search) and optimize for 1000 steps. During the optimization, we transition an initial scale of  $\tau = 10^{-1}$  logarithmically to a final value of  $\tau = 10^{-7}$ . This allows us to avoid a grid search for the optimal scale, and makes sense since an initially large  $\tau$  is beneficial for pose optimization, because a smoother model has a higher probability of finding the correct orientation of the object. This contrasts with the setting of shape estimation, where this would be fatal because the vertices would collapse to the center.

**Results** In Figure 5.5, we display the results of this experiment. The metric is the fraction of settings which achieved matching the ground truth pose up to  $3^\circ$ . We find that in this experiment, the results are similar to those in the shape optimization experiment. Note that there are larger yellow areas because the color map ranges from 0% to 90%, while in the shape optimization plot the color map ranges in a rather narrow loss range. Additional results for the model of a chair can be found in [5].

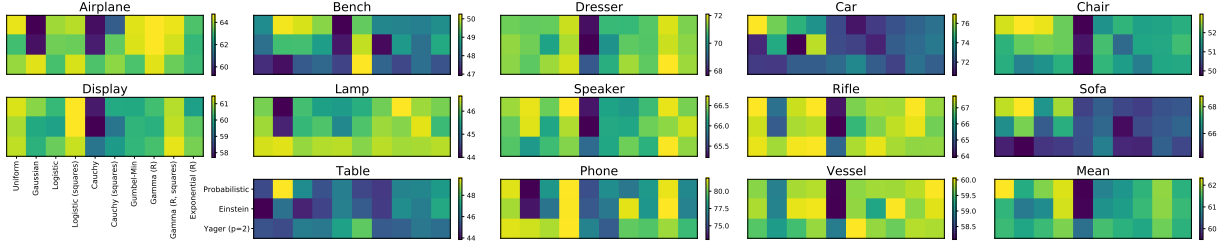


**Figure 5.6:** Single-view reconstruction results for each of the 30 selected renderers as a 3D IoU (in %) heatmap for each class. While the uniform distribution ( $w/\perp^P$ ) performs best on average and the square root of logistic ( $w/\perp^P, \perp^E$ ) performs second-best on average, the optimal setting depends on the characteristics of the respective classes. For the ‘Airplane’ class, the Gamma distribution performed best, and this is also the distribution that performed best in the airplane shape optimization experiment in Section 5.4.1. For classes of furniture with legs, such as ‘Bench’, ‘Chair’, and ‘Table’, we find that the Gaussian distribution consistently performs best. The pairs of similar classes ‘Display’+‘Phone’, ‘Dresser’+‘Speaker’, and ‘Vessel’+‘Rifle’ also show similar performance patterns. For example, dresser and speakers tend to be cuboid, while rifles and vessels tend to be rather long and slim. Considering the Gaussian distribution, it is interesting to see that for some classes  $\perp^P$  and  $\perp^E$  perform better, while for other classes  $\perp_2^Y$  performs much better.

**Table 5.4:** Selected single-view reconstruction results measured in 3D IoU.

| Method                                                   | Airplane      | Bench         | Dresser       | Car           | Chair         | Display       | Lamp          | Speaker       | Rifle         | Sofa          | Table         | Phone         | Vessel        | Mean          |
|----------------------------------------------------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|
| Kato <i>et al.</i> [63] N3MR (Uniform Backward)          | 0.6172        | 0.4998        | 0.7143        | 0.7095        | 0.4990        | 0.5831        | 0.4126        | 0.6536        | 0.6322        | 0.6735        | 0.4829        | 0.7777        | 0.5645        | 0.6015        |
| Liu <i>et al.</i> [46] SoftRas (Square-root of Logistic) | 0.6419        | 0.5080        | 0.7116        | 0.7697        | 0.5270        | 0.6156        | 0.4628        | 0.6654        | 0.6811        | 0.6878        | 0.4487        | 0.7895        | 0.5953        | 0.6234        |
| Chen <i>et al.</i> [47] DIB-R (Exponential)              | 0.570         | 0.498         | 0.763         | 0.788         | 0.527         | 0.588         | 0.403         | 0.726         | 0.561         | 0.677         | 0.508         | 0.743         | 0.609         | 0.612         |
| Probabilistic + Uniform ( $\approx$ [63, 65])            | 0.6456        | 0.4855        | 0.7113        | <b>0.7696</b> | 0.5276        | 0.6126        | 0.4611        | 0.6651        | <b>0.6773</b> | 0.6835        | 0.4514        | 0.8148        | 0.5971        | <b>0.6232</b> |
| Probabilistic + Logistic ( $\approx$ Ch. 2)              | 0.6396        | 0.5005        | 0.7105        | 0.7471        | 0.5288        | 0.6022        | 0.4586        | 0.6639        | 0.6742        | 0.6660        | 0.4666        | 0.7771        | 0.5980        | 0.6179        |
| Probabilistic + Logistic (squares) ( $\approx$ [46])     | 0.6416        | 0.4966        | 0.7175        | 0.7386        | 0.5224        | <b>0.6147</b> | 0.4550        | <b>0.6673</b> | <b>0.6771</b> | 0.6818        | 0.4529        | <b>0.8186</b> | 0.5984        | 0.6217        |
| Probabilistic + Exponential (R) ( $\approx$ [47])        | 0.6321        | 0.4857        | 0.7123        | 0.7298        | 0.5178        | 0.5983        | 0.4611        | 0.6642        | 0.6713        | 0.6546        | 0.4700        | 0.7717        | <b>0.6005</b> | 0.6130        |
| Probabilistic + Gaussian ( $\approx$ [205])              | 0.5922        | <b>0.5020</b> | 0.7104        | 0.7561        | <b>0.5297</b> | 0.6080        | 0.4399        | <b>0.6668</b> | 0.6533        | <b>0.6879</b> | <b>0.4961</b> | 0.7301        | 0.5894        | 0.6125        |
| Probabilistic + Gamma (R)                                | <b>0.6473</b> | 0.4842        | 0.7093        | 0.7220        | 0.5159        | 0.6033        | <b>0.4665</b> | 0.6626        | 0.6719        | 0.6505        | 0.4642        | 0.7778        | 0.5978        | 0.6133        |
| Einstein + Gamma (R, squares)                            | 0.6438        | 0.4816        | <b>0.7174</b> | 0.7284        | 0.5170        | 0.6111        | 0.4654        | 0.6647        | 0.6760        | 0.6546        | 0.4626        | <b>0.8189</b> | 0.5973        | 0.6184        |
| Yager ( $p=2$ ) + Cauchy (squares)                       | 0.6380        | <b>0.5026</b> | 0.7047        | 0.7359        | 0.5188        | 0.5976        | 0.4617        | 0.6612        | 0.6726        | 0.6619        | 0.4819        | 0.7560        | <b>0.6006</b> | 0.6149        |

### 5.4.3 Single-View 3D Reconstruction

**Setup** Finally, we reproduce the popular ShapeNet single-view 3D reconstruction experiment from Section 2.5. We select three T-conorms ( $\perp^P$ ,  $\perp^E$ ,  $\perp_2^Y$ ) and 10 distributions (Uniform, Gaussian, Logistic, Logistic (squares), Cauchy, Cauchy (squares), Gumbel-Min, Gamma (R,  $p = 0.5$ ), Gamma (R,  $p = 0.5$ , squares), and Exponential (R)). These have been selected because they have been used in previous works, are notable (Cauchy, Gumbel-Min, Einstein), or have performed especially well in the aircraft shape optimization experiment (Gamma, Yager). For each setting, we perform a grid search of  $\tau$  at resolution  $10^{0.5}$ . We use the same model architecture as in Chapter 2 and train with a batch size of 64 for 250 000 steps using the Adam optimizer [34]. This also corresponds to the setup by Liu *et al.* [46]. We also schedule the learning rate to  $10^{-4}$  for the first 150 000 steps and use a learning rate of  $3 \cdot 10^{-5}$  for the remaining training. At this point (after the first 150 000 steps), we also decrease the temperature  $\tau$  by a factor of 0.3. Using different learning rates (which we did as an ablation) did not improve the results.

**Results** In Figure 5.6, we display and discuss the class-wise results for all 30 selected renderers. In Table 5.4, we show the (self-) reported results for existing differentiable renderers in the top block. In the bottom block, we display our results for the methods that are equivalent ( $\approx$ ) or very similar ( $\approx$ ) to the six existing differentiable renderers. The differences for equivalent methods can be explained with small variations in the setting and minor implementation and framework differences. Additionally, we include three noteworthy alternative renderers, such as the one that also performed best on the prior airplane shape optimization task. We conclude that the optimal

choice of renderer heavily depends on the characteristics of the 3D models and the task. Surprisingly, we find that the simple uniform method achieves consistently good results and the best average score.

## Conclusion

In this chapter, we generalized differentiable mesh renderers and explored a large space of instantiations of our generalized renderer GenDR, extending our work in Chapter 2. We found that there are significant differences between different distributions for the occlusion test, but also between different T-conorms for the aggregation. In our experiments, we observed that the choice of renderer has a large impact on the kind of models that can be rendered most effectively. We find that the uniform distribution outperforms the other tested distributions on average, which is surprising considering its simplicity. Remarkably, the uniform distribution had already been used implicitly for the early surrogate gradient renderers but was later discarded for the approximate differentiable renderers.



# Differentiable Logic

In this chapter, we cover differentiable logic, which is an integral part of differentiable algorithms. In contrast to the previous chapters, the primary goal of this chapter is not to relax an existing algorithm to apply algorithmic supervision but instead to find an algorithm or logical expression via a differentiable relaxation of the space of logical expressions. Specifically, we consider the space of logic gate networks, and relax it to differentiable logic gate networks, such that the logic gate network or algorithm can be learned. Differentiable logic gate networks are a novel kind of neural networks, and in their discretized (non-relaxed) form, they operate on logic gates only, which allows for very fast inference.

With the success of neural networks, there has also always been strong interest in research and industry in making the respective computations as fast and efficient as possible, especially at inference time. Various techniques have been proposed to solve this problem, including reduced computational precision [217, 218], binary [219] and sparse [220] neural nets. In this chapter, we want to train a different kind of architecture, which is well known in the domain of computer architectures: logic (gate) networks.

The problem in training networks of discrete components like logic gates, is that they are non-differentiable and, therefore, conventionally, cannot be optimized via standard methods such as gradient descent [35]. One approach for this would be gradient-free optimization methods such as evolutionary training [221, 222], which works for small models but becomes infeasible for larger ones.

In this chapter, we propose an approach for gradient-based training of logic gate networks. Logic gate networks are based on binary logic gates, such as “and” and “xor” (see Table 6.1). For training logic gate networks, we continuously relax them to differentiable logic gate networks, which allows efficiently training them with gradient descent. For this, we use real-valued logic and learn which logic gate to use at each neuron. After training, the resulting network is binarized to a (hard) logic gate network by choosing the logic gate with the highest probability. As the (hard) logic gate network comprises logic gates only, it can be executed very fast. Additionally, as the logic gates are binary, every neuron / logic gate has only 2 inputs, and the network is extremely sparse.

In contrast to binary neural networks, logic gate networks do not have weights, are intrinsically sparse as they have only 2 inputs to each neuron, and are not simply a form of low precision (wrt. weights and/or activations) neural networks.

They also differ from current sparse neural network approaches, as our goal is to learn which logic gate operators are present at each neuron while the connections between neurons are (pseudo-)randomly initialized and remain fixed. The network is, thus, parameterized by the choice of the binary function for each neuron. As there is a total of 16 functions of signature  $f : \{0, 1\} \times \{0, 1\} \rightarrow \{0, 1\}$ , only 4 bits are required to store the information about which operation a neuron executes. The objective is to learn which

|     |                                              |    |
|-----|----------------------------------------------|----|
| 6.1 | Related Work . . . . .                       | 78 |
| 6.2 | Logic Gate Networks . . .                    | 80 |
| 6.3 | Differentiable Logic Gate Networks . . . . . | 81 |
| 6.4 | Experiments . . . . .                        | 85 |

of those 16 operations is optimal for each neuron. Specifically, for each neuron, we learn a probability distribution over possible logic gates, which we parameterize via softmax. We find that this approach allows learning logic gate networks very effectively via gradient descent.

Logic gate networks allow for very fast classification, with speeds beyond a million images per second on a single CPU core (for MNIST at  $> 97.5\%$  accuracy). The computational cost of a layer with  $n$  neurons is  $\Theta(n)$  with very small constants (as only logic gates of Booleans are required), while, in comparison, a fully connected layer (with  $m$  input neurons) requires  $\Theta(n \cdot m)$  computations with significantly larger constants (as it requires floating-point arithmetic). While the training can be more expensive than for regular neural networks (however, just by a constant and asymptotically less expensive), to our knowledge, the proposed method is the fastest available architecture at inference time. Overall, our method accelerates inference speed (in comparison to fully connected ReLU neural networks) by around two orders of magnitude. In the experiments, we scale the training of logic gate networks up to 5 million parameters, which can be considered relatively small in comparison to other architectures. In comparison to the fastest neural networks at 98.4% on MNIST, our method is more than  $12\times$  faster than the best binary neural networks and 2 – 3 orders of magnitude faster than the theoretical speed of sparse neural networks.

## 6.1 Related Work

In this section, we discuss related work on differentiable logics, learning logic gate networks, and methods with methodological or conceptual similarity. In addition, we discuss binary and sparse neural networks, which are baselines because they are very fast network architectures, but which are not conceptually similar to differentiable logic gate networks.

### 6.1.1 Differentiable Logics and Triangular Norms

Differentiable logics (aka. real-valued logics, or infinite-valued logic) are well-known in the fields of fuzzy logics [223] and probabilistic metric spaces [214, 224]. In Chapter 5, we provide an introduction to differentiable logics and triangular norms (T-norms), and in Supplementary Material B we give examples for T-norms and T-conorms. An additional reference for differentiable real-valued logics is Van *et al.* [215].

### 6.1.2 Learning Logic Gate Networks

Chatterjee [225] explored “memorization”, a method for memorizing binary classification data sets with a network of binary lookup tables. He does this to explore principles of learning and memorization, as well as their trade-off and generalization capabilities. He constructs the networks of lookup tables by counting conditional frequencies of data points. We mention this work here because binary logic gates may be seen as the special case of 2-input lookup tables. That is, his method has some similarities to our resulting networks. However, as he memorizes the data set, while this leads to some generalization, this generalization is limited. In his experiments, he considers

the binary classification task of distinguishing the combined classes ‘0’–‘4’ from the combined classes ‘5’–‘9’ of MNIST and achieves a test accuracy of 90%.

Brudermueller *et al.* [226] propose a method where they train a neural network on a classification task and then translate it, first into random forests, and then into networks of AND-Inverter logic gates, i.e., networks based only on “and” and “not” logical gates. They evaluate their approach on the “gastrointestinal bleeding” and “veterans aging cohort study” data sets and argue for the verifiability and interpretability of small logical networks in patient care and clinical decision-making.

### 6.1.3 Relaxed Connectivity in Networks

For differentiable logic gate network, we relax *which logic operator* is applied at each node, while the connections are predefined.

Zimmer *et al.* [227] propose differentiable logic machines for inductive logic programming. For this, they propose logic modules, which contain one level of logic and for which they predefine that the first half of operators are fuzzy “and”s and the second half are fuzzy “or”s. They relax *which nodes* are the inputs to the “and”s and “or”s of their logic modules.

Similarly, Chen [228] proposes Gumbel-Max Equation Learner Networks, where he predefines a set of arithmetic operations in each layer and learns via Gumbel-Softmax [75, 229], *which outputs* of the previous layer should be used as inputs of a respective arithmetic operation. He uses this to learn symbolic expressions from data.

While these works relax which nodes are connected to which nodes, this is fixed in our method, and we relax which operator is at which node.

### 6.1.4 Evolutionary Learning of Networks

Mocanu *et al.* [230] propose training neural networks with sparse evolutionary training inspired by network science. Their method evolves an initial sparse topology of two consecutive layers of neurons into a scale-free topology. On MNIST, they achieve (with 89 797 parameters) an accuracy of 98.74%.

Gaier *et al.* [231] propose learning networks of operators such as ReLU, sin, inverse, absolute, step, and tanh using evolutionary strategies. Specifically, they use the population-based neuroevolution algorithm NEAT. They achieve learning those floating-point function-based networks and achieve an accuracy of 94.2% on MNIST with a total of 1 849 connections.

### 6.1.5 Learning of Decision Trees

Zantedeschi *et al.* [232] propose to learn decision trees by quadratically relaxing the decision trees from mixed-integer programs that learn the discrete parameters of the tree (input traversal and node pruning). This allows them to differentiate in order to simultaneously learn the continuous parameters of splitting decisions.

While decision trees are different from logic gate networks, the methods presented in Chapter 2 could be used to continuously relax decision trees.

Logic gate networks can be seen as collections of binary (logic gate-based) trees, but binary (logic gate-based) trees are conceptually vastly different from *decision* trees: decision trees rely on splitting decisions instead of logical operations, and the tree structure of decision trees and logic gate-based trees are in the opposite directions [233]. Logic gate-based trees begin with a number of inputs (leafs) and apply logic gates to aggregate them to a binary value (root). Decision trees begin at the root and apply splitting decisions (for which they consider an external input) to decide between children, such that they end up at a leaf node corresponding to a value.

### 6.1.6 Binary Neural Networks

Binary neural networks (BNNs) [219] are conceptually very different from logic gate networks. For binary neural networks, “binary” refers to representing activations and weights of a neural network with binary states (e.g.,  $\{-1, +1\}$ ). This allows approximating the expensive matrix multiplication by faster XNOR and bitcount (popcount) operations. The logical operations involved in BNNs are not learned but instead predefined to approximate floating-point operations, and, as such, a regular weight-based neural network. This is not the case for logic gate network, where we learn the logic operations, we do not approximate weight-based neural networks, and do not have weights. While BNNs are defined via their weights and not via their logic operations, logic gate networks do not have weights and are purely defined via their logic operations. We include BNNs as baselines in our experiments because they achieve the best inference speed.

### 6.1.7 Sparse Neural Networks

Sparse neural networks [220] are neural networks where only a selected subset of connections is present, i.e., instead of fully-connected layers, the layers are *sparse*. In the literature of sparse neural networks, usually, the task is to distill a sparse neural network from a dense neural network and the choice of connections is important. While logic gate networks are sparse by definition, their connections remain fixed and their initialization is random. We include sparse nets in our experiments because they can provide efficient inference.

## 6.2 Logic Gate Networks

Logic gate networks are networks similar to neural networks where each neuron is represented by a binary logic gate like ‘and’, ‘nand’, and ‘nor’ and accordingly has only two inputs (instead of all neurons in the previous layer as it is the case in fully-connected neural networks). Given a binary vector as input, pairs of Boolean values are selected, binary logic gates are applied to them, and their output is then used as input for layers further downstream. Logic gate networks do not use weights. Instead, they are parameterized via the choice of logic gate at each neuron. In contrast to fully connected neural networks, binary logic gate networks are sparse because each neuron has only 2 instead of  $n$  inputs, where  $n$  is the number of neurons per layer.

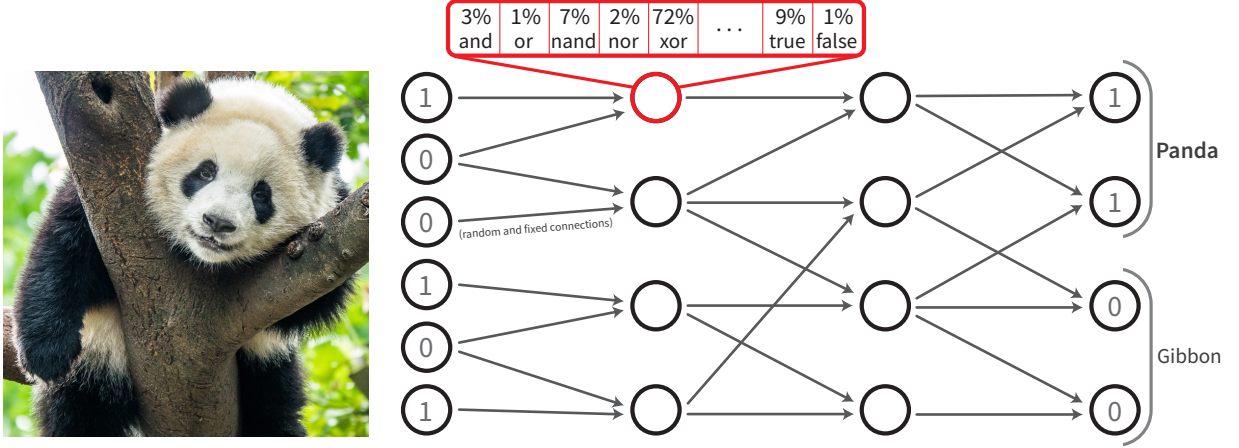


**Figure 6.1:** Overview of the proposed differentiable logic gate networks: Boolean valued inputs are processed by a layer of neurons such that each neuron receives two inputs. The connectivity of neurons remains fixed after an initial pseudo-random initialization. Each neuron is continuously parameterized by a distribution over logical operators. During training, this distribution is learned for each neuron, and, during inference, the most likely operator is used for each neuron. There are multiple outputs per class, which are aggregated by bit-counting, which yields the class scores.

In logic gate networks, we do not need activation functions as they are intrinsically non-linear.

While it is possible to make a prediction simply with a single binary output or  $k$  binary outputs for  $k$  classes, this is not ideal. This is because in the crisp case, we only get 0s or 1s and no graded prediction, which would be necessary for a “greatest activation” classification scheme. By using multiple neurons per class and aggregating them by summation, even the crisp case allows for grading, with as many levels as there are neurons per class. Each of these neurons could capture a different piece of evidence for a class, and this allows finer grade predictions.

Figure 6.1 illustrates a small logic gate network. In the illustration, each node corresponds to a single logic operator. Note that the distribution over operators (red) is part of the differentiable relaxation discussed in the next section.

## 6.3 Differentiable Logic Gate Networks

Training binary logic gate networks is hard because they are not differentiable, and thus no gradient descent-based training is conventionally possible. Thus, we propose relaxing logic gate networks to differentiable logic gate networks to allow for gradient-based training.

**Differentiable Logics** To make binary logic networks differentiable, we leverage the following relaxation. First, instead of hard binary activations / values  $a \in \{0, 1\}$ , we relax all values to probabilistic activations  $a \in [0, 1]$ . Second, we replace the logic gates by computing the expected value of the activation given probabilities of independent inputs  $a_1$  and  $a_2$ . For example, the probability that two independent events with probabilities  $a_1$  and  $a_2$  both occur is  $a_1 \cdot a_2$ . We report differentiable real-valued relaxations of all binary logic gate operators in Table 6.1. These operators correspond to the probabilistic T-norm and T-conorm.

Accordingly, we define the activation of a neuron with the  $i$ th operator as

$$a' = f_i(a_1, a_2), \quad (6.1)$$

where  $f_i$  is the  $i$ th real-valued operator corresponding to Table 6.1 and  $a_1, a_2$  are the inputs to the neuron. There are also alternative real-valued logics like the Hamacher T-(co)norm, the relativistic Einstein sum, and the Łukasiewicz T-(co)norm. While, in this chapter, we use the probabilistic interpretation, we review an array of possible alternative T-norms and T-conorms in Supplementary Material B.

**Table 6.1:** Differentiable real-valued binary logic gate operators corresponding to the probabilistic T-norm and T-conorm.

| ID | Operator                | real-valued         | 00 | 01 | 10 | 11 |
|----|-------------------------|---------------------|----|----|----|----|
| 0  | False                   | 0                   | 0  | 0  | 0  | 0  |
| 1  | $A \wedge B$            | $A \cdot B$         | 0  | 0  | 0  | 1  |
| 2  | $\neg(A \Rightarrow B)$ | $A - AB$            | 0  | 0  | 1  | 0  |
| 3  | $A$                     | $A$                 | 0  | 0  | 1  | 1  |
| 4  | $\neg(A \Leftarrow B)$  | $B - AB$            | 0  | 1  | 0  | 0  |
| 5  | $B$                     | $B$                 | 0  | 1  | 0  | 1  |
| 6  | $A \oplus B$            | $A + B - 2AB$       | 0  | 1  | 1  | 0  |
| 7  | $A \vee B$              | $A + B - AB$        | 0  | 1  | 1  | 1  |
| 8  | $\neg(A \vee B)$        | $1 - (A + B - AB)$  | 1  | 0  | 0  | 0  |
| 9  | $\neg(A \oplus B)$      | $1 - (A + B - 2AB)$ | 1  | 0  | 0  | 1  |
| 10 | $\neg B$                | $1 - B$             | 1  | 0  | 1  | 0  |
| 11 | $A \Leftarrow B$        | $1 - B + AB$        | 1  | 0  | 1  | 1  |
| 12 | $\neg A$                | $1 - A$             | 1  | 1  | 0  | 0  |
| 13 | $A \Rightarrow B$       | $1 - A + AB$        | 1  | 1  | 0  | 1  |
| 14 | $\neg(A \wedge B)$      | $1 - AB$            | 1  | 1  | 1  | 0  |
| 15 | True                    | 1                   | 1  | 1  | 1  | 1  |

**Differentiable Choice of Operator** While real-valued logics allow differentiation, they do not allow training as the operators are not continuously parameterized, and thus (under hard binary inputs) the activations in the network will always be  $a \in \{0, 1\}$ . Thus, we propose to represent the choice of *which* logic gate is present at each neuron by a categorical probability distribution. For this, we parameterize each neuron with 16 floats (i.e.,  $\mathbf{w} \in \mathbb{R}^{16}$ ), which, by softmax, map to the probability simplex (i.e., a categorical probability distribution such that all entries sum up to 1 and it has only non-negative values). That is,  $\mathbf{p}_i = e^{\mathbf{w}_i} / (\sum_j e^{\mathbf{w}_j})$ , and thus  $\mathbf{p}$  lies in the probability simplex  $\mathbf{p} \in \Delta^{15}$ . During training, we evaluate for each neuron all 16 relaxed binary logic gates and use the categorical probability distribution to compute their weighted average. Thus, we define the activation  $a'$  of a differentiable logic gate neuron as

$$a' = \sum_{i=0}^{15} \mathbf{p}_i \cdot f_i(a_1, a_2) = \sum_{i=0}^{15} \frac{e^{\mathbf{w}_i}}{\sum_j e^{\mathbf{w}_j}} \cdot f_i(a_1, a_2). \quad (6.2)$$

**Aggregation of Output Neurons** Now, we may have  $n$  output neurons  $a_1, a_2, \dots, a_n \in [0, 1]$ , but we may want the logic gate network to only predict  $k < n$  values of a larger range than  $[0, 1]$ . Further, we may want to be able to produce graded outputs. Thus, we can aggregate the outputs as

$$\hat{y}_i = \sum_{j=i \cdot n/k + 1}^{(i+1) \cdot n/k} a_j / \tau + \beta \quad (6.3)$$

where  $\tau$  is a normalization temperature and  $\beta$  is an optional offset.

