ALTERNATIVE OPTIMIZATION METHODS FOR TRAINING OF LARGE DEEP NEURAL NETWORKS

Thesis approved by
the Department of Computer Science
University of Kaiserslautern-Landau
for the award of the Doctoral Degree
Doctor of Natural Sciences (Dr. rer. nat.)

to

Avraam Chatzimichailidis

Date of Defense:	29.03.2023
Dean:	Prof. Dr. Christoph Garth
Reviewer:	Prof. Dr. Nicolas R. Gauger
Reviewer:	Prof. Dr. Janis Keuper

DE-386

Abstract

Due to its performance, the field of deep learning has gained a lot of attention, with neural networks succeeding in areas like *Computer Vision* (CV), *Neural Language Processing* (NLP), and *Reinforcement Learning* (RL). However, high accuracy comes at a computational cost as larger networks require longer training time and no longer fit onto a single GPU. To reduce training costs, researchers are looking into the dynamics of different optimizers, in order to find ways to make training more efficient. Resource requirements can be limited by reducing model size during training or designing more efficient models that improve accuracy without increasing network size.

This thesis combines eigenvalue computation and high-dimensional loss surface visualization to study different optimizers and deep neural network models. Eigenvectors of different eigenvalues are computed, and the loss landscape and optimizer trajectory are projected onto the plane spanned by those eigenvectors. A new parallelization method for the stochastic Lanczos method is introduced, resulting in faster computation and thus enabling high-resolution videos of the trajectory and second-order information during neural network training. Additionally, the thesis presents the loss landscape between two minima along with the eigenvalue density spectrum at intermediate points for the first time.

Secondly, this thesis presents a regularization method for *Generative Adversarial Networks* (GANs) that uses second-order information. The gradient during training is modified by subtracting the eigenvector direction of the biggest eigenvalue, preventing the network from falling into the steepest minima and avoiding mode collapse. The thesis also shows the full eigenvalue density spectra of GANs during training.

Thirdly, this thesis introduces ProxSGD, a proximal algorithm for neural network training that guarantees convergence to a stationary point and unifies multiple popular optimizers. Proximal gradients are used to find a closed-form solution to the problem of training neural networks with smooth and non-smooth regularizations, resulting in better sparsity and more efficient optimization. Experiments show that ProxSGD can find sparser networks while reaching the same accuracy as popular optimizers.

Lastly, this thesis unifies sparsity and *neural architecture search* (NAS) through the framework of group sparsity. Group sparsity is achieved through $\ell_{2,1}$-regularization during training, allowing for filter and operation pruning to reduce model size with minimal sacrifice in accuracy. By grouping multiple operations together, group sparsity can be used for NAS as well. This approach is shown to be more robust while still achieving competitive accuracies compared to state-of-the-art methods.

Acknowledgements

First of all, this endeavor would not have been possible without my supervisor Prof. Janis Keuper. He gave me the opportunity to pursue this path even though I had no prior experience in the field of deep learning. I am immensely grateful for his guidance, support, and feedback throughout this journey. I will always remember his mentorship with fondness. I would also like to express my heartfelt thanks to Prof. Nicolas Gauger, who was always available to help and advise me. I could not have asked for a better professor on my academic journey. Additionally, I would like to thank Prof. Sören Laue for his time and effort in serving as the head of my committee.

I am deeply grateful to my colleague Yang Yang, who was like a second supervisor to me. He generously shared his time, knowledge, and expertise, helping me countless times and providing valuable feedback on my research. I owe him a great debt of gratitude and I am fortunate to have had him as a colleague and friend. I would like to express my gratitude to my friends and colleagues Kalun Ho and Ricard Durall Lopez, with whom I not only found extraordinary friends, but also research collaborators. I appreciate the countless discussions about various topics and ideas we had, as well as the equally important hours we spent at the foosball table at our institute.

I owe a huge debt of gratitude to my parents, Nektaria Tsetsiou and Eleftherios Chatzimichailidis, who have always supported and encouraged me throughout my academic journey. Their unwavering belief in my abilities and their constant guidance have been instrumental in shaping me into the person I am today. I could not have achieved any of my goals without their love, sacrifices, and unwavering support. I am truly blessed to have such amazing parents. I would like to express my deep appreciation to Christina Chatzimichailidou, who has been the best sister I could have ever hoped for. I cannot thank her enough for all the times she has been there for me, and I know that I would not be where I am today without her. I would also like to express my gratitude to my grandparents, Eleftheria Tsetsiou and Aristidis Tsetsios, who came from Greece to Germany with nothing in their pockets, and who inspire me daily to never give up on my dreams. Last but not least, I am incredibly grateful for my partner, Georgia Blioumi. Her constant support, love, and guidance have been instrumental in helping me reach my goals, and I cannot thank her enough. She is a constant source of inspiration, and I feel truly blessed to have her in my life. I hope to make her as proud of me as I am of her.

April 15, 2023, Avraam Chatzimichailidis

Contents

1. Introduction 1
- 1.1. From Machine Learning to Deep Learning 2
- 1.2. Overview Deep Learning 5
- 1.3. Deep Learning Models 8
- 1.4. Datasets 15

2. Methods 19
- 2.1. Optimization Methods in Deep Learning 19
- 2.2. Regularization 30
- 2.3. Eigenvalue Computation 37

3. Loss Surface Visualization 53
- 3.1. Overview 53
- 3.2. Principal Component Analysis in High-Dimensional Spaces 55
- 3.3. Loss Surface Visualization through Neural Network Eigenvectors . . 56
- 3.4. Parallelization of Loss Surface Visualization 59
- 3.5. Experiments 63

4. Stabilizing GANs through Eigenvalue Regularization 69
- 4.1. General Adversarial Networks 69
- 4.2. Mode Collapse 74
- 4.3. Loss Surface of GANs 77
- 4.4. NudgedAdam: Preventing GANs from Mode Collapse 79

5. Proximal Gradient Methods 85
- 5.1. Proximal Gradient Optimization 86
- 5.2. ProxSGD 93
- 5.3. ProxSGD with ℓ_1-Regularization 99

6. ProxSGD with $\ell_{2,1}$-Regularization 107
- 6.1. Overview Neural Architecture Search 107
- 6.2. GSparsity: ProxSGD with $\ell_{2,1}$-Regularization 111
- 6.3. Filter Pruning 113
- 6.4. Operation Pruning 115
- 6.5. Neural Architecture Search with GSparsity 117

7. Conclusions and Outlook 127

Bibliography 131

A. Proof of Theorem 5.1 149

B. ResNet-50 Structure after Pruning 155

C. CIFAR-10 and CIFAR-100 on DARTS Search Space 157

D. Convergence of ProxSGD vs. SGD 159

E. GSparsity and NAS: Ablation Study 165

F. Publications in the context of this work 175

G. Curriculum Vitae 177

List of Figures

1.1. Illustration of a Linearly Separable Dataset 3
1.2. Example of a Non-Linear Transformation on a Dataset 4
1.3. Illustration of the Training Pipeline in Deep Learning 5
1.4. Neural Network Training for one Epoch 7
1.5. Illustration of a Fully-Connected Layer 9
1.6. Illustration of a Convolution Layer 10
1.7. Illustration of a Batch-Normalization Layer 12
1.8. Activation Functions 13
1.9. LeNet-5 Architecture 14
1.10. ResNet Architecture 15
1.11. Examples of common Deep Learning Datasets 17

2.1. Illustration of different Types of Minima 20
2.2. Depiction of different Loss Functions and their Gradient 23
2.3. Steepest Descent and Newton's method on McCormick's function . 27
2.4. Comparison of steepest descent and Newton's method on Himmelblau's function 29
2.5. Depiction of Regularization Functions 31
2.6. Effect of ℓ_2-Norm on the Solution 34
2.7. Difference between the Squared ℓ_2-Norm and the Regular ℓ_2-Norm 35
2.8. Block Soft Thresholding Operator 37
2.9. Forward Pass of the $\mathcal{R}$-Operator 41
2.10. Hessian Eigenvalues and Gaussian Curvature 46
2.11. Data-Parallel Lanczos Method 49
2.12. Iteration Parallel Stochastic Lanczos Quadrature Algorithm 50
2.13. Speedup Comparison of Parallel Stochastic Lanczos Quadrature Methods 51

3.1. Illustration of Projecting an Optimization Surface onto a Line 54
3.2. Comparison between different Visualization Methods 57
3.3. Loss Landscape Visualization using different Eigenvectors 58
3.4. Illustration of the Parallel Visualization Method 61
3.5. Strong Scaling Plot for the Parallel Visualization Algorithm 62
3.6. Absolute Runtime and Efficiency Plots for the Parallel Visualization Method 62

3.7. LeNet-5 per Iteration Loss Landscape and Eigenvalue Visualizations 64
3.8. ResNet-32 per Iteration Loss Landscape and Eigenvalue Visualizations 66
3.9. Visualization of the Interpolation between two Minima 67

4.1. Illustration of a Generative Adversarial Network 70
4.2. Visualization of the Learning Process of GANs 72
4.3. Plots of the GANs Losses 74
4.4. Depiction of the Difference between using KL-Divergence and Reverse KL-Divergence 76
4.5. GAN Loss Landscape Visualization 78
4.6. Evolution of the Top-k Eigenvalues of GANs for different Datasets . 79
4.7. Depiction of the Idea behind NudgedAdam 80
4.8. Evolution of the Top-k Eigenvalues of NSGAN at different Epochs . 81
4.9. Spectral Density Plots of the Generator and Discriminator 82

5.1. Visualization of Functions and their Subdifferential 87
5.2. Depiction of the Subgradient Method 88
5.3. Visualization of the Trajectory of SGD on ResNet-32 with ℓ_1-Regularization 89
5.4. Visualization of the Difference between the Subgradient Method and Proximal Point Method 92
5.5. Depiction of the Soft-Thresholding Function for different values of the Parameter λ 100
5.6. Visualization of the Trajectory of ProxSGD on ResNet-32 with ℓ_1-Regularization 101
5.7. Performance comparison for DenseNet-201 on CIFAR-100. 104
5.8. Hyperparameters and Sparsity for DenseNet-201 on CIFAR-100. . . 105

6.1. Illustration of the Process and the Main Components used in Neural Architecture Search 108
6.2. Illustration of a Cell as used in Differentiable Architecture Search . 109
6.3. Accuracy and CDF Plots for Filter Pruning 116
6.4. Percentage of Parameters pruned vs. the Accuracy for Operation pruning. 117
6.5. Illustrative Example of applying the Group Sparsity Approach to NAS 118
6.6. Example of one Architecture found by GSparsity on CIFAR-10 . . . 122
6.7. Example of one Architecture found by GSparsity on CIFAR-100 . . 123

D.1. Neural Architecture Search using SGD and ProxSGD 160
D.2. Normal cell after training with SGD with $\mu = 50$ and fixed $\varepsilon = 0.0001$. 161
D.3. Reduction cell after training with SGD with $\mu = 50$ and fixed $\varepsilon = 0.0001$. 162
D.4. Normal and reduction cell for $\mu = 50$ trained with ProxSGD. 163

E.1. Resulting normal cell after training with GSparsity with $\mu = 0.1$ and fixed $\varepsilon = 0.001$. 166

E.2. Resulting network structure after training with GSparsity with $\mu = 0.1$ and fixed $\varepsilon = 0.001$. 167
E.3. Resulting normal cell after training with GSparsity with $\mu = 1$ and fixed $\varepsilon = 0.001$. 168
E.4. Resulting network structure after training with GSparsity with $\mu = 1$ and fixed $\varepsilon = 0.001$. 169
E.5. Resulting normal cell after training with GSparsity with $\mu = 10$ and fixed $\varepsilon = 0.001$. 170
E.6. Resulting network structure after training with GSparsity with $\mu = 10$ and fixed $\varepsilon = 0.001$. 171
E.7. Resulting network structure after training with GSparsity with $\mu = 50$ and fixed $\varepsilon = 0.001$. 172
E.8. Resulting network structure after training with GSparsity with $\mu = 100$ and fixed $\varepsilon = 0.001$. 173
E.9. Resulting network structure after training with GSparsity with $\mu = 200$ and fixed $\varepsilon = 0.001$. 173
E.10. Resulting network structure after training with GSparsity with $\mu = 500$ and fixed $\varepsilon = 0.001$. 174
E.11. Resulting network structure after training with GSparsity with $\mu = 1000$ and fixed $\varepsilon = 0.001$. 174

Acronyms

χ^l	Activation function of layer l
$\boldsymbol{D}_k$	Preconditioning matrix at iteration k
$\boldsymbol{D}$	Discriminator network
$\boldsymbol{G}$	Generator network
$\boldsymbol{H}$	Hessian matrix
$\boldsymbol{K}_{a,b,p.q}$	Value of convolutional filter of input channel p, output channel q at filter height a and filter width b
$\boldsymbol{L}$	Lipschitz constant
$\boldsymbol{\Phi}$	Spectral density
$\boldsymbol{\delta}_{\mathbb{W}}(w)$	Indicator function
$\boldsymbol{\epsilon}$	Learning rate
$\boldsymbol{\kappa}$	Gaussian curvature
$\boldsymbol{\lambda}$	Eigenvalue
$N(x;\mu,\sigma)$	Gaussian function at x for mean μ and standard deviation σ
$\boldsymbol{\mu}$	Regularization parameter
$\boldsymbol{\phi}$	Non-linear transformation function
$\boldsymbol{\rho}$	Stepsize hyperparameter for momentum
$\mathbb{F}(k)$	Trajectory generated by an optimizer at iteration k
$\mathbb{M}(k)$	Minibatch of samples at iteration k
$\mathbb{W}$	Constraint set of parameters
$\boldsymbol{\sigma}$	Standard deviation
$\boldsymbol{\tau}$	Quadratic gain of approximation subproblem
$\boldsymbol{\theta}$	Trainable parameters of the generator network
$\boldsymbol{\zeta}$	Trainable parameters of the discriminator network
$\boldsymbol{b}^\star$	Filter-normalized directional vector
$\boldsymbol{b}$	Bias parameter
$\boldsymbol{f},\mathcal{L}$	Objective function
$\boldsymbol{g}$	Gradient of the objective function

$\boldsymbol{h}^l$	Output of activation function of layer *l*
$\boldsymbol{m}(k)$	Number of samples in a minibatch at iteration *k*
$\boldsymbol{r}$	Regularization function
$\boldsymbol{v}$	Momentum
$\boldsymbol{w}^\star$	Optimal parameters
$\boldsymbol{w}$	Trainable parameters
$\boldsymbol{x}$	Input sample
$\boldsymbol{y}$	Data label
$\boldsymbol{z}^l$	Output of layer *l*
$\mathfrak{u}$	Eigenvector
$\mathcal{D}$	Dataset
$\mathcal{F}$	Layer function
$\boldsymbol{\psi}$	Neural network model

Chapter 1

Introduction

In recent years, the field of deep learning has gained a significant amount of interest. World leading performance in various different tasks, like image recognition [49, 176, 235], speech recognition [15, 43], speech translation [61, 237] and playing games of Go [202] or no-limit Texas Hold'em [33, 34] has partly been achieved thanks to recent advances in the field of deep learning. Deep learning methods are able to tackle a variety of problems, from image recognition on various datasets to segmentation [145, 230, 231] and inpainting [40, 239]. Also, deep learning outperforms other methods in different datasets on clustering [103, 156], outlier detection [85, 126] and synthetic data generation [72, 181]. There exist many other fields where deep learning has had a significant impact [67, 127, 201].

First developed by Frank Rosenblatt in 1958 [189], his 3-layer perceptron is one of the earliest deep learning models. In 1990, Yann LeCun applied the backpropagation algorithm [108] to his neural network and was able to train this model on handwritten digits [122]. Over time, as compute power increased, it was possible to train bigger and bigger networks and with the introduction of new deep learning models like LSTMs [93], deep belief networks [88] and generative adversarial networks [72], deep learning has gained significant popularity in various fields.

The reasons for the sudden spike in deep learning applications are twofold. Firstly, an increasing amount of available data in the past years has boosted the applicability of data driven methods like machine learning and deep learning [216]. Secondly, computational capabilities have increased to a point where deep learning models of reasonable size can be trained in a reasonable time frame [216]. Here, one important component is the switch to *Graphics Processing Units* (GPUs) for neural network training. GPUs have had significant increases in computational power as well as memory in the past. Their widespread use has made deep learning more accessible to a wide audience and their parallel compute capabilities have sped up the training of deep learning models significantly [227].

In deep learning, artificial neural networks are trained in order to learn the relationship between input and output variables. These networks have parameters that are updated iteratively, with the goal of reducing the discrepancy between the network prediction for different samples of a dataset and their true label. This discrepancy is usually measured through a cost function (see Section 2 for an in-depth discussion).

Finding the optimal parameters of a neural network, that allow it to make correct predictions on its training set, is extremely difficult and this problem has been shown to be NP-hard [26, 105]. Thus, neural network training boils down to finding a suitable local minimum of the optimization surface instead. The use of an optimizer helps navigate this surface, in order to find a point that minimizes the cost function locally.

Even though deep learning methods have achieved impressive performance on various tasks, these performance gains are starting to stagnate. As explained in [222], neural networks have grown tremendously in their number of parameters and if they continue to grow at this rate, they will soon hit a ceiling where hardware and monetary cost for training these models would be prohibitive. This has tremendous impact on the environment as well, as the environmental cost scales with the computation required for training the neural network model. As shown by the authors of this paper, the computational cost for training a deep learning model scales at least quadratically with the number of samples in the dataset. Also, using statistical learning theory, one can show that popular error metrics like root mean square error can only drop with $1/\sqrt{n}$, where n represents the number of samples in the dataset. Combined, this means that the computational cost scales with $O(performance^4)$. By fitting the size of deep learning models to their corresponding performance, the authors find that the computational cost grows as a ninth-order polynomial with respect to performance. This empirical figure is five orders of magnitude higher than the theoretically lowest value. They also find that the computational cost is able to explain 43% of the variance in the performance. This shows that much of the performance improvement seen in many fields of deep learning can be attributed to increases in computing power. If this trend continues, there will soon be a hardware limit that is going to be hit. Moving from one of the current best errors of 11.5% on ImageNet [224], a challenging classification dataset, to an error of 5% in the future, will require approximately 5 orders of magnitude more computational power and cost around 100 billion dollars to train the deep neural network [222]. In sight of current computational limits, this means that the field of deep learning has to come up with solutions that lower its computational burden, or discover other methods that are able to scale closer to the theoretical bound.

This thesis will tackle different areas in deep learning that can help lower the computational burden of training neural networks. In this chapter the basic concepts of machine learning are introduced and the connection to deep learning is made. Next, the basic ideas behind deep learning are explored together with the usual pipeline used for training these models. Afterwards, the structure and the different parts of neural network models are shown, together with some important architectures that first introduced those new concepts. The last part introduces different datasets that serve as benchmarks for assessing the performance of various neural network models, thus making them important to the deep learning community as well as to this thesis.

1.1. From Machine Learning to Deep Learning

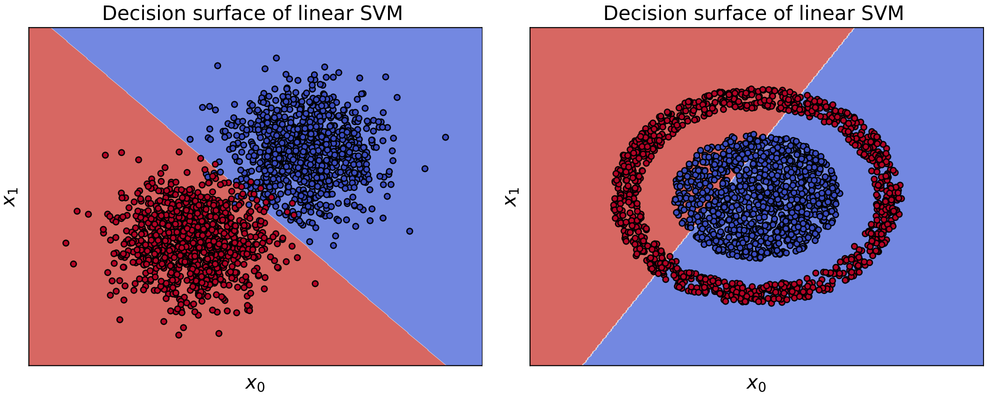
Deep learning has emerged as a subfield of machine learning. Thus, it is important to first learn the concepts of classical machine learning algorithms and to understand the difference between those and deep learning algorithms. Machine learning emerged as a means of automated methods for data analysis. These methods are able to detect pattern in the data instead of relying on a rules based approach [158].

In this thesis, the main focus is on the supervised setting, which refers to the case where the dataset is labeled. The dataset is given by the set $\mathcal{D} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^{n}$. The input samples $\boldsymbol{x}_i$ are called features and the y_i are the corresponding labels. There are cases, like in the unsupervised setting or in reinforcement learning, where no labels exist.

Machine learning can be formalized as a function approximation problem [158]. Assuming that there exists some function f that maps each $\boldsymbol{x}$ to a specific label $y = f(\boldsymbol{x})$, the goal is to learn this function given the dataset $\mathcal{D}$. For example, in binary classification each datapoint is assigned a distinct label, with $y \in \{-1, 1\}$. For a linear classifier this results in the following function

$$f(\boldsymbol{x}; \boldsymbol{w}, b) = sign(\boldsymbol{w}^T \boldsymbol{x} + b) \tag{1.1}$$

where $\boldsymbol{w}$ are the trainable parameters of the linear classifier and b its trainable bias parameter. The goal is to learn the parameters in such a way that the linear classifier is able separate the two classes of the dataset. Figure 1.1 depicts the linear classifier on two cases. The left hand side shows the case where the data is linearly separable, while the right hand side shows a case where the data is not linearly separable.

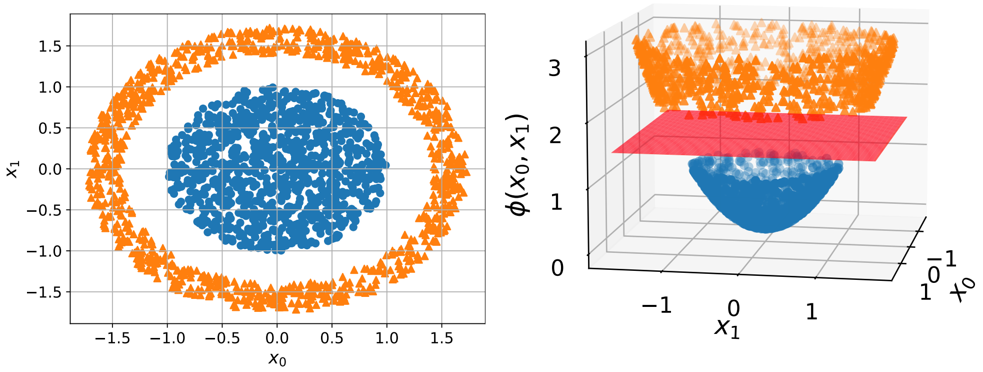


(a) *Depiction of a linear classifier on a linearly separable dataset. The dataset is generated using two multivariate Gaussian distributions with equal equal covariance matrix and different mean vectors.*

(b) *Depiction of a linear classifier on a dataset which is not linearly separable. The dataset is generated by randomly sampling different angles and restricting the radius for the different classes.*

Figure 1.1.: *Depiction of a linear classifier on two different datasets. Plot (a) shows a linearly separable dataset, which is correctly classified, while plot (b) shows a dataset which is not linearly separable. In this case the linear classifier is not able to separate the two classes.*

In order to still be able to classify the dataset on right hand side of Figure 1.1, a non-linear map $\phi(\boldsymbol{x})$ can be used, that transforms the feature vectors into a new (usually high-dimensional) space where they are now linearly separable. Kernel methods [4] offer an efficient way for converting different linear algorithms into a non-linear version.



(a) *Depiction of a dataset which is not linearly separable in two dimensions*

(b) *Non-linear transformation of the dataset using $\phi(x_0,x_1) = (x_0,x_1,x_0^2+x_1^2)$. The red plane represents the linear decision boundary in the transformed space.*

Figure 1.2.: *Depiction of the effect of a non-linear transformation on a dataset. Plot (a) depicts the original dataset in two dimensions, while plot (b) shows the same dataset after a non-linear transformation. This is now linearly separable in the new space, as can be seen by the red plane between the clusters of both classes.*

Using a non-linear transformation $\phi(\boldsymbol{x})$ turns Equation (1.1) into

$$f(\boldsymbol{x}) = sign(\hat{\boldsymbol{w}}^T \phi(\boldsymbol{x}) + b) \tag{1.2}$$

Figure 1.2 depicts how the dataset in Figure 1.2a, which is not linearly separable, can be transformed into a linearly separable set using a polynomial function. The transformed dataset is shown in Figure 1.2b.

Sometimes it is not clear how to choose an appropriate non-linear map, which makes it hard to achieve good performance with classical machine learning methods. Deep neural networks are parametrized non-linear maps $\phi_{\boldsymbol{w}}(\boldsymbol{x})$, that are able to learn a suitable feature representation from the data. One difference between classical machine learning approaches and deep learning is that while machine learning relies on hand-crafted features, deep learning is able to automatically learn those directly from the data. This distinction allows deep neural networks to achieve better performance on high-dimensional complex datasets, where manually constructing features is too complex or tedious [101]. Some of the benefits of classical machine learning algorithms is that they are typically faster than deep learning models, which is especially useful when dealing with large amounts of data. Similar to deep learning, they are also able to work with data that is not linearly separable by using kernel methods. The main drawback is the feature representation and for some datasets, e.g. images, it is not clear how to choose a feature representation in order for those classical machine learning models to work properly.

1.2. Overview Deep Learning

A sketch of the general pipeline used in deep learning is shown in Figure 1.3.

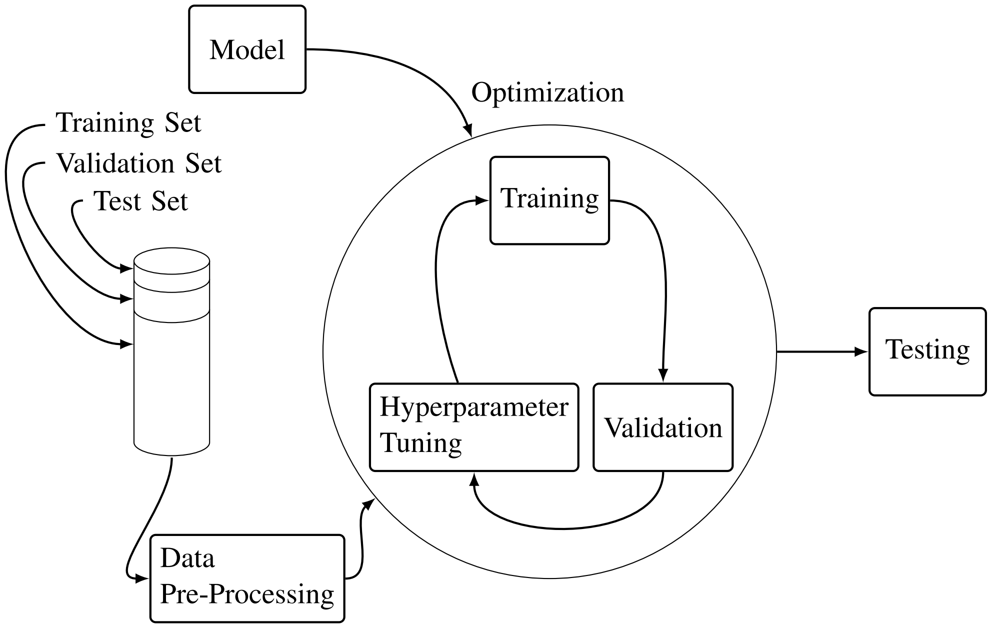


Figure 1.3.: *Illustration of the overall training pipeline used in deep learning. The dataset is split into a training-, validation- and test-set. These samples are pre-processed and used for optimization of a certain model. After optimization the final trained model is assessed by evaluating it using the test-set.*

For every deep learning problem the first step is to collect some data for which one wants to learn the relationship between the input samples and the corresponding labels. This dataset is then pre-processed in order to handle missing or corrupted values and to transform the data in a way that increases the efficiency of training the neural network [161]. Afterwards, a suitable model is selected, together with a cost function (also called loss function) which measures how close the network predictions are to the true label. The model choice and the cost function are both important for the problem, as a suitable choice can lead to better performance, more stable training and better generalization to unseen data [5].

Next, the neural network is trained on the dataset. The performance of the network can be monitored by looking at its accuracy on the training and validation data. If the performance of the neural network is satisfactory after training, it is evaluated using a separate dataset of unseen data which is called the test set. Otherwise, the hyperparameters of the optimization algorithm are tuned and the model is trained again.

Data Pre-Processing In this step of the process, the dataset is first split into a training set, a validation set and a test set. A careful choice has to be made on how to split the dataset into these three parts, in order to have the maximum amount of training data with enough samples in the validation and test set in order to accurately assess the network performance on unseen data. There exist methods that are able to utilize the entire dataset during training, like k-fold cross-validation [82], though their use is limited due to the size of the datasets and the longer training time of deep neural networks compared to their faster machine learning counterparts. There exist a multitude of pre-processing techniques that are used on the data. Broadly speaking, pre-processing is utilized in order to clean, normalize or transform and augment the data.

The original dataset can have missing values or some parts of the data can be corrupted. There exist many different techniques for filling missing values, such as replacing them with a constant value [80] or by using the mean of some parts of the data [147]. Normalization refers to transformations of the features that change their numerical values. This is done in order to make sure that the range of the features does not vary significantly. There are different ways of transforming the data, such as Min-max or Z-score normalization [80]. Depending on the task at hand, sometimes other transformations are used in order to append the dataset with additional samples. This is also referred to as data augmentation. When dealing with images, it is common to apply random rotations and crops to an image or to flip the image on some axis. This creates new data samples and also trains the network to be invariant to the rotation, position and orientation of the objects in the images [36]. Another type of data pre-processing introduces corruptions to existing samples. These corruptions typically refer to different types and levels of noise that are added to the samples. This ensures that the network learns robust features of the input images and does not overfit on high frequency noise present in the samples. This area of research is also commonly know as adversarial training [25]. The right pre-processing and data augmentation steps can increase the accuracy of the deep learning model [129].

Model Selection Selecting or crafting a suitable model is a crucial step in order to achieve good performance. Different problems require different neural network architectures and selecting the right layers can greatly improve the generalization accuracy. By increasing the number of parameters and thus making the model bigger, the accuracy can sometimes be improved as well [86, 106, 164, 165]. This has inflated the model size over the past years and many require distributed training [51] over multiple GPUs in order to be able to fit the model onto the hardware.

Neural Network Training An illustration of neural network training or optimization is shown in Figure 1.4. In this step, the different samples of the training set are fed into the model, which returns an output distribution. The output of the network is evaluated using the loss function. In the next step the gradient of the loss with respect to the parameters of the model is computed. This is done in order to update the parameters of the network in a meaningful way. The gradient is computed efficiently using backpropagation [123]. This refers to the fact that the gradient at each parameter can be efficiently computed by using the chain-rule. There exist a multitude of different optimizers, where most use gradient information in order to train the network efficiently.

Instead of pushing the entire dataset through the network in order to compute one

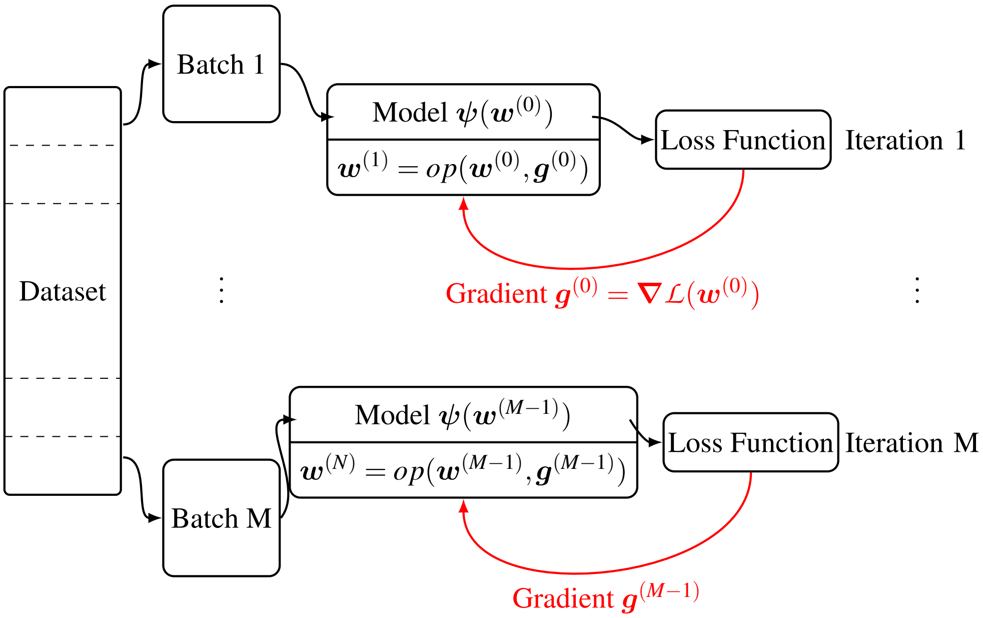

Batch 1
Model $\psi(\boldsymbol{w}^{(0)})$
$\boldsymbol{w}^{(1)} = op(\boldsymbol{w}^{(0)}, \boldsymbol{g}^{(0)})$
Loss Function
Iteration 1
Dataset
Gradient $\boldsymbol{g}^{(0)} = \nabla\mathcal{L}(\boldsymbol{w}^{(0)})$
Model $\psi(\boldsymbol{w}^{(M-1)})$
$\boldsymbol{w}^{(N)} = op(\boldsymbol{w}^{(M-1)}, \boldsymbol{g}^{(M-1)})$
Loss Function
Iteration M
Batch M
Gradient $\boldsymbol{g}^{(M-1)}$

Figure 1.4.: *Illustration of one epoch of neural network training. The samples of each batch are fed into the model* ψ *and its outputs are passed to the loss function. The gradient of the loss function with respect to the parameters of the model is computed and fed into an optimizer* op *that updates the parameters of the model. After all* M *batches are passed through the model, the model has been trained for one epoch.*

gradient, it is common to split the dataset into smaller batches of data. These batches are fed into the neural network and only their batch-gradient is used for updating the parameters of the model. The benefits of this approach are twofold. Firstly, the use of mini-batches increases the test set accuracy [205]. This has been empirically observed and is attributed to the stochasticity of the gradient information, which results in the optimizer not getting stuck in sharp minima [109] or saddle points [66]. Complementary to that observation, [206] show that the there exists a noise scale for stochastic optimizers, where the size of the mini-batches introduces different amounts of noise into the gradient. Similarly, this noise scale can also be tuned with the magnitude of the learning rate. The authors show that instead of reducing the learning rate, an increase in the mini-batch size can have similar effects. Secondly, the use of mini-batches requires significantly less computation per iteration and reduces the training time of deep neural networks on large datasets significantly [28]. Instead of computing the gradients of all the samples in the entire dataset before updating the parameters of the network, mini-batches only compute gradients of a few samples at once. The gradient information of a mini-batch will on average point in the same direction as the one computed using the entire dataset.

Each update using a single mini-batch is called an iteration during training. Once the whole training set has been passed through the network, it has been trained for one epoch. Usually neural networks are trained for several hundred epochs, which means that the entire training set has been fed through the network several hundred times.

Evaluation In order to be able to assess the performance of the neural network, one has to measure how close the predictions of the network are to the observed data. Given a training and a validation dataset, there is no guarantee that a low error on the training dataset will also result in a low error on the validation dataset. Additionally, in classical machine learning methods one has to be cautious of the number of parameters of the model compared to the number of samples in the training set. While the error of the training dataset gets progressively lower with a higher number of parameters, the validation error exhibits a U-curve shape. When the training set error is low and the validation set error is high, the model *overfits*. The U-shape of the validation set error is a result of the trade-off between bias and variance of the model [100]. Variance refers to the amount by which the method would differ when it is trained using a different set of training samples. Ideally, the method would always find a very similar model and thus exhibit almost no variance at all. More complex models with many parameters typically have higher variance. Bias on the other hand is a result of trying to fit a simple model to a complicated dataset. This can occur for example when trying to fit a linear model to a dataset where the features and the labels have a non-linear relationship (see Figure 1.1b for an example). No matter how many training samples, the simple model will never be able to accurately fit the data. This error is due to bias. Thus, variance and bias typically are seen as competing properties of machine learning methods [100].

Contrary to classical machine learning methods, deep learning models are almost always overparametrized [209, 250], that is the number of parameters in the network exceed the number of samples in the dataset. These neural networks are able to partly overcome the overfitting issue by early stopping [19], and so the network is able to fit to the data while reaching good accuracy on unseen datapoints [20].

The test set usually contains less samples than the training set and is used in order to prevent implicit overfitting on the validation set [100]. This can happen whenever different settings are tweaked during the training and validation of the model with the goal of increasing the validation set accuracy. In order to prevent this from misrepresenting the true accuracy of the model on unseen data, after several different training runs and after hyperparameter tuning on a certain model, the accuracy of the final trained model is assessed on the test set.

1.3. Deep Learning Models

Choosing an appropriate deep learning model is crucial for achieving a good performance. The right choice of layers can for example have an effect on the smoothness of the loss landscape and thus make training faster [131, 194, 241]. This section will take a closer look at some neural network architectures and discuss some of the main ideas behind them. Unless stated otherwise, this thesis will make the distinction between layers, which represent operations with trainable parameters and activation functions, which do not contain any trainable parameters themselves. When building an architecture, one has to carefully choose between different layers and activation functions, as well as how to connect these layers with each other.

The basic structure of most models is as follows:

$$\boldsymbol{h}^l = \mathcal{F}^l_{\boldsymbol{w}_l}(\boldsymbol{z}^l) \tag{1.3}$$

$$\boldsymbol{z}^{l+1} = \chi^l(\boldsymbol{h}^l) \tag{1.4}$$

where $\boldsymbol{z}^l$ represents the input to layer l (note that $\boldsymbol{z}^0 = \boldsymbol{x}$ with $\boldsymbol{x}$ the input sample) and $\boldsymbol{h}^l$ represents the output of layer l. The non-linear activation function at layer l is represented by χ^l. Also, $\mathcal{F}^l_{\boldsymbol{w}_l}(\boldsymbol{z}^l)$ represents the parametrized layer function at layer l. These parametrized functions are typically linear in their inputs. Thus, the construction of neural networks usually follows a nested structure of linear, parametrized layers followed by non-linear activation functions.

Layers The purpose of neural network layers is to learn some abstract features from the inputs. Because most layers are linear in their inputs, their outputs are passed through non-linear functions which enables them to learn more complex representations. The most common types of layers found in neural networks are fully-connected, convolutional and batch-norm layers.

Fully-connected layers take the n-dimensional input and transform it into a k-dimensional output through multiplication with a parameter matrix:

$$\boldsymbol{h} = W\boldsymbol{z} \tag{1.5}$$

where $\boldsymbol{z} \in \mathbb{R}^n$ is the input vector to this layer, $W \in \mathbb{R}^{k \times n}$ is the parameter matrix of the fully-connected layer and $\boldsymbol{h} \in \mathbb{R}^k$ is the output vector. A sketch of a fully-connected layer is shown in Figure 1.5.

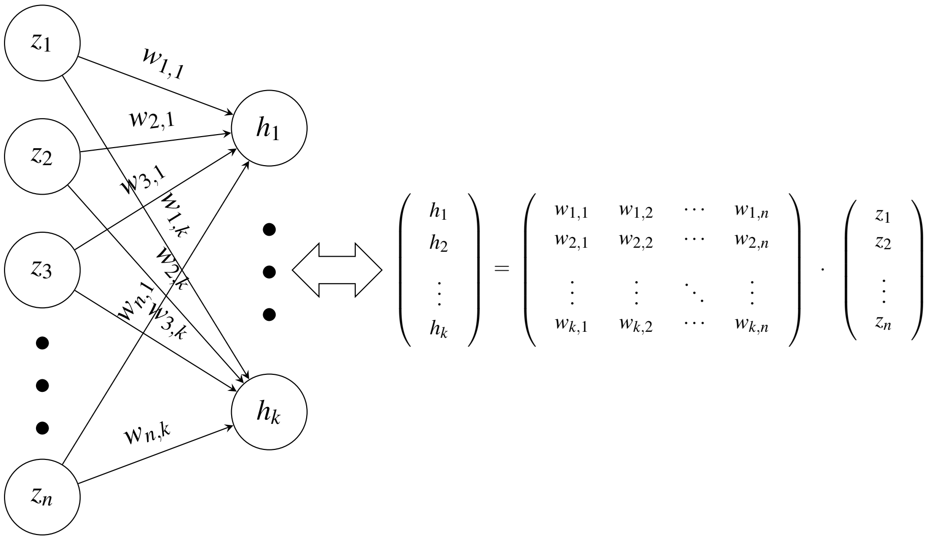

z_1
z_2
z_3
z_n
h_1
h_k
$w_{1,1}$
$w_{2,1}$
$w_{3,1}$
$w_{1,k}$
$w_{2,k}$
$w_{n,1}$
$w_{3,k}$
$w_{n,k}$
$\begin{pmatrix} h_1 \\ h_2 \\ \vdots \\ h_k \end{pmatrix} = \begin{pmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,n} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k,1} & w_{k,2} & \cdots & w_{k,n} \end{pmatrix} \cdot \begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{pmatrix}$

Figure 1.5.: *Illustration of a fully-connected layer. The left-hand side is a depiction of the fully-connected layer, which can be described through matrix multiplication as shown on the right-hand side.*

In this layer each output is connected to each input and each of those connections is

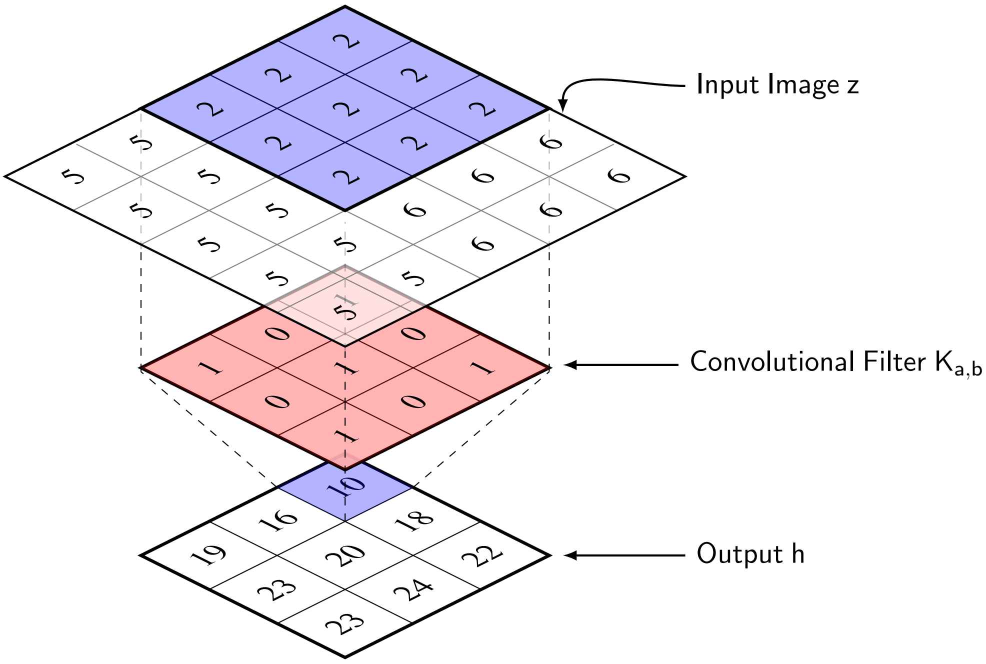


Figure 1.6.: *Illustration of a convolution for an input image $\boldsymbol{h}$ with one channel and output $\boldsymbol{z}$ with one channel as well. The convolutional filter $K_{a,b}$ is depicted in red and has one input channel dimension and one output channel dimension. The blue colored areas in the input and output depict how the convolutional filter is applied to different parts of the input image in order to produce the output.*

scaled by a parameter. Thus the total number of parameters in this layer is $n \times k$. This property makes fully connected layers very expensive. For datasets where the feature vectors are very large, like in images for example, using neural networks with only fully connected layers can lead to extremely large networks. Another problem with fully connected layers is that they require the input to always have the same fixed size. This can pose a problem for datasets where the samples vary in size. One solution to these problems, especially for image samples, is to use convolutions instead.

Convolutional layers are able to handle any input size, contrary to fully-connected layers. They are defined by a kernel-size and the number of input and output channels. There are also parameters like stride and padding, but for the sake of clarity these will be not covered in the description of convolution layers. The 2-dimensional convolution with a kernel of size $n \times m$ with P input channels and Q output channels for an input image $\boldsymbol{z}$ of size $s \times t$ with P channels is given by

$$h_{i,j,q} = \sum_{a=0}^{n-1} \sum_{b=0}^{m-1} \sum_{p=0}^{P-1} z_{i+a,j+b,p} K_{a,b,p,q} \tag{1.6}$$

where $\boldsymbol{h} \in \mathbb{R}^{(s-n+1)\times(t-m+1)\times Q}$ is the output, $\boldsymbol{z} \in \mathbb{R}^{s\times t\times P}$ is the input and $K \in \mathbb{R}^{m\times n\times P\times Q}$ are the trainable parameters of the convolution. An illustration of a convolution is depicted in Figure 1.6.

Convolutional layers have allowed neural networks to increase in size compared to neural networks that consist solely of fully-connected layers. The dimensions of the input samples also do not matter, except for the input channel dimension. Related to this is the fact that the number of parameters of the kernel in convolutions can be much smaller than the input image. This is also referred to as sparse interactions [76]. Since each parameter of the kernel processes different parts of the input image, this weight-sharing greatly increases the efficiency of convolutions compared to fully-connected layers [76, 252]. Also, weight-sharing causes the convolutional layer to be equivariant to translations of the input. This means that any change in translation of the input will result in an equal translation of the output [53, 111].

During training, different filters learn abstract features of the input image. The resulting outputs are passed to the next layer which repeats this process. This way neural networks are able to learn high-level features from the inputs. One important concept in convolutions in that of their receptive field. The receptive field size of a unit is the area of the input image that affects its value [208]. Fully-connected layers cover the entire input image, since each input is connected to each output. The resulting receptive field from convolutions on the other hand only covers part of the input image. One can increase the receptive field size by stacking multiple convolutional layers, thus making the network deeper. Another way is to downsample the image by using pooling operations [149].

Batch-norm layers were first introduced in [98] and the main idea behind this layer is to normalize the input signal. Empirical evidence suggests that neural networks that employ batch-norm layers have faster and more stable training [194]. Its effectiveness has been associated to the reduction of the difference between input distributions of different layers. Recently, [194] have disputed this idea and have shown that the reason batch-norm layers are so effective is that they make the optimization surface of the neural network smoother. Thus, they allow for a larger learning-rate while maintaining the same performance, which results in faster training times [194].

The batch-norm layer has trainable parameters that learn to transform the input in a way, such that the mean and variance of the distribution are zero and one respectively. For the i-th entry of an input feature vector the batch-norm is defined as:

$$h^{(i)} = \gamma^{(i)} \frac{z^{(i)} - \mu^{(i)}}{\sigma^{(i)}} + \beta^{(i)} \tag{1.7}$$

where $\gamma \in \mathbb{R}^n$ and $\beta \in \mathbb{R}^n$ are the trainable parameters of the layer, $\mu \in \mathbb{R}^n$ represents the mean of the incoming batch and $\sigma \in \mathbb{R}^n$ represents the standard deviation of the samples inside the batch. The trainable parameters γ and β make sure that the representation power of the neural network is not lost because of the resulting normalization of the input signal at each batch-norm layer [98]. The effect of normalizing the input features of a batch of samples is shown in Figure 1.7.

Activation Functions Most neural network layers are linear functions with respect to their inputs. Activation functions introduce nonlinearities into the neural network, which allows these networks to learn complex patterns in the dataset [200].

One of the most common activation functions used in deep neural networks is the *rectified linear unit* (ReLU) [160]. There are many advantages to ReLU activation functions, as listed in [247]. Using ReLU activation functions allows for successful training of deep neural networks, which was difficult to achieve with other activa-

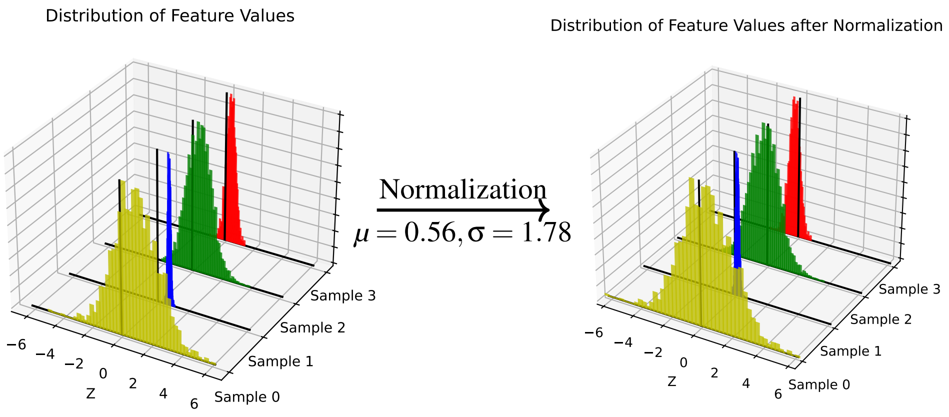
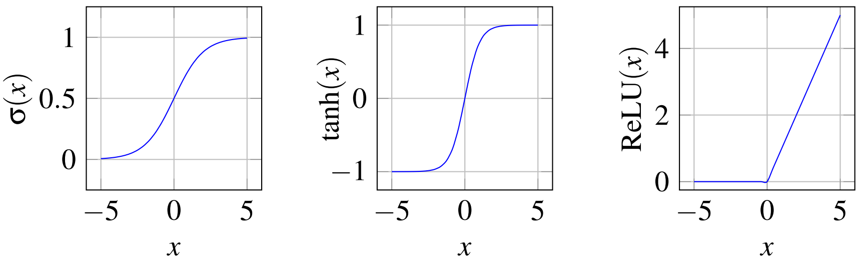


Figure 1.7.: *Depiction of the effects of the normalization step of the batch-normalization. In this example a batch consists of 4 samples. The distribution of their feature values that are passed into the batch-norm layer are shown on the left. The black lines indicate the x- and y-axis for each sample. The mean and standard deviation of the batch is $\mu = 0.56$ and $\sigma = 1.78$, respectively. One can see that after batch normalization the distributions are closer to that of a normal distribution with zero mean and unit variance. Note that in the batch-norm layer there are parameters that scale and shift the distributions again after this step and these are learned by the network during training.*

tion functions. Also, the use of ReLUs results in faster convergence of deep neural networks and it is faster to compute as well. Another advantage to this activation function is that it generalizes better to unseen data. The authors of [121] show that neural networks using only ReLU activations and the hinge loss decompose the optimization surface into different cells and as a result only have two types of local minima. Either the local minimum is flat, meaning the cell has a constant loss value or it is sharp, which means the minimum is non-differentiable.

Another common activation function is the sigmoid function [81]. This function takes the input and squishes it down into the interval between zero and one. The sigmoid function is commonly used after the last layer of the network, in order to turn the output of the neural network model into probabilities. The softmax function [27] is similar to the sigmoid function, but is instead used for multi-class predictions, whenever the output of the network is supposed to represent a probability distribution for the different classes present in the dataset. Three different activation functions are depicted in Figure 1.8 on the interval $x \in [-5, 5]$.

LeNet-5 The LeNet-5 architecture [122] is a seven-layer convolution network designed for handwritten digits recognition. The image samples have a size of 28×28. The network is built in a way, such that the centers of the receptive field of the final convolutional layer form a 20×20 square area. As stated by the authors, the largest character in the dataset covers an area of 20×20 in the center of the image, thus this makes sure that the network is able to cover the corners of the digit. The inputs are passed through two blocks that consist of a convolution followed by a pooling layer. After each convolution and fully-connected layer there is a tanh activation function. After a third convolution layer the output is a one-dimensional vector of 120 units.



(a) *Plot of the sigmoid function.* **(b)** *Plot of the tanh function.* **(c)** *Plot of the ReLU function.*

Figure 1.8.: *Visualization of three activation functions on the interval $x \in [-5, 5]$. Plot (a) depicts the sigmoid function, plot (b) the tanh function and plot (c) the ReLU function.*

This is passed through two fully-connected layers which shrink the output down to 10 dimensions. Finally, this output is passed through a softmax activation function, which returns a distribution over the 10 class labels. An illustration of the LeNet-5 architecture is shown in Figure 1.9.

AlexNet With the introduction of larger datasets with more classes and high-resolution images (e.g. ImageNet), neural network architectures like LeNet-5 are not suitable anymore. The AlexNet architecture [115] was designed to perform well for these types of datasets and outperformed the competition on ImageNet at that time. Some key modifications made to AlexNet are the introduction of ReLU activation functions. These are used after each convolutional and fully-connected layer. The only exception is the very last fully-connected layer, which passes its outputs through a softmax activation function in order to obtain a distribution over the class labels. This network also used dropout [213] in order to prevent it from overfitting. Dropout refers to the technique, where each parameter has a certain probability of being set to zero during the forward propagation. This essentially samples a different architecture each time and thus forces the network to learn more robust features. AlexNet was also one of the earliest deep learning models to use GPUs for its computations and it also employed model-parallelism by placing parts of its model onto different GPUs during training.

ResNet Many deep learning models suffer from the vanishing gradient problem [91]. This issue arises commonly in deep neural networks, and is due to many stacked layers that each contain an activation function. The ResNet architecture [83] is built by stacking the same cell structure and using skip-connections in order to skip over a certain cell. These connections help combat the vanishing gradient problem and it has been shown that they can make the optimization surface smoother [131], which makes it easier to train the network. An illustration of skip-connections is depicted in Figure 1.10.

There are many other model and layer types which have not been covered by this section, such as generative adversarial networks [72], autoencoders [16, 191], recurrent neural networks [104, 191], transformers [226] and diffusion models [207, 210].

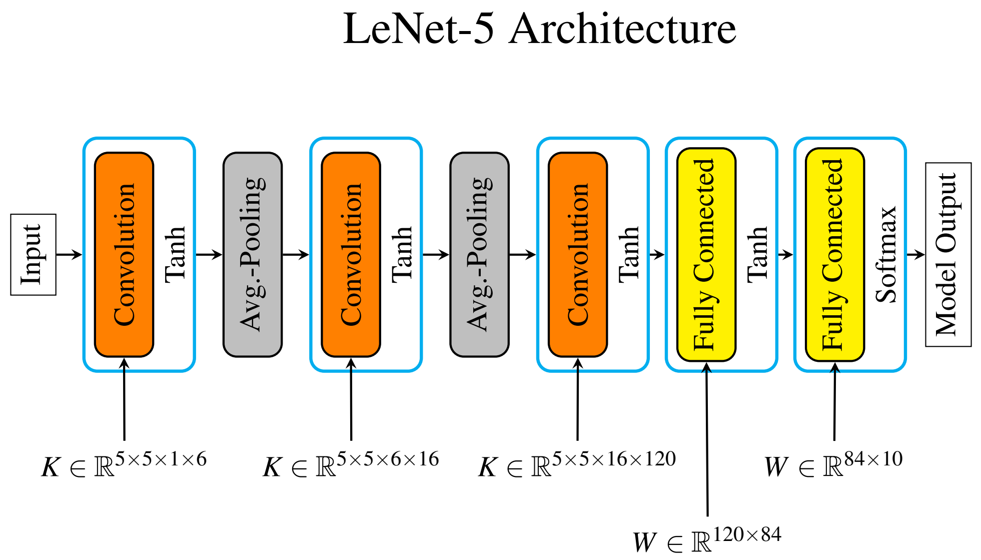

LeNet-5 Architecture
Input
Convolution
Tanh
Avg.-Pooling
Convolution
Tanh
Avg.-Pooling
Convolution
Tanh
Fully Connected
Tanh
Fully Connected
Softmax
Model Output
$K \in \mathbb{R}^{5\times5\times1\times6}$
$K \in \mathbb{R}^{5\times5\times6\times16}$
$K \in \mathbb{R}^{5\times5\times16\times120}$
$W \in \mathbb{R}^{84\times10}$
$W \in \mathbb{R}^{120\times84}$

Figure 1.9.: *Illustration of the LeNet-5 architecture. The input images have dimension 28×28 and are padded in order to reach a size of 32×32. The first convolution layer has a kernel of size 5×5 with one input channel and six output channels. Each convolutional and fully-connected layer passes their output through a non-linear activation function. The second layer described in [122] is a subsampling layer that is very similar to the average-pooling layer [123]. This pattern repeats until the fifth layer where the outputs are fed into a fully-connected layer. The very last layer passes the outputs through a softmax function.*

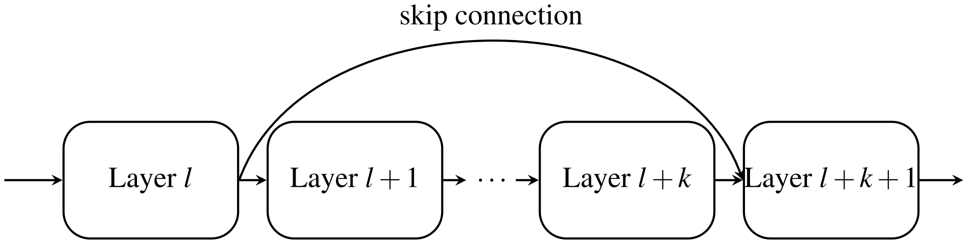


Figure 1.10.: *Depiction of skip connections used in the ResNet architecture. In this figure the output of layer l is passed into two different layers, namely layer $l+1$ and layer $l+k+1$. Before passing into layer $l+k+1$, the output of layer l is added to the output of layer $l+k$. Note that both, the output of layer l, as well as the output of layer $l+k$ have to have the same size in order to add both together. Some works [95, 218, 219] have concatenated the outputs of these two layers instead of adding them. This can help with the information of lower layers flowing through the network [95]*

1.4. Datasets

The use of standardized datasets in the field of deep learning serves as a benchmark for different tasks. They are important for comparing different approaches to each other. Throughout this thesis, there will be references to different datasets, so this sections serves as a short overview of those.

MNIST The *Modified National Institute of Standards and Technology* (MNIST) dataset [124] was introduced in 1998 and is used for handwritten digit recognition. It is derived from the larger NIST Special Database [78], contains 10 different classes (the numbers 0 to 9) and the dataset consists of 50000 samples where each sample is greyscale and has dimension of 28×28. A selection of some samples is shown in Figure 1.11 (a).

The *Extended MNIST* (EMNIST) dataset [45] is an extension of the MNIST dataset that includes handwritten characters. Similar to MNIST, the images are greyscale and have dimension 28×28. In its balanced form it contains 47 classes (10 digits and 37 letters), with 112800 samples in the training dataset and 18800 samples in the test dataset. Five randomly chosen samples of the balanced dataset can be seen in Figure 1.11 (b).

The *Kuzushiji-MNIST* (KMNIST) dataset [44] aims to be a benchmark that is more applicable to real world problems. It depicts cursive Japanese (Kuzushiji) and the full dataset contains 3999 character types and 403242 characters. The Kuzushiji-MNIST is a subset that contains 10 different classes with 60000 samples in the training dataset and 10000 in the test dataset. There are bigger Kuzushiji datasets (Kuzushiji-49 and Kuzushiji-Kanji), though they are unbalanced and not of interest in this thesis. A set of randomly chosen samples of the dataset are depicted in Figure 1.11 (c).

CIFAR The CIFAR-10 and CIFAR-100 datasets [114] consist of 60000 images of different objects and animals, which are of size 32×32 with three color channels. The CIFAR-10 dataset has 10 different classes while the CIFAR-100 has 100 different classes. Both datasets are split into 50000 samples in the training set and 10000 samples in the test set. The CIFAR-100 dataset is more challenging and consists of 600 images per class. These 100 classes can also be classified into 10 superclasses. A selection of five random samples of the CIFAR-10 dataset is shown in Figure 1.11 (d).

ImageNet The ImageNet dataset [52] consists of around 15 million samples which are classified into roughly 22000 different classes. All images have been collected from the web and classified by humans. The *ImageNet Large-Scale Visual Recognition Challenge* (ILSVRC) dataset was introduced in 2010 and serves as a more challenging dataset to CIFAR-10 and CIFAR-100. This dataset is a subset of ImageNet and consists of around 1.2 million training samples, 50000 validation images and 150000 testing images [1]. The images are grouped into 1000 different classes. The samples of the dataset have three color channels and vary in their resolution, thus they are commonly down-sampled and cropped, in order for all of them to reach the same dimension of 256×256. Because of its difficulty, it is common to report the top-1 as well as the top-5 accuracy on this dataset. The top-5 accuracy counts the classification of the network as valid if it correctly classifies the sample in its top-5 picks. Five samples of this dataset are depicted in Figure 1.11 (e).

Due to the resolution of the images and the size of the dataset, training on ImageNet is prohibitive. On the flip side, the performance of neural networks has increased to a point where classification on the CIFAR datasets is not as challenging anymore. In order to find a solution to these problems, [41] introduced downsampled variants of the ImageNet dataset, where the images have for example been downsampled to 16×16 to form ImageNet16x16. Using downsampling, the authors construct the datasets ImageNet16x16, ImageNet32x32 and ImageNet64x64.

In order to shrink the dataset size down and also reduce the number of classes, [57] introduced the ImageNet-16-120 dataset, which takes the ImageNet16x16 dataset and only uses samples from the first 120 classes. This reduces the dataset down to 151.7×10^3 samples in the training dataset, 3000 samples in the validation dataset and 3000 samples in the test dataset.

---

[1] Note that in many works this ILSVRC dataset is often times just referred to as ImageNet.

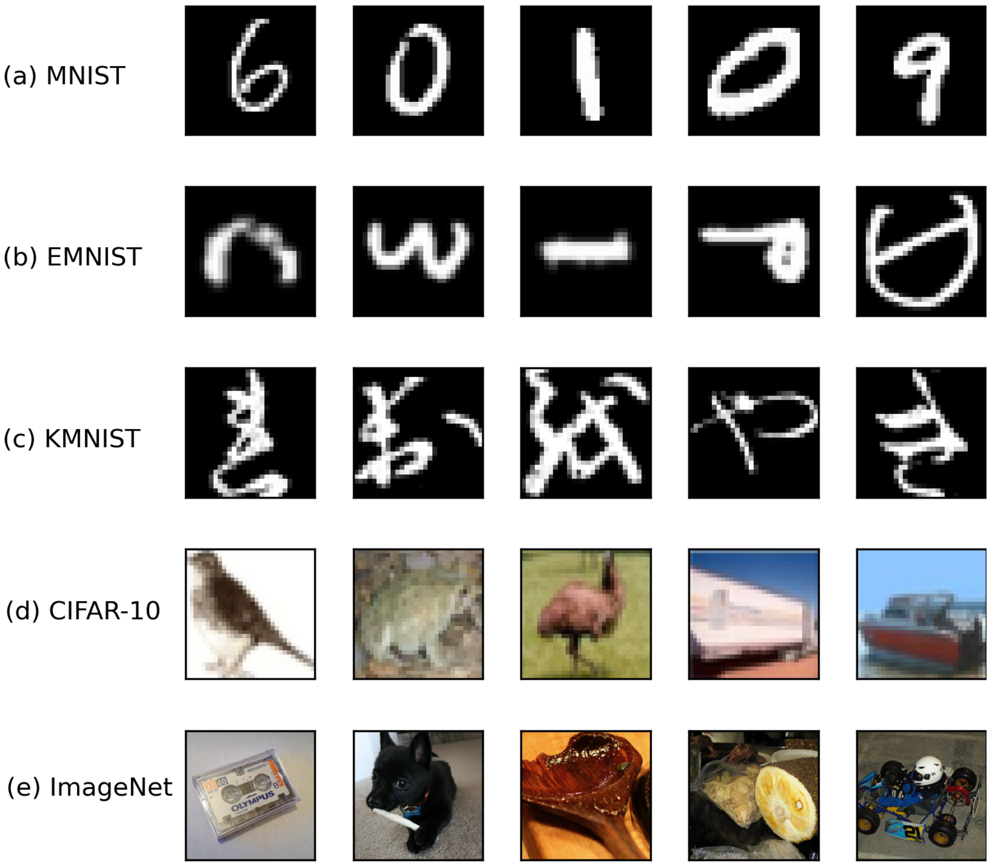


Figure 1.11.: *Visualization of samples from five different datasets. (a) shows five samples of the MNIST dataset. (b) shows five samples of the EMNIST dataset while (c) depicts five samples of the KMNIST dataset. (d) depicts five samples of the CIFAR-10 dataset and (e) shows five samples of the ImageNet dataset.*

Chapter 2

Methods

This chapter will introduce some key concepts and mathematical formulations in deep learning that will also become important in later chapters. First, a general overview over different optimization methods for deterministic gradients will be given, with special emphasis on first- and second-order optimization. This will be expanded to also cover stochastic gradient settings, which are crucial in deep learning. Next, regularization is introduced, where the ℓ_1- and $\ell_{2,1}$-regularization will become especially important for dealing with large neural networks in later chapters. The last part of this chapter will first introduce the Hessian as well as the R-operator, which will later be used in combination with the Lanczos algorithm for efficient eigenvalue computation. The Lanczos and stochastic Lanczos quadrature algorithms are introduced as an efficient method for eigenvalue computation of very large matrices. All of those concepts and methods will come up again in the next chapter, which will use eigenvalue computations for investigating the loss landscapes and trajectories of neural networks and their optimizers. Since very large neural networks have too many dimensions to compute eigenvalues in reasonable time, even with these efficient Lanczos methods, this chapter will also introduce parallel versions of these methods in order to speed up computation.

2.1. Optimization Methods in Deep Learning

The general goal in neural network training boils down to non-linear optimization, which aims to find solutions to problems of the form:

$$\begin{aligned}&\text{find } \boldsymbol{w}^\star \\ &\text{such that } \boldsymbol{w}^\star \in \mathbb{W} \text{ and } \boldsymbol{w}^\star = \inf_{\boldsymbol{w}\in\mathbb{W}} \mathcal{L}(\boldsymbol{w})\end{aligned}$$

where the set $\mathbb{W} \subset \mathbb{R}^n$ in constrained optimization or $\mathbb{W} = \mathbb{R}^n$ in the unconstrained case [24]. The cost or error function $\mathcal{L}(\boldsymbol{w})$ is a scalar function and is a measure of how high the error is for choosing the point $\boldsymbol{w}$. In this thesis, the cost function without regularization terms will be denoted by $f(\boldsymbol{w})$. In the presence of regularization terms the combination of $f(\boldsymbol{w})$ with regularization will be denoted by $\mathcal{L}(\boldsymbol{w})$.

The non-convex nature of the optimization problem makes it NP-hard to find the global minimum of the optimization surface [90]. Thus, the goal is typically not to try and find the global minimum, but a suitable local minimum instead. A sketch of some different types of minima that can be found in non-convex optimization is shown in Figure 2.1.

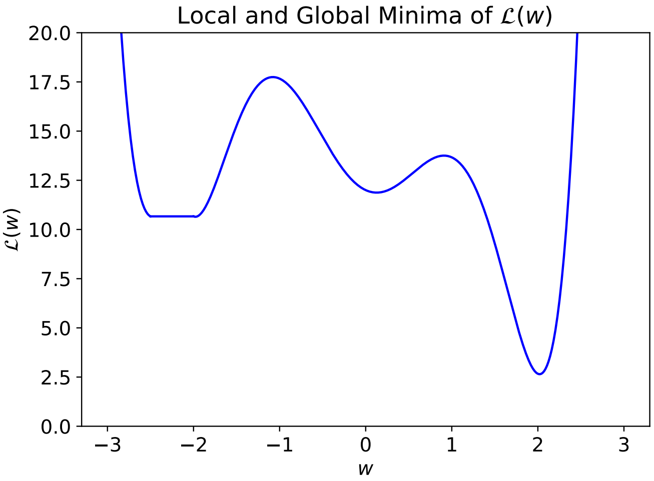


Figure 2.1.: *Illustrative example of a loss function with a global minimum at $w = 2$, a strict local minimum at $w = 0$ and a local minimum at $w \in [-2.5, 2]$.*

In deep learning, the error function is typically a composition of a convex objective function $g(\boldsymbol{x})$ and a non-convex neural network $\boldsymbol{\psi}(\boldsymbol{w})$, which turns the optimization problem for the objective function $f(\boldsymbol{w}) = (g \circ \boldsymbol{\psi})(\boldsymbol{w})$ non-convex.

This thesis will focus on iterative optimization methods, such as gradient descent. Other methods for optimization not covered in the following section include derivative free optimization [47, 112] and non-iterative optimization methods [196, 229]. Most gradient methods rely on the idea of iterative descent, which starts at some initial guess $\boldsymbol{w}^{(0)}$ and iteratively decreases the function $f(\boldsymbol{w})$ at every iteration [24]:

$$f(\boldsymbol{w}^{(k+1)}) < f(\boldsymbol{w}^{(k)}). \tag{2.1}$$

At every iteration k a new direction $\boldsymbol{d}^{(k)} \in \mathbb{R}^n$ is chosen and the point at the next iteration is given by

$$\boldsymbol{w}^{(k+1)} = \boldsymbol{w}^{(k)} + \varepsilon_k \boldsymbol{d}^{(k)} \tag{2.2}$$

with the stepsize or learning rate ε_k, which scales the update direction in order to prevent the algorithm from overshooting the minimum.

Using Taylor's theorem, the cost function $f(\boldsymbol{w})$ is approximated to first order

$$f(\boldsymbol{w}^{(k+1)}) = f(\boldsymbol{w}^{(k)}) + \varepsilon_k \boldsymbol{\nabla} f(\boldsymbol{w}^{(k)})^T \boldsymbol{d}^{(k)} + o(\varepsilon_k). \tag{2.3}$$

It can be seen from Equation (2.3), that (2.1) is satisfied whenever the direction $\boldsymbol{d}^{(k)}$ forms an angle greater than 90° with the gradient, such that

$$\nabla f(\boldsymbol{w})^T \boldsymbol{d} < 0. \tag{2.4}$$

Equation (2.3) also shows that $\varepsilon_k \nabla f(\boldsymbol{w}^{(k)})^T \boldsymbol{d}^{(k)}$ dominates the $o(\varepsilon_k)$ term for small ε_k. Many descent methods have the following general form [24]

$$\boldsymbol{w}^{(k+1)} = \boldsymbol{w}^{(k)} - \varepsilon_k D_k \nabla f(\boldsymbol{w}^{(k)}) \tag{2.5}$$

with the preconditioning matrix $D_k \in \mathbb{R}^{N \times N}$ and the stepsize $\varepsilon_k \in \mathbb{R}^+$. In order for the descent direction to satisfy Equation (2.4), it must hold that

$$\nabla f(\boldsymbol{w}^{(k)})^T D_k \nabla f(\boldsymbol{w}^{(k)}) > 0 \tag{2.6}$$

which is satisfied whenever the matrix D_k is positive definite.

Convergence Properties In this paragraph the rate of convergence of gradient methods is shown using local analysis, that is in a neighborhood of a local solution. In this neighborhood the loss function can be accurately described by a quadratic function [24]. This fact can be shown by using a Taylor approximation around an optimal solution $\boldsymbol{w}^\star$:

$$f(\boldsymbol{w}) = f(\boldsymbol{w}^\star) + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}^\star)^T \nabla^2 f(\boldsymbol{w}^\star)(\boldsymbol{w} - \boldsymbol{w}^\star) + o(\|\boldsymbol{w} - \boldsymbol{w}^\star\|^2) \tag{2.7}$$

Because local analysis assumes that $\boldsymbol{w}$ is close to an optimal solution $\boldsymbol{w}^\star$, the $o(\|\boldsymbol{w} - \boldsymbol{w}^\star\|^2)$ term will be much smaller than the other terms. Thus, near the optimal solution the cost function can be reasonably approximated by a quadratic function.

For a quadratic function defined as

$$f(\boldsymbol{w}) = \frac{1}{2}\boldsymbol{w}^T Q \boldsymbol{w} \tag{2.8}$$

where the positive definite matrix $Q \in \mathbb{R}^{N \times N} (Q = \nabla^2 f(\boldsymbol{w}^\star))$ has a biggest eigenvalue M and a smallest eigenvalue m, the convergence rate for steepest descent ($D_k = I$ in Equation (2.5)) is given as [24]

$$\frac{\|\boldsymbol{w}^{(k+1)} - \boldsymbol{w}^\star\|}{\|\boldsymbol{w}^{(k)} - \boldsymbol{w}^\star\|} \leq \frac{M - m}{M + m}. \tag{2.9}$$

Also, when the stepsize is chosen according to a line minimization rule, it can be shown that the cost function decreases as [24]

$$\frac{f(\boldsymbol{w}^{(k+1)})}{f(\boldsymbol{w}^{(k)})} \leq \left(\frac{M - m}{M + m}\right)^2. \tag{2.10}$$

These results show that the condition number, defined by the ratio M/m, is important for the convergence of steepest descent methods. A high condition number results in ill-conditioned problems with slow convergence. Ideally, the condition number is equal to 1.

In the case of the general form in Equation (2.5), the matrix D_k transforms the parameters and thus each iteration can be seen as the regular steepest descent algorithm applied to a different coordinate system. By assigning $\boldsymbol{w}^{(k)} = D_k^{1/2}\boldsymbol{u}^{(k)}$, and $f(D_k^{1/2}\boldsymbol{u}) = \mathfrak{h}(\boldsymbol{u})$, the problem of minimizing $\mathfrak{h}(\boldsymbol{u})$ becomes:

$$\boldsymbol{u}^{(k+1)} = \boldsymbol{u}^{(k)} - \varepsilon_k \boldsymbol{\nabla} \mathfrak{h}(\boldsymbol{u}^{(k)}). \tag{2.11}$$

This is just the steepest descent method that was introduced in Equation (2.5) with $D_k = I$. When multiplied from the left by $D_k^{1/2}$, Equation (2.11) becomes

$$\boldsymbol{w}^{(k+1)} = \boldsymbol{w}^{(k)} - \varepsilon_k D_k^{1/2} \boldsymbol{\nabla} \mathfrak{h}(\boldsymbol{u}^{(k)}). \tag{2.12}$$

Here, the relation $\boldsymbol{w}^{(k)} = D_k^{1/2}\boldsymbol{u}^{(k)}$ was used. Using the fact that $\boldsymbol{\nabla}\mathfrak{h}(\boldsymbol{y}^{(k)}) = D_k^{1/2}\boldsymbol{\nabla} f(\boldsymbol{w}^{(k)})$, this turns Equation (2.11) back to Equation (2.5).

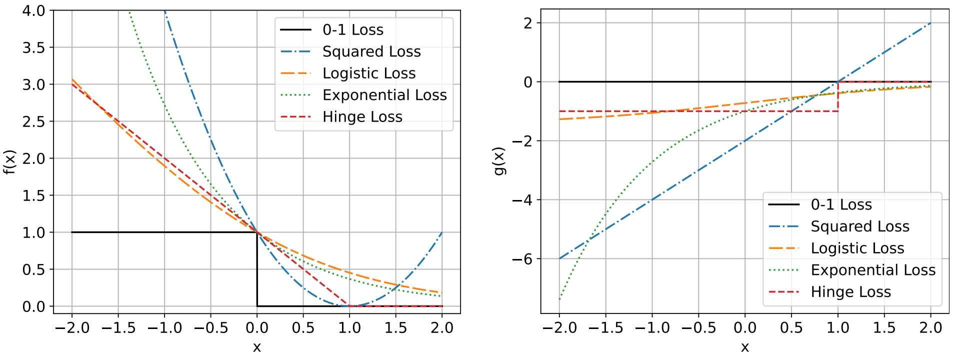
In order to find the rate of convergence of the general descent method from Equation (2.5) for a quadratic function using local analysis, instead of looking at the biggest and smallest eigenvalue of the matrix Q, now the biggest and smallest eigenvalues of the matrix $(D_k)^{\frac{1}{2}}Q(D_k)^{\frac{1}{2}}$ are relevant for convergence. This shows that choosing D_k close to the inverse of Q will result in a condition number that is close to one and thus result in very fast convergence. Similar results can be shown for the case where the neighborhood of the cost function can not be approximated by a quadratic function [24]. Even though local analysis does not say anything about how a method behaves far away from a local solution, in practice the descent method will quickly make progress in the early iterations and then slow down near the solution [24].

Cost Functions The cost function is a measure of the cost that the network incurs for an incorrect decision. For example, in classification tasks the goal of the network is to classify objects into one of many different categories. In this case the ideal cost function is the 0-1 cost function J, which is given by

$$J(i,j) = \begin{cases} 0 & \text{if } i = j \\ 1 & \text{if } i \neq j \end{cases}$$

for prediction i and label j. This cost function is non-convex and discontinuous, and in order to find an optimal solution in the binary case one has to find parameters $\boldsymbol{w}$ which minimize $I(y_i f(\boldsymbol{w}, \boldsymbol{x}_i) \leq 0)$ for each sample $\boldsymbol{x}_i$, where I represents the indicator function. Finding the solution is thus exponential in the number of inputs and therefore intractable to optimize, as there is no way of figuring out how to change the parameters in order to minimize the cost function.

Therefore, in cases where the 0-1 cost function would be ideal, it is approximated with more suitable functions, for example the hinge, logistic or exponential cost function. Figure 2.2 summarizes common cost functions as well as the 0-1 cost function. The plot on the right hand side shows the gradient of each method. While the gradients of other cost functions give a sense of direction for updating the parameters, the gradient of the 0-1 cost function is flat and thus contains no information. Commonly used cost functions in classification include the mean squared error loss as well as the cross-entropy loss.



(a) *Plot of different loss functions. Most loss functions try to approximate the 0-1 loss while providing a gradient in the direction of correct classification.*

(b) *The gradients show how misclassified samples push the input toward the correct classification. It also shows that the gradient of the 0-1 loss is zero everywhere and undefined at the point $x = 0$.*

Figure 2.2.: *Visualization of the 0-1 loss along with the squared loss, the logistic loss, the exponential loss and the hinge loss. Plot (a) shows the respective loss functions plotted on the interval $x \in [-2,2]$, while plot (b) shows the gradient of each loss function.*

2.1.1. First-Order Optimization

First-order optimization refers to the fact that the only information used for updating the model are first-order derivatives with respect to the parameters of the model.

One of the simplest first-order methods is steepest descent, where the matrix D_k of Equation (2.5) is set to the identity matrix $D_k = I$, which results in the following iterative method:

$$\boldsymbol{w}^{(k+1)} = \boldsymbol{w}^{(k)} - \varepsilon_k \nabla f(\boldsymbol{w}^{(k)}) \tag{2.13}$$

This is one of the simplest methods, but in theory also one of the slowest to converge. One issue that steepest descent is facing is loss landscapes that are flat in one direction and steep in the other. In this case the steepest descent method will zig-zag slowly toward the minimum [24].

Momentum In order to speed up the convergence of first-order methods, there exist methods that employ a momentum term, sometimes also referred to as ”heavy-ball” methods. The momentum method [179] accumulates gradients across iterations into a velocity vector $\boldsymbol{v}$. For steepest descent this results in the following update method:

$$\boldsymbol{v}^{(k+1)} = \rho_k \boldsymbol{v}^{(k)} - \varepsilon_k \nabla f(\boldsymbol{w}^{(k)}) \tag{2.14}$$

$$\boldsymbol{w}^{(k+1)} = \boldsymbol{w}^{(k)} + \boldsymbol{v}^{(k+1)} \tag{2.15}$$

where $\rho_k \in [0,1]$ is the momentum parameter. A value of $\rho = 0$ recovers the steepest descent method, while higher values of ρ_k influence how strongly past gradient values influence the direction of the current iteration. [179] shows that using momentum can accelerate convergence to a local minimum in the case of deterministic gradients.

Another type of momentum method is Nesterov's accelerated gradient method [162], which updates the parameter vector iteratively as:

$$\boldsymbol{v}^{(k+1)} = \rho_k \boldsymbol{v}^{(k)} - \varepsilon_k \nabla f(\boldsymbol{w}^{(k)} + \boldsymbol{v}^{(k)}) \tag{2.16}$$

$$\boldsymbol{w}^{(k+1)} = \boldsymbol{w}^{(k)} + \boldsymbol{v}^{(k+1)} \tag{2.17}$$

The authors of [217] argue that Nesterov's accelerated gradient method is more stable than the classical momentum method in many situations, because Nesterov's accelerated gradient method allows the velocity vector $\boldsymbol{v}$ to change more quickly during training.

While local analysis shows faster convergence rates for momentum methods compared to pure steepest descent methods, these are only achieved in the setting with deterministic gradients. Moving to a setting where stochastic gradients are used, [169, 232] show that the benefits from using momentum vanish when close to a local minimum. Nevertheless, [217] detail how these methods can still result in faster convergence when farther away from any local minimum. This transient phase seems to be much more important in deep neural network optimization [50].

Deep Learning Optimizers Up until now the theory covered in this thesis considered deterministic gradients, where the entire dataset is used when computing the gradient of the objective function. Optimization of deep neural networks usually involves stochastic gradients, which are computed using mini-batches instead of the entire dataset during one iteration. By going from deterministic to stochastic gradients, many of the convergence properties for the different methods change as well.

Most of the optimizers commonly used in deep learning are first-order methods. The simplest is *Stochastic Gradient Descent* (SGD) [186] and SGD with momentum. This optimizer is essentially the same as steepest descent, which has been described in this section and the stochastic part in its name refers to the fact that the network is optimized while considering only a subset of the samples of the total dataset in each iteration. This can be viewed as steepest descent where the gradient is offset by an additional error term [24].

Many optimizers that have become popular in recent years have been first-order methods that are curvature adaptive. This refers to the fact that they take into account knowledge of the geometry of previous iterations for their current update step. The momentum methods that were introduced previously are one example of such curvature adaptive algorithms. Other examples include Adagrad [58], AdaDelta [246], RMSprop [223] and Adam [110]. All of these methods belong to adaptive gradient based momentum algorithms, which are first-order gradient methods that can adapt their implicit learning rates and can also include some form of momentum. The convergence rate of these first-order gradient methods for non-convex objective functions was only recently shown to be $O(\log T/\sqrt{T})$ [38].

The main idea in Adagrad for example is to question whether all features in the data should have the same learning rate. Infrequent features in the data are often much more informative, while many commonly occurring ones are almost irrelevant

[58]. In order to achieve this adaptive learning rate, the authors find that they have to precondition the gradient with a matrix that accumulates the outer product of past gradients:

$$G_k = \left(\sum_{\tau=1}^{k} \boldsymbol{g}^{(\tau)} \left(\boldsymbol{g}^{(\tau)} \right)^T \right) \tag{2.18}$$

Optimizing the parameters then results in

$$\boldsymbol{w}^{(k+1)} = \boldsymbol{w}^{(k)} - \varepsilon_k G_k^{-1/2} \boldsymbol{g}^{(k)}. \tag{2.19}$$

One issue that Adagrad faces is that the implicit learning rate is rapidly decaying. In order to solve this issue, other adaptive optimizers like RMSprop or Adam use *exponential moving averages* (EMA) in order to achieve an adaptive learning rate, where the matrix G_k is now given as

$$\tilde{G}_k = (1-\beta) \left(\sum_{\tau=1}^{k} \beta^{k-\tau} \boldsymbol{g}^{(\tau)} \left(\boldsymbol{g}^{(\tau)} \right)^T \right). \tag{2.20}$$

These EMA algorithms do not suffer from rapidly decaying learning rates, and for suitable values of the hyperparameter β the matrix $\tilde{G}_k \approx \mathbb{E}[\boldsymbol{g}^{(k)} \left(\boldsymbol{g}^{(k)} \right)^T] = G_k$ [214].

In practice, computing the outer product of the gradients is infeasible due to the size of the resulting matrix, which scales quadratically with the number of trainable parameters in the neural network. Therefore, all those methods approximate G_k by only computing its diagonal entries $\operatorname{diag}(G_k)$. For Adagrad, this results in the following updates:

$$\boldsymbol{w}^{(k+1)} = \boldsymbol{w}^{(k)} - \frac{\varepsilon_k}{\sqrt{\sum_{\tau=1}^{k} \boldsymbol{g}^{(\tau)} \odot \boldsymbol{g}^{(\tau)}}} \boldsymbol{g}^{(k)} \tag{2.21}$$

where $\boldsymbol{g} \odot \boldsymbol{g}$ denotes the Hadamard product between two vectors.

Note that in the Adam paper, the authors claim that their optimizer approximates the Fisher matrix by only taking the diagonal elements of the matrix $\tilde{G}_k$ into account, in order to precondition the gradient of the cost function [110]. The Fisher matrix is primarily used in natural gradient methods [6], that try to optimize the parameters of the network while also restricting the output distribution of the model from changing too drastically between iterations.

The authors of [214] argue that this reasoning in the Adam paper is wrong. Firstly, the empirical Fisher information matrix mentioned in the Adam paper differs from the true Fisher information matrix and the connection to G_k is only approximately true near an optimum. Secondly, natural gradient descent methods use the inverse of the Fisher information matrix, while Adam uses the inverse of the *square-root* of matrix G_k. This issue is not exclusive to Adam and many of these adaptive gradient based algorithms have to use the inverse of the square-root of the matrix G_k, because the inverse itself has been reported to result in unstable algorithms [190]. Furthermore, there exist many misconceptions in literature regarding the Fisher matrix. It is often argued that the empirical Fisher information matrix can be viewed as a generalized Gauss-Newton matrix, which approximates the Hessian near an optimum. Another argument is that the empirical Fisher information matrix converges to the true Fisher information matrix near an optimum. Both those arguments only hold in very special circumstances that are rarely satisfied in practice [118].

The authors of [17] give some insight into why these curvature adaptive first-order methods are so successful in practice. In their paper, they are able to show that one property of adaptive gradient based algorithms is that they equalize the noise present in the stochastic gradients in each direction. This is helpful in nonconvex problems, because at stationary points this gradient noise is approximately isotropic, which helps the network escape saddle points with high probability [214].

2.1.2. Second-Order Optimization

Second-order optimization expands the Taylor approximation of the objective function $f(\boldsymbol{w})$ up to second-order [24]

$$\begin{aligned} f(\boldsymbol{w}) =& f(\boldsymbol{w}^{(k)}) + (\boldsymbol{w}-\boldsymbol{w}^{(k)})^T \nabla f(\boldsymbol{w}^{(k)}) + \\ & \frac{1}{2}(\boldsymbol{w}-\boldsymbol{w}^{(k)})^T \nabla^2 f(\boldsymbol{w}^{(k)})(\boldsymbol{w}-\boldsymbol{w}^{(k)}) + o(\left\|\boldsymbol{w}-\boldsymbol{w}^{(k)}\right\|^2). \end{aligned} \tag{2.22}$$

Minimizing this expression results in the following iterative update scheme

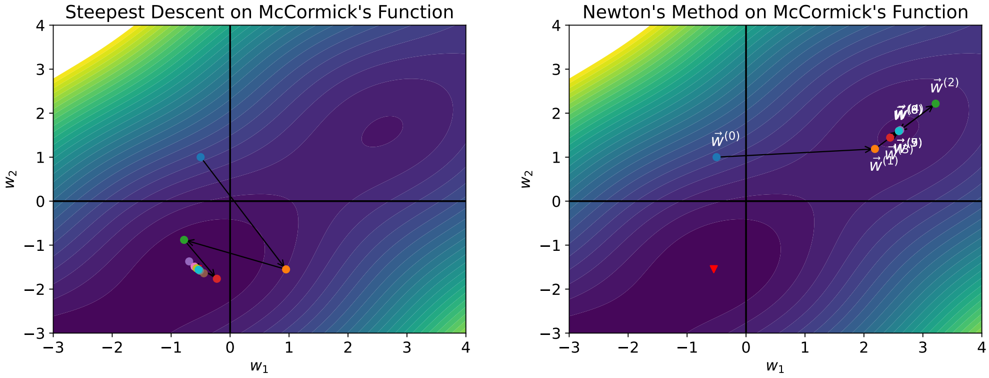
$$\boldsymbol{w}^{(k+1)} = \boldsymbol{w}^{(k)} - \varepsilon_k \frac{1}{\nabla^2 f(\boldsymbol{w}^{(k)})} \nabla f(\boldsymbol{w}^{(k)}). \tag{2.23}$$

This is also called Newton's method. Using $D_k = (\nabla^2 f(\boldsymbol{w}^{(k)}))^{-1}$, Newton's method can be rewritten in the general form introduced in Equation (2.5). It does not suffer from elongated minima the way that steepest descent does for example. This is because in Newton's method the gradient is scaled optimally (the eigenvalues of $(D_k)^{\frac{1}{2}} Q (D_k)^{\frac{1}{2}}$ are $M \approx 1$ and $m \approx 1$ for $D_k = (\nabla^2 f(\boldsymbol{w}^{(k)}))^{-1}$ [24].

The second-order derivative of the loss function measures the curvature at a specific point. If the second-order derivative is zero, there is no curvature present and the first-order derivative is sufficient in order to move optimally along the descent direction. If the curvature is positive or negative, the first-order descent will result in a lower or higher loss than expected.

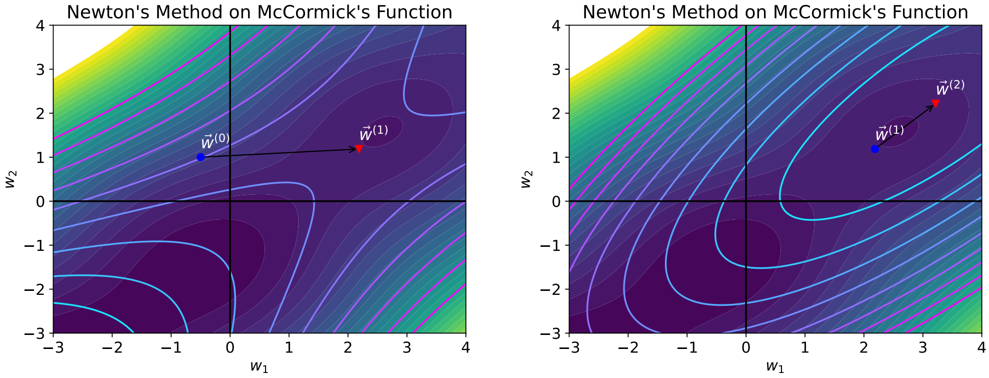
An example that illustrates Newton's method is shown in Figure 2.3. Here, both steepest descent and Newton's method are initialized at the same point. The objective function is non-convex and has a global as well as a local minimum. The upper left plot depicts steepest descent. This method is able to converge into the narrow valley that contains the global minimum. Inside this valley the steepest descent method starts to oscillate, which slows down its convergence toward the minimum. Newton's method on the other hand converges into the local minimum, where it is able to converge to the minimum after 4 iterations. The contour lines in the plots 2.3c-2.3f depict the optimization of the local approximation problem that is solved by Newton's method at each iteration.

Convergence It can be shown through local analysis close to a minimum, where the Hessian is positive definite, that Newton's method converges superlinearly [119].



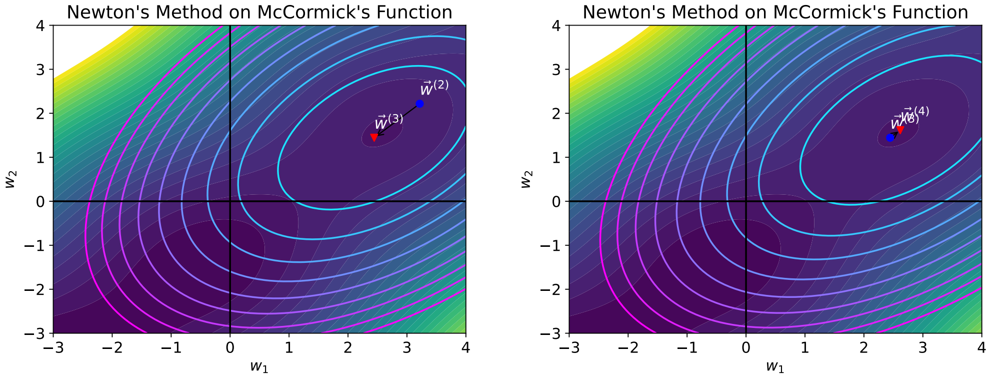
(a) *Steepest descent on McCormick's function with learning rate* $\varepsilon = 0.4$. *Note how steepest descent slows down inside the narrow valley.*

(b) *Newton's method on McCormick's function.*



(c) *Newton's method on McCormick's function in the first iteration.*

(d) *Newton's method on McCormick's function in the second iteration.*



(e) *Newton's method on McCormick's function in the third iteration.*

(f) *Newton's method on McCormick's function in the fourth iteration.*

Figure 2.3.: *Steepest descent (plot (a)) and Newton's method (plots (b)-(f)) on McCormick's function [3], which is defined by* $f(w_1, w_2) = \sin w_1 + w_2 + (w_1 - w_2)^2 - 1.5w_1 + 2.5w_2 + 1$ *and has a global solution at* $\boldsymbol{w} = (-0.547, -1.547)$ *(the red triangle in plots (a) and (b)). The initialization point for both methods is* $\boldsymbol{w}^{(0)} = (-0.5, 1)$. *While steepest descent is able to converge into the global minimum, Newton's method only converges into the local minimum. Plots (c)-(f) depict individual iterations of Newton's method together with contour lines of the local second-order approximation to the error function* $f(w_1, w_2)$ *that is minimized at each iteration of Newton's method. The approximate problem that is solved by Newton's method is given by Equation (2.22).*

Though it is difficult to say in practice when a given point is sufficiently close to a solution, one can expect that eventually the fast convergence rate of Newton's method will come into effect [24].

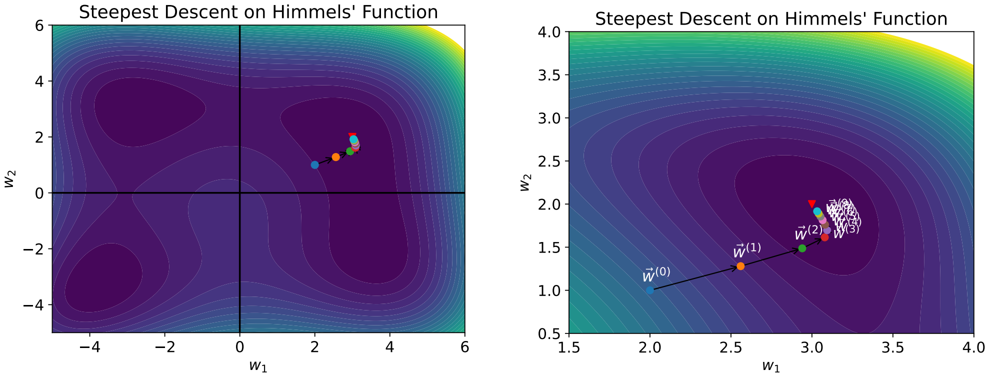
On the other hand, one cannot make the assumption of a positive definite Hessian when looking at the global convergence, that is when further away from any potential solution. There, Newton's method has several drawbacks. Firstly, the inverse of the Hessian $(\nabla^2 f(\boldsymbol{w}^{(k)}))^{-1}$ might not exist, which is the case whenever the Hessian is singular (has at least one zero eigenvalue). As will be shown in the next chapter, most neural networks have eigenspectra where the bulk of the eigenvalues is exactly or close to zero, even close to a potential solution. Secondly, Newton's method used with step size of $\varepsilon_k = 1$ is not a descent direction, because the value of the cost function of the next iteration can be higher than that of the current iteration [24]. Thirdly, as shown by the derivation in (2.22), Newton's method only tries to solve for $\nabla f(\boldsymbol{w}^{(k)}) = 0$, which is also fulfilled by maxima and saddle points. Therefore, it is important to keep in mind that this method is not only attracted to minima. There exist several solutions to these global problems that modify Newton's method into a viable gradient method, such as trust region methods [151] for example.

Figure 2.4 highlights the difference between steepest descent, a first-order method, and Newton's method. In this case the objective function is non-convex and both methods are initialized at the same point. One can observe in this example how Newton's method struggles to converge to the minimum, while the steepest descent method is able to converge to the optimal point. This simple example illustrates that one has to be careful with the choice of optimization method, and sometimes second-order methods can behave worse than the simpler first-order methods, e.g. when the Hessian is not positive definite.

For large deep neural networks another problem arises. Due to the size of the Hessian, it is computationally intractable to compute the inverse at every iteration. Storing the Hessian itself is already infeasible due to its size, effectively rendering the practicality of Newton's method for deep learning useless. Some methods attempt to solve these issues by only reevaluating the Hessian after every few iterations [170], or by computing an approximation of the Hessian [69].

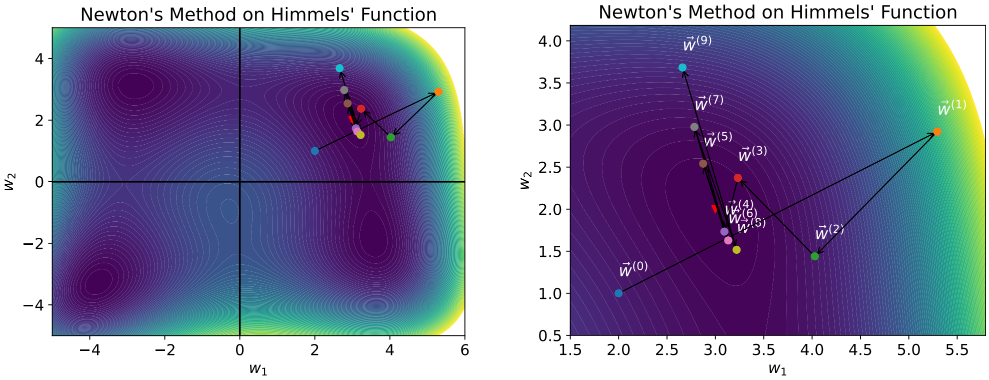
For difficult problems, where the Hessian is discontinuous or not positive definite near minima of interest, the rate of convergence of second-order methods like Newton's method can be worse than that of simpler first-order methods. Another problem arises when the initialization starts far away from any local minima. In that case second-order methods may progress very slowly until they get to a small neighborhood of the solution where their convergence is favorable to other methods [24]. This can hint at an explanation why there has not been much practical benefit to Hessian methods and their approximations and why simpler and therefore faster methods like stochastic gradient descent still outperform these more sophisticated methods in practice.

Another issue is the convergence rate of Newton's method in the stochastic case. Using local convergence analysis, [113] show that the superlinear rate of convergence for the non-stochastic Newton's method reduces to a linear convergence rate for a batch size of one. This fact together with the high computational cost per iteration do not make Newton's method an attractive choice for many problems in non-convex



(a) *Steepest descent on Himmelblau's function.*

(b) *Closer look on the iterations from steepest descent.*



(c) *Newton's method on Himmelblau's function.*

(d) *Closer look on the iterations from Newton's method.*

Figure 2.4.: *Comparison of steepest descent and Newton's method on Himmelblau's function [87]. Himmelblau's function is given by* $f(w_1,w_2) = (w_1^2 + w_2 - 11)^2 + (w_1 + w_2^2 - 7)^2$ *and has four different minima with the closest to initialization being at* $\boldsymbol{w}^\star = (3,2)$*. Both methods are initialized at* $\boldsymbol{w}^{(0)} = (2,1)$ *and run for 9 iterations. The x- and y-axis represent the values of the two parameters of the function.*

optimization. Though recently, [153] were able to show that a regularized subsampled Newton method is able to achieve quadratic convergence on over-parameterized models. One downside to this method is that in order to achieve this convergence it requires an exponentially growing batch-size.

2.2. Regularization

Some problems in optimization require a constrained set of possible parameters. These constrains are defined through equalities or inequalities. There are different reasons for choosing to constrain the set of parameters instead of using unconstrained optimization, some of the reasons being to stabilize the network, to shrink the network size or to increase generalization performance [243].

The constrained optimization problem can be written as [24]

$$\begin{aligned}&\text{minimize } f(\boldsymbol{w})\\&\text{subject to } r_i(\boldsymbol{w}) \leq 0 \text{ and } h_j(\boldsymbol{w}) = 0\end{aligned}$$

where $r_i(\boldsymbol{w}) \leq 0$ represent inequality constrains and $h_j(\boldsymbol{w}) = 0$ represent equality constrains.

By using the *Karush–Kuhn–Tucker* (KKT) conditions [117], this optimization problem can also be written in terms of the generalized Lagrangian $\mathcal{L}(\boldsymbol{w})$

$$\mathcal{L}(\boldsymbol{w};\boldsymbol{\lambda},\boldsymbol{\gamma}) = f(\boldsymbol{w}) + \sum_{i=1}^{s} \lambda_i r_i(\boldsymbol{w}) + \sum_{j=1}^{q} \gamma_j h_j(\boldsymbol{w}) \tag{2.24}$$

where λ_i and γ_j represent regularization factors, which control the relaxation from the constrained optimization problem.

Because the succeeding chapters will only deal with a single constrain in the Lagrangian, this thesis will just use $r(\boldsymbol{w})$ as the regularization term, which simplifies the above expression to

$$\mathcal{L}(\boldsymbol{w};\lambda) = f(\boldsymbol{w}) + \lambda r(\boldsymbol{w}). \tag{2.25}$$

The Lagrangian can now be solved by [24]

$$\boldsymbol{w}^{\star} = \arg\min_{\boldsymbol{w}} \max_{\lambda} \mathcal{L}(\boldsymbol{w};\lambda). \tag{2.26}$$

As shown in the next subsections, most regularization terms used in this thesis are norm penalties. In this case, the goal is to restrict the norm of the parameters of the neural network model to be smaller than some predefined value C

$$r(\boldsymbol{w}) < C. \tag{2.27}$$

This turns the Lagrangian into

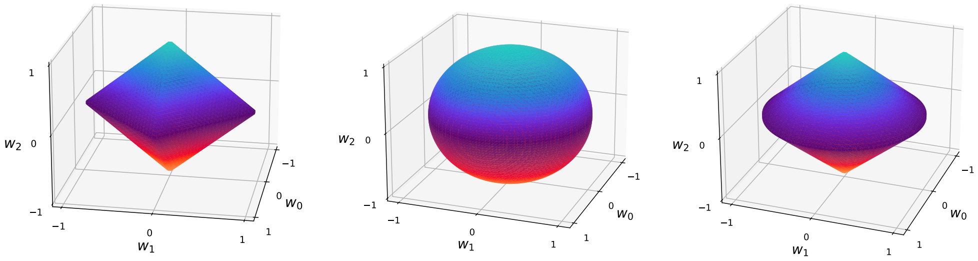
$$\mathcal{L}(\boldsymbol{w};\lambda) = f(\boldsymbol{w}) + \lambda(r(\boldsymbol{w}) - C) \tag{2.28}$$

In order to solve (2.26), whenever the regularization term $r(\boldsymbol{w})$ is bigger than C, the value of λ has to increase in order to force $r(\boldsymbol{w})$ to shrink faster. The optimal value $\lambda^\star$ will make sure that the constraint of the regularization term in Equation (2.27) is satisfied and that the gradient of the Lagrangian at $\lambda^\star$ and the converged point $\boldsymbol{w}^\star$ will be zero [24]

$$\nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}^\star; \lambda^\star) = \mathbf{0}$$
$$\nabla_{\lambda} \mathcal{L}(\boldsymbol{w}^\star; \lambda^\star) = \mathbf{0}.$$

In practice, the constant C is removed and instead of finding a suitable $\lambda^\star$ for a predefined C, a fixed value for λ is chosen. In this thesis this fixed value will be called the regularization constant μ. This μ will be the optimal value for some implicit constant C', so the network will experience some level of regularization, though this level is not known a priori. This usually requires many different runs with different values for μ in order to find a suitable regularization of the network. Also, by fixing the regularization constant, the constant C can be dropped from the expression in Equation (2.28), due to the fact that it will disappear when taking the gradient of the loss function with respect to the parameters.

The following subsections will introduce the ℓ_1-, ℓ_2- and $\ell_{2,1}$-norm regularizations, where the ℓ_1- and $\ell_{2,1}$-norm regularizations will be especially important in this thesis.



(a) *Depiction of the ℓ_1-norm. The x-, y- and z-axis represent three parameters w_0, w_1 and w_2. The plot shows the surface where the value of the ℓ_1-norm equals one.*

(b) *Depiction of the ℓ_2-norm. The x-, y- and z-axis represent three parameters w_0, w_1 and w_2. The plot shows the surface where the value of the ℓ_2-norm equals one.*

(c) *Depiction of the $\ell_{2,1}$-norm. The x-, y- and z-axis represent three parameters w_0, w_1 and w_2. The parameters x_0 and x_1 belong to the same group, while the parameter x_2 belongs to a different group. The plot shows the contour surface where the value of the $\ell_{2,1}$-norm equals one.*

Figure 2.5.: *Plot of the ℓ_1-, ℓ_2- and $\ell_{2,1}$-norm. In all cases the function is depicted in three dimensions, and the surface depicts where the value of the three parameters is equal to one. In the case of the $\ell_{2,1}$-norm, the parameters depicted by w_0 and w_1 are placed into the same group, while the parameter w_2 belongs to a different group.*

2.2.1. ℓ_1-Regularization

The ℓ_1-regularization is most commonly used for pruning of individual weights. It uses the ℓ_1-norm and is defined as

$$r(\boldsymbol{w}) = \sum_{i=1}^{N} |w_i| \tag{2.29}$$

for a parameter vector $\boldsymbol{w} \in \mathbb{R}^N$. Thus, the loss function is written as

$$\mathcal{L}(\boldsymbol{w}) = f(\boldsymbol{w}) + \mu \sum_{i=1}^{N} |w_i| \tag{2.30}$$

with $f(\boldsymbol{w})$ the original objective function and μ the regularization parameter. Since the regularization term $r(\boldsymbol{w})$ is not continuous at zero, optimization relies on stochastic subgradient descent in order to converge to a solution.

Popular frameworks, like PyTorch [174] and TensorFlow [152] for example, use the following rules to compute the subgradient of the ℓ_1 term:

$$\frac{\partial r(x)}{\partial x} = \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ 1 & \text{if } x > 0 \end{cases}$$

In order to understand the effects of ℓ_1-regularization, the rest of this subsection will follow the steps taken in [76] in order to observe how the local optimum of the unregularized objective function changes in the presence of the regularization term. This will be achieved by taking the quadratic approximation of the unregularized objective function around the optimal point $\boldsymbol{w}^*$.

The subgradient of the loss function is given by:

$$\nabla \mathcal{L}(\boldsymbol{w}) = \nabla f(\boldsymbol{w}) + \mu \times sign(\boldsymbol{w}) \tag{2.31}$$

where $sign(\boldsymbol{w})$ is applied element-wise.

The quadratic approximation around the minimum $\boldsymbol{w}^*$ of the unconstrained objective function is

$$f(\boldsymbol{w}) = f(\boldsymbol{w}^*) + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}^*)^T H_{\boldsymbol{w}^*}(\boldsymbol{w} - \boldsymbol{w}^*). \tag{2.32}$$

In order to simplify this equation, assume that the Hessian is diagonal with $H = diag(H_{1,1}, \ldots, H_{N,N})$, where each entry is a positive number. This assumption simplifies the Taylor approximation and together with the regularization term this results in

$$f(\boldsymbol{w}) = f(\boldsymbol{w}^*) + \sum_{i=1}^{N} \left[\frac{1}{2} H_{i,i}(w_i - w_i^*)^2 + \mu |w_i| \right]. \tag{2.33}$$

This function is minimized if

$$w_i = sign(w_i^*) \max\{|w_i^*| - \frac{\mu}{H_{i,i}}, 0\}. \tag{2.34}$$

Whenever the value of $w_i^* \in [0, \mu/H_{i,i}]$, the optimal value of the regularized weight will be equal to zero. If the value of w_i^* is higher than $\mu/H_{i,i}$, it will be shifted by an amount of $\mu/H_{i,i}$. Because of the $sign(w_i^*)$ expression in front of the equation, both cases also hold for the case where $w_i^* < 0$.

This explains how ℓ_1-regularization introduces sparsity into a neural network by setting part of its weights exactly to zero. This property can be very beneficial, especially when dealing with very large neural networks, that require large amounts of memory and storage. By shrinking the networks down in size, this can theoretically make them faster and more memory efficient, while sacrificing as little accuracy as possible.

2.2.2. ℓ_2-Regularization

In ℓ_2-regularization the ℓ_2-norm of the weight vector $\boldsymbol{w}$ is added to the objective function. The ℓ_2-regularization is given by

$$r(\boldsymbol{w}) = \frac{1}{2}\left(\sum_{i=1}^{N} w_i^2\right) \tag{2.35}$$

or in vector notation, $r(\boldsymbol{w}) = \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w}$. Thus, the new loss function with ℓ_2-regularization has the following form

$$\mathcal{L}(\boldsymbol{w}) = f(\boldsymbol{w}) + \frac{\mu}{2}\left(\sum_{i=1}^{N} w_i^2\right). \tag{2.36}$$

The gradient of this expression becomes

$$\nabla\mathcal{L}(\boldsymbol{w}) = \nabla f(\boldsymbol{w}) + \mu\boldsymbol{w} \tag{2.37}$$

and for SGD this results in the following update step

$$\begin{aligned} \boldsymbol{w}^{(k+1)} &= \boldsymbol{w}^{(k)} - \varepsilon_k \nabla f(\boldsymbol{w}^{(k)}) - \varepsilon_k \mu \boldsymbol{w}^{(k)} \\ &= (1 - \varepsilon_k \mu)\boldsymbol{w}^{(k)} - \varepsilon_k \nabla f(\boldsymbol{w}^{(k)}) \end{aligned}$$

Contrary to ℓ_1-regularization, training a neural network with ℓ_2-regularization results in a rotationally invariant algorithm [166]. Rotationally invariant algorithms are not effective in selecting relevant features during training, with the worst case sample complexity growing linearly in the number of irrelevant features. Thus, this form of regularization does not make the neural network sparser, but it can result in faster convergence and in better generalization of the neural network [130].

Following the steps of [76], the effects of this regularization are shown by making a quadratic approximation of the unconstrained objective function around its minimum $\boldsymbol{w}^*$:

$$f(\boldsymbol{w}) = f(\boldsymbol{w}^*) + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}^*)^T H_{\boldsymbol{w}^*}(\boldsymbol{w} - \boldsymbol{w}^*). \tag{2.38}$$

Its derivative is given by

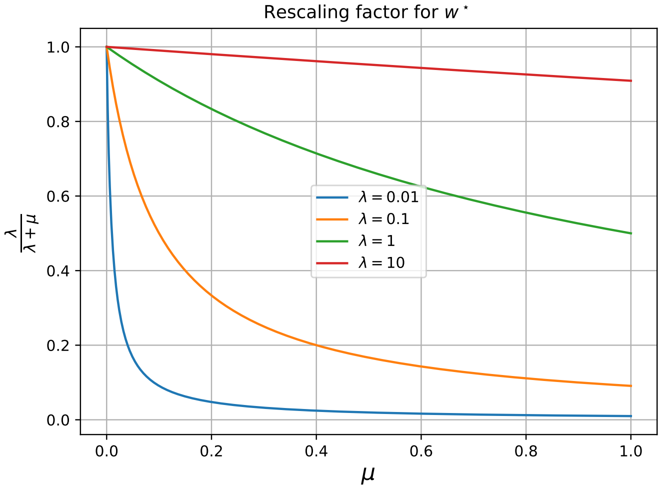


Figure 2.6.: *Depiction of how the value of $\boldsymbol{w}^*$ is rescaled for different values of μ and different eigenvalues λ. One can see how larger values of μ shrink the value of $\boldsymbol{w}^*$ further toward zero and how directions with large eigenvalues do not shrink as drastically as directions with smaller eigenvalues.*

$$\nabla f(\boldsymbol{w}) = H_{\boldsymbol{w}^*}(\boldsymbol{w} - \boldsymbol{w}^*). \tag{2.39}$$

The first order necessary condition for reaching the minimum is that this expression is zero. Now the gradient of the ℓ_2-regularization is added to this expression in order to observe how this term affects the solution:

$$H_{\boldsymbol{w}^*}(\boldsymbol{w} - \boldsymbol{w}^*) + \mu \boldsymbol{w} = 0. \tag{2.40}$$

Rearranging this equation

$$\boldsymbol{w} = (H + \mu I)^{-1} H \boldsymbol{w}^* \tag{2.41}$$

and using the fact that it is possible to decompose H into $H = Q\Lambda Q^T$, the following result is obtained

$$\boldsymbol{w} = Q(\Lambda + \mu I)^{-1} \Lambda Q^T \boldsymbol{w}^*. \tag{2.42}$$

One can see that along a certain eigenvector, the value of the unregularized optimum $\boldsymbol{w}^*$ is rescaled with $\frac{\lambda_i}{\lambda_i+\mu}$. Figure 2.6 depicts how the unregularized optimum $\boldsymbol{w}^*$ is rescaled for different values of μ and λ.

2.2.3. $\ell_{2,1}$-Regularization

The $\ell_{2,1}$-regularization, sometimes also called group sparsity, describes the following norm

$$r(\boldsymbol{w}) = \sum_{g \in \mathcal{G}} \left(\sum_{i=1}^{|g|} w_{g_i}^2 \right)^{\frac{1}{2}} \tag{2.43}$$

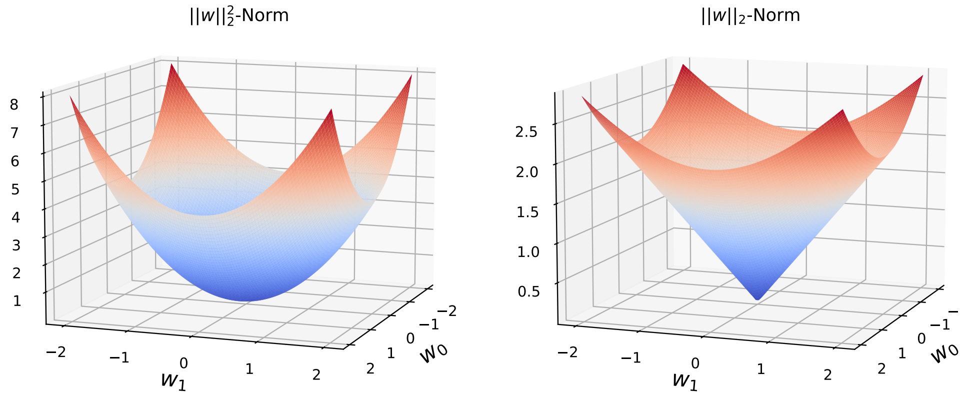
with $\mathcal{G}$ the set of groups and $|g|$ the number of elements of group g. One can ease the notation by decomposing the vector $\boldsymbol{w}$ into its L different groups with $\boldsymbol{w} = (\boldsymbol{w}_l)_{l=1}^{L}$. This turns the expression into:

$$r(\boldsymbol{w}) = \sum_{l=1}^{L} \|\boldsymbol{w}_l\|_2 \tag{2.44}$$

The $\ell_{2,1}$-norm is a non-smooth and convex function and serves as a natural extension to the ℓ_1-norm when $\boldsymbol{w}$ is a vector. The subgradient of the $\ell_{2,1}$-norm is given by

$$\partial \|\boldsymbol{w}\|_2 = \begin{cases} \frac{\boldsymbol{w}}{\|\boldsymbol{w}\|_2}, & \text{if } \boldsymbol{w} \neq \mathbf{0} \\ \{\boldsymbol{v} \,|\, \|\boldsymbol{v}\|_2 \leq 1\}, & \text{otherwise} \end{cases} \tag{2.45}$$

An important difference between the ℓ_2-regularization from the previous section and the $\ell_{2,1}$-regularization of this section is that the ℓ_2-regularization uses the squared ℓ_2-norm, which makes it differentiable everywhere. This difference is depicted in Figure 2.7.