### 6.3.1 Training Considerations

**Training** For learning, we randomly initialize the connections and the parameterization of each neuron. For the initial parameterization of each neuron, we draw elements of  $q$  independently from a standard normal distribution. In all reported experiments, we use the same number of neurons in each layer (except for the input) and between 4 and 8 layers, which we call straight network. We train all models with the Adam optimizer [34] at a constant learning rate of 0.01.

**Discretization** After training, during inference, we discretize the probability distributions by only taking their mode (i.e., their most likely value), and thus the network can be computed with Boolean values, which makes inference very fast. In practice, we observe that most neurons converge to one logic gate operation; therefore, the discretization step introduces only a small error. We note that all reported results are accuracies after discretization.

**Classification** In the application of a classification learning setting with  $k$  classes (e.g., 10) and  $n$  output neurons (e.g., 1 000), we group the output into  $k$  groups of size  $n/k$  (e.g., 100). Then, we count the number of 1s which corresponds to the classification score such that the predicted class can be retrieved via the  $\arg \max$  of the class scores. During differentiable training, we sum up the probabilities of the outputs in each group instead of counting the 1s, and we can train the model using a softmax cross-entropy classification loss. For a reference on choosing the hyperparameter  $\tau$ , see Section 6.4.6; the offset  $\beta$  is not relevant for the classification setting, as  $\arg \max$  is shift-invariant. A heuristic for choosing  $\tau$  is that when increasing  $n$ , we have to reduce  $\tau$ . Empirically, when increasing  $\tau$  by a factor of 10,  $\tau$  should be decreased by a factor of around 2 to  $\sqrt{10}$ .

**Regression** For regression learning, let us assume that we need to predict a  $k$ -dimensional output vector. Here,  $\tau$  and  $\beta$  play the role of an affine transformation to transform the range of possible predictions from 0 to  $n/k$  to an application specific and more suitable range. Here, the optional bias  $\beta$  is important, e.g., if we want to predict values outside the range of  $[0, n/k/\tau]$ . In some cases, it is desirable to cover the entire range of real numbers, which may be achieved using a logit transform  $\text{logit}(x) = \sigma^{-1}(x) = \log \frac{x}{1-x}$  in combination with  $\tau = n/k, \beta = 0$ . During differentiable training, we sum up the probabilities of the outputs in each group instead of counting the 1s, and we can train the model, e.g., using an MSE loss.

### 6.3.2 Remarks

**Boolean Vectorization via Larger Data Types** One important computational detail for inference time is that we do not use Boolean data types but instead use larger data types such as, e.g., int64 for a batch size of 64, and thus perform bit-wise logics on larger batches which significantly improves speed on current hardware. For int64, we batch 64 data points such that the  $i$ th Boolean value of the  $j$ th data point is the  $j$ th bit in the  $i$ th int64 integer. Thus, it is possible to compute on average around 250 binary logic

gates on each core in each CPU clock cycle (i.e., per Hz) on a typical desktop / notebook computer. This is the case because modern CPUs execute many instructions per clock cycle even on a single core, and additionally (for Booleans) allow single-instruction multiple-data (SIMD) by batching bits from multiple data points into one integer (e.g., int64). Using advanced vector extensions (AVX), even larger speedups would be possible. On GPU, this computational speedup is also available in addition to typical GPU parallelization.

**Aggregation of Output Neurons via Binary Adders** In addition, during inference, we aggregate the output neurons directly using logic gate nets that make up respective adders, as writing all outputs to memory would constitute a bottleneck and aggregating them using logic gate networks is fast. Specifically, using logic gates, we construct adders that can add exactly one bit to a binary number. Thus, the logic gate networks return the aggregated results in the form of  $\lceil \log_2(n/k + 1) \rceil$ -bit binary numbers.

**Memory Considerations** Since we pseudo-randomly initialize the connections in binary logic gate networks, i.e., which are the two inputs for each neuron, we do not need to store the connections as they can be reproduced from a single seed. Thus, it suffices to store the 4-bit information which of the 16 logic gate operators is used for each neuron. Thus, the memory footprint of logic gate networks is drastically reduced in comparison to neural networks, binary neural networks, and sparse neural networks.

**Pruning the Model** An additional speedup for the inference of logic gate networks is available by pruning neurons that are not used, or by simplifying logical expressions. However, this requires storing the connections, posing a (minor) trade-off between memory and speed.

**Subset of Operators** We investigated reducing the set of operators; however, we found that, in all settings, the more expressive full set of 16 operators performed better. Nevertheless, a smaller set of operators could be a good trade-off for reducing the model size.

**Half Precision** We also investigated training with half precision (float16). In our experiments, half precision (in comparison to full precision) did not degrade training performance; nevertheless, all reported results were trained with full precision (float32).

### 6.3.3 Current Limitations and Opportunities

**Expensive Training** A limitation of differentiable logic gate networks is their relatively higher training cost compared to (performance-wise) comparable conventional neural networks. The higher training cost is because multiple differentiable operators need to be evaluated for each neuron, and in their real-valued differentiable form, most of these operators require floating-point value multiplications. However, the practical computational

cost can be reduced through improved implementations. Additionally, differentiable logic gate networks are asymptotically cheaper to train compared to conventional neural networks due to their sparsity.

**Convolutions and Other Architectures** Convolutional logic gate networks and other architectural components such as residual connections are interesting and important directions for future research.

**Edge Computing and Embedded Machine Learning** We would like to emphasize that the current limitations to rather small architectures (compared to large deep learning architectures) does not need to be a limitation: For example, in edge computing and embedded machine learning [234–237], models are already limited to tiny architectures because they run, e.g., on mobile CPUs, microcontrollers, or IoT devices. In these cases, training cost is not a concern because it is done before deployment.

We also note that there are many other applications in industry where the training cost is negligible in comparison to the inference cost.

## 6.4 Experiments

To empirically validate our method, we perform an array of experiments. We start with the three MONK data sets and continue to the Adult Census and Breast Cancer data sets. For each experiment, we compare our method to other methods in terms of model memory footprint, evaluation speed, and accuracy. To demonstrate that our method also performs well on small-scale image recognition, we benchmark it on the MNIST as well as the CIFAR-10 data sets. We benchmark the speeds and computational complexity of our method in comparison to baselines. Finally, we investigate the distributions of logic gates for each layer.

### 6.4.1 MONK’s Problems

The MONK’s problems [238] are 3 classic machine learning tasks that have been used to benchmark learning algorithms. They consist of 3 binary classification tasks on a data set with 6 attributes with 2 – 4 possible values each. Correspondingly, the data points can be encoded as binary vectors of size 17. In Table 6.2, we show the performance of our method, a regular neural network, and a few of the original learning methods that have been benchmarked. We give the prediction speed for a single CPU thread, the number of parameters, and storage requirements.

On all three data sets, our method performs better than logistic regression and on MONK-3 (which is the data set with label noise) our method even outperforms the much larger neural network.

**Table 6.2:** Results on the MONK data sets. The inference times are per data point for 1 CPU thread. Averaged over 10 runs. For Diff Logic Nets, # Parameters and Space vary between the MONK data sets as we use different architectures.

| Method                             | MONK-1        | MONK-2    | MONK-3          |
|------------------------------------|---------------|-----------|-----------------|
| Decision Tree Learner (ID3) [239]  | 98.6%         | 67.9%     | 94.4%           |
| Decision Tree Learner (C4.5) [240] | 100%          | 70.4%     | 100%            |
| Rule Learner (CN2) [233]           | 100%          | 69.0%     | 89.1%           |
| Logistic Regression                | 71.1%         | 61.4%     | 97.0%           |
| Neural Network                     | 100%          | 100%      | 93.5%           |
| Diff Logic Net ( <i>ours</i> )     | 100%          | 90.9%     | 97.7%           |
|                                    | # Parameters  | Inf. Time | Space           |
| Decision Tree Learner              | $\approx 30$  | 49ns      | $\approx 60B$   |
| Logistic Regression                | 20            | 68ns      | 80B             |
| Neural Network                     | 162           | 152ns     | 648B            |
| Diff Logic Net ( <i>ours</i> )     | 144   72   72 | 18ns      | 72B   36B   36B |

## 6.4.2 Adult and Breast Cancer Data Sets

For our second set of experiments, we consider the Adult Census [241] and the Breast Cancer data set [242]. We find that our method performs very similar to neural networks and logistic regression on the Adult data set while achieving a much faster inference speed. On the Breast Cancer data set, our method achieves the best performance while still being the fastest model. We present the results in Table 6.3.

**Table 6.3:** Results for the Adult and Breast Cancer data sets averaged over 10 runs.

| <b>Adult</b>                   | Acc.  | # Param.      | Infer. Time | Space          |
|--------------------------------|-------|---------------|-------------|----------------|
| Decision Tree Learner          | 79.5% | $\approx 50$  | 86ns        | $\approx 130B$ |
| Logistic Regression            | 84.8% | 234           | 63ns        | 936B           |
| Neural Network                 | 84.9% | 3810          | 635ns       | 15KB           |
| Diff Logic Net ( <i>ours</i> ) | 84.8% | 1280          | 5.1ns       | 640B           |
| <b>Breast Cancer</b>           | Acc.  | # Param.      | Infer. Time | Space          |
| Decision Tree Learner          | 71.9% | $\approx 100$ | 82ns        | $\approx 230B$ |
| Logistic Regression            | 72.9% | 104           | 34ns        | 416B           |
| Neural Network                 | 75.3% | 434           | 130ns       | 1.4KB          |
| Diff Logic Net ( <i>ours</i> ) | 76.1% | 640           | 2.8ns       | 320B           |

## 6.4.3 MNIST

For our comparison to the fastest neural networks, we start by considering MNIST [160].

Qin *et al.* [219] give a current overview of binary neural networks in their survey. They discuss the challenges of training binary neural networks or translating existing neural networks into their binarized counterparts. They identify FINN by Umuroglu *et al.* [243] as the fastest method for classifying MNIST at an accuracy of 98.4% at a frame rate of 1 561 000 images per second on specialized FPGA hardware. FPGAs (field-programmable gate arrays) are configurable hardware accelerated processors that can achieve extreme speeds for fixed, predefined tasks that are expressed via logic gates. Note that, in the context of FINN, these logic gates are not learned but rather used as a low-precision approximation of neural networks. In comparison to FINN, our logic network achieves better performance while requiring less than 10% of the number of binary operations. That is, our model is objectively more than  $10\times$  cheaper to evaluate. When comparing real times, for an NVIDIA A6000 GPU, our model is  $12\times$  faster than the model by Umuroglu *et al.* [243] on their specialized FPGA hardware, even though our model only achieves a 7% utilization of the GPU. For the other BNNs, OPs

**Table 6.4:** Results for MNIST, all of our results are averaged over 10 runs. Times (T.) are inference times per image, the GPU is an NVIDIA A6000, and the CPU is a single thread at 2.5 GHz. For our experiments, i.e., the top block, we use binarized MNIST.

| MNIST                           | Acc.   | # Param.          | Space | T. [CPU]            | T. [GPU]          | OPs    | FLOPs |
|---------------------------------|--------|-------------------|-------|---------------------|-------------------|--------|-------|
| Linear Regression               | 91.6%  | 4 010             | 16KB  | $3\mu\text{s}$      | 2.4ns             | (4M)   | 4K    |
| Neural Network ( <i>small</i> ) | 97.92% | 118 282           | 462KB | $14\mu\text{s}$     | 12.4ns            | (236M) | 236K  |
| Neural Network                  | 98.40% | 22 609 930        | 86MB  | 2.2ms               | 819ns             | (45G)  | 45M   |
| Diff Logic Net ( <i>small</i> ) | 97.69% | 48 000            | 23KB  | 625ns               | 6.3ns             | 48K    |       |
| Diff Logic Net                  | 98.47% | 384 000           | 188KB | $7\mu\text{s}$      | (50ns)            | 384K   |       |
| <i>Binary Neural Networks</i>   |        |                   |       | T. [FPGA]           |                   |        |       |
| FINN [243]                      | 98.40% |                   |       | (96 $\mu\text{s}$ ) | 641ns             | 5.28M  |       |
| BinaryEye [244]                 | 98.40% |                   |       |                     | 50 $\mu\text{s}$  |        |       |
| ReBNet [245]                    | 98.29% |                   |       |                     | 3 $\mu\text{s}$   |        |       |
| LowBitNN [246]                  | 99.2%  |                   |       |                     | 152 $\mu\text{s}$ |        |       |
| <i>Sparse Neural Networks</i>   |        |                   |       | Sparsity            |                   |        |       |
| Var. Dropout [247]              | 98.08% | 4 000             |       |                     | 98.5%             | (8M)   | 8K    |
| $L_0$ regularization [248]      | 98.6%  |                   |       |                     | 2/3               | (200M) | 200K  |
| SET-MLP [230]                   | 98.74% | 89 797            |       |                     | 96.8%             | (180M) | 180K  |
| Sparse Function Net [231]       | 94.2%  | $3 \times 1\,849$ |       |                     |                   |        | > 2K  |

have not been reported, but their inference speed is also substantially slower than FINN.

Specifically, the binary FINN MNIST model by Umuroglu *et al.* [243] requires 5.82 MOPs (Mega binary OPERations) per frame, which means that their FPGA achieves around  $5.82 \cdot 10^6 \cdot 1.561 \cdot 10^6 = 9.09 \cdot 10^{12}$  binary operations per second, i.e., 9.09 TOPS (Tera binary OPERations per Second). On different FPGAs, Ghasemzadeh *et al.* [245] achieve 330 000 images per second on MNIST at an accuracy of 98.29%, and Jokic *et al.* [244] propose an FPGA based embedded camera system achieving 20 000 images per second at 98.4% accuracy.

When compared to the smallest sparse neural network, our model requires substantially fewer operations than each of the baselines. Specifically, Hoefler *et al.* [220] give an overview of sparsity on deep learning in their recent literature review. They identify Molchanov *et al.* [247] to achieve the sparsest (originally fully-connected) model on MNIST with a sparsity of 98.5% achieving an accuracy of 98.08%. For this, Molchanov *et al.* [247] propose variational dropout with unbounded dropout rates to sparsify neural networks. Louizos *et al.* [248] propose sparsification via  $L_0$  regularization and report an MNIST accuracy of 98.6% for a model with around  $2 \cdot 10^5$  FLOPs (Floating point OPERations), which corresponds to a sparsity of around 2/3.

Zhan *et al.* [246] concentrate on deploying Low-Bit Neural Networks (LBNNs) on FPGAs and achieve an accuracy of 99.2% on MNIST at 6 580 images per second. Shani *et al.* [253] explore analog logic gate networks (a physical approximation to boolean networks) and achieve accuracies up to 89% on MNIST.

Sparse function networks [231], which have been learned evolutionarily, achieve an accuracy of 94.2%.

A FLOP generally corresponds to many binary OPs. Specifically, a float32 adder / multiplier requires usually at least 1 000 logical gates or look up tables and usually has a delay of tens of logical levels. Practically, float32 adders / multipliers are implemented directly in hardware in CPUs and GPUs, as it is an essential operation on such platforms. Nevertheless, also in practice, a float32 adder / multiplier is much more expensive than performing a bit-wise logical operation on int64 data types (even on float32 and int32 focused GPUs). On CPUs, around 3 – 10 int64 bit-wise operations can be performed per cycle, while floating-point operations usually require a full clock cycle. To convert a non-sparse model, we assume a very conservative 100 OPs per 1 FLOP. Note that speeds for sparse neural networks are also only theoretical because sparse execution usually brings an overhead of factor 10 – 100 $\times$ . So overall, in practice, 1 000 (binary) OPs per 1 sparse (float32) FLOP is a very conservative estimate in favor of sparse float32 models. Further, in theory, 1 000 OPs per 1 FLOP is an accurate estimate (assuming sparsity to come without cost and assuming floating-point operations to not be hardware accelerated).

**Table 6.5:** Results on CIFAR-10. Times (T.) are inference times per image, the GPU is an NVIDIA A6000, and the CPU is a single thread at 2.5 GHz. For our experiments, i.e., the top block, we use a color-channel resolution of 4 for the first 3 lines and a color-channel resolution of 32 for the *large* models. The other baselines were provided with the full resolution of 256 color-channel values. The numbers in parentheses are extrapolated / estimated.

| CIFAR-10                                                         | Acc.   | # Param | Space  | T. [CPU]     | T. [GPU] | OPs    | FLOPs  |
|------------------------------------------------------------------|--------|---------|--------|--------------|----------|--------|--------|
| Neural Network (color-channel res. = 4)                          | 50.79% | 12.6M   | 48MB   | 1.2ms        | 370ns    | (25G)  | 25M    |
| Diff Logic Net ( <i>small</i> )                                  | 51.27% | 48K     | 24KB   | 1.3 $\mu$ s  | 19ns     | 48K    |        |
| Diff Logic Net ( <i>medium</i> )                                 | 57.39% | 512K    | 250KB  | 7.3 $\mu$ s  | 29ns     | 512K   |        |
| Diff Logic Net ( <i>large</i> )                                  | 60.78% | 1.28M   | 625KB  | (18 $\mu$ s) | (73ns)   | 1.28M  |        |
| Diff Logic Net ( <i>large</i> $\times$ 2)                        | 61.41% | 2.56M   | 1.22MB | (37 $\mu$ s) | (145ns)  | 2.56M  |        |
| Diff Logic Net ( <i>large</i> $\times$ 4)                        | 62.14% | 5.12M   | 2.44MB | (73 $\mu$ s) | (290ns)  | 5.12M  |        |
| <i>Best Fully-Connected Baselines (color-channel res. = 256)</i> |        |         |        |              |          |        |        |
| Regularized SReLU NN [230]                                       | 68.70% | 20.3M   | 77MB   | 1.9ms        | 565ns    | (40G)  | 40M    |
| Student-Teacher NN [249]                                         | 65.8%  | 1M      | 4MB    | 112 $\mu$ s  | 243ns    | (2G)   | 2M     |
| Student-Teacher NN [249]                                         | 74.3%  | 31.6M   | 121MB  | 2.9ms        | 960ns    | (63G)  | 63M    |
| <i>Sparse Neural Networks</i>                                    |        |         |        |              | Sparsity |        |        |
| PBW (ResNet32) [250]                                             | 38.64% |         |        |              | 99.9%    | (140M) | (140K) |
| MLPrune (ResNet32) [251]                                         | 36.09% |         |        |              | 99.9%    | (140M) | (140K) |
| ProbMask (ResNet32) [252]                                        | 76.87% |         |        |              | 99.9%    | (140M) | (140K) |
| SET-MLP [230]                                                    | 74.84% | 279K    | 4.7MB  |              | 98.6%    | (558M) | 558K   |

#### 6.4.4 CIFAR-10

In addition to MNIST, we also benchmark our method on CIFAR-10 [183]. For CIFAR-10, we reduce the color-channel resolution of the CIFAR-10 images and employ a binary embedding: For a color-channel resolution of 4 (the first three rows of Table 6.5), we use three binary values with the three thresholds 0.25, 0.5, and 0.75. For a color-channel resolution of 32 (the large models, i.e., the last three rows of the top block of Table 6.5), we use 31 binary values with thresholds  $(i/32)_{i \in \{1..31\}}$ . We do not apply data augmentation / dropout for our experiments, which could additionally improve performance. For all baselines, we copied the reported accuracies, and thus those results are with data augmentation and the original color-channel resolution, and may include dropout [254] as well as other techniques such as student-teacher learning with a convolutional teacher [249].

The results are displayed in Table 6.5. We find that our method outperforms neural networks in the first setting (color-channel resolution of 4) by a small margin, while requiring less than 0.1% of the memory footprint and (with a larger model) by a large margin while requiring less than 1% of the memory footprint. In comparison to the best fully-connected neural network baselines, which are trained with various tricks such as student-teacher learning and obtain the full color-channel resolution, our model does not achieve the same performance, while it is also much smaller and has access to fewer data. With 1 million parameters, the student-teacher model [249] has a footprint that is about 64% larger than the footprint of our largest model (*large* $\times$ 4), and achieves an accuracy which is only 3.7% better than ours. It is important to note that this models requires 2 million floating-point operations, while our model requires 5 million bit-wise logic operations (before pruning/optimization). On float-arithmetic hardware-accelerated integrated circuits (as current GPUs and many CPUs), the 2 million floating-point operations are around  $100\times$  slower than 5 million bit-wise logic operations. On general purpose hardware (i.e., without float acceleration) the speed difference would be one order of magnitude larger, i.e.,  $1000\times$ .

The results in parentheses are estimated because compilation to binaries did not finish / for GPU the largest models could also not be compiled due to compiler limitations. This can be resolved if desired with moderate implementation effort, e.g., compiling the model directly to PTX (CUDA Assembly) without compiling via gcc and nvcc. The actual problem is that the compilers used by the implementation have a quadratic compile time in the number of lines of code / statements.

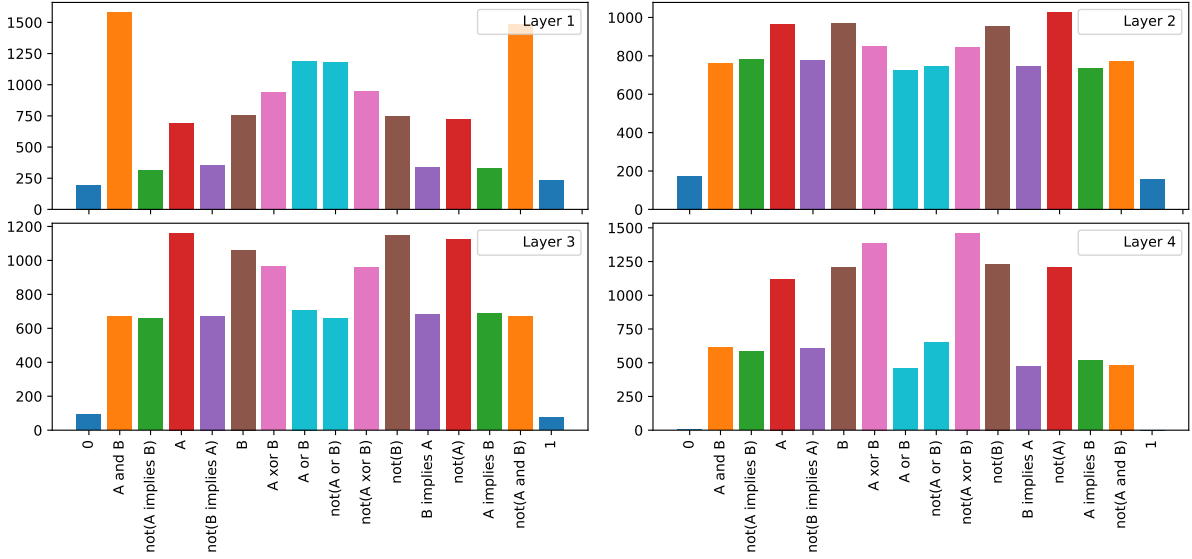


**Figure 6.2:** Distribution of logic gates in a trained four layer logic network.

More competitive with respect to speed are sparse neural networks. In the final block of Table 6.5, we report the sparsest models for CIFAR-10. Note that two of the methods resulted in performances below 50%, and even those methods which achieve around 75% accuracy require a significantly more expensive inference.

Zhou *et al.* [252] proposed ProbMask and give an overview over the sparsest CIFAR-10 models. Specifically, they report up to a sparsity of 99.9%, which corresponds to around 140 kFLOPs for their smallest network architecture (ResNet32, which has a base cost of 140 MFLOPs [255].) This model is (depending on hardware) 1 – 2 orders of magnitude more expensive than our largest model. Other architectures reported in the literature are substantially less sparse and, therefore, more expensive. While these models have the advantage of being based on the VGG and ResNet CNN architectures, our models are still very competitive, especially considering that our models are much smaller than their smallest reported results.

Blalock *et al.* [256] report in their survey that for CIFAR-10 with a VGG, to achieve a theoretical speedup of  $32\times$ , all evaluated methods drop significantly below 70% test accuracy. Note that this speedup corresponds to a sparsity of around 97%, which makes up a much larger model than the models considered in this chapter.

Mocanu *et al.* [230] propose training neural networks with sparse evolutionary training inspired by network science. Their method evolves an initial sparse topology of two consecutive layers of neurons into a scale-free topology. They achieve an accuracy of 74.84% on CIFAR-10 with 278 630 floating-point parameters. We estimate this to correspond to a theoretical cost of around 550 kFLOPs (multiplication + addition), corresponding to 550 MOPs as per our conservative estimate. On MNIST, they achieve (with 89 797 parameters) an accuracy of 98.74%. The model is by two orders of magnitude more expensive to evaluate than the largest logic gate network considered in this chapter.

### 6.4.5 Distribution of Logic Gates

To gain additional insight into learned logic gate networks, we consider histograms of operators present in each layer of a trained model. Specifically, we consider a 4 layer CIFAR-10 model with 12 000 neurons per layer in Figure 6.2.

We observe that, generally, the constant 0/1 “operator” is learned to be used only very infrequently as it does not actually provide value to the model. Especially interesting is that it does not occur at all in the last layer. In the first layer, we observe a stronger presence of ‘and’, ‘nand’, ‘or’, and ‘nor’. In the second and third layers, there are more ‘ $A$ ’, ‘ $B$ ’, ‘ $\neg A$ ’, and ‘ $\neg B$ ’s, which can be seen as a residual / direct connection. This enables the network to model lower-order dependencies more efficiently by expressing it with fewer layers than the predefined number of layers. In the last layer, the most frequent operations are ‘xor’ and ‘xnor’, which can create conditional dependencies of activations of the previous layers. Interestingly, however, implications (e.g.,  $A \Rightarrow B$ ) are only infrequently used.

### 6.4.6 Hyperparameters and Model Architectures

In each experiment, we used so-called *straight* architectures, i.e., architectures with the same number of neurons per layer. In Tables 6.6 and 6.7, we display the numbers of layers numbers of neurons per layer for each network architecture. The architecture search for all models was performed via grid search with the number of layers in  $\{2, 3, 4, 5, 6, 7, 8, 9, 10\}$  and number of neurons per layer with a resolution of factor 2.

**Table 6.6:** Logic gate network architectures.

| Data set      | Model            | Layers | Neurons / layer | Total num. of p. | $\tau$  |
|---------------|------------------|--------|-----------------|------------------|---------|
| MONK-1        | —                | 6      | 24              | 144              | 1       |
| MONK-2        | —                | 6      | 12              | 72               | 1       |
| MONK-3        | —                | 6      | 12              | 72               | 1       |
| Adult         | —                | 5      | 256             | 1 280            | 1/0.075 |
| Breast Cancer | —                | 5      | 128             | 640              | 1/0.1   |
| MNIST         | small            | 6      | 8 000           | 48 000           | 1/0.1   |
|               | normal           | 6      | 64 000          | 384 000          | 1/0.03  |
| CIFAR-10      | small            | 4      | 12 000          | 48 000           | 1/0.03  |
|               | medium           | 4      | 128 000         | 512 000          | 1/0.01  |
|               | large            | 5      | 256 000         | 1 280 000        | 1/0.01  |
|               | large $\times 2$ | 5      | 512 000         | 2 560 000        | 1/0.01  |
|               | large $\times 4$ | 5      | 1 024 000       | 5 120 000        | 1/0.01  |

For all models, we use the Adam optimizer [34]. For all neural networks, we use a learning rate of 0.001 and for all logic gate networks, we use a learning rate of 0.01. We train all models up to 200 epochs at a batch size of 100. The softmax temperature  $\tau$  was searched over a grid of  $\{1, 1/0.3, 1/0.1, 1/0.03, 1/0.01\}$  (except for Adult). The optimal temperature primarily depends on the number of outputs per class. If there are more outputs per class, the range of predictions is larger, and thus, we use a larger temperature to counter this effect.

| Data set      | Model  | Layers | Neurons / layer | Total num. of parameters |
|---------------|--------|--------|-----------------|--------------------------|
| MONK-1        | —      | 2      | 8               | 162                      |
| MONK-2        | —      | 2      | 8               | 162                      |
| MONK-3        | —      | 2      | 8               | 162                      |
| Adult         | —      | 2      | 32              | 3 810                    |
| Breast Cancer | —      | 2      | 8               | 434                      |
| MNIST         | small  | 3      | 128             | 118 282                  |
|               | normal | 7      | 2 048           | 22 609 930               |
| CIFAR-10      | —      | 5      | 1 024           | 12 597 258               |

**Table 6.7:** Multi-layer perceptron / neural network baseline architectures. All architectures are ReLU activated.

## Conclusion

In this chapter, we leveraged real-valued logics and continuous relaxations to present a novel approach to training logic gate networks, which allows us to effectively train extremely efficient neural networks that—for their level of accuracy—are one or more orders of magnitude more efficient than the state-of-the-art. At the moment, the capabilities of logic gate networks are mostly limited to tasks that can be sufficiently solved with fully-connected neural networks. However, one direction of future research could be convolutional differentiable logic gate networks. A limitation of differentiable logic gate networks is their relatively higher training cost compared to (performance-wise) comparable conventional neural networks; however, since their original development, we have already reduced their training cost by up to 50 times.



# Alternative Optimization Methods

## 7

After discussing the construction of differentiable algorithms and their applications, in this chapter, we present alternative optimization methods. Specifically, we introduce splitting backpropagation, a general two-stage optimization method for combining different iterative optimization methods. Further, we present two instances of splitting backpropagation: First, Newton losses, a method for combining second-order optimization of the loss function and first-order optimization of the model. Second, RESGRO, a simple stochastic optimization method which allows optimizing non-differentiable objectives and can be combined with first-order optimization of the model. In our empirical evaluation, we find that both instances deliver strong empirical performance on algorithmic supervision tasks.

Neural network training has gained a tremendous amount of attention in machine learning in recent years. This is primarily due to the success of backpropagation and stochastic gradient descent for first-order optimization. However, there has also been a strong line of work on second-order optimization for neural network training; see the work by Agrawal *et al.* [257] and references therein. While these second-order optimization methods (such as Newton's method and natural gradient descent) exhibit improved convergence rates and therefore require fewer training steps, they have two major limitations [258], namely (i) computing the curvature (or its inverse) for a large and deep neural network is computationally substantially more expensive than simply computing the gradient with backpropagation, which makes second-order methods practically inapplicable in most cases; (ii) networks trained with second-order information exhibit reduced generalization capabilities [259].

When working with complicated and non-standard loss functions, the optimization problem for neural network training is often challenging. In these cases, a natural approach is to look for a decomposition, i.e., to break up the optimization problem into two (or more) sub-problems, which are then solved sequentially. The idea of decomposing an optimization problem is not new, see [260] for an overview of decomposition methods. A decomposition method, which has become particularly popular in machine learning, is the alternating direction method of multipliers [261].

In this work, we propose a novel method for incorporating second-order information of the loss function into training while training the actual neural network with gradient descent. As loss functions are usually substantially cheaper to evaluate than a neural network, the idea is to apply second-order optimization to the loss function while training the actual neural network with first-order optimization. For this, we decompose the original iterative optimization problem into a two-stage iterative optimization problem, which leads to Newton losses. This is especially interesting for intrinsically hard-to-optimize loss functions, i.e., where second-order optimization of the inputs to the loss is superior to first-order optimization. This includes the algorithmic supervision training objectives / algorithmic losses discussed in this thesis.

|     |                                                                          |     |
|-----|--------------------------------------------------------------------------|-----|
| 7.1 | Related Work . . . . .                                                   | 94  |
| 7.2 | Splitting Back-propagation:<br>A Two-Stage Optimization Method . . . . . | 94  |
| 7.3 | Newton Losses . . . . .                                                  | 98  |
| 7.4 | Regularized Sampling Greedy Optimizer (RESGRO) . . . . .                 | 102 |
| 7.5 | Experiments . . . . .                                                    | 103 |

We evaluate the proposed methods for various algorithmic losses on two benchmarks: the four-digit MNIST sorting benchmark and the Warcraft shortest-path benchmark, following up on Chapters 2 and 3. We find that they improve the performance in the case of hard-to-optimize losses and maintain the original performance in the case of easy-to-optimize losses.

The contributions of this chapter are (i) introducing a mathematical framework for splitting iterative optimization methods into two-stage schemes, which we show to be equal to the original optimization methods; (ii) introducing Newton losses as combinations of first-order and second-order optimization methods; (iii) introducing RESGRO, a stochastic method for optimizing non-differentiable objectives, which can be combined with gradient descent on differentiable sections of the architecture.

In the context of second-order methods, we would also like to mention ISAAC Newton by Petersen *et al.* [14]. Newton Losses can be seen as a complement to ISAAC Newton. While ISAAC Newton integrates second-order information from the input, Newton losses integrate second-order information from the loss into training.

## 7.1 Related Work

The related work specific to this chapter primarily comprises second-order optimization methods. Second-order methods have recently gained popularity in machine learning due to their fast convergence properties when compared to first-order methods [257]. One alternative to vanilla Newton are quasi-Newton methods, which, instead of computing an inverse Hessian in the Newton step (which is expensive), approximate this curvature from the change in gradients [258]. In addition, a number of new approximations to the pre-conditioning matrix have been proposed in the recent literature, i.e., [262–264]. While the vanilla Newton methods rely on the Hessian, there are variants which use the empirical Fisher information matrix, which can coincide in specific cases with the Hessian, but generally exhibits somewhat different behavior. For an overview and discussion of Fisher-based methods (sometimes referred to as natural gradient descent), see [265, 266]. To the best of our knowledge, this is the first work combining second-order optimization of loss functions with first-order optimization of neural networks, especially for algorithmic losses.

## 7.2 Splitting Backpropagation: A Two-Stage Optimization Method

We consider the training of a neural network  $f(x; \theta)$ , where  $x \in \mathbb{R}^n$  is the vector of inputs,  $\theta \in \mathbb{R}^d$  is the vector of weights and  $y = f(x; \theta) \in \mathbb{R}^m$  is the vector of outputs. We assume that we have access to a data set of  $N$  samples drawn from the input distribution, which describes the empirical input  $\mathbf{x} = [x_1, \dots, x_N]^\top \in \mathbb{R}^{N \times n}$ . As per vectorization, we denote  $\mathbf{y} = f(\mathbf{x}; \theta) \in \mathbb{R}^{N \times m}$  as the matrix describing the outputs of the neural network corresponding to the empirical inputs. Further, let  $\ell : \mathbb{R}^{N \times m} \rightarrow \mathbb{R}$  denote the loss function, and let the ground truth output be implicitly encoded in  $\ell$  (because for many algorithmic losses, it is not simply a label, but could be ordinal information). In a general setting, the training of a neural network can be expressed as the optimization problem

$$\arg \min_{\theta \in \Theta} \ell(f(\mathbf{x}; \theta)), \quad (7.1)$$

where  $\Theta \subseteq \mathbb{R}^d$  is the domain of the parameters  $\theta$ , and  $f$  and  $\ell$  are such that the minimum in (7.1) exists. Note that the formulation (7.1) is extremely general and includes, e.g., optimization of non-decomposable loss functions (i.e., not composed of individual losses per training sample), which is relevant for some algorithmic losses like ranking losses.

Typically, the optimization problem (7.1) is solved by using some iterative algorithm like gradient descent (or Newton's method) updating the weights  $\theta$  by repeatedly applying the following step:

$$\theta \leftarrow \text{One optimization step of } \ell(f(\mathbf{x}; \theta)) \text{ wrt. } \theta. \quad (7.2)$$

However, in this work, we consider decomposing the optimization problem (7.1) into two problems, which may be solved by applying the following two updates in an alternating fashion:

$$\mathbf{z}^* \leftarrow \text{One optimization step of } \ell(\mathbf{z}) \text{ wrt. } \mathbf{z} = f(\mathbf{x}; \theta), \quad (7.3a)$$

$$\theta \leftarrow \text{One optimization step of } \frac{1}{2} \|\mathbf{z}^* - f(\mathbf{x}; \theta)\|_2^2 \text{ wrt. } \theta. \quad (7.3b)$$

This split allows us later to use two different iterative optimization algorithms for (7.3a) and (7.3b), respectively. This is especially interesting for optimization problems where the loss function  $\ell$  is non-convex and its minimization is a difficult optimization problem itself, and as such those problems, where a stronger optimization method exhibits a superior rate of convergence.

We can express individual optimization steps (corr. to (7.2) and (7.3a)) via

$$\theta \leftarrow \arg \min_{\theta' \in \Theta} \ell(f(\mathbf{x}; \theta')) + \Omega(\theta', \theta), \quad (7.4)$$

$$\mathbf{z}^* \leftarrow \arg \min_{\mathbf{z} \in \mathcal{Y}} \ell(\mathbf{z}) + \Omega(\mathbf{z}, f(\mathbf{x}; \theta)) \quad (7.5)$$

where  $\Omega$  is a regularizer such that one step of a respective optimization method corresponds to the global optimum of the regularized optimization problems in (7.4) and (7.5). The regularizer  $\Omega$  has the standard property that  $\Omega(a, b) = 0$  for any  $a = b$ . Note that the explicit form of the regularizer  $\Omega$  does not need to be known. Nevertheless, in Supplementary Material 7.2, we discuss explicit choices of  $\Omega$ .

This allows us to express the set of points of convergence for the iterative optimization methods. Recall that an iterative optimization method has converged if it has arrived at a fixed point, i.e., the parameters do not change when applying an update. The set of points of convergence for (7.2) is

$$\mathcal{A} = \left\{ \theta \mid \theta \in \arg \min_{\theta'} \ell(f(\mathbf{x}; \theta')) + \Omega(\theta', \theta) \right\}, \quad (7.6)$$

i.e., those points at which the update does not change  $\theta$ . For the two-stage optimization method (7.3), the set of points of convergence is

$$\mathcal{B} = \left\{ \theta \mid f(\mathbf{x}; \theta) = \mathbf{z}^* \in \arg \min_{\mathbf{z}} \ell(\mathbf{z}) + \Omega(\mathbf{z}, f(\mathbf{x}; \theta)) \right\} \quad (7.7)$$

as the method has converged if the update (7.3a) yields  $\mathbf{z}^* = \mathbf{z}$  because the subsequent update (7.3b) will not change  $\theta$  as  $\mathbf{z}^* = \mathbf{z} = f(\mathbf{x}; \theta)$  already holds, and thus  $\frac{1}{2} \|\mathbf{z}^* - f(\mathbf{x}; \theta)\|_2^2 = 0$ . Now, we show that the iterative method (7.2) and the alternating method (7.3) lead to the same sets of convergence points.

The split does not need to be between loss and neural network. We choose this location to split because it is meaningful for algorithmic losses; however, this shall not restrict us to this case. Further, we note that the method can be adapted to more than one split, and the theory for this case is analogous, as can be seen via induction.

**Lemma 7.1** (Equality of the Sets of Convergence Points) *The set  $\mathcal{A}$  of points of convergence obtained by the iterative optimization method (7.2) is equal to the set  $\mathcal{B}$  of points of convergence obtained by the two-step iterative optimization method (7.3).*

*Proof.* ( $\mathcal{A} \subset \mathcal{B}$ ) First, we show that any point in  $\mathcal{A}$  also lies in  $\mathcal{B}$ . By definition, for each point in  $\mathcal{A}$ , the optimization step (7.2) does not change  $\theta$ , i.e.,  $\theta' = \theta$ . Thus,  $f(\mathbf{x}; \theta) = f(\mathbf{x}; \theta') \in \arg \min_{\mathbf{z}} \ell(\mathbf{z}) + \Omega(\mathbf{z}, f(\mathbf{x}; \theta))$ , and therefore  $\theta \in \mathcal{B}$ .

( $\mathcal{B} \subset \mathcal{A}$ ) Second, we show that any point in  $\mathcal{B}$  also lies in  $\mathcal{A}$ . For each  $\theta \in \mathcal{B}$ , we know that, by definition,  $f(\mathbf{x}; \theta) = \mathbf{z}^* \in \arg \min_{\mathbf{z}} \ell(\mathbf{z}) + \Omega(\mathbf{z}, f(\mathbf{x}; \theta))$ , therefore  $\theta \in \arg \min_{\theta'} \ell(f(\mathbf{x}; \theta')) + \Omega(f(\mathbf{x}; \theta'), f(\mathbf{x}; \theta))$  where  $\Omega(f(\mathbf{x}; \theta), f(\mathbf{x}; \theta)) = 0 = \Omega(\theta, \theta)$ , and, therefore  $\theta \in \mathcal{A}$ .  $\square$

While Lemma 7.1 states the equivalence of the original training (7.2) and its counterpart (7.3) wrt. their possible points of convergence (i.e., solutions) for an arbitrary choice of the iterative method, the two approaches are also equal when applying standard first-order or second-order optimization schemes. In other words, running a gradient descent step according to (7.2) coincides with two gradient steps of the alternating scheme (7.3a) and (7.3b), namely one step for (7.3a) and one step for (7.3b).

**Theorem 7.2** (Gradient Descent Step Equality between (7.2) and (7.3a) + (7.3b)) *A gradient descent step according to (7.2) with arbitrary step size  $\eta$  coincides with two gradient descent steps according to (7.3a) and (7.3b), where the optimization over  $\theta$  has a step size of  $\eta$  and the optimization over  $\mathbf{z}$  has a unit step size.*

*Proof.* Let  $\theta \in \Theta$  be the current parameter vector and let  $\mathbf{z} = f(\mathbf{x}; \theta)$ . Then, the gradient descent steps according to (7.3a) and (7.3b) with step sizes 1 and  $\eta > 0$  are expressed as

$$\mathbf{z} \leftarrow \mathbf{z} - \nabla_{\mathbf{z}} \ell(\mathbf{z}) = f(\mathbf{x}; \theta) - \nabla_f \ell(f(\mathbf{x}; \theta)) \quad (7.8)$$

$$\theta \leftarrow \theta - \eta \nabla_{\theta} \frac{1}{2} \|\mathbf{z} - f(\mathbf{x}; \theta)\|_2^2 = \theta - \eta \frac{\partial f(\mathbf{x}; \theta)}{\partial \theta} \cdot (f(\mathbf{x}; \theta) - \mathbf{z}). \quad (7.9)$$

Combining (7.8) and (7.9) eventually leads to

$$\theta \leftarrow \theta - \eta \frac{\partial f(\mathbf{x}; \theta)}{\partial \theta} \cdot (f(\mathbf{x}; \theta) - f(\mathbf{x}; \theta) + \nabla_f \ell(f(\mathbf{x}; \theta))) = \theta - \eta \nabla_{\theta} \ell(f(\mathbf{x}; \theta)), \quad (7.10)$$

which is exactly a gradient descent step of problem (7.1) starting at  $\theta \in \Theta$  with step size  $\eta$ .  $\square$

**Theorem 7.3** (Newton Step Equality between (7.2) and (7.3a)+(7.3b)) *In the case of  $m = 1$ , a Newton step according to (7.2) with arbitrary step size  $\eta$  coincides with two Newton steps according to (7.3a) and (7.3b), where the optimization over  $\theta$  has a step size of  $\eta$  and the optimization over  $\mathbf{z}$  has a unit step size.*

*Proof.* Let  $\theta \in \Theta$  be the current parameter vector and let  $\mathbf{z} = f(\mathbf{x}; \theta)$ . Then, applying Newton steps according to (7.3a) and (7.3b) leads to

$$\mathbf{z} \leftarrow \mathbf{z} - (\nabla_{\mathbf{z}}^2 \ell(\mathbf{z}))^{-1} \nabla_{\mathbf{z}} \ell(\mathbf{z}) = f(\mathbf{x}; \theta) - (\nabla_f^2 \ell(f(\mathbf{x}; \theta)))^{-1} \nabla_f \ell(f(\mathbf{x}; \theta)) \quad (7.11)$$

$$\theta \leftarrow \theta - \eta \left( \nabla_{\theta}^2 \frac{1}{2} \|\mathbf{z} - f(\mathbf{x}; \theta)\|_2^2 \right)^{-1} \nabla_{\theta} \frac{1}{2} \|\mathbf{z} - f(\mathbf{x}; \theta)\|_2^2 \quad (7.12)$$

$$= \theta - \eta \left( \frac{\partial}{\partial \theta} \left[ \frac{\partial f(\mathbf{x}; \theta)}{\partial \theta} \cdot (f(\mathbf{x}; \theta) - \mathbf{z}) \right] \right)^{-1} \frac{\partial f(\mathbf{x}; \theta)}{\partial \theta} \cdot (f(\mathbf{x}; \theta) - \mathbf{z}) \quad (7.13)$$

$$= \theta - \eta \left( \frac{\partial}{\partial \theta} \left[ \frac{\partial f(\mathbf{x}; \theta)}{\partial \theta} \right] (f(\mathbf{x}; \theta) - \mathbf{z}) + \left( \frac{\partial f(\mathbf{x}; \theta)}{\partial \theta} \right)^2 \right)^{-1} \cdot \frac{\partial f(\mathbf{x}; \theta)}{\partial \theta} \cdot (f(\mathbf{x}; \theta) - \mathbf{z}) \quad (7.14)$$

Inserting (7.11), we can rephrase the update above as

$$\begin{aligned} \theta \leftarrow \theta - \eta \left( \frac{\partial}{\partial \theta} \left[ \frac{\partial f(\mathbf{x}; \theta)}{\partial \theta} \right] (\nabla_f^2 \ell(f(\mathbf{x}; \theta)))^{-1} \nabla_f \ell(f(\mathbf{x}; \theta)) + \left( \frac{\partial f(\mathbf{x}; \theta)}{\partial \theta} \right)^2 \right)^{-1} \\ \cdot \frac{\partial f(\mathbf{x}; \theta)}{\partial \theta} \cdot (\nabla_f^2 \ell(f(\mathbf{x}; \theta)))^{-1} \nabla_f \ell(f(\mathbf{x}; \theta)) \end{aligned} \quad (7.15)$$

By applying the chain rule twice, we further obtain

$$\nabla_\theta^2 \ell(f(\mathbf{x}; \theta)) = \frac{\partial}{\partial \theta} \left[ \frac{\partial f(\mathbf{x}; \theta)}{\partial \theta} \right] \nabla_f \ell(f(\mathbf{x}; \theta)) + \left( \frac{\partial f(\mathbf{x}; \theta)}{\partial \theta} \right)^2 \nabla_f^2 \ell(f(\mathbf{x}; \theta)),$$

which allows us to rewrite (7.15) as

$$\begin{aligned} \theta' &= \theta - ((\nabla_f^2 \ell(f(\mathbf{x}; \theta)))^{-1} \nabla_\theta^2 \ell(f(\mathbf{x}; \theta)))^{-1} (\nabla_f^2 \ell(f(\mathbf{x}; \theta)))^{-1} \nabla_\theta \ell(f(\mathbf{x}; \theta)) \\ &= \theta - (\nabla_\theta^2 \ell(f(\mathbf{x}; \theta)))^{-1} \nabla_\theta \ell(f(\mathbf{x}; \theta)), \end{aligned}$$

which is precisely one Newton step of problem (7.1) starting at  $\theta \in \Theta$ .  $\square$

## Explicit Forms of the Regularization Term $\Omega$

For the regularization term  $\Omega$  in (7.4) and (7.5), an explicit form does not need to be known nor computable. For the cases of a gradient descent step as well as a Newton step, we discuss explicit forms of  $\Omega$ :

For an update on  $\mathbf{z}$  (7.3a) corresponding to a *gradient descent step*, i.e.,

$$\mathbf{z}^* \leftarrow \mathbf{z} - \eta \nabla \ell(\mathbf{z}), \quad \mathbf{z} = f(\mathbf{x}; \theta), \quad (7.16)$$

we can explicitly choose the regularization term  $\Omega$  as

$$\begin{aligned} \Omega(\mathbf{z}, f(\mathbf{x}; \theta)) &= \ell(f(\mathbf{x}; \theta)) + \eta \nabla_f \ell(f(\mathbf{x}; \theta))^\top (\mathbf{z} - f(\mathbf{x}; \theta)) - \ell(\mathbf{z}) \\ &\quad + \frac{1}{2} (\mathbf{z} - f(\mathbf{x}; \theta))^\top (\mathbf{z} - f(\mathbf{x}; \theta)). \end{aligned} \quad (7.17)$$

Here, the first-order optimality conditions for  $\min_{\mathbf{z}} \ell(\mathbf{z}) + \Omega(\mathbf{z}, f(\mathbf{x}; \theta))$  are

$$\nabla_{\mathbf{z}} (\ell(\mathbf{z}) + \Omega(\mathbf{z}, f(\mathbf{x}; \theta))) = \eta \nabla_f \ell(f(\mathbf{x}; \theta)) + \mathbf{z} - f(\mathbf{x}; \theta) = 0, \quad (7.18)$$

leading to the gradient step  $\mathbf{z} = f(\mathbf{x}; \theta) - \eta \nabla_f \ell(f(\mathbf{x}; \theta))$ .

In the case, where the update on  $\mathbf{z}$  (7.3a) corresponds to a *Newton step*, i.e.,

$$\mathbf{z}^* \leftarrow \mathbf{z} - \eta (\nabla^2 \ell(\mathbf{z}))^{-1} \nabla \ell(\mathbf{z}), \quad \mathbf{z} = f(\mathbf{x}; \theta), \quad (7.19)$$

we can explicitly choose the regularization term  $\Omega$  as

$$\begin{aligned} \Omega(\mathbf{z}, f(\mathbf{x}; \theta)) &= \ell(f(\mathbf{x}; \theta)) + \eta (\mathbf{z} - f(\mathbf{x}; \theta))^\top \nabla_f \ell(f(\mathbf{x}; \theta)) \\ &\quad + \frac{1}{2} (\mathbf{z} - f(\mathbf{x}; \theta))^\top \nabla_f^2 \ell(f(\mathbf{x}; \theta)) (\mathbf{z} - f(\mathbf{x}; \theta)) - \ell(\mathbf{z}). \end{aligned} \quad (7.20)$$

The first-order optimality conditions for  $\min_{\mathbf{z}} \ell(\mathbf{z}) + \Omega(\mathbf{z}, f(\mathbf{x}; \theta))$  are

$$\begin{aligned} \nabla_{\mathbf{z}} (\ell(\mathbf{z}) + \Omega(\mathbf{z}, f(\mathbf{x}; \theta))) \\ = \eta \nabla_f \ell(f(\mathbf{x}; \theta)) + \nabla_f^2 \ell(f(\mathbf{x}; \theta)) (\mathbf{z} - f(\mathbf{x}; \theta)) = 0, \end{aligned} \quad (7.21)$$

leading to the Newton step  $\mathbf{z} = f(\mathbf{x}; \theta) - \eta (\nabla^2 \ell(f(\mathbf{x}; \theta)))^{-1} \nabla \ell(f(\mathbf{x}; \theta))$ .