(a) *Depiction of the squared ℓ_2-norm. The x- and y-axis represent two parameters x_0 and x_1.*

(b) *Depiction of the ℓ_2-norm. The x- and y-axis represent two parameters x_0 and x_1.*

Figure 2.7.: *Plot of the squared ℓ_2- and the regular ℓ_2-norm. Plot (a) depicts the squared ℓ_2-norm, which is used in ℓ_2-regularization and it is differentiable everywhere. Plot (b) depicts the regular ℓ_2-norm, which is used in $\ell_{2,1}$-regularization. This function is not differentiable at zero.*

Following similar steps as in the other two regularizations, the second-order approximation around the minimum of the unconstrained objective function is written as

$$f(\boldsymbol{w}) = f(\boldsymbol{w}^*) + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}^*)^T H_{\boldsymbol{w}^*}(\boldsymbol{w} - \boldsymbol{w}^*). \tag{2.46}$$

Now, looking at a certain group $\boldsymbol{w}_g$, one can write the derivative as

$$0 \in \Lambda_g(\boldsymbol{w}_g - \boldsymbol{w}_g^\star) + \mu\partial \|\boldsymbol{w}_g\|_2 \tag{2.47}$$

where the simplifying assumption is made that the Hessian can be decomposed into the different groups with $H_{w_g} = \Lambda_g$, with $\Lambda_g = \lambda_g I$ a diagonal matrix that contains the same eigenvalue along its diagonal and $\lambda_g > 0$. First, consider the case where $\boldsymbol{w}_g \neq 0$:

$$0 \in \Lambda_g(\boldsymbol{w}_g - \boldsymbol{w}_g^\star) + \mu\frac{\boldsymbol{w}_g}{\|\boldsymbol{w}_g\|_2} \tag{2.48}$$

$$\Lambda_g\boldsymbol{w}_g + \mu\frac{\boldsymbol{w}_g}{\|\boldsymbol{w}_g\|_2} = \Lambda_g\boldsymbol{w}_g^\star \tag{2.49}$$

$$\boldsymbol{w}_g + \mu\Lambda_g^{-1}\frac{\boldsymbol{w}_g}{\|\boldsymbol{w}_g\|_2} = \boldsymbol{w}_g^\star \tag{2.50}$$

$$\boldsymbol{w}_g + \frac{\mu}{\lambda_g}\frac{\boldsymbol{w}_g}{\|\boldsymbol{w}_g\|_2} = \boldsymbol{w}_g^\star \tag{2.51}$$

$$\boldsymbol{w}_g + \frac{\mu}{\lambda_g}\frac{\boldsymbol{w}_g^\star}{\|\boldsymbol{w}_g^\star\|_2} = \boldsymbol{w}_g^\star \tag{2.52}$$

$$\boldsymbol{w}_g = (I - \frac{\mu}{\lambda_g\|\boldsymbol{w}_g^\star\|_2})\boldsymbol{w}_g^\star \tag{2.53}$$

In Equation (2.51), the left hand side is just the addition of the vector $\boldsymbol{w}_g$ with its own directional vector that is scaled by a positive constant. Thus, vector $\boldsymbol{w}_g^\star$ will always point in the same direction and therefore the normed directional vector can be replaced, from $\frac{\boldsymbol{w}_g}{\|\boldsymbol{w}_g\|_2}$ in Equation (2.51) to $\frac{\boldsymbol{w}_g^\star}{\|\boldsymbol{w}_g^\star\|_2}$ in Equation (2.52).

In the second case, where $\boldsymbol{w}_g = \mathbf{0}$, Equation (2.47) results in

$$\begin{aligned} 0 &\in -\Lambda_g\boldsymbol{w}_g^\star + \mu\{\boldsymbol{v} \mid \|\boldsymbol{v}\|_2 \leq 1\} \\ \Lambda_g\boldsymbol{w}_g^\star &\in \mu\{\boldsymbol{v} \mid \|\boldsymbol{v}\|_2 \leq 1\} \\ \|\Lambda_g\boldsymbol{w}_g^\star\|_2 &\leq \mu \end{aligned}$$

The last inequality can be further simplified by using $\|\Lambda_g\boldsymbol{w}_g^\star\|_2 = \lambda_g\|\boldsymbol{w}_g^\star\|_2$:

$$\lambda_g\|\boldsymbol{w}_g^\star\|_2 \leq \mu \tag{2.54}$$

$$\|\boldsymbol{w}_g^\star\|_2 \leq \frac{\mu}{\lambda_g} \tag{2.55}$$

This result shows that group sparsity forces groups of parameters exactly to zero whenever the ℓ_2-norm of the group is below a certain threshold. Also, Equation (2.55)

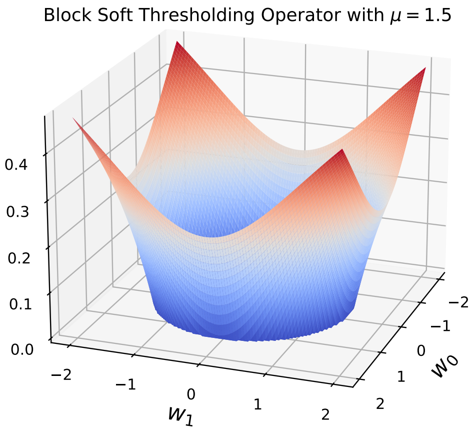


Figure 2.8.: *Depiction of the block soft thresholding operator from Equation (2.56) with $\mu = 1.5$ and $\lambda = 1$.*

shows that this threshold is rescaled with the eigenvalue of the group.

By combining both results for the two cases, this yields the following equation between $\boldsymbol{w}$ and $\boldsymbol{w}_g^\star$:

$$\boldsymbol{w}_g = \left(1 - \frac{\mu}{\lambda_g \left\|\boldsymbol{w}_g^\star\right\|_2}\right)^+ \boldsymbol{w}_g^\star \tag{2.56}$$

where $(x)^+$ is equivalent to $\max\{x, 0\}$, which in the equation above returns the zero vector whenever $\left\|\boldsymbol{w}_g^\star\right\|_2 \leq \frac{\mu}{\lambda_g}$. An example of this equation for two parameters is shown in Figure 2.8.

The implications of this equation are twofold. First, the bigger the norm of the group, the smaller the term on the right hand side inside the brackets. This forces groups with large ℓ_2-norms less toward zero. Second, the bigger the eigenvalue corresponding to the group, the smaller the effect of the $\ell_{2,1}$-regularization on the group.

The hyperparameter μ indirectly controls the achieved sparsity and the higher the value of μ, the sparser the solution will be. Due to the randomness of stochastic algorithms and the non-convexity of the optimization problem, it is not possible to assign a certain predefined sparsity level given some μ and the same μ can lead to different sparsity levels. Similar to the ℓ_1-norm, optimization using the $\ell_{2,1}$-norm often relies on stochastic subgradient descent in order to be able to converge to a solution, which can be very inefficient.

2.3. Eigenvalue Computation

This section will go step by step through all the necessary theory and concepts that are needed in order to compute eigenvalues of the Hessian for large neural networks and will largely follow [10].

Given a matrix $A \in \mathbb{R}^{N \times N}$, the general eigenvalue problem is to find eigenvalues $\lambda \in \mathbb{R}$ and eigenvectors $u \in \mathbb{R}^N$, such that

$$Au = \lambda u \tag{2.57}$$

Eigenvectors describe directions in space that are unaffected by the transformation induced by matrix A (up to a scaling constant). Furthermore, the corresponding eigenvalues describe how these vectors change their magnitude and direction. Equation (2.57) is solved by

$$det(A - \lambda I) = 0. \tag{2.58}$$

For symmetric matrices ($A^T = A$), it follows that their eigenvalues are real, $\lambda_i \in \mathbb{R}$ and the N eigenvectors $u_1, \ldots, u_N$ are orthogonal, real valued and non-zero [24]. This symmetric matrix A can be decomposed as $A = Q \Lambda Q^T$, where the matrix Q contains the orthonormalized eigenvectors u_i, and the matrix Λ is diagonal with the corresponding eigenvalues along its diagonal.

Computing eigenvalues and eigenvectors is important in many different fields in science, ranging from the computation of energy levels in quantum physics [197] to the computation of the vibrations of a string [178]. The eigenvalues of the Hessian of a neural network are of particular interest in deep learning. These reveal the local curvature at a given point in parameter space [125], which can give insights into the training dynamics and potentially help guide us to find better optimizers in the future.

2.3.1. The Hessian

The Hessian and its corresponding eigenspectrum can be useful in many different ways, for example in deriving more efficient optimizers or in analyzing different properties of a neural network model.

The Hessian of a function $f : \mathbb{R}^n \longrightarrow \mathbb{R}$ is defined as

$$H_{i,j} = \frac{\partial^2 f(\boldsymbol{w})}{\partial w_i \partial w_j} \tag{2.59}$$

If the derivative of the function f is continuous, the order of the partial derivatives does not matter and can be interchanged, which makes the Hessian symmetric. Because it is a real and symmetric matrix, the Hessian can be decomposed into a set of real eigenvalues and an orthonormal basis of eigenvectors.

The eigenvalues describe the curvature along their corresponding eigenvector [125]. Using the eigenvalues of the decomposition, it is often possible to see if the converged point is at a maximum, minimum or saddle point. If all the eigenvalues are positive, the function has converged into a minimum. If they are negative, the function converged to a maximum. If there are negative and positive eigenvalues present, the function has converged onto a saddle point. If the eigenvalues are zero, it is not possible to tell to what point the function has converged to and one needs to further investigate.

The Hessian matrix of a trained model can be also used for outlier identification, to quantify other sorts of uncertainties or for model validation [137]. It has also been shown that the fraction of negative eigenvalues of the Hessian is related to the rate of convergence for different optimizers in the non-convex case [99] as well as to the overall number of critical points in a high-dimensional loss landscape [30].

The condition number C of a matrix A is given by the ratio of the biggest and the smallest eigenvalue of A

$$C = \max_{i,j} \left|\frac{\lambda_i}{\lambda_j}\right| \tag{2.60}$$

The condition number of a matrix A measures the how sensitive some function $\mathcal{K}(\boldsymbol{x}) = A^{-1}\boldsymbol{x}$ is to small changes in its inputs. Poor conditioning of the Hessian makes it difficult to choose a good step size [24].

If the domain of the function is N-dimensional, the size of the Hessian matrix will be N^2. In real world examples, the dimension of the domain of common neural networks is high-dimensional, which results in an extremely large number of elements of the Hessian. For a VGG-16 network [204] with 138 million parameters, its corresponding Hessian will have 1.9×10^{16} entries. Storing this Hessian using floats would require approximately 76 Petabytes of storage.

2.3.2. The R-Operator

The R-Operator was first introduced by [175] and is a linear operator that allows for efficient Hessian-vector computation. The operator wraps around the forward and backward propagation of a neural network in order to compute the Hessian-vector product efficiently.

The Hessian-vector product $H\boldsymbol{\nu}$ is computed using

$$\begin{aligned}(H\boldsymbol{\nu})_i &= \sum_{j=1}^{n} \frac{\partial^2 f(\boldsymbol{w})}{\partial w_j \partial w_i} v_j \\ &= \nabla \frac{\partial f(\boldsymbol{w})}{\partial w_i} \boldsymbol{\nu}.\end{aligned}$$

This is just the directional derivative of the gradient of the objective function along $\boldsymbol{\nu}$. The directional derivative of some function g along $\boldsymbol{\nu}$ is given by

$$\nabla_{\boldsymbol{\nu}} g(\boldsymbol{w}) = \lim_{r \to 0} \frac{g(\boldsymbol{w} + r\boldsymbol{\nu}) - g(\boldsymbol{w})}{r}. \tag{2.61}$$

In the Hessian-vector case this means that

$$H\boldsymbol{\nu} = \lim_{r \to 0} \frac{\nabla f(\boldsymbol{w} + r\boldsymbol{\nu}) - \nabla f(\boldsymbol{w})}{r} \tag{2.62}$$

which is the same as

$$H\boldsymbol{\nu} = \frac{\partial}{\partial r} \nabla f(\boldsymbol{w} + r\boldsymbol{\nu}) \bigg|_{r=0} \tag{2.63}$$

The R-operator with respect to the vector $\boldsymbol{\nu}$ is defined as

$$R_{\boldsymbol{\nu}}\{f(\boldsymbol{w})\} = \frac{\partial}{\partial r} f(\boldsymbol{w} + r\boldsymbol{\nu})\bigg|_{r=0} \tag{2.64}$$

which is the directional derivative of f along the direction $\boldsymbol{\nu}$ evaluated at the current point $\boldsymbol{w}$. Note that it follows from this definition that $H\boldsymbol{\nu} = R_{\boldsymbol{\nu}}\{\nabla L(\boldsymbol{w})\}$. Using the properties of linear operators, computing the Hessian-vector product requires the computation of the R-operator during a forward pass and a backward pass.

As an example, consider a fully-connected network, which is given by:

$$\begin{aligned} h_i^l &= \sum_j w_{i,j}^l z_j^l, \\ z_i^{l+1} &= \chi_i^l(h_i^l) \end{aligned} \tag{2.65}$$

where $w_{i,j}^l$ represents the parameters of the model at layer l, z_j^l represents the j-th input to layer l and $\chi^l(\boldsymbol{h}^l)$ the non-linear activation function at layer l. Note that $\boldsymbol{z}^0 = \boldsymbol{x}$, with $\boldsymbol{x}$ the input samples. The backward pass for layer l is given by

$$\begin{aligned} \frac{\partial f}{\partial z_i^l} &= \sum_m \frac{\partial f}{\partial h_m^l} w_{i,m}^l, \\ \frac{\partial f}{\partial h_i^l} &= \frac{\partial f}{\partial z_i^{l+1}} \frac{\partial \chi_i^l(s)}{\partial s}\bigg|_{s=h_i^l}, \\ \frac{\partial f}{\partial w_{i,j}^l} &= \frac{\partial f}{\partial h_i^l} z_j^l \end{aligned} \tag{2.66}$$

When the R-operator is now applied to the forward pass, Equations (2.65) become

$$\begin{aligned} \mathcal{R}\{h_i^l\} &= \sum_j \left(v_{i,j}^l z_j^l + w_{i,j}^l \mathcal{R}\{z_j^l\} \right), \\ \mathcal{R}\{z_i^{l+1}\} &= \frac{\partial \chi_i^l(s)}{\partial s}\bigg|_{s=h_i^l} \mathcal{R}\{h_i^l\}. \end{aligned} \tag{2.67}$$

Note that the R-operator at the inputs is $\mathcal{R}\{\boldsymbol{z}^0\} = 0$ and that $\mathcal{R}\{\boldsymbol{w}\} = \boldsymbol{v}$. An illustration of the forward pass without and with the R-operator is depicted in Figure 2.9.

Finally, applying the R-operator during the backward pass one obtains

$$\begin{aligned} \mathcal{R}\left\{\frac{\partial f}{\partial z_i^l}\right\} &= \sum_m \left(\mathcal{R}\left\{\frac{\partial f}{\partial h_m^l}\right\} w_{i,m}^l + \frac{\partial f}{\partial h_m^l} \nu_{i,m}^l \right), \\ \mathcal{R}\left\{\frac{\partial f}{\partial h_i^l}\right\} &= \mathcal{R}\left\{\frac{\partial f}{\partial z_i^{l+1}}\right\} \frac{\partial \chi_i^l(s)}{\partial s}\bigg|_{s=h_i^l} + \frac{\partial f}{\partial z_i^{l+1}} \frac{\partial^2 \chi_i^l(s)}{\partial s^2}\bigg|_{s=h_i^l} \mathcal{R}\left\{h_i^l\right\}, \\ \mathcal{R}\left\{\frac{\partial f}{\partial w_{i,j}^l}\right\} &= \mathcal{R}\left\{\frac{\partial f}{\partial h_i^l}\right\} z_j^l + \frac{\partial f}{\partial h_i^l} \mathcal{R}\left\{z_j^l\right\} \end{aligned} \tag{2.68}$$

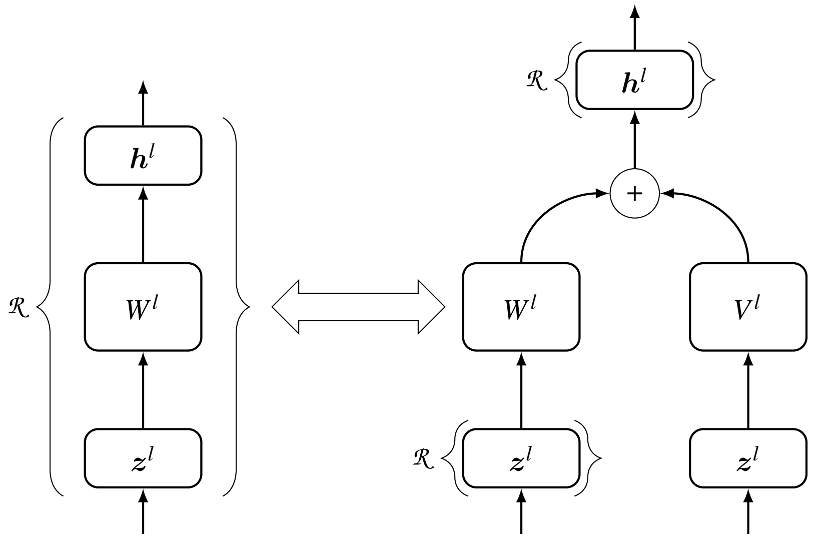


Figure 2.9.: *Illustration of the $\mathcal{R}$-operator for a linear operator in the forward pass. Applying the operator splits the forward pass into two paths as seen on the right-hand side.*

Looking at Equation (2.68), it becomes evident why a forward pass using the R-operator is needed, which is due to the existence of $\mathcal{R}\left\{z_j^l\right\}$ and $\mathcal{R}\left\{h_i^l\right\}$ in these backward pass equations. Using this operator allows for an efficient computation of the Hessian-vector product in $O(N)$ instead of $O(N^2)$, without having to store the Hessian during this process.

2.3.3. The Lanczos Algorithm

Given a symmetric matrix A, the goal is to efficiently compute (some of) its eigenvalues and eigenvectors. Here, the eigenvalues of a matrix A are represented by λ_i and their corresponding eigenvectors by u_i. This means that the following equation is satisfied

$$A\mathrm{u}_i = \lambda_i \mathrm{u}_i \tag{2.69}$$

and without loss of generality assume that the eigenvalues are sorted by magnitude $|\lambda_1| \geq |\lambda_2| \geq \dots |\lambda_n|$.

If A is a symmetric $n \times n$ matrix, it follows that its eigenvalues are real and it has a set of n mutually orthogonal, real and nonzero eigenvectors $\mathrm{u}_1, \dots, \mathrm{u}_n$. Because of this property, the eigenvectors form a basis in this space and any vector can be written in terms of these eigenvectors. This means that an arbitrary vector $\boldsymbol{x}$ can be decomposed into

$$\boldsymbol{x} = \sum_{i=1}^{n} \xi_i \mathrm{u}_i. \tag{2.70}$$

Another property that A satisfies is that the matrix A^k has eigenvalues $\lambda_1^k, \lambda_2^k, \dots, \lambda_n^k$. This can be shown by repeatedly multiplying the matrix A with an eigenvector.

The simplest algorithm that is able to compute the largest eigenvalue and its corresponding eigenvector is the power iteration method [157]. The basic idea behind the power iteration method is that by multiplying a randomly chosen vector repeatedly with the matrix A, the largest eigenvalue will eventually start to dominate. This is shown in Equation (2.71), where the matrix A^k is multiplied by an arbitrary vector $\boldsymbol{x}$

$$A^k \boldsymbol{x} = \sum_{i=1}^{n} \xi_i \lambda_i^k \mathrm{u}_i. \tag{2.71}$$

Here, the decomposition of $\boldsymbol{x}$ shown in Equation (2.70) was used. At iteration k, the power iteration method is given by

$$\boldsymbol{x}^{(k+1)} = \frac{A\boldsymbol{x}^{(k)}}{\|A\boldsymbol{x}^{(k)}\|} \tag{2.72}$$

where the vector $\boldsymbol{x}^{(k)}$ converges to the eigenvector associated with the biggest eigenvalue of matrix A.

While the power iteration method is able to compute eigenvalues, two issues arise. One is that computations of previous iterations in the power method are discarded, which could be used in later iterations. The second issue with with the power iteration method and most other methods that compute eigenvalues of a matrix, is that they require the entire matrix to be computed. In case of very large neural networks, computing and storing the Hessian in infeasible, and thus most methods used in eigenvalue computation are not suitable for this problem at hand.

One method that makes use of previous iterations and only requires Hessian-vector products is the Lanczos algorithm [120]. The Lanczos algorithm aims to find an invariant subspace of the Krylov space [116], which is defined as

$$\mathcal{K}^m(\boldsymbol{x}, A) := span\{\boldsymbol{x}, A\boldsymbol{x}, A^2\boldsymbol{x}, \ldots, A^{m-1}\boldsymbol{x}\} \subset \mathbb{R}^N. \tag{2.73}$$

As evident from Definition (2.73), a natural basis of this space is $\{\boldsymbol{x}, A\boldsymbol{x}, A^2\boldsymbol{x}, \ldots, A^{m-1}\boldsymbol{x}\}$. This basis converges to the direction of the largest eigenvalue of A. Thus, it is badly conditioned with increasing m. Therefore, in the Lanczos method the vectors $\boldsymbol{q}_j$ of this basis are successively orthogonalized against all the others

$$\boldsymbol{r}_j = A\boldsymbol{q}_j - \sum_{i=1}^{j} \boldsymbol{q}_i \boldsymbol{q}_i^T A \boldsymbol{q}_j \tag{2.74}$$

and

$$\boldsymbol{q}_j = \frac{\boldsymbol{r}_j}{\|\boldsymbol{r}_j\|} \tag{2.75}$$

This orthogonalized basis $\{\boldsymbol{q}_1, \boldsymbol{q}_2, \ldots, \boldsymbol{q}_m\}$ is an orthonormal basis of $\mathcal{K}^{m+1}(\boldsymbol{x}, A)$ and is called the Arnoldi basis. In the case of symmetric matrices A, as will be the case in this thesis, it is also called the Lanczos basis. The Lanczos algorithm is a method to compute an orthonormal basis of the Krylov space for a symmetric matrix A.

By multiplying the matrix A with the basis $Q_k = [\boldsymbol{q}_1, \boldsymbol{q}_2, \dots, \boldsymbol{q}_k]$, one obtains

$$Q_k^T A Q_k = Q_k^T Q_k H_k = H_k \tag{2.76}$$

with H_k the Hessenberg matrix, which in the case of a symmetric matrix simplifies to a tridiagonal matrix T_k. The matrix T_k has the following form

$$T_k = \begin{bmatrix} \alpha_1 & \beta_1 & & \\ \beta_1 & \ddots & \ddots & \\ & \ddots & \ddots & \beta_{k-1} \\ & & \beta_{k-1} & \alpha_k \end{bmatrix} \tag{2.77}$$

Thus, Equation (2.74) simplifies to

$$\boldsymbol{r}_j = A\boldsymbol{q}_j - \boldsymbol{q}_j \underbrace{(\boldsymbol{q}_j^T A \boldsymbol{q}_j)}_{\alpha_j} - \boldsymbol{q}_{j-1} \underbrace{(\boldsymbol{q}_{j-1} A \boldsymbol{q}_j)}_{\beta_{j-1}} \tag{2.78}$$

It is easy to compute the eigenvalues of T_k and it can be shown that the eigenvalues of the tridiagonal matrix T_k are also eigenvalues of A:

$$\begin{aligned} T_k \boldsymbol{s}_i^{(k)} &= \lambda_i^{(k)} \boldsymbol{s}_i^{(k)} \\ A Q_k \boldsymbol{s}_i^{(k)} &= Q_k T_k \boldsymbol{s}_i^{(k)} = Q_k \lambda_i^{(k)} \boldsymbol{s}_i^{(k)} \end{aligned}$$

with $\boldsymbol{s}_i^{(k)}$ the i-th eigenvector of the tridiagonal matrix T_k that has been computed at iteration k.

The pseudocode for the Lanczos algorithm is shown in Algorithm 1.

Algorithm 1 Lanczos algorithm

```
Require: Matrix $A \in \mathbb{R}^{N \times N}$, initialization $\boldsymbol{x}$
  $\boldsymbol{q} = \boldsymbol{x} / \|\boldsymbol{x}\|$; $Q_1 = [\boldsymbol{q}]$
  $\boldsymbol{r} = A\boldsymbol{q}$
  $\alpha_1 = \boldsymbol{q}^T r$
  $\boldsymbol{r} = \boldsymbol{r} - \alpha_1 \boldsymbol{q}$
  for $j = 2, 3, \dots$ do
      $\boldsymbol{v} = \boldsymbol{q}$; $\boldsymbol{q} = \boldsymbol{r} / \beta_{j-1}$; $Q_j = [Q_{j-1}, \boldsymbol{q}]$
      $\boldsymbol{r} = A\boldsymbol{q} - \beta_{j-1} \boldsymbol{v}$
      $\alpha_j = \boldsymbol{q}^T \boldsymbol{r}$
      $\boldsymbol{r} = \boldsymbol{r} - \alpha_j \boldsymbol{q}$
      $\beta_j = \|\boldsymbol{r}\|$
      if $\beta_j = 0$ then return $\left(Q \in \mathbb{R}^{N \times j}; \alpha_1, \dots, \alpha_j; \beta_1, \dots, \beta_{j-1}\right)$
      end if
  end for
```

The computational cost for one iteration of the Lanczos algorithm is independent of the index of the iteration. Each iteration requires one matrix-vector multiplication and $7N$ additional floating point operations, with N the length of the rows or columns of the symmetric matrix A [10].

There exist other variants of the Lanczos algorithm, like the block Lanczos algorithm [48] or the randomized block Lanczos algorithm [188], which are more memory efficient and have a faster convergence with respect to iterations.

2.3.4. The Stochastic Lanczos Algorithm

For problems where individual eigenvalues and their eigenvectors are not of main interest but rather the distribution of all the eigenvalues of the matrix, the regular Lanczos algorithm is too computationally expensive, especially in the case of very large neural networks where million to billion different eigenvalues would have to be computed using this method.

The stochastic Lanczos quadrature algorithm [71, 139] is a method for the approximation of the spectral density of very large matrices. The eigenvalue density spectrum is given by:

$$\Phi(t) = \frac{1}{N}\sum_{i=1}^{N}\delta(t-\lambda_i) \tag{2.79}$$

where N is the number of parameters in the network, λ_i is the i-th eigenvalue of the Hessian and $\delta(x)$ is the Dirac delta function given by

$$\delta(x) = \begin{cases} 1 & \text{if } x = 0 \\ 0 & \text{else} \end{cases}$$

In the stochastic Lanczos quadrature algorithm, the eigenvalue density spectrum is approximated by a sum of Gaussian functions:

$$\Phi_\sigma(t) = \frac{1}{N}\sum_{i=1}^{N}\mathcal{N}(t;\lambda_i,\sigma) \tag{2.80}$$

where

$$\mathcal{N}(t;\lambda_i,\sigma) = \frac{1}{\sigma\sqrt{2\pi}}\exp(-\frac{(t-\lambda_i)^2}{2\sigma^2}) \tag{2.81}$$

The stochastic Lanczos quadrature algorithm uses the regular Lanczos algorithm with full reorthogonalization [203] in order to compute eigenvalues and eigenvectors of the Hessian and to ensure orthogonality between the different eigenvectors. The Lanczos algorithm runs for m iterations with starting vector $\boldsymbol{\nu}_i$ and returns a tridiagonal matrix T_m, which is diagonalized:

$$T_m = ULU^T \tag{2.82}$$

By setting $\omega_i = (U_{1,i}^2)_{i=1}^m$ and $l_i = (L_{ii})_{i=1}^m$, the resulting eigenvalues and eigenvectors are used to estimate the true eigenvalue density spectrum:

$$\hat{\Phi}(t;\boldsymbol{\nu}_i,\sigma) = \sum_{i=1}^{m}\omega_i\mathcal{N}(t;l_i,\sigma) \tag{2.83}$$

This process is repeated k times with different starting vectors $\boldsymbol{\nu}_i$ for the Lanczos computations. The resulting approximation to the spectral density $\hat{\Phi}_\sigma(t)$ is

$$\hat{\Phi}_\sigma(t) = \frac{1}{k}\sum_{i=1}^{k} \hat{\Phi}(t;\boldsymbol{\nu}_i,\sigma) \tag{2.84}$$

The pseudocode for the stochastic Lanczos quadrature algorithm is shown in Algorithm 2.

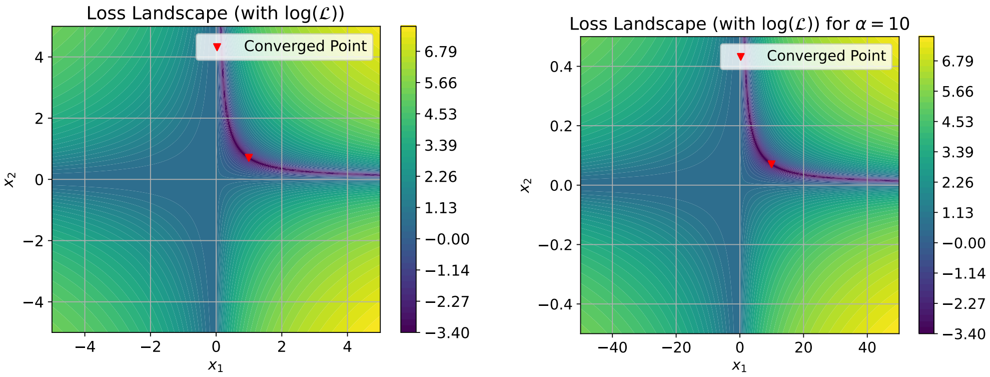
Algorithm 2 Stochastic Lanczos quadrature algorithm

```
Require: Number of iterations k, number of eigenvalues m
  Initialize Gaussian vectors $(\boldsymbol{\nu}_1, ..., \boldsymbol{\nu}_k)$
  for i from 1 to k do
      Run Lanczos with reorthogonalization for starting vector $\boldsymbol{\nu}_i$
      Obtain tridiagonal matrix $T_m$
      Diagonalize $T_m = ULU^T$
      Set $l_i = (L_{ii})_{i=1}^m$ and $\omega_i = (U_{1,i}^2)_{i=1}^m$
      Compute $\hat{\Phi}(t;\boldsymbol{\nu}_i,\sigma) = \sum_{i=1}^m \omega_i \mathcal{N}(t;l_i,\sigma)$
  end for
  Compute average $\hat{\Phi}_\sigma(t) = \frac{1}{k}\sum_{i=1}^k \hat{\Phi}(t;\boldsymbol{\nu}_i,\sigma)$
  return $\hat{\Phi}_\sigma(t)$
```

2.3.5. Computing Eigenvalues of Neural Networks

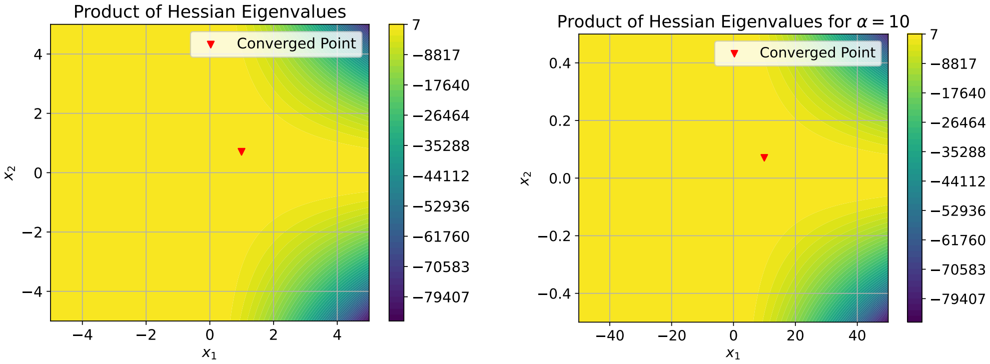
At critical points of the loss function, the determinant of the Hessian can be used to give insight into the curvature of the underlying loss landscape. Since the determinant of the Hessian is equal to the multiplication of all its eigenvalues, computing the eigenvalues can give important insight into the underlying critical point the network has converged to. If all the eigenvalues are positive, this corresponds to a positive determinant of the Hessian, which is turn means that the network has converged to a minimum with a given positive curvature. In general, local curvature can only be described by the determinant of the Hessian whenever the network has exactly converged to some extremum, i.e. its gradient is zero. There has been some debate about whether flat minima allow the network to generalize better, as has been argued by [89, 92, 109]. The authors of [55] show that the structure of neural network models results in many different symmetric configurations, where the model behaves exactly the same. By looking at different examples, the authors are able to make the minima of the neural network arbitrarily sharp or flat with regard to common sharpness measures, without changing the generalization properties of the network itself. They achieve this in a toy example by rescaling the parameters w_1, w_2 of a neural network given by $f(x^{(i)}; w_1, w_2) = \text{relu}(x^{(i)} w_1) w_2$ for input sample $x^{(i)}$. Rescaling them by $w_1' = \alpha w_1$ and $w_2' = w_2/\alpha$ for some value of the scalar α leaves the output of the network unaffected, while the perceived sharpness of a minimum changes a certain amount. This property of neural networks also applies to architectures which contain convolutions.

Generally, curvature in three-dimensional space is described by the Gaussian curvature [180], which is intrinsic to the surface. For a hypersurface given by $z = f(x,y)$,



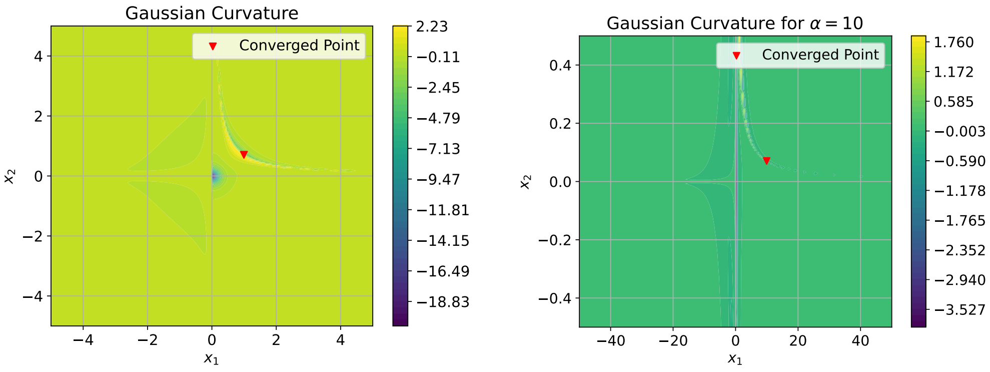
(a) *Contour plot of the function $\mathcal{L}$ for $\alpha = 1$. The converged point is located at $w^\star = (1, 0.7)$.*

(b) *Contour plot of the function $\mathcal{L}$ for $\alpha = 10$. The converged point is located at $w^\star = (10, 0.07)$.*



(c) *Determinant of the Hessian of $\mathcal{L}$. The eigenvalues of the converged point at $w^\star = (1, 0.7)$ are $\lambda_1 = 9.9$ and $\lambda_2 = 1.11 \times 10^{-4}$. The determinant of the Hessian at $w^\star$ is 0.0011.*

(d) *Determinant of the Hessian of $\mathcal{L}$ for $\alpha = 10$. The eigenvalues of the converged point at $w^\star = (10, 0.07)$ are $\lambda_1 = 660$ and $\lambda_2 = 1.67 \times 10^{-6}$. The determinant of the Hessian at $w^\star$ is 0.0011.*



(e) *Gaussian curvature of $\mathcal{L}$. The Gaussian curvature at the converged point $w^\star = (1, 0.7)$ is 0.0011.*

(f) *Gaussian curvature of $\mathcal{L}$ for $\alpha = 10$. The Gaussian curvature at the converged point $w^\star = (10, 0.07)$ is 0.0011.*

Figure 2.10.: *Depiction of the function $\mathcal{L}(x, y; w_1, w_2) = \sum_{i=1}^{n} (f(x^{(i)}; w_1, w_2) - y^{(i)})^2$. The neural network is given by $f(x^{(i)}; w_1, w_2) = relu(x^{(i)} w_1) w_2$. Plot (a) depicts the contours of the loss landscape of $\mathcal{L}$, while plot (b) depicts the same function with the weights rescaled by $w_1' = \alpha w_1$ and $w_2' = w_2/\alpha$ for $\alpha = 10$. Plots (c) and (d) show the determinant of the Hessian for $\alpha = 1$ and $\alpha = 10$, while plots (e) and (f) show the Gaussian curvature for $\alpha = 1$ and $\alpha = 10$.*

which is embedded in three-dimensional space, a Monge patch is given by $\mathfrak{g}(x,y) = (x,y,f(x,y))$. For this case, the Gaussian curvature κ is given by

$$\kappa = \frac{(\partial_{xx}f)(\partial_{yy}f) - (\partial_{xy}f)^2}{(1+(\partial_x f)^2 + (\partial_y f)^2)^2}. \tag{2.85}$$

This shows how at an extremum, where $\partial_x f = 0$ and $\partial_y f = 0$, the curvature is described by the determinant of the Hessian. This means that in order to have a measure of curvature inside minima that does not change upon rescaling of the weights, one has to multiply the eigenvalues of the Hessian in order to compute its determinant. It is also important to note that the determinant of the Hessian gives wrong curvature estimates at points where the gradient does not vanish. At these points the Gaussian curvature gives the correct value of the optimization surface. The Gaussian curvature changes when the weights are rescaled, which is due to the denominator in Equation (2.85). At points where the gradient vanishes, the this measure is unaffected by the rescaling of the weights.

A similar argument holds in higher dimensions, where the notion of the Gaussian curvature can be extended to a high-dimensional hypersurface. There, this measure is again an intrinsic property of the surface up to the sign [212, Corollary 23]. The important fact to keep in mind is that while other commonly used measures for curvature (certain eigenvalues or the trace of the Hessian) are not intrinsic to a hypersurface, the Gaussian curvature is. This means that isometric hypersurfaces have the same Gaussian curvature up to the sign [212].

Figure 2.10 shows the effects of rescaling the weights. One can observe how the overall shape of the loss landscape remains unchanged after the rescaling, while the eigenvalues of the Hessian change drastically. Also, while the product of all the eigenvalues remains constant, the difference between the Hessian determinant and the Gaussian curvature across the parameter space shows how these two measures differ substantially when the gradient is non-zero. This highlights some important aspects when considering the curvature of minima. Firstly, the value of different eigenvalues can be misleading and making judgements about the sharpness of certain minima based on these values can be ill-advised. Taking the product of all the eigenvalues removes this issue in theory, though practically the computation of the determinant of the Hessian for large neural networks is infeasible. Another issue is that this determinant only gives a correct curvature interpretation whenever the network has converged to an extremum, which can make the curvature estimation based on the Hessian away from those points incorrect.

On the practical side, the issues for computing the eigenvalues of neural networks are twofold. Firstly, computing the Hessian for any reasonably sized neural network is computationally intractable due to the number of parameters of the network. Secondly, most common methods for computing eigenvalues of matrices require the matrix to be stored explicitly.

The methods described in the preceding subsections solve these problems and make it possible to compute eigenvalues of neural networks efficiently. The main idea is to use the Lanczos method for eigenvalue computation without having to store the entire Hessian matrix. The Lanczos method is very fast for eigenvalue computation of symmetric matrices. Excluding the Hessian-vector multiplication, each iteration of the Lanczos algorithm for computing m eigenvalues has a computational complexity of $O(N)$. The remaining issue is that regular Hessian-vector multiplication has a complexity of $O(N^2)$. In total this amounts to a complexity of $O(mN^2)$, which is too high for large neural networks. The key trick in order to be able to compute eigenvalues of large neural networks is the use of the R-operator [175]. Using the R-operator, the Hessian-vector computation has a computational complexity of $O(N)$, which reduces the overall complexity of computing eigenvalues to $O(mN)$. For a small number of eigenvalues and eigenvectors, this computation is now feasible even for very large neural networks.

This section also introduced the stochastic Lanczos quadrature method, which allows computation of the full eigenvalue density spectrum of the Hessian. This method uses the Lanczos algorithm [120] and as [71] shows, one can approximate the full eigenvalue spectrum by only computing m Lanczos iteration steps and then diagonalizing the resulting $m \times m$ tridiagonal matrix. This procedure is repeated k times with different starting vectors. Using the resulting eigenvalues and eigenvectors of this tridiagonal matrix, one can approximate the full eigenvalue spectrum using Gaussians to high accuracy in only $O(Nmk)$. The first paper to apply this method for neural networks was [172].

2.3.6. Parallelization Techniques

Even though the Lanczos and the stochastic Lanczos algorithm are better suited for large matrices, very large neural networks are too big to compute eigenvalues in reasonable time. Thus, the need for parallelization techniques arises. For the Lanczos algorithm only the Hessian-vector product is needed instead of the full Hessian matrix. As mentioned before, this allows for the use of the R-operator to compute eigenvalues efficiently. Since the R-operator can essentially be treated like a forward and backward pass of a neural network, this opens up different possibilities to employ parallelization techniques.

By using the *Message-Passing Interface Standard* (MPI) [155], the Lanczos algorithm can be implemented in a data-parallel fashion, as was introduced in [68]. The dataset is distributed among the different workers and each worker accumulates its own Hessian-vector products by using the R-operator. After each worker has finished the computations, their resulting vector is sent to a master-worker where all the different vectors are summed by using the MPI_Reduce operation. The pseudocode is shown in Algorithm 3.

The data parallel approach is able to process far more samples at the same time and thus produce more accurate estimates of the true Hessian-vector product. Since the R-operator behaves very similar to the regular forward and backward propagation in neural networks, future research could look into model parallelism as well. A sketch of the data-parallel Lanczos method is depicted in Figure 2.11.

Novel Iteration-Parallel Stochastic Lanczos. This thesis also introduces another approach to parallelize the stochastic Lanczos quadrature algorithm, that proves to be much more scalable than the data-parallel approach up to a certain number of workers. The basic idea is to let each worker compute one iteration of the stochastic Lanczos quadrature algorithm for different initializations and then accumulate all the

Algorithm 3 Data-parallel Lanczos. Calculation of Hessian-vector products $H\boldsymbol{\nu}$ by using the R-operator

```
Require: Vector $\boldsymbol{\nu}$ and neural network model $\psi(\boldsymbol{w})$
  Set batch size $m$ and therefore divide the dataset $\mathcal{D}$ into $M = |\mathcal{D}|/m$ mini-batches
  for i from 1 to M do
      Worker $\mathcal{W}_j$ grabs batch $\mathbb{M}(i)$ and starts computing the Hessian-vector product $(H\boldsymbol{\nu})_i = \mathcal{R}(\psi(\boldsymbol{w},\mathbb{M}(i)),\boldsymbol{\nu})$
  end for
  The resulting Hessian vector product gets accumulated during this loop for each worker $(H\boldsymbol{\nu})_j = \frac{1}{|I_j|}\sum_{r \in I_j}(H\boldsymbol{\nu})_r$ where $I_j$ contains the indices of batches that were computed for worker $\mathcal{W}_j$
  Send all resulting $(H\boldsymbol{\nu})_j$ to the master-worker using $H\boldsymbol{\nu} = MPI\_Reduce((H\boldsymbol{\nu})_j)$
  return Hessian vector product over all samples $H\boldsymbol{\nu}$
```

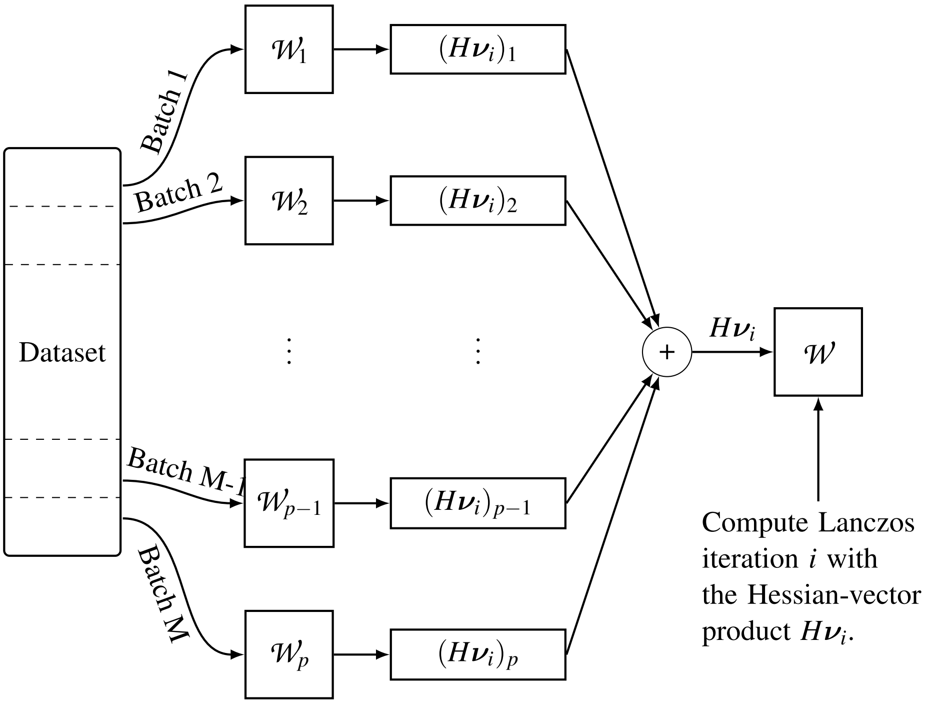

Dataset
Batch 1
Batch 2
Batch M-1
Batch M
$\mathcal{W}_1$
$\mathcal{W}_2$
$\mathcal{W}_{p-1}$
$\mathcal{W}_p$
$(H\boldsymbol{\nu}_i)_1$
$(H\boldsymbol{\nu}_i)_2$
$(H\boldsymbol{\nu}_i)_{p-1}$
$(H\boldsymbol{\nu}_i)_p$
+
$H\boldsymbol{\nu}_i$
$\mathcal{W}$
Compute Lanczos iteration i with the Hessian-vector product $H\boldsymbol{\nu}_i$.

Figure 2.11.: *Sketch of the data-parallel Lanczos method shown in Algorithm 3. There exist p different workers $\mathcal{W}_j$ which grab different batches from the dataset and compute the Hessian-vector product for their batch in parallel. These vectors are sent to the master worker $\mathcal{W}$, which uses the Hessian-vector product from all the workers in order to compute the current iteration of the Lanczos method. Afterwards, in the next iteration of the Lanczos method the master worker sends the next vector $\boldsymbol{\nu}_{i+1}$ to each worker. Every worker then computes the Hessian-vector product with the new vector $\boldsymbol{\nu}_{i+1}$. This last step has been omitted from the sketch in order to preserve clarity.*

results at the end on the master-worker. This approach is summarized in Algorithm 4. An illustration of this algorithm is shown in Figure 2.12.

Algorithm 4 Parallel stochastic Lanczos quadrature algorithm with MPI

```
Require: Number of iterations k, number of eigenvalues m
  Initialize Gaussian vectors $(\boldsymbol{\nu}_1, ..., \boldsymbol{\nu}_k)$ and split this set to $p$ different workers
  for $\boldsymbol{\nu}_i$ from the set assigned to each worker do
      Run Lanczos with reorthogonalization on worker $\mathcal{W}_j$
      Obtain tridiagonal matrix $T_m$
      Diagonalize $T_m = ULU^T$
      Set $l_i = (L_{ii})_{i=1}^m$ and $\omega_i = (U_{1,i}^2)_{i=1}^m$
  end for
  Each worker sends its computed $l_i$ and $\omega_i$ from all different initializations to the
  master-worker using MPI_Gatherv.
  Compute average on the master-worker $\mathcal{W}$:
  $\hat{\Phi}_\sigma(t) = \frac{1}{k} \sum_{j=1}^k \sum_{i=1}^m \omega_{j,i} \mathcal{N}(t; l_{j,i}, \sigma^2)$
  return $\hat{\Phi}_\sigma(t)$
```

Instead of parallelizing on the dataset as in the previous method, this method computes the different iterations in parallel. Since each iteration is independent of the others, the results of each worker are summed and averaged at the end in order to obtain the result.

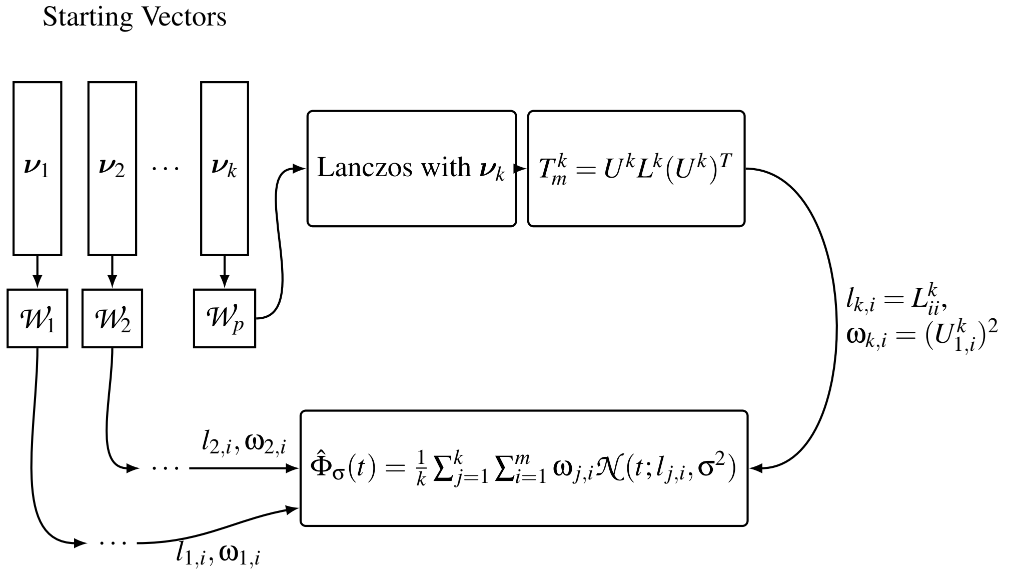

Starting Vectors
$\boldsymbol{\nu}_1$
$\boldsymbol{\nu}_2$
$\cdots$
$\boldsymbol{\nu}_k$
$\mathcal{W}_1$
$\mathcal{W}_2$
$\mathcal{W}_p$
Lanczos with $\boldsymbol{\nu}_k$
$T_m^k = U^k L^k (U^k)^T$
$l_{k,i} = L_{ii}^k$,
$\omega_{k,i} = (U_{1,i}^k)^2$
$\hat{\Phi}_\sigma(t) = \frac{1}{k} \sum_{j=1}^k \sum_{i=1}^m \omega_{j,i} \mathcal{N}(t; l_{j,i}, \sigma^2)$
$\cdots$ $l_{2,i}, \omega_{2,i}$
$\cdots$ $l_{1,i}, \omega_{1,i}$

Figure 2.12.: *Sketch of the parallel stochastic Lanczos quadrature algorithm with iteration parallelism. The k different iterations are distributed among the different workers $\mathcal{W}_j$. Each iteration uses a different starting vector v_s which is used as the starting vector for the Lanczos algorithm, that runs for m iterations. Afterwards, the tridiagonal matrix T_m^s is decomposed into $T_m^s = U^s L^s (U^s)^T$ using a LU-decomposition. Next, $l_{s,i}$ and $\omega_{s,i}$ are extracted from the L^s and U^s matrices and after all the workers have computed their corresponding values those are summed using Equation (2.84).*

Scalability. The scalability of a parallel algorithm is measured by measuring the speedup given by

$$S = \frac{T_1}{T_p}, \tag{2.86}$$

where T_1 is the time for one process and T_p the time for p processes [182]. In this experiment the scalability of the parallel stochastic Lanczos quadrature algorithm is measured and each node contains two Nvidia GTX 1080ti GPUs, which are assigned sequentially by first filling one node and then populating the next node with increasing number of ranks.

The standard deviation on the datapoints is calculated as follows:

$$\sigma_s = \sqrt{(\frac{1}{T_p})^2\sigma^2_{T_1} + (\frac{T_1}{T_p^2})^2\sigma^2_{T_p}} \tag{2.87}$$

Strong scaling is measured by keeping the problem size fixed and varying the number of GPUs. The parallelizable fraction of this parallel implementation is measured according to Amdahl's law [7]:

$$S = \frac{1}{(1-f)+f/p}, \tag{2.88}$$

where f is the parallelizable fraction of the implementation and p refers to the number of GPUs working in parallel on the problem. A scaling plot of both methods, the data-parallel and the iteration-parallel method, is depicted in Figure 2.13.

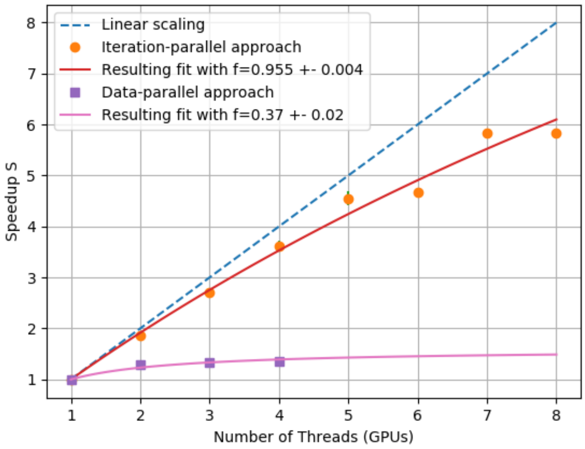


Figure 2.13.: *Speedup of the stochastic Lanczos quadrature algorithm parallelized with the data-parallel and iteration-parallel approach.*

One can see that fitting Formula 2.88 to the data, one obtains a parallelizable fraction of $f = 95.5 \pm 0.4\%$ for the iteration-parallel method. Fitting the model to the data-parallel approach yields a parallelizable fraction of $f = 37 \pm 2\%$, which is worse than the novel method.

One benefit of the iteration-parallel method, besides the speedup, is the much easier implementation than that of the data-parallel approach. But there also exist limitations to the iteration-parallel method. In the case where the number of nodes exceeds the number of iterations, the method is not able to scale anymore. This is one strength of the data-parallel method, which can scale in theory up to the number of samples in the dataset, which is typically very large in deep learning.

Future research could try and combine both methods into one, where different groups of workers compute one iteration in parallel and all workers of a specific group compute the individual Hessian-vector products using the data-parallel approach.

Chapter 3

Loss Surface Visualization

Training deep neural networks boils down to very high-dimensional and non-convex optimization problems. These are usually solved by a wide range of stochastic gradient descent methods like SGD or Adam. While these current training methods tend to work well in practice, many gaps exist in their theoretical understanding. Some examples of these theoretical gaps include convergence and generalization guarantees, which are induced by properties of the optimization surface (sometimes referred to as the loss landscape). In order to gain deeper insights into the optimization surface and how different optimizers navigate through it during training, a number of recent publications have proposed methods in order to visualize and analyze those optimization surfaces. However, some of these proposed methods have shortcomings that hinders their applicability. In this section, the topic of loss surface visualization is explained, together with the potential shortcomings of certain methods. Pseudocode is provided for efficient computation of loss surfaces and projecting training trajectories onto 2D planes. This thesis also introduces a method for visualizing loss surfaces and trajectories in a new way, improving upon the existing methods. Also, this section details how to parallelize the visualization computations and shows several examples of use cases. These methods, as well as their parallel versions have been released in the toolbox GradVis. GradVis is an open source library for efficient and scalable visualization and analysis of deep neural network loss landscapes in TensorFlow and PyTorch. It allows to plot 2D and 3D projections of optimization surfaces and trajectories, as well as high resolution second-order gradient information for large neural networks.

3.1. Overview

The main idea behind loss surface visualization is to project the high-dimensional parameter space down onto a lower-dimensional plane. The goal is to find a projection that is able to capture a lot of information of the high-dimensional space. In general, the objective is to find a mapping

$$\begin{aligned}&\phi : \mathbb{R}^k \longrightarrow \Theta,\\&\text{such that } \mathcal{L} \circ \phi : \mathbb{R}^k \longrightarrow \mathbb{R}\end{aligned}$$

such that the composition of ϕ with the loss function $\mathcal{L}$ results in a lower-dimensional domain that can be visualized. The dimensionality of the domain of ϕ usually has a value of $k \in \{1,2\}$.

A simple example of a 1-dimensional path is a linear interpolation between two points in parameter space.

$$\phi(t) = (1-t)\boldsymbol{w}^{(1)} + t\boldsymbol{w}^{(2)} \tag{3.1}$$

for $t \in [0,1]$, where $\boldsymbol{w}^{(1)}$ and $\boldsymbol{w}^{(2)}$ are two points in parameter space. Figure 3.1 shows an example of projecting a 2-dimensional plane onto a 1-dimensional line.