### 7.3 Newton Losses

In this section, we build on the two-stage optimization method (7.3a)+(7.3b), and propose optimizing (7.3a) with Newton's method while optimizing (7.3b) with stochastic gradient descent. Let us begin by considering the quadratic approximation of the loss function at the location  $\mathbf{y} = f(\mathbf{x}; \theta)$ , i.e.,

$$\tilde{\ell}_{\mathbf{y}}(\mathbf{z}) = \ell(\mathbf{y}) + (\mathbf{z} - \mathbf{y})^\top \nabla_{\mathbf{y}} \ell(\mathbf{y}) + \frac{1}{2} (\mathbf{z} - \mathbf{y})^\top \nabla_{\mathbf{y}}^2 \ell(\mathbf{y}) (\mathbf{z} - \mathbf{y}). \quad (7.22)$$

To find the location  $\mathbf{z}^*$  of the minimum of  $\tilde{\ell}_{\mathbf{y}}(\mathbf{z})$ , we set its derivative to 0:

$$\begin{aligned} \nabla_{\mathbf{z}} \tilde{\ell}_{\mathbf{y}}(\mathbf{z}^*) = 0 &\Leftrightarrow \nabla_{\mathbf{y}} \ell(\mathbf{y}) + \nabla_{\mathbf{y}}^2 \ell(\mathbf{y}) (\mathbf{z}^* - \mathbf{y}) = 0 \\ &\Leftrightarrow \nabla_{\mathbf{y}} \ell(\mathbf{y}) = -\nabla_{\mathbf{y}}^2 \ell(\mathbf{y}) (\mathbf{z}^* - \mathbf{y}) \\ &\Leftrightarrow -(\nabla_{\mathbf{y}}^2 \ell(\mathbf{y}))^{-1} \nabla_{\mathbf{y}} \ell(\mathbf{y}) = \mathbf{z}^* - \mathbf{y} \end{aligned} \quad (7.23)$$

Thus, the minimum of  $\tilde{\ell}_{\mathbf{y}}(\mathbf{z})$  is

$$\mathbf{z}^* = \arg \min_{\mathbf{z}} \tilde{\ell}_{\mathbf{y}}(\mathbf{z}) = \mathbf{y} - (\nabla_{\mathbf{y}}^2 \ell(\mathbf{y}))^{-1} \nabla_{\mathbf{y}} \ell(\mathbf{y}). \quad (7.24)$$

When  $\ell$  is quadratic, it can be readily seen that  $\mathbf{z}^*$  is independent of the choice of  $\mathbf{y}$ . For non-quadratic functions, we heuristically assume independence as  $\mathbf{z}^*$  is the projected optimum / goal of the function  $\ell$ . In implementations, this independence can be achieved using `.detach()` or `.stop_gradient()`. Using  $\mathbf{z}^*$ , we can derive the Newton loss  $\ell^*$  as

$$\ell_{\mathbf{z}^*}^*(\mathbf{y}) = \frac{1}{2} (\mathbf{z}^* - \mathbf{y})^\top (\mathbf{z}^* - \mathbf{y}) = \frac{1}{2} \|\mathbf{z}^* - \mathbf{y}\|_2^2 \quad (7.25)$$

where

$$\mathbf{z}^* = \arg \min_{\mathbf{z}} \tilde{\ell}_{\mathbf{y}}(\mathbf{z}) \quad (7.26)$$

and its derivative as

$$\nabla_{\mathbf{y}} \ell_{\mathbf{z}^*}^*(\mathbf{y}) = \mathbf{y} - \mathbf{z}^*. \quad (7.27)$$

With this construction, we obtain the Newton loss  $\ell_{\mathbf{z}^*}^*$ , a new convex loss, which has a gradient that corresponds to the Newton step of the original loss.

Note that (7.25) is an instance of (7.5) for a regularization term describing the quadratic approximation error, i.e.,  $\Omega(\mathbf{z}, f(\mathbf{x}; \theta)) = \tilde{\ell}_{f(\mathbf{x}; \theta)}(\mathbf{z}) - \ell(\mathbf{z})$ . As  $\Omega$  is already implicitly part of the Newton step, we do not need to evaluate it.

In general,  $\ell_{\mathbf{z}^*}^*$  exhibits more desirable behavior than  $\ell$ , as a single gradient descent step can solve any quadratic problem, and it possesses the same convergence properties as the Newton method when optimizing  $\mathbf{y}$ . In the case of non-convex  $\ell$ , as it is common for many algorithmic losses, the incorporation of second-order information also substantially improves empirical performance. Note that, in the case of non-convex or ill-conditioned settings, using Tikhonov regularization [267] stabilizes  $\ell_{\mathbf{z}^*}^*$ .

In the following, we define Newton losses and use  $x$  and  $z^*$  to denote samples / rows of  $\mathbf{x}$  and  $\mathbf{z}^*$ .

**Definition 7.1** (Element-wise Hessian-based Newton Losses) *For a loss function  $\ell$ , and a given current parameter vector  $\theta$ , we define the element-wise Hessian-based Newton loss as  $\ell_{z^*}^*(y) = \frac{1}{2} \|z_E^* - y\|_2^2$ , where*

$$z_E^* = \bar{y} - (\nabla_{\bar{y}}^2 \ell(\bar{y}))^{-1} \nabla_{\bar{y}} \ell(\bar{y}) \quad \text{and} \quad \bar{y} = f(x; \theta).$$

However, instead of using the element-wise Hessian-based Newton loss, it is typically more stable to use the empirical Hessian-based Newton loss.

**Definition 7.2** (Empirical Hessian-based Newton Losses) *For a loss function  $\ell$  and a given current parameter vector  $\theta$ , we define the empirical Hessian-based Newton loss as  $\ell_{z^*}^*(y) = \frac{1}{2} \|z_H^* - y\|_2^2$ , where*

$$z_H^* = \bar{y} - \left( \mathbb{E}_{\bar{y}} \left[ \nabla_{\bar{y}}^2 \ell(\bar{y}) \right] \right)^{-1} \nabla_{\bar{y}} \ell(\bar{y}) \quad \text{and} \quad \bar{y} = f(x; \theta).$$

Instead of using the Hessian, it is also possible to use the Fisher information matrix for second-order information. While this coincides with the Hessian in certain cases, it yields different results in most cases. The Fisher-based Newton loss can be seen as using natural gradient descent for optimizing the loss, while using regular gradient descent for optimizing the neural network.

**Definition 7.3** (Fisher-based Newton Losses) *For a loss function  $\ell$ , and a given current parameter vector  $\theta$ , we define the Fisher-based Newton loss as  $\ell_{z^*}^*(y) = \frac{1}{2} \|z_F^* - y\|_2^2$ , where*

$$z_F^* = \bar{y} - \left( \mathbb{E}_{\bar{y}} \left[ \nabla_{\bar{y}} \ell(\bar{y}) \nabla_{\bar{y}} \ell(\bar{y})^\top \right] \right)^{-1} \nabla_{\bar{y}} \ell(\bar{y}) \quad \text{and} \quad \bar{y} = f(x; \theta).$$

After presenting the formulations of different types of Newton losses, we would like to give some remarks on the computational cost and the derivative of Newton losses.

**Remark 7.1** (Computational Considerations) The Hessian of the loss function  $\nabla_y^2 \ell(y)$  may be approximated using the empirical Fisher matrix  $F = \mathbb{E} [\nabla_y \ell(y) \nabla_y \ell(y)^\top]$ . However, as only the Hessian of the loss function (and not the Hessian of the neural network) needs to be computed, computing the exact Hessian  $\nabla_y^2 \ell(y)$  is usually also fast.

**Remark 7.2** (Derivative of the Newton Loss) The derivative of the Newton loss is

$$\frac{\partial}{\partial y} \frac{1}{2} \|z^* - y\|_2^2 = y - z^*. \quad (7.28)$$

Note that the derivative of  $z^*$  wrt.  $y$  is zero because it is the projected optimum of the original loss.

### 7.3.1 Examples

We have seen in (7.25) how a given loss function  $\ell$  induces a corresponding Newton loss  $\ell^*$ . For specific loss functions, the Newton loss can be explicitly computed. We begin with the trivial example of the MSE loss. For notational simplicity we often drop the subscript  $z^*$  in the definition of the Newton loss (7.25).

**Example 7.1** (MSE Loss) Consider the classical MSE loss, i.e.,  $\ell(y) = \frac{1}{2} \|y - y^*\|_2^2$ , where  $y^*$  denotes the ground truth. Then,  $z^* = y^*$  and accordingly the Newton loss is given as

$$\ell_{z^*}^*(y) = \frac{1}{2} \|z^* - y\|_2^2 = \frac{1}{2} \|y^* - y\|_2^2 = \ell(y).$$

Therefore, the MSE loss  $\ell$  and its induced Newton loss  $\ell^*$  are equivalent.

A popular loss function for classification is the softmax cross-entropy (SMCE) loss, defined as

$$\ell_{\text{SMCE}}(y) = \sum_{i=1}^k -p_i \log q_i, \quad \text{where} \quad q_i = \frac{\exp(y_i)}{\sum_{j=1}^k \exp(y_j)}. \quad (7.29)$$

**Example 7.2** (Softmax Cross-Entropy Loss) For the SMCE loss, the induced Newton loss is given as

$$\ell_{\text{SMCE}}^*(y) = \frac{1}{2} \|z^* - y\|_2^2 \quad (7.30)$$

where the element-wise Hessian variant is

$$z_E^* = -(\text{diag}(q) - qq^\top)^{-1} (q - p) + y, \quad (7.31)$$

the empirical Hessian variant is

$$z_H^* = -\mathbb{E} [\text{diag}(q) - qq^\top]^{-1} (q - p) + y, \quad (7.32)$$

and the empirical Fisher variant is

$$z_F^* = -\mathbb{E} [(q - p)(q - p)^\top]^{-1} (q - p) + y. \quad (7.33)$$

In the experiments, we use the SMCE loss for a classification experiment.

A less trivial example is the binary cross-entropy (BCE) loss with

$$\ell_{\text{BCE}}(y) = \text{BCE}(y, p) = -\sum_{i=1}^m p_i \log y_i + (1 - p_i) \log(1 - y_i), \quad (7.34)$$

where  $p \in \Delta_m$  is a probability vector encoding of the ground truth.

**Example 7.3** (Binary Cross-Entropy Loss) For the BCE loss, the induced Newton loss is given as

$$\ell_{\text{BCE}}^*(y) = \frac{1}{2} \|z^* - y\|_2^2, \quad (7.35)$$

where the element-wise Hessian variant is

$$z_E^* = -\text{diag} \left( -p \oslash y^2 + (1-p) \oslash (1-y)^2 \right)^{-1} (p \oslash y - (1-p) \oslash (1-y)) + y, \quad (7.36)$$

the empirical Hessian variant is

$$z_H^* = -\text{diag} \left( \mathbb{E} \left[ -p \oslash y^2 + (1-p) \oslash (1-y)^2 \right] \right)^{-1} (p \oslash y - (1-p) \oslash (1-y)) + y, \quad (7.37)$$

the empirical Fisher variant is

$$z_F^* = -\mathbb{E} \left[ (p \oslash y - (1-p) \oslash (1-y)) (p \oslash y - (1-p) \oslash (1-y))^\top \right]^{-1} (p \oslash y - (1-p) \oslash (1-y)) + y, \quad (7.38)$$

and  $\odot$  and  $\oslash$  are element-wise operations.

The BCE loss is often extended using the logistic sigmoid function to what is called the sigmoid binary cross-entropy loss (SBCE), defined as

$$\ell_{\text{SBCE}}(y) = \text{BCE}(\sigma(y), p) \quad \text{where} \quad \sigma(x) = \frac{1}{1 + \exp(-x)}. \quad (7.39)$$

**Example 7.4** (Sigmoid Binary Cross-Entropy Loss) For the SBCE loss, the induced Newton loss is given as

$$\ell_{\text{SBCE}}^*(y) = \frac{1}{2} \|z^* - y\|_2^2 \quad (7.40)$$

where the element-wise Hessian variant is

$$z_E^* = -\text{diag} \left( \sigma(y) - \sigma(y)^2 \right)^{-1} (\sigma(y) - p) + y, \quad (7.41)$$

the empirical Hessian variant is

$$z_H^* = -\text{diag} \left( \mathbb{E} \left[ \sigma(y) - \sigma(y)^2 \right] \right)^{-1} (\sigma(y) - p) + y, \quad (7.42)$$

and the empirical Fisher variant is

$$z_F^* = -\mathbb{E} \left[ (\sigma(y) - p)(\sigma(y) - p)^\top \right]^{-1} (\sigma(y) - p) + y, \quad (7.43)$$

and where  $\odot$  and  $\oslash$  are element-wise operations.

## 7.4 Regularized Sampling Greedy Optimizer (RESGRO)

In this section, we introduce the REGularized Sampling GREedy Optimizer (RESGRO), an optimization algorithm for our two-stage optimization setting, which can be used for step (7.3a). Importantly, RESGRO does not require differentiability of  $\ell$ ; therefore, it provides an alternative to the differentiable relaxations presented in this thesis.

The key idea behind RESGRO is replacing the difficult optimization of  $\ell(z)$  simply by *sampling* a number of samples around the current value and taking the value leading to the best accuracy / objective function value. This procedure is done for each optimization step (7.3a), which makes the algorithm *greedy*. The core ingredient is limiting the number of samples and sampling only around the current value, which *regularizes* RESGRO. We note that RESGRO can be seen as a form of Hill Climbing. We can formally introduce RESGRO as follows: Let  $\mathcal{A} \subset \mathcal{Y}$  be a random set, where each element  $z \in \mathcal{A}$  is drawn from  $z \sim y + \epsilon$  with  $y \in \mathcal{Y}$ . The random set  $\mathcal{A}$  consists of  $K$  elements, i.e.,  $|\mathcal{A}| = K$ .  $\epsilon$  may be drawn from a Gaussian or a Cauchy distribution, among others. Then, we can find a regularizer  $\Omega$  such that

$$z^* = \arg \min_{z \in \mathcal{Y}} \ell(z) + \Omega(z, f(x; \theta)) = \mathbb{E}_{\mathcal{A}} \left[ \arg \min_{z \in \mathcal{A}} \ell(z) \right] \quad (7.44)$$

$$= f(\mathbf{x}; \theta) + \mathbb{E}_{\epsilon_1, \dots, \epsilon_K} \left[ \arg \min_{\epsilon_i \in \{\epsilon_1, \dots, \epsilon_K\}} \ell(f(\mathbf{x}; \theta) + \epsilon_i) \right]. \quad (7.45)$$

Identifying an explicit form of the regularizer  $\Omega$  is not necessary in this case, as we only require its implicit form, and  $\Omega$  does not need to be computable because the minimum can be found through the application of RESGRO. While the right-hand side of (7.44) is the expectation value of the optimal input among  $K$  samples, in many cases, we can simply approximate it via a single set of samples using

$$z^* \triangleq \arg \min_{z \in \mathcal{A}} \ell(z). \quad (7.46)$$

In other cases, however, we want to average over multiple sets of samples to obtain a better and less stochastic training response, for which we could simply compute an empirical estimate of (7.44). As, e.g., for shortest-path problems, the algorithm is quite costly, we want to avoid sampling large numbers of samples. Thus, we can use bootstrapping to more efficiently use the samples, i.e., *bootstrapped RESGRO*. Also, large numbers of samples reduce the implicit regularization  $\Omega$  induced by limited local sampling, an essential component to RESGRO. Thus, we propose using bootstrapping to effectively reuse samples to approximate  $z^*$  as

$$z^* \triangleq \mathbb{E}_{\mathcal{A} \subset \mathcal{B}} \left[ \arg \min_{z \in \mathcal{A}} \ell(z) \right], \quad (7.47)$$

where  $\mathcal{B}$  is a set of  $M > K$  uniform samples from  $y + \epsilon$  and  $\mathcal{A} \subset \mathcal{B}$  such that  $|\mathcal{A}| = K$ .

### 7.4.1 Discussion

The reason why this simple algorithm performs well when used in our proposed two-stage optimization scheme is that the space of outputs  $\mathcal{Y}$  is moderately low dimensional. In contrast, the space of neural network parameters  $\Theta$  is typically extremely high-dimensional, which would make the algorithm non-applicable. However, it is plausible to apply split backpropagation with *RESGRO* and without classical gradient-based backpropagation, by splitting at each layer, and dividing the neurons of each layer into smaller groups, which are handled independently.

We postulate that *RESGRO* may also be used for reinforcement learning as a replacement for stochastic smoothing / REINFORCE [40]. Further, *RESGRO* may be combined with REINFORCE, as the samples for which the score / loss function  $\ell$  has to be evaluated may be from the same distribution and, therefore, could be shared between *RESGRO* and REINFORCE, i.e., we can have two training objectives at the computational cost of one (assuming that cost is measured in function evaluations of  $\ell$ ).

## 7.5 Experiments

For the experiments, we start with a simple classification problem and then extend the evaluation to two applications of algorithmic supervision. The first algorithmic supervision task is sorting and ranking supervision, where only the relative order of a set of samples is known, while their absolute values remain unsupervised. The second algorithmic supervision task is shortest-path supervision, where only the shortest path is supervised, while the underlying cost matrix remains unsupervised. These algorithmic supervision tasks were introduced in Chapter 2.

### 7.5.1 Classification

In this section, we explore the utility of Newton losses for the simple case of MNIST classification with a softmax cross-entropy loss. Note that, as softmax is already a well-behaved objective, we cannot expect large improvements, and the purpose of the experiment is rather to demonstrate that no loss in performance is induced through the Newton losses.

To facilitate a fair and extensive comparison, we benchmark training on 5 models and with 2 optimizers: We use 5-layer fully connected ReLU networks with 100 (M1), 400 (M2) and 1 600 (M3) neurons per layer, as well as the convolutional LeNet-5 with sigmoid activations (M4) and LeNet-5 with ReLU activations (M5). Further, we use SGD and Adam as optimizers. To evaluate both early performance and full training performance, we test after 1 and 200 epochs. As computing the Hessian inverse is trivial for these settings, we also include the element-wise Hessian (e.w. H); however, as it performs (expectedly) poorly and as it would also be expensive in the following experiments, we do not include it for the algorithmic losses experiments in the following sections.

We run each experiment with 20 seeds, which allows us to perform significance tests (significance level 0.05). As displayed in Table 7.1, we find that the element-wise Hessian (e.w. H) performs similar to regular training.

**Table 7.1:** MNIST classification learning results. The models are a 5-layer fully connected ReLU networks with 100 (M1), 400 (M2) and 1 600 (M3) neurons per layer, as well as the convolutional LeNet-5 with sigmoid activations (M4) and LeNet-5 with ReLU activations (M5). The results are averaged over 20 seeds, and significance tests between regular training and the Newton methods are conducted. A better mean is indicated by a gray bold-face number, and a significantly better result is indicated by a black bold-face number.

| Ep. | Optim. Method / Model | SGD Optimizer |               |               |               |               | Adam Optimizer |               |               |               |               |
|-----|-----------------------|---------------|---------------|---------------|---------------|---------------|----------------|---------------|---------------|---------------|---------------|
|     |                       | M1            | M2            | M3            | M4            | M5            | M1             | M2            | M3            | M4            | M5            |
| 1   | Regular               | 93.06%        | 94.26%        | 94.77%        | 10.57%        | 96.25%        | 94.75%         | <b>96.36%</b> | 95.90%        | 90.60%        | 97.56%        |
| 1   | Newton L. (e.w. H)    | 92.95%        | 94.23%        | 94.74%        | 10.57%        | 96.14%        | 94.63%         | 96.30%        | <b>96.06%</b> | 90.32%        | <b>97.63%</b> |
| 1   | Newton L. (H)         | 93.06%        | <b>94.28%</b> | <b>94.77%</b> | 10.57%        | 96.23%        | <b>94.75%</b>  | 96.36%        | <b>95.91%</b> | <b>90.63%</b> | <b>97.63%</b> |
| 1   | Newton L. (F)         | <b>94.56%</b> | <b>95.47%</b> | <b>95.36%</b> | 10.64%        | <b>97.77%</b> | <b>94.86%</b>  | 96.28%        | <b>95.95%</b> | 90.57%        | 97.49%        |
| 200 | Regular               | 98.13%        | 98.40%        | 98.46%        | 99.06%        | 99.07%        | 98.12%         | 98.46%        | 98.62%        | 98.95%        | <b>99.23%</b> |
| 200 | Newton L. (e.w. H)    | 98.11%        | 98.39%        | 98.44%        | 99.02%        | <b>99.09%</b> | <b>98.16%</b>  | 98.44%        | 98.54%        | 98.95%        | 99.22%        |
| 200 | Newton L. (H)         | 98.11%        | <b>98.42%</b> | <b>98.46%</b> | 99.04%        | <b>99.11%</b> | 98.12%         | <b>98.50%</b> | <b>98.63%</b> | <b>98.97%</b> | 99.20%        |
| 200 | Newton L. (F)         | <b>98.22%</b> | <b>98.56%</b> | <b>98.68%</b> | <b>99.11%</b> | <b>99.23%</b> | 98.09%         | <b>98.53%</b> | <b>98.66%</b> | <b>98.98%</b> | 99.21%        |

Using the empirical Hessian (H) is indistinguishable from regular training; specifically, in 12 out of 20 cases it is better and significantly better in one 1 out of 20 cases, which is to be expected from equal methods (on average 1/20 tests will be significant at a significance level of 0.05). Finally, we find that the Fisher-based Newton losses (F), perform better than regular training. Specifically, with the SGD optimizer, in 9 out of 10 settings, it is significantly better and on the remaining setting, it has a higher mean. Using the Adam optimizer [34], both methods perform similarly.

### 7.5.2 Sorting and Ranking Supervision

After exploring the simple softmax cross-entropy classification loss in the previous section, in this section, we explore the more complex sorting and ranking supervision setting with an array of differentiable sorting-based losses. Here, we perform the four-digit MNIST sorting benchmark as in Chapter 3 with differentiable sorting networks, NeuralSort [58], and SoftSort [59]. Since, in this experiment, the objectives are harder to optimize, we can achieve substantial improvements over the baselines. We train the CNN using the Adam optimizer [34] at a learning rate of  $10^{-3}$  for 100 000 steps and with a batch size of 100.

We explore NeuralSort, SoftSort, and differentiable sorting networks (DSNs) with logistic and Cauchy distributions in Table 7.3. For NeuralSort and SoftSort, we find that using Newton losses, either based on the Hessian or based on the Fisher matrix, improves performance substantially. In this case, using the Hessian performs better than using the Fisher matrix. For DSNs, we find that for logistic DSNs, the improvements are substantial. In Chapter 3, we have shown that monotonic differentiable sorting networks, i.e., the Cauchy DSNs, provide an improved variant of differentiable sorting networks. Thus, in this case, for  $n = 5$  and Cauchy DSNs, there is no improvement over the default loss. However, for the somewhat harder setting of  $n = 10$ , we can observe an improvement of more than 1% using the Hessian-based Newton loss, even in the Cauchy DSN case.

In summary, we obtain strong improvements on losses that are difficult to optimize, while on well-behaving losses only small or no improvements can be achieved. This aligns with our goal of improving performance on losses that are hard to optimize.

**Table 7.3:** Differentiable sorting results. The metric is the percentage of rankings correctly identified (EM) (and individual element ranks correctly identified (EW)) averaged over 10 seeds.

|              | $n = 5$              |                      |                      |                      | $n = 10$             |                      |                      |                      |
|--------------|----------------------|----------------------|----------------------|----------------------|----------------------|----------------------|----------------------|----------------------|
|              | Cauchy DSN           | Logistic DSN         | NeuralSort           | SoftSort             | Cauchy DSN           | Logistic DSN         | NeuralSort           | SoftSort             |
| Regular      | 85.09 (93.31)        | 53.56 (77.04)        | 71.33 (87.10)        | 70.70 (86.75)        | 55.29 (87.06)        | 12.31 (58.81)        | 24.26 (74.47)        | 27.46 (76.02)        |
| NL (Hessian) | <b>85.11 (93.31)</b> | <b>75.02 (88.53)</b> | 83.31 (92.54)        | 83.87 (92.72)        | <b>56.49 (87.44)</b> | <b>42.14 (75.35)</b> | <b>48.76 (84.83)</b> | <b>55.07 (86.89)</b> |
| NL (Fisher)  | 84.95 (93.25)        | 63.11 (79.28)        | <b>83.93 (92.80)</b> | <b>84.03 (92.82)</b> | 56.12 (87.35)        | 25.72 (52.18)        | 39.23 (81.14)        | 54.00 (86.56)        |

## RESGRO

In addition to Newton losses, we also explore RESGRO on the ranking supervision task. Here, we apply RESGRO to maximizing the Kendall’s  $\tau$  coefficient between the ground truth and the predictions, which is a discontinuous metric. RESGRO can be evaluated extremely fast even for large numbers of samples in this setting. As shown in Table 7.2, RESGRO achieves competitive performance compared to differentiable sorting methods. In fact, the only method that performs better than RESGRO is monotonic differentiable sorting networks (i.e., Cauchy).

**Table 7.2:** Sorting results for RESGRO. The metric is the percentage of rankings correctly identified (EM) (and individual element ranks correctly identified (EW)) averaged over 10 seeds.

|        | $n = 5$       | $n = 10$      |
|--------|---------------|---------------|
| RESGRO | 74.34 (88.37) | 42.23 (82.56) |

### 7.5.3 Shortest-Path Supervision

In this section, we apply Newton losses to the shortest-path supervision task of the  $12 \times 12$  Warcraft shortest-path problem [1, 37, 66] as in Chapter 2. Here,  $12 \times 12$  Warcraft terrain maps are given as  $96 \times 96$  RGB images, and the supervision is the shortest path from the top left to the bottom right according to a hidden cost embedding. The goal is to predict  $12 \times 12$  cost embeddings of the terrain maps such that the shortest path according to the predicted embedding corresponds to the ground truth shortest path. For this task, we explore three approaches: the relaxed AlgoVision Bellman-Ford algorithm, stochastic smoothing, and perturbed optimizers.

For the relaxed AlgoVision Bellman-Ford algorithm, we explore two variants of the algorithm (an outer For loop and an outer While loop) and two losses ( $L_1$  and  $L_2^2$ ), i.e., a total of four settings. As computing the Hessian of the AlgoVision Bellman-Ford algorithm is too expensive with the PyTorch implementation, we restrict this case to the Fisher-based Newton loss. As displayed in Table 7.4, the Newton loss improves performance in three out of four settings, and the overall best performance is also provided by the Newton loss.

After discussing analytical relaxations, we continue with stochastic methods, the results of which are displayed in Table 7.5. For stochastic smoothing of the loss function, (i.e., stochastic smoothing applied to the algorithm and loss as one unit), we find that Newton losses improve the performance for 10 and 30 samples, while the regular training performs best if only 3 samples can be drawn. This makes sense as the estimation of Hessian or Fisher with stochastic smoothing is not good enough with too few samples, but as soon

| Algorithm Loop<br>Loss | For          |              | While        |              |
|------------------------|--------------|--------------|--------------|--------------|
|                        | $L_1$        | $L_2^2$      | $L_1$        | $L_2^2$      |
| Regular                | 94.19        | <b>95.90</b> | 94.30        | 95.77        |
| Fisher Newton          | <b>94.52</b> | 95.37        | <b>94.47</b> | <b>95.93</b> |

**Table 7.4:** Shortest-path benchmark results for different variants of the AlgoVision-relaxed Bellman-Ford algorithm. The displayed metric is the percentage of perfect matches averaged over 10 seeds.

**Table 7.5:** Shortest-path benchmark results for the stochastic smoothing of the loss (including the algorithm), stochastic smoothing of the algorithm (excluding the loss), and perturbed optimizers with the Fenchel-Young loss. The metric is the percentage of perfect matches averaged over 10 seeds.

| Loss<br># Samples | SS of loss   |              |              | SS of algorithm |              |              | PO w/ FY loss |              |              |
|-------------------|--------------|--------------|--------------|-----------------|--------------|--------------|---------------|--------------|--------------|
|                   | 3            | 10           | 30           | 3               | 10           | 30           | 3             | 10           | 30           |
| Regular           | <b>62.83</b> | 77.01        | 85.48        | <b>57.55</b>    | 78.70        | 87.26        | 80.64         | 80.39        | 80.71        |
| Hessian Newton    | 62.40        | <b>78.82</b> | <b>85.94</b> | —               | —            | —            | <b>83.09</b>  | <b>81.13</b> | <b>83.45</b> |
| Fisher Newton     | 58.80        | <b>78.74</b> | <b>86.10</b> | 53.82           | <b>79.24</b> | <b>87.41</b> | 80.70         | 80.37        | 80.45        |

as we have at least 10 samples, it is good enough to improve performance. For stochastic smoothing of the algorithm, (i.e., stochastic smoothing applied only to the algorithm, and the gradient of the loss afterward computed using backpropagation), we can observe a very similar behavior. While estimating the Hessian is intractable in this case, we can see improvements using the Fisher for  $\geq 10$  samples. For perturbed optimizers with a Fenchel-Young loss [37], we can confirm that the number of samples drawn barely affects performance. By extending the formulation to also computing the Hessian of the Fenchel-Young loss, we can compute the Newton loss, and find that we achieve improvements of more than 2% in this case.

Additionally, we make two interesting observations:

We find that perturbed optimizers are more sample efficient but do not improve with more samples. Thus, a rule of thumb is that if only a few samples can be afforded, perturbed optimizers are better, and if many samples can be computed, plain stochastic smoothing performs better.

An interesting comparison is also stochastic smoothing of the loss vs. stochastic smoothing of the algorithm: Here, we find that stochastic smoothing of the loss, which computes a gradient, is more sample efficient for few samples. On the other hand, stochastic smoothing of the algorithm, which computes a large Jacobian, requires more samples but performs better for  $\geq 10$  samples. This makes sense, as estimating a vector (in this case of length 144) requires fewer samples than estimating a matrix (in this case of size  $144 \times 144$ ).

**Table 7.6:** Bootstrapped RESGRO applied to shortest-path supervision. Displayed is the percentage of perfect matches on the test set averaged over 10 seeds.

|             | 3 sets | 10 sets | 30 sets | 100 sets |
|-------------|--------|---------|---------|----------|
| 6 samples   | 52.85  | 54.98   | 62.67   | 59.24    |
| 10 samples  | 68.47  | 72.21   | 74.70   | 75.83    |
| 20 samples  | 76.95  | 81.53   | 82.82   | 83.84    |
| 30 samples  | 80.16  | 83.83   | 84.60   | 86.01    |
| 60 samples  | 82.98  | 85.11   | 86.58   | 87.17    |
| 100 samples | 83.63  | 86.21   | 87.58   | 88.11    |

## RESGRO

**Table 7.7:** Vanilla one-set RESGRO applied to shortest-path supervision. Displayed is the percentage of perfect matches on the test set averaged over 10 seeds.

|             |       |
|-------------|-------|
| 3 samples   | 36.34 |
| 10 samples  | 72.02 |
| 30 samples  | 77.22 |
| 100 samples | 79.31 |

Using RESGRO for the shortest-path supervision task, we achieve competitive performance compared to stochastic smoothing and perturbed optimizers. In Table 7.7, we apply the vanilla one-set RESGRO variant with between 3 and 100 samples. In Table 7.6, we extend this to the multi-set bootstrapped RESGRO variant, and achieve performance of up to 88.11% with 100 samples and 86.01% with 30 samples. In comparison, two out of the three methods in Table 7.5 underperform the bootstrapped RESGRO for 30 samples. Overall, we find that bootstrapping substantially improves performance for RESGRO in the shortest-path setting.

As RESGRO is very competitive compared to stochastic smoothing, which is popular in reinforcement learning, we postulate that RESGRO may also be used for reinforcement learning. Further, we postulate that Newton losses may also be applied for reinforcement learning.

## Conclusion

In this chapter, we proposed splitting backpropagation, a novel class of alternative optimization methods, which allows combining different optimizers. We proposed two instances of splitting backpropagation: First, Newton losses, a method for combining second-order optimization of the loss function and first-order optimization of the model. Second, RESGRO, a simple stochastic method for backpropagating through non-differentiable functions. Both methods deliver strong empirical performance on algorithmic supervision tasks.



To conclude this thesis, in this chapter, we discuss the implications of learning with differentiable algorithms on the field of machine learning, and discuss interesting avenues of future research.

In this thesis, we combined machine learning and classical algorithms. First, we developed a general framework for differentiable algorithms. Then, we focused on specific classes of algorithms, allowing us to cover them in greater detail. Finally, we proposed alternative optimization methods for learning with algorithms. These concepts are novel and have a broad range of applications and implications for different fields, as discussed in Section 8.1. Further, the methods presented in this work provide many direct paths of future research, as discussed in Section 8.2.

The author's future research will continue in the direction of learning with differentiable algorithms and will build upon the methods presented in this work. In addition, the author will continue research in the areas of individual fairness, efficient and biologically plausible neural architectures, as well as optimization.

## 8.1 Implications and Future Perspectives

Learning with differentiable algorithms is a rapidly growing subject in machine learning. Differentiable algorithms are the foundation for integrating algorithmic concepts into neural architectures, whether to enable algorithmic supervision or an algorithm being part of a neural network model. Algorithmic supervision enables learning with limited information, reducing manual annotation requirements and reducing human bias in labeling. In the following, we discuss the direct and indirect implications of differentiable algorithms for a selection of machine learning communities.

**Weakly-Supervised Learning** As algorithmically-supervised learning is a setting of weakly-supervised learning, this work directly contributes to weakly-supervised learning. Apart from our direct contributions to weakly-supervised learning, in recent time, the number of weakly-supervised architectures that integrate differentiable algorithms is rapidly rising.

**Self-Supervised Learning** Very recently, the utility of differentiable sorting for self-supervised learning has been demonstrated for learning on audio [100]. We have applied differentiable sorting networks for self-supervised representation learning on images [20].

**Algorithm-Enhanced Models** In this work, we considered algorithm-enhanced models, e.g., in the setting of shortest-path supervision. Another interesting example of algorithm-enhanced models is the image segmentation model by Cho *et al.* [117], which they enhance by using differentiable splines.

**Neuro-Symbolic Learning** In the area of neuro-symbolic learning [103, 268, 269], the idea is to integrate symbolic knowledge and reasoning into neural architectures. Thus, differentiable algorithms and logic are natural candidates for advancing neuro-symbolic systems. We believe that our alternative optimization methods could also be interesting for the neuro-symbolic community.

**Edge Computing and Embedded Machine Learning** In the growing field of machine learning for edge computing and embedded systems [234–237], the focus lies on inference at low computational cost, e.g., on a mobile CPU, a microcontroller, or an IoT device. As differentiable logic gate networks provide great inference at a very low computational cost, their application in edge computing and embedded systems comes naturally. While they are currently limited to moderately small architectures, in many edge computing applications, large neural architectures are not feasible in the first place, which substantially reduces the impact of this current limitation. We believe that deep differentiable logic gate networks are a great opportunity for practitioners in edge computing and embedded systems.

**Reinforcement Learning** The domain of reinforcement learning has some intersections with learning with differentiable algorithms. Specifically, the idea of stochastic smoothing is well known in reinforcement learning under the name REINFORCE [40] among many other names. As we propose alternative optimization strategies for learning with blackbox functions such as Newton losses and RESGRO, we postulate that these methods could also be used for reinforcement learning tasks. Specifically, Newton losses could extend existing reinforcement learning objectives, and for stochastic smoothing, we have demonstrated the utility of Newton losses. RESGRO could also be used for reinforcement learning and could be promising through its simplicity as well as competitiveness to stochastic smoothing.

**Computer Vision and Computer Graphics** Recently, in computer vision, Neural Radiance Fields (NeRFs) have been popularized. They do not only deliver neural representations of 3D shapes, but also achieve stunning results on novel view generation tasks [270]. NeRFs and similar methods (like [271]) build on differentiable volume rendering and advances presented in this work could help improve these methods, e.g., interesting directions are using distributions besides the logistic distribution, or using aggregations via T-norms besides the probabilistic T-conorm. In computer graphics, in recent years, not only have differentiable renderers become very relevant [272], but also many other building blocks have been made differentiable, e.g., differentiable material graphs for procedural materials [210], or differentiable simulations [273].

## 8.2 Extensions and Future Work

Finally, after discussing implications of learning with differentiable algorithms, we would like to point out immediate extensions for and interesting directions of future work for the methodologies presented in this thesis.

**Differentiable Algorithms** For the proposed general method of differentiable algorithms, natural extensions are using alternative distributions and / or T-norms and T-conorms. Due to the generality of the method, it can be readily used to relax other algorithms or non-differentiable expressions. The framework already supports evaluating a neural network as a function as part of a differentiable algorithm; we have not empirically evaluated this case in this thesis, but it is a great direction for future exploration.

**Differentiable Sorting and Ranking** A natural application of differentiable sorting networks are classic learning-to-rank tasks in the domain of recommender systems [79, 80]. Apart from this, there are also many other applications of differentiable sorting and ranking, as can be seen from the literature published in recent years [85–87, 94, 100, 101].

**Differentiable Top- $k$**  Differentiable top- $k$  classification learning may be applied in any classification task. Herein, interesting settings are those with large label-noise and label-ambiguities or settings in which there is uncertainty about the ground truth label; however, the method also achieves improvements on vanilla classification settings.

**Differentiable Rendering** Our work on differentiable rendering could be extended by generalizing differentiable rendering also wrt. other aspects, e.g., wrt. shading. Further, the ideas presented in this work could be applied to other forms of differentiable rendering, such as implicit differentiable rendering.

**Differentiable Logic** For differentiable logic gate networks, there are many opportunities for future work: From an engineering perspective, there are opportunities for improving the computational efficiency of frameworks for training logic gate networks, as well as for improving inference speed. Another direction is combinatorial optimization and pruning of trained logic gate networks to reduce the number of computations required for inference. For practitioners, there is the opportunity of using differentiable logic gate networks in edge computing and embedded systems—both for replacing existing models by better and more efficient models, or enabling the use of machine learning models in extremely low-end embedded systems and microcontrollers in the first place. From a research perspective, methods that have shown success for training conventional neural networks may be adapted for training differentiable logic gate networks. Furthermore, developing more scalable architectures is an interesting research direction.

**Alternative Optimization Methods** The proposed two-stage optimization method can readily be extended to an  $n$ -stage optimization method by applying the split at multiple locations. Moreover, splitting backpropagation can also be directly applied to other optimization methods, such as convex optimization. We believe that splitting backpropagation can also improve performance in many other applications of differentiable algorithms, as we saw improvements over state-of-the-art methods in both of the empirically evaluated settings. But we also believe that the method has many applications beyond learning with differentiable algorithms.

## **SUPPLEMENTARY MATERIALS**



In this supplementary material, we define each of the presented distributions / sigmoid functions. These are relevant to Chapters 2, 3, 4, and 5, as they are either used in or comprise extension to the methods in the respective chapters. Figure A.1 displays the respective CDFs and PDFs.

An extensive discussion, including a taxonomy of the presented distributions, can be found in Chapter 5, Section 5.3.

Note that, for each distribution, the PDF  $f$  is defined as the derivative of the CDF  $F$ . Also, note that a reversed (Rev.) CDF is defined as  $F_{\text{Rev.}}(x) = 1 - F(-x)$ , which means that  $F_{\text{Rev.}} = F$  for symmetric distributions. The square-root distribution  $F_{\text{sq}}$  is defined in terms of  $F$  as in Equation (5.5). Therefore, in the following, we define the distributions via their CDFs  $F$ .

## Heaviside

$$x \mapsto \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{otherwise} \end{cases} \quad (\text{A.1})$$

## Uniform

$$x \mapsto \begin{cases} 0 & \text{if } x < -1 \\ 0.5 \cdot (1 + x) & \text{if } -1 \leq x \leq 1 \\ 1 & \text{otherwise} \end{cases} \quad (\text{A.2})$$

## Cubic Hermite

$$x \mapsto \begin{cases} 0 & \text{if } x < -1 \\ 3y^2 - 2y^3 & \text{if } -1 \leq x \leq 1 \\ 1 & \text{otherwise} \end{cases} \quad (\text{A.3})$$

where  $y := (x + 1)/2$ .

## Wigner Semicircle

$$x \mapsto \begin{cases} 0 & \text{if } x < -1 \\ \frac{1}{2} + \frac{x\sqrt{1-x^2}}{\pi} + \frac{\arcsin(x)}{\pi} & \text{if } -1 \leq x \leq 1 \\ 1 & \text{otherwise} \end{cases} \quad (\text{A.4})$$

## Gaussian

$$x \mapsto \frac{1}{2} \left( 1 + \operatorname{erf} \left( \frac{x}{\sqrt{2}} \right) \right) \quad (\text{A.5})$$

**Laplace**

$$x \mapsto \begin{cases} \frac{1}{2} \exp(x) & \text{if } x \leq 0 \\ 1 - \frac{1}{2} \exp(-x) & \text{if } x \geq 0 \end{cases} \quad (\text{A.6})$$

**Logistic**

$$x \mapsto \frac{1}{1 + \exp(-x)} \quad (\text{A.7})$$

**Hyperbolic secant / Gudermannian**

$$x \mapsto \frac{2}{\pi} \arctan \left( \exp \left( \frac{\pi}{2} x \right) \right) \quad (\text{A.8})$$

**Cauchy**

$$x \mapsto \frac{1}{\pi} \arctan(x) + \frac{1}{2} \quad (\text{A.9})$$

**Reciprocal**

$$x \mapsto \frac{x}{2 + 2|x|} + \frac{1}{2} \quad (\text{A.10})$$

**Gumbel-Max**

$$x \mapsto e^{-e^{-x}} \quad (\text{A.11})$$

**Gumbel-Min**

$$x \mapsto e^{-e^x} \quad (\text{A.12})$$

**Exponential**

$$x \mapsto 1 - e^{-x} \quad (\text{A.13})$$

**Levy**

$$x \mapsto 2 - 2\Phi \left( \sqrt{\frac{1}{x}} \right) \quad (\text{A.14})$$

where  $\Phi$  is the CDF of the standard normal distribution.

**Gamma**

$$x \mapsto \frac{1}{\Gamma(p)} \gamma(p, x) \quad (\text{A.15})$$

where  $\gamma(p, x)$  is the lower incomplete gamma function and  $p > 0$  is the shape parameter.

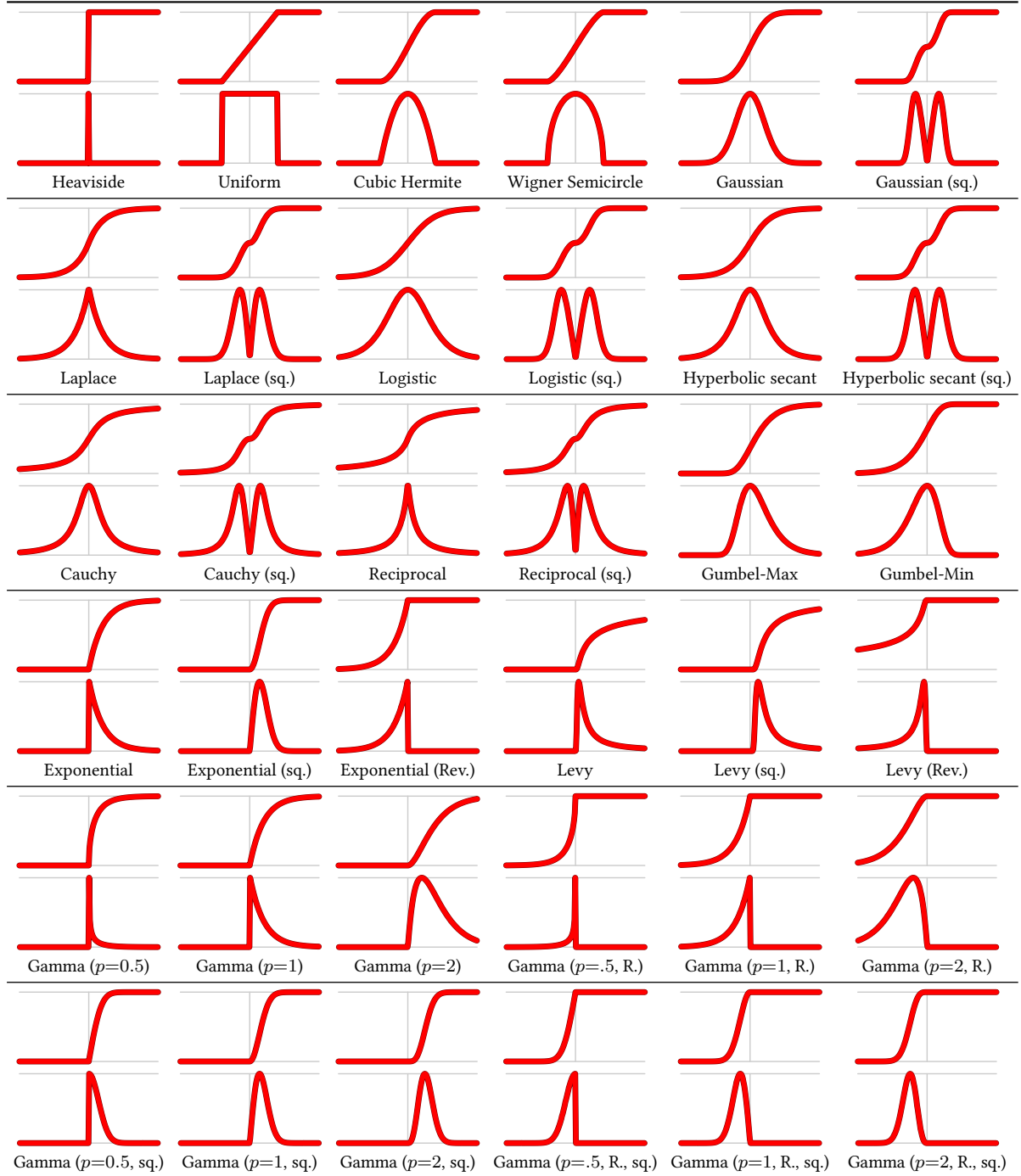


**Figure A.1:** Visualization of CDFs (*top*) and PDFs (*bottom*) for different distributions.



# T-Norms and T-Conorms

In this supplementary material, we cover various T-norms and T-conorms, which are integral to real-valued logic as used in the aggregation within differentiable rendering, in differentiable logic networks, as well as in general differentiable algorithms.

An extensive discussion, including a characterization of the presented families of T-norms and T-conorms, can be found in Chapter 5, Section 5.3.

The axiomatic approach to multi-valued logics is based on defining reasonable properties for truth functions. We stated the axioms for multi-valued generalizations of the disjunction (logical “or”), called T-conorms, in Definition 5.1. Here, we complement this with the axioms for multi-valued generalizations of the conjunction (logical “and”), called T-norms.

**Definition B.1 (T-norm)** A T-norm (triangular norm) is a binary operation  $\top : [0, 1] \times [0, 1] \rightarrow [0, 1]$ , which satisfies

- ▶ *associativity*:  $\top(a, \top(b, c)) = \top(\top(a, b), c)$ ,
- ▶ *commutativity*:  $\top(a, b) = \top(b, a)$ ,
- ▶ *monotonicity*:  $(a \leq c) \wedge (b \leq d) \Rightarrow \top(a, b) \leq \top(c, d)$ ,
- ▶ *1 is a neutral element*:  $\top(a, 1) = a$ .

These axioms ensure that the corners of the unit square, that is, the value pairs considered in classical logic, are processed as with a standard conjunction: neutral element and commutativity imply that  $(1, 1) \mapsto 1$ ,  $(0, 1) \mapsto 0$ ,  $(1, 0) \mapsto 0$ . From one of the latter two and monotonicity, it follows  $(0, 0) \mapsto 0$ . Analogously, the axioms of T-conorms ensure that the corners of the unit square are processed as with a standard disjunction. Actually, the axioms already fix the values not only at the corners but on the boundaries of the unit square. Only inside the unit square (that is, for  $(0, 1)^2$ ) T-norms (as well as T-conorms) can differ.

In the theory of multi-valued logics, and especially in fuzzy logic [223], it was established that the largest possible T-norm is the minimum and the smallest possible T-conorm is the maximum: for any T-norm  $\top$  it is  $\top(a, b) \leq \min(a, b)$  and for any T-conorm  $\perp$  it is  $\perp(a, b) \geq \max(a, b)$ . The

|                 |                                                                                                                                         |
|-----------------|-----------------------------------------------------------------------------------------------------------------------------------------|
| Minimum         | $\top^M(a, b) = \min(a, b)$                                                                                                             |
| Probabilistic   | $\top^P(a, b) = ab$                                                                                                                     |
| Einstein        | $\top^E(a, b) = \frac{ab}{2-a-b+ab}$                                                                                                    |
| Hamacher        | $\top_p^H(a, b) = \frac{ab}{p+(1-p)(a+b-ab)}$                                                                                           |
| Frank           | $\top_p^F(a, b) = \log_p \left( 1 + \frac{(p^a-1)(p^b-1)}{p-1} \right)$                                                                 |
| Yager           | $\top_p^Y(a, b) = \max \left( 0, 1 - ((1-a)^p + (1-b)^p)^{\frac{1}{p}} \right)$                                                         |
| Aczél-Alsina    | $\top_p^A(a, b) = \exp \left( -( \log(a) ^p +  \log(b) ^p)^{\frac{1}{p}} \right)$                                                       |
| Dombi           | $\top_p^D(a, b) = \left( 1 + \left( \left( \frac{1-a}{a} \right)^p + \left( \frac{1-b}{b} \right)^p \right)^{\frac{1}{p}} \right)^{-1}$ |
| Schweizer-Sklar | $\top_p^S(a, b) = (a^p + b^p - 1)^{\frac{1}{p}}$                                                                                        |

**Table B.1:** (Families of) T-norms.

**Table B.2:** (Families of) T-conorms.

|                 |                                                                                                                                           |
|-----------------|-------------------------------------------------------------------------------------------------------------------------------------------|
| Maximum         | $\perp^M(a, b) = \max(a, b)$                                                                                                              |
| Probabilistic   | $\perp^P(a, b) = a + b - ab$                                                                                                              |
| Einstein        | $\perp^E(a, b) = \perp_2^H(a, b) = \frac{a+b}{1+ab}$                                                                                      |
| Hamacher        | $\perp_p^H(a, b) = \frac{a+b+(p-2)ab}{1+(p-1)ab}$                                                                                         |
| Frank           | $\perp_p^F(a, b) = 1 - \log_p \left( 1 + \frac{(p^{1-a}-1)(p^{1-b}-1)}{p-1} \right)$                                                      |
| Yager           | $\perp_p^Y(a, b) = \min \left( 1, (a^p + b^p)^{\frac{1}{p}} \right)$                                                                      |
| Aczél-Alsina    | $\perp_p^A(a, b) = 1 - \exp \left( -( \log(1-a) ^p +  \log(1-b) ^p)^{\frac{1}{p}} \right)$                                                |
| Dombi           | $\perp_p^D(a, b) = \left( 1 + \left( \left( \frac{1-a}{a} \right)^p + \left( \frac{1-b}{b} \right)^p \right)^{-\frac{1}{p}} \right)^{-1}$ |
| Schweizer-Sklar | $\perp_p^S(a, b) = 1 - ((1-a)^p + (1-b)^p - 1)^{\frac{1}{p}}$                                                                             |

other extremes, i.e., the smallest possible T-norm and the largest possible T-conorm are the so-called drastic T-norm, defined as  $\top^\circ(a, b) = 0$  for  $(a, b) \in (0, 1)^2$ , and the drastic T-conorm, defined as  $\perp^\circ(a, b) = 1$  for  $(a, b) \in (0, 1)^2$ . Hence, it is  $\top(a, b) \geq \top^\circ(a, b)$  for any T-norm  $\top$  and  $\perp(a, b) \leq \perp^\circ(a, b)$  for any T-conorm  $\perp$ . We do not consider the drastic T-conorm because it clearly does not yield useful gradients.

As mentioned, it is common to combine a T-norm  $\top$ , a T-conorm  $\perp$  and a negation  $N$  (or complement, most commonly  $N(a) = 1 - a$ ) so that De-Morgan's laws hold. Such a triplet is often called a *dual triplet*. In Tables B.1 and B.2 we show the formulas for the families of T-norms and T-conorms, respectively, where matching lines together with the standard negation  $N(a) = 1 - a$  form dual triplets. Note that, for some families, we limited the range of values for the parameter  $p$  (see Table 5.2) compared to more general definitions [223].