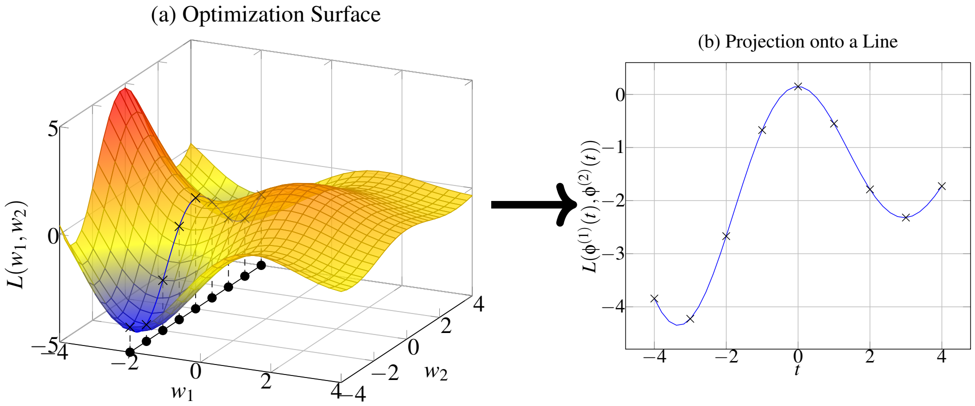


Figure 3.1.: *Toy example of an optimization surface with a projection onto a 1-dimensional line. Plot (a) shows the function*

$$L(w_1, w_2) = \exp^{-0.25w_1 - 0.1w_2}(\sin(w_1) + \cos(w_2))$$

together with a line that is parametrized in the following way:

$$\phi(t) = \begin{pmatrix} -2 \\ t \end{pmatrix}$$

Note that in this example w_1 and w_2 denote the different dimensions of the parameters space and not different points in the parameter space as used in this chapter up to this point. Plot (b) depicts the optimization function along this parametrized line for $t \in [-4,4]$.

An example for a visualization onto two dimensions is given by a plane that is spanned by three points in parameter space

$$\phi(\alpha,\beta) = \boldsymbol{w}^{(1)} + \alpha(\boldsymbol{w}^{(2)} - \boldsymbol{w}^{(1)}) + \beta(\boldsymbol{w}^{(3)} - \boldsymbol{w}^{(1)}) \tag{3.2}$$

where $\boldsymbol{w}^{(1)}$, $\boldsymbol{w}^{(2)}$ and $\boldsymbol{w}^{(3)}$ are points in parameter space.

First attempts that aimed to visualize loss landscapes were presented by [73]. The authors drew a line from the initialization point to the converged minimum. Along this line, the loss is calculated and plotted. They were able to show that the loss along this line follows a monotonically decreasing path. The authors of [131] introduced a new method to visualize the loss landscape. Their main contributions include the use of filter-wise normalization to combat the scaling invariance of the neural network weights, using a PCA [64] to find meaningful directions in weight space and reducing the problem onto a 2D plane instead of just a 1D line. The projection of the trajectory onto a 2D plane was performed by spanning the plane in parameter space around the converged point $\boldsymbol{w}^*$ and taking the loss $\mathcal{L}(\boldsymbol{w}^* + \alpha \boldsymbol{b}_1^* + \beta \boldsymbol{b}_2^*)$ at each point on this plane, with $\boldsymbol{b}_1^*$ and $\boldsymbol{b}_2^*$ the directional vectors from PCA that contain most of the information of the trajectory of the neural network during training.

Filter-Wise Normalization As previously stated in Section 2.3.5, [55] show how non-negative homogeneous activation functions lead to scale invariance in deep neural networks. Some of the commonly used activation functions that fall into this category are ReLUs, leaky ReLUs [150] and maxout-networks [74]. Using this property, the authors are able to construct arbitrarily sharp minima without changing the generalization of the network.

In deep neural network visualization, this scale invariance of the network poses a problem. When comparing two different minima, their apparent sharpness should be independent of the individual weight scaling. To deal with this issue, [131] introduce filter-wise normalization. Their goal is to rescale the directional vector in order to have the same norm for each convolutional filter.

Filter-wise normalization takes some directional vector in weight space $\boldsymbol{b}$ and transforms it using

$$\boldsymbol{b}_{i,j}^* = \frac{\boldsymbol{b}_{i,j}}{\left\|\boldsymbol{b}_{i,j}\right\|_F} \left\|\boldsymbol{w}_{i,j}\right\|_F \tag{3.3}$$

where the indices i and j denote the j-th filter in the i-th convolution of the network. The filters are normalized using the Frobenius norm. Normalizing the directional vectors using the filter-wise normalization method scales the directional vector so that it locally has the same magnitude as the corresponding filter. This prevents cases in which the directional vector is much larger in certain dimensions compared to the corresponding weight of the neural network or vise versa.

3.2. Principal Component Analysis in High-Dimensional Spaces

Principal component analysis [64] is a dimensionality reduction technique. PCA is a method that aims to represent data in a subspace of lower dimension with as little loss of information as possible.

Given data $\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_l \in \mathbb{R}^d$, write the matrix containing the datapoints as $X = (\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_l)$. Then the PCA is given as

$$\begin{aligned}&\max_{\Theta \in \mathbb{R}^{d \times k}} \textstyle\sum_{j=1}^{k} \boldsymbol{\theta}_j^T X X^T \boldsymbol{\theta}_j,\\ &\text{s.t. } \boldsymbol{\theta}_i \perp \boldsymbol{\theta}_j \text{ for } i \neq j \text{ and } \|\boldsymbol{\theta}_1\| = ... = \|\boldsymbol{\theta}_k\| = 1.\end{aligned}$$

This can be solved via *singular value decomposition* (SVD) [70] of the scatter matrix $S_l = XX^T$. Note that the scatter matrix is a symmetric matrix and is therefore diagonalizable. Sorting the resulting eigenvectors with respect to their corresponding eigenvalue in descending order, one chooses the k eigenvectors corresponding to the biggest eigenvalues in order to form the matrix $\Theta = (\boldsymbol{\theta}_1, \dots, \boldsymbol{\theta}_k)$ with $\Theta \in \mathbb{R}^{d \times k}$. This matrix of eigenvectors is used to transform the initial samples onto the new, lower dimensional subspace

$$\boldsymbol{s}_i = \Theta^T \boldsymbol{x}_i \tag{3.4}$$

with $\boldsymbol{s}_i \in \mathbb{R}^k$ the transformed sample in the new subspace.

The authors of [131] use this approach in order to find a plane that captures most of the information of the trajectory of the neural network during training. By taking different points in parameter space along the training path $X = (\boldsymbol{w}^{(1)}, \boldsymbol{w}^{(2)}, ..., \boldsymbol{w}^{(l)})$ and then applying the PCA in order to project these samples down to $k = 2$ dimensions, the authors find that these two components contain around 80% of the variance of the original trajectory.

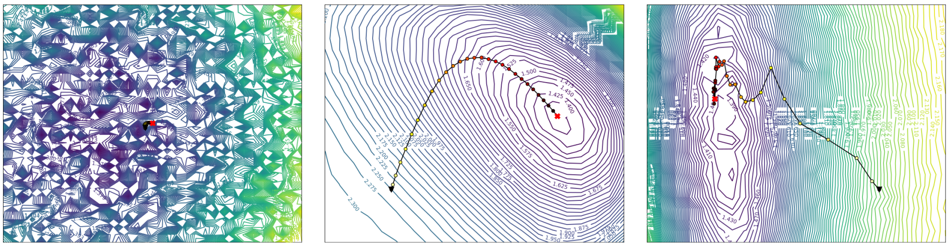
According to [135], the time complexity of performing a PCA on the dataset $X \in \mathbb{R}^{d \times l}$ using SVD is $O\left(2dl^2 + l^3 + l + dl\right)$. In the context of deep neural networks, even though the dimension of the individual datapoints is extremely large compared to the number of datapoints ($l << d$), the PCA computation is still feasible.

Recently, this approach of using PCA in order to find good directions in parameter space has been criticised by [9]. The authors show that using PCA on high-dimensional random walks always results in Lissajous trajectories. They argue that the training trajectory, if plotted using PCA, always performs the same patterns, thereby rendering the information one can extract from the trajectory useless. In the limit of infinite dimensions, the PCA of a random walk with arbitrary noise distribution will generate a Lissajous curve when projected down onto any two PCA components. This is also shown with a random walk with momentum, where the first PCA component contains around 60% of the variance, while 80% of the variance is in the first two components. This is similar to what is being observed using the method of [131] for finding suitable directions in the parameter space of neural networks. Thus, even though this method apparently captures a lot of information of the training path in the projected subspace, in reality this can be deceiving. The fact that this method yields very similar Lissajous trajectories for every training run indicates that no real information of interest is captured. Thus, another method is needed in order to visualize the trajectories taken during training in a meaningful way.

3.3. Loss Surface Visualization through Neural Network Eigenvectors

In order to solve the issue posed by using PCA on the high-dimensional network weights (as discussed by [9]), this thesis proposes using different eigenvectors of the Hessian of the converged neural network as directional vectors in order to plot the loss landscape together with the trajectory. The eigenvectors are computed using the Lanczos algorithm combined with the R-operator in order to efficiently compute the Hessian-vector product.

Figure 3.2 compares three different methods used to find directions in the loss surface in order to plot the trajectory. In the first method, two randomly initialized vectors are chosen, with entries drawn from a normal distribution. The resulting plot is shown in Figure 3.2a. Next, PCA is used on a set of points in the training trajectory of the neural network in order to extract the two directions with the highest variance. The resulting visualization is shown in Figure 3.2b. Lastly, the proposed method of choosing eigenvectors to plot the trajectory is presented in Figure 3.2c. Here the eigenvectors corresponding to the highest two eigenvalues of the converged point are chosen.



(a) *Loss landscape with trajectory along two random directions for LeNet on CIFAR10.*

(b) *Loss landscape with trajectory of the same training run, visualized along two PCA directions with highest variance for LeNet on CIFAR10, as suggested by [131].*

(c) *Loss landscape with trajectory of the same training run, visualized along eigenvectors corresponding to the two highest eigenvalues for LeNet on CIFAR10.*

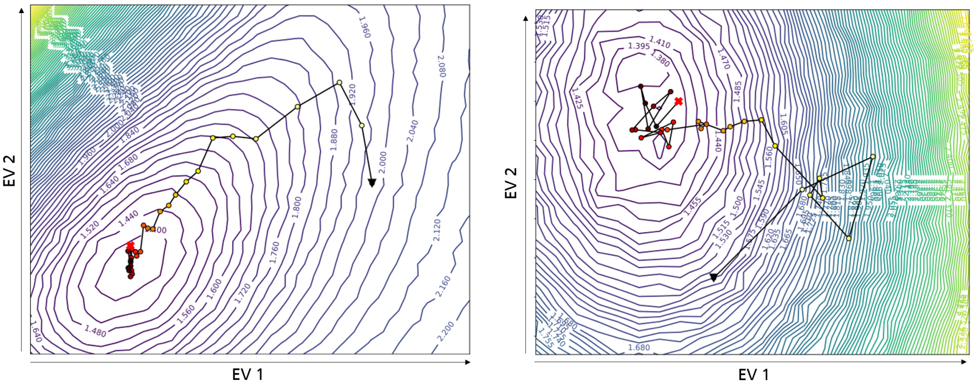
Figure 3.2.: *Depiction of the loss landscape of LeNet-5 on CIFAR-10 using three different visualization methods. Plot (a) depicts the loss landscape along two random directions, plot (b) depicts the use of PCA for finding directions in parameter space and plot (c) depicts the loss landscape along the two eigenvectors which belong to the biggest eigenvalues of the converged point.*

The random direction method shown in Figure 3.2a provides little to no information or insight into the neural network training process. All the training points of the trajectory are projected onto a small point in space and the training path barely moves from its initial position. The loss landscape in this projected region appears to be very noisy and relatively flat.

On the contrary, using PCA to find meaningful directions shows the trajectory spiraling into the minimum, which appears to be elliptical. As mentioned in Section 3.2, the path of this trajectory is entirely predictable, as it resembles one where the PCA is taken on a high-dimensional random walk with drift ([9]). Thus, even though it seems like the trajectory offers some insight into the training procedure, in reality this method will always result in a similar trajectory when applied to neural networks that have been trained using common stochastic gradient methods, thereby rendering this method useless.

The method of using eigenvectors on the other hand shows how the trajectory initially moves along the gradient of the loss and after reaching the valley it slowly creeps towards the minimum. This method also makes it possible to pick interesting eigenvalues that are more challenging for the optimizer. These could for example be zero or negative eigenvalues, where the network seems to struggle or even ”fail” during training. This allows monitoring how the training trajectory behaves in these directions.

An example of how different eigenvectors can reveal different behaviors of the neural network is shown in Figure 3.3. Figure 3.3a depicts the loss landscape visualization along the two eigenvectors corresponding to the two biggest eigenvalues of the converged point. On the other hand Figure 3.3b shows the same training run, but visualized along the eigenvectors corresponding to the biggest and smallest eigenvalue of the converged point.



(a) *Loss landscape of ResNet-32 trained on CIFAR-10 using Adam. The two directional vectors are the eigenvectors corresponding to the two biggest eigenvalues of the converged point.*

(b) *Loss landscape of ResNet-32 trained on CIFAR-10 using Adam. The two directional vectors are the eigenvectors corresponding to the biggest and the smallest eigenvalue of the converged point.*

Figure 3.3.: *Loss landscape of ResNet-32 for different choices of eigenvectors. Here a ResNet-32 architecture is trained on CIFAR-10 using Adam. A learning rate of 0.001 was used and the network was trained for 30 iterations. Both plots were generated using 20 % of the training samples. The plots have a grid size of 50 on each side. The black triangle indicates the network initialization, while the red cross denotes the converged point of the network.*

As shown in Figure 3.3a, the network finds itself in a locally convex optimization problem along those two eigenvectors. On the other hand, Figure 3.3b shows that the network initially struggles to descent into the minimum, which can be due to the fact that the loss landscape becomes much more noisy for small batch sizes. Both of these Figures compute the loss landscape visualization for 20% of all the training samples, while the network only sees a small fraction of those at each training step. Nonetheless, when the network converges into the minimum in Figure 3.3b, there are two areas above and below the converged point that have a smaller value then the converged point. This explains the saddle point structure one would expect, given the projection along the eigenvectors corresponding to the biggest and smallest eigenvalue. In this case the network seems to have converged to a saddle point inside a much larger minimum.

Computing the Loss Surface Visualization Computation of the two-dimensional loss surface visualization starts by choosing two directional vectors $\boldsymbol{d}_1, \boldsymbol{d}_2$ in weight space together with a point $\boldsymbol{w}^{(l)}$ in weight space that anchors the plane with $\phi(0,0) = \boldsymbol{w}^{(l)}$. Next, the directional vectors are normalized using filter-wise normalization.

In case the trajectory is visualized as well, each point of the trajectory has to be projected onto the plane as well. Thus, for each point $\boldsymbol{w}^{(i)}$ in the trajectory, the following equation has to be solved for α_i and β_i:

$$\phi(\alpha_i, \beta_i) = \boldsymbol{w}^{(i)} \tag{3.5}$$

which in the case of a two-dimensional plane results in:

$$\alpha_i \boldsymbol{b}_1^* + \beta_i \boldsymbol{b}_2^* = \boldsymbol{w}^{(i)} - \boldsymbol{w}^{(l)} \tag{3.6}$$

where $\boldsymbol{b}_1^*$ and $\boldsymbol{b}_2^*$ are the filter-normalized directional vectors $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$. The last step is to discretize the plane into a two dimensional grid. Each point of the grid refers to a specific value of (α_i, β_j). At each point of the grid the corresponding loss values are computed using $z_{i,j} = \mathcal{L}(\phi(\alpha_i, \beta_j))$. The pseudocode is described in Algorithm 5.

Algorithm 5 Calculate the loss landscape for a neural network with loss $\mathcal{L}$ for the l weights along the trajectory $(\boldsymbol{w}^{(1)}, ..., \boldsymbol{w}^{(l)})$. The resulting 2D landscape has N points along each axis on the grid

```
Require: Weights $(\boldsymbol{w}^{(1)}, ..., \boldsymbol{w}^{(l)})$
  Calculate the directional vectors (e.g. using PCA)
  $\boldsymbol{b}_1, \boldsymbol{b}_2 \leftarrow \mathrm{PCA}(\boldsymbol{w}^{(1)} - \boldsymbol{w}^{(l)}, ..., \boldsymbol{w}^{(l-1)} - \boldsymbol{w}^{(l)})$
  Use filter-wise normalization
  $\boldsymbol{b}_1^*\ \boldsymbol{b}_2^* \leftarrow \mathrm{Normalize}(\boldsymbol{b}_1, \boldsymbol{b}_2)$
  Calculate the coefficients of the training path
  $\alpha_i, \beta_i \leftarrow$ Solve for $\alpha_i \boldsymbol{b}_1^* + \beta_i \boldsymbol{b}_2^* = \boldsymbol{w}^{(i)} - \boldsymbol{w}^{(l)}$ for each i
  Calculate the loss values of the path points
  $z_i \leftarrow L(\alpha_i \boldsymbol{b}_1^* + \beta_i \boldsymbol{b}_1^* + \boldsymbol{w}^{(l)})$
  Calculate the loss values for each point on the grid
  Make grid of N samples going from $min(\alpha_i)$ to $\max(\alpha_i)$ for the x-direction and $\min(\beta_i)$ to $\max(\beta_i)$ for the y-direction of the grid
  for i from 0 to N do
      for j from 0 to N do
          $p_i = i(\max(\alpha) - \min(\alpha))/N$
          $q_j = j(\max(\beta) - \min(\beta))/N$
          $z_{i,j} \leftarrow L(p_i \boldsymbol{b}_1^* + q_j \boldsymbol{b}_2^* + \boldsymbol{w}^{(l)})$
      end for
  end forreturn $(z_{1,1}, ..., z_{N,N})$
```

3.4. Parallelization of Loss Surface Visualization

To speed up computation of the loss surface visualization plots, the visualization method can be trivially parallelized. This is achieved by assigning parts of the evaluation grid to different workers. After each worker has performed the computation of the assigned subgrid, the master-worker collects their values using MPI_Gatherv. The pseudocode for the parallelized version is shown in Algorithm 6 and a visualization of this method is illustrated in Figure 3.4.

Algorithm 6 Parallel visualization method. Calculate the loss landscape for a neural network with loss $\mathcal{L}$ for the l weights along the trajectory $(\boldsymbol{w}^{(1)},...,\boldsymbol{w}^{(l)})$. The resulting 2D landscape has N points along each axis on the grid

```
Require: Weights $(\boldsymbol{w}^{(1)},...,\boldsymbol{w}^{(l)})$
  Calculate the directional vectors (e.g. using PCA)
  $\boldsymbol{b}_1, \boldsymbol{b}_2 \leftarrow$ PCA($\boldsymbol{w}^{(1)} - \boldsymbol{w}^{(l)},...,\boldsymbol{w}^{(l-1)} - \boldsymbol{w}^{(l)}$)
  Use filter-wise normalization
  $\boldsymbol{b}_1^* \ \boldsymbol{b}_2^* \leftarrow$ Normalize($\boldsymbol{b}_1, \boldsymbol{b}_2$)
  Calculate the coefficients of the training path
  $\alpha_i, \beta_i \leftarrow$ Solve for $\alpha_i \boldsymbol{b}_1^* + \beta_i \boldsymbol{b}_2^* = \boldsymbol{w}^{(i)} - \boldsymbol{w}^{(l)}$ for each i
  Calculate the loss values of the path points
  $z_i \leftarrow L(\alpha_i \boldsymbol{b}_1^* + \beta_i \boldsymbol{b}_2^* + \boldsymbol{w}^{(l)})$
  Calculate the loss values for each point on the grid
  Make grid of N samples going from $min(\{\alpha_i\}_{i\in\{1,...,N\}})$ to $\max(\{\alpha_i\}_{i\in\{1,...,N\}})$ for x and $min(\{\beta_i\}_{i\in\{1,...,N\}})$ to $\max(\{\beta_i\}_{i\in\{1,...,N\}})$ for y
  Split grid into subgrids according to number of workers p: $X_i$,$Y_i$ for i = 1,...,p
  Assign part of the grid to each worker
  for $x$ in $X_i$ do
    for $y$ in $Y_i$ do
      $z_{x,y} \leftarrow L(x\boldsymbol{b}_1^* + y\boldsymbol{b}_2^* + \boldsymbol{w}^{(l)})$
    end for
  end for
  Each worker has set of values $Z_i$
  $Z \leftarrow$ MPI_Gatherv($Z_i$)
  return Z = $(z_{1,1},...,z_{N,N})$
```

In order to estimate the computation time, consider again the algorithm presented in Algorithm 5. On a square grid, the toolbox evaluates the neural network on the training samples N^2 times, with N the number of points on the grid along each direction. At each grid point, the neural network has to evaluate n_b batches, where the computation of each batch takes time $T_{inference}$. Therefore, the evaluation time for all points on the grid is of the order of $\alpha + N^2 T_{inference} n_b$, with some overhead α for the computation of the PCA components, as well as for performing the filter-wise normalization. For large enough N this overhead is assumed to be small in comparison.

For example, performing experiments with a ResNet-32 for a batch size of 256 and 2 batches resulted in $\alpha = 8s$ and $T_{inference} = 0.005s$. Table 3.1 shows, that the overhead α stays constant while computation time for the loss landscape scales like N^2.

Scalability As already shown in Section 2.3.6, scalability is measured by measuring the speedup

$$S = \frac{T_1}{T_p}, \tag{3.7}$$

with T_1 the time for one process and T_p the time for p processes. The measurements

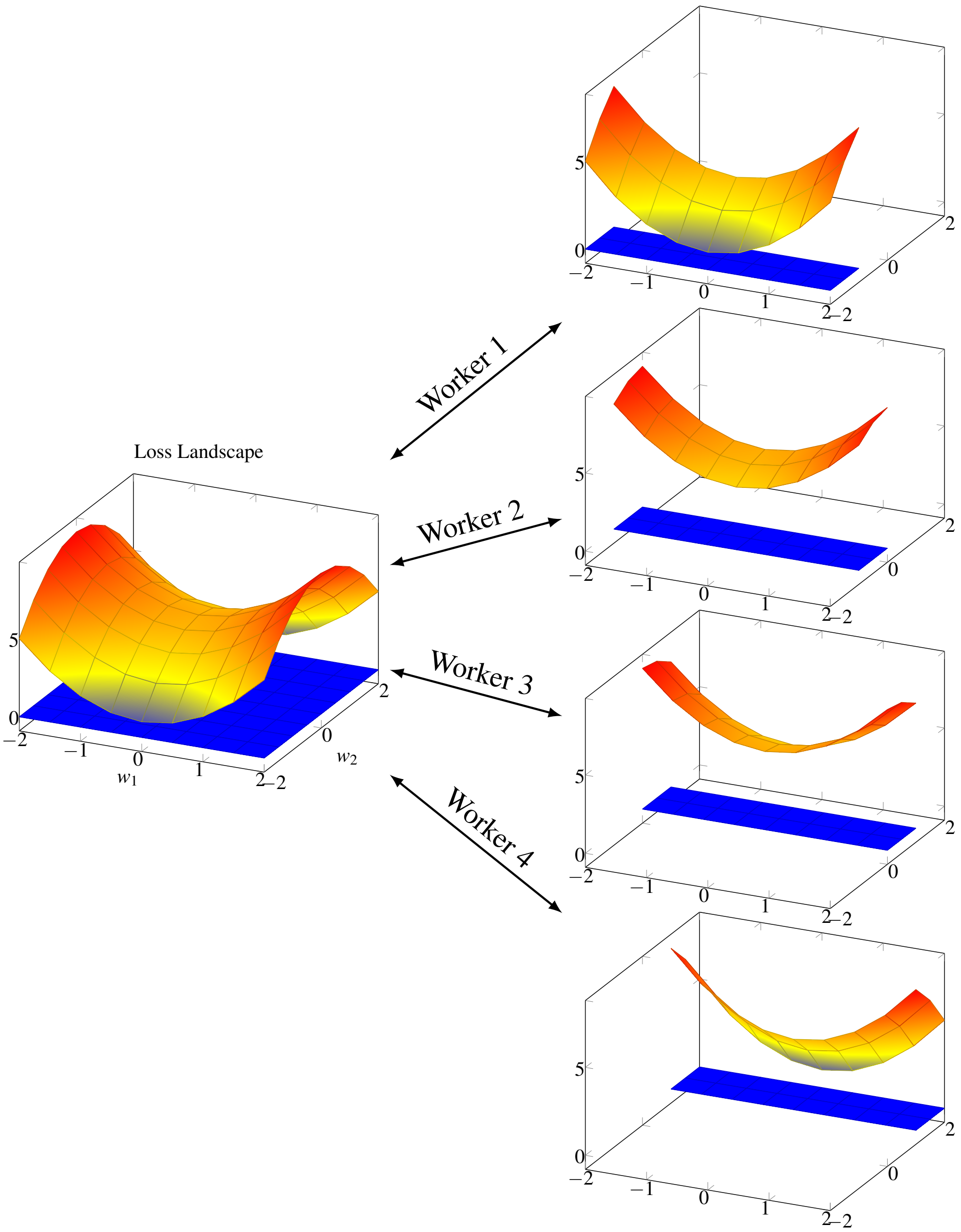


Figure 3.4.: *Illustration of the parallel visualization method. The grid is shown in blue and on the left hand side the entire visualization is depicted. The parallel visualization method splits the grid into multiple subgrids which are distributed among the workers and each worker computes only part of the entire grid. Afterwards all the workers send their computations back to a master-worker.*

for parallel visualization were performed on a GPU cluster where each node contains two Nvidia GTX 1080ti GPUs which are assigned sequentially by first filling one node and then populating the next node with increasing number of ranks.

For the parallel visualization method presented in Algorithm 6, measuring the speedup and fitting Amdahl's law from Equation (2.88) to the recorded datapoints yields Figure 3.5.

Table 3.1.: *Timing of loss landscape calculation and overhead for different grid sizes N*

Timing in seconds	N=2	N=10	N=50	N=100
$N^2 T_{inference} n_b$	0.51	12.72	320	1890
α	8.21	8.01	8.09	8.11

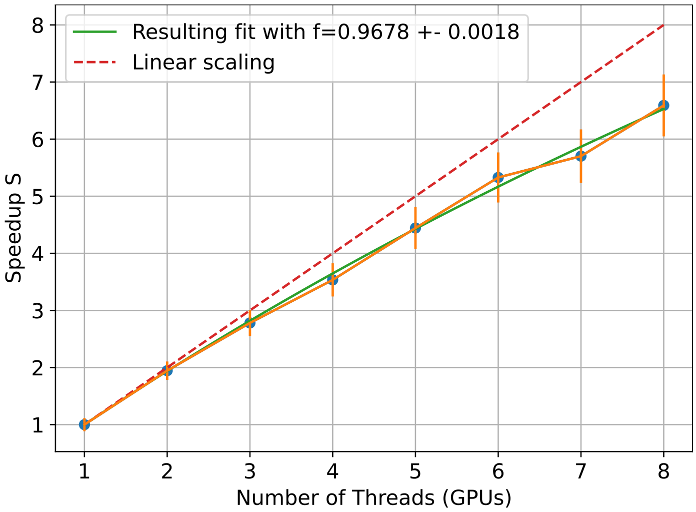
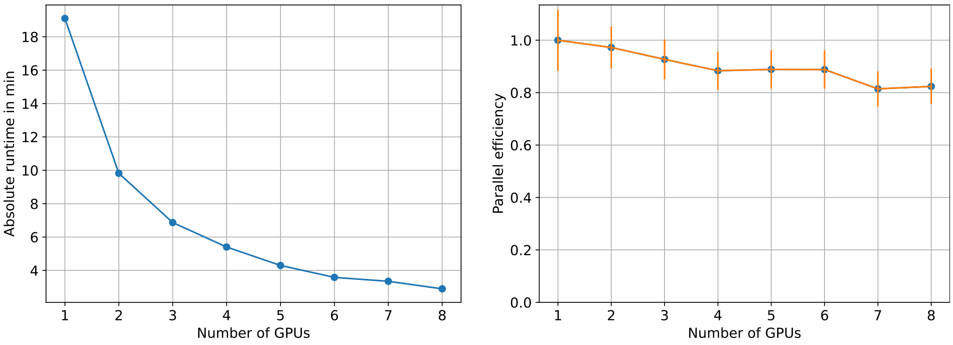


Figure 3.5.: *Strong scaling plot for the parallel visualization algorithm.*

Fitting Formula 2.88 to the data results in a parallelizable fraction of $f = 96.78 \pm 0.18\%$. The parallelizable fraction is very high, which is likely due to the trivially parallelizable nature of the algorithm. Additionally, Figure 3.6 plots the absolute runtime in minutes together with the parallel efficiency of the algorithm.



(a) *Plot of the absolute runtime for the parallel visualization algorithm.* **(b)** *Efficiency plot for the parallel visualization algorithm.*

Figure 3.6.: *Depiction of the absolute runtime as well as the efficiency for the parallel visualization algorithm. Experiments were performed on a ResNet-32 network trained on CIFAR-10.*

As depicted in Figure 3.6*a*, the absolute runtime drops from around 19 minutes on one GPU to around 3 minutes for 8 GPUs in parallel. The parallel efficiency shown in Figure 3.6*b* drops down to a value of about 0.8 for 7 and 8 GPUs.

3.5. Experiments

In this section several examples are presented that investigate the behavior of different optimizers and networks using the method of eigenvectors of the Hessian for visualization. Using the aforementioned parallelization techniques, this allows computation at each iteration of a neural network at a reasonable timeframe. Also, in combination with the spectral densities, this offers better insight into the training dynamics of deep neural networks.

3.5.1. SGD with Momentum

In this experiment, the loss landscape is computed together with the training trajectory of a LeNet-5 architecture that has been trained on CIFAR-10. The loss landscape visualization as well as the eigenvalue spectra are computed for each of the 196 iterations of the first epoch. During training, the batch size is set to $|\mathbb{M}(k)| = 256$ and the SGD optimizer uses a learning rate of $\varepsilon = 0.001$ and momentum of $\rho = 0.9$. After each iteration the parameters of the neural network model are saved onto the hard drive.

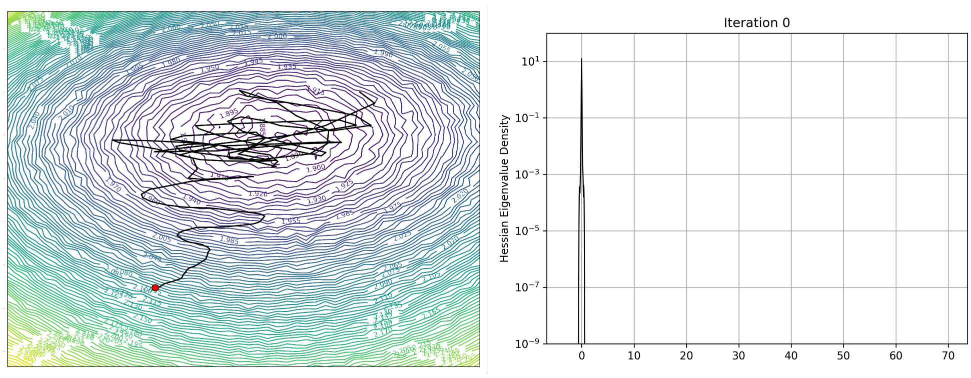
For the loss landscape visualization a square grid with size $N = 50$ on each side is chosen, with an additional border of 40% around the trajectory. For each point on the grid 20% of all the samples in the training set are used and the loss landscape is plotted along the two highest eigenvalues of the Hessian at the converged point. The eigenvalue density plots are computed using the stochastic Lanczos quadrature algorithm, with $k = 10$ iterations and $m = 80$ iterations of the Lanczos algorithm. These computations were done using 20% of the CIFAR-10 samples as well.

The resulting plots are shown in Figure 3.7. Note that these only show a select number of plots out of the 196 generated [1]. Figure 3.7a depicts the network during initialization. From a global perspective one can observe from the eigenvalue spectrum that at initialization the network finds itself inside a relatively flat saddle point where most eigenvalues are close to zero. Looking at the loss landscape in the direction of the two eigenvectors which correspond to the two biggest eigenvalues of the Hessian at the converged point, it can be seen that the network is close to a local minimum in this subspace. The network struggles to escape this flat saddle point and only manages to converge towards minima in certain directions after 89 iterations, which is depicted in Figure 3.7b. In this Figure, the loss landscape plot shows that the network has converged closer toward the local minimum. The density spectrum reveals that the bulk of eigenvalues around zero is not as concentrated anymore and the first eigenvalues are starting to separate from it, which indicates that the network is starting to converge into minima in certain directions.

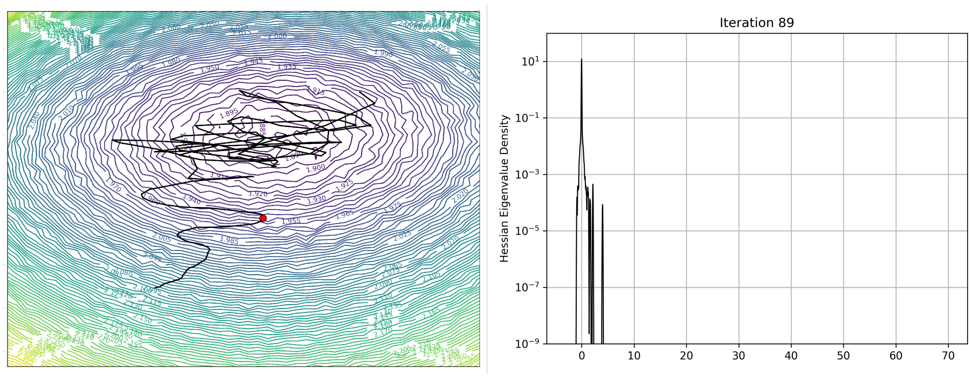
The last iteration of the first epoch is depicted in Figure 3.7c. The density spectrum reveals how there are several distinct eigenvalues bigger than zero that have separated from the bulk, which indicates that the network has converged into a minimum in those directions. Looking at the bulk at zero, this plot reveals that there are several

---

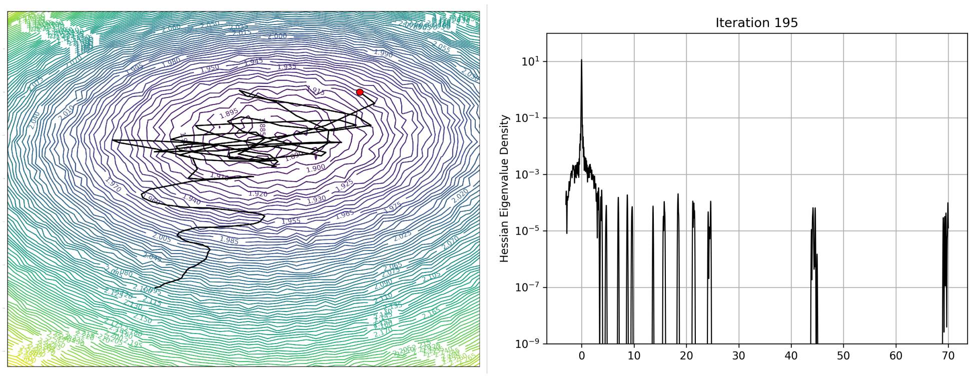
[1] A video of the full trajectory can be viewed on https://youtu.be/0AKSjp-SHlo



(a) *Loss landscape and eigenvalue density plot of LeNet at initialization.*



(b) *Loss landscape and eigenvalue density plot of LeNet. Same training run at 89 iterations.*



(c) *Loss landscape and eigenvalue density plot of LeNet. Same trainig run at 195 iterations.*

Figure 3.7.: *Loss landscape and Hessian eigenvalue density for three out of 195 iterations. Here a LeNet architecture is trained in CIFAR10 with using SGD with momentum. A learning rate of 0.01 was used as well as a momentum of 0.9. Both plots were generated using 20 % of the training samples. The visualization is depicted along two eigenvectors corresponding to the two highest eigenvalues of the Hessian. The plot has a grid size of 50 on each side.*

negative eigenvalues present, therefore the network has converged to a maximum in these dimensions and overall it has converged to a saddle point. Looking at the loss landscape, the network has converged into the local minimum of the subspace. The trajectory oscillates around the minimum chaotically, which is probably a result of the fixed learning rate and the momentum which was used in the SGD optimizer. It is also interesting to note that the loss landscape in this visualization is convex around the entire trajectory and not only locally at the converged point.

3.5.2. Adam

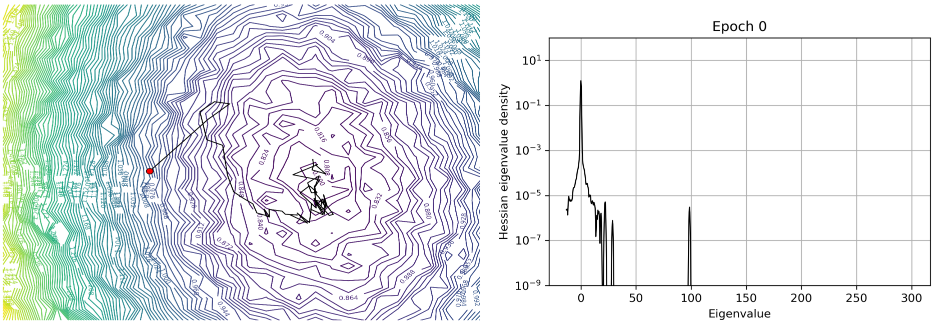
This section depicts the loss landscape of a ResNet-32 trained with the Adam optimizer on CIFAR-10. Contrary to the SGD optimizer in Section 3.5.1, the Adam optimizer has an adaptive learning rate. The network is trained for 60 Epochs and the Adam optimizer uses a learning rate of $\varepsilon = 6 \times 10^{-4}$ and momentum values of $\beta = (0.9, 0.999)$. The loss surface visualization uses 15% of all the training samples and visualizes the trajectory along the two eigenvectors that correspond to the biggest eigenvalues at the converged point. The Hessian eigenvalue density spectrum is computed using 5% of all the training samples.

The Hessian eigenvalue spectrum for epochs 0, 6 and 59 together with the loss landscape visualization along the eigenvectors corresponding to the two big eigenvalues is shown in Figure 3.8. The trajectory in the visualization does not oscillate as much as in the SGD case. Once the network has converged inside the minimum, the adaptive learning rate gets smaller.

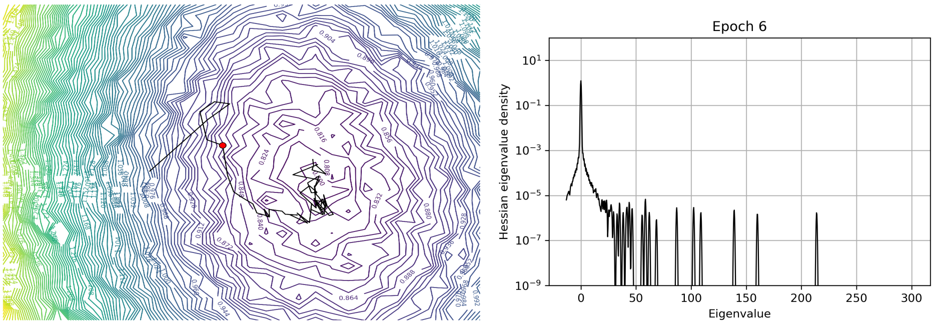
In Figure 3.8a the network initialization is shown. Looking at the visualization plot at the left, it appears that the network is located on the edge of a convex optimization surface. The eigenvalue density shows that there is one distinct eigenvalue separated from the bulk at a value of 100. Figure 3.8b shows the network at epoch 6. Now the network has converged towards the minimum of the convex optimization surface, as shown in the visualization on the left. The eigenvalue density plot at the right shows that multiple eigenvalues have separated from the bulk, with the highest one reaching a value of around 210. Lastly, the converged point in Figure 3.8c shows that the network reaches the bottom of the visualized loss surface. There are more then 10 distinct eigenvalues that have separated from the bulk sitting at zero, and the highest eigenvalue has a value of close to 300. One can also note that the bulk has shifted from a symmetric distribution to an asymmetric one, skewed toward the positive side. This means that the network has slowly escaped some of the very flat regions that represented maxima along the corresponding eigenvector.

3.5.3. Interpolation between Minima

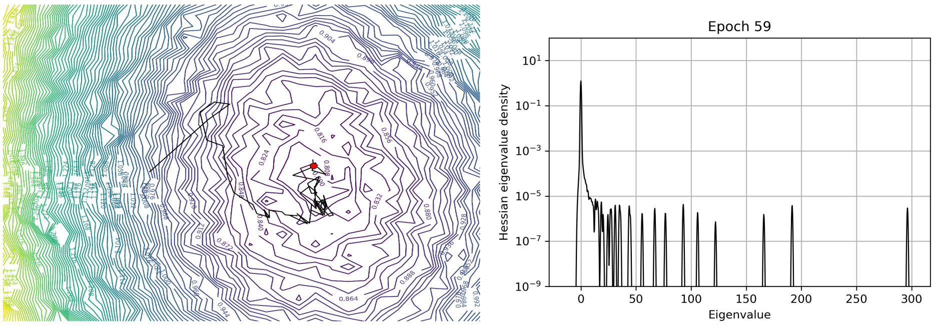
In this section a LeNet-5 architecture is trained two times with different initializations. The network is trained on CIFAR-10 using SGD with momentum with a learning rate of $\varepsilon = 0.01$ and a momentum step size of $\rho = 0.9$. The batch size is set to $|\mathbb{M}(k)| = 256$ and the network is trained for 10 epochs. In order to capture both minima, the one directional vector is chosen to be the direction from one converged point to the other. The other direction is the eigenvector corresponding to the highest eigenvalue of one of the minima. The visualization is computed using a square grid of size $N = 50$ and 20% of all samples. In order to investigate the area between the two minima in more detail, both are connected by a straight line. At 20 equidistant points along this line the spectral densities are computed using the stochastic Lanczos quadrature algorithm with $k = 10$ iterations and $m = 80$ iterations of the Lanczos



(a) *Loss landscape and eigenvalue density plot of ResNet-32 trained on CIFAR-10 at initialization.*



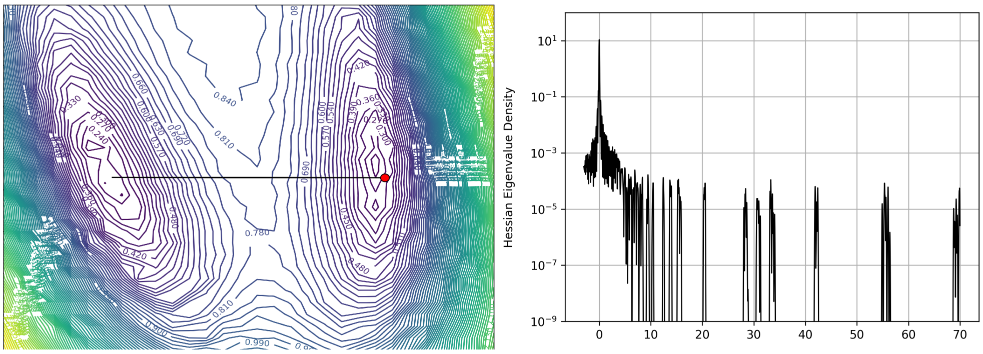
(b) *Loss landscape and eigenvalue density plot of ResNet-32 trained on CIFAR-10 at epoch 6.*



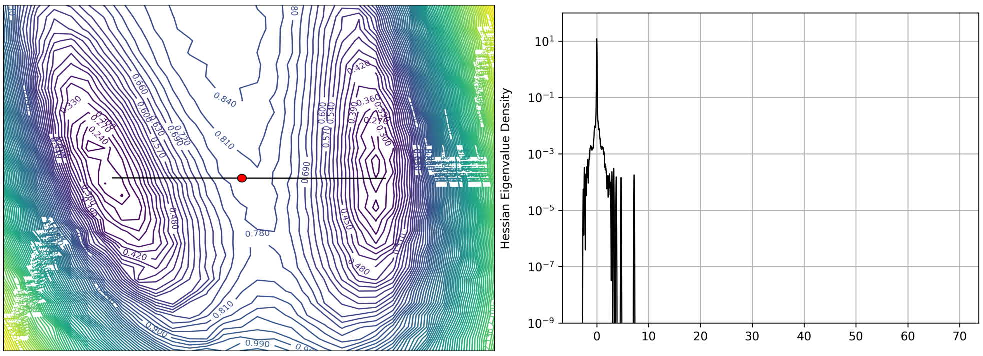
(c) *Loss landscape and eigenvalue density plot of ResNet-32 trained on CIFAR-10 at epoch 59.*

Figure 3.8.: *Visualization of the loss landscape and the Hessian eigenvalue density of a ResNet-32 network trained on CIFAR-10 at different Epochs. The ResNet-32 architecture has been trained with the Adam optimizer with a learning rate of* 6×10^{-4} *and* $\beta = (0.9, 0.999)$. *For each plot 15 % of the CIFAR-10 training samples were used for the loss landscape visualization plots and 5 % of the samples were used for the density spectrum computations. The loss landscape plot was crated with a grid size of 50 on each side.*

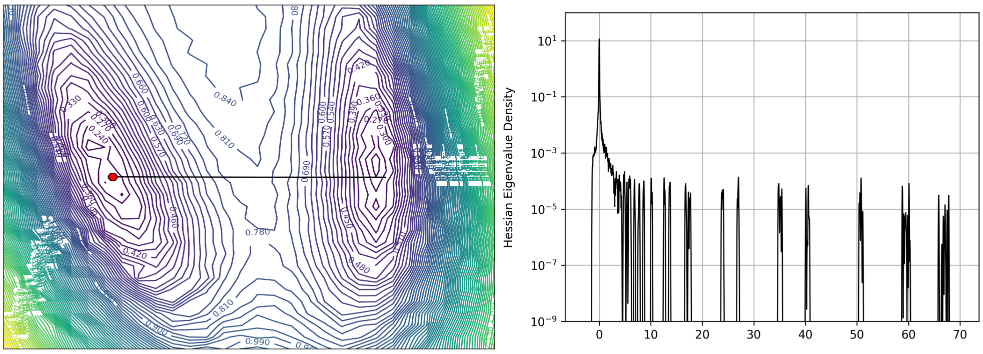
algorithm. The results for three different points along the line are summarized in Figure 3.9[2].



(a) *Interpolation between 2 minima with eigenvalue density spectrum in minimum 1*



(b) *Interpolation between 2 minima with eigenvalue density spectrum in between those*



(c) *Interpolation between 2 minima with eigenvalue density spectrum in minimum 2*

Figure 3.9.: *Visualization of the loss landscape and the Hessian eigenvalue density at different points on the line connecting both minima. A LeNet architecture has been trained with two different initializations to end up in two different minima. For both plots 20 % of the CIFAR10 training samples were used. The loss landscape plot was crated with a grid size of 50 on each side.*

As can be seen from the spectral densities inside both minima, the network has converged to a saddle point, with negative eigenvalues present on the left of the bulk at zero. There are several distinct positive eigenvalues in both cases as well, indicating that the network has converged to a minimum in those directions. Going toward

---

[2] A video showing all points connecting the minima can be viewed on https://youtu.be/8UIwPV6yU6I

the middle of the interpolation, in between both minima, reveals that the network is relatively flat at this point. All the distinct eigenvalues that are positive vanish except one. The loss landscape plots show that the two minima seem to be located inside a much bigger minimum. Both minima also have a similar shape and are separated by a relatively flat region as is also indicated by the spectral density depicted in Figure 3.9b.

This chapter introduced the concept of visualizing the high-dimensional loss landscape by projecting it down onto lower-dimensional subspaces. The drawbacks of commonly used methods were presented, like PCA for example, and an alternative method of finding meaningful directions in parameter space was introduced. By computing the eigenvectors of different eigenvalues of interest, one can observe the trajectory in several different ways, which can for example reveal situations where the optimizer struggles to converge. Next, a novel parallelization method was presented that allows for fast loss landscape computation by partitioning the underlying grid into several subgrids which are computed by the different workers. This parallelization was used together with the previously introduced parallel stochastic Lanczos quadrature algorithm for fast per-iteration computations of the eigenvalue density spectra and loss landscapes for large deep neural networks. Experiments reveal several interesting behaviors of different optimizers, for example how the optimizer bounces around inside a minimum when using a constant learning rate. Also, the loss landscape and the corresponding eigenvalue density spectra between two different minima was shown for the first time.

This work can be extended in several different ways. Firstly, one can try and investigate the loss landscape and eigenvalue density spectra of different parts of the network, for example by looking at different filters in convolutions and how those converge during training. Secondly, one can try and investigate the behavior of different network structures in order to understand why certain networks are easier to train than others for example. The same can be done for various optimizers in order to understand why some work better than others and where these struggle to converge. Thirdly, one can try and investigate interesting phenomena, like the large batch-size problem [94, 199] for example.

The next chapter will use the tools from this chapter together with the eigenvalue spectra computations in order to investigate generative adversarial networks and introduce a new optimizer that is able to regularize these networks for the purpose of preventing them from mode collapse.

Chapter 4

Stabilizing GANs through Eigenvalue Regularization

Unlike previous examples of neural networks that were used for image classification, *Generative Adversarial Networks* (GANs) provide state-of-the-art results in image generation. However, despite their impressive results, these types of networks still remain very challenging to train. This is in particular caused by their non-convex optimization space that can lead to a number of instabilities. Among them, mode collapse stands out as one of the most daunting ones. This undesirable event occurs when the model can only fit a few modes of the data distribution, while ignoring the majority of them.

This section serves as a practical example of how loss surface visualization and second-order information during training can help stabilize networks. Using second-order gradient information, the GAN can be successfully trained without suffering from mode collapse. To do so, the loss surface is analyzed through its Hessian eigenvalues, which reveals that mode collapse is related to the convergence of the network towards sharp minima. In particular, the eigenvalues of the generator and the discriminator are directly correlated with the occurrence of mode collapse. Finally, motivated by these findings, a new optimization algorithm called *nudged-Adam* (NuGAN) is designed, that uses spectral information to overcome mode collapse, leading to empirically more stable convergence properties.

4.1. General Adversarial Networks

GANs belong to the family of unsupervised generative models [1] and consist of a generator network (G) and a discriminator network (D). These networks are trained within an adversarial game, in which the generator produces new samples from a noise distribution and the discriminator tries to distinguish between real and generated samples [72]. Within this adversarial game, the generator learns to produce new samples that are distributed according to the desired data distribution of the real samples. Figure 4.1 presents an overview of a typical GAN network.

Training can be formulated in terms of minimax optimization of a value function

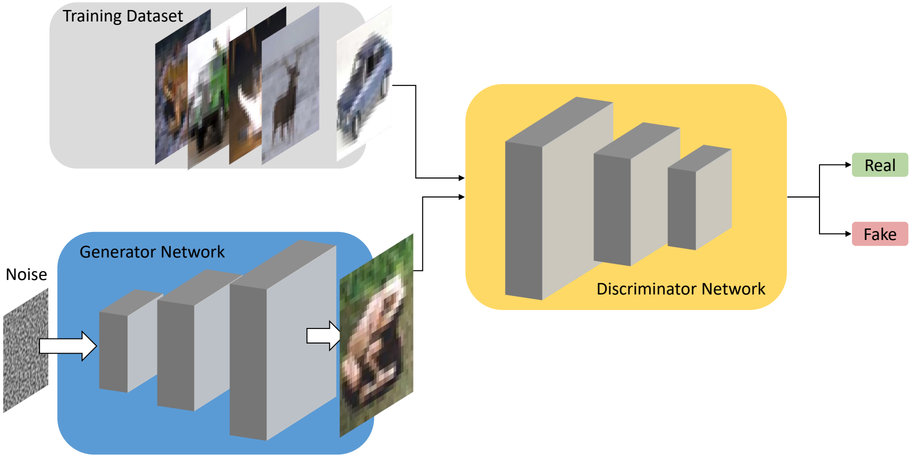


Figure 4.1.: *Illustration of a generative adversarial network. On the lower left side the generator network gets random noise as an input and outputs a generated image. The discriminator network on the right side has to decide whether the image presented to it is from the training dataset (images are taken from the CIFAR-10 dataset [114]) or from the generator network.*

$V(G,D)$ [76]:

$$G^* = \arg\min_G \max_D V(G,D). \tag{4.1}$$

The goal of this adversarial game is to train the generator network to generate high quality samples that the discriminator struggles to distinguish from the real ones, thus returning a probability of 0.5 that a given sample is fake. Since the primary interest lies in the generator, the discriminator can be dropped after training.

Even though GANs can be very powerful in modeling the probability distribution underlying the set of the training data, they are hard to train. Due to the adversarial zero-sum game played by the generator and the discriminator, their optimal solution is a Nash equilibrium [171] of this game. The generator and the discriminator are neural network models, therefore their training is equivalent to the search of Nash equilibria in a high-dimensional, highly non-convex optimization space. The most common algorithm used for solving this optimization problem is *gradient descent-ascent* (GDA) [163], where generator and discriminator models perform alternating update steps using first order gradient information w.r.t. the loss function. In practice, GDA is often combined with regularization in order to yield many state-of-the-art results on various benchmark datasets. Despite its popularity, GDA is known to suffer from undesirable convergence properties that may lead to instabilities, divergence, cyclic behavior, catastrophic forgetting and mode collapse [141]. This thesis will focus particularly on mode collapse, which refers to the scenario in which the generator only produces a limited variety of samples.

Recently, many different works have tried to tackle these issues. One of the first attempts was [181], that used convolutional neural networks in order to improve training stability as well as the visual quality of the samples that were generated. Other works focused on improving different aspects of GAN training, with some achieving improvements through the use of new objective functions [11, 193] and additional regularization terms [59, 79]. Other works have advanced the theoretical understanding of the training of GANs. The convergence properties of GAN training using first-order information were investigated by [154, 159]. They show that a local analysis of the eigenvalues of the Jacobian of the loss function can provide guarantees on local stability properties. References [22, 63] have pushed this theoretical analysis further, by looking at the k biggest eigenvalues of the Hessian of the loss in order to investigate the convergence and dynamics of GANs during training.

The goal of a generative model is to approximate a real data distribution p_r with a surrogate data distribution p_f. One way to achieve this is to minimize the "distance" between these two distributions. This process is shown in Figure 4.2. In order to minimize the distance between two distributions, the original GAN formulation used the Jensen-Shannon divergence [72, 138] between p_r and p_f. This is achieved using the feedback of the discriminator. More recently, the Wasserstein distance [225] has also been introduced and has shown to improve GAN performance compared to the Jensen-Shannon divergence.

During the training of a GAN, the discriminator D tries to maximize the probability of correctly classifying a given input as real or fake by updating its loss function

$$\mathcal{L}_D = \mathbb{E}_{\boldsymbol{x}\sim p_\mathrm{r}}[\log(D(\boldsymbol{x}))] + \mathbb{E}_{\boldsymbol{z}\sim p_z}[\log(1-D(G(\boldsymbol{z}))], \tag{4.2}$$

through stochastic gradient ascent. Here, $\boldsymbol{x}$ is a sample from the dataset and $\boldsymbol{z}$ is drawn randomly from a specified distribution (most common examples are the normal distribution $\boldsymbol{z} \sim \mathcal{N}(0,1)$ or the uniform distribution $\boldsymbol{z} \sim U(-1,1)$). The generator G, on the other hand, tries to minimize the probability that D classifies its generated data correctly. This is done by updating its loss function

$$\mathcal{L}_G = \mathbb{E}_{\boldsymbol{z}\sim p_z}[\log(1-D(G(\boldsymbol{z}))] \tag{4.3}$$

via stochastic gradient descent.

Thus, this joint optimization represents a minimax game between G and D, where G learns to generate new samples that have the same distribution as p_r, and D learns to distinguish between real and generated samples. The Nash-equilibrium of this game is reached when the generator is able to generate samples that look as if they are drawn from the training data distribution, while the discriminator is indecisive whether the input is generated or real, meaning that it outputs a probability of 0.5 for each sample that is presented.