Figures B.1 and B.2 illustrate the considered set of T-conorms.

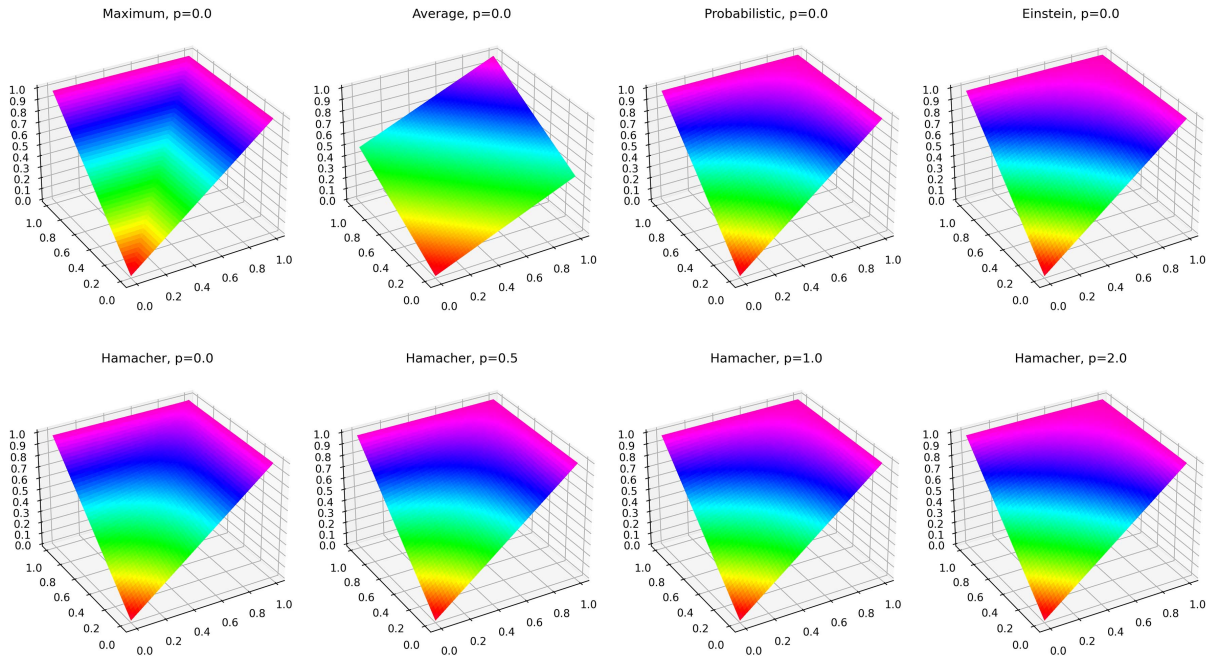


**Figure B.1:** T-conorm plots (1/2). Note that ‘Average’ is not a T-conorm and just included for reference. Also, Note how ‘Probabilistic’ is equal to ‘Hamacher  $p = 1$ ’ and ‘Einstein’ is equal to ‘Hamacher  $p = 2$ ’.

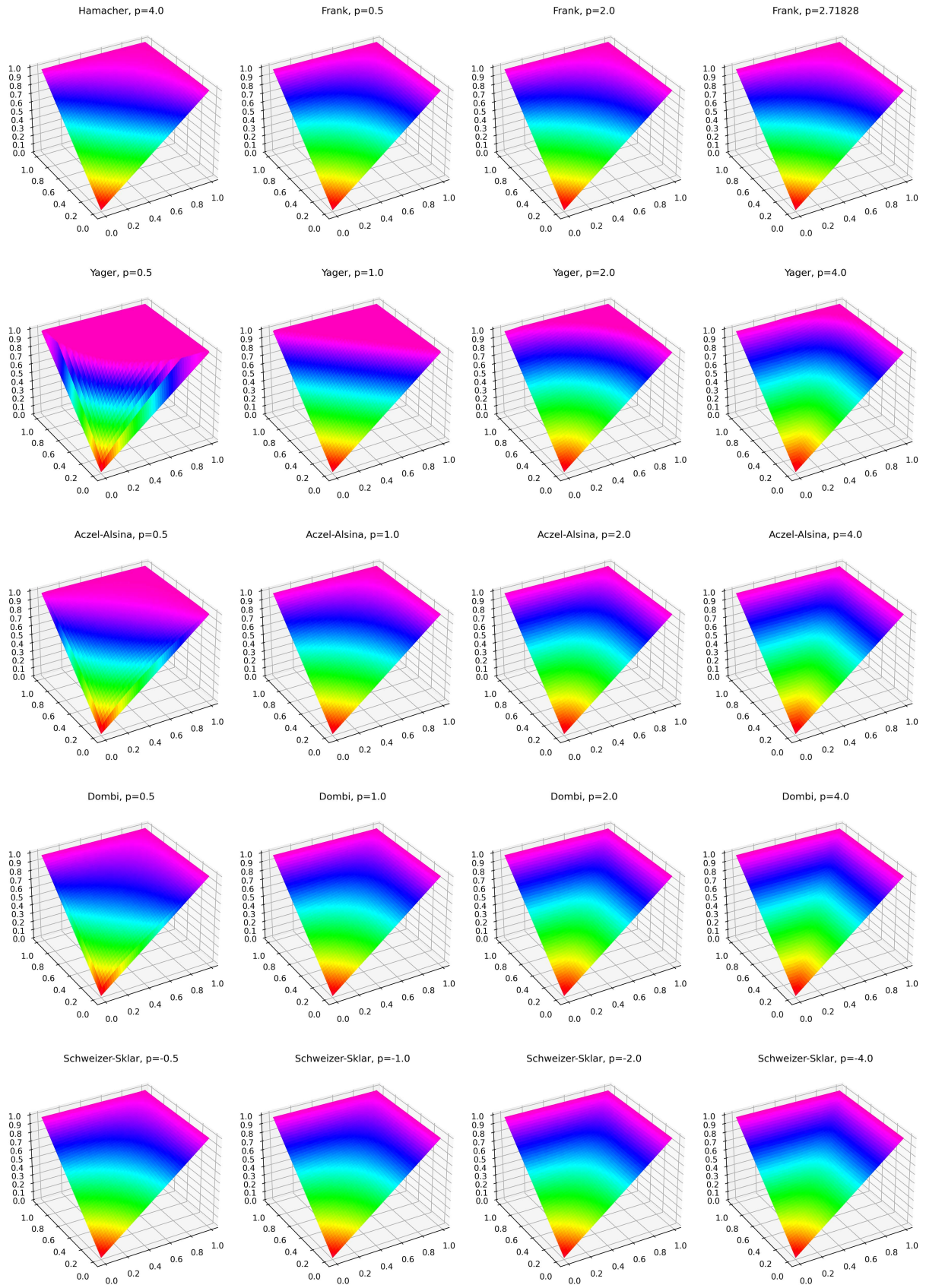


Figure B.2: T-conorm plots (2/2).



# Relaxed Minimum and Maximum

# C

A core element of differentiable algorithms is the relaxation of the min and max operators. For example, for differentiable sorting networks, the relaxations of min and max (i.e.,  $\min$  and  $\max$ ) are used to relax the conditional swap operation, allowing for a soft transition between passing through and swapping, such that the sorting operator becomes differentiable. It is natural to try to achieve this by using soft versions of the minimum (denoted by  $\min$ ) and maximum operators (denoted by  $\max$ ). However, before we consider concrete examples, let us collect some desirable properties that such relaxations should have. Naturally,  $\min$  and  $\max$  should satisfy many properties that their crisp / hard counterparts min and max satisfy, as well as a few others (for  $a, b, c \in \mathbb{R}$ ):

**Symmetry / Commutativity** Since min and max are symmetric/commutative, so should be their soft counterparts:  $\min(a, b) = \min(b, a)$  and  $\max(a, b) = \max(b, a)$ .

**Ordering** A (soft) maximum of two numbers should be at least as large as a (soft) minimum of the same two numbers:  $\min(a, b) \leq \max(b, a)$ .

**Continuity in Both Arguments** Both  $\min$  and  $\max$  should be continuous in both arguments.

**Idempotency** If the two arguments are equal in value, this value should be the result of  $\min$  and  $\max$ , that is,  $\min(a, a) = \max(a, a) = a$ .

**Inversion** As for min and max, the two operators  $\min$  and  $\max$  should be connected in such a way that the result of one operator equals the negated result of the other operator applied to negated arguments:  $\min(a, b) = -\max(-a, -b)$  and  $\max(a, b) = -\min(-a, -b)$ .

**Stability / Shift Invariance** Shifting both arguments by some value  $c \in \mathbb{R}$  should shift each operator's result by the same value:  $\min(a + c, b + c) = \min(a, b) + c$  and  $\max(a + c, b + c) = \max(a, b) + c$ . Stability implies that the values of  $\min$  and  $\max$  depend effectively only on the difference of their arguments. Specifically, choosing  $c = -a$  yields  $\min(a, b) = \min(0, b - a) + a$  and  $\max(a, b) = \max(0, b - a) + a$ , and  $c = -b$  yields  $\min(a, b) = \min(a - b, 0) + b$  and  $\max(a, b) = \max(a - b, 0) + b$ .

**Sum preservation** The sum of  $\min$  and  $\max$  should equal the sum of min and max:  $\min(a, b) + \max(a, b) = \min(a, b) + \max(a, b) = a + b$ . Note that sum preservation follows from stability, inversion and symmetry:  $\min(a, b) = \min(a - b, 0) + b = b - \max(0, b - a) = b - (\max(a, b) - a) = a + b - \max(a, b)$

**Bounded by Hard Versions** Soft operators should not yield values more extreme than their crisp / hard counterparts:  $\min(a, b) \leq \min(a, b)$  and  $\max(a, b) \leq \max(a, b)$ . Note that together with ordering this property implies idempotency, viz.:  $a = \min(a, a) \leq \min(a, a) \leq \max(a, a) \leq$

$\max(a, a) = a$ . Otherwise, they cannot be defined via a convex combination of their inputs, making it impossible to define proper *argmin* and *argmax*, and hence we could not compute, e.g., differentiable permutation matrices.

**Monotonicity in Both Arguments** For any  $c > 0$ , it should be  $\min(a + c, b) \geq \min(a, b)$ ,  $\min(a, b + c) \geq \min(a, b)$ ,  $\max(a + c, b) \geq \max(a, b)$ , and  $\max(a, b + c) \geq \max(a, b)$ . Note that the second expression for each operator follows from the first with the help of symmetry / commutativity.

**Bounded Error / Minimum Deviation from Hard Versions** Soft versions of minimum and maximum should differ as little as possible from their crisp / hard counterparts. However, this condition needs to be made more precise to yield concrete properties (see below for details).

Note that *min* and *max* cannot satisfy associativity, as this would force them to be identical to their hard counterparts. Associativity means that  $\max(a, \max(b, c)) = \max(\max(a, b), c)$  and that  $\min(a, \min(b, c)) = \min(\min(a, b), c)$ . Now, consider  $a, b \in \mathbb{R}$  with  $a < b$ . With associativity and idempotency  $\max(a, \max(a, b)) = \max(\max(a, a), b) = \max(a, b)$  and hence  $\max(a, b) = b = \max(a, b)$  (by comparison of the second arguments). Analogously, one can show that if associativity held, we would have  $\min(a, b) = a = \min(a, b)$ . That is, one cannot have both associativity and idempotency. Note that, without idempotency, the soft operators would not be bounded by their hard versions. As idempotency is necessary, associativity has to be given up.

If *min* and *max* are to be bounded by the crisp / hard version and symmetry, ordering, inversion and stability (which imply sum preservation) hold, they must be convex combinations of the arguments  $a$  and  $b$  with weights that depend only on the difference of  $a$  and  $b$ . That is,

$$\begin{aligned} \min(a, b) &= f(b - a) \cdot a + (1 - f(b - a)) \cdot b \\ \max(a, b) &= (1 - f(b - a)) \cdot a + f(b - a) \cdot b, \end{aligned}$$

where  $f(x)$  yields a value in  $[0, 1]$  (due to the boundedness of *min* and *max* by their crisp / hard counterparts). Due to inversion,  $f$  must satisfy  $f(x) = 1 - f(-x)$  and hence  $f(0) = \frac{1}{2}$ . Monotonicity of *min* and *max* requires that  $f$  is a monotonically increasing function. Continuity requires that  $f$  is a continuous function. In summary,  $f$  must be a continuous sigmoid function (in the older meaning of this term, i.e., an s-shaped function, of which the logistic function is only a special case) satisfying  $f(x) = 1 - f(-x)$ . Such a sigmoid function is the CDF of a corresponding probability distribution, and, therefore, CDFs of probability distributions are candidates for  $f$ . We note that we discuss a selection of distributions in Supplementary Material A.

As mentioned, the condition that the soft versions of minimum and maximum should deviate as little as possible from the crisp / hard versions causes a slight problem: this deviation can always be made smaller by making the sigmoid function steeper (reaching the crisp / hard versions in the limit for infinite inverse temperature, when the sigmoid function turns into the Heaviside step function). Hence, in order to find the best shape of the sigmoid function, we have to limit its inverse temperature. Therefore, for the respective analysis, we require w.l.o.g. the sigmoid function to be Lipschitz-continuous with Lipschitz constant  $\alpha = 1$ .

## C.1 Monotonicity of the Reciprocal Relaxed Min and Max

**Theorem C.1**  $\min_{f_{\mathcal{R}}}$  and  $\max_{f_{\mathcal{R}}}$  are monotonic functions with the sigmoid function  $f_{\mathcal{R}}$ .

*Proof.* W.l.o.g., we assume  $a_i = x$  and  $a_j = 0$ .

$$\min_{f_{\mathcal{R}}}(x, 0) = x \cdot f_{\mathcal{R}}(-x) = x \frac{1}{2} \left( \frac{x}{1 + |x|} + 1 \right) \quad (\text{C.1})$$

To show monotonicity, we consider its derivative / slope.

$$\frac{d}{dx} \min_{f_{\mathcal{R}}}(x, 0) = \frac{d}{dx} \left( x \frac{1}{2} \left( \frac{x}{1 + |x|} + 1 \right) \right) \quad (\text{C.2})$$

$$= \frac{1}{2} \left( \frac{x}{1 + |x|} + 1 \right) + x \frac{1}{2} \frac{d}{dx} \left( \frac{x}{1 + |x|} + 1 \right) \quad (\text{C.3})$$

$$= \frac{1}{2} \left( \frac{x}{1 + |x|} + 1 \right) + x \frac{1}{2} \frac{d}{dx} \left( \frac{x}{1 + |x|} \right) \quad (\text{C.4})$$

$$= \frac{1}{2} \left( \frac{x}{1 + |x|} + 1 \right) + x \frac{1}{2} \frac{\frac{dx}{dx} \cdot (1 + |x|) - x \cdot \frac{d|x|+1}{dx}}{(1 + |x|)^2} \quad (\text{C.5})$$

$$= \frac{1}{2} \left( \frac{x}{1 + |x|} + 1 \right) + x \frac{1}{2} \frac{(1 + |x|) - x \operatorname{sgn}(x)}{(1 + |x|)^2} \quad (\text{C.6})$$

$$= \frac{1}{2} \left( \frac{x}{1 + |x|} + 1 \right) + x \frac{1}{2} \frac{1 + |x| - |x|}{(1 + |x|)^2} \quad (\text{C.7})$$

$$= \frac{1}{2} \left( \frac{x}{1 + |x|} + 1 \right) + x \frac{1}{2} \frac{1}{1 + 2|x| + |x|^2} \quad (\text{C.8})$$

$$= \frac{1}{2} \left( \frac{x}{1 + |x|} + 1 + \frac{x}{1 + 2|x| + |x|^2} \right) \quad (\text{C.9})$$

$$= \frac{1}{2} \left( \frac{x(1 + |x|)}{1 + 2|x| + |x|^2} + \frac{1 + 2|x| + |x|^2}{1 + 2|x| + |x|^2} + \frac{x}{1 + 2|x| + |x|^2} \right) \quad (\text{C.10})$$

$$= \frac{1}{2} \left( \frac{2x + 2|x| + x|x| + |x|^2 + 1}{1 + 2|x| + |x|^2} \right) \quad (\text{C.11})$$

$$= \frac{1}{2} \left( \frac{2(x + |x|) + |x|(x + |x|) + 1}{1 + 2|x| + |x|^2} \right) \quad (\text{C.12})$$

$$\geq \frac{1}{2} \left( \frac{1}{1 + 2|x| + |x|^2} \right) \quad (\text{because } x + |x| \geq 0) \quad (\text{C.13})$$

$$> 0 \quad (\text{C.14})$$

$\max_{f_{\mathcal{R}}}$  is analogous. □

## C.2 Monotonicity of the Cauchy Relaxed Min and Max

**Theorem C.2**  $\min_{f_C}$  and  $\max_{f_C}$  are monotonic functions with the sigmoid function  $f_C$ .

*Proof.* W.l.o.g., we assume  $a_i = x$  and  $a_j = 0$ .

$$\min_{f_C}(x, 0) = x \cdot f_C(-x) = x \cdot \left( \frac{1}{\pi} \arctan(-\beta x) + \frac{1}{2} \right) \quad (\text{C.15})$$

To show monotonicity, we consider its derivative.

$$\begin{aligned} \frac{\partial}{\partial x} \min_{f_C}(0, x) &= \frac{\partial}{\partial x} (f_C(-x) \cdot x) = x \cdot \frac{\partial}{\partial x} f_C(-x) + f_C(-x) \cdot \frac{\partial}{\partial x} x \\ &= x \cdot \frac{\partial}{\partial x} \left( \frac{1}{\pi} \arctan(-\beta x) + \frac{1}{2} \right) + \frac{1}{\pi} \arctan(-\beta x) + \frac{1}{2} \\ &= x \cdot \frac{1}{\pi} \frac{-\beta}{1 + (\beta x)^2} - \frac{1}{\pi} \arctan(\beta x) + \frac{1}{2} \\ &= \frac{1}{2} - \frac{1}{\pi} \arctan(\beta x) - \frac{1}{\pi} \frac{\beta x}{1 + (\beta x)^2} \\ &= \frac{1}{2} - \frac{1}{\pi} \arctan(z) - \frac{1}{\pi} \frac{z}{1 + z^2} \quad (\text{with } z = \beta x) \end{aligned} \quad (\text{C.16})$$

To reason about the derivative, we also consider the second derivative:

$$\begin{aligned} \lim_{z \rightarrow \infty} \frac{1}{2} - \frac{1}{\pi} \arctan(z) - \frac{1}{\pi} \frac{z}{1 + z^2} &= \frac{1}{2} - \lim_{z \rightarrow \infty} \frac{1}{\pi} \arctan(z) - \lim_{z \rightarrow \infty} \frac{1}{\pi} \frac{z}{1 + z^2} \\ &= \frac{1}{2} - \frac{1}{\pi} \frac{\pi}{2} - 0 = 0 \end{aligned} \quad (\text{C.17})$$

For  $z \in [0, \infty)$ : The derivative of  $\min_{f_C}(0, x)$  converges to 0 for  $z \rightarrow \infty$ .

$$\begin{aligned} \lim_{z \rightarrow -\infty} \frac{1}{2} - \frac{1}{\pi} \arctan(z) - \frac{1}{\pi} \frac{z}{1 + z^2} &= \frac{1}{2} - \lim_{z \rightarrow -\infty} \frac{1}{\pi} \arctan(z) - \lim_{z \rightarrow -\infty} \frac{1}{\pi} \frac{z}{1 + z^2} \\ &= \frac{1}{2} - \frac{1 - \pi}{\pi} - 0 = \frac{1}{2} + \frac{1}{\pi} \frac{\pi}{2} - 0 = 1 \end{aligned} \quad (\text{C.18})$$

For  $z \in (-\infty, 0]$ : The derivative of  $\min_{f_C}(0, x)$  converges to 1 for  $z \rightarrow -\infty$ .

$$\frac{\partial}{\partial z} \frac{1}{2} - \frac{1}{\pi} \arctan(z) - \frac{1}{\pi} \frac{z}{1 + z^2} = -\frac{2}{\pi(1 + z^2)^2} < 0 \quad (\text{C.19})$$

The second derivative of  $\min_{f_C}(0, x)$  is always negative.

Therefore, the derivative is always in  $(0, 1)$ , and therefore always positive. Thus,  $\min_{f_C}(0, x)$  is strictly monotonic.  $\max_{f_C}$  is analogous.  $\square$

# Bibliography

- [1] Felix Petersen, Christian Borgelt, Hilde Kuehne, and Oliver Deussen. “Learning with Algorithmic Supervision via Continuous Relaxations”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2021.
- [2] Felix Petersen, Christian Borgelt, Hilde Kuehne, and Oliver Deussen. “Differentiable Sorting Networks for Scalable Sorting and Ranking Supervision”. In: *Proc. Machine Learning Research (PMLR), International Conference on Machine Learning (ICML)*. 2021.
- [3] Felix Petersen, Christian Borgelt, Hilde Kuehne, and Oliver Deussen. “Monotonic Differentiable Sorting Networks”. In: *International Conference on Learning Representations (ICLR)*. 2022.
- [4] Felix Petersen, Christian Borgelt, Hilde Kuehne, and Oliver Deussen. “Differentiable Top-k Classification Learning”. In: *Proc. Machine Learning Research (PMLR), International Conference on Machine Learning (ICML)*. 2022.
- [5] Felix Petersen, Bastian Goldluecke, Christian Borgelt, and Oliver Deussen. “GenDR: A Generalized Differentiable Renderer”. In: *Proc. International Conference on Computer Vision and Pattern Recognition (CVPR)*. 2022.
- [6] Felix Petersen, Christian Borgelt, Hilde Kuehne, and Oliver Deussen. “Deep Differentiable Logic Gate Networks”. In: *under review*. 2022.
- [7] Felix Petersen, Tobias Sutter, Christian Borgelt, Hilde Kuehne, and Oliver Deussen. “Newton Losses: Efficiently Including Second-Order Information into Gradient Descent”. In: *under review*. 2022.
- [8] Felix Petersen, Christian Borgelt, Mikhail Yurochkin, Hilde Kuehne, and Oliver Deussen. “Propagating Distributions through Neural Networks”. In: *under review*. 2022.
- [9] Felix Petersen, Debarghya Mukherjee, Christian Borgelt, Hilde Kuehne, and Oliver Deussen. “APE-VAE: Training Variational Auto-Encoders with Approximate Evidence”. In: *under review*. 2022.
- [10] Felix Petersen, Bastian Goldluecke, Oliver Deussen, and Hilde Kuehne. “Style Agnostic 3D Reconstruction via Adversarial Style Transfer”. In: *IEEE Winter Conference on Applications of Computer Vision (WACV)*. 2022.
- [11] Felix Petersen, Amit H Bermano, Oliver Deussen, and Daniel Cohen-Or. “Pix2Vex: Image-to-Geometry Reconstruction using a Smooth Differentiable Renderer”. In: *Computing Research Repository (CoRR) in arXiv* (2019).
- [12] Felix Petersen, Christian Borgelt, and Oliver Deussen. “AlgoNet:  $C^\infty$  Smooth Algorithmic Neural Networks”. In: *Computing Research Repository (CoRR) in arXiv*. 2019.
- [13] Felix Petersen, Christian Borgelt, and Oliver Deussen. “ $C^\infty$  Smooth Algorithmic Neural Networks for Solving Inverse Problems”. In: *NeurIPS Deep Inverse Workshop*. 2019.

- [14] Felix Petersen, Tobias Sutter, Christian Borgelt, Huh Dongsung, Hilde Kuehne, Yuekai Sun, and Oliver Deussen. “ISAAC Newton: Input-based Approximate Curvature for Newton’s Method”. In: *under review*. 2022.
- [15] Felix Petersen and Tobias Sutter. “Distributional Quantization”. In: *under review*. 2022.
- [16] Felix Petersen, Debarghya Mukherjee, Yuekai Sun, and Mikhail Yurochkin. “Post-processing for Individual Fairness”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2021.
- [17] Debarghya Mukherjee, Felix Petersen, Mikhail Yurochkin, and Yuekai Sun. “Domain Adaptation meets Individual Fairness. And they get along.” In: *Computing Research Repository (CoRR) in arXiv*. 2022.
- [18] Felix Petersen, Moritz Schubotz, and Bela Gipp. “Towards Formula Translation using Recursive Neural Networks”. In: *Proceedings of the 11th Conference on Intelligent Computer Mathematics (CICM), Work-in-Progress Paper Track*. 2018.
- [19] Felix Petersen, Moritz Schubotz, André Greiner-Petter, and Bela Gipp. “Neural Machine Translation for Mathematical Formulae”. In: *under review*. 2022.
- [20] Nina Shvetsova, Felix Petersen, Rogerio Feris, and Hilde Kuehne. “Differentiable K-Nearest Neighbor Sorting for Self-supervised Learning”. In: *under review*. 2022.
- [21] O. Neugebauer. *The Exact Sciences in Antiquity*. Acta historica scientiarum naturalium et medicinalium. Dover Publications, 1969.
- [22] Luigi F Menabrea and Ada A Lovelace. “Sketch of the Analytical Engine, invented by Charles Babbage, Esq., by LF Menabrea, of Turin, officer of the Military Engineers”. In: *Translated and with notes by AA, L. Taylor’s Scientific Memoirs* 3 (1843), pp. 666–731.
- [23] John Fuegi and Jo Francis. “Lovelace & Babbage and the creation of the 1843 ‘notes’”. In: *IEEE Annals of the History of Computing* 25.4 (2003), pp. 16–26.
- [24] Warren S McCulloch and Walter Pitts. “A logical calculus of the ideas immanent in nervous activity”. In: *The Bulletin of Mathematical Biophysics* 5.4 (1943), pp. 115–133.
- [25] David B Kirk and W Hwu Wen-Mei. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2016.
- [26] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2019.

- [27] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. “TensorFlow: A system for Large-scale Machine Learning”. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016.
- [28] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. “Caffe: Convolutional Architecture for Fast Feature Embedding”. In: *Proceedings of the 22nd ACM International Conference on Multimedia*. 2014.
- [29] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. *JAX: composable transformations of Python+NumPy programs*. 2018.
- [30] Kuniyiko Fukushima. “Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position”. In: *Biological Cybernetics* 36 (1980), pp. 193–202.
- [31] Yann LeCun, Patrick Haffner, Léon Bottou, and Yoshua Bengio. “Object Recognition with Gradient-Based Learning”. In: *Shape, contour and grouping in computer vision*. Springer, 1999, pp. 319–345.
- [32] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2012.
- [33] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. “Going Deeper with Convolutions”. In: *Proc. International Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015.
- [34] Diederik Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *International Conference on Learning Representations (ICLR)*. 2015.
- [35] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. “Learning representations by back-propagating errors”. In: *Nature* 323.6088 (1986), pp. 533–536.
- [36] Jacob Abernethy, Chansoo Lee, and Ambuj Tewari. “Perturbation techniques in online learning and optimization”. In: *Perturbations, Optimization, and Statistics* (2016).
- [37] Quentin Berthet, Mathieu Blondel, Olivier Teboul, Marco Cuturi, Jean-Philippe Vert, and Francis Bach. “Learning with Differentiable Perturbed Optimizers”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2020.
- [38] Peter W Glynn. “Likelihood Ratio Gradient Estimation for Stochastic Systems”. In: *Communications of the ACM* 33.10 (1990), pp. 75–84.
- [39] Jack PC Kleijnen and Reuven Y Rubinstein. “Optimization and Sensitivity Analysis of Computer Simulation Models by the Score Function Method”. In: *European Journal of Operational Research* 88.3 (1996), pp. 413–427.

- [40] Ronald J Williams. “Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning”. In: *Machine learning* 8.3 (1992), pp. 229–256.
- [41] Brendan J. Frey and Geoffrey E. Hinton. “Variational Learning in Nonlinear Gaussian Belief Networks”. In: *Neural Computation* 11 (1 1999), pp. 193–213.
- [42] Hao Wang, Xingjian Shi, and Dit-Yan Yeung. “Natural-Parameter Networks: A Class of Probabilistic Neural Networks”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2016.
- [43] Jochen Gast and Stefan Roth. “Lightweight Probabilistic Deep Networks”. In: *Proc. International Conference on Computer Vision and Pattern Recognition (CVPR)*. 2018.
- [44] Alexander Shekhovtsov and Boris Flach. “Feed-Forward Propagation in Probabilistic Neural Networks with Categorical and Max Layers”. In: *International Conference on Learning Representations (ICLR)*. 2019.
- [45] Anqi Wu, Sebastian Nowozin, Edward Meeds, Richard E Turner, Jose Miguel Hernandez-Lobato, and Alexander L Gaunt. “Deterministic variational inference for robust bayesian neural networks”. In: *International Conference on Learning Representations (ICLR)*. 2019.
- [46] Shichen Liu, Tianye Li, Weikai Chen, and Hao Li. “Soft Rasterizer: A Differentiable Renderer for Image-based 3D Reasoning”. In: *Proc. International Conference on Computer Vision (ICCV)*. 2019.
- [47] Wenzheng Chen, Jun Gao, Huan Ling, Edward J. Smith, Jaakko Lehtinen, Alec Jacobson, and Sanja Fidler. “Learning to Predict 3D Objects with an Interpolation-based Differentiable Renderer”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2019.
- [48] Swarat Chaudhuri and Armando Solar-Lezama. “Smooth Interpretation”. In: *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2010.
- [49] Quentin Le Lidec, Ivan Laptev, Cordelia Schmid, and Justin Carpentier. “Differentiable Rendering with Perturbed Optimizers”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2021.
- [50] Merlin Nimier-David, Delio Vicini, Tizian Zeltner, and Wenzel Jakob. “Mitsuba 2: A retargetable forward and inverse renderer”. In: *ACM Transactions on Graphics (TOG)* 38.6 (2019), pp. 1–17.
- [51] Jean-Baptiste Cordonnier, Aravindh Mahendran, Alexey Dosovitskiy, Dirk Weissenborn, Jakob Uszkoreit, and Thomas Unterthiner. “Differentiable Patch Selection for Image Recognition”. In: *Proc. International Conference on Computer Vision and Pattern Recognition (CVPR)*. 2021.
- [52] Hiroharu Kato, Deniz Beker, Mihai Morariu, Takahiro Ando, Toru Matsuoka, Wadim Kehl, and Adrien Gaidon. “Differentiable Rendering: A Survey”. In: *Computing Research Repository (CoRR) in arXiv* (2020).
- [53] Yuxuan Zhang, Wenzheng Chen, Huan Ling, Jun Gao, Yinan Zhang, Antonio Torralba, and Sanja Fidler. “Image GANs meet Differentiable Rendering for Inverse Graphics and Interpretable 3D Neural Rendering”. In: *International Conference on Learning Representations (ICLR)*. 2021.

- [54] Tzu-Mao Li, Miika Aittala, Fredo Durand, and Jaakko Lehtinen. “Differentiable Monte Carlo Ray Tracing Through Edge Sampling”. In: *ACM Transactions on Graphics (Proc. SIGGRAPH Asia)* (2018).
- [55] Cheng Zhang, Zhao Dong, Michael Doggett, and Shuang Zhao. “Antithetic sampling for Monte Carlo differentiable rendering”. In: *ACM Transactions on Graphics (TOG)* 40.4 (2021), pp. 1–12.
- [56] Marco Cuturi, Olivier Teboul, and Jean-Philippe Vert. “Differentiable Ranking and Sorting using Optimal Transport”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2019.
- [57] Marco Cuturi. “Sinkhorn Distances: Lightspeed Computation of Optimal Transport”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2013.
- [58] Aditya Grover, Eric Wang, Aaron Zweig, and Stefano Ermon. “Stochastic Optimization of Sorting Networks via Continuous Relaxations”. In: *International Conference on Learning Representations (ICLR)*. 2019.
- [59] Sebastian Prillo and Julian Eisenschlos. “SoftSort: A continuous relaxation for the argsort operator”. In: *Proc. Machine Learning Research (PMLR), International Conference on Machine Learning (ICML)*. 2020.
- [60] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. “Attention is All you Need”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2017.
- [61] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. “You Only Look Once: Unified, Real-Time Object Detection”. In: *Proc. International Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016.
- [62] Andrew W. Senior, Richard Evans, John Jumper, James Kirkpatrick, Laurent Sifre, Tim Green, Chongli Qin, Augustin Židek, Alexander W. R. Nelson, Alex Bridgland, Hugo Penedones, Stig Petersen, Steve Simonyan, Kareem Crossan, Pushmeet Kohli, David T. Jones, David Silver, Koray Kavukcuoglu, and Demis Hassabis. “Improved Protein Structure Prediction using Potentials from Deep Learning”. In: *Nature* (2020).
- [63] Hiroharu Kato, Yoshitaka Ushiku, and Tatsuya Harada. “Neural 3D Mesh Renderer”. In: *Proc. International Conference on Computer Vision and Pattern Recognition (CVPR)*. 2018.
- [64] Mathieu Blondel, Olivier Teboul, Quentin Berthet, and Josip Djolonga. “Fast Differentiable Sorting and Ranking”. In: *Proc. Machine Learning Research (PMLR), International Conference on Machine Learning (ICML)*. 2020.
- [65] Matthew M. Loper and Michael J. Black. “OpenDR: An approximate differentiable renderer”. In: *Proc. European Conference on Computer Vision (ECCV)*. 2014.
- [66] Marin Vlastelica, Anselm Paulus, Vit Musil, Georg Martius, and Michal Rolínek. “Differentiation of blackbox combinatorial solvers”. In: *International Conference on Learning Representations (ICLR)*. 2020.
- [67] Maksim Lapin, Matthias Hein, and Bernt Schiele. “Top-k Multi-class SVM”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2015.

- [68] Ian J. Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.
- [69] Leonard Berrada, Andrew Zisserman, and M Pawan Kumar. “Smooth loss functions for deep top-k classification”. In: *International Conference on Learning Representations (ICLR)*. 2018.
- [70] Arthur Mensch and Mathieu Blondel. “Differentiable Dynamic Programming for Structured Prediction and Attention”. In: *Proc. Machine Learning Research (PMLR), International Conference on Machine Learning (ICML)*. 2018.
- [71] Caio Corro and Ivan Titov. “Differentiable Perturb-and-Parse: Semi-Supervised Parsing with a Structured Variational Autoencoder”. In: *International Conference on Learning Representations (ICLR)*. 2019.
- [72] Michal Rolínek, Vit Musil, Anselm Paulus, Marin Vlastelica, Claudio Michaelis, and Georg Martius. “Optimizing rank-based metrics with blackbox differentiation”. In: *Proc. International Conference on Computer Vision and Pattern Recognition (CVPR)*. 2020.
- [73] Sergey Shirobokov, Vladislav Belavin, Michael Kagan, Andrey Ustyuzhanin, and Atılım Günes Baydin. “Black-box Optimization with Local Generative Surrogates”. In: *Workshop on Real World Experiment Design and Active Learning at International Conference on Machine Learning (ICML)*. 2020.
- [74] Max B Paulus, Dami Choi, Daniel Tarlow, Andreas Krause, and Chris J Maddison. “Gradient estimation with stochastic softmax tricks”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2020.
- [75] Eric Jang, Shixiang Gu, and Ben Poole. “Categorical Reparameterization with Gumbel-Softmax”. In: *International Conference on Learning Representations (ICLR)*. 2017.
- [76] Marco Cuturi and Mathieu Blondel. “Soft-DTW: A Differentiable Loss Function for Time-Series”. In: *Proc. Machine Learning Research (PMLR), International Conference on Machine Learning (ICML)*. 2017.
- [77] Mathieu Blondel, Arthur Mensch, and Jean-Philippe Vert. “Differentiable Divergences Between Time Series”. In: *International Conference on Artificial Intelligence and Statistics (AISTATS)*. 2021.
- [78] Shib Sankar Dasgupta, Michael Boratko, Dongxu Zhang, Luke Vilnis, Xiang Lorraine Li, and Andrew McCallum. “Improving Local Identifiability in Probabilistic Box Embeddings”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2020.
- [79] Olivier Chapelle and Yi Chang. “Yahoo! Learning to Rank Challenge Overview”. In: *Proceedings of the Learning to Rank Challenge*. PMLR. 2011.
- [80] Tao Qin and Tie-Yan Liu. “Introducing LETOR 4.0 datasets”. In: *Computing Research Repository (CoRR) in arXiv* (2013).
- [81] Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. “Learning to Rank using Gradient Descent”. In: *Proc. Machine Learning Research (PMLR), International Conference on Machine Learning (ICML)*. 2005.
- [82] Christopher J Burges, Robert Ragno, and Quoc V Le. “Learning to rank with nonsmooth cost functions”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2007.

- [83] Michael Taylor, John Guiver, Stephen Robertson, and Tom Minka. “SoftRank: Optimizing non-smooth Rank Metrics”. In: *Proceedings of the 2008 International Conference on Web Search and Data Mining*. 2008.
- [84] Eyke Hüllermeier, Johannes Fürnkranz, Weiwei Cheng, and Klaus Brinker. “Label ranking by learning pairwise preferences”. In: *Artificial Intelligence* 172.16-17 (2008), pp. 1897–1916.
- [85] Hyunsung Lee, Sangwoo Cho, Yeongjae Jang, Jaekwang Kim, and Honguk Woo. “Differentiable Ranking Metric Using Relaxed Sorting for Top-K Recommendation”. In: *IEEE Access* (2021).
- [86] Robin Swezey, Aditya Grover, Bruno Charron, and Stefano Ermon. “PiRank: Learning To Rank via Differentiable Sorting”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2021.
- [87] Przemysław Pobrotyn and Radosław Białobrzęski. “NeuralNDCG: Direct Optimisation of a Ranking Metric via Differentiable Relaxation of Sorting”. In: *Computing Research Repository (CoRR) in arXiv* (2021).
- [88] Ryan Prescott Adams and Richard S Zemel. “Ranking via Sinkhorn Propagation”. In: *Computing Research Repository (CoRR) in arXiv* (2011).
- [89] Richard Sinkhorn and Paul Knopp. “Concerning nonnegative matrices and doubly stochastic matrices”. In: *Pacific Journal of Mathematics* 21.2 (1967), pp. 343–348.
- [90] Philip A Knight. “The Sinkhorn–Knopp algorithm: convergence and applications”. In: *SIAM Journal on Matrix Analysis and Applications* 30.1 (2008), pp. 261–275.
- [91] Weiwei Kong, Walid Krichene, Nicolas Mayoraz, Steffen Rendle, and Li Zhang. “Rankmax: An Adaptive Projection Alternative to the Softmax Function”. In: *Advances in Neural Information Processing Systems (NeurIPS)* (2020).
- [92] Tobias Plötz and Stefan Roth. “Neural Nearest Neighbors Networks”. In: *Advances in Neural Information Processing Systems (NeurIPS)* (2018).
- [93] Yujia Xie, Hanjun Dai, Minshuo Chen, Bo Dai, Tuo Zhao, Hongyuan Zha, Wei Wei, and Tomas Pfister. “Differentiable Top-k with Optimal Transport”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2020.
- [94] Kartik Goyal, Graham Neubig, Chris Dyer, and Taylor Berg-Kirkpatrick. “A continuous relaxation of beam search for end-to-end training of neural sequence models”. In: *AAAI Conference on Artificial Intelligence*. 2018.
- [95] Michał Pietruszka, Łukasz Borchmann, and Filip Galiński. “Successive Halving Top-k Operator”. In: *Computing Research Repository (CoRR) in arXiv* (2020).
- [96] Yash Patel, Giorgos Tolias, and Jiri Matas. “Recall@k Surrogate Loss with Large Batches and Similarity Mixup”. In: *Proc. International Conference on Computer Vision and Pattern Recognition (CVPR)*. 2022.
- [97] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. “Neural Machine Translation by jointly Learning to Align and Translate”. In: *International Conference on Learning Representations (ICLR)*. 2015.

- [98] Ronan Collobert, Awni Hannun, and Gabriel Synnaeve. “A Fully Differentiable Beam Search Decoder”. In: *Proc. Machine Learning Research (PMLR), International Conference on Machine Learning (ICML)*. 2019.
- [99] Dzmitry Bahdanau, Dmitriy Serdyuk, Philémon Brakel, Nan Rosemary Ke, Jan Chorowski, Aaron Courville, and Yoshua Bengio. “Task loss estimation for sequence prediction”. In: *Workshops at the International Conference on Learning Representations*. 2016.
- [100] Andrew N Carr, Quentin Berthet, Mathieu Blondel, Olivier Teboul, and Neil Zeghidour. “Self-supervised learning of audio representations from permutations with differentiable ranking”. In: *IEEE Signal Processing Letters* (2021).
- [101] Tao Huang, Zekang Li, Hua Lu, Yong Shan, Shusheng Yang, Yang Feng, Fei Wang, Shan You, and Chang Xu. “Relational Surrogate Loss Learning”. In: *International Conference on Learning Representations (ICLR)*. 2022.
- [102] Kexin Yi, Jiajun Wu, Chuang Gan, Antonio Torralba, Pushmeet Kohli, and Josh Tenenbaum. “Neural-symbolic vqa: Disentangling reasoning from vision and language understanding”. In: *Advances in Neural Information Processing Systems (NeurIPS)* (2018).
- [103] Kexin Yi, Chuang Gan, Yunzhu Li, Pushmeet Kohli, Jiajun Wu, Antonio Torralba, and Joshua B Tenenbaum. “Clevrer: Collision events for video representation and reasoning”. In: *International Conference on Learning Representations (ICLR)* (2020).
- [104] Chenxi Yang and Swarat Chaudhuri. “Safe Neurosymbolic Learning with Differentiable Symbolic Execution”. In: *International Conference on Learning Representations (ICLR)*. 2022.
- [105] Xinchun Yan, Jimei Yang, Ersin Yumer, Yijie Guo, and Honglak Lee. “Perspective Transformer Nets: Learning Single-View 3D Object Reconstruction without 3D Supervision”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2016.
- [106] Eldar Insafutdinov and Alexey Dosovitskiy. “Unsupervised Learning of Shape and Pose with Differentiable Point Clouds”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2018.
- [107] Wang Yifan, Felice Serena, Shihao Wu, Cengiz Öztireli, and Olga Sorkine-Hornung. “Differentiable Surface Splatting for Point-based Geometry Processing”. In: *ACM Transactions on Graphics* 38.6 (2019).
- [108] Yue Jiang, Dantong Ji, Zhizhong Han, and Matthias Zwicker. “SDFDiff: Differentiable Rendering of Signed Distance Fields for 3D Shape Optimization”. In: *Proc. International Conference on Computer Vision and Pattern Recognition (CVPR)*. 2020.
- [109] Delio Vicini, Sebastien Speierer, and Wenzel Jakob. “Differentiable Signed Distance Function Rendering”. In: *ACM Transactions on Graphics*. 2022.
- [110] Shichen Liu, Shunsuke Saito, Weikai Chen, and Hao Li. “Learning to Infer Implicit Surfaces without 3D Supervision”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2019.

- [111] Vincent Sitzmann, Michael Zollhöfer, and Gordon Wetzstein. “Scene Representation Networks: Continuous 3D-Structure-Aware Neural Scene Representations”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2019.
- [112] Michael Niemeyer, Lars Mescheder, Michael Oechsle, and Andreas Geiger. “Differentiable Volumetric Rendering: Learning Implicit 3D Representations without 3D Supervision”. In: *Proc. International Conference on Computer Vision and Pattern Recognition (CVPR)*. 2020.
- [113] Lior Yariv, Yoni Kasten, Dror Moran, Meirav Galun, Matan Atzmon, Basri Ronen, and Yaron Lipman. “Multiview neural surface reconstruction by disentangling geometry and appearance”. In: *Advances in Neural Information Processing Systems (NeurIPS)* (2020).
- [114] Boyang Deng, Kyle Genova, Soroosh Yazdani, Sofien Bouaziz, Geoffrey Hinton, and Andrea Tagliasacchi. “Cvxnet: Learnable convex decomposition”. In: *Proc. International Conference on Computer Vision and Pattern Recognition (CVPR)*. 2020.
- [115] Songyou Peng, Chiyu Jiang, Yiyi Liao, Michael Niemeyer, Marc Pollefeys, and Andreas Geiger. “Shape As Points: A Differentiable Poisson Solver”. In: *Advances in Neural Information Processing Systems (NeurIPS)* (2021).
- [116] Marie-Julie Rakotosaona, Noam Aigerman, Niloy J Mitra, Maks Ovsjanikov, and Paul Guerrero. “Differentiable Surface Triangulation”. In: *ACM Transactions on Graphics* (2021).
- [117] Minsu Cho, Aditya Balu, Ameya Joshi, Anjana Deva Prasad, Biswajit Khara, Soumik Sarkar, Baskar Ganapathysubramanian, Adarsh Krishnamurthy, and Chinmay Hegde. “Differentiable Spline Approximations”. In: *Advances in Neural Information Processing Systems (NeurIPS)* (2021).
- [118] Daniela Mihai and Jonathon Hare. “Differentiable Drawing and Sketching”. In: *Computing Research Repository (CoRR) in arXiv* (2021).
- [119] Daniela Mihai and Jonathon Hare. “Learning to Draw: Emergent Communication through Sketching”. In: *Advances in Neural Information Processing Systems (NeurIPS)* (2021).
- [120] Tzu-Mao Li, Michal Lukáč, Michaël Gharbi, and Jonathan Ragan-Kelley. “Differentiable Vector Graphics Rasterization for Editing and Learning”. In: *ACM Transactions on Graphics* (2020).
- [121] Cory Braker Scott. “Differentiable IFS Fractals”. In: *Computing Research Repository (CoRR) in arXiv* (2022).
- [122] Filipe de Avila Belbute-Peres, Kevin Smith, Kelsey Allen, Josh Tenenbaum, and J Zico Kolter. “End-to-End Differentiable Physics for Learning and Control”. In: *Advances in Neural Information Processing Systems (NeurIPS)* (2018).
- [123] Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, and Frédo Durand. “DiffTaichi: Differentiable Programming for Physical Simulation”. In: *International Conference on Learning Representations (ICLR)*. 2020.
- [124] Jonas Degraeve, Michiel Hermans, Joni Dambre, et al. “A Differentiable Physics Engine for Deep Learning in Robotics”. In: *Frontiers in Neurorobotics* (2019).

- [125] Philipp Holl, Vladlen Koltun, Kiwon Um, and Nils Thuerey. “phiflow: A Differentiable PDE Solving Framework for Deep Learning via Physical Simulations”. In: *Workshop on Differentiable Computer Vision, Graphics, and Physics in Machine Learning at NeurIPS 2020*. 2020.
- [126] Philipp Holl, Vladlen Koltun, and Nils Thuerey. “Learning to Control PDEs with Differentiable Physics”. In: *International Conference on Learning Representations (ICLR)*. 2020.
- [127] Vincent Sitzmann, Steven Diamond, Yifan Peng, Xiong Dun, Stephen Boyd, Wolfgang Heidrich, Felix Heide, and Gordon Wetzstein. “End-to-end optimization of optics and image processing for achromatic extended depth of field and super-resolution imaging”. In: *ACM Transactions on Graphics* (2018).
- [128] John Ingraham, Adam Riesselman, Chris Sander, and Debora Marks. “Learning protein structure with a differentiable simulator”. In: *International Conference on Learning Representations (ICLR)*. 2018.
- [129] Tianfan Fu, Wenhao Gao, Cao Xiao, Jacob Yasonik, Connor W Coley, and Jimeng Sun. “Differentiable Scaffolding Tree for Molecular Optimization”. In: *International Conference on Learning Representations (ICLR)*. 2022.
- [130] Anselm Paulus, Michal Rolínek, Vít Musil, Brandon Amos, and Georg Martius. “CombOptNet: Fit the Right NP-Hard Problem by Learning Integer Programming Constraints”. In: *Proc. Machine Learning Research (PMLR), International Conference on Machine Learning (ICML)*. 2021.
- [131] Sai Praveen Bangaru, Jesse Michel, Kevin Mu, Gilbert Bernstein, Tzu-Mao Li, and Jonathan Ragan-Kelley. “Systematically differentiating parametric discontinuities”. In: *ACM Transactions on Graphics* (2021).
- [132] Justin Domke. “Implicit differentiation by perturbation”. In: *Advances in Neural Information Processing Systems (NeurIPS)* (2010).
- [133] Mathias Niepert, Pasquale Minervini, and Luca Franceschi. “Implicit MLE: Backpropagating through discrete exponential family distributions”. In: *Advances in Neural Information Processing Systems (NeurIPS)* (2021).
- [134] Mathieu Blondel, Quentin Berthet, Marco Cuturi, Roy Frostig, Stephan Hoyer, Felipe Llinares-López, Fabian Pedregosa, and Jean-Philippe Vert. “Efficient and Modular Implicit Differentiation”. In: *Computing Research Repository (CoRR) in arXiv* (2021).
- [135] Brandon Amos and J Zico Kolter. “Optnet: Differentiable optimization as a layer in neural networks”. In: *Proc. Machine Learning Research (PMLR), International Conference on Machine Learning (ICML)*. 2017.
- [136] Akshay Agrawal, Brandon Amos, Shane Barratt, Stephen Boyd, Steven Diamond, and J Zico Kolter. “Differentiable Convex Optimization Layers”. In: *Advances in Neural Information Processing Systems (NeurIPS)* (2019).
- [137] Josip Djolonga and Andreas Krause. “Differentiable Learning of Submodular Models”. In: *Advances in Neural Information Processing Systems (NeurIPS)* (2017).

- [138] Michal Rolínek, Paul Swoboda, Dominik Zietlow, Anselm Paulus, Vít Musil, and Georg Martius. “Deep Graph Matching via Blackbox Differentiation of Combinatorial Solvers”. In: *Proc. European Conference on Computer Vision (ECCV)*. 2020.
- [139] Bertrand Charpentier, Simon Kibler, and Stephan Günnemann. “Differentiable DAG Sampling”. In: *International Conference on Learning Representations (ICLR)*. 2022.
- [140] Xun Zheng, Bryon Aragam, Pradeep K Ravikumar, and Eric P Xing. “DAGs with NO TEARS: Continuous Optimization for Structure Learning”. In: *Advances in Neural Information Processing Systems (NeurIPS)* (2018).
- [141] Lars Lorch, Jonas Rothfuss, Bernhard Schölkopf, and Andreas Krause. “DiBS: Differentiable Bayesian Structure Learning”. In: *Advances in Neural Information Processing Systems (NeurIPS)* (2021).
- [142] Rachid Riad, Olivier Teboul, David Grangier, and Neil Zeghidour. “Learning Strides in Convolutional Neural Networks”. In: *International Conference on Learning Representations (ICLR)*. 2022.
- [143] Oren Rippel, Jasper Snoek, and Ryan P Adams. “Spectral representations for convolutional neural networks”. In: *Advances in Neural Information Processing Systems (NeurIPS)* (2015).
- [144] David W Romero, Robert-Jan Brintjes, Jakub M Tomczak, Erik J Bekkers, Mark Hoogendoorn, and Jan C van Gemert. “FlexConv: Continuous Kernel Convolutions with Differentiable Kernel Sizes”. In: *International Conference on Learning Representations (ICLR)*. 2022.
- [145] Kevin Thandiackal, Boqi Chen, Pushpak Pati, Guillaume Jaume, Drew FK Williamson, Maria Gabrani, and Orcun Goksel. “Differentiable Zooming for Multiple Instance Learning on Whole-Slide Images”. In: *Computing Research Repository (CoRR) in arXiv* (2022).
- [146] Ricky TQ Chen, Brandon Amos, and Maximilian Nickel. “Semi-Discrete Normalizing Flows through Differentiable Voronoi Tessellation”. In: *ICLR Workshop on Deep Generative Models for Highly Structured Data*. 2022.
- [147] Adrien Corenflos, James Thornton, George Deligiannidis, and Arnaud Doucet. “Differentiable Particle Filtering via Entropy-Regularized Optimal Transport”. In: *Proc. Machine Learning Research (PMLR), International Conference on Machine Learning (ICML)*. 2021.
- [148] Jesse Engel, Lamtharn Hantrakul, Chenjie Gu, and Adam Roberts. “DDSP: Differentiable digital signal processing”. In: *International Conference on Learning Representations (ICLR)*. 2020.
- [149] Gonzalo Mena, David Belanger, Scott Linderman, and Jasper Snoek. “Learning Latent Permutations with Gumbel-Sinkhorn Networks”. In: *International Conference on Learning Representations (ICLR)*. 2018.
- [150] Will Grathwohl, Dami Choi, Yuhuai Wu, Geoffrey Roeder, and David Duvenaud. “Backpropagation through the Void: Optimizing control variates for black-box gradient estimation”. In: *International Conference on Learning Representations (ICLR)*. 2018.
- [151] George Tucker, Andriy Mnih, Chris J Maddison, John Lawson, and Jascha Sohl-Dickstein. “REBAR: Low-variance, unbiased gradient estimates for discrete latent variable models”. In: *Advances in Neural Information Processing Systems (NeurIPS)* (2017).