4.1.1. Training of GANs

Because the training of GANs requires joint optimization of several objectives, this makes their convergence intrinsically different from the case of a single objective

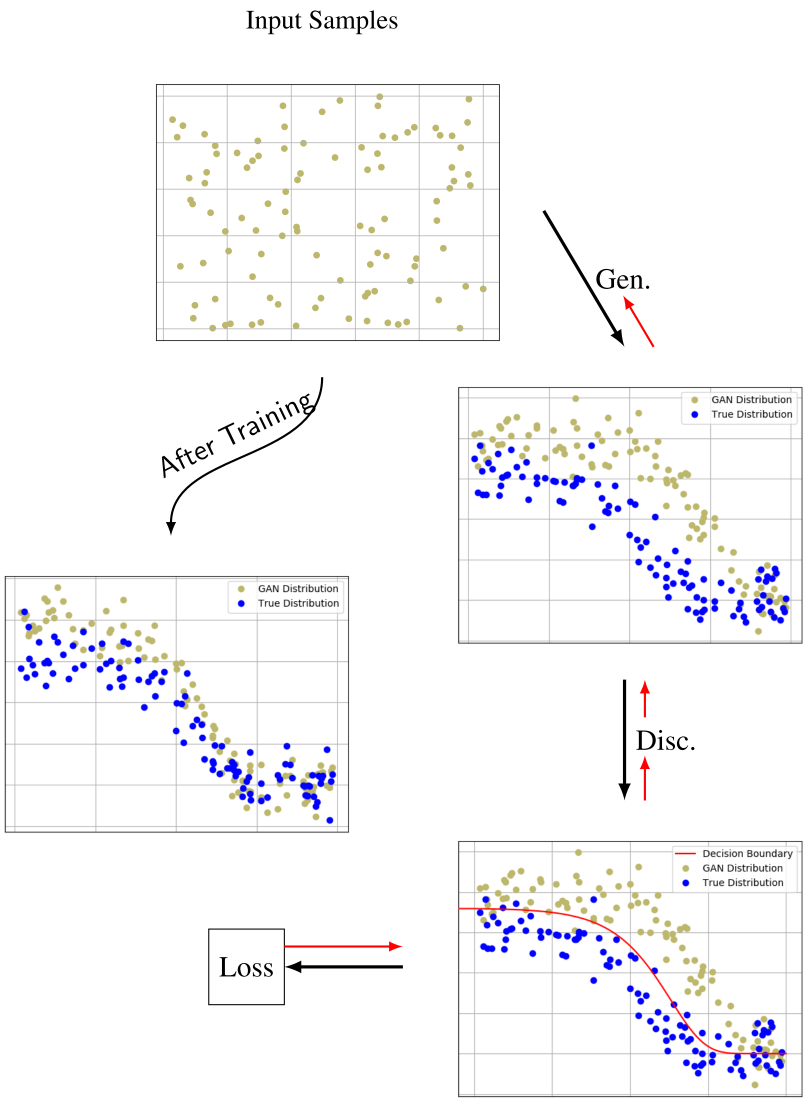


Figure 4.2.: *Visualization of the learning process of GANs. The top plot shows samples that are drawn from a uniform distribution. These are fed into the generator, which outputs its generated distribution. Ideally, this generated distribution is indistinguishable from the true distribution of the data samples. The middle right plot depicts the generated distribution of the GAN together with the true distribution of the samples. In the next step the discriminator has to decide which samples are to the true and which from the generated distribution. This is depicted in the bottom plot. Afterwards, using backpropagation (red arrows), the parameters of the discriminator and the generator are updated. This process is repeated for a number of steps during training. Ideally, after the training, the generator is able to generate samples that are almost indistinguishable from the true distribution, as is shown in the left plot.*

function that is frequently encountered in deep learning (e.g. classification). In general, the optimal joint solution to a minimax game is called Nash-equilibrium. Because the objectives are non-convex, one can only expect to find local optima in the case where local gradient information is used. These local optima are called *local Nash-equilibria* (LNE) [2].

If the GAN converges into a LNE, it has reached a point for which there exists a local neighborhood in parameter space where neither the generator nor the discriminator can unilaterally decrease/increase their respective losses. This means the network has reached a saddle point, where the gradients of the generator and discriminator vanish, while their respective second derivative matrix is positive and negative semi-definite:

$$\begin{aligned} ||\nabla_{\boldsymbol{\theta}}\mathcal{L}_G|| = ||\nabla_{\boldsymbol{\zeta}}\mathcal{L}_D|| = 0, \\ \nabla^2_{\boldsymbol{\theta}}\mathcal{L}_G \succeq 0 \text{ and } \nabla^2_{\boldsymbol{\zeta}}\mathcal{L}_D \preceq 0 \end{aligned} \tag{4.4}$$

Here, $\boldsymbol{\theta}$ and $\boldsymbol{\zeta}$ are the weights of the generator and the discriminator respectively.

4.1.2. Evaluation of GANs

In recent years, GANs have seen a dramatic improvement in terms of image quality. Their improvements have reached a point where it is possible to generate artificial high-resolution faces indistinguishable from real images of humans [107]. Despite their successes, it is not clear how to quantitatively evaluate and compare GANs, and so far there is no consensus as to which metric can best capture strengths and limitations of different models. One attempt at providing such a metric is the *Inception Score* (IS), proposed by [193]. This score is the most widely adopted metric as it provides a numerical value that reasonably correlates with the quality and diversity of output images. The IS is based on the assumption that the generator should output images that can be classified and are uniformly distributed between the different classes. The IS ranges from zero to infinity, and higher inception scores correlate with higher quality and diversity of output images.

In order to compute the IS, the KL-Divergence is used to compute the distance between the label distribution of different generated samples and the marginal distribution of multiple samples. Since in the ideal case there would be a uniform marginal distribution between the classes and a more focused label distribution for individual samples, the goal is to maximize the KL-Divergence. In the case of mode collapse, the marginal distribution is not uniformly distributed but is shifted toward certain classes. This will reduce the IS and thus allows evaluation of the amount of mode collapse happening in the generator.

4.1.3. Non-Saturating GAN

In practice, GANs have been found to train better when using an alternative cost function that ensures that the generated samples have a high probability of being classified as real by the discriminator.

A *non-saturating GAN* (NSGAN) is a modified version of a GAN that trains the generator by maximizing the alternative objective

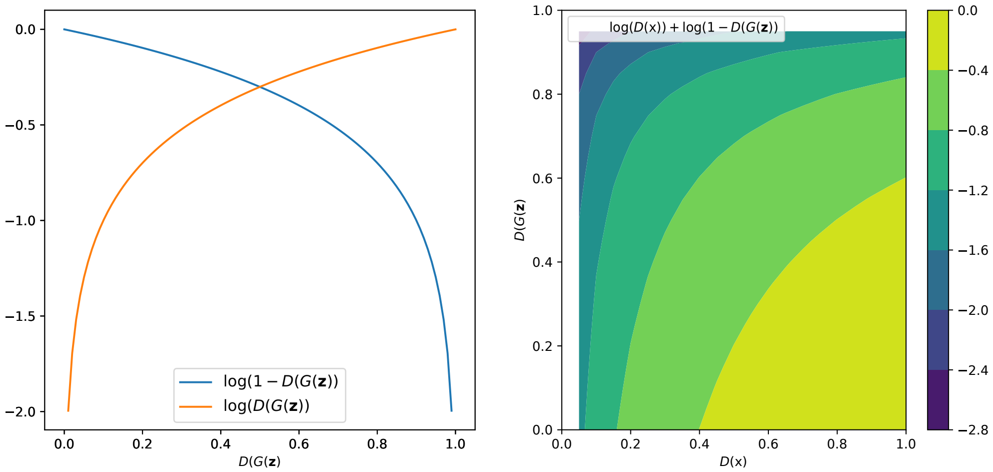


Figure 4.3.: *Plots of the different losses found in GANs. (Left) Generator losses, either minimization of* $\log(1 - D(G(\mathbf{z}))$ *or maximization of* $\log(D(G(\mathbf{z}))$. *(Right) Discriminator loss, maximization of* $\log(D(\mathrm{x})) + \log(1 - D(G(\mathbf{z}))$.

$$\mathbb{E}_{z \sim p_z}[\log(D(G(z))]. \tag{4.5}$$

The intuition why NSGANs perform better than GANs is as follows. If the model distribution highly differs from the data distribution, as is the case during early iterations, the NSGAN is able to bring the model distribution closer to the data distribution because the loss function generates a strong gradient. This gradient will only start to vanish once the discriminator starts being indecisive whether the input is from the data distribution or from the model distribution. This makes sure that the generated samples will have already reached a distribution that is close to the data distribution by that time. Figure 4.3 shows the loss function of the original and non-saturating generator and discriminator, respectively.

4.2. Mode Collapse

As was explained above, mode collapse (also called mode dropping) refers to the issue where the generator G only generates a limited amount of variety in the samples. The problem of mode collapse can arise when the discriminator is not properly trained and thus gets easily fooled by specific images from the generator. Because of the minimax nature of the adversarial training, the generator will exploit this weakness of the discriminator by generating more samples from these classes that the discriminator struggles to distinguish. On the other hand, this forces the discriminator to only learn to detect these types of samples. In order to avoid this, the generator can start to generate samples from a different mode and the GAN starts to cycle between these modes [75]. This leads to the situation where both networks have overfitted and the GAN has collapsed into generating and predicting only a few modes of the multimodal dataset [75].

There are theoretical explanations on the appearance of mode collapse ([12, 13]).

Firstly, [13] finds that a discriminator network is not able to detect mode dropping if the network has a polynomial number of parameters with respect to the dimensionality of the data. Also, [12] show that the commonly used training objective in adversarial training, which does not suffer as much from vanishing gradients as the original maximum likelihood objective function, is to blame for the missing modes. As explained in [132], the traditional formulation for generative models used maximum likelihood during training. Since maximum likelihood is the product of the individual densities of all the training examples, assigning low densities to certain examples would reduce the maximum likelihood objective function. Due to this property, mode dropping is discouraged and the model does not suffer from mode collapse. With this in mind, model performance can easily be assessed through visual inspection.

The following section serves as an intuitive explanation on mode collapse in generative models and partly follows explanations from [75, 97, 221]. Using maximum likelihood for training generative models stems from using the KL-divergence [97]. The KL-divergence measures the difference between two distributions. It is not a distance measure per se, since it is not symmetric in its inputs. In order to train a model, such that it accurately models the underlying distribution $p(x)$, the KL-divergence between $p(x)$ and the model $q_{\boldsymbol{\theta}}(x)$ is minimized

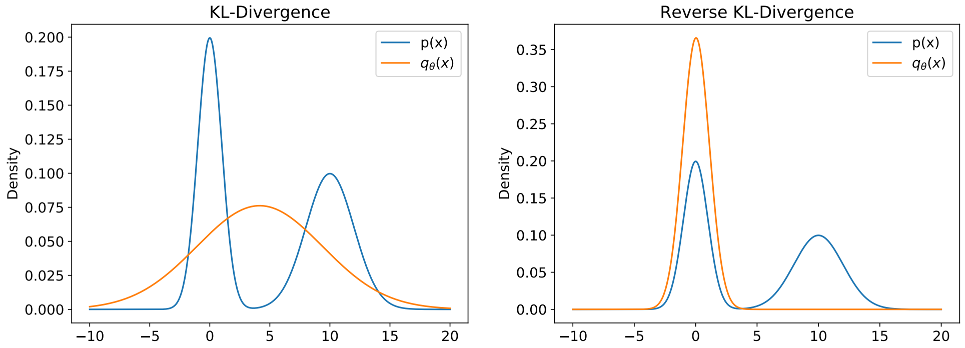
$$\begin{aligned}
\boldsymbol{\theta}^{\star} &= \arg\min_{\boldsymbol{\theta}} KL(p||q_{\boldsymbol{\theta}}) && (4.6)\\
&= \arg\min_{\boldsymbol{\theta}} \mathop{\mathbb{E}}_{x\sim p}[\log p(x)] - \mathop{\mathbb{E}}_{x\sim p}[\log q_{\boldsymbol{\theta}}(x)] && (4.7)\\
&= \arg\max_{\boldsymbol{\theta}} \mathop{\mathbb{E}}_{x\sim p}[\log q_{\boldsymbol{\theta}}(x)] && (4.8)
\end{aligned}$$

Because the first term in Equation (4.7) does not depend on the variable θ, its gradient will be zero, so it can be removed from this equation. The resulting expression in Equation (4.8) is the maximum likelihood equation. Figure 4.4a depicts a bimodal target distribution $p(x)$ and the model $q_{\boldsymbol{\theta}}(x)$ that has been trained using Equation (4.7). One drawback of using maximum likelihood is that it tries to cover the entire support of the underlying target distribution. As a consequence, the model has to sacrifice its ability to accurately model $p(x)$, which can result in low quality samples.

In order to deal with these issues, the reverse KL-divergence can be used [12]. It swaps the order of the distributions in the KL-divergence such that

$$\begin{aligned}
\boldsymbol{\theta}^{\star} &= \arg\min_{\boldsymbol{\theta}} KL(q_{\boldsymbol{\theta}}||p) && (4.9)\\
&= \arg\min_{\boldsymbol{\theta}} \mathop{\mathbb{E}}_{x\sim q_{\boldsymbol{\theta}}}[\log q_{\boldsymbol{\theta}}(x)] - \mathop{\mathbb{E}}_{x\sim q_{\boldsymbol{\theta}}}[\log p(x)]. && (4.10)
\end{aligned}$$

Note that in this case the samples are drawn from the model instead of the target distribution. Therefore the second term in Equation (4.10) can not be omitted. This term tries to maximize the expected value of the target distribution given samples from the model distribution, meaning that any sample drawn from $q_{\boldsymbol{\theta}}$ should be close to $p(x)$. In theory, the model could collapse onto a single point which is extremely close to the true distribution. The first term in Equation (4.10) prevents this from happening.



(a) *Density of the underlying data distribution $p(x)$ and the model distribution $q_\theta(x)$ after minimizing the KL-divergence (equal to maximum likelihood estimation). The final parameters of the model distribution after training for 100 iterations are $\theta = (5.01, 5.17)$.*

(b) *Density of the underlying data distribution $p(x)$ and the model distribution $q_\theta(x)$ after minimizing the reverse KL-divergence. The final parameters of the model distribution after training for 100 iterations are $\theta = (0.01, 1.06)$.*

Figure 4.4.: *Visualization of the difference between minimizing the Kl-divergence and the reverse KL-divergence. In both plots the target distribution is represented by $p(x)$, which is a mixture of Gaussians. The first Gaussian has mean $\mu_1 = 0$ and standard deviation $\sigma_1 = 1$, while the second Gaussian has mean $\mu_2 = 10$ and standard deviation $\sigma_2 = 2$. The model distribution $q_\theta(x)$ is a Gaussian distribution which contains two parameters $\theta = (\mu, \sigma)$, the mean and standard deviation of the distribution. In both cases the optimization was performed using an Evolutionary Strategy (ES) [184] with Monte-Carlo sampling. In this example, the distribution $p(x)$ represents a bimodal distribution.*

This term describes the negative entropy of the model distribution, therefore Equation (4.10) tries to maximize the entropy of $q_{\boldsymbol{\theta}}$, which spreads its probability mass, thus countering the effect of the second term.

Figure 4.4b depicts a bimodal target distribution $p(x)$ and the model $q_{\boldsymbol{\theta}}(x)$ that has been trained using Equation (4.10). The parameterized distribution $q_{\boldsymbol{\theta}}(x)$ learns to accurately model one mode of the target distribution, while ignoring its second mode. Thus, training using reverse KL-divergence can lead to more accurate samples, but it is prone to mode collapse. For real world datasets, the true distribution $p(x)$ is rarely known, so evaluating Equation (4.10) is impossible.

Nowadays, generative models like GANs have shifted away from using maximum likelihood during training and thus there is no guarantee whether the network sacrifices its ability to accurately model the diversity of the underlying data distribution in order to generate samples with higher quality. GANs minimize the following general objective function [75]:

$$\boldsymbol{\theta}^\star = \arg\min_{\boldsymbol{\theta}}\max_{\boldsymbol{\zeta}} \mathop{\mathbb{E}}_{\mathrm{x}\sim \mathrm{p},\hat{\mathrm{x}}\sim \mathrm{q}_{\boldsymbol{\theta}}} \mathcal{L}(f_{\boldsymbol{\zeta}}(x), f_{\boldsymbol{\zeta}}(\hat{x})). \tag{4.11}$$

In this example the generator model is given by $q_{\boldsymbol{\theta}}$ and the discriminator model by $f_{\boldsymbol{\zeta}}$. Similar to the above examples of the forward and reverse KL-divergence in Equations

(4.6)-(4.10), Equation (4.11) also minimizes a divergence, given by

$$\max_{\zeta} \mathop{\mathbb{E}}_{x \sim p, \hat{x} \sim q_\theta} \mathcal{L}(f_\zeta(x), f_\zeta(\hat{x})).$$

This divergence is learned by the discriminator network and the structure of the loss function $\mathcal{L}$ determines the type of divergence. For example, [167] shows that

$$\mathcal{L}(f_\zeta(x), f_\zeta(\hat{x})) = -\exp(f_\zeta(x)) + 1 + f_\zeta(\hat{x})$$

is consistent with the reverse KL-divergence formulation.

Many early GANs used the following loss function

$$\mathcal{L}(f_\zeta(x), f_\zeta(\hat{x})) = \log f_\zeta(x) + \log[1 - f_\zeta(\hat{x})]$$

which was also introduced in Equation (4.2). This expression is equivalent to minimizing the Jensen-Shannon divergence [167], which consists of the forward and the reverse KL-divergence and has been found to suffer from mode collapse as well.

In summary, using the maximum likelihood method results in more stable training, prevents mode dropping and allows for an easy assessment of model performance. The downside of using maximum likelihood can be the poor quality of the samples generated after training, since most of the models capacity is used for covering the entire support of the true data distribution. Moving to a method that does not require the model to cover the entire support of the true data distribution, as seen in the example of the reverse KL-divergence or GAN objective function, has the benefit that the model is able to produce samples of higher quality, though it can come at the expense of mode dropping. Also, there is no straightforward way of assessing model performance anymore.

4.3. Loss Surface of GANs

This section introduces the loss landscape visualization of GANs. This will be helpful for better understanding the non-convergent nature of GANs and the general problem of LNEs. The loss landscape visualization features 2D visualization as well as plots of the biggest eigenvalue of the generator and the discriminator during training.

The loss landscape of GANs has been studied before [22, 154] . In [22], the authors study the training dynamics of GANs by observing their game vector-field, which is the gradient of the generator loss and the discriminator loss taken with respect to their respective parameters. They are able to show for real datasets that GAN training involves rotations around local stable stationary points of the game vector-field. Furthermore, the authors provide evidence that the generator does not converge into a local minimum, but into a saddle point instead.

This thesis computes the loss landscape visualization of an NSGAN that is trained for 180 epochs. Additionally, the two biggest eigenvalues of the generator and the discriminator at the converged point are computed. These serve as the directional vectors for the loss landscape visualization. Figure 4.5 shows the loss landscape and the training trajectory of the NSGAN after 180 epochs. The left plot shows the generator while the right plot displays the trajectory and loss landscape of the

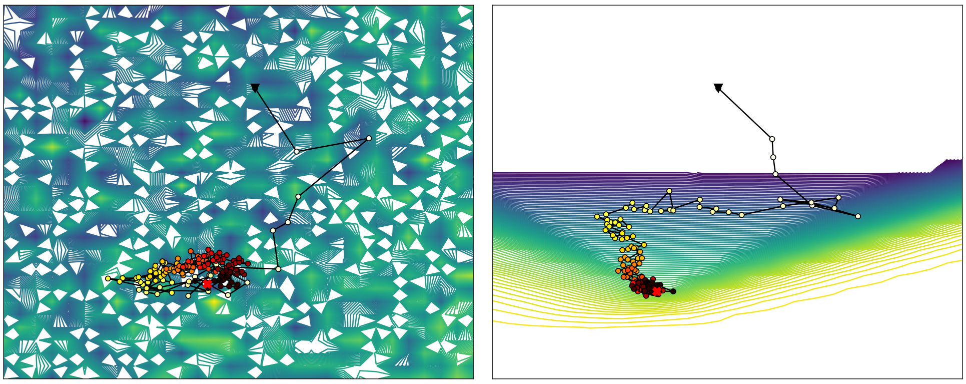


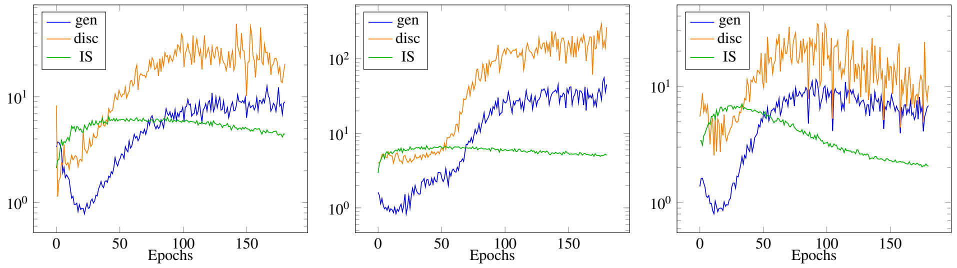
Figure 4.5.: *Logarithmic loss landscapes with trajectory of the same training run, visualized along eigenvectors corresponding to the two highest eigenvalues of NSGAN on MNIST.(Left) Generator loss landscape. (Right) Discriminator loss landscape.*

discriminator. These plots show that the discriminator on the right hand side starts descending toward a minimum, while the generator on the left hand side finds itself in a very chaotic loss landscape where it bounces around inside a certain area. These findings are in line with the second-order literature on GANs [22].

After gaining some intuition on the training dynamics of GANs and their problem to find and stay at an LNE, the goal is to gain insight into the issue of mode collapse. Empirical evidence is provided of a plausible relationship between mode collapse and the behavior of the eigenvalues of the generator and discriminator networks. This is achieved by evaluating the spectral density of the model throughout the optimization process. In the first experiment, the largest eigenvalues from the generator as well as discriminator network are computed for each epoch, together with the inception score. Several experiments are conducted, in which the original non-saturating GAN architecture is trained on the MNIST, Kuzushiji, Fashion and EMNIST datasets. Figure 4.6 depicts the biggest eigenvalue of the generator and discriminator together with the Inception Score for each of those datasets. There are several patterns of interest present in each experiment:

First, the evolution of the eigenvalues of the generator and discriminator behave visually very similar. In particular, when the discriminator exhibits an increasing tendency in its eigenvalues, the generator does so as well. Second, it is observed that the correlation between the top-k eigenvalues of the generator and discriminator is very high. The observed correlation in all the setups ranges from 0.72 up to 0.90. Third, there seems to exist a connection between the inception score and the behaviour of the eigenvalues. The eigenvalues and the inception score seem to be inversely correlated, as there are sections where the eigenvalues have a decreasing tendency while the IS tends to increase and the exist other sections where it is the other way around. Moreover, note that all the models start to suffer from mode collapse after 25 epochs (approximately when the eigenvalues tendency changes and starts to increase).

The empirical observations found in this analysis lead to the conclusion that eigenvalues can give an indication on the state of convergence of a GAN, as pointed out in [22]. Furthermore, experiments indicate that the eigenvalue evolution is correlated



(a) *Kuzushiji dataset. Correlation 0.80.* **(b)** *Fashion dataset. Correlation 0.90.* **(c)** *EMNIST dataset. Correlation 0.72.*

Figure 4.6.: *Evolution of the top k-eigenvalues of the Hessian from generator (gen) and discriminator (disc), and the correspondence IS over the whole training phase. The correlation score is measured between the generator and the discriminator.*

between the generator and discriminator network and these are correlated with the likely occurrence of a mode collapse event.

4.4. NudgedAdam: Preventing GANs from Mode Collapse

Using the insights gained from the experiments in the previous section, this thesis presents a new optimizer that aims at reducing the probability of a mode collapse event. In order to prevent the neural network from converging into sharp minima during the optimization, the gradient information in the direction of high eigenvalues is removed. This keeps the network from converging into sharp minima by ignoring these directions in the loss landscape and thus converging into wider minima. NudgedAdam is an optimizer inspired by [102], that ignores gradient information in the direction of the eigenvectors of the k biggest eigenvalues. It is based on the Adam optimizer but can theoretically be extended to any optimizer that uses gradient information.

Given the biggest k eigenvectors u_i and the gradient $\boldsymbol{g}$ at a specific point, the eigenvector directions are removed by

$$\boldsymbol{g}^* = \boldsymbol{g} - \sum_{i=1}^{k} < \boldsymbol{g}, \mathrm{u}_i > \mathrm{u}_i \tag{4.12}$$

with $\boldsymbol{g}^*$ the resulting gradient that is then used by the regular Adam optimizer. Using this technique, it is possible to create a multitude of nudged versions of popular optimizers, like SGD for example, by using $\boldsymbol{g}^*$ instead of the true gradient $\boldsymbol{g}$. A sketch of how the nudged optimizer works is shown in Figure 4.7. The eigenvectors are computed by using the Lanczos method together with the R-operator, which allows fast computation of eigenvalues and eigenvectors without having to store the full Hessian.

The NSGAN is trained using the NudgedAdam optimizer for 180 epochs. The goal is to prevent the network from suffering from mode collapse. Figure 4.8 shows the results from this training where only the top-2 eigenvectors are subtracted from the gradient information. The top row shows samples of the NSGAN generator trained with the regular Adam optimizer at epoch 2, epoch 25 and epoch 160. The bottom

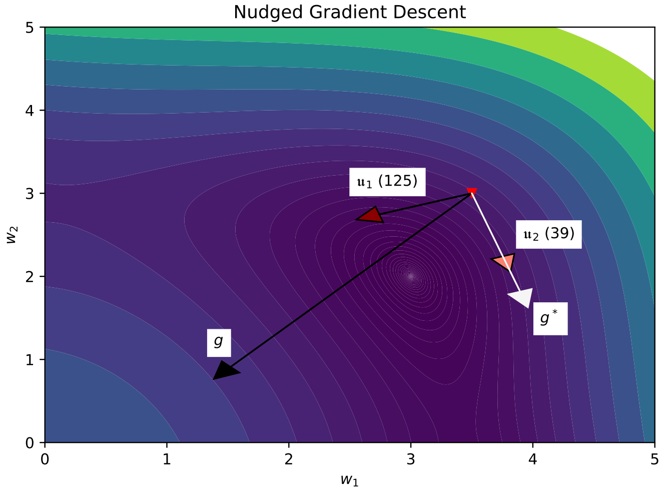


Figure 4.7.: *Depiction of the idea behind NudgedAdam. This figure depicts Himmelblau's function on the domain* $\boldsymbol{w} \in [0,5] \times [0,5]$. *The starting point is located at* $\boldsymbol{w}^{(0)} = (3.5, 3)$. *The red and orange arrows depict the two eigenvectors* $\mathfrak{u}_1$ *and* $\mathfrak{u}_2$ *respectively (Their respective eigenvalues are shown in parentheses next to the vectors). The initial gradient* $\boldsymbol{g}$ *is shown in black, while the final, nudged gradient* $\boldsymbol{g}^\star$ *is shown in white. One can see that while the original gradient points towards the center of the minimum, the nudged gradient points perpendicular to it, thus avoiding the convergence into the potentially sharp minimum.*

row depicts samples of the NSGAN generator trained with NudgedAdam at the same epochs. Looking at the inception score of the NSGAN trained with each optimizer, it becomes evident that the network trained with Adam suffers from mode collapse while the network trained with NudgedAdam does not. This can also be seen in the generator samples at epoch 160. The top row displays much more 1-images compared to all the other numbers. This mode collapse is not present in the bottom row, where the network was trained with NudgedAdam.

The full spectrum of the Hessian of the generator as well as the discriminator at different stages of training is depicted in Figure 4.9. The first column shows the eigenvalue spectra of the NSGAN trained with Adam. Looking at the generator spectrum, one interesting observation to make is that the spectrum stays symmetric during all stages of the training. At epoch 160 there are seemingly as many negative eigenvalues present as there are positive ones. The discriminator on the other hand falls into a minimum, with the largest eigenvalues reaching values of around 350. On the right column, the NSGAN is trained with the NudgedAdam optimizer. There are still negative eigenvalues present in the spectrum of the generator, which indicates that the converged point that was reached during training is not an LNE (c.f. Formula 4.4). In both cases, the generator only reaches a saddle point. The discriminator of the GAN trained with the NudgedAdam optimizer converges into a minimum, though this minimum is much flatter than the case where the network was trained with Adam. An interesting observation is the connection between the eigenvalue spectra and mode collapse. In the left column, where mode collapse happened, the generator spectrum is much more spread out around zero compared to the right column. Also, the dis-

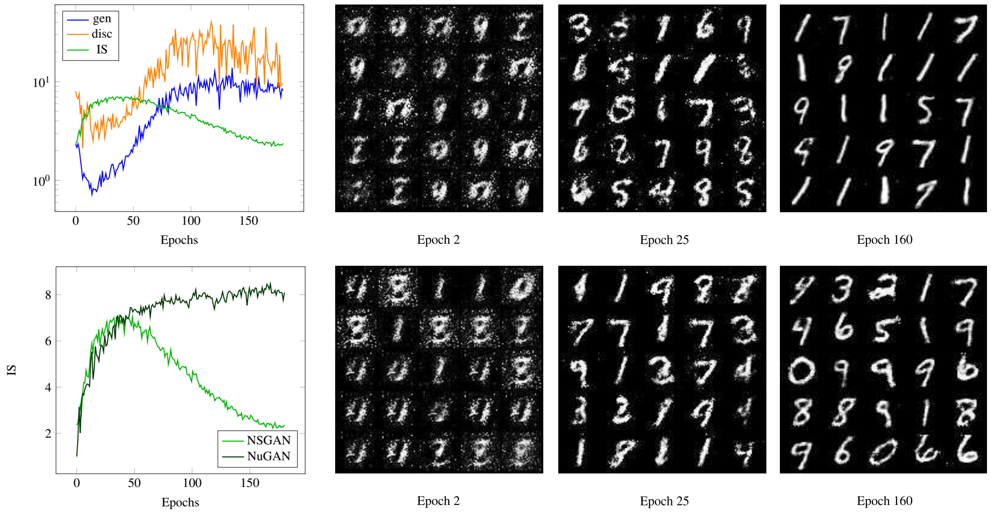


Figure 4.8.: *(First row) Evolution of the top-k eigenvalues of the Hessian, the IS and random generated samples at different epochs of NSGAN on MNIST. (Second row) Comparison of IS evolution of NSGAN and NuGAN, and random generated samples at different epochs of NuGAN.*

criminator spectrum has eigenvalues that are much higher compared to the network trained with nudgedAdam on the right column.

Table 4.1 shows more quantitative results that show the benefit of the NudgedAdam optimizer approach. The Table compares the inception score for both optimizers evaluated on four different datasets. In all cases, the NudgedAdam approach achieves a higher mean and also maximum inception score than the Adam optimizer.

Table 4.1.: *Mean and max IS from the different datasets and methods (with and without mode collapse). Higher values are better.*

Methods	NSGAN		NuGAN	
	mean	max	mean	max
MNIST	4.30	7.03	7.14	8.46
Kuzushiji	5.24	6.50	6.12	7.20
Fashion	5.74	6.82	6.35	7.20
EMNIST	3.77	7.02	8.53	7.67

These quantitative results together with the visual inspection of the image quality suggest that the NuGAN algorithm has a direct influence on the behavior of the eigenvalues and the loss landscape of the adversarial model, resulting in the avoidance of mode collapse.

In summary, experiments show that the algorithm does not converge to an LNE, while still achieving good results with respect to the evaluation metric (the Inception Score). This raises the question whether convergence to an LNE is actually needed in order to achieve good generator performance of a GAN.

This chapter introduced the concept of generative adversarial networks and explained how these generative networks can suffer from mode collapse during training.

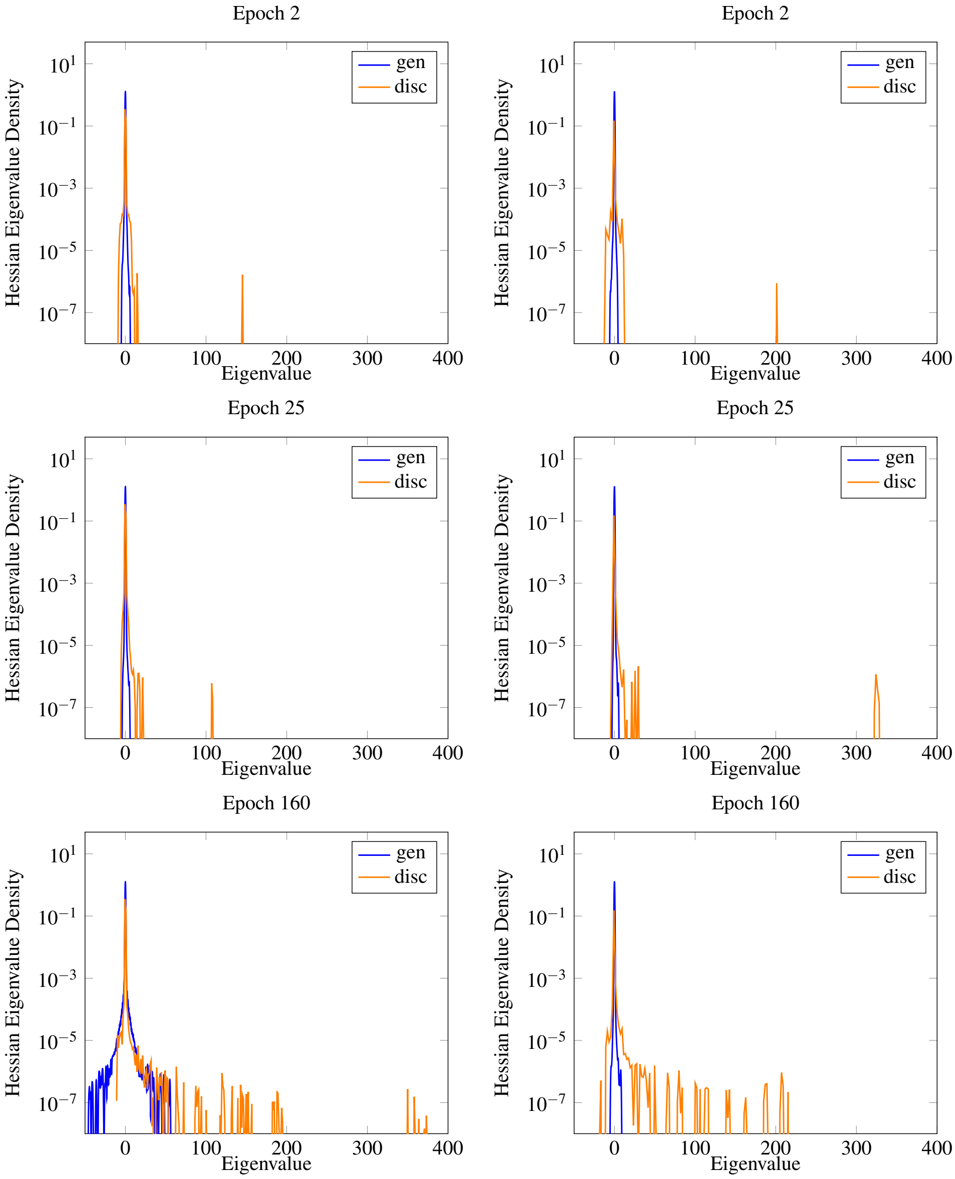


Figure 4.9.: *Plots of the whole spectrum of the Hessian at different stage of the training on MNIST. (First column) Results on NSGAN: we can identify an abnormal behaviour (mode collapse) in the generator at epoch 160. (Second column) Results on NuGAN: the spectrum remains stable during the whole training. We can observe how the discriminator for both cases finds local minima, while the generator remains all the time in a saddle point.*

Using the visualization methods introduced in the previous chapter, the eigenvalues of the generator as well as discriminator network were computed. These reveal how the top-k eigenvalues of both those networks are correlated with each other. Furthermore, the inception score, a measure of the amount of mode collapse in the network, is negatively correlated with the top-k eigenvalues of the network. Using this intuition, NudgedAdam was introduced, an optimizer that steers away from sharp minima by removing gradient information in the direction of the eigenvectors related to the biggest eigenvalues of the Hessian. This results in more robust GANs that do not suffer from mode collapse during training. Also, for the first time, the full spectral densities of the generator and discriminator eigenvalues are displayed for different stages of training. The spectrum of the discriminator trained with NudgedAdam shows that it has converged into a flatter minimum than the one that suffers from mode collapse. Also, the spectrum of the generator trained with NudgedAdam reaches a saddle point that is flatter than the generator of the GAN that has collapsed.

These results are promising and future research can extend this line of work. One could investigate the difference between the type of GANs featured in this work and GANs that use convolutions. These still suffered from mode collapse after training them with NudgedAdam, therefore it would be interesting to see their eigenvalue spectrum and how they differ from other GAN architectures. This could also result in another type of optimizer that is able to regularize GANs that use convolutions in order to prevent them from suffering from mode collapse.

Chapter 5

Proximal Gradient Methods

Proximal gradient methods belong to the class of *successive convex approximation* (SCA) methods [198], which approximate the objective function with a (strongly) convex approximation at every iteration. Contrary to second-order methods, which try to minimize the quadratic approximation of the objective function at every iteration, SCA methods are not limited to quadratic approximations and any specific structure of the objective function can be leveraged in order to design convex approximations. In theory, this algorithm only requires the first-order information in order to converge, though incorporating approximations that preserve the structure of the objective function can enhance its practical convergence [198]. Proximal gradient methods are one way to design a surrogate objective function that is strongly convex and can be easily computed. Thus, contrary to second-order methods, SCA algorithms can be both more efficient to compute strongly convex approximations of the original objective function while also being able to incorporate more information than just solving a quadratic approximation of the original problem.

This chapter considers the problem of training structured neural networks with nonsmooth regularization (e.g. ℓ_1-norm) and constraints (e.g. interval constraints). As shown previously, many methods aim at reducing the size of very large neural networks with minimal sacrifice to their performance. Often times, this is achieved through regularization terms that are non-differentiable at some points in the parameter domain (e.g. ℓ_1-norm). This chapter will start by first explaining the issue that arises from the commonly used subgradient method that is used for training neural networks with nonsmooth regularization. Next, the proximal operator as well as the proximal point algorithm and the proximal gradient method are introduced as a solution to the training of nonsmooth objective functions. All these explanations will be in the context of the deterministic setting. Afterwards, the next subsection focuses specifically on the training of neural networks and deals with the stochastic setting of training neural networks. To this end, the training of neural networks is formulated as a constrained nonsmooth nonconvex optimization problem.

Next, a convergent proximal-type stochastic gradient descent (ProxSGD) algorithm is proposed. Under properly selected learning rates, it is shown that with probability 1, every limit point of the sequence generated by the proposed ProxSGD algorithm is a stationary point. Finally, in order to support the theoretical analysis and demonstrate the flexibility of ProxSGD, the last section will focus on the training of neural networks with ℓ_1-regularization and provide extensive numerical tests that show how ProxSGD can be used to train sparse neural networks through an adequate selection of the regularization function and constraint set.

5.1. Proximal Gradient Optimization

Many of the techniques that are used to reduce the size of large neural network models rely on using regularization, like ℓ_1-regularization, in order to introduce unstructured sparsity into the model. Applying gradient descent methods to these objective functions involves computing the gradient of a term that is not differentiable at a certain point in its domain. The common solution to this problem is to compute the subgradient of the function, which exists even though the function itself is non-differentiable.

The subgradient $\boldsymbol{g}$ of a function $f(\boldsymbol{w})$ at a specific point $\boldsymbol{w}$ is defined as:

$$f(\boldsymbol{z}) \geq f(\boldsymbol{w}) + \boldsymbol{g}^T(\boldsymbol{z} - \boldsymbol{w}) \tag{5.1}$$

for all $\boldsymbol{z} \in \mathbf{dom} f$. The subdifferential $\partial f(\boldsymbol{w})$ is the set of all possible subgradients at $\boldsymbol{w}$, and a function is called subdifferentiable if it is subdifferentiable at all $\boldsymbol{w} \in \mathbf{dom} f$.

As an example, consider $f(w) = |w|$, which is non-differentiable at $w = 0$. Applying Equation (5.1) to this example:

$$|z| \geq gz \tag{5.2}$$

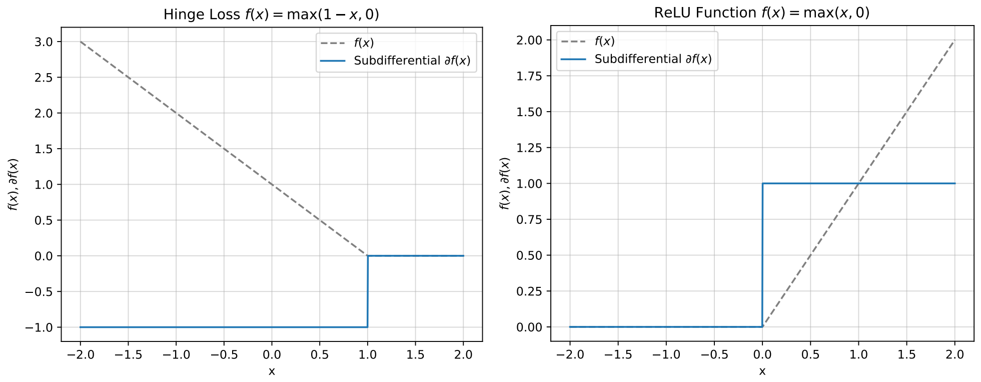
$$\begin{cases} z \geq gz \rightarrow g \leq 1, & \text{if } z \geq 0 \\ -z \geq gz \rightarrow g \geq -1, & \text{if } z < 0 \end{cases} \tag{5.3}$$

$$\tag{5.4}$$

which results in $g \in \partial f(w = 0) = [-1, 1]$. Non-differentiable functions that are subdifferentiable include the Hinge loss, ReLU activation functions, Max-Pooling layers and the ℓ_1-regularization. Their subdifferential is shown in Figure 5.1.

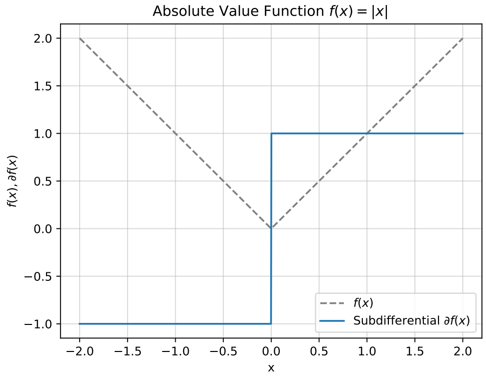
Even though the subgradient method is often times called the subgradient descent method, it is not a descent method per se. There is no guarantee that a step in the direction of the negative subgradient will result in a lower objective value, no matter the step size (see [168] Chap. 6 for an in-depth discussion). An example of this can be seen in Figure 5.2.

Applied to deep neural network training, using the visualization tools introduced in the previous sections reveals that regular stochastic subgradient descent with momentum struggles to converge to sparse solutions for objective functions with ℓ_1-regularization. Figure 5.3 shows the trajectory of a ResNet-32 architecture trained on CIFAR-100 for 100 epochs. In order to generate the plots two parameters were randomly selected among all parameters of the converged network with an absolute value smaller than 10^{-7}. The plot depicts the trajectory along these to randomly selected parameters. One can see that the network struggles to force the parameters towards zero, and their value oscillates while slowly approaching zero. Once the parameters are close to zero, they start to bounce around without converging ex-



(a) *Depiction of the Hinge loss together with its subdifferential on the interval* $x \in [-2, 2]$. *The function is non-differentiable at the point* $x_0 = 1$. *The subdifferential at this point is* $\partial f(x_0) = [-1, 0]$.

(b) *Depiction of the ReLU function together with its subdifferential on the interval* $x \in [-2, 2]$. *The function is non-differentiable at the point* $x_0 = 0$. *The subdifferential at this point is* $\partial f(x_0) = [0, 1]$.



(c) *Depiction of the absolute value function together with its subdifferential on the interval* $x \in [-2, 2]$. *The function is non-differentiable at the point* $x_0 = 0$. *The subdifferential at this point is* $\partial f(x_0) = [-1, 1]$.

Figure 5.1.: *Visualization of (a) the hinge loss, (b) the ReLU function and (c) the absolute value function together with their corresponding subdifferentials.*

actly to this value. The final value of the parameters after 100 epochs of training is $(w_1^\star, w_2^\star) = (-1.9 \times 10^{-8}, -3.4 \times 10^{-8})$.

In order to deal with non-differentiable functions that are subdifferentiable, the (local) minimum of the objective function $f(\boldsymbol{w})$ is evaluated by finding the zero vector of its subdifferential ∂f:

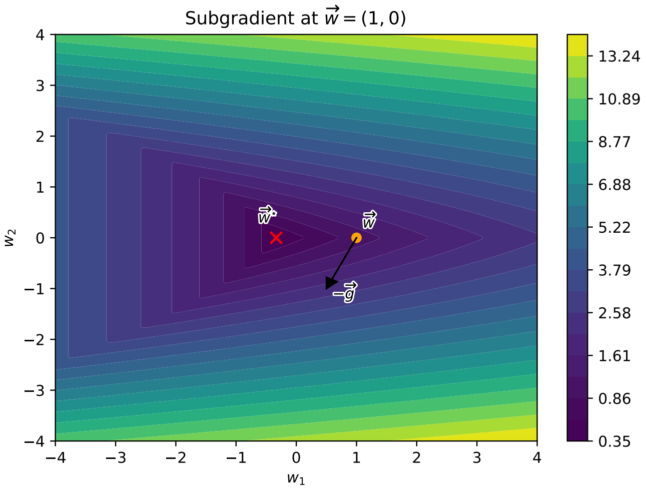
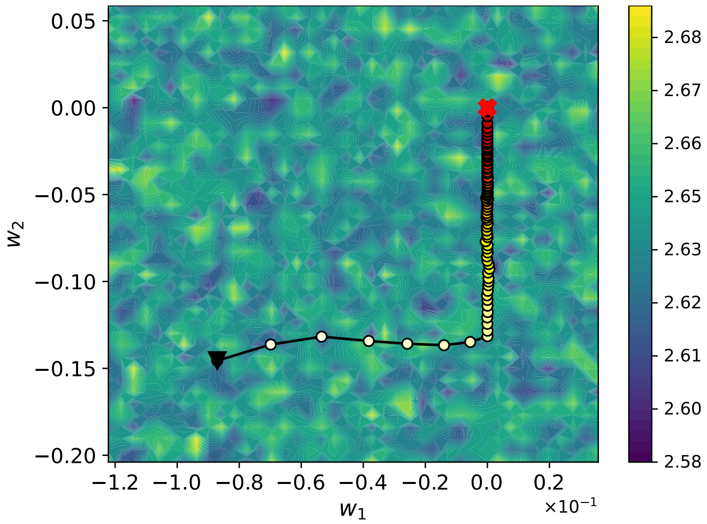


Figure 5.2.: *Depiction of the contour lines of an objective function* $f(w_1, w_2)$ *together with the optimum at* $\boldsymbol{w}^\star = (-1/3, 0)$*, and the point* $\boldsymbol{w} = (1, 0)$*. The objective function is* $f(w_1, w_2) = \max\left(-w_1, 1/2w_1^2 + 1/2(w_2+1)^2, 1/2w_1^2 + 1/2(w_2-1)^2\right)$*. At the point* $\boldsymbol{w}$*, the subgradient* $\boldsymbol{g} \in \partial f(\boldsymbol{w})$ *is* $\boldsymbol{g} = (1/2, 1)$*. The direction of the next iteration implied by the subgradient method is shown by the arrow* $-\boldsymbol{g}$*. Observe that no step size will result in a reduction of the objective function.*

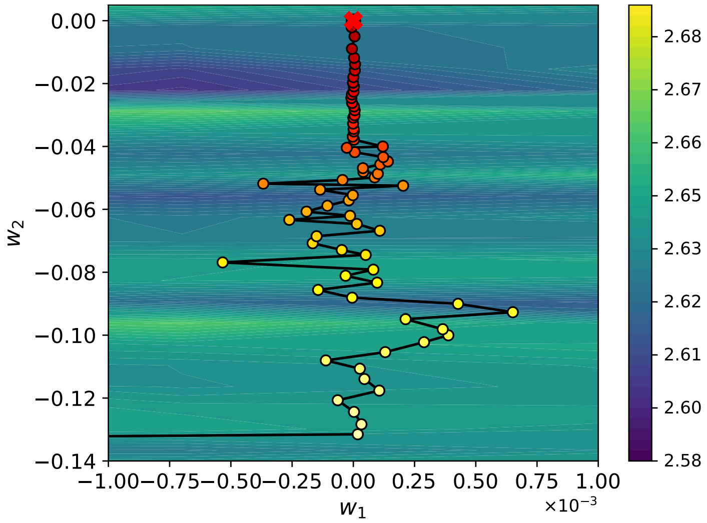
$$\min f(\boldsymbol{w}) \tag{5.5}$$
$$\boldsymbol{0} \in \partial f(\boldsymbol{w}) \tag{5.6}$$
$$\boldsymbol{0} \in \lambda \partial f(\boldsymbol{w}) \tag{5.7}$$
$$\boldsymbol{w} \in (I + \lambda \partial f)(\boldsymbol{w}) \tag{5.8}$$
$$\boldsymbol{w} = (I + \lambda \partial f)^{-1}(\boldsymbol{w}) \tag{5.9}$$

The resulting expression on the right-hand side of Equation (5.9) is the resolvent of the subdifferential operator [173]. Points that satisfy this resolvent equation coincide with solutions to Equation (5.5) [187]. The resolvent of the subdifferential can be rewritten as follows:

$$\boldsymbol{z} \in (I + \lambda \partial f)^{-1}(\boldsymbol{w}) \tag{5.10}$$
$$\boldsymbol{w} \in (I + \lambda \partial f)(\boldsymbol{z}) \tag{5.11}$$
$$\boldsymbol{0} \in \partial f(\boldsymbol{z}) + \frac{1}{\lambda}(\boldsymbol{z} - \boldsymbol{w}) \tag{5.12}$$
$$\boldsymbol{0} \in \partial_{\boldsymbol{z}} \left(f(\boldsymbol{z}) + \frac{1}{2\lambda} \|\boldsymbol{z} - \boldsymbol{w}\|_2^2 \right) \tag{5.13}$$
$$\boldsymbol{z} = \arg\min_{\boldsymbol{v}} \left(f(\boldsymbol{v}) + \frac{1}{2\lambda} \|\boldsymbol{v} - \boldsymbol{w}\|_2^2 \right) \tag{5.14}$$
$$\boldsymbol{z} = \operatorname{prox}_{\lambda f}(\boldsymbol{w}) \tag{5.15}$$



(a) *Loss landscape visualization of ResNet-32 trained with stochastic subgradient descent on CIFAR-100. The starting vector is* $(w_1^{(0)}, w_2^{(0)}) = (-0.09, -0.15)$.



(b) *Closeup of the loss landscape visualization. One can see how parameter* w_2 *slowly converges toward zero while* w_1 *oscillates around zero.*

Figure 5.3.: *Visualization of the trajectory of a ResNet-32 architecture trained on CIFAR-100 for 100 epochs. The network was trained with a batch-size of 128, a learning rate of* 4.91×10^{-5} *and momentum of 0.9. The value of the regularization parameter is set to* $\mu = 0.001$. *The learning rate was determined using random-search. The network reaches a final validation accuracy of* 34.48%.

One can see that Equation (5.14) minimizes a convex function, which is the sum of function $f(w)$ and the squared ℓ_2-norm, which is strongly convex. The right hand side of the equation is called the proximal mapping of f and is defined as follows [173]:

$$\mathrm{prox}_{\lambda f}(\boldsymbol{v}) = \arg\min_{\boldsymbol{w}} \left(f(\boldsymbol{w}) + \frac{1}{2\lambda} \|\boldsymbol{w} - \boldsymbol{v}\|_2^2 \right) \tag{5.16}$$

There are cases where there exist closed form solutions to the proximal operator. Some examples will be shown in the next section. The proximal operator allows for a proper treatment when dealing with non-differentiable functions that are subdifferentiable. Contrary to the commonly used subgradient method, using the proximal operator results in true descent methods for appropriately chosen step sizes.

The proximal point algorithm [187], which allows to converge to the set of minimizers of a closed proper convex function f, is given by:

$$\boldsymbol{w}^{(k+1)} = \mathrm{prox}_{\lambda f}(\boldsymbol{w}^{(k)}). \tag{5.17}$$

The function f is minimized by the point $\boldsymbol{w}^{\star}$ if and only if

$$\boldsymbol{w}^{\star} = \mathrm{prox}_{\lambda f}(\boldsymbol{w}^{\star}) \tag{5.18}$$

It can be shown that the proximal point algorithm can be interpreted as disappearing Tikhonov regularization or as a backward Euler method for solving the gradient flow for f [173].

The proximal gradient method, on the other hand, is used to converge to the set of minimizers of a composite objective function. In order to derive the proximal gradient method, consider the following objective:

$$\min f(\boldsymbol{w}) + r(\boldsymbol{w}) \tag{5.19}$$

where f is smooth (differentiable and L-Lipschitz continuous) and r convex and nonsmooth. The proximal gradient method is derived as follows:

$$\begin{aligned}
\min f(\boldsymbol{w}) &+ r(\boldsymbol{w}) && (5.20)\\
\mathbf{0} &\in \nabla f(\boldsymbol{w}) + \partial r(\boldsymbol{w}) && (5.21)\\
\mathbf{0} &\in \lambda \nabla f(\boldsymbol{w}) + \lambda \partial r(\boldsymbol{w}) && (5.22)\\
\boldsymbol{w} &\in \lambda \nabla f(\boldsymbol{w}) + (I + \lambda \partial r)(\boldsymbol{w}) && (5.23)\\
\boldsymbol{w} - \lambda \nabla f(\boldsymbol{w}) &\in (I + \lambda \partial r)(\boldsymbol{w}) && (5.24)\\
\boldsymbol{w} &= (I + \lambda \partial r)^{-1}(\boldsymbol{w} - \lambda \nabla f(\boldsymbol{w})) && (5.25)\\
\boldsymbol{w} &= \mathrm{prox}_{\lambda r}(\boldsymbol{w} - \lambda \nabla f(\boldsymbol{w})) && (5.26)
\end{aligned}$$

The proximal gradient algorithm for minimizing the objective function $f(\boldsymbol{w}) + r(\boldsymbol{w})$ is thus given by

$$\boldsymbol{w}^{(k+1)} = \mathrm{prox}_{\lambda_k r}\left(\boldsymbol{w}^{(k)} - \lambda_k \nabla f(\boldsymbol{w}^{(k)}) \right). \tag{5.27}$$

Note that typically there are different ways to split the objective functions into different terms. Different splits will result in different implementations of the proximal gradient method (see [18] for different examples).

The composite objective function is minimized by the point $\boldsymbol{w}^{\star}$ if and only if

$$\boldsymbol{w}^\star = \operatorname{prox}_{\lambda r}\left(\boldsymbol{w}^\star - \lambda \nabla f(\boldsymbol{w}^\star)\right). \tag{5.28}$$

As explained above, the subgradient method is not a descent method and is thus slow to converge in many situations where the objective function is non-differentiable. This is especially true for neural network optimization, where the network often times consists of non-differential functions like ReLU activation functions and Max-Pooling layers, or the objective function is regularized by non-differentiable functions, like the commonly used ℓ_1-regularization.

Proximal gradient methods serve as a solution for properly handling this problem. Faster (or any) convergence is especially valuable for very large neural networks, where each iteration is very time consuming and resource intensive. A toy example, which is shown in Figure 5.4, serves as a visual example of the difference in convergence between the subgradient method compared to the proximal point algorithm.

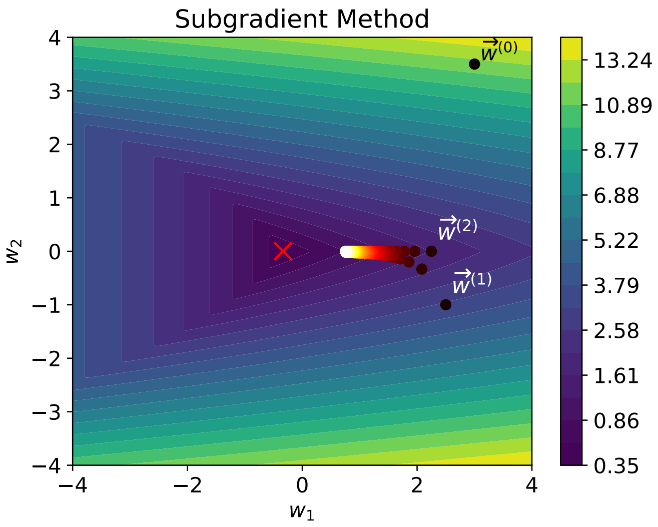
The rest of this chapter will focus on the problem of training neural networks under constraints and regularization. It is formulated as an optimization problem

$$\underset{\boldsymbol{w}\in\mathbb{W}\subseteq\mathbb{R}^n}{\text{minimize}} \underbrace{\frac{1}{m}\sum_{i=1}^{m} f_i(\boldsymbol{w},\boldsymbol{y}_i)}_{\triangleq f(\boldsymbol{w})} + r(\boldsymbol{w}), \tag{5.29}$$

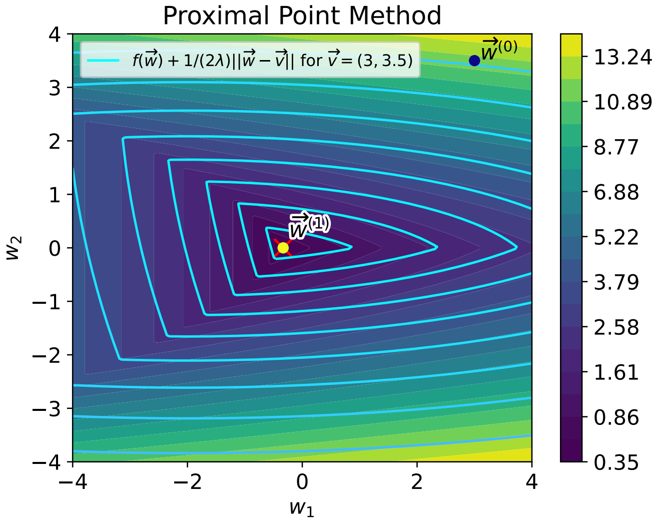
where $\boldsymbol{w}$ represents the parameter vector to optimize, $\boldsymbol{y}_i$ is the i-th training example which consists of the training input and desired output, and m is the number of training examples. The training loss f is assumed to be smooth and nonconvex with respect to $\boldsymbol{w}$, the regularization $r(\boldsymbol{w})$ is assumed to be convex but nonsmooth, proper and lower semicontinuous, and the constraint set $\mathbb{W}$ is convex and compact (closed and bounded).

Commonly, SGD is used to solve an optimization problem (5.29) when $r(\boldsymbol{w}) = 0$ and $\mathbb{W} = \mathbb{R}^n$. At each iteration, a small batch of m training samples is randomly sampled, and the resulting gradient is an unbiased estimate of the true gradient. Therefore SGD usually moves in the descent direction, see [23]. SGD can be sped up by replacing the current gradient estimate with momentum that aggregates all gradients from past iterations. Despite the success and popularity of SGD, its convergence has been an open problem. Assuming f is convex, the convergence analysis was first attempted in [110] and later concluded in [185]. The proof for non-convex f is given later in [38, 128].

The regularization function r is commonly used in machine learning to promote a certain structure in the optimal solution, such as sparsity as in feature selection and compressed sensing, or a zero-mean-Gaussian prior on the parameters [14, 29]. It can be interpreted as a penalty function because the value $r(\boldsymbol{w}^\star)$ will be small at the optimal point $\boldsymbol{w}^\star$ of problem (5.29). The Tikhonov regularization $r(\boldsymbol{w}) = \mu\|\boldsymbol{w}\|_2^2$ for some specified constant μ, for example, can be used to reduce ill-conditioning and ensure that the size of the weights does not become excessively large. Another commonly used regularization, the ℓ_1-norm where $r(\boldsymbol{w}) = \mu\|\boldsymbol{w}\|_1 = \mu\sum_{j=1}^{n}|w_j|$ (the convex surrogate of the ℓ_0-norm), would encourage a sparse solution. In the context of neural networks, it is used to (i) promote a sparse neural network (SNN) to alleviate overfitting and to allow for improved generalization, (ii) accelerate the training



(a) *Subgradient method trained for 50 iterations using a decaying learning rate defined by* $\varepsilon_t = 1/t$*. This method is not able to converge to the optimal point after 50 iterations. The path through the loss landscape oscillates around the* $w_2 = 0$ *axis.*



(b) *Proximal point method trained for one iteration with* $\lambda = 10$*. This figure depicts the contour plot of the function* $f(\boldsymbol{w})$*, together with the contour lines of the objective function inside the proximal operator for* $\boldsymbol{v} = (3, 3.5)$*. The minimum of the function that is minimized by the proximal operator coincides with the minimum of* $f(w)$*.*

Figure 5.4.: *Visualization of the difference between the subgradient method and proximal point method. The objective function is* $f(w_1, w_2) = \max\left(-w_1, 1/2w_1^2 + 1/2(w_2+1)^2, 1/2w_1^2 + 1/2(w_2-1)^2\right)$*, with the optimal point depicted by the red cross at* $\boldsymbol{w}^\star = (-1/3, 0)$*. Both methods are initialized at* $\boldsymbol{w}^{(0)} = (3, 3.5)$*. Plot (a) shows the subgradient method for 50 iterations with decaying learning rate* $\varepsilon_t = 1/t$*. This learning rate schedule yields the best result compared to different constant learning rates. Plot (b) shows the proximal gradient method, which converged after one iteration. This plot also depicts contour lines of the objective function that is minimized in the proximal operator. In this example the scaling parameter is set to* $\lambda = 10$*.*

process, and (iii) prune the network to reduce its complexity, see [146] and [65] as well as Section 2.2 for more detail.

Technically, analyzing the regularizations is difficult since several regularly used convex regularizers, such as ℓ_1-norm, are nonsmooth. When $w = 0$, the gradient of $|w|$ in current TensorFlow [152] implementations is simply set to zero. As explained in Section 5.1, this is the stochastic subgradient descent method, which usually exhibits slow convergence rate. Magnitude pruning and variational dropout are two further strategies for promoting an SNN, see [65].

Although regularization can be interpreted as a constraint from the duality theory, it is sometimes preferable to employ explicit constraints, such as $\sum w_j^2 \leq \alpha$, where the summation is over the same layer's weights. This is useful when it is already known how to choose α. Another example is the weights' lower and upper bounds, which are $\boldsymbol{l} \leq \boldsymbol{w} \leq \boldsymbol{u}$ for specified $\boldsymbol{l}$ and $\boldsymbol{u}$. Constraints, unlike regularization, do not encourage weights to stay in a local neighborhood of the initial weight. For further information, see Chapter 7.2 of [76]. The set $\mathbb{W}$ represents such explicit constraints, but stochastic gradient algorithms pose an additional challenge because the new weight obtained from the SGD method (with or without momentum) must be projected back to the set $\mathbb{W}$ to preserve its feasibility. However, because projection is a nonlinear operator, the random gradient's unbiasedness would be lost. As a result, constrained problem convergence analysis is substantially more involved than convergence analysis for unconstrained problems.

The rest of this chapter will propose a convergent proximal-type stochastic gradient algorithm (ProxSGD) to train neural networks under nonsmooth regularization and constraints. It turns out momentum plays a central role in the convergence analysis. The next section will establish that with probability 1, every limit point of the sequence generated by ProxSGD is a stationary point of the nonsmooth nonconvex problem (5.29). This is in sharp contrast to unconstrained optimization, where the convergence of the vanilla SGD method has long been well understood while the convergence of the SGD method with momentum was only settled recently. Nevertheless, the convergence rate of ProxSGD is not derived and is worth further investigating.

5.2. ProxSGD

In this section, ProxSGD is described, a new proximal algorithm for training deep neural networks with convergence guarantee.

Background and setup The following blanket assumptions on problem (5.29) are made.

- $f_i(\boldsymbol{w}, \boldsymbol{y}^{(i)})$ is smooth (continuously differentiable) but not necessarily convex.
- $\nabla_{\boldsymbol{w}} f_i(\boldsymbol{w}, \boldsymbol{y}^{(i)})$ is Lipschitz continuous with a finite constant L_i for any $\boldsymbol{y}^{(i)}$. Thus $\nabla f(\boldsymbol{w})$ is Lipschitz continuous with constant $L \triangleq \frac{1}{m}\sum_{i=1}^{m} L_i$.
- $r(\boldsymbol{w})$ is convex, proper and lower semicontinuous (not necessarily smooth).

- $\mathbb{W}$ is convex and compact.

The goal is to develop algorithms that can find a stationary point of (5.29). A stationary point $\boldsymbol{w}^\star$ satisfies the optimality condition: at $\boldsymbol{w} = \boldsymbol{w}^\star$,

$$(\boldsymbol{w} - \boldsymbol{w}^\star)^T \nabla f(\boldsymbol{w}^\star) + r(\boldsymbol{w}) - r(\boldsymbol{w}^\star) \geq 0, \forall \boldsymbol{w} \in \mathbb{W}. \tag{5.30}$$

When $r(\boldsymbol{w}) = 0$ and $\mathbb{W} = \mathbb{R}^n$, the deterministic optimization problem (5.29) can be solved using the (batch) gradient descent method for unconstrained optimization. As previously stated, computing the gradient of the complete dataset (which comprises of m training samples) at each iteration is computationally expensive. Instead, the gradient is estimated by a minibatch of m_k training examples. Denote the minibatch by $\mathbb{M}_k$: its elements are drawn uniformly from $\{1, 2, \ldots, m\}$ and there are m_k elements. Then the estimated gradient is

$$\boldsymbol{g}^{(k)} \triangleq \frac{1}{m_k} \sum_{i \in \mathbb{M}_k} \nabla f_i(\boldsymbol{w}^{(k)}, \boldsymbol{y}^{(i)}) \tag{5.31}$$

and it is an unbiased estimate of the true gradient.

The proposed algorithm The instantaneous gradient $\boldsymbol{g}^{(k)}$ is used to form an aggregate gradient (momentum) $\boldsymbol{v}^{(k)}$, which is updated recursively as follows

$$\boldsymbol{v}^{(k)} = (1 - \rho_k)\boldsymbol{v}^{(k-1)} + \rho_k \boldsymbol{g}^{(k)}, \tag{5.32}$$

where ρ_k is the stepsize (learning rate) for the momentum and $\rho_k \in (0, 1]$ [1].

At iteration k, the following approximation subproblem is solved and its solution is denoted as $\widehat{\boldsymbol{w}}(\boldsymbol{w}^{(k)}, \boldsymbol{v}^{(k)}, \tau^{(k)})$, or simply $\widehat{\boldsymbol{w}}^{(k)}$

$$\widehat{\boldsymbol{w}}^{(k)} \triangleq \underset{\boldsymbol{w} \in \mathbb{W}}{\operatorname{argmin}} \left\{ (\boldsymbol{w} - \boldsymbol{w}^{(k)})^T \boldsymbol{v}^{(k)} + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}^{(k)})^T \operatorname{diag}(\tau^{(k)})(\boldsymbol{w} - \boldsymbol{w}^{(k)}) + r(\boldsymbol{w}) \right\}. \tag{5.33}$$

A quadratic regularization term is incorporated so that the subproblem (5.33) is strongly convex and its modulus is the minimum element of the vector $\tau^{(k)}$, denoted as $\underline{\tau}_k$ and $\underline{\tau}_k = \min_{j=1,\ldots,N}(\tau^{(k)})_j$. Note that $\underline{\tau}_k$ should be lower bounded by a positive constant that is strictly larger than zero, so that the quadratic regularization in (5.33) will not vanish.

The difference between two vectors $\widehat{\boldsymbol{w}}^{(k)}$ and $\boldsymbol{w}^{(k)}$ specifies a direction starting at $\boldsymbol{w}^{(k)}$ and ending at $\widehat{\boldsymbol{w}}^{(k)}$. This update direction is used to refine the weight vector

$$\boldsymbol{w}^{(k+1)} = \boldsymbol{w}^{(k)} + \varepsilon_k(\widehat{\boldsymbol{w}}^{(k)} - \boldsymbol{w}^{(k)}), \tag{5.34}$$

where ε_k is a stepsize (learning rate) for the weight and $\varepsilon_k \in (0, 1]$. The learning rate allows choosing a specific point along this line. Note that $\boldsymbol{w}^{(k+1)}$ is feasible as long as $\boldsymbol{w}^{(k)}$ is feasible, as it is the convex combination of two feasible points $\boldsymbol{w}^{(k)}$ and $\widehat{\boldsymbol{w}}^{(k)}$ while the set $\mathbb{W}$ is convex.

---

[1] Note that this definition differs from the aggregate gradient that is sometimes used in the definition and implementation of optimizers like SGD with momentum or Adam, which is given by $\boldsymbol{v}^{(k)} = \rho_k \boldsymbol{v}^{(k-1)} + (1 - \rho_k)\boldsymbol{g}^{(k)}$.

The preceding steps (5.31)-(5.34) are summarized in Algorithm 7, which is referred to as proximal-type Stochastic Gradient Descent (ProxSGD), because the explicit constraint $\boldsymbol{w} \in \mathbb{W}$ in (5.33) can also be formulated implicitly as a regularization function, more specifically, the indicator function $\delta_{\mathbb{W}}(\boldsymbol{w})$. The indicator functions aid in the projection of any solution back onto the constrained set $\mathbb{W}$. If all elements of $\tau^{(k)}$ are equal, then $\widehat{\boldsymbol{w}}^{(k)}$ is exactly the proximal operator

$$\begin{aligned}\widehat{\boldsymbol{w}}^{(k)} &= \underset{\boldsymbol{w}}{\operatorname{argmin}} \left\{ \left\| \boldsymbol{w} - \left(\boldsymbol{w}^{(k)} - \frac{1}{\tau_k} \boldsymbol{v}^{(k)} \right) \right\|_2^2 + \frac{1}{\tau_k} r(\boldsymbol{w}) + \delta_{\mathbb{W}}(\boldsymbol{w}) \right\} \\ &\triangleq \operatorname{Prox}_{\frac{1}{\tau_k} r(\boldsymbol{w}) + \delta_{\mathbb{W}}(\boldsymbol{w})} \left(\boldsymbol{w}^{(k)} - \frac{1}{\tau_k} \boldsymbol{v}^{(k)} \right).\end{aligned}$$

Algorithm 7 Proximal-type Stochastic Gradient Descent (ProxSGD) Method

```
Input: $\boldsymbol{w}^{(0)} \in \mathbb{W}$, $\boldsymbol{v}^{(-1)} = \mathbf{0}$, $k = 0$, $T$, $\{\rho_k\}_{k=0}^{T}$, $\{\varepsilon_k\}_{k=0}^{T}$.
for $k = 0:1:T$ do
    1. Compute the instantaneous gradient $\boldsymbol{g}^{(k)}$ based on the minibatch $\mathbb{M}_k$:
       $$\boldsymbol{g}^{(k)} = \frac{1}{m_k} \sum_{i \in \mathbb{M}_k} \nabla_{\boldsymbol{w}} f_i(\boldsymbol{w}^{(k)}, \boldsymbol{y}^{(i)}).$$
    2. Update the momentum: $\boldsymbol{v}^{(k)} = (1 - \rho_k)\boldsymbol{v}^{(k-1)} + \rho_k \boldsymbol{g}^{(k)}$.
    3. Compute $\widehat{\boldsymbol{w}}^{(k)}$ by solving the approximation subproblem:
       $$\widehat{\boldsymbol{w}}^{(k)} = \underset{\boldsymbol{w} \in \mathbb{W}}{\arg\min} \left\{ (\boldsymbol{w} - \boldsymbol{w}^{(k)})\boldsymbol{v}^{(k)} + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}^{(k)})^T \operatorname{diag}(\tau^{(k)})(\boldsymbol{w} - \boldsymbol{w}^{(k)}) + r(\boldsymbol{w}) \right\}.$$
    4. Update the weight: $\boldsymbol{w}^{(k+1)} = \boldsymbol{w}^{(k)} + \varepsilon_k(\widehat{\boldsymbol{w}}^{(k)} - \boldsymbol{w}^{(k)})$.
end for
```

The ProxSGD optimizer in Algorithm 7 has a comparable structure to various SGD algorithms, both without and with momentum, as shown in Table 5.1, and it allows several existing algorithms to be interpreted as special cases of the proposed framework. In SGD, for example, no momentum is used, which translates to setting $\rho_k = 1$ in Algorithm 7. The learning rate for momentum in ADAM is a constant ρ, while the learning rate for the weight vector is given by $\varepsilon/(1-\rho^k)$ for some ε, which essentially equates to setting $\rho_k = \rho$ and $\varepsilon_k = \varepsilon/(1-\rho^k)$ in Algorithm 7. This interpretation also implies that the proposed convergence conditions will suffice for existing methods (although they are not meant to be the weakest conditions available in literature).

algorithm	momentum	weight	quadratic gain in subproblem	regularization	constraint set
ProxSGD	ρ_k	ε_k	$\tau^{(k)}$	convex	convex, compact
SGD (w. momentum)	$1(\rho)$	ε	$\mathbf{1}$	0	$\mathbb{R}^N$
AdaGrad	1	ε	$\sqrt{\mathbf{r}_k} + \delta\mathbf{1}^\dagger$	0	$\mathbb{R}^N$
RMSProp	1	ε	$\sqrt{\mathbf{r}_k + \delta\mathbf{1}}^\ddagger$	0	$\mathbb{R}^N$
ADAM	ρ	$\frac{\varepsilon}{1-\rho^k}$	$\sqrt{\frac{\mathbf{r}_k}{1-\beta^k}} + \delta\mathbf{1}^\ddagger$	0	$\mathbb{R}^N$
AMSGrad	ρ	ε	$\sqrt{\widehat{\boldsymbol{r}}_k}$ $\widehat{\boldsymbol{r}}_k = \max(\widehat{\boldsymbol{r}}_{k-1}, \boldsymbol{r}_k)^\ddagger$	0	$\mathbb{R}^n$
ADABound	ρ	1	$\frac{\mathbf{1}}{\mathrm{clip}(\varepsilon_k/\sqrt{\boldsymbol{r}_k}, \boldsymbol{\eta}_{lk}, \boldsymbol{\eta}_{uk})}^\ddagger$	0	$\mathbb{R}^N$

Table 5.1.: *Connection between the proposed framework and existing methods, where* ρ, β, ε *and* δ *are some predefined constants.* $^\dagger\mathbf{r}_k = \mathbf{r}_{k-1} + \boldsymbol{g}^{(k)} \odot \boldsymbol{g}^{(k)}$, $^\ddagger\mathbf{r}_k = \beta\mathbf{r}_{k-1} + (1-\beta)\boldsymbol{g}^{(k)} \odot \boldsymbol{g}^{(k)}$.

Solving the approximation subproblem (5.33) Since (5.33) is strongly convex, $\widehat{\boldsymbol{w}}^{(k)}$ is unique. Generally $\widehat{\boldsymbol{w}}^{(k)}$ in (5.33) does not admit a closed-form expression and should be solved by a generic solver. However, some important special cases that are frequently used in practice can be solved efficiently.

- The trivial case is $\mathbb{W} = \mathbb{R}^n$ and $r = 0$, where

$$\widehat{\boldsymbol{w}}^{(k)} = \boldsymbol{w}^{(k)} - \frac{\boldsymbol{v}^{(k)}}{\tau^{(k)}}, \tag{5.35}$$

where the vector division is understood to be element-wise. When $\mathbb{W} = \mathbb{R}^n$ and $r(\boldsymbol{w}) = \mu\|\boldsymbol{w}\|_1$, $\widehat{\boldsymbol{w}}^{(k)}$ has a closed-form expression that is known as the soft-thresholding operator

$$\widehat{\boldsymbol{w}}^{(k)} = S_{\frac{\mu \boldsymbol{1}}{\tau^{(k)}}}\left(\boldsymbol{w}^{(k)} - \frac{\boldsymbol{v}^{(k)}}{\tau^{(k)}}\right), \tag{5.36}$$

where $S_{\boldsymbol{a}}(\boldsymbol{b}) \triangleq \max(\boldsymbol{b} - \boldsymbol{a}, \boldsymbol{0}) - \max(-\boldsymbol{b} - \boldsymbol{a}, \boldsymbol{0})$ [14].

- If $\mathbb{W} = \mathbb{R}^n$ and $r(\boldsymbol{w}) = \mu\|\boldsymbol{w}\|_2$ and $\tau^{(k)} = \tau\boldsymbol{I}$ for some τ, then [173]

$$\widehat{\boldsymbol{w}}^{(k)} = \begin{cases} \left(1 - \mu/\|\tau\boldsymbol{w}^{(k)} - \boldsymbol{v}^{(k)}\|_2\right)(\boldsymbol{w}^{(k)} - \boldsymbol{v}^{(k)}/\tau), & \text{if } \|\tau\boldsymbol{w}^{(k)} - \boldsymbol{v}^{(k)}\|_2 \geq \mu, \\ 0, & \text{otherwise.} \end{cases} \tag{5.37}$$

If $\boldsymbol{w}$ is divided into blocks $\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots$, the ℓ_2-regularization is commonly used to promote block sparsity (rather than element sparsity by ℓ_1-regularization).

- When a bound constraint $\boldsymbol{l} \leq \boldsymbol{w} \leq \boldsymbol{u}$ exists, $\widehat{\boldsymbol{w}}^{(k)}$ can be simply obtained by first solving the approximation subproblem (5.33) without the bound constraint and then projecting the optimal point onto the interval $[\boldsymbol{l}, \boldsymbol{u}]$. For example, when $\mathbb{W} = \mathbb{R}^n$ and $r = 0$,

$$\widehat{\boldsymbol{w}}^{(k)} = \left[\boldsymbol{w}^{(k)} - \frac{\boldsymbol{v}^{(k)}}{\tau^{(k)}}\right]_{\boldsymbol{l}}^{\boldsymbol{u}}, \tag{5.38}$$

with $[\boldsymbol{w}]_{\boldsymbol{l}}^{\boldsymbol{u}} = \operatorname{clip}(\boldsymbol{w}, \boldsymbol{l}, \boldsymbol{u}) \triangleq \min(\max(\boldsymbol{w}, \boldsymbol{l}), \boldsymbol{u})$.

- If the constraint function is quadratic: $\mathbb{W} = \{\boldsymbol{w} : \|\boldsymbol{w}\|_2^2 \leq 1\}$, $\widehat{\boldsymbol{w}}^{(k)}$ has a semi-analytical expression (up to a scalar Lagrange multiplier which can be found efficiently by the bisection method).

Approximation subproblem This paragraph will explain why the weights are updated by solving an approximation subproblem (5.33). First, $\widetilde{f}$ denotes the smooth part of the objective function in (5.33). Clearly it depends on $\boldsymbol{w}^{(k)}$ and $\boldsymbol{v}^{(k)}$ (and thus $\mathbb{M}_k$), while $\boldsymbol{w}^{(k)}$ and $\boldsymbol{v}^{(k)}$ depend on the old weights $\boldsymbol{w}^{(0)}, \ldots, \boldsymbol{w}^{(k-1)}$ and momentum $\boldsymbol{v}^{(0)}, \ldots, \boldsymbol{v}^{(k-1)}$. Define $\mathbb{F}(k) \triangleq \{\boldsymbol{w}^{(0)}, \ldots, \boldsymbol{w}^{(k)}, \mathbb{M}_0, \ldots, \mathbb{M}_k\}$ as a shorthand notation for the trajectory generated by ProxSGD. Formally write $\widetilde{f}$ as

$$\widetilde{f}(\boldsymbol{w}; \mathbb{F}(k)) \triangleq (\boldsymbol{w} - \boldsymbol{w}^{(k)})^T \boldsymbol{v}^{(k)} + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}^{(k)})^T \operatorname{diag}(\tau^{(k)})(\boldsymbol{w} - \boldsymbol{w}^{(k)}). \tag{5.39}$$

It follows from the optimality of $\widehat{\boldsymbol{w}}^{(k)}$ that

$$\widetilde{f}(\boldsymbol{w}^{(k)};\mathbb{F}(k))+r(\boldsymbol{w}^{(k)}) \geq \widetilde{f}(\widehat{\boldsymbol{w}}^{(k)};\mathbb{F}(k))+r(\widehat{\boldsymbol{w}}^{(k)}).$$

After inserting (5.39) and reorganizing the terms, the above inequality becomes

$$(\widehat{\boldsymbol{w}}^{(k)}-\boldsymbol{w}^{(k)})^T\boldsymbol{v}^{(k)}+r(\widehat{\boldsymbol{w}}^{(k)})-r(\boldsymbol{w}^{(k)}) \leq -\tau_k\|\widehat{\boldsymbol{w}}^{(k)}-\boldsymbol{w}^{(k)}\|_2^2. \tag{5.40}$$

Since $\nabla f(\boldsymbol{w})$ is Lipschitz continuous with constant L, it follows that

$$\begin{aligned}
&f(\boldsymbol{w}^{(k+1)})+r(\boldsymbol{w}^{(k+1)})-(f(\boldsymbol{w}^{(k)})+r(\boldsymbol{w}^{(k)}))\\
&\leq (\boldsymbol{w}^{(k+1)}-\boldsymbol{w}^{(k)})^T\nabla f(\boldsymbol{w}^{(k)})+\frac{L}{2}\|\boldsymbol{w}^{(k+1)}-\boldsymbol{w}^{(k)}\|_2^2+r(\boldsymbol{w}^{(k+1)})-r(\boldsymbol{w}^{(k)}) \qquad (5.41)\\
&\leq \varepsilon_k\left((\widehat{\boldsymbol{w}}^{(k)}-\boldsymbol{w}^{(k)})^T\nabla f(\boldsymbol{w}^{(k)})+r(\widehat{\boldsymbol{w}}^{(k)})-r(\boldsymbol{w}^{(k)})+\frac{L}{2}\varepsilon_k\|\widehat{\boldsymbol{w}}^{(k)}-\boldsymbol{w}^{(k)}\|_2^2\right), \qquad (5.42)
\end{aligned}$$

where the first inequality follows from the descent lemma (applied to f) and the second inequality follows from the Jensen's inequality of the convex function r and the update rule (5.34).

If $\boldsymbol{v}^{(k)}=\nabla f(\boldsymbol{w}^{(k)})$ (which is true asymptotically as is shown shortly later), by replacing $\nabla f(\boldsymbol{w}^{(k)})$ in (5.42) by $\boldsymbol{v}^{(k)}$ and inserting (5.40) into (5.42), one obtains

$$f(\boldsymbol{w}^{(k+1)})+r(\boldsymbol{w}^{(k+1)})-(f(\boldsymbol{w}^{(k)})+r(\boldsymbol{w}^{(k)})) \leq \varepsilon_k\left(\frac{L}{2}\varepsilon_k-\tau_k\right)\|\widehat{\boldsymbol{w}}^{(k)}-\boldsymbol{w}^{(k)}\|_2^2. \tag{5.43}$$

When $\varepsilon_k<\frac{2\tau_k}{L}$, the right hand side (RHS) will be negative: this will eventually be satisfied by using a decaying ε_k. This implies that after each iteration, the proposed update (5.34) will reduce the objective value of (5.29).

Momentum and algorithm convergence. The momentum (gradient averaging step) in (5.32) turns out to be essential for ProxSGD convergence. The aggregate gradient $\boldsymbol{v}^{(k)}$ will converge to the true (unknown) gradient $\nabla f(\boldsymbol{w}^{(k)})$ under some mild technical assumptions. This remark is made rigorous in the following theorem.

Theorem 5.1 *Assume that the unbiased gradient $\boldsymbol{g}^{(k)}$ has a bounded second moment*

$$\mathbb{E}\left[\|\boldsymbol{g}^{(k)}\|_2^2|\mathbb{F}(k)\right] \leq C, \tag{5.44}$$

for some finite and positive constant C, and the sequence of stepsizes $\{\rho_k\}$ and $\{\varepsilon_k\}$ satisfy

$$\sum_{k=0}^{\infty}\rho_k=\infty, \sum_{k=0}^{\infty}\rho_k^2<\infty, \sum_{k=0}^{\infty}\varepsilon_k=\infty, \sum_{k=0}^{\infty}\varepsilon_k^2<\infty, \lim_{k\to\infty}\frac{\varepsilon_k}{\rho_k}=0. \tag{5.45}$$

Then $\lim_{k\to\infty}\|\boldsymbol{v}^{(k)}-\nabla f(\boldsymbol{w}^{(k)})\|=0$, and every limit point of the sequence $\{\boldsymbol{w}^{(k)}\}$ is a stationary point of (5.29) w.p.1.

Proof Under the assumptions (5.44) and (5.45), it follows from Lemma 1 of [192] that $\boldsymbol{v}^{(k)} \to \nabla f(\boldsymbol{w}^{(k)})$. Since the descent direction $\widehat{\boldsymbol{w}}^{(k)} - \boldsymbol{w}^{(k)}$ is a descent direction in view of (5.40), the convergence of the ProxSGD algorithm can be obtained by generalizing the line of analysis in Theorem 1 of [240] for smooth optimization problems. The detailed proof is included in the appendix.

There are some comments to consider on the convergence analysis in Theorem 5.1.

- In stochastic optimization and SGD, standard assumptions include the bounded second moment assumption on the gradient $\boldsymbol{g}$ in (5.44) and decreasing stepsizes in (5.45). It's worth noting that ε_k should decrease faster than ρ_k in order for $\boldsymbol{v}^{(k)} \to \nabla f(\boldsymbol{w}^{(k)})$ to hold. This result is more interesting from a theoretical perspective and in practice, even the relationship $\varepsilon_k/\rho_k = a$ for some constant a that is smaller than 1 usually yields satisfactory performance, as will be show numerically in the next section.
- According to Theorem 5.1, the momentum $\boldsymbol{v}^{(k)}$ converges to the (unknown) true gradient $\nabla f(\boldsymbol{w}^{(k)})$, hence the ProxSGD algorithm eventually behaves similar to the (deterministic) gradient descent algorithm. This property is essential to guarantee the convergence of the ProxSGD algorithm.
- The quadratic gain $\underline{\tau}_k$ in the approximation subproblem (5.33) should be lower bounded by a positive constant to guarantee theoretical convergence (and it does not even have to be time-varying). In practice, there are various rationales to define it (see Table 5.1), and they lead to different empirical convergence speed and generalization performance.
- Due to functions and layers such as the nonsmooth ReLU activation function, batch normalization, and dropout, the technical assumptions in Theorem 5.1 may not always be fully satisfied by the neural networks implemented in practice. Nonetheless, Theorem 5.1 still provides valuable guidance on the algorithm's practical performance and the choice of the hyperparameters.

5.3. ProxSGD with ℓ_1-Regularization

As mentioned in Section 2.2, ℓ_1-regularization helps reduce the size of large neural networks by introducing unstructured sparsity in the parameters. The loss function looks as follows

$$\underset{\boldsymbol{w}}{\text{minimize}} \quad \frac{1}{m}\sum_{i=1}^{m} f_i(\boldsymbol{w}, \boldsymbol{y}_i) + \mu||\boldsymbol{w}||_1. \tag{5.46}$$

Typical optimizers like SGD or Adam rely on stochastic subgradient descent in order to train the model and thus suffer from slow convergence. Using the soft-thresholding function introduced in Equation (5.36), the proximal operator can be solved using a closed form solution. The soft-thresholding function is shown in Figure 5.5. Observe that the soft-thresholding with parameter λ maps any input variable w to zero whenever $|w| < \lambda$. Any other input value will leave the variable w unaffected.

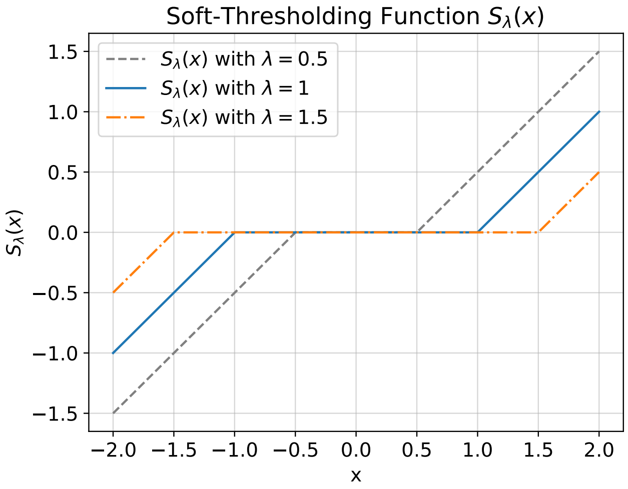


Figure 5.5.: *Depiction of the soft-thresholding function for different values of the parameter* λ.

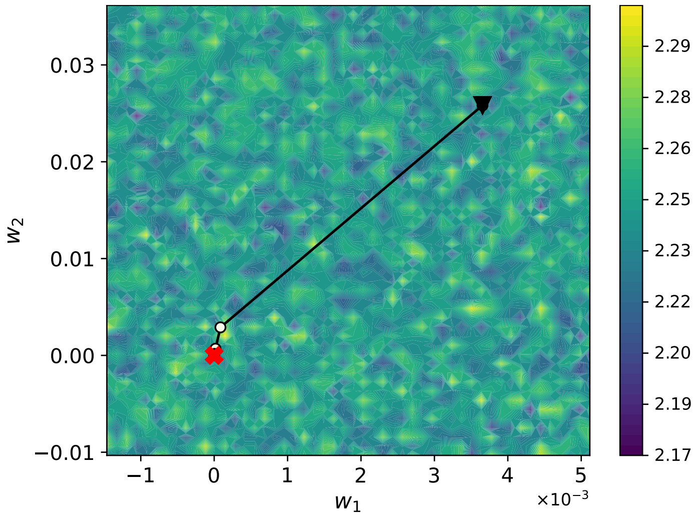
For clarity, the entire algorithm for minimizing the objective defined in Equation (5.46) is summarized in Algorithm 8.

Algorithm 8 ProxSGD for $r(\boldsymbol{w}) = \|\boldsymbol{w}\|_1$

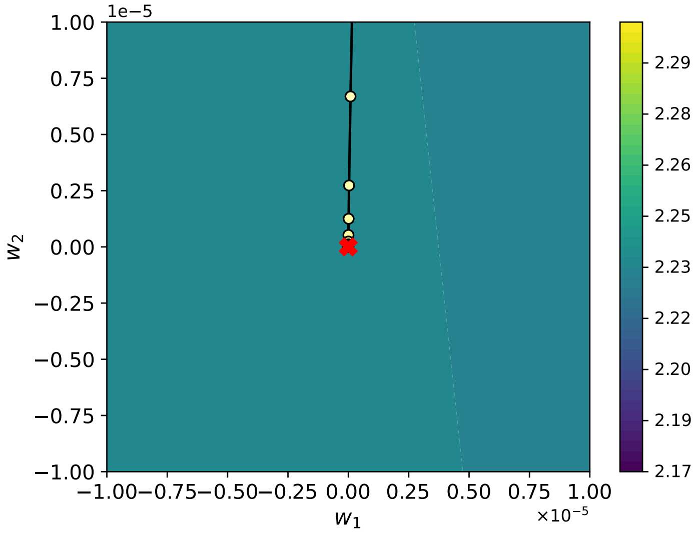
```
Input: $\boldsymbol{w}^{(0)} \in \mathbb{W}$, $\boldsymbol{v}^{(-1)} = \mathbf{0}$, $k = 0$, $T$, $\{\rho_k\}_{k=0}^{T}$, $\{\varepsilon_k\}_{k=0}^{T}$.
for $k = 0 : 1 : T$ do
    1. Compute the instantaneous gradient $\boldsymbol{g}^{(k)}$ based on the minibatch $\mathbb{M}_k$:
       $\boldsymbol{g}^{(k)} = \frac{1}{m_k} \sum_{i \in \mathbb{M}_k} \nabla_{\boldsymbol{w}} f_i(\boldsymbol{w}^{(k)}, \boldsymbol{y}^{(i)})$.
    2. Update the momentum: $\boldsymbol{v}^{(k)} = (1 - \rho_k)\boldsymbol{v}^{(k-1)} + \rho_k \boldsymbol{g}^{(k)}$.
    3. Compute $\widehat{\boldsymbol{w}}^{(k)}$:
       $\widehat{\boldsymbol{w}}^{(k)} = S_{\frac{\mu\mathbf{1}}{\tau^{(k)}}}\left(\boldsymbol{w}^{(k)} - \frac{\boldsymbol{v}^{(k)}}{\tau^{(k)}}\right)$.
    4. Update the weight: $\boldsymbol{w}^{(k+1)} = \boldsymbol{w}^{(k)} + \varepsilon_k(\widehat{\boldsymbol{w}}^{(k)} - \boldsymbol{w}^{(k)})$.
end for
```

5.3.1. Experiments

Compared to the example of stochastic subgradient descent with momentum, which was shown in Figure 5.3, the same experiment is repeated using ProxSGD with ℓ_1-regularization. The results are depicted in Figure 5.6. The ResNet-32 architecture is trained using the same hyperparameters as in the stochastic subgradient descent case, only the learning rate is changed to $\varepsilon = 0.06$ after running a random search. One can observe that contrary to the stochastic subgradient descent method, ProxSGD is able to quickly converge to sparse solutions. The final value of the parameters after 100 epochs of training is $(w_1^\star, w_2^\star) = (-8.5 \times 10^{-24}, 2.5 \times 10^{-21})$, which is significantly more sparse than what is achieved using subgradients.



(a) *Loss landscape visualization of ResNet-32 trained with ProxSGD on CIFAR-100. The starting vector is* $(w_1^{(0)}, w_2^{(0)}) = (0.004, 0.026)$.



(b) *Closeup of the loss landscape visualization. One can see how both parameters quickly converge toward zero.*

Figure 5.6.: *Visualization of the trajectory of a ResNet-32 architecture trained on CIFAR-100 for 100 epochs. The network was trained with a batch-size of 128, a learning rate of* $\varepsilon = 0.06$ *and momentum of* $\rho = 0.9$. *The value of the regularization parameter is set to* $\mu = 0.001$. *The learning rate was determined using random-search. The network reaches a final validation accuracy of* 43.57%.

In the remaining subsection, the performance of ProxSGD is evaluated by training the DenseNet-201 network [95] on CIFAR-100 [114]. DenseNet-201 is the deepest topology of the DenseNet family and belonged to the state-of-the-art networks in image classification tasks at the time of its introduction. Three different optimizers are used to train the network: SGD with momentum, ADAM and ProxSGD. The learning rate is not explicitly decayed during training to ensure a fair comparison between the different methods. Furthermore, random grid-search was used to determine the ideal hyperparameters for each algorithm, and all curves were averaged over five runs. For training, a batch size of 128 is chosen. For ProxSGD, the regularization parameter is $\mu = 10^{-5}$, the learning rate for the weight and momentum is, respectively,

$$\varepsilon_k = \frac{0.15}{(k+4)^{0.5}}, \;\; \rho_k = \frac{0.9}{(k+4)^{0.5}}.$$

For ADAM, $\varepsilon = 6 \cdot 10^{-4}$ and $\rho = 0.1$. SGD with momentum uses a learning rate of $\varepsilon = 6 \cdot 10^{-3}$ and a momentum of 0.9 (equivalent to $\rho = 0.1$). The regularization parameter for both ADAM and SGD with momentum is $\mu = 10^{-4}$.

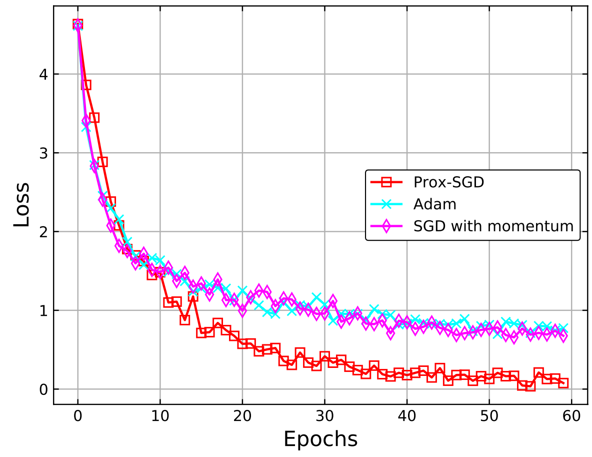
The performance of ProxSGD and other methods for DenseNet-201 trained on CIFAR-100 is shown in Figure 5.7. ProxSGD achieves the lowest training loss after 10 epochs, as seen in Figure 5.7a. Figure 5.7b illustrates that all algorithms attain similar accuracy, although ProxSGD outperforms the other two during the early phases of training. ProxSGD is able to accomplish this with a much sparser network, as seen in Figure 5.7c. When looking at the zoomed-in area of Figure 5.7c, one can see that SGD with momentum has approximately 70% of the weights at zero, whereas most of the weights learned by ADAM are not exactly zero (although they are very small). ProxSGD, on the other hand, achieves a sparsity of 92-94%.

Figure 5.8 shows several ProxSGD runs with various learning rates, demonstrating that ProxSGD is substantially more efficient in constructing a sparse neural network, irrespective of the hyperparameters (related to the learning rate). The performance of various different initial learning rates ε_0 for ProxSGD is examined in particular. As expected, the hyperparameters affect the achieved training loss and test accuracy, and several lead to a worse training loss and/or test accuracy than ADAM and SGD with momentum, as shown in Figure 5.8(a)-(b). However, looking at the cumulative distribution functions of the weights in Figure 5.8(c), one can see that most of them (except when they are too small: $\varepsilon_0 = 0.01$ and 0.001) generate a much sparser neural network than both ADAM and SGD with momentum. These observations are also consistent with the theoretical framework in Section 5.2: interpreting ADAM and SGD with momentum as special cases of ProxSGD implies that they have the same convergence rate, and the sparsity is due to the explicit use of the nonsmooth ℓ_1-norm regularization.

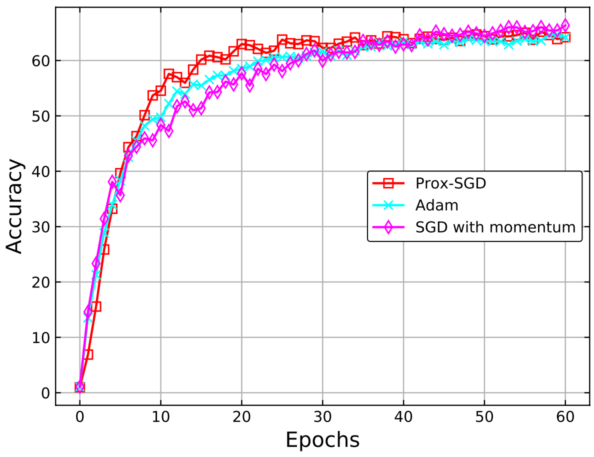
Regarding the training time for these experiments, the use of the soft-thresholding proximal operator in ProxSGD increases training time: the average time per epoch for ProxSGD is 3.5 min, SGD with momentum 2.8 min and ADAM 2.9 min. In view of the higher level of sparsity achieved by ProxSGD, this increase in computation time is reasonable and affordable.

In summary, this chapter proposed ProxSGD, a proximal-type stochastic gradient descent algorithm with momentum for constrained optimization problems where the smooth loss function is augmented by a nonsmooth and convex regularization. Experiments on the stochastic training of sparse neural networks show that regularization and constraints can effectively promote structures in the learned network. More generally, incorporating regularization and constraints allows to use a more accurate and interpretable model for the problem at hand and the proposed convergent ProxSGD algorithm ensures efficient training. Numerical tests show that ProxSGD outperforms state-of-the-art algorithms, in terms of convergence speed, achieved training loss and/or the desired structure in the learned neural networks.

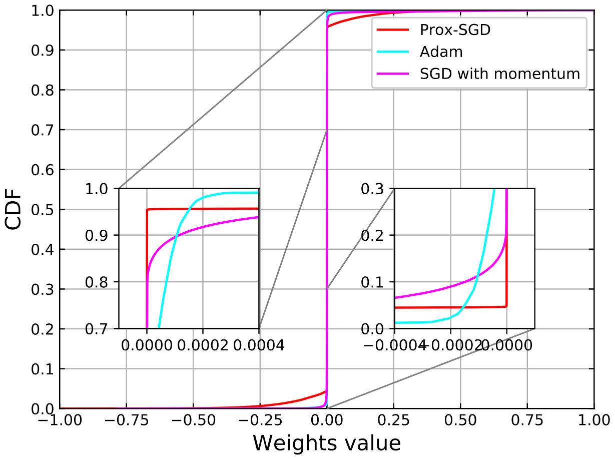
There are many directions to extend this framework. One possible direction is the development of a distributed stochastic proximal gradient method, as outlined by [173]. SCA methods in general lend themselves to the distributed setting by making the surrogate objective function block-separable, in which case different blocks of variables can be updated in parallel [198]. One such example, where the surrogate objective function can be efficiently computed in parallel, is for block-separable regularization functions, which include the ℓ_1-, ℓ_2- and $\ell_{2,1}$-regularization [195]. Another direction is to use another distance metric in the proximal operator definition. ProxSGD uses the Euclidean norm, but there exist many other norms to consider, like the Bregmann divergence (see [31, 35]) or the entropic divergence [220]. Lastly, one can extend ProxSGD to cover more general cases, like non-convex regularization terms, which has been attempted by [245]. One issue with these approaches is that in many cases it is not possible to use a closed form solution, which can severely increase the computation time of one iteration. Other approaches have tried to leverage the convex nature of the cost function, by using a linear approximation of the neural network around the current set of parameters while preserving the convex cost function [195]. By incorporating sparsity inducing regularization terms through a proximal component in order to ensure strong convexity of the surrogate function, the resulting SCA algorithm is able to incorporate much more information of the underlying structure of the optimization problem and in theory this should benefit its practical convergence. Combining this idea with ProxSGD could be an interesting direction for future research.



(a) *Training Loss*

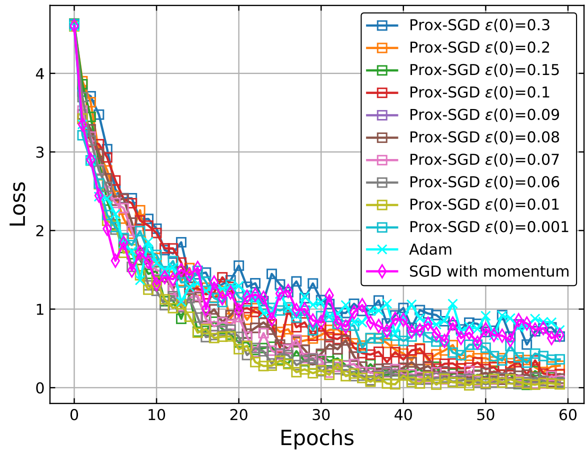


(b) *Test Accuracy*

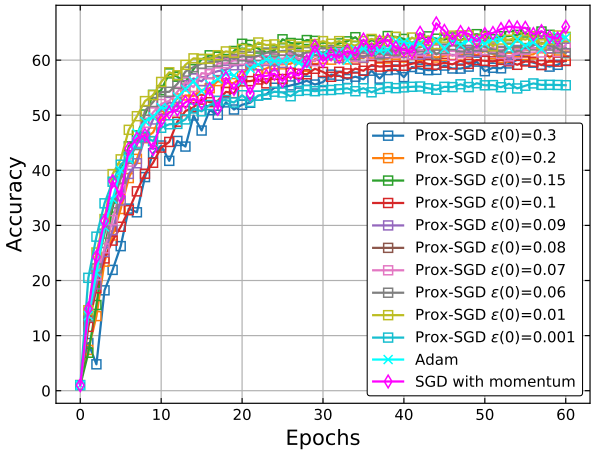


(c) *CDF of Weights*

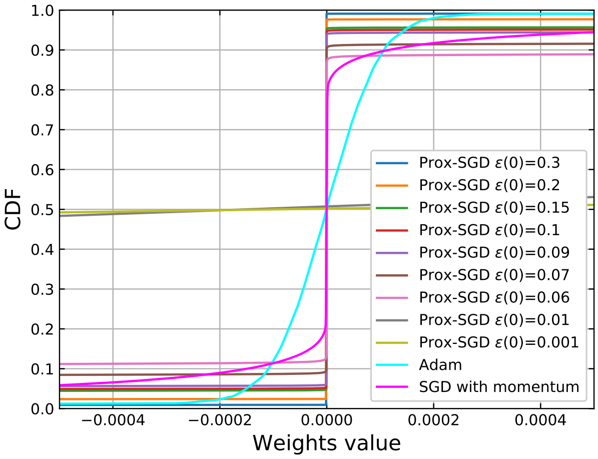
Figure 5.7.: Performance comparison for DenseNet-201 on CIFAR-100.



(a) *Training Loss*



(b) *Test Accuracy*



(c) *CDF of Weights*

Figure 5.8.: Hyperparameters and sparsity for DenseNet-201 on CIFAR-100.

Chapter 6

ProxSGD with $\ell_{2,1}$-Regularization

This chapter will explore the use of ProxSGD with $\ell_{2,1}$-regularization and show how this regularization can be used in order to do structured pruning. First, the main ideas behind *neural architecture search* (NAS) will be explained and some of the limitations of this field in deep learning will be highlighted. Second, using group sparsity via ProxSGD with $\ell_{2,1}$-regularization, a unified approach for network pruning and one-shot neural architecture search via group sparsity will be proposed. Next, experiments show that this flexible optimizer is able to achieve new state-of-the-art results for filter pruning, which refers to pruning entire filters in convolution layers. In the succeeding section, this approach is extended to operation pruning, where each layer (operation) or groups of operations in a neural network can be pruned individually. Thus, group sparsity directly yields a gradient-based NAS method.

When compared to existing gradient-based algorithms, there are three advantages to the group sparsity approach. Firstly, instead of relying on a costly bilevel optimization problem, the NAS problem is formulated as a single-level optimization problem. This can be solved optimally and efficiently using ProxSGD with its convergence guarantees. Secondly, using operation-level sparsity, the network architecture can be discretized by pruning less important operations without *any* performance degradation. Thirdly, this approach to NAS finds architectures that are well-performing on a variety of search spaces and datasets. Also, this approach is robust in generating architectures that perform well on a variety on search spaces and datasets, where the architectures of other methods have collapsed to include only skip-connections for example. In the last section the performance and robustness of group sparsity on NAS, dubbed GSparsity, is shown on a variety of datasets and benchmarks.

6.1. Overview Neural Architecture Search

For new problems at hand, designing performant network architectures requires substantial efforts by human experts. As an algorithmic solution, NAS has been developed to automate the architecture search process. The main idea behind neural architecture search is to automate the process of designing neural network architectures for a given dataset. An illustrative overview of neural architecture search is shown in Figure 6.1.

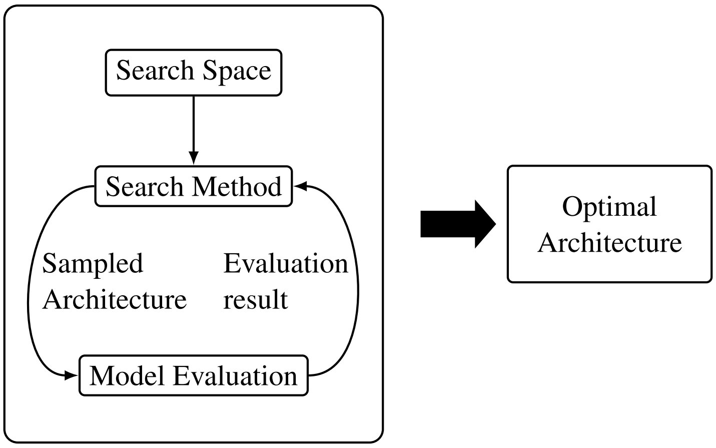


Figure 6.1.: *Illustration of the process and the main components used in neural architecture search. The main components consist of the search space, the search method as well as the model evaluation. After the search method has found a performant architecture this candidate is returned.*

As can be seen in this Figure, the main components to NAS are the search space, the search algorithm and the evaluation strategy.

Search Space The search space refers to the types of operations that are allowed to be used in the architecture, as well as any restrictions on the overall architecture design. The overall architecture design usually requires some level of expert knowledge, such as the number of output channels at each layer of the network or the use of padding and strides.

While there exist architecture designs that define the network topology by hand and let the algorithm choose the best hyperparameters of individual layers [254] (e.g. for convolution layers: filter size, stride, number of output channels), most of the newer approaches employ a cell-based structure. These cell-based architectures define certain types of cells and build the architecture by repeating these cells in a certain order. This approach simplifies the architecture search to finding a good structure inside each cell-type and not over the whole architecture. An example of this cell based approach is shown in Figure 6.2, which depicts one type of cell with four different operations connecting three different nodes.

Search Algorithm A simple, yet surprisingly effective approach is to use random search [134]. Randomly sampling different architectures given the search space and choosing the best performing one can result in a competitive architecture. Some of the earliest approaches of NAS were based on reinforcement learning [254] and evolutionary algorithms [8, 215], which treat NAS as a black-box optimization problem.

In reinforcement learning, the agent typically tries to optimize the accuracy of a given architecture and this is used as its reward function. The action space are the different operations present in the search space, so the agent has to decide which operations to keep in order to optimize the accuracy of the evaluated architecture.

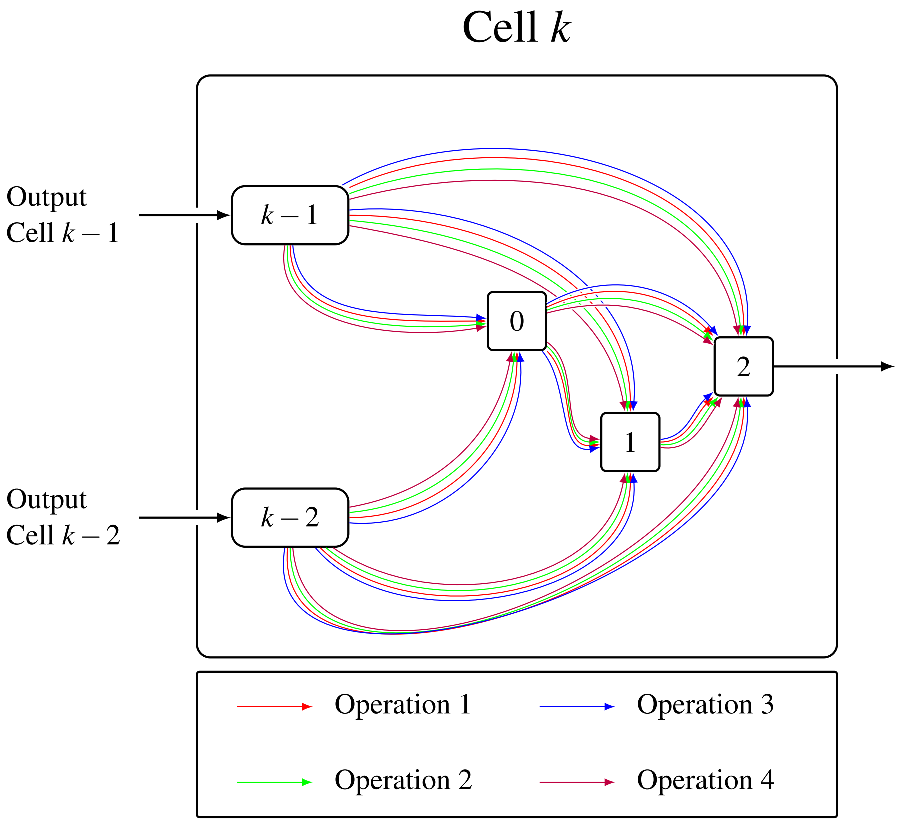


Figure 6.2.: *Illustration of a cell as used in differentiable architecture search. The cell has two inputs, where the output of the cell $k-2$ and cell $k-1$ is fed into the current cell. Apart from the two input nodes this cell has three nodes which are connected by all the operations in the search space. The search space in this figure consists of four different operations.*

Evolutionary algorithms use a genetic algorithm that randomly mutates connections between operations or even the type of operation itself. Also, two different parent architectures can spawn an offspring architecture that retains different operations and connections of the parents. One of the most popular evolutionary methods is the AmoebaNet [183]. Compared to recent methods, the reinforcement learning and evolutionary approaches are extremely slow, using up to 3150 GPU hours in order to find a competitive architecture [183, 255].

A more recent approach is to use gradient based methods to find well-performing architectures [143]. In these methods, each operation of the search space has an associated architecture parameter. Using gradient based methods, these architecture parameters are updated and operations are kept or removed based on the value of those parameters.

Evaluation Strategy One of the main challenges in NAS is the architecture evaluation. In theory, each architecture that is found has to be trained from scratch until convergence, in order to be able to evaluate its performance. This represents a major computational burden and is one of the reasons that many reinforcement learning and evolutionary methods are so resource intensive. There are ways to lower the training time of the architecture, like reducing the number of samples in the dataset, or not training until convergence, but even with these approaches the compute time is still high (see [62] for a more in depth discussion).

The one-shot approach tries to solve this problem by sharing the weights of the individual operations between the architectures. Commonly, the overall architecture design is used with all of the possible operations present. In this thesis this network will be referred to as the supernet. The supernet of a single cell-type is illustrated in Figure 6.2, where each node is connected by all four operations of the search space. Due to the benefits of the one-shot differential NAS approach, the remaining chapter will only focus on this setting.

Differentiable Architecture Search The one-shot approach of NAS [21, 32, 177] based on weight sharing enables search on a single large supernet with the goal of finding a promising small subnet. In their paper *differentiable architecture search* (DARTS) [143], the authors formulate the one-shot NAS problem as a differentiable optimization problem that can be solved by standard stochastic algorithms. They let the architecture parameters vary between zero and one and only choose the operations with the biggest parameters after training, in order to obtain their final architecture. During training, the algorithm first updates the model parameters and in the next step updates the architecture parameters using second-order information. Thus, DARTS is able to train much faster than reinforcement or evolutionary approaches. However, there are several limitations to DARTS.

Firstly, modelling the importance of each operation in the architecture as a learnable scalar, which scales the contribution of the specific operation in the current objective function value can introduce several problems. One issue with this approach is that the discretization of these continuous architecture parameters after training can lead to a performance degradation [228, 248, 249]. This indicates the mismatch between the true objective function and the one which is optimized by the NAS algorithm. Secondly, DARTS frames the NAS problem as a bi-level optimization problem [46]. In this approach, the upper-level architecture parameters depend on the optimal response function value, which is defined by the lower-level solutions. Since this is complex and computationally intractable to solve exactly, several approximations have to be defined, without taking any convergence guarantees into account. Lastly, as shown in [248], the behaviour of DARTS is not robust across search spaces, often producing degenerate results with architectures composed by parameterless operations only.

Many follow-up works have been proposed to overcome these limitations of DARTS [37, 38, 228, 236, 248]. Closest to the method presented in this thesis is the HAPG algorithm recently proposed in [234], where an additional group sparsity regularization is applied to the architecture parameters in order to reduce the discretization gap after the search procedure. The core difference between the proposed algorithm and HAPG is that in unlike HAPG, the group sparsity regularization is applied directly to the one-shot model weights. Therefore, this makes the architecture parameters unnecessary and the NAS problem reduces to a single-level optimization problem. Besides, the proximal algorithm used in [234] does not have a guaranteed convergence. Another method that is close to the method proposed in this thesis is [251], which also addresses the NAS problem from the pruning perspective. But it is essentially pruning operations from a much larger supernet, which is not always possible to load into GPUs with limited memory.

There are also other works that are similar in some regard to some aspect of the proposed method in this thesis, though all have some significant drawbacks to their method. The authors of [233] propose an algorithm to learn the connectivity pattern in neural networks by imposing a constraint on the maximal number of edges that the target network has to contain. In contrast to this work, the GSparsity algorithm determines automatically what the connectivity pattern is, based on the group sparsity regularization on the parameters. [228] revisit the discretization step in many famous one-shot NAS algorithms and propose a new architecture selection method by evaluating the one-shot model after pruning some operations following a predefined post-hoc heuristic. This is done implicitly with the GSparsity method in the search routine without such heuristics.

6.2. GSparsity: ProxSGD with $\ell_{2,1}$-Regularization

In this section, GSparsity, the unified group-sparsity approach for both network pruning and NAS is presented.

Let the vector $\boldsymbol{w}$ represent all of the trainable parameters of the neural network. This vector is decomposed into different subvectors $\boldsymbol{w} = (\boldsymbol{w}_l)_{l=1}^{L}$, where each $\boldsymbol{w}_l$ represents a group of weights (e.g. all weights in the same filter of a convolutional layer) and the total number of (non-overlapping) groups is denoted by L. As explained in Section 5.1, in order to minimize the loss function f augmented by the group sparsity regularization on $\boldsymbol{w}$, the training objective of the neural network is formulated as the following optimization problem:

$$\underset{\boldsymbol{w}}{\text{minimize}} \quad \frac{1}{|\mathbb{X}|} \sum_{\boldsymbol{x} \in \mathbb{X}} f(\boldsymbol{w}, \boldsymbol{x}) + \sum_{l=1}^{L} \mu_l \left\| \boldsymbol{w}_l \right\|_2 , \tag{6.1}$$

where $\boldsymbol{x}$ is a training sample from the training dataset $\mathbb{X}$ (with $|\mathbb{X}|$ denoting the number of training samples). Besides, the ℓ_2-norm defined as $\|\boldsymbol{w}\|_2 \triangleq \sqrt{\boldsymbol{w}^T \boldsymbol{w}}$ is a nonsmooth convex function (see Figure 2.7).

The $\ell_{2,1}$-norm can encourage a group-sparse neural network. After training, most groups will be forced to zero and can thus be safely removed from the neural network without incurring any performance loss. Due to the fact that group sparsity is a more structured sparsity, this can be much better exploited by hardware. This property is a major benefit over the widely used ℓ_1-norm, which only promotes unstructured sparsity that can not be efficiently exploited by most modern hardware in order to gain speedup during training and inference.

The regularization parameter μ in (6.1) is a predefined parameter that indirectly controls the achieved sparsity level in the neural network: a larger μ implies that more groups will be zero. However, due to this formulation, it is not possible to assign a certain sparsity level a priori (for example, exactly two kernels in each filter are zero). Furthermore, even if the same μ is used in different runs, the randomness in stochastic algorithms and the nonconvexity in the optimization problem renders it possible that different optimal solutions of (6.1) exhibit different sparsity levels. There are possibilities to relate μ_l to other constraints, such as hardware constraints (e.g. FLOPs and memory access cost) [251], so the optimal solution of (6.1) will also be bounded by these constraints.

The formulation of the optimization problem in (6.1) directly regularizes the weights of the network. This differs from other recent works like [144], [136] and [251], where the output of each group is scaled by a single scaling factor and the regularization is applied to these scaling factors. There are different ways to define the regularization parameter μ_l, depending on the situation at hand. The most straightforward choice is to set each μ_l to the same value for all groups: $\mu_l = \mu$, $\forall l$. Often when using the $\ell_{2,1}$-norm, in order to reduce the number of parameters, larger groups are penalized more. This is achieved by using $\mu_l = \mu \cdot \sqrt{|\boldsymbol{w}_l|}$, $\forall l$. The idea is that the larger the group the more unknown parameters are estimated and thus, for larger groups, one should penalize these groups more. Though, this fact does not imply that sometimes other penalization schemes could not work better and whenever the size of the groups does not matter it can be beneficial to normalize the regularization gain μ by the size of the group:

$$\mu_l = \frac{\mu}{\sqrt{|\boldsymbol{w}_l|}}, \forall l. \tag{6.2}$$

There is no theoretical justification that favors one over another (see [244]), and for the experiments done in this chapter it is empirically found that $\mu_l = \mu, \forall l$ works well when the sizes of the groups are not very different, as in filter pruning and operation pruning. However, this is not the case for NAS, where the normalization specified in (6.2) yields the best result. The algorithm used for training the neural networks using ProxSGD with $\ell_{2,1}$-norm regularization is shown in Algorithm 9.

If $\mathbb{W}_l = \mathbb{R}^{n_l}$ is used, the approximation subproblem in Step 3) of Algorithm 9 has a closed form solution:

$$\hat{\boldsymbol{w}}_l^{(k)} = \left(1 - \frac{\mu_l}{\tau_l^{(k)} \left\| \boldsymbol{w}_l^{(k)} - \frac{1}{\tau_l^{(k)}} \boldsymbol{v}_l^{(k)} \right\|_2}\right)^+ \left(\boldsymbol{w}_l^{(k)} - \frac{1}{\tau_l^{(k)}} \boldsymbol{v}_l^{(k)}\right) \tag{6.5}$$

where $(.)^+$ represents the $\max\{0,.\}$ function. One can see from (6.5), that all the parameters inside a group get forced towards zero whenever

$$\tau_l^{(k)} \left\| \boldsymbol{w}_l^{(k)} - \frac{1}{\tau_l^{(k)}} \boldsymbol{v}_l^{(k)} \right\|_2 \leq \mu_l. \tag{6.6}$$

An example of the difference between using ProxSGD, which is able to efficiently converge to a group sparse solution, and SGD with momentum is shown in Appendix D.

Algorithm 9 GSparsity

```
Initialize: $\boldsymbol{w}^{(0)} \in \mathbb{W}, \boldsymbol{v}^{(-1)} = \mathbf{0}$, Total number of iterations $T$, Momentum parameter $\{\rho_k\}_{k=0}^{T}$, learning rate $\{\varepsilon_k\}_{k=0}^{T}$, weight vector $\boldsymbol{w}$ split into $L$ different groups $\boldsymbol{w} = (\boldsymbol{w}_1, \ldots, \boldsymbol{w}_L)$
for k=0 to T do
    1) Compute the instantaneous gradient $\boldsymbol{g}^{(k)}$ based on the minibatch $\mathbb{M}_k$:
        $\boldsymbol{g}^{(k)} = \frac{1}{m_k} \sum_{i \in \mathbb{M}_k} \nabla_{\boldsymbol{w}} f_i(\boldsymbol{w}^{(k)}, \boldsymbol{x}^{(i)})$    (6.3)
    2) Update the momentum: $\boldsymbol{v}^{(k)} = (1-\rho_k)\boldsymbol{v}^{(k-1)} + \rho_k \boldsymbol{g}^{(k)}$
    3) Compute $\hat{\boldsymbol{w}}^{(k)}$ by solving the approximation subproblem:
        $\hat{\boldsymbol{w}}_l^{(k)} = \underset{\boldsymbol{w}_l \in \mathbb{W}_l}{\arg\min} \{ (\boldsymbol{w}_l - \boldsymbol{w}_l^{(k)})^T \boldsymbol{v}_l^{(k)} + \frac{\tau_l^{(k)}}{2} \left\| \boldsymbol{w}_l - \boldsymbol{w}_l^{(k)} \right\|_2^2 + \mu_l \left\| \boldsymbol{w}_l \right\|_2 \}$    (6.4)
    4) Update the weight: $\boldsymbol{w}^{(k+1)} = \boldsymbol{w}^{(k)} + \varepsilon_k (\hat{\boldsymbol{w}}^{(k)} - \boldsymbol{w}^{(k)})$
end for
```

6.3. Filter Pruning

This section illustrates the performance of filter pruning for ResNet-50 [83] on ImageNet 2012 using GSparsity. In order to perform filter pruning, all the parameters of the same filter are placed into one group. This allows to prune individual filters away inside different convolutions.

Experiment setup. Firstly, in order to determine which filters to prune away, the ResNet-50 model is trained-*with* the $\ell_{2,1}$-norm regularization ($\mu_l = \mu/\sqrt{|\boldsymbol{w}_l^{(k)}|}$)-by ProxSGD for 90 epochs, with the following hyperparameters: initial learning rate $\varepsilon_0 = 0.001$ (which is linearly decayed by 10 every 30 epochs) and momentum $\rho = 0.9$. Furthermore, $\tau_l^{(k)}$ in (6.5) is defined as $\tau_l^{(k)} = \text{mean}\left(\sqrt{\boldsymbol{r}^{(k)}/(1-\beta^k)}\right) + \delta$ ($\beta = 0.999$, $\delta = 10^{-8}$), and $\boldsymbol{r}^{(k)}$ is the aggregate squared gradient updated iteratively as $\boldsymbol{r}^{(k)} = \beta \boldsymbol{r}^{(k-1)} + (1-\beta)(\boldsymbol{g}^{(k)})^2$ [110].

The ResNet-50 model consists of 4 bottleneck layers, and bottleneck layer 1/2/3/4 consists of 3/4/6/3 blocks. Each block consists of three convolutional layers. In the experiments, only the filters of the first two layers in each block are pruned, for two reasons: 1) the experiments show that even if all filters of the three convolutional layers are set to be prunable, only very few filters from the last layer are actually pruned; 2) the output of the last convolutional layer would be required to have the same dimension as the residual layer in order to add them both together. Future experiments could place the weights of the convolutional filters that are connected through a residual layer in the same group, allowing them to be pruned simultaneously. This approach would remove this issue and could lead to better performance.

Note that the objective in filter pruning is to reduce the *multiple-accumulate operations* (MACs) with as little performance degradation as possible. MACs represent the

algorithm	top-1 acc	MACs	code available?
baseline (reported)	76.13	100% (4.12G)	Y
SSS [96] (reported)	74.18	68.55%	Y (MXNet)
ThiNet-70 [148] (reported)	72.04	63.21%	Y (Caffe)
GSparsity (ours, $\mu = 0.02$)	75.21	63.11% (2.60G)	Y
GSparsity (ours, $\mu = 0.05$)	74.33	50.00% (2.06G)	Y
FPGM [84] (reported/reproduced)	74.83/69.69	46.50%/49.03% (2.02G)	Y
Hinge [136](reported)	74.70	46.55%	N
RRBP [253] (reported)	73.00	45.45%	N
ResRep [54] (reproduced)	0.10	45.15% (1.86G)	Y
GAL [140] (reported)	72.80	44.98%	N
GSparsity (ours, $\mu = 0.07$)	74.00	44.42% (1.83G)	Y
GSparsity (ours, $\mu = 0.1$)	73.34	42.23% (1.74G)	Y

Table 6.1.: *Filter pruning of ResNet-50 on ImageNet 2012.*

basic operation of multiplying two numbers and adding those with an accumulator and are a useful unit in order to gauge the computational complexity of a neural network. The second and third bottleneck layers constitute 60.83% of the total MACs, although they have only 35.55% of the total parameters. Therefore, in order to push more filters in the second/third layers to be 0, the regularization gain is set to be $\mu_l = \mu/\sqrt{|\boldsymbol{w}_l|}$, and it is further doubled if it is in the second/third bottleneck layers.

After training with ProxSGD is completed, the filters from the model with zero ℓ_2-norm are pruned (rather than just masked). Because the convergence of iterative algorithms to an optimal solution $\boldsymbol{w}^\star$ is in the sense that $\|\boldsymbol{w}^{(k)} - \boldsymbol{w}^\star\| \leq c$ for an arbitrarily small but strictly positive c, in these experiments the filters whose ℓ_2-norm is smaller than $c = 10^{-6}$ are pruned. In order to achieve higher performance, the pruned network is retrained -*without* the $\ell_{2,1}$-norm regularization- for 90 epochs by SGD with momentum with an initial learning rate 0.1 (which is linearly decayed by 10 every 30 epochs), momentum 0.9 and batch size 256.

Results. The performance of the baseline (unpruned) ResNet-50 model and various filter pruning methods is summarized in Table 6.1. The unpruned ResNet50 model has a total of 25.56M parameters and 4.12GMACs, and this network achieves an accuracy of 76.13%. For many pruning methods, the code for ResNet-50 and ImageNet 2012 is not available in the repository. It is thus impossible to verify and reproduce their results. Besides, it is not clear how other methods have measured the MACs, making a fair comparison difficult. As an example, FPGM in Table 6.1 reported 46.50% of the original MACs, but it is 49.03% according to the MACs calculator used in this thesis [211].

Currently, the FPGM method is regarded as state-of-the-art because it is a recent work achieving good results and its code is available. Nevertheless, there exists a gap between the reported top-1 accuracy and the one that experiments could reproduce. The gap is due to the fact that the reported top-1 accuracy is measured after masking the zero filters in the original model, but the batch-norm following these filters are not masked. On the one hand, it is possible to partially recover the performance if the compressed network is retrained, and the retrained accuracy is 74.27%. On the other hand, since the third convolutional layers in each block are also pruned, the indices

μ	top-1 acc after search	top-1 acc after retrain	MACs	Params
0.01	73.99	76.01	3.22G	22.70M
0.02	73.81	75.21	2.60G	17.08M
0.03	73.49	74.93	2.36G	15.59M
0.04	72.23	74.35	2.17G	14.67M
0.05	73.00	74.33	2.06G	14.03M
0.06	72.83	74.10	1.97G	13.56M
0.07	73.01	74.00	1.92G	13.22M
0.08	72.73	73.62	1.83G	12.86M
0.09	72.47	73.41	1.77G	12.51M
0.10	72.45	73.34	1.74G	12.36M

Table 6.2.: *GSparsity and filter pruning: Ablation study.*

of the nonzero must be saved so that the output of the third layer can be added to the residual layer. This additional complexity slows down the training (70 mins/epoch vs. 48 mins/epoch in GSparsity) in retraining.

Table 6.1 shows that the proposed GSparsity method achieves the same performance as FPGM, but its retraining cost is much lower than that of FPGM. When μ is increased, a larger reduction in MACs is achieved and the performance is better than other methods in literature. A closer look at the initial and final structure of the ResNet-50 network is summarized in Appendix B.

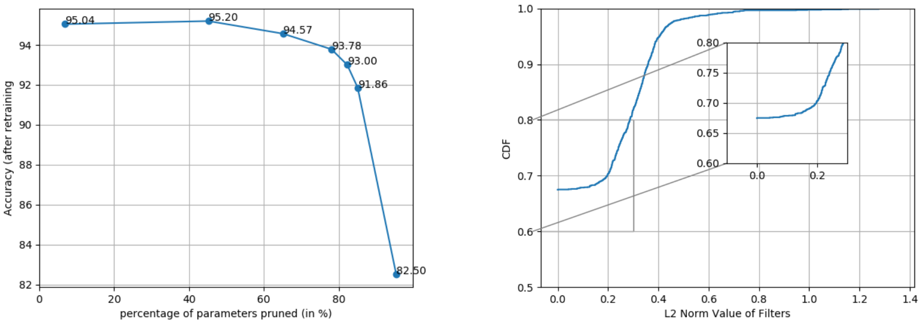
Table 6.2 shows an ablation study that tests the performance of GSparsity with respect to μ. It is evident that the MACs as well as the top-1 accuracy are monotonic functions of μ, thus an appropriate value of μ achieving a target sparsity level can be found efficiently by the bisection search.

In order to show the effects of pruning, the trade-off between accuracy and sparsity of the proposed GSparsity is illustrated in Figure 6.3a, where the x-axis specifies the percentage of parameters pruned from the original network, while the y-axis specifies the achieved accuracy. When the sparsity is low, it is possible to prune filters without any performance loss. As the target sparsity level increases, the achieved accuracy decreases. In practice, one could either prune as many filters as possible, as long as a target accuracy is achieved, or strive for the best accuracy, as long as the model is smaller than a target size.

To demonstrate that the ProxSGD algorithm can converge to a group-sparse solution where most groups are exactly zero, the cumulative distribution function (CDF) of the filters' ℓ_2 norm is examined. An example CDF from a randomly picked experimental run is shown in Figure 6.3b, where one can see that about 68% of filters are exactly zero. Nevertheless, pruning filters according to a threshold of 0.2 for example may prune not only zero but also some nonzero filters and pruning nonzero filters may be harmful for the performance of the pruned network.

6.4. Operation Pruning

In this section, the proposed GSparsity algorithm is used for operation pruning, where each group consists of all parameters of the same operation.



(a) *The percentage of parameters pruned vs. the accuracy*

(b) *The CDF of the filters' norm after* `Search` *is completed*

Figure 6.3.: *Plot (a) depicts the accuracy of the Resnet-50 network after retraining for different amounts of parameters that have been pruned using filter pruning. Plot (b) depicts the CDF of the ℓ_2-norm of the filters after the Resnet-50 network has been searched.*

Experiment setup. As a base network to be pruned, one of the networks found in DARTS [143] is chosen: DARTS-V2, which has 3.3M parameters. To perform operation pruning, a group should consist of all trainable parameters of the same operation. Here, an operation can also consist of many different basic operations that act as one overall layer. For example, the dilated convolution operation consists of four suboperations: `ReLU`, `Conv2d(C_in,C_in)`, `Conv2d(C_in, C_out)`, `BatchNorm2d(C_out)`. The group should consist of the trainable parameters of all suboperations.

First, ProxSGD is used to train DARTS-V2 -*with* the $\ell_{2,1}$-norm regularization- on CIFAR-10, where the regularization gain μ is identical for all groups. Various values of $\mu \in \{0.0001, 0.0002, 0.0005, 0.002, 0.004\}$ are used to get different sparsity levels. After training with ProxSGD is finished, operations whose ℓ_2-norm is smaller than $c = 10^{-6}$ are pruned. The resulting pruned network will be retrained -*without* the $\ell_{2,1}$-norm regularization- by SGD with momentum, and the retrained accuracy is reported in the following paragraph.

Results. One can see from Table 6.3 and Figure 6.4 that when operations are pruned away such that 38.40% of weights are pruned, the network's retrained accuracy (97.45%) is almost identical to the unpruned baseline (97.50%). When 60.46% of weights are pruned, the retrained accuracy is 97.09%, i.e., 0.41% worse than the unpruned baseline. This tradeoff between sparsity and accuracy is observed in many papers, and the final sparsity depends naturally on the target accuracy.

Figure 6.4 depicts the trade-off between accuracy and sparsity of GSparsity for operation pruning. The x-axis specifies the percentage of parameters that are pruned and the y-axis specifies the accuracy of the *retrained* model after pruning. Table 6.3 also shows the accuracy before and after the operations (with an ℓ_2-norm smaller than a given threshold, namely, 1e-6, 1e-3 or 0.5) are pruned. Observe that the proposed GSparsity approach does not incur any discretization error, even when the pruning threshold is only modestly small (such as 1e-3). Comparing the accuracies before

μ	accuracy before pruning	accuracy after pruning			accuracy after retraining		inference time
		1e-6	1e-3	0.5	accuracy	parameters	
0	97.50	-	-	-	-	100%	7.13s
0.0001	96.50	96.50	96.50	92.08	97.45	78.25%	6.73s
0.0002	96.46	96.46	96.46	75.50	97.44	61.60%	6.40s
0.0005	96.36	96.36	96.36	13.34	97.32	52.09%	6.24s
0.002	96.47	96.47	96.47	10	97.09	39.54%	4.96s
0.004	96.48	96.48	96.48	10	96.84	32.80%	4.43s

Table 6.3.: *Operation pruning: The accuracy before/after operation pruning (but before retraining, with pruning thresholds 1e-6, 1e-3 and 0.5) and after retraining (with pruning threshold 1e-6).*

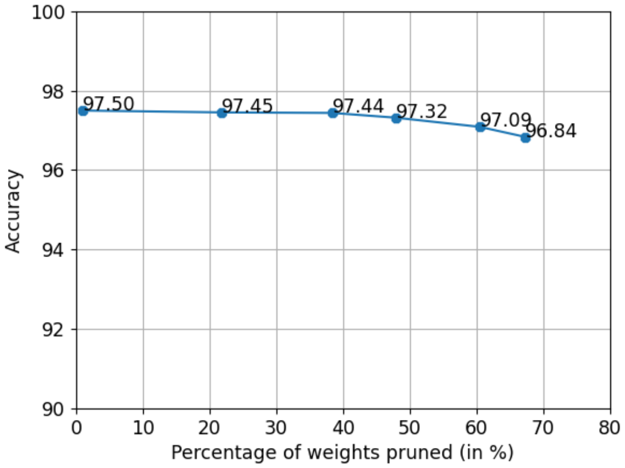
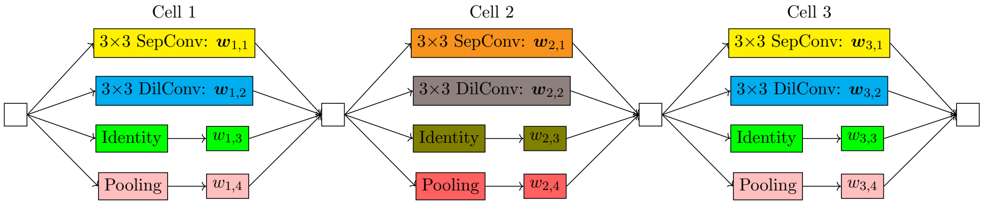


Figure 6.4.: *The percentage of parameters pruned vs. the accuracy for operation pruning.*

and after retraining, observe that retraining can further enhance the performance of the pruned network.

6.5. Neural Architecture Search with GSparsity

As shown in the past sections, the notion of groups provides sufficient flexibility to achieve different levels of pruning. Depending on the specific task at hand, a group could be a kernel or filter inside a convolution layer. Another possible group could also be all the parameters that belong to an operation: an operation with trainable parameters can be safely removed if all of its parameters are zero. One issue arises for operations such as pooling and identity that do not have trainable parameters. There, an additional scaling factor is added to the output of the parameterless operation. For example, the 3×3 SepConv and Identity in Cell 2 in Figure 6.5(a) can be removed if $\boldsymbol{w}_{2,1} = \boldsymbol{0}$ and $w_{2,3} = 0$, respectively. As mentioned in the previous section, in case an operation is a concatenation of several suboperations, $\boldsymbol{w}_l$ should be comprised of the trainable parameters of all suboperations. Alternatively, the trainable parameters of suboperations can be put into separate groups, and the operation can be removed if any group is zero.


Cell 1
3×3 SepConv: $\boldsymbol{w}_{1,1}$
3×3 DilConv: $\boldsymbol{w}_{1,2}$
Identity
$w_{1,3}$
Pooling
$w_{1,4}$
Cell 2
3×3 SepConv: $\boldsymbol{w}_{2,1}$
3×3 DilConv: $\boldsymbol{w}_{2,2}$
Identity
$w_{2,3}$
Pooling
$w_{2,4}$
Cell 3
3×3 SepConv: $\boldsymbol{w}_{3,1}$
3×3 DilConv: $\boldsymbol{w}_{3,2}$
Identity
$w_{3,3}$
Pooling
$w_{3,4}$

(a) *Supernet in* `Search`*: Cell 1 and Cell 3 are of the same type and Cell 2 is of a different type (Operations in the same color shall be preserved or removed simultaneously.)*

$$f(\boldsymbol{w}) + \mu\left(\left\|\begin{bmatrix}\boldsymbol{w}_{\mathbf{1,1}}\\ \boldsymbol{w}_{\mathbf{3,1}}\end{bmatrix}\right\|_2 + \left\|\begin{bmatrix}\boldsymbol{w}_{\mathbf{1,2}}\\ \boldsymbol{w}_{\mathbf{3,2}}\end{bmatrix}\right\|_2 + \left\|\begin{bmatrix}w_{1,3}\\ w_{3,3}\end{bmatrix}\right\|_2 + \left\|\begin{bmatrix}w_{1,4}\\ w_{3,4}\end{bmatrix}\right\|_2 + \|\boldsymbol{w}_{\mathbf{2,1}}\|_2 + \|\boldsymbol{w}_{\mathbf{2,2}}\|_2 + \|w_{2,3}\|_2 + \|w_{2,4}\|_2\right)$$

(b) *Definition of groups in* `Search` *(The regularization gains $\{\mu_l\}$ are assumed to be identical for all groups.)*

Figure 6.5.: *Illustrative example of applying the group sparsity approach to NAS.*

This section illustrates how the concept of group sparsity can be used to perform network architecture search. As mentioned before, one-shot approaches for NAS typically consist of two steps: `Search` and `Evaluate`. The first is the `Search` step, and it is usually performed on a relatively shallow proxy supernet consisting of consecutive cells connected in a predefined manner. This has to be done because all candidate operations in the search space are assumed to be active in the supernet and thus its size grows fast with the number of cells. The cells can have the same or a different structure. To make sure that cells of the same type will have the same structure, operations of the same type (encoded in the same color in the example network in Figure 6.5(a)) should be removed or preserved simultaneously. To this end, these operations are placed into the same group, as visualized in Figure 6.5(b). In the example network shown in Figure 6.5(a), Cell 1 and Cell 3 are of the same type and Cell 2 is of a different type. As shown in Figure 6.5 for example, since Cell 1 and Cell 3 are assumed to be the same Cell type, the parameters of the 3×3 SepConv of Cell 1 and Cell 3 all belong to the same group. If the ℓ_2-norm of this group is forced to zero by GSparsity, this operation will be pruned away in both cells simultaneously. Thus, this shows that different cells can be linked together in order to generate different types of cells. The aim of the `Search` step is to find the optimal structure of these cells, i.e. the operations and the connectivity between different nodes, such that it is optimal with respect to some objective (e.g. validation loss). If a group is zero after `Search` is completed, it implies that the same operation in all cells of the same type is zero and can thus be removed from these cells.

After this pruning step, the found cell(s) can then be stacked repetitively to form a much deeper network. This is now possible, since most of the operations in the initial supernet will be pruned away, reducing the overall size of the cells. In the `Evaluate` step, the deeper networks will be retrained, and the one with the best performance will be used for future inference.

Except for operations with no trainable parameters (such as pooling and identity), the proposed formulation (6.1) does *not* need the architecture parameter (the so-called $\boldsymbol{\alpha}$ parameters in DARTS and many follow-up papers). Its implications are twofold.

Firstly, (6.1) is a single-level optimization problem and it is much easier to solve than the otherwise costly bilevel optimization problem (the architecture parameter in the upper level and the network weights in the lower level). Secondly, it is no longer necessary to split the training dataset into two parts, one used to update the architecture parameters and the other for the network weights. The group sparsity regularization in (6.1) will also alleviate overfitting from which bilevel optimization with architecture parameters may suffer.

The method proposed in [251] addresses the NAS problem from a pruning perspective, but it is essentially operation pruning: operations in different cells are pruned independently and similar cells are not linked by grouping parameters of the same operation across these cells. This proposed approach should be applied to the deep network (as deep as in `Evaluate`) but with all candidate operations active (as in `Search`). The resulting supernet is more often than not too large (much larger than the commonly used proxy supernet in `Search`) for GPUs with limited memory, and hence this approach is not always practical.

In the following subsections, the proposed GSparsity algorithm is used for NAS and compared with state-of-the-art differentiable algorithms. All experiments follow the NAS best practice checklist [142].

Experimental protocol. To study the robustness of GSparsity and various baselines, each method is run 3 times; next, each of the 3 resulting architectures are evaluated 3 times and the means and standard deviations over the resulting 9 results are reported. Note, that this is different from the popular strategy in existing work (such as [143]), where the neural architecture search is repeated several (often, 4) times and only the best architecture (in terms of the validation accuracy in `Search`) is evaluated. The experiments in this section deviate from this strategy because, on the one hand, it increases the search cost (which is proportional to the number of searches), and, on the other hand, it does not reflect the expected performance and stability of the search algorithm. As a consequence, the performance could be worse than reported in the original papers. All methods have been re-run (using the authors' original implementation) on the same hardware and using the same evaluation protocol.

6.5.1. CIFAR-10 and CIFAR-100.

Neural architecture search on the CIFAR-10 and CIFAR-100 dataset typically uses the DARTS search space and the same architecture space. The DARTS search space consists of the following operations: 3×3 MaxPooling, 3×3 AvgPooling, Identity, 3×3 SepConv, 5×5 SepConv, 3×3 DilConv, and 5×5 DilConv.

Architecture space and search settings. The `Search` is carried out on a small supernet, where the initial inputs are passed though a convolutional layer that outputs 16 initial channels, and the supernet consists of 8 stacked cells. To apply the group sparsity approach, the same operation in different cells of the same type is put into the same group. There are two types of cells: normal cells and reduction cells, which preserve and reduce the spatial resolution, respectively. Each cell consists of 2 input

	CIFAR-10		CIFAR-100	
	accuracy	search cost	accuracy	search cost
DARTS (2nd)	96.98 ± 0.13	1.46 days	73.40 ± 7.79	1.33 days
P-DARTS	97.05 ± 0.20	0.25 day	**83.46 ± 0.24**	0.34 day
PC-DARTS	**97.13 ± 0.16**	0.13 day	82.57 ± 0.71	0.15 day
DrNAS	96.95 ± 0.08	0.83 day	83.15 ± 0.23	0.90 day
GDAS [56]	96.63 ± 0.12	0.18 day	80.99 ± 0.34	0.36 day
ISTA-NAS [242]	96.64 ± 0.15	0.03 day	82.25 ± 0.77	0.03 day
GAEA [133]	96.12 ± 0.29	0.22 day	79.10 ± 0.89	0.22 day
GSparsity (prop.)	**97.17 ± 0.11**	0.42 day	**83.56 ± 0.34**	0.78 day

Table 6.4.: *NAS on DARTS search space for CIFAR-10/-100. All results are reproduced from their authors' implementations.*

nodes, 4 intermediate nodes and 1 output node. Each intermediate node is connected to the 2 input nodes and previous intermediate nodes. Nodes are connected though the 7 different operations of the search space, which results in a total of 98 different operations inside each cell. Figure 6.2 provides an illustration of this description with 3 intermediate nodes and 4 different operations in the search space.

The supernet is trained for 50 epochs on the full training set with 50k samples using ProxSGD, with a learning rate of $\varepsilon = 0.001$ (without a learning rate scheduler) and momentum of $\rho = 0.8$. The regularization gain μ_l is chosen according to (6.2) with $\mu = 60$. In order to find the architecture, the value of τ is set to $\tau_l^{(k)} = 1$. This removes the dependency of the squared gradient, as it is commonly used in optimizers like Adam for example. The resulting algorithm is more in line with optimizers like SGD with momentum. This approach speeds up the GSparsity method, while it is still able to find architectures that are able to outperform NAS methods like DARTS and DrNAS [37].

The regularization gain μ is tuned in a similar way as the bisection method. Firstly, values of μ are tried spanning a big range (such as μ =0.1, 1, 10, 100) to determine a small range in which the desirable μ (i.e. the desired sparsity level) lies. Then the bisection method is repeated in the small range, for example $[50, 100]$. One can typically find an appropriate μ within 10 trials. Note that in order to reduce the effort to tune μ, a seemingly obvious way is to use a small μ and only keep the top k operations with the largest ℓ_2-norm. However, this discretization step would incur notable performance degradation. Therefore, searching for the appropriate μ will reduce the performance degradation due to discretization and it is not an extra burden compared to other methods.

Evaluation settings. For `Evaluation`, similar settings are used as in DARTS [143]: 36 initial channels (after the first convolution), SGD with momentum (now *without* the $\ell_{2,1}$-norm regularization) for 600 epochs, learning rate $\varepsilon = 0.025$, momentum parameter $\rho = 0.9$, weight decay 3e-4, an auxiliary tower with weight 0.4, cutout regularization with length 16 and ScheduledDropPath [255] with the maximum drop probability 0.3. In order to have a network size that is comparable to other methods, 14 cells are stacked to form the evaluation network.

Results on CIFAR-10 and CIFAR-100. The comparison between the proposed GSparsity method and seven recent NAS algorithms is summarized in Table 6.4: DARTS [143], P-DARTS [39], PC-DARTS [238], DrNAS [37], GDAS [56], ISTA-NAS [242] and GAEA [133].[1] The GSparsity approach has the highest average accuracy, with a low standard deviation. Note that the accuracies of the baselines, albeit reproduced by using the authors' original implementations, are generally worse than reported in the respective papers. One possible reason is that, as discussed in the experimental protocol, in these experiments the average performance based on all architectures is considered (instead of the best architecture w.r.t. validation performance, as done by many recent papers)[2]. Thus, this concludes that the performance of GSparsity is both good and stable. Appendix C also summarizes the results for the best performing architecture. Figure 6.6 shows the normal and reduction cell of one of the three architectures that was found with the GSparsity method on CIFAR-10.

Similar observations are drawn from experiments on CIFAR-100 [114]. Here, the GSparsity method achieves the highest average accuracy of all methods. Again, there is a gap between the reproduced results and the reported accuracy in the respective papers for the same reasons as mentioned for the CIFAR-10 experiments. Figure 6.7 shows the normal and reduction cell of one of the three architectures that was found with the GSparsity method.

There is also the approach, where scaling factors are appended to the operations and then pruned (instead of directly pruning the weights). It turns out that directly pruning weights yields better results, see Appendix C for details. An ablation study on the effect of the regularization parameter μ on the structure of the final network is presented in Appendix E.

6.5.2. ImageNet 2012.

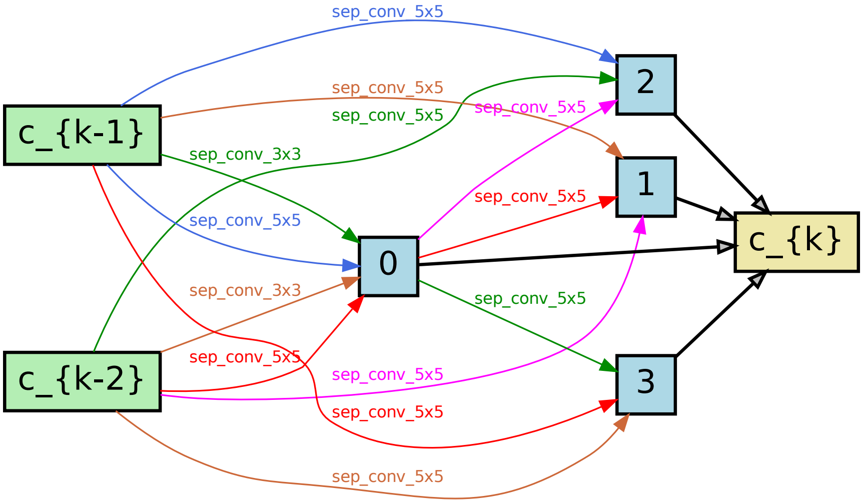
The search and evaluation network on ImageNet 2012 differs from the network used to search and evaluate CIFAR-10 and CIFAR-100. Similar to [37], [39] and [238], three convolutions with stride 2 are used in the beginning of the network in order to downscale the spatial resolution of the ImageNet samples to 28×28. This greatly reduces the size of the features in the hidden layers. Also, following the same approach as in [37], [39] and [238], the network is trained on only 10% of the ImageNet 2012 samples during `Search`. The hyperparameter settings in the search phase are the same as for the CIFAR-10 dataset. The model was trained in parallel on 4 Titan GPUs.

During `Evaluation`, the network is scaled to 14 cells with 48 initial channels, following previous works ([37], [39], [238]). The network is trained for 250 epochs using the SGD optimizer with momentum. An initial learning rate of $\varepsilon = 0.5$ is used, which is decayed down to zero using a cosine annealing scheduler, momentum $\rho = 0.9$, a batch-size of 512 and a weight decay value of 3e-5. The `evaluation` network also uses an auxiliary tower with auxiliary weight of 0.4, as well as label smoothing. The results are summarized in Table 6.5.

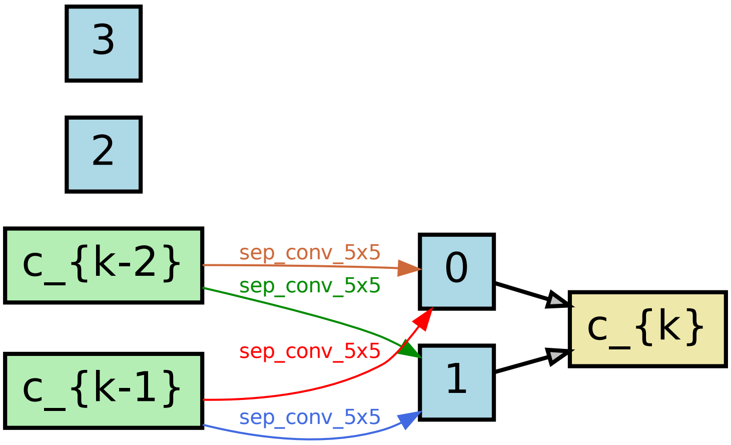
---

[1] It would be interesting to compare with HAPG [234], but the authors' implementation is not available yet.

[2] However, the performance of the best architecture that was experimentally reproduced is still worse than originally reported (with a non-negligible gap).



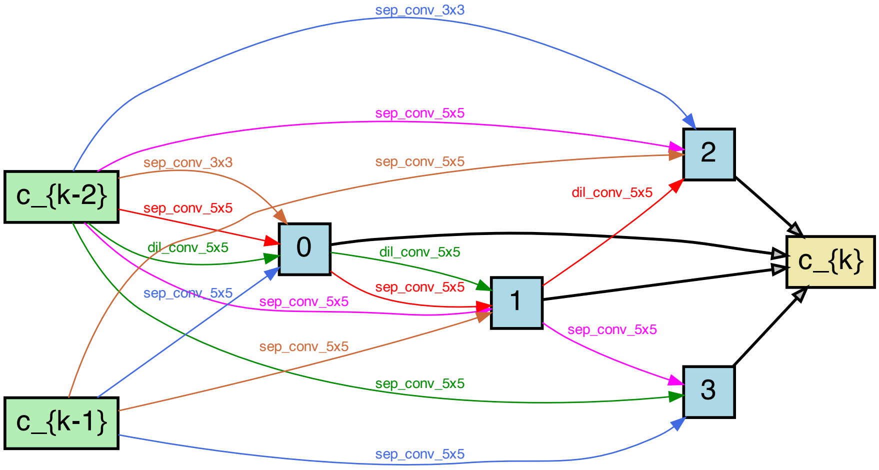
(a) *Normal cell found by GSparsity for the CIFAR-10 dataset.*



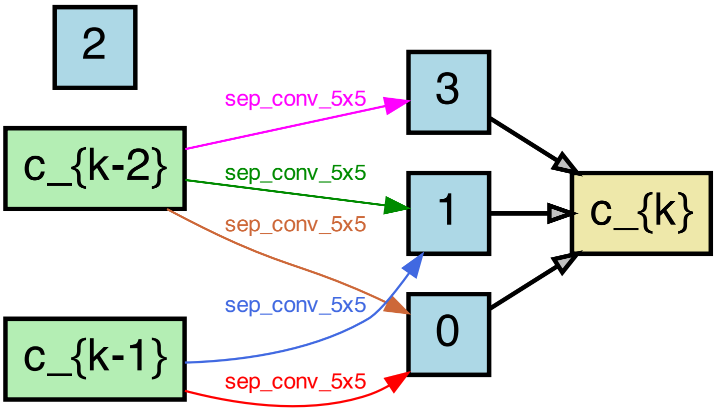
(b) *Reduction cell found by GSparsity for the CIFAR-10 dataset.*

Figure 6.6.: *Example of one architecture found by GSparsity on CIFAR-10. The restriction on the number of operations per node has been removed. In order to match the network size to other methods, less cells are stacked when evaluating the architecture.*

Note that because ImageNet 2012 is computationally demanding, each method in this table was run only once. The PC-DARTS method is able to achieve the highest top-1 accuracy of 75.71%, followed by the GSparsity approach, which reached



(a) *Normal cell found by GSparsity for the CIFAR-100 dataset.*



(b) *Reduction cell found by GSparsity for the CIFAR-100 dataset.*

Figure 6.7.: *Example of one architecture found by GSparsity on CIFAR-100. The restriction on the number of operations per node has been removed. In order to match the network size to other methods, less cells are stacked when evaluating the architecture.*

75.52%. DrNas was reproduced using the authors' implementation, but the accuracy reached only 65.25%, and it is notably below the one reported in [37]. It takes PC-DARTS and GSparsity roughly the same time to search for an architecture: 3.1 and 3.3 GPU days, respectively. DrNAS takes the longest with 7.3 GPU days. Also, it is important to note that the difference in the number of parameters is mainly due to the type of operations kept after `Search` is completed. For example, Identity has con-

	top-1 acc	top-5 acc	search cost	params
DARTS*	71.09	89.83	2.9 days	5.54M
P-DARTS*	74.11	91.73	0.3 days	3.67M
GSparsity* (ours)	75.29	92.42	0.5 days	6.29M
PC-DARTS	75.71	92.68	3.1 days	5.57M
DrNAS	65.25	86.23	7.3 days	3.33M
GSparsity (ours)	75.52	92.60	3.3 days	6.22M

Table 6.5.: *NAS on DARTS search space for ImageNet 2012 (*Architecture has been searched on CIFAR-10 or CIFAR-100). All results are reproduced from their authors' implementations.*

siderably less parameters than convolutions, and DARTS tends to keep the Identity operation. Since each method is expected to only keep 8 operations for each cell, the number of parameters can vary widely.

6.5.3. NAS-Bench-201 Search Space

In this subsection the GSparsity method is evaluated on the tabular NAS benchmark NAS-Bench-201 [57]. NAS-Bench-201 aims to serve as a NAS benchmark and consists of a large database that contains all possible 15,625 architectures of a specific search space and a predefined architecture structure. Every architecture in this database has a test and validation accuracy assigned to it. This allows for a fairer comparison between different methods, as it removes any possible differences in the evaluation procedure. It features experiments on the three different datasets CIFAR-10, CIFAR-100 and ImageNet-16-120.

Architecture space and search settings. NAS-Bench-201 employs a fixed cell-based structure similar to DARTS. The search space consists of the five operations Zero, Identity, 1×1 Convolution, 3×3 Convolution and 1×1 Average Pooling. The architecture has only one type of searchable cell and each cell contains 4 intermediate nodes, where each node is connected to the previous intermediate nodes. There are a total of 15 cells in the network and after every five cells there is a residual block with stride 2 which downsamples the spatial size and doubles the number of channels. To search for a single cell structure, operations of the same type across different cells are placed into the same group (cf. Figure 6.5). The network is trained with GSparsity for 100 epochs using the full training set. During training a learning rate of $\varepsilon_0 = 0.001$ is used that is decayed down to $\varepsilon_T = 0.0001$ using a cosine annealing scheduler. The momentum is set to $\rho_k = 0.8$ and $\tau_l^{(k)} = 1$. The regularization gain μ_l follows (6.2) where $\mu = 200$.

Evaluation settings. The architectures found in the `Search` phase are simply queried from the NAS-Bench-201 database to obtain validation/test accuracy. For these experiments the performance for each architecture was queried and the table reports the mean/standard deviation for the architectures resulting from 3 `Search` runs using different seeds.

	CIFAR-10		**CIFAR-100**		**ImageNet-16-120**	
	validation	test	validation	test	validation	test
DARTS (1st)	49.27 ± 13.4	59.84 ± 7.84	38.57 ± 0.00	38.97 ± 0.00	18.87 ± 0.00	18.41 ± 0.00
DARTS (2nd)	58.78 ± 13.4	65.38 ± 7.84	38.57 ± 0.00	38.97 ± 0.00	18.87 ± 0.00	18.41 ± 0.00
P-DARTS	64.72 ± 19.1	71.43 ± 14.2	38.57 ± 0.00	38.97 ± 0.00	28.03 ± 13.0	27.72 ± 13.2
GAEA	83.31 ± 1.31	83.18 ± 1.20	54.94 ± 0.26	54.88 ± 0.17	29.31 ± 3.19	28.42 ± 3.31
PC-DARTS	89.46 ± 1.05	93.06 ± 0.99	67.19 ± 1.36	67.76 ± 1.00	40.57 ± 0.77	40.84 ± 0.85
DrNAS	90.20 ± 0.00	**93.76 ± 0.00**	67.84 ± 1.74	67.62 ± 1.69	40.78 ± 0.00	**41.44 ± 0.00**
GSparsity (prop.)	90.20 ± 0.00	**93.76 ± 0.00**	70.71 ± 0.00	**71.11 ± 0.00**	40.78 ± 0.00	**41.44 ± 0.00**

Table 6.6.: *NAS on NAS-Bench-201 search space (reproduced from their authors' implementations).*

Results. Table 6.6 shows that GSparsity performs amongst the best for all three datasets on the NAS-Bench-201 search space. DrNAS performs similar to GSparsity, finding the same architectures on CIFAR-10 and ImageNet-16-120. On CIFAR-100 the proposed GSparsity method is able to find the best performing network compared to the other methods.

6.5.4. Robustness of GSparsity

Many NAS methods are prone to overfit their architecture parameters, which results in a "collapse" of the architecture. These architectures often result in a lot of skip-connections, and they do not perform well during evaluation. Thus, it is important to devise NAS methods that are robust to this type of collapse. Experiments in [248] show that DARTS performs poorly on different search spaces that only allow a subset of operations from the original DARTS search space. In this subsection, GSparsity is tested on the **S1**, **S2** and **S4** [3] spaces from [248]. **S1** consists of the same search network as in the DARTS paper, but with only 2 operation choices per edge (refer to Figure 9 in [248]). **S2** is similar but the choices at every edge are from the set {3×3 SepConv, Identity}. **S4** adds one harmful *Noise* operation to the set of operations in **S2**.

Search Space	DARTS*	DARTS-ES*	GSparsity (prop.)
S1	95.34 ± 0.71	**96.95 ± 0.07**	**96.94 ± 0.14**
S2	95.58 ± 0.40	96.59 ± 0.14	**97.40 ± 0.11**
S4	93.05 ± 0.18	95.83 ± 0.21	**97.36 ± 0.12**

Table 6.7.: *Performance of DARTS, DARTS-ES and the proposed GSparsity on CIFAR-10 (*Results taken from Table 1 of [248]).*

The search and evaluation settings are the same as for the DARTS search space, except that ScheduledDropPath has a maximum drop probability of 0.2 in `Evaluation`. Note that [248] proposed several methods to robustify DARTS, such as adaptive regularization and early stopping, which require computing the Hessian of the validation loss w.r.t. the architecture parameters in DARTS as proxy for the flatness of the loss landscape. These methods impose an additional overhead to `Search`. GSparsity does

---

[3] The search space **S3** in [248] is {3×3 SepConv, Identity, Zero}. This search space is not considered, because implicitly the operation Zero results from **S2** when none of {3×3 SepConv, Identity} is selected.

not rely on such heuristics and therefore has much lower runtime and memory requirements.

Table 6.7 evaluates the performance of GSparsity on these search spaces, comparing it against DARTS and DARTS-ES (DARTS with early stopping from [248]). GSparsity performs amongst the best on all three search spaces, and substantially better than DARTS-ES on **S2** and **S4**, with up to 1.53% absolute test accuracy improvement on **S4**. DARTS-ES in turn clearly outperforms DARTS on all spaces. This verifies the robustness of the proposed GSparsity algorithm.

This chapter introduced GSparsity, a unified approach for network pruning and one-shot neural architecture search via group sparsity. Experiments show its flexibility in performing various tasks while maintaining competitive performance. GSparsity is able to perform filter pruning, operation pruning as well as one-shot neural architecture search. This method is able to find well performing architectures that can be scaled to arbitrary size by stacking multiple cells. The benefits of this approach for very large neural networks are twofold. Firstly, very large neural networks can be trained from scratch with GSparsity using filter or operation pruning and the resulting network can be pruned without incurring any performance degradation. This removes the need of retraining very large neural networks after pruning in order to maintain competitive performance. Secondly, structured sparsity can be better exploited by hardware, which can significantly speed up inference and reduce the size of neural network architectures.

Further research can explore several different directions. The proximal algorithm used in GSparsity lends itself for efficient distributed computing [173]. One could develop a distributed proximal version of GSparsity for pruning and neural architecture search, which could tackle even bigger neural networks. Another possible direction for extending this work could involve relaxing the DARTS supernet architecture. Most bi-level optimization methods have to use costly second-order methods in order to update the architecture parameters. By keeping the same operations of different cells linked together, each operation of the search space for a unique cell type is represented by one α-parameter. This reduces the overall number of parameters that other methods have to update using second-order methods. GSparsity does not use those α-parameter and it does not rely on second-order optimization methods, thus one could prune each operation in each cell individually. This more flexible approach could increase the accuracy of the model, but it is not clear how to stack the resulting pruned cells in the evaluation network when each cell is unique. A third direction to extend this method is to consider a more general approximation subproblem than what is considered in ProxSGD. This would amount to solving a (strongly) convex function iteratively for a certain number of iterations in order to find a solution. One benefit would be a possible reduction in the number of steps until convergence. The drawback of this approach is the use of an iterative solver, which will increase the compute time per iteration.

Chapter 7

Conclusions and Outlook

Despite the impressive performance of deep neural networks, many of the recent advances in accuracy correlate with an increase in the size of these models. Nowadays, in order to push for even higher performance, very large neural networks have been developed by certain groups that have the resources to distribute their training onto multiple nodes. Due to their size, each iteration during training becomes increasingly costly. Also, very large neural networks take up a large portion of the GPU memory, which in turn limits the number of samples that can be used in each mini-batch, which further increases the training time. Different methods have been proposed to reduce the number of iterations to convergence. There have also been many new proposed optimizers that try to efficiently train large neural networks across different nodes on a cluster.

Tools that allow for an investigation of various loss landscape properties per iteration can assist researchers in exploring different aspects of proposed optimizers and neural network models. This can help in making informed decisions on future optimizer and neural network design, which are more efficient than current methods and models. Also, since the performance of most state-of-the-art networks correlates with an increase in network size, cutting edge research is only possible for institutions with access to huge compute resources. In order to democratize the field of deep learning and to make it more environmentally friendly by reducing the amount of compute used for training, the field of AutoML and NAS tries to automatically find performant models with a reasonable size.

This thesis introduced several ways to deal with very large neural networks, by providing tools for loss landscape investigation, which can help in the development of more efficient optimizers and neural network models. Also, this thesis explores the use of proximal methods, which on the one hand provide an efficient way for shrinking the model size of neural networks through unstructured pruning, and on the other hand allow for efficient structured pruning as well as neural architecture search.

Firstly, this thesis contributes to the field of optimization and deep learning by combining efficient eigenvalue computation with high dimensional loss surface visualization in order to find meaningful directions in the parameter space of deep neural networks. This allows for an investigation into the behavior of different optimizers as well as different types of deep neural network models. In order to speed up the computations of loss surface visualization, the different points of the input grid are computed in parallel. Also, a novel iteration parallel evaluation method is introduced for computing the stochastic Lanczos quadrature algorithm, which allows for more efficient parallelization than the data parallel approach. All of these methods are available in the GradVis toolbox, which works with the popular libraries PyTorch and TensorFlow. Using these tools, this thesis depicts for the fist time the loss landscape of a neural network in the direction of various eigenvectors as well as between two local minima.

Future possible research directions include the investigation of different types of models and optimizers using the eigenvalues and eigenvectors. One could also investigate the eigenspectra of different parts of a network, for example how different filters in a convolution converge during training. Also, one could investigate certain phenomena in deep learning that are not well understood, like the large batch size problem [199], in order to observe how the loss landscape differs for different batch sizes. This could potentially motivate the development of a novel optimizer that does not suffer from the large batch size problem. A third possible direction is to combine the proposed method-parallel stochastic Lanczos algorithm with the previously introduced data parallel method. This could potentially speed up the calculation of the full eigenvalue spectrum even further.

Next, this thesis contributes to the field of GANs, by using the eigenvalue and visualization tools in order to visualize their full eigenvalue densities at different iterations for the first time. This allowed to observe how GANs that suffer from mode collapse behave differently, compared to instances where the network does not suffer from mode collapse. This lead to the introduction of NudgedAdam, an optimizer that effectively regularizes GANs and thus prevents the networks from mode collapse.

This line of work could be extended by investigating other GAN architectures, in order to observe how these architectures behave under mode collapse. In this work, convolutional GANs still suffered from mode collapse, even after optimization with NudgedAdam. Thus, observing the eigenvalue spectra of convolutional GANs and comparing them to their fully-connected version could reveal how they differ during optimization and possibly lead to new optimizers that prevent more types of GANs from mode collapse.

In order to reduce the size of very large neural networks, this thesis introduced ProxSGD, the first optimizer using proximal updates for stochastic preconditioned gradient methods. This optimizer has a convergence guarantee and is able to unify multiple popular optimizers into one framework. Experiments show that ProxSGD is able to find sparser networks while reaching similar accuracies compared to commonly used optimizers.

This work could be extended in a number of ways. Firstly, there is no reason why the ℓ_2-norm in the proximal operator should be favored over other distance measures. One could formulate the optimizer using the Bregman divergence [31, 60] or entropic penalties [220]. Secondly, instead of using the quadratic objective in the ProxSGD formulation, one could aim at solving the proximal operator using a more general convex function. This would probably require solving the subproblem iteratively, which slows the per iteration computation, but on the other hand this could reduce the number of iterations to convergence. Thirdly, one could extend ProxSGD to a distributed setting. There are several ways to define proximal gradient methods for distributed training. One could for example use ProxSGD with ℓ_1-regularization, in order to more effectively compress the information that is sent between workers (see [77] for a similar approach), though there are many other directions one could exploit the proximal operator in the distributed setting [173].

Lastly, this thesis unifies sparsity and one-shot NAS through operation pruning by using ProxSGD with $\ell_{2,1}$-regularization. Experiments show that using group sparsity via ProxSGD achieves better results for filter pruning compared to previous heuristic proximal algorithms. Also, the group sparsity approach allows for pruning entire operations, which is achieved by grouping all trainable parameters of each operation together. In the NAS setting, GSparsity casts the NAS problem as a single-level optimization problem, which renders the architecture parameters used in most one-shot methods useless. This problem can be solved optimally by the ProxSGD algorithm, due to its convergence guarantee. GSparsity forces the weights of non-important groups exactly to zero and is thus able to converge to a group-sparse solution. Thus, GSparsity does not suffer any performance degradation in the discretization step, contrary to previous methods. An additional benefit to this approach is that GSparsity is robust to a ”collapse” of the architecture, which is an issue for many other methods.

The GSparsity method can be extended in multiple ways. Future research could investigate whether Bregman divergences or entropic penalties improve performance even further. Also, one could incorporate the partial-channel connections introduced in PC-DARTS [238], which reduce the computation time significantly and have been used in many recent NAS methods [37, 42]. Also, since GSparsity does not rely on architecture parameters, one can remove the dependence on cell structure during training, and treat each operation individually. This is very costly for other methods, as many rely on second-order optimization of the architectural parameters in order to converge to a solution. Lastly, one could try and use the structured pruning capabilities of group sparsity for distributed training, by using structured pruning in order to speed up time to convergence.

Bibliography

[1] Abukmeil, M., Ferrari, S., Genovese, A., Piuri, V., & Scotti, F. (2021). "A survey of unsupervised generative models for exploratory data analysis and representation learning". *ACM Comput. Surv.*, *54*(5).

[2] Adolphs, L., Daneshmand, H., Lucchi, A., & Hofmann, T. (2019). "Local saddle point optimization: A curvature exploitation approach". *Proceedings of the Twenty-Second International Conference on Artificial Intelligence and Statistics*, *89*, 486–495.

[3] Adorio, E. P. (2005). "Mvf - multivariate test functions library in c for unconstrained global optimization".

[4] Aizerman, M. A. (1964). "Theoretical foundations of the potential function method in pattern recognition learning". *Automation and remote control*, *25*, 821–837.

[5] Akbari, A., Awais, M., Bashar, M., & Kittler, J. (2021). "How does loss function affect generalization performance of deep learning? application to human age estimation". *Proceedings of the 38th International Conference on Machine Learning*, *139*, 141–151.

[6] Amari, S. (1998). "Natural gradient works efficiently in learning". *Neural Computation*, *10*(2), 251–276.

[7] Amdahl, G. M. (1967). "Validity of the single processor approach to achieving large scale computing capabilities". *American Federation of Information Processing Societies: Proceedings of the AFIPS '67 Spring Joint Computer Conference, April 18-20, 1967, Atlantic City, New Jersey, USA*, *30*, 483–485.

[8] Angeline, P., Saunders, G., & Pollack, J. (1994). "An evolutionary algorithm that constructs recurrent neural networks". *IEEE Transactions on Neural Networks*, *5*(1), 54–65.

[9] Antognini, J., & Sohl-Dickstein, J. (2018). "Pca of high dimensional random walks with comparison to neural network training". *Advances in Neural Information Processing Systems*, *31*.

[10] Arbenz, P. (2016). "Lecture notes on solving large scale eigenvalue problems".

[11] Arjovsky, M., Chintala, S., & Bottou, L. (2017). "Wasserstein generative adversarial networks". *Proceedings of the 34th International Conference on Machine Learning*, *70*, 214–223.

[12] Arjovsky, M., & Bottou, L. (2017). "Towards principled methods for training generative adversarial networks". *5th International Conference on Learning*

Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings.

[13] Arora, S., Ge, R., Liang, Y., Ma, T., & Zhang, Y. (2017). “Generalization and equilibrium in generative adversarial nets (GANs)”. *Proceedings of the 34th International Conference on Machine Learning*, *70*, 224–232.

[14] Bach, F., Jenatton, R., Mairal, J., & Obozinski, G. (2011). “Optimization with Sparsity-Inducing Penalties”. *Foundations and Trends in Machine Learning*, *4*(1), 1–106.

[15] Baevski, A., Zhou, Y., Mohamed, A., & Auli, M. (2020). “Wav2vec 2.0: A framework for self-supervised learning of speech representations”. *Advances in Neural Information Processing Systems*, *33*, 12449–12460.

[16] Ballard, D. H. (1987). “Modular learning in neural networks.” *Aaai*, *647*, 279–284.

[17] Balles, L., & Hennig, P. (2018). “Dissecting adam: The sign, magnitude and variance of stochastic gradients”. *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, *80*, 413–422.

[18] Beck, A. (2017). “First-order methods in optimization”. Society for Industrial; Applied Mathematics.

[19] Belkin, M., Hsu, D., Ma, S., & Mandal, S. (2019). “Reconciling modern machine-learning practice and the classical bias–variance trade-off”. *Proceedings of the National Academy of Sciences*, *116*(32), 15849–15854.

[20] Belkin, M., Hsu, D. J., & Mitra, P. (2018). “Overfitting or perfect fitting? risk bounds for classification and regression rules that interpolate”. *Advances in neural information processing systems*, *31*.

[21] Bender, G., Kindermans, P.-J., Zoph, B., Vasudevan, V., & Le, Q. (2018). “Understanding and simplifying one-shot architecture search”. *Proceedings of the 35th International Conference on Machine Learning*, *80*, 550–559.

[22] Berard, H., Gidel, G., Almahairi, A., Vincent, P., & Lacoste-Julien, S. (2020). “A closer look at the optimization landscapes of generative adversarial networks”. *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*.

[23] Bertsekas, D. P., & Tsitsiklis, J. N. (2000). “Gradient convergence in gradient methods with errors”. *SIAM Journal on Optimization*, *10*(3), 627–642.

[24] Bertsekas, D. (1999). “Nonlinear programming”. Athena Scientific.

[25] Biggio, B., Corona, I., Maiorca, D., Nelson, B., Šrndić, N., Laskov, P., Giacinto, G., & Roli, F. (2013). “Evasion attacks against machine learning at test time”. *Joint European conference on machine learning and knowledge discovery in databases*, 387–402.

[26] Blum, A. L., & Rivest, R. L. (1992). “Training a 3-node neural network is np-complete”. *Neural Networks*, *5*(1), 117–127.

[27] Boltzmann, L. (1868). “Studien uber das gleichgewicht der lebenden kraft”. *Wissenschaftliche Abhandlungen*, *1*, 49–96.

[28] Bottou, L. (2010). “Large-scale machine learning with stochastic gradient descent”. *Proceedings of COMPSTAT’2010*, 177–186.

[29] Boyd, S., Parikh, N., Chu, E., Peleato, B., & Eckstein, J. (2010). "Distributed optimization and statistical learning via the alternating direction method of multipliers". *Foundations and Trends in Machine Learning*, *3*(1).

[30] Bray, A. J., & Dean, D. S. (2007). "Statistics of critical points of gaussian fields on large-dimensional spaces". *Phys. Rev. Lett.*, *98*, 150201.

[31] Bregman, L. (1967). "The relaxation method of finding the common point of convex sets and its application to the solution of problems in convex programming". *USSR Computational Mathematics and Mathematical Physics*, *7*(3), 200–217.

[32] Brock, A., Lim, T., Ritchie, J., & Weston, N. (2018). "SMASH: One-shot model architecture search through hypernetworks". *International Conference on Learning Representations*.

[33] Brown, N., Bakhtin, A., Lerer, A., & Gong, Q. (2020). "Combining deep reinforcement learning and search for imperfect-information games". *Advances in Neural Information Processing Systems*, *33*, 17057–17069.

[34] Brown, N., & Sandholm, T. (2019). "Superhuman ai for multiplayer poker". *Science*, *365*, eaay2400.

[35] Censor, Y., & Zenios, S. (1992). "On the proximal minimization algorithm with d-functions". *Journal of Optimization Theory and Applications*, *73*, 451–464.

[36] Chen, S., Dobriban, E., & Lee, J. (2020). "A group-theoretic framework for data augmentation". *Advances in Neural Information Processing Systems*, *33*, 21321–21333.

[37] Chen, X., Wang, R., Cheng, M., Tang, X., & Hsieh, C.-J. (2021). "DrNAS: Dirichlet neural architecture search". *International Conference on Learning Representations*.

[38] Chen, X., Liu, S., Sun, R., & Hong, M. (2019). "On the convergence of a class of ADAM-type algorithms for non-convex optimization". *International Conference on Learning Representations*.

[39] Chen, X., Xie, L., Wu, J., & Tian, Q. (2019). "Progressive differentiable architecture search: Bridging the depth gap between search and evaluation". *Proceedings of the IEEE International Conference on Computer Vision*, 1294–1303.

[40] Chen, Y., & Hu, H. (2019). "An improved method for semantic image inpainting with gans: Progressive inpainting". *Neural Processing Letters*, *49*.

[41] Chrabaszcz, P., Loshchilov, I., & Hutter, F. (2017). "A downsampled variant of imagenet as an alternative to the CIFAR datasets". *CoRR*, *abs/1707.08819*.

[42] Chu, X., Wang, X., Zhang, B., Lu, S., Wei, X., & Yan, J. (2021). "DARTS-: robustly stepping out of performance collapse without indicators". *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*.

[43] Chung, Y.-A., Zhang, Y., Han, W., Chiu, C.-C., Qin, J., Pang, R., & Wu, Y. (2021). "W2v-bert: Combining contrastive learning and masked language modeling for self-supervised speech pre-training". *2021 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*, 244–250.

[44] Clanuwat, T., Bober-Irizar, M., Kitamoto, A., Lamb, A., Yamamoto, K., & Ha, D. (2018). “Deep learning for classical japanese literature”. *CoRR*, *abs/1812.01718*.

[45] Cohen, G., Afshar, S., Tapson, J., & van Schaik, A. (2017). “Emnist: An extension of mnist to handwritten letters”.

[46] Colson, B., Marcotte, P., & Savard, G. (2007). “An overview of bilevel optimization”. *Annals of Operations Research*, *153*(1), 235–256.

[47] Conn, A. R., Scheinberg, K., & Vicente, L. N. (2009). “Introduction to derivative-free optimization” (Vol. 8). SIAM.

[48] Cullum, J., & Donath, W. E. (1974). “A block lanczos algorithm for computing the q algebraically largest eigenvalues and a corresponding eigenspace of large, sparse, real symmetric matrices”. *1974 IEEE Conference on Decision and Control including the 13th Symposium on Adaptive Processes*, 505–509.

[49] Dai, Z., Liu, H., Le, Q. V., & Tan, M. (2021). “Coatnet: Marrying convolution and attention for all data sizes”. *CoRR*, *abs/2106.04803*.

[50] Darken, C., & Moody, J. (1991). “Towards faster stochastic gradient search”. *Advances in Neural Information Processing Systems*, *4*.

[51] Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., Ranzato, M., Senior, A., Tucker, P., Yang, K., Le, Q., & Ng, A. (2012). “Large scale distributed deep networks”. *Advances in Neural Information Processing Systems*, *25*.

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

[53] Dieleman, S., Fauw, J. D., & Kavukcuoglu, K. (2016). “Exploiting cyclic symmetry in convolutional neural networks”. *Proceedings of The 33rd International Conference on Machine Learning*, *48*, 1889–1898.

[54] Ding, X., Hao, T., Tan, J., Liu, J., Han, J., Guo, Y., & Ding, G. (2021). “Resrep: Lossless cnn pruning via decoupling remembering and forgetting”. *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*, 4490–4500.

[55] Dinh, L., Pascanu, R., Bengio, S., & Bengio, Y. (2017). “Sharp minima can generalize for deep nets”. *International Conference on Machine Learning*, 1019–1028.

[56] Dong, X., & Yang, Y. (2019). “Searching for a robust neural architecture in four GPU hours”. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, *2019-June*, 1761–1770.

[57] Dong, X., & Yang, Y. (2020). “NAS-Bench-201: Extending the scope of reproducible neural architecture search”. *International Conference on Learning Representations*.

[58] Duchi, J. C., Hazan, E., & Singer, Y. (2011). “Adaptive subgradient methods for online learning and stochastic optimization”. *J. Mach. Learn. Res.*, *12*, 2121–2159.

[59] Durall, R., Keuper, M., & Keuper, J. (2020). “Watch your up-convolution: Cnn based generative deep neural networks are failing to reproduce spectral

distributions". *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 7887–7896.

[60] Eckstein, J. (1993). "Nonlinear proximal point algorithms using bregman functions, with applications to convex programming". *Mathematics of Operations Research*, *18*(1), 202–226.

[61] Edunov, S., Ott, M., Auli, M., & Grangier, D. (2018). "Understanding back-translation at scale". *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, 489–500.

[62] Elsken, T., Metzen, J. H., & Hutter, F. (2019). "Neural architecture search: A survey". *J. Mach. Learn. Res.*, *20*, 55:1–55:21.

[63] Fiez, T., Chasnov, B., & Ratliff, L. (2020). "Implicit learning dynamics in stackelberg games: Equilibria characterization, convergence analysis, and empirical study". *Proceedings of the 37th International Conference on Machine Learning*, *119*, 3133–3144.

[64] F.R.S., K. P. (1901). "Liii. on lines and planes of closest fit to systems of points in space". *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, *2*(11), 559–572.

[65] Gale, T., Elsen, E., & Hooker, S. (2019). *The State of Sparsity in Deep Neural Networks*.

[66] Ge, R., Huang, F., Jin, C., & Yuan, Y. (2015). "Escaping from saddle points—online stochastic gradient for tensor decomposition". *Conference on learning theory*, 797–842.

[67] Getreuer, P., Garcia-Dorado, I., Isidoro, J., Choi, S., Ong, F., & Milanfar, P. (2018). "Blade: Filter learning for general purpose computational photography". *2018 IEEE International Conference on Computational Photography (ICCP)*, 1–11.

[68] Ghorbani, B., Krishnan, S., & Xiao, Y. (2019). "An investigation into neural net optimization via hessian eigenvalue density". *CoRR*, *abs/1901.10159*.

[69] Gill, P., Murray, W., & Wright, M. (1981). "Practical optimization". Academic Press.

[70] Golub, G. H., & Reinsch, C. (1971). "Singular value decomposition and least squares solutions". *Handbook for automatic computation: Volume ii: Linear algebra* (pp. 134–151). Springer Berlin Heidelberg.

[71] Golub, G. H., & Welsch, J. H. (1969). "Calculation of gauss quadrature rules". *Mathematics of Computation*, *23*(106), 221–221.

[72] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., & Bengio, Y. (2014). "Generative adversarial nets". *Advances in Neural Information Processing Systems*, *27*.

[73] Goodfellow, I., Vinyals, O., & Saxe, A. (2015). "Qualitatively characterizing neural network optimization problems". *International Conference on Learning Representations*.

[74] Goodfellow, I., Warde-Farley, D., Mirza, M., Courville, A., & Bengio, Y. (2013). "Maxout networks". *Proceedings of the 30th International Conference on Machine Learning*, *28*(3), 1319–1327.

[75] Goodfellow, I. J. (2017). "NIPS 2016 tutorial: Generative adversarial networks". *CoRR*, *abs/1701.00160*.

[76] Goodfellow, I. J., Bengio, Y., & Courville, A. (2016). "Deep learning" [http://www.deeplearningbook.org]. MIT Press.

[77] Grishchenko, D., Iutzeler, F., Malick, J., & Amini, M. (2021). "Distributed learning with sparse communications by identification". *SIAM J. Math. Data Sci.*, *3*(2), 715–735.

[78] Grother, P. (1995). "Nist special database 19 handprinted forms and characters database".

[79] Gulrajani, I., Ahmed, F., Arjovsky, M., Dumoulin, V., & Courville, A. C. (2017). "Improved training of wasserstein gans". *Advances in Neural Information Processing Systems*, *30*.

[80] Han, J., Kamber, M., & Pei, J. (2012). "Data mining: Concepts and techniques". Elsevier Inc.

[81] Han, J., & Moraga, C. (1995). "The influence of the sigmoid function parameters on the speed of backpropagation learning". *International workshop on artificial neural networks*, 195–201.

[82] Hastie, T., Tibshirani, R., Friedman, J. H., & Friedman, J. H. (2009). "The elements of statistical learning: Data mining, inference, and prediction" (Vol. 2). Springer.

[83] He, K., Zhang, X., Ren, S., & Sun, J. (2016). "Deep residual learning for image recognition", 770–778.

[84] He, Y., Liu, P., Wang, Z., Hu, Z., & Yang, Y. (2019). "Filter pruning via geometric median for deep convolutional neural networks acceleration". *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, 4340–4349.

[85] Hendrycks, D., & Gimpel, K. (2016). "A baseline for detecting misclassified and out-of-distribution examples in neural networks". *CoRR*, *abs/1610.02136*.

[86] Hestness, J., Narang, S., Ardalani, N., Diamos, G. F., Jun, H., Kianinejad, H., Patwary, M. M. A., Yang, Y., & Zhou, Y. (2017). "Deep learning scaling is predictable, empirically". *CoRR*, *abs/1712.00409*.

[87] Himmelblau, D. M. (1972). "Applied nonlinear programming". McGraw-Hill.

[88] Hinton, G. E., Osindero, S., & Teh, Y.-W. (2006). "A fast learning algorithm for deep belief nets". *Neural Comput.*, *18*(7), 1527–1554.

[89] Hinton, G. E., & van Camp, D. (1993). "Keeping the neural networks simple by minimizing the description length of the weights". *Proceedings of the Sixth Annual ACM Conference on Computational Learning Theory, COLT 1993, Santa Cruz, CA, USA, July 26-28, 1993*, 5–13.

[90] Hochbaum, D. S. (2007). "Complexity and algorithms for nonlinear optimization problems". *Annals of Operations Research*, *153*(1), 257–296.

[91] Hochreiter, S. (1991). "Untersuchungen zu dynamischen neuronalen Netzen". *Master's thesis, Institut für Informatik, Technische Universität, München*, *1*, 1–150.

[92] Hochreiter, S., & Schmidhuber, J. (1997a). "Flat minima". *Neural Computation*, *9*(1), 1–42.

[93] Hochreiter, S., & Schmidhuber, J. (1997b). “Long short-term memory”. *Neural Comput.*, *9*(8), 1735–1780.

[94] Hoffer, E., Hubara, I., & Soudry, D. (2017). “Train longer, generalize better: Closing the generalization gap in large batch training of neural networks”. *Advances in Neural Information Processing Systems*, *30*.

[95] Huang, G., Liu, Z., Van Der Maaten, L., & Weinberger, K. Q. (2017). “Densely connected convolutional networks”. *Proceedings of the IEEE conference on computer vision and pattern recognition*, 4700–4708.

[96] Huang, Z., & Wang, N. (2018). “Data-driven sparse structure selection for deep neural networks”. *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part XVI*, *11220*, 317–334.

[97] Huszar, F. (2015). “How (not) to train your generative model: Scheduled sampling, likelihood, adversary?” *CoRR*, *abs/1511.05101*.

[98] Ioffe, S., & Szegedy, C. (2015). “Batch normalization: Accelerating deep network training by reducing internal covariate shift”. *CoRR*, *abs/1502.03167*.

[99] Jain, P., & Kar, P. (2017). “Non-convex optimization for machine learning”. *Found. Trends Mach. Learn.*, *10*(3-4), 142–336.

[100] James, G., Witten, D., Hastie, T., & Tibshirani, R. (2013). “An introduction to statistical learning” (Vol. 112). Springer.

[101] Janiesch, C., Zschech, P., & Heinrich, K. (2021). “Machine learning and deep learning”. *Electronic Markets*, *31*(3), 685–695.

[102] Jastrzebski, S., Kenton, Z., Ballas, N., Fischer, A., Bengio, Y., & Storkey, A. J. (2019). “On the relation between the sharpest directions of DNN loss and the SGD step length”. *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*.

[103] Ji, P., Zhang, T., Li, H., Salzmann, M., & Reid, I. (2017). “Deep subspace clustering networks”. *Advances in Neural Information Processing Systems*, *30*.

[104] Jordan, M. I. (1986). “Serial order: A parallel distributed processing approach. technical report, june 1985-march 1986”.

[105] Judd, J. S. (1990). “Neural network design and the complexity of learning”. MIT Press.

[106] Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., & Amodei, D. (2020). “Scaling laws for neural language models”. *CoRR*, *abs/2001.08361*.

[107] Karras, T., Laine, S., Aittala, M., Hellsten, J., Lehtinen, J., & Aila, T. (2020). “Analyzing and improving the image quality of stylegan”. *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 8107–8116.

[108] Kelley, H. J. (1960). “Gradient theory of optimal flight paths”. *Ars Journal*, *30*(10), 947–954.

[109] Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., & Tang, P. T. P. (2017). “On large-batch training for deep learning: Generalization gap and sharp minima”. *5th International Conference on Learning Representations,*

ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings.

[110] Kingma, D. P., & Ba, J. (2015). “Adam: A method for stochastic optimization”. *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

[111] Kondor, R., & Trivedi, S. (2018). “On the generalization of equivariance and convolution in neural networks to the action of compact groups”. *International Conference on Machine Learning*, 2747–2755.

[112] Kovachki, N. B., & Stuart, A. M. (2019). “Ensemble kalman inversion: A derivative-free technique for machine learning tasks”. *Inverse Problems*, *35*(9).

[113] Kovalev, D., Mishchenko, K., & Richtárik, P. (2019). “Stochastic newton and cubic newton methods with simple local linear-quadratic rates”. *CoRR*, *abs/1912.01597*.

[114] Krizhevsky, A. (2009). “Learning multiple layers of features from tiny images”, 32–33.

[115] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). “Imagenet classification with deep convolutional neural networks”. *Advances in Neural Information Processing Systems*, *25*.

[116] Krylov, A. N. (1931). “On the numerical solution of the equation by which in technical questions frequencies of small oscillations of material systems are determined”. *Izvestija AN SSSR (News of Academy of Sciences of the USSR), Otdel. mat. i estest. nauk*, *7*(4), 491–539.

[117] Kuhn, H. W., & Tucker, A. W. (1951). “Nonlinear programming”. *Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability*, 481–492.

[118] Kunstner, F., Hennig, P., & Balles, L. (2019). “Limitations of the empirical fisher approximation for natural gradient descent”. *Advances in Neural Information Processing Systems*, *32*.

[119] L. V. Kantorovich. (1949). “On newton’s method”. *Collected works on approximation analysis of the Leningrad Branch of the Institute*, *28*, 104–144.

[120] Lanczos, C. (1950). “An iteration method for the solution of the eigenvalue problem of linear differential and integral operators.” *Journal of research of the National Bureau of Standards, 45, 255–282.*

[121] Laurent, T., & Brecht, J. (2018). “The multilinear structure of relu networks”. *International conference on machine learning*, 2908–2916.

[122] Lecun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). “Gradient-based learning applied to document recognition”. *Proceedings of the IEEE*, *86*(11), 2278–2324.

[123] LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., & Jackel, L. D. (1989). “Backpropagation applied to handwritten zip code recognition”. *Neural computation*, *1*(4), 541–551.

[124] LeCun, Y., & Cortes, C. (2010). “MNIST handwritten digit database”.

[125] Lee, J. (2006). “Riemannian manifolds: An introduction to curvature”. Springer New York.

[126] Lee, K., Lee, K., Lee, H., & Shin, J. (2018). “A simple unified framework for detecting out-of-distribution samples and adversarial attacks”. *Advances in Neural Information Processing Systems*, *31*.

[127] Lee, K., Lee, I., & Lee, S. (2018). “Propagating lstm: 3d pose estimation based on joint interdependency”. *Computer Vision – ECCV 2018: 15th European Conference, Munich, Germany, September 8–14, 2018, Proceedings, Part VII*, 123–141.

[128] Lei, Y., Hu, T., & Tang, K. (2019). “Stochastic gradient descent for nonconvex learning without bounded gradient assumptions”.

[129] Lemley, J., Bazrafkan, S., & Corcoran, P. (2017). “Smart augmentation learning an optimal data augmentation strategy”. *IEEE Access*, *5*, 5858–5869.

[130] Lewkowycz, A., & Gur-Ari, G. (2020). “On the training dynamics of deep networks with l_2 regularization”. *Advances in Neural Information Processing Systems*, *33*, 4790–4799.

[131] Li, H., Xu, Z., Taylor, G., Studer, C., & Goldstein, T. (2018). “Visualizing the loss landscape of neural nets”. *Advances in Neural Information Processing Systems*, *31*.

[132] Li, K., & Malik, J. (2018). “On the implicit assumptions of gans”. *CoRR*, *abs/1811.12402*.

[133] Li, L., Khodak, M., Balcan, M., & Talwalkar, A. (2020). “Geometry-aware gradient algorithms for neural architecture search”. *CoRR*, *abs/2004.07802*.

[134] Li, L., & Talwalkar, A. (2020). “Random search and reproducibility for neural architecture search”. *Proceedings of The 35th Uncertainty in Artificial Intelligence Conference*, *115*, 367–377.

[135] Li, X., Wang, S., & Cai, Y. (2019). “Tutorial: Complexity analysis of singular value decomposition and its variants”.

[136] Li, Y., Gu, S., Mayer, C., Gool, L. V., & Timofte, R. (2020). “Group sparsity: The hinge between filter pruning and decomposition for network compression”. *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020*, 8015–8024.

[137] Liao, Z., & Mahoney, M. W. (2021). “Hessian eigenspectra of more realistic nonlinear models”. *Advances in Neural Information Processing Systems*, *34*, 20104–20117.

[138] Lin, J. (1991). “Divergence measures based on the shannon entropy”. *IEEE Trans. Inf. Theory*, *37*(1), 145–151.

[139] Lin, L., Saad, Y., & Yang, C. (2016). “Approximating spectral densities of large matrices”. *SIAM Rev.*, *58*(1), 34–65.

[140] Lin, S., Ji, R., Yan, C., Zhang, B., Cao, L., Ye, Q., Huang, F., & Doermann, D. S. (2019). “Towards optimal structured CNN pruning via generative adversarial learning”. *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, 2790–2799.

[141] Lin, T., Jin, C., & Jordan, M. (2020). “On gradient descent ascent for nonconvex-concave minimax problems”. *Proceedings of the 37th International Conference on Machine Learning*, *119*, 6083–6093.

[142] Lindauer, M., & Hutter, F. (2020). "Best practices for scientific research on neural architecture search". *Journal of Machine Learning Research*, *21*(243), 1–18.

[143] Liu, H., Simonyan, K., & Yang, Y. (2019). "DARTS: Differentiable architecture search". *International Conference on Learning Representations*.

[144] Liu, Z., Li, J., Shen, Z., Huang, G., Yan, S., & Zhang, C. (2017). "Learning efficient convolutional networks through network slimming". *IEEE International Conference on Computer Vision, ICCV 2017, Venice, Italy, October 22-29, 2017*, 2755–2763.

[145] Long, J., Shelhamer, E., & Darrell, T. (2015). "Fully convolutional networks for semantic segmentation". *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 3431–3440.

[146] Louizos, C., Welling, M., & Kingma, D. P. (2018). "Learning Sparse Neural Networks through L_0 Regularization". *International Conference on Learning Representations*, 1–13.

[147] Luengo, J., García, S., & Herrera, F. (2011). "On the choice of the best imputation methods for missing values considering three groups of classification methods". *Knowledge and Information Systems*, *32*, 77–108.

[148] Luo, J., Wu, J., & Lin, W. (2017). "Thinet: A filter level pruning method for deep neural network compression". *IEEE International Conference on Computer Vision, ICCV 2017, Venice, Italy, October 22-29, 2017*, 5068–5076.

[149] Luo, W., Li, Y., Urtasun, R., & Zemel, R. (2016). "Understanding the effective receptive field in deep convolutional neural networks". *Advances in Neural Information Processing Systems*, *29*.

[150] Maas, A. L., Hannun, A. Y., Ng, A. Y. et al. (2013). "Rectifier nonlinearities improve neural network acoustic models". *Proc. icml*, *30*(1), 3.

[151] Marquardt, D. W. (1963). "An algorithm for least-squares estimation of nonlinear parameters". *Journal of the Society for Industrial and Applied Mathematics*, *11*(2), 431–441.

[152] Martın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Jia, Y., Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, … Xiaoqiang Zheng. (2015). "TensorFlow: Large-scale machine learning on heterogeneous systems" [Software available from tensorflow.org].

[153] Meng, S. Y., Vaswani, S., Laradji, I. H., Schmidt, M., & Lacoste-Julien, S. (2020). "Fast and furious convergence: Stochastic second order methods under interpolation". *The 23rd International Conference on Artificial Intelligence and Statistics, AISTATS 2020, 26-28 August 2020, Online [Palermo, Sicily, Italy]*, *108*, 1375–1386.

[154] Mescheder, L., Nowozin, S., & Geiger, A. (2017). "The numerics of gans". *Advances in Neural Information Processing Systems*, *30*.

[155] Message Passing Interface Forum. (2021). *MPI: A message-passing interface standard version 4.0*.

[156] Min, E., Guo, X., Liu, Q., Zhang, G., Cui, J., & Long, J. (2018). “A survey of clustering with deep learning: From the perspective of network architecture”. *IEEE Access*, *6*, 39501–39514.

[157] Müntz, H. et al. (1913). “Solution directe de l’équation séculaire et de quelques problemes analogues transcendants”. *CR Acad. Sci. Paris*, *156*, 43–46.

[158] Murphy, K. P. (2013). “Machine learning : A probabilistic perspective”. MIT Press.

[159] Nagarajan, V., & Kolter, J. Z. (2017). “Gradient descent gan optimization is locally stable”. *Advances in Neural Information Processing Systems*, *30*.

[160] Nair, V., & Hinton, G. E. (2010). “Rectified linear units improve restricted boltzmann machines”. *Proceedings of the 27th International Conference on International Conference on Machine Learning*, 807–814.

[161] Nawi, N. M., Atomi, W. H., & Rehman, M. (2013). “The effect of data preprocessing on optimized training of artificial neural networks” [4th International Conference on Electrical Engineering and Informatics, ICEEI 2013]. *Procedia Technology*, *11*, 32–39.

[162] Nesterov, Y. E. (1983). “A method for solving the convex programming problem with convergence rate o (1/kˆ 2)”. *Dokl. akad. nauk Sssr*, *269*, 543–547.

[163] Neumann, J. v. (1928). “Zur theorie der gesellschaftsspiele”. *Mathematische Annalen*, *100*, 295–320.

[164] Neyshabur, B., Li, Z., Bhojanapalli, S., LeCun, Y., & Srebro, N. (2018). “Towards understanding the role of over-parametrization in generalization of neural networks”. *CoRR*, *abs/1805.12076*.

[165] Neyshabur, B., Tomioka, R., & Srebro, N. (2015). “In search of the real inductive bias: On the role of implicit regularization in deep learning”. *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Workshop Track Proceedings*.

[166] Ng, A. Y. (2004). “Feature selection, l1 vs. l2 regularization, and rotational invariance”. *Proceedings of the Twenty-First International Conference on Machine Learning*, 78.

[167] Nowozin, S., Cseke, B., & Tomioka, R. (2016). “F-gan: Training generative neural samplers using variational divergence minimization”. *Advances in Neural Information Processing Systems*, *29*.

[168] Orabona, F. (2019). “A modern introduction to online learning”. *CoRR*, *abs/1912.13213*.

[169] Orr, G. B. (1996). *Dynamics and algorithms for stochastic search* (Doctoral dissertation) [UMI Order No. GAX96-08998]. USA, Oregon Graduate Institute of Science Technology.

[170] Ortega, J. M., & Rheinboldt, W. C. (1970). “Iterative solution of nonlinear equations in several variables”. Academic Press.

[171] Osborne, M. J., & Rubinstein, A. (1994). “A Course in Game Theory” (Vol. 1). The MIT Press.

[172] Papyan, V. (2018). “The full spectrum of deep net hessians at scale: Dynamics with sample size”. *CoRR*, *abs/1811.07062*.

[173] Parikh, N., & Boyd, S. (2014). “Proximal algorithms”. *Found. Trends Optim.*, *1*(3), 127–239.

[174] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., … Chintala, S. (2019). “Pytorch: An imperative style, high-performance deep learning library”. *Advances in Neural Information Processing Systems*, *32*.

[175] Pearlmutter, B. A. (1994). “Fast exact multiplication by the hessian”. *Neural Comput.*, *6*(1), 147–160.

[176] Pham, H., Dai, Z., Xie, Q., & Le, Q. V. (2021). “Meta pseudo labels”. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 11557–11568.

[177] Pham, H., Guan, M. Y., Zoph, B., Le, Q. V., & Dean, J. (2018). “Efficient neural architecture search via parameter sharing”. *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, July 10-15, 2018*, *80*, 4092–4101.

[178] Pólya, G. (1961). “On the eigenvalues of vibrating membranes†”. *Proceedings of the London Mathematical Society*, *s3-11*(1), 419–433.

[179] Polyak, B. (1964). “Some methods of speeding up the convergence of iteration methods”. *USSR Computational Mathematics and Mathematical Physics*, *4*(5), 1–17.

[180] Pressley, A. (2010). “Gauss’ theorema egregium”. *Elementary differential geometry* (pp. 247–268). Springer London.

[181] Radford, A., Metz, L., & Chintala, S. (2016). “Unsupervised representation learning with deep convolutional generative adversarial networks”. *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*.

[182] Rahm, E. (1994). “Mehrrechner-datenbanksysteme - grundlagen der verteilten und parallelen datenbankverarbeitung”. Addison-Wesley.

[183] Real, E., Aggarwal, A., Huang, Y., & Le, Q. V. (2019). “Regularized evolution for image classifier architecture search”. *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence and Thirty-First Innovative Applications of Artificial Intelligence Conference and Ninth AAAI Symposium on Educational Advances in Artificial Intelligence*.

[184] Rechenberg, I. (1994). “Evolutionsstrategie — optimieren wie in der natur”. *Technik und natur* (pp. 227–244). Springer Berlin Heidelberg.

[185] Reddi, S. J., Kale, S., & Kumar, S. (2018). “On the convergence of adam and beyond”. *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*.

[186] Robbins, H., & Monro, S. (1951). “A Stochastic Approximation Method”. *The Annals of Mathematical Statistics*, *22*(3), 400–407.

[187] Rockafellar, R. T. (1976). “Monotone operators and the proximal point algorithm”. *SIAM Journal on Control and Optimization*, *14*(5), 877–898.

[188] Rokhlin, V., Szlam, A., & Tygert, M. (2009). "A randomized algorithm for principal component analysis". *SIAM J. Matrix Anal. Appl.*, *31*(3), 1100–1124.

[189] Rosenblatt, F. (1958). "The perceptron: A probabilistic model for information storage and organization in the brain." *Psychological review*, *65 6*, 386–408.

[190] Ruder, S. (2016). "An overview of gradient descent optimization algorithms". *CoRR*, *abs/1609.04747*.

[191] Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1985). *Learning internal representations by error propagation* (tech. rep.). California Univ San Diego La Jolla Inst for Cognitive Science.

[192] Ruszczynski, A. (1980). "Feasible direction methods for stochastic programming problems". *Mathematical Programming*, *19*(1), 220–229.

[193] Salimans, T., Goodfellow, I., Zaremba, W., Cheung, V., Radford, A., Chen, X., & Chen, X. (2016). "Improved techniques for training gans". *Advances in Neural Information Processing Systems*, *29*.

[194] Santurkar, S., Tsipras, D., Ilyas, A., & Madry, A. (2018). "How does batch normalization help optimization?" *Advances in Neural Information Processing Systems*, *31*.

[195] Scardapane, S., & Di Lorenzo, P. (2018). "Stochastic training of neural networks via successive convex approximations". *IEEE Transactions on Neural Networks and Learning Systems*, *29*(10), 4947–4956.

[196] Schmidt, W. F., Kraaijveld, M., & Duin, R. P. (1991). "A non-iterative method for training feed forward networks." *International Joint Conference on Neural Networks.*, *2*, 19–24.

[197] Schrödinger, E. (1940). "A method of determining quantum-mechanical eigenvalues and eigenfunctions". *Proceedings of the Royal Irish Academy. Section A: Mathematical and Physical Sciences*, *46*, 9–16.

[198] Scutari, G., & Sun, Y. (2018). "Parallel and distributed successive convex approximation methods for big-data optimization". *Multi-agent optimization: Cetraro, italy 2014* (pp. 141–308). Springer International Publishing.

[199] Shallue, C. J., Lee, J., Antognini, J. M., Sohl-Dickstein, J., Frostig, R., & Dahl, G. E. (2018). "Measuring the effects of data parallelism on neural network training". *CoRR*, *abs/1811.03600*.

[200] Sharma, S., Sharma, S., & Athaiya, A. (2020). "Activation functions in neural networks". *International Journal of Engineering Applied Sciences and Technology*, *04*, 310–316.

[201] Shen, D., Wu, G., & Suk, H.-I. (2017). "Deep learning in medical image analysis" [PMID: 28301734]. *Annual Review of Biomedical Engineering*, *19*(1), 221–248.

[202] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., & Hassabis, D. (2016). "Mastering the game of Go with deep neural networks and tree search". *Nature*, *529*(7587), 484–489.

[203] Simon, H. D. (1984). “Analysis of the symmetric lanczos algorithm with reorthogonalization methods”. *Linear Algebra and its Applications*, *61*, 101–131.

[204] Simonyan, K., & Zisserman, A. (2015). “Very deep convolutional networks for large-scale image recognition”. *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

[205] Smith, S., Elsen, E., & De, S. (2020). “On the generalization benefit of noise in stochastic gradient descent”. *International Conference on Machine Learning*, 9058–9067.

[206] Smith, S. L., Kindermans, P., Ying, C., & Le, Q. V. (2018). “Don’t decay the learning rate, increase the batch size”. *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*.

[207] Sohl-Dickstein, J., Weiss, E. A., Maheswaranathan, N., & Ganguli, S. (2015). “Deep unsupervised learning using nonequilibrium thermodynamics”. *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, 2256–2265.

[208] Solla, S. A., & Le Cun, Y. (1991). “Constrained neural networks for pattern recognition”. *Neural networks: concepts, applications and implementations. Prentice Hall*.

[209] Soltanolkotabi, M., Javanmard, A., & Lee, J. D. (2019). “Theoretical insights into the optimization landscape of over-parameterized shallow neural networks”. *IEEE Transactions on Information Theory*, *65*(2), 742–769.

[210] Song, Y., & Ermon, S. (2019). “Generative modeling by estimating gradients of the data distribution”. *Advances in Neural Information Processing Systems*, *32*.

[211] Sovrasov, V. (2019). *Flops counter for convolutional networks in pytorch framework*.

[212] Spivak, M. (1999). “A comprehensive introduction to differential geometry vol. 4”. Publish or Perish.

[213] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). “Dropout: A simple way to prevent neural networks from overfitting”. *Journal of Machine Learning Research*, *15*(56), 1929–1958.

[214] Staib, M., Reddi, S. J., Kale, S., Kumar, S., & Sra, S. (2019). “Escaping saddle points with adaptive gradient methods”. *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, *97*, 5956–5965.

[215] Stanley, K. O., & Miikkulainen, R. (2002). “Evolving neural networks through augmenting topologies”. *Evolutionary Computation*, *10*(2), 99–127.

[216] Sun, C., Shrivastava, A., Singh, S., & Gupta, A. (2017). “Revisiting unreasonable effectiveness of data in deep learning era”. *2017 IEEE International Conference on Computer Vision (ICCV)*, 843–852.

[217] Sutskever, I., Martens, J., Dahl, G. E., & Hinton, G. E. (2013). “On the importance of initialization and momentum in deep learning”. *Proceedings of the*

30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013, *28*, 1139–1147.

[218] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., & Rabinovich, A. (2015). “Going deeper with convolutions”. *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 1–9.

[219] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., & Wojna, Z. (2016). “Rethinking the inception architecture for computer vision”. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2818–2826.

[220] Teboulle, M. (1992). “Entropic proximal mappings with applications to nonlinear programming”. *Mathematics of Operations Research*, *17*(3), 670–690.

[221] Theis, L., Oord, A. v. d., & Bethge, M. (2016). “A note on the evaluation of generative models”. *International Conference on Learning Representations*.

[222] Thompson, N. C., Greenewald, K. H., Lee, K., & Manso, G. F. (2020). “The computational limits of deep learning”. *CoRR*, *abs/2007.05558*.

[223] Tieleman, T., Hinton, G. et al. (2012). “Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude”. *COURSERA: Neural networks for machine learning*, *4*(2), 26–31.

[224] Touvron, H., Vedaldi, A., Douze, M., & Jegou, H. (2019). “Fixing the train-test resolution discrepancy”. *Advances in Neural Information Processing Systems*, *32*.

[225] Vaserstein, L. N. (1969). “Markov processes over denumerable products of spaces, describing large systems of automata”. *Problemy Peredachi Informatsii*, *5*(3), 64–72.

[226] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). “Attention is all you need”. *Advances in Neural Information Processing Systems*, *30*.

[227] Voulodimos, A., Doulamis, N., Doulamis, A., Protopapadakis, E., & Andina, D. (2018). “Deep learning for computer vision: A brief review”. *Intell. Neuroscience*, *2018*.

[228] Wang, R., Cheng, M., Chen, X., Tang, X., & Hsieh, C.-J. (2021). “Rethinking architecture selection in differentiable NAS”. *International Conference on Learning Representations*.

[229] Wang, X., & Cao, W. (2018). “Non-iterative approaches in training feed-forward neural networks and their applications”. *Soft Comput.*, *22*(11), 3473–3476.

[230] Wei, Y., Liang, X., Chen, Y., Jie, Z., Xiao, Y., Zhao, Y., & Yan, S. (2016). “Learning to segment with image-level annotations”. *Pattern Recognition*, *59*, 234–244.

[231] Weinland, D., Ronfard, R., & Boyer, E. (2011). “A survey of vision-based methods for action representation, segmentation and recognition”. *Computer Vision and Image Understanding*, *115*(2), 224–241.

[232] Wiegerinck, W., Komoda, A., & Heskes, T. (1995). “Stochastic dynamics of learning with momentum in neural networks”. *Journal of Physics A General Physics*, *27*.

[233] Wortsman, M., Farhadi, A., & Rastegari, M. (2019). "Discovering neural wirings". *33rd Conference on Neural Information Processing Systems (NeurIPS 2019)*.

[234] Wu, Y., Liu, A., Huang, Z., Zhang, S., & Van Gool, L. (2021). "Neural architecture search as sparse supernet". *2021 AAAI Conference on Artificial Intelligence*.

[235] Xie, Q., Luong, M.-T., Hovy, E., & Le, Q. V. (2020). "Self-training with noisy student improves imagenet classification". *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 10684–10695.

[236] Xie, S., Zheng, H., Liu, C., & Lin, L. (2019). "SNAS: Stochastic neural architecture search". *International Conference on Learning Representations*.

[237] Xu, H., Van Durme, B., & Murray, K. (2021). "BERT, mBERT, or BiBERT? a study on contextualized embeddings for neural machine translation". *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 6663–6675.

[238] Xu, Y., Xie, L., Zhang, X., Chen, X., Qi, G., Tian, Q., & Xiong, H. (2020). "PC-DARTS: partial channel connections for memory-efficient architecture search". *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*.

[239] Xue, H., Zhang, S., & Cai, D. (2017). "Depth image inpainting: Improving low rank matrix completion with low gradient regularization". *Trans. Img. Proc.*, *26*(9), 4311–4320.

[240] Yang, Y., Scutari, G., Palomar, D. P., & Pesavento, M. (2016). "A parallel decomposition method for nonconvex stochastic multi-agent optimization problems". *IEEE Transactions on Signal Processing*, *64*(11), 2949–2964.

[241] Yang, Y., Hodgkinson, L., Theisen, R., Zou, J., Gonzalez, J. E., Ramchandran, K., & Mahoney, M. W. (2021). "Taxonomizing local versus global structure in neural network loss landscapes". *Advances in Neural Information Processing Systems*, *34*, 18722–18733.

[242] Yang, Y., Li, H., You, S., Wang, F., Qian, C., & Lin, Z. (2020). "Ista-nas: Efficient and consistent neural architecture search by sparse coding". *Advances in Neural Information Processing Systems*, *33*, 10503–10513.

[243] Ying, X. (2019). "An overview of overfitting and its solutions". *Journal of Physics: Conference Series*, *1168*, 022022.

[244] Yuan, M., & Lin, Y. (2006). "Model selection and estimation in regression with grouped variables". *Journal of the Royal Statistical Society. Series B (Methodological)*, *68*(1), 49–67.

[245] Yun, J., Lozano, A. C., & Yang, E. (2020). "A general family of stochastic proximal gradient methods for deep learning". *CoRR*, *abs/2007.07484*.

[246] Zeiler, M. D. (2012). "ADADELTA: an adaptive learning rate method". *CoRR*, *abs/1212.5701*.

[247] Zeiler, M., Ranzato, M., Monga, R., Mao, M., Yang, K., Le, Q., Nguyen, P., Senior, A., Vanhoucke, V., Dean, J., & Hinton, G. (2013). "On rectified linear units for speech processing". *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, 3517–3521.

[248] Zela, A., Elsken, T., Saikia, T., Marrakchi, Y., Brox, T., & Hutter, F. (2020). "Understanding and robustifying differentiable architecture search". *International Conference on Learning Representations*.

[249] Zela, A., Siems, J., & Hutter, F. (2020). "NAS-Bench-1Shot1: Benchmarking and dissecting one-shot neural architecture search". *International Conference on Learning Representations*.

[250] Zhang, C., Bengio, S., Hardt, M., Recht, B., & Vinyals, O. (2021). "Understanding deep learning (still) requires rethinking generalization". *Commun. ACM*, *64*(3), 107–115.

[251] Zhang, X., Huang, Z., Wang, N., Xiang, S., & Pan, C. (2021). "You Only Search Once: Single Shot Neural Architecture Search via Direct Sparse Optimization". *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *43*(9), 2891–2904.

[252] Zhang, X., Xv, C., Shen, M., He, X., & Du, W. (2018/05). "Survey of convolutional neural network". *Proceedings of the 2018 International Conference on Network, Communication, Computer Engineering (NCCE 2018)*, 93–97.

[253] Zhou, Y., Zhang, Y., Wang, Y., & Tian, Q. (2019). "Accelerate CNN via recursive bayesian pruning". *2019 IEEE/CVF International Conference on Computer Vision, ICCV 2019, Seoul, Korea (South), October 27 - November 2, 2019*, 3305–3314.

[254] Zoph, B., & Le, Q. V. (2017). "Neural architecture search with reinforcement learning". *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*.

[255] Zoph, B., Vasudevan, V., Shlens, J., & Le, Q. V. (2018). "Learning transferable architectures for scalable image recognition". *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, 8697–8710.

Appendix A

Proof of Theorem 5.1

Proof The claim $\lim_{k\to\infty}\|\boldsymbol{v}^{(k)}-\nabla f(\boldsymbol{w}^{(k)})\|=0$ is a consequence of [192, Lemma 1]. To see this, one just needs to verify that all the technical conditions therein are satisfied by the problem at hand. Specifically, Condition (a) of [192, Lemma 1] is satisfied because $\mathbb{W}$ is closed and bounded. Condition (b) of [192, Lemma 1] is exactly (5.44). Conditions (c)-(d) of [192, Lemma 1] come from the stepsize rules in (5.45) of Theorem 5.1. Condition (e) of [192, Lemma 1] comes from the Lipschitz property of ∇f and stepsize rule in (5.45) of Theorem 5.1.

The following intermediate result is needed to prove the limit point of the sequence $\boldsymbol{w}^{(k)}$ is a stationary point of (5.29).

Lemma There exists a constant $\widehat{L}$ such that

$$\left\|\widehat{\boldsymbol{w}}(\boldsymbol{w}^{(k_1)},\boldsymbol{\xi}_{k_1})-\widehat{\boldsymbol{w}}(\boldsymbol{w}^{(k_2)},\boldsymbol{\xi}_{k_2})\right\|\le\widehat{L}\left\|\boldsymbol{w}^{(k_1)}-\boldsymbol{w}^{(k_2)}\right\|+e(k_1,k_2),$$

and $\lim_{k_1,k_2\to\infty}e(k_1,k_2)=0$ w.p.1.

Proof Assume without loss of generality (w.l.o.g.) that $\boldsymbol{\tau}_k=\tau\mathbf{1}$, and the approximation subproblem (5.33) reduces to

$$\widehat{\boldsymbol{w}}^{(k)}\triangleq\underset{\boldsymbol{w}\in\mathbb{X}}{\operatorname{argmin}}\left\{(\boldsymbol{w}-\boldsymbol{w}^{(k)})^T\boldsymbol{v}^{(k)}+\frac{\tau}{2}\|\boldsymbol{w}-\boldsymbol{w}^{(k)}\|_2^2+r(\boldsymbol{w})\right\}.$$

It is further equivalent to

$$\min_{\boldsymbol{w}\in\mathbb{X},r(\boldsymbol{w})\le y}\left\{(\boldsymbol{w}-\boldsymbol{w}^{(k)})^T\boldsymbol{v}^{(k)}+\frac{\tau}{2}\|\boldsymbol{w}-\boldsymbol{w}^{(k)}\|_2^2+y\right\},\tag{A.1}$$

where the (unique) optimal $\boldsymbol{w}$ and y is $(\widehat{\boldsymbol{w}}^{(k)}$ and $r(\widehat{\boldsymbol{w}}^{(k)}))$, respectively.

Assume w.l.o.g. that $k_2>k_1$. It follows from first-order optimality condition that

$$(\boldsymbol{w}-\widehat{\boldsymbol{w}}^{(k_1)})^T(\boldsymbol{v}^{(k_1)}+\tau(\widehat{\boldsymbol{w}}^{(k_1)}-\boldsymbol{w}^{(k_1)}))+y-r(\widehat{\boldsymbol{w}}^{(k_1)})\ge0,\forall\boldsymbol{w},y\text{ such that }r(\boldsymbol{w})\le y\tag{A.2a}$$

$$(\boldsymbol{w}-\widehat{\boldsymbol{w}}^{(k_2)})^T(\boldsymbol{v}^{(k_2)}+\tau(\widehat{\boldsymbol{w}}^{(k_2)}-\boldsymbol{w}^{(k_2)}))+y-r(\widehat{\boldsymbol{w}}^{(k_2)})\ge0,\forall\boldsymbol{w},y\text{ such that }r(\boldsymbol{w})\le y.\tag{A.2b}$$

Setting $(\boldsymbol{w},y)=(\widehat{\boldsymbol{w}}^{(k_2)},r(\widehat{\boldsymbol{w}}^{(k_2)}))$ in (A.2a) and $(\boldsymbol{w},y)=(\widehat{\boldsymbol{w}}^{(k_1)},r(\widehat{\boldsymbol{w}}^{(k_1)}))$ in (A.2b), and adding them up, one obtains

$$(\widehat{\boldsymbol{w}}^{(k_1)}-\widehat{\boldsymbol{w}}^{(k_2)})^T(\boldsymbol{v}^{(k_1)}-\boldsymbol{v}^{(k_2)})-\tau(\boldsymbol{w}^{(k_1)}-\boldsymbol{w}^{(k_2)})^T(\widehat{\boldsymbol{w}}^{(k_1)}-\widehat{\boldsymbol{w}}^{(k_2)})\leq-\tau\|\widehat{\boldsymbol{w}}^{(k_1)}-\widehat{\boldsymbol{w}}^{(k_2)}\|_2^2. \tag{A.3}$$

The term on the left hand side can be lower bounded as follows:

$$\begin{aligned}
&\big\langle\widehat{\boldsymbol{w}}^{(k_1)}-\widehat{\boldsymbol{w}}^{(k_2)},\boldsymbol{v}^{(k_1)}-\nabla f(\boldsymbol{w}^{(k_1)})-\boldsymbol{v}^{(k_2)}+\nabla f(\boldsymbol{w}^{(k_2)})\big\rangle\\
&\quad+\big\langle\widehat{\boldsymbol{w}}^{(k_1)}-\widehat{\boldsymbol{w}}^{(k_2)},\nabla f(\boldsymbol{w}^{(k_1)})-\nabla f(\boldsymbol{w}^{(k_2)})\big\rangle-\tau\big\langle\widehat{\boldsymbol{w}}^{(k_1)}-\widehat{\boldsymbol{w}}^{(k_2)},\boldsymbol{w}^{(k_1)}-\boldsymbol{w}^{(k_2)}\big\rangle\\
&\geq-\big\|\widehat{\boldsymbol{w}}^{(k_1)}-\widehat{\boldsymbol{w}}^{(k_2)}\big\|(\varepsilon_{k_1}+\varepsilon_{k_2})-(L+\tau)\big\|\widehat{\boldsymbol{w}}^{(k_1)}-\widehat{\boldsymbol{w}}^{(k_2)}\big\|\big\|\boldsymbol{w}^{(k_1)}-\boldsymbol{w}^{(k_2)}\big\| \qquad (A.4)
\end{aligned}$$

where the inequality comes from the Lipschitz continuity of $\nabla f(\boldsymbol{w})$, with $\varepsilon_k\triangleq\big\|\boldsymbol{v}^{(k)}-\nabla f(\boldsymbol{w}^{(k)})\big\|$.

Combining the inequalities (A.3) and (A.4)

$$\big\|\widehat{\boldsymbol{w}}^{(k_1)}-\widehat{\boldsymbol{w}}^{(k_2)}\big\|\leq(L+\tau)\tau^{-1}\big\|\boldsymbol{w}^{(k_1)}-\boldsymbol{w}^{(k_2)}\big\|+\tau^{-1}(\varepsilon_{k_1}+\varepsilon_{k_2}),$$

which leads to the desired (asymptotic) Lipschitz property:

$$\big\|\widehat{\boldsymbol{w}}^{(k_1)}-\widehat{\boldsymbol{w}}^{(k_2)}\big\|\leq\widehat{L}\big\|\boldsymbol{w}^{(k_1)}-\boldsymbol{w}^{(k_2)}\big\|+e(k_1,k_2),$$

with $\widehat{L}\triangleq\tau^{-1}(L+\tau)$ and $e(k_1,k_2)\triangleq\tau^{-1}(\varepsilon_{k_1}+\varepsilon_{k_2})$, and $\lim_{k_1\to\infty,k_2\to\infty}e(k_1,k_2)=0$ w.p.1.

Define $U(\boldsymbol{w})\triangleq f(\boldsymbol{w})+r(\boldsymbol{w})$. Following the line of analysis from (5.41) to (5.42), one obtains

$$\begin{aligned}
&U(\boldsymbol{w}^{(k+1)})-U(\boldsymbol{w}^{(k)}) \qquad (A.5)\\
&\leq\varepsilon_k((\widehat{\boldsymbol{w}}^{(k)}-\boldsymbol{w}^{(k)})^T(\nabla f(\boldsymbol{w}^{(k)})+r(\widehat{\boldsymbol{w}}^{(k)})-r(\boldsymbol{w}^{(k)}))+\frac{L}{2}\varepsilon_k^2\big\|\widehat{\boldsymbol{w}}^{(k)}-\boldsymbol{w}^{(k)}\big\|^2\\
&=\varepsilon_k(\widehat{\boldsymbol{w}}^{(k)}-\boldsymbol{w}^{(k)})^T(\nabla f(\boldsymbol{w}^{(k)})-\boldsymbol{v}^{(k)}+\boldsymbol{v}^{(k)}+r(\widehat{\boldsymbol{w}}^{(k)})-r(\boldsymbol{w}^{(k)}))+\frac{L}{2}\varepsilon_k^2\big\|\widehat{\boldsymbol{w}}^{(k)}-\boldsymbol{w}^{(k)}\big\|^2\\
&\leq-\varepsilon_k\left(\tau-\frac{L}{2}\varepsilon_k\right)\big\|\widehat{\boldsymbol{w}}^{(k)}-\boldsymbol{w}^{(k)}\big\|^2+\varepsilon_k\big\|\widehat{\boldsymbol{w}}^{(k)}-\boldsymbol{w}^{(k)}\big\|\big\|\nabla f(\boldsymbol{w}^{(k)})-\boldsymbol{v}^{(k)}\big\|, \qquad (A.6)
\end{aligned}$$

where in the last inequality (5.40) was used together with the Cauchy-Schwarz inequality.

Next, it is shown by contradiction that $\liminf_{k\to\infty}\big\|\widehat{\boldsymbol{w}}^{(k)}-\boldsymbol{w}^{(k)}\big\|=0$ w.p.1. Suppose $\liminf_{k\to\infty}\big\|\widehat{\boldsymbol{w}}^{(k)}-\boldsymbol{w}^{(k)}\big\|\geq\chi>0$ with a positive probability. Then one can find a realization such that at the same time $\big\|\widehat{\boldsymbol{w}}^{(k)}-\boldsymbol{w}^{(k)}\big\|\geq\chi>0$ for all k and $\lim_{k\to\infty}\big\|\nabla f(\boldsymbol{w}^{(k)})-\boldsymbol{v}^{(k)}\big\|=0$; focus next on such a realization. Using $\big\|\widehat{\boldsymbol{w}}^{(k)}-\boldsymbol{w}^{(k)}\big\|\geq\chi>0$, the inequality (A.6) is equivalent to

$$U(\boldsymbol{w}^{(k+1)})-U(\boldsymbol{w}^{(k)})\leq-\varepsilon_k\left(\tau-\frac{L}{2}\varepsilon_k-\frac{1}{\chi}\left\|\nabla f(\boldsymbol{w}^{(k)})-\boldsymbol{v}^{(k)}\right\|\right)\big\|\widehat{\boldsymbol{w}}^{(k)}-\boldsymbol{w}^{(k)}\big\|^2. \tag{A.7}$$

Since $\lim_{k\to\infty}\|\nabla f(\boldsymbol{w}^{(k)})-\boldsymbol{v}^{(k)}\|=0$, there exists a k_0 sufficiently large such that

$$\tau-\frac{L}{2}\varepsilon_k-\frac{1}{\chi}\left\|\nabla f(\boldsymbol{w}^{(k)})-\boldsymbol{v}^{(k)}\right\|\geq\bar{\tau}>0,\quad\forall k\geq k_0. \tag{A.8}$$

Therefore, it follows from (A.7) and (A.8) that

$$U(\boldsymbol{w}^{(k)})-U(\boldsymbol{w}_{k_0})\leq-\bar{\tau}\chi^2\textstyle\sum_{n=k_0}^{k}\varepsilon^{n+1}, \tag{A.9}$$

which, in view of $\sum_{n=k_0}^{\infty}\varepsilon^{n+1}=\infty$, contradicts the boundedness of $\{U(\boldsymbol{w}^{(k)})\}$. Therefore it must be $\liminf_{k\to\infty}\|\widehat{\boldsymbol{w}}^{(k)}-\boldsymbol{w}^{(k)}\|=0$ w.p.1.

Let us show by contradiction that $\limsup_{k\to\infty}\|\widehat{\boldsymbol{w}}^{(k)}-\boldsymbol{w}^{(k)}\|=0$ w.p.1. Suppose $\limsup_{k\to\infty}\|\widehat{\boldsymbol{w}}^{(k)}-\boldsymbol{w}^{(k)}\|>0$ with a positive probability. Next, focus on a realization along with $\limsup_{k\to\infty}\|\widehat{\boldsymbol{w}}^{(k)}-\boldsymbol{w}^{(k)}\|>0$, $\lim_{k\to\infty}\|\nabla f(\boldsymbol{w}^{(k)})-\boldsymbol{v}^{(k)}\|=0$, $\liminf_{k\to\infty}\|\widehat{\boldsymbol{w}}^{(k)}-\boldsymbol{w}^{(k)}\|=0$, and $\lim_{k_1,k_2\to\infty}e(k_1,k_2)=0$, where $e(k_1,k_2)$ is defined in Lemma 1. It follows from $\limsup_{k\to\infty}\|\widehat{\boldsymbol{w}}^{(k)}-\boldsymbol{w}^{(k)}\|>0$ and $\liminf_{k\to\infty}\|\widehat{\boldsymbol{w}}^{(k)}-\boldsymbol{w}^{(k)}\|=0$ that there exists a $\delta>0$ such that $\|\triangle\boldsymbol{w}^{(k)}\|\geq2\delta$ (with $\triangle\boldsymbol{w}^{(k)}\triangleq\widehat{\boldsymbol{w}}^{(k)}-\boldsymbol{w}^{(k)}$) for infinitely many k and also $\|\triangle\boldsymbol{w}^{(k)}\|<\delta$ for infinitely many k. Therefore, one can always find an infinite set of indexes, say $\mathcal{T}$, having the following properties: for any $k\in\mathcal{T}$, there exists an integer $i_k>k$ such that

$$\left\|\triangle\boldsymbol{w}^{(k)}\right\|<\delta,\quad\left\|\triangle\boldsymbol{w}^{(i_k)}\right\|>2\delta,\quad\delta\leq\left\|\triangle\boldsymbol{w}^{(n)}\right\|\leq2\delta,k<n<i_k. \tag{A.10}$$

Given the above bounds, the following holds: for all $k\in\mathcal{T}$,

$$\begin{aligned}\delta&\leq\left\|\triangle\boldsymbol{w}^{(i_k)}\right\|-\left\|\triangle\boldsymbol{w}^{(k)}\right\|\\&\leq\left\|\triangle\boldsymbol{w}^{(i_k)}-\triangle\boldsymbol{w}^{(k)}\right\|=\left\|(\widehat{\boldsymbol{w}}^{(i_k)}-\boldsymbol{w}^{(i_k)})-(\widehat{\boldsymbol{w}}^{(k)}-\boldsymbol{w}^{(k)})\right\|\\&\leq\|\widehat{\boldsymbol{w}}^{(i_k)}-\widehat{\boldsymbol{w}}^{(k)}\|+\|\boldsymbol{w}^{(i_k)}-\boldsymbol{w}^{(k)}\|\\&\leq(1+\widehat{L})\|\boldsymbol{w}^{(i_k)}-\boldsymbol{w}^{(k)}\|+e(i_k,k)\\&\leq(1+\widehat{L})\textstyle\sum_{n=k}^{i_k-1}\varepsilon_n\left\|\triangle\boldsymbol{w}^{(n)}\right\|+e(i_k,k)\\&\leq2\delta(1+\widehat{L})\textstyle\sum_{n=k}^{i_k-1}\varepsilon_n+e(i_k,k),\end{aligned} \tag{A.11}$$

implying that

$$\liminf_{\mathcal{T}\ni k\to\infty}\textstyle\sum_{n=k}^{i_k-1}\varepsilon_n\geq\bar{\delta}_1\triangleq\dfrac{1}{2(1+\widehat{L})}>0. \tag{A.12}$$

Proceeding as in (A.11): for all $k\in\mathcal