- [152] Mathieu Blondel, André FT Martins, and Vlad Niculae. “Learning with Fenchel-Young Losses”. In: *Journal of Machine Learning Research (JMLR)* (2020).
- [153] Swarat Chaudhuri and Armando Solar-Lezama. “Smoothing a Program Soundly and Robustly”. In: *Proceedings of the 23rd International Conference on Computer Aided Verification*. 2011.
- [154] Matko Bošnjak, Tim Rocktäschel, Jason Naradowsky, and Sebastian Riedel. “Programming with a Differentiable Forth Interpreter”. In: *Proc. Machine Learning Research (PMLR), International Conference on Machine Learning (ICML)*. 2017.
- [155] Alexander L. Gaunt, Marc Brockschmidt, Nate Kushman, and Daniel Tarlow. “Differentiable Programs with Neural Libraries”. In: *Proc. Machine Learning Research (PMLR), International Conference on Machine Learning (ICML)*. 2017.
- [156] John K Feser, Marc Brockschmidt, Alexander L Gaunt, and Daniel Tarlow. “Neural Functional Programming”. In: *Workshops at the International Conference on Learning Representations*. 2017.
- [157] Ameesh Shah, Eric Zhan, Jennifer J Sun, Abhinav Verma, Yisong Yue, and Swarat Chaudhuri. “Learning Differentiable Programs with Admissible Neural Heuristics”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2020.
- [158] William J Cody. “Rational Chebyshev approximations for the error function”. In: *Mathematics of Computation* 23.107 (1969), pp. 631–637.
- [159] Alex Graves, Greg Wayne, and Ivo Danihelka. “Neural Turing Machines”. In: *Computing Research Repository (CoRR) in arXiv* (2014).
- [160] Yann LeCun, Corinna Cortes, and CJ Burges. “MNIST Handwritten Digit Database”. In: (2010).
- [161] Owen Astrachan. “Bubble Sort: an Archaeological Algorithmic Analysis”. In: *ACM Sigcse Bulletin* 35.1 (2003), pp. 1–5.
- [162] Edsger W Dijkstra et al. “A note on two problems in connexion with graphs”. In: *Numerische Mathematik* 1.1 (1959), pp. 269–271.
- [163] Richard Bellman. “On a routing problem”. In: *Quarterly of Applied Mathematics* 16.1 (1958), pp. 87–90.
- [164] Angel X. Chang, Thomas Funkhouser, Leonidas Guibas, Pat Hanrahan, Qixing Huang, Zimo Li, Silvio Savarese, Manolis Savva, Shuran Song, Hao Su, Jianxiong Xiao, Li Yi, and Fisher Yu. “ShapeNet: An Information-Rich 3D Model Repository”. In: *Computing Research Repository (CoRR) in arXiv* (2015).
- [165] Gregory Cohen, Saeed Afshar, Jonathan Tapson, and André van Schaik. “EMNIST: an extension of MNIST to handwritten letters”. In: *Computing Research Repository (CoRR) in arXiv* (2017).
- [166] Vladimir I Levenshtein. “Binary Codes Capable of Correcting Deletions, Insertions, and Reversals”. In: *Soviet physics doklady*. Vol. 10. 8. 1966, pp. 707–710.
- [167] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching (2nd Ed.)* Addison Wesley, 1998.
- [168] Naga K. Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. “GPUTeraSort: High Performance Graphics Co-Processor Sorting for Large Database Management”. In: *SIGMOD Conference*. 2006.

- [169] Michal Bidlo and Michal Dobeš. “Evolutionary Development of Growing Generic Sorting Networks by Means of Rewriting Systems”. In: *IEEE Transactions on Evolutionary Computation* (2019).
- [170] Krzysztof C Kiwiel. “Convergence and Efficiency of Subgradient Methods for Quasiconvex Minimization”. In: *Mathematical Programming* 90 (2001).
- [171] Kenneth E Batchier. “Sorting networks and their applications”. In: *Proc. AFIPS Spring Joint Computing Conference (Atlantic City, NJ)*. 1968, pp. 307–314.
- [172] Tie-Yan Liu. “Learning to Rank for Information Retrieval”. In: (2011).
- [173] Rodica Ceterchi and Alexandru I. Tomescu. “Spiking Neural P Systems – A Natural Model for Sorting Networks”. In: *Proc. of the Sixth Brainstorming Week on Membrane Computing*. 2008.
- [174] Venkata Padmavati Metta and Alica Kelemenova. “Sorting Using Spiking Neural P Systems with Anti-spikes and Rules on Synapses”. In: *International Conference on Membrane Computing*. 2015.
- [175] Oriol Vinyals, Samy Bengio, and Manjunath Kudlur. “Order Matters: Sequence to sequence for sets”. In: *International Conference on Learning Representations (ICLR)*. 2016.
- [176] Sherenaz W Al-Haj Baddar and Kenneth E Batchier. *Designing sorting networks: A new paradigm*. Springer Science & Business Media, 2012.
- [177] M. Ajtai, J. Komlós, and E. Szemerédi. “An  $O(n \log n)$  Sorting Network”. In: *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*. 1983.
- [178] Michael Gowanlock and Ben Karsin. “A hybrid CPU GPU approach for optimizing sorting throughput”. In: *Parallel Computing* 85 (2019).
- [179] A Nico Habermann. “Parallel Neighbor-Sort (or the Glory of the Induction Principle)”. In: (1972).
- [180] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. “Reading digits in natural images with unsupervised feature learning”. In: (2011).
- [181] Ian J Goodfellow, Yaroslav Bulatov, Julian Ibarz, Sacha Arnoud, and Vinay Shet. “Multi-digit number recognition from street view imagery using deep convolutional neural networks”. In: *Computing Research Repository (CoRR) in arXiv* (2013).
- [182] Maksim Lapin, Matthias Hein, and Bernt Schiele. “Loss Functions for Top-k Error: Analysis and Insights”. In: *Proc. International Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016.
- [183] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. “CIFAR-10 (Canadian Institute for Advanced Research)”. In: (2009).
- [184] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. “ImageNet: A Large-Scale Hierarchical Image Database”. In: *Proc. International Conference on Computer Vision and Pattern Recognition (CVPR)*. 2009.
- [185] Tal Ridnik, Emanuel Ben-Baruch, Asaf Noy, and Lihi Zelnik-Manor. “Imagenet-21k Pretraining for the Masses”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2021.

- [186] Yanbo Fan, Siwei Lyu, Yiming Ying, and Bao-Gang Hu. “Learning with Average Top-k Loss”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2017.
- [187] Forest Yang and Sanmi Koyejo. “On the Consistency of Top-k Surrogate Losses”. In: *Proc. Machine Learning Research (PMLR), International Conference on Machine Learning (ICML)*. 2020.
- [188] Evgenii Chzhen, Christophe Denis, Mohamed Hebiri, and Titouan Lorieul. “Set-valued Classification—Overview via a Unified Framework”. In: *Computing Research Repository (CoRR) in arXiv* (2021).
- [189] Benjamin W. Wah and Kuo-Liang Chen. “A Partitioning Approach to the Design of Selection Networks”. In: *IEEE Transactions on Computers* 33 (3 1984), pp. 261–268.
- [190] Moshe Zazon-Ivry and Michael Codish. “Pairwise Networks are Superior for Selection”. Unpublished manuscript. 2012.
- [191] Michał Karpiński and Marek Piotrów. “Smaller Selection Networks for Cardinality Constraints Encoding”. In: *Proc. Principles and Practice of Constraint Programming (CP 2015, Cork, Ireland)*. 2015.
- [192] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. “Deep Residual Learning for Image Recognition”. In: *Proc. International Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016.
- [193] Dhruv Mahajan, Ross Girshick, Vignesh Ramanathan, Kaiming He, Manohar Paluri, Yixuan Li, Ashwin Bharambe, and Laurens Van Der Maaten. “Exploring the Limits of Weakly Supervised Pretraining”. In: *Proc. European Conference on Computer Vision (ECCV)*. 2018.
- [194] Qizhe Xie, Minh-Thang Luong, Eduard Hovy, and Quoc V Le. “Self-training with noisy student improves imagenet classification”. In: *Proc. International Conference on Computer Vision and Pattern Recognition (CVPR)*. 2020.
- [195] Chen Sun, Abhinav Shrivastava, Saurabh Singh, and Abhinav Gupta. “Revisiting Unreasonable Effectiveness of Data in Deep Learning Era”. In: *Proc. International Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017.
- [196] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. “An image is worth 16x16 words: Transformers for image recognition at scale”. In: *International Conference on Learning Representations (ICLR)*. 2021.
- [197] Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Joan Puigcerver, Jessica Yung, Sylvain Gelly, and Neil Houlsby. “Big Transfer (BiT): General Visual Representation Learning”. In: *Proc. European Conference on Computer Vision (ECCV)*. 2020.
- [198] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. “Learning transferable visual models from natural language supervision”. In: *Computing Research Repository (CoRR) in arXiv* (2021).

- [199] Chao Jia, Yinfei Yang, Ye Xia, Yi-Ting Chen, Zarana Parekh, Hieu Pham, Quoc V Le, Yunhsuan Sung, Zhen Li, and Tom Duerig. “Scaling up visual and vision-language representation learning with noisy text supervision”. In: *Proc. Machine Learning Research (PMLR), International Conference on Machine Learning (ICML)*. 2021.
- [200] Hieu Pham, Zihang Dai, Qizhe Xie, and Quoc V Le. “Meta Pseudo Labels”. In: *Proc. International Conference on Computer Vision and Pattern Recognition (CVPR)*. 2021.
- [201] Xiaohua Zhai, Alexander Kolesnikov, Neil Houlsby, and Lucas Beyer. “Scaling vision transformers”. In: *Computing Research Repository (CoRR) in arXiv* (2021).
- [202] Zihang Dai, Hanxiao Liu, Quoc V Le, and Mingxing Tan. “CoAt-Net: Marrying Convolution and Attention for All Data Sizes”. In: *Computing Research Repository (CoRR) in arXiv* (2021).
- [203] Federica Bogo, Angjoo Kanazawa, Christoph Lassner, Peter Gehler, Javier Romero, and Michael J Black. “Keep it SMPL: Automatic Estimation of 3D Human Pose and Shape from a Single Image”. In: *Proc. European Conference on Computer Vision (ECCV)*. 2016.
- [204] Andrea Palazzi, Luca Bergamini, Simone Calderara, and Rita Cucchiara. “End-to-End 6-DoF Object Pose Estimation Through Differentiable Rasterization”. In: *Proc. European Conference on Computer Vision Workshops (ECCVW)*. 2019.
- [205] Helge Rhodin, Nadia Robertini, Christian Richardt, Hans-Peter Seidel, and Christian Theobalt. “A Versatile Scene Model with Differentiable Visibility applied to Generative Pose Estimation”. In: *Proc. International Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015.
- [206] Hsueh-Ti Derek Liu, Michael Tao, Chun-Liang Li, Derek Nowrouzezahrai, and Alec Jacobson. “Adversarial Geometry and Lighting using a Differentiable Renderer”. In: *Computing Research Repository (CoRR) in arXiv* (2018).
- [207] Guillaume Loubet, Nicolas Holzschuch, and Jakob Wenzel. “Reparameterizing Discontinuous Integrands for Differentiable Rendering”. In: *ACM Transactions on Graphics (Proc. SIGGRAPH Asia)* 38.6 (2019).
- [208] Cheng Zhang, Bailey Miller, Kan Yan, Ioannis Gkioulekas, and Shuang Zhao. “Path-Space Differentiable Rendering”. In: *ACM Transactions on Graphics (Proc. SIGGRAPH)*. 2020.
- [209] Guilin Liu, Duygu Ceylan, Ersin Yumer, Jimei Yang, and Jyh-Ming Lien. “Material Editing using a Physically based Rendering Network”. In: *Proc. International Conference on Computer Vision (ICCV)*. 2017.
- [210] Liang Shi, Beichen Li, Miloš Hašan, Kalyan Sunkavalli, Tamy Boubekeur, Radomir Mech, and Wojciech Matusik. “MATch: Differentiable Material Graphs for Procedural Material Capture”. In: *ACM Transactions on Graphics (Proc. SIGGRAPH Asia)*. 2020.
- [211] Philipp Henzler, Niloy Mitra, and Tobias Ritschel. “Escaping Plato’s Cave using Adversarial Training: 3D Shape From Unstructured 2D Image Collections”. In: *Proc. International Conference on Computer Vision (ICCV)*. 2019.

- [212] Christoph H. Lampert Paul Henderson Vagia Tsiminaki. “Leveraging 2D Data to Learn Textured 3D Mesh Generation”. In: *Proc. International Conference on Computer Vision and Pattern Recognition (CVPR)*. 2020.
- [213] Hsueh-Ti Derek Liu, Michael Tao, Chun-Liang Li, Derek Nowrouzezahrai, and Alec Jacobson. “Beyond Pixel Norm-Balls: Parametric Adversaries using an Analytically Differentiable Renderer”. In: *International Conference on Learning Representations (ICLR)*. 2019.
- [214] Erich Peter Klement, Radko Mesiar, and Endre Pap. *Triangular Norms*. Springer Science & Business Media, 2013.
- [215] Emile van Krieken, Erman Acar, and Frank van Harmelen. “Analyzing Differentiable Fuzzy Logic Operators”. In: *Artificial Intelligence (2022)*.
- [216] Stuart Coles. *An Introduction to Statistical Modeling of Extreme Values*. Springer Series in Statistics. 2001.
- [217] Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. “PACT: Parameterized Clipping Activation for Quantized Neural Networks”. In: *Computing Research Repository (CoRR) in arXiv* (2018).
- [218] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. “Deep Learning with Limited Numerical Precision”. In: *Proc. Machine Learning Research (PMLR), International Conference on Machine Learning (ICML)*. 2015.
- [219] Haotong Qin, Ruihao Gong, Xianglong Liu, Xiao Bai, Jingkuan Song, and Nicu Sebe. “Binary Neural Networks: A Survey”. In: *Pattern Recognition* (2020).
- [220] Torsten Hoefer, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. “Sparsity in Deep Learning: Pruning and Growth for Efficient Inference and Training in Neural Networks”. In: *Computing Research Repository (CoRR) in arXiv* (2021).
- [221] Akbar Telikani, Amirhessam Tahmassebi, Wolfgang Banzhaf, and Amir H Gandomi. “Evolutionary Machine Learning: A Survey”. In: *ACM Computing Surveys (CSUR)* (2021).
- [222] J. Rapin and O. Teytaud. *Nevergrad - A gradient-free optimization platform*. 2018. URL: <https://GitHub.com/FacebookResearch/Nevergrad>.
- [223] George J. Klir and Bo Yuan. *Fuzzy Sets and Fuzzy Logic: Theory and Applications*. Prentice Hall, 1997.
- [224] Karl Menger. “Statistical metrics”. In: *Proceedings of the National Academy of Sciences of the United States of America* 28.12 (1942), p. 535.
- [225] Satrajit Chatterjee. “Learning and Memorization”. In: *Proc. Machine Learning Research (PMLR), International Conference on Machine Learning (ICML)*. 2018.
- [226] Tobias Brudermueller, Dennis L Shung, Adrian J Stanley, Johannes Stegmaier, and Smita Krishnaswamy. “Making Logic Learnable With Neural Networks”. In: *Computing Research Repository (CoRR) in arXiv* (2020).

- [227] Matthieu Zimmer, Xuening Feng, Claire Glanois, Zhaohui Jiang, Jianyi Zhang, Paul Weng, Li Dong, Hao Jianye, and Liu Wulong. “Differentiable Logic Machines”. In: *Computing Research Repository (CoRR) in arXiv* (2021).
- [228] Gang Chen. “Learning Symbolic Expressions via Gumbel-Max Equation Learner Network”. In: *Computing Research Repository (CoRR) in arXiv* (2020).
- [229] Chris J Maddison, Andriy Mnih, and Yee Whye Teh. “The Concrete Distribution: A Continuous Relaxation of Discrete Random Variables”. In: *International Conference on Learning Representations (ICLR)*. 2017.
- [230] Decebal Constantin Mocanu, Elena Mocanu, Peter Stone, Phuong H Nguyen, Madeleine Gibescu, and Antonio Liotta. “Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science”. In: *Nature Communications* (2018).
- [231] Adam Gaier and David Ha. “Weight Agnostic Neural Networks”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2019.
- [232] Valentina Zantedeschi, Matt Kusner, and Vlad Niculae. “Learning Binary Decision Trees by Argmin Differentiation”. In: *Proc. Machine Learning Research (PMLR), International Conference on Machine Learning (ICML)*. 2021.
- [233] Peter Clark and Tim Niblett. “The CN2 Induction Algorithm”. In: *Machine Learning* (1989).
- [234] MG Sarwar Murshed, Christopher Murphy, Daqing Hou, Nazar Khan, Ganesh Ananthanarayanan, and Faraz Hussain. “Machine Learning at the Network Edge: A Survey”. In: *ACM Computing Surveys (CSUR)* (2021).
- [235] Taiwo Samuel Ajani, Agbotiname Lucky Imoize, and Aderemi A Atayero. “An overview of machine learning within embedded and mobile devices—optimizations and applications”. In: *Sensors* 21.13 (2021).
- [236] Kah Phooi Seng, Paik Jen Lee, and Li Minn Ang. “Embedded intelligence on fpga: Survey, applications and challenges”. In: *Electronics* 10.8 (2021), p. 895.
- [237] Sérgio Branco, André G Ferreira, and Jorge Cabral. “Machine learning in resource-scarce embedded systems, FPGAs, and end-devices: A survey”. In: *Electronics* 8.11 (2019), p. 1289.
- [238] Sebastian B Thrun, Jerzy W Bala, Eric Bloedorn, Ivan Bratko, Bojan Cestnik, John Cheng, Kenneth A De Jong, Saso Dzeroski, Douglas H Fisher, Scott E Fahlman, et al. “The monk’s problems: A performance comparison of different learning algorithms”. In: (1991).
- [239] J. Ross Quinlan. “Induction of Decision Trees”. In: *Machine Learning* (1986).
- [240] Steven L Salzberg. “C4. 5: Programs for machine learning by j. ross quinlan. morgan kaufmann publishers, inc., 1993”. In: (1994).
- [241] Ronny Kohavi and Barry Becker. “UCI Machine Learning Repository: Adult Data Set”. In: (1996).
- [242] M Zwitter and M Soklic. “UCI Machine Learning Repository: Breast Cancer Dataset”. In: (1988).

- [243] Yaman Umuroglu, Nicholas J Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. “Finn: A Framework for Fast, Scalable Binarized Neural Network Inference”. In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2017.
- [244] Petar Jokic, Stephane Emery, and Luca Benini. “Binaryeye: A 20 kfps Streaming Camera System on FPGA with Real-Time On-Device Image Recognition Using Binary Neural Networks”. In: *IEEE 13th International Symposium on Industrial Embedded Systems (SIES)*. 2018.
- [245] Farinaz Koushanfar Mohammad Ghasemzadeh Mohammad Samragh. “ReBNet: Residual Binarized Neural Network”. In: *Proceedings of the 26th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2018.
- [246] Jinyu Zhan, Xingzhi Zhou, and Wei Jiang. “Field Programmable Gate Array-based All-Layer Accelerator with Quantization Neural Networks for Sustainable Cyber-Physical Systems”. In: *Software: Practice and Experience, Wiley Online Library* (2020).
- [247] Dmitry Molchanov, Arsenii Ashukha, and Dmitry Vetrov. “Variational Dropout Sparsifies Deep Neural Networks”. In: *Proc. Machine Learning Research (PMLR), International Conference on Machine Learning (ICML)*. 2017.
- [248] Christos Louizos, Max Welling, and Diederik P Kingma. “Learning Sparse Neural Networks through  $L_0$  Regularization”. In: *International Conference on Learning Representations (ICLR)*. 2018.
- [249] Gregor Urban, Krzysztof J Geras, Samira Ebrahimi Kahou, Ozlem Aslan, Shengjie Wang, Rich Caruana, Abdelrahman Mohamed, Matthai Philipose, and Matt Richardson. “Do Deep Convolutional Nets Really Need to be Deep and Convolutional?” In: *International Conference on Learning Representations (ICLR)*. 2018.
- [250] Song Han, Huizi Mao, and William J Dally. “Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding”. In: *International Conference on Learning Representations (ICLR)*. 2016.
- [251] Wenyuan Zeng and Raquel Urtasun. “MLPrune: Multi-Layer Pruning for Automated Neural Network Compression”. In: *OpenReview preprint* (2019).
- [252] Xiao Zhou, Weizhong Zhang, Hang Xu, and Tong Zhang. “Effective Sparsification of Neural Networks with Global Sparsity Constraint”. In: *Proc. International Conference on Computer Vision and Pattern Recognition (CVPR)*. 2021.
- [253] Itamar Shani, Liam Shaughnessy, John Rzasa, Alessandro Restelli, Brian R Hunt, Heidi Komkov, and Daniel P Lathrop. “Dynamics of Analog Logic-Gate Networks for Machine Learning”. In: *Chaos: An Interdisciplinary Journal of Nonlinear Science, AIP Publishing LLC* (2019).
- [254] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. “Dropout: a Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research (JMLR)* (2014).

- [255] Sangkug Lym, Esha Choukse, Siavash Zangeneh, Wei Wen, Sujay Sanghavi, and Mattan Erez. “PruneTrain: Fast Neural Network Training by Dynamic Sparse Model Reconfiguration”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2019.
- [256] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Gutttag. “What is the State of Neural Network Pruning?”. In: *Proceedings of the 3rd MLSys Conference, Austin, TX, USA*. 2020.
- [257] Naman Agarwal, Brian Bullins, and Elad Hazan. “Second-Order Stochastic Optimization for Machine Learning in Linear Time”. In: *Journal of Machine Learning Research (JMLR)* (2017).
- [258] J. Nocedal and S.J. Wright. *Numerical Optimization*. Springer New York, 2006.
- [259] Neha Wadia, Daniel Duckworth, Samuel S Schoenholz, Ethan Dyer, and Jascha Sohl-Dickstein. “Whitening and second order optimization both make information in the dataset unusable during training, and can reduce or prevent generalization”. In: *Proc. Machine Learning Research (PMLR), International Conference on Machine Learning (ICML)*. 2021.
- [260] D.P. Bertsekas. *Nonlinear Programming*. Athena Scientific, 1999.
- [261] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. “Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers”. In: *Foundations and Trends in Machine Learning* 3.1 (2011).
- [262] James Martens and Roger Grosse. “Optimizing Neural Networks with Kronecker-Factored Approximate Curvature”. In: *Proc. Machine Learning Research (PMLR), International Conference on Machine Learning (ICML)*. 2015.
- [263] Mert Pilanci and Martin J. Wainwright. “Newton Sketch: A Near Linear-Time Optimization Algorithm with Linear-Quadratic Convergence”. In: *SIAM Journal on Optimization* (2017).
- [264] Elias Frantar, Eldar Kurtic, and Dan Alistarh. “M-FAC: Efficient Matrix-Free Approximations of Second-Order Information”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2021.
- [265] Frederik Kunstner, Lukas Balles, and Philipp Hennig. “Limitations of the empirical Fisher approximation for natural gradient descent”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2019.
- [266] James Martens. “New Insights and Perspectives on the Natural Gradient Method”. In: *Journal of Machine Learning Research (JMLR)* (2020).
- [267] A. N. Tikhonov and V. Y. Arsenin. *Solutions of Ill-posed problems*. W.H. Winston, 1977.
- [268] Tarek R Besold, Artur d’Avila Garcez, Sebastian Bader, Howard Bowman, Pedro Domingos, Pascal Hitzler, Kai-Uwe Kühnberger, Luis C Lamb, Daniel Lowd, Priscila Machado Vieira Lima, et al. “Neural-symbolic learning and reasoning: A survey and interpretation”. In: *Computing Research Repository (CoRR) in arXiv* (2017).
- [269] Masataro Asai and Alex Fukunaga. “Classical planning in deep latent space: Bridging the subsymbolic-symbolic boundary”. In: *AAAI Conference on Artificial Intelligence*. 2018.

- [270] Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng. “NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis”. In: *Proc. European Conference on Computer Vision (ECCV)*. 2020.
- [271] Yuan Liu, Sida Peng, Lingjie Liu, Qianqian Wang, Peng Wang, Christian Theobalt, Xiaowei Zhou, and Wenping Wang. “Neural Rays for Occlusion-aware Image-based Rendering”. In: *Proc. International Conference on Computer Vision and Pattern Recognition (CVPR)*. 2022.
- [272] A. Tewari, O. Fried, J. Thies, V. Sitzmann, S. Lombardi, Z. Xu, T. Simon, M. Nießner, E. Tretschk, L. Liu, B. Mildenhall, P. Srinivasan, R. Pandey, S. Orts-Escolano, S. Fanello, M. Guo, G. Wetzstein, J.-Y. Zhu, C. Theobalt, M. Agrawala, D. B Goldman, and M. Zollhöfer. “Advances in Neural Rendering”. In: *ACM SIGGRAPH 2021 Courses*. 2021.
- [273] Stelian Coros, Miles Macklin, Bernhard Thomaszewski, and Nils Thürey. “Differentiable Simulation”. In: *ACM SIGGRAPH Asia 2021 Courses*. 2021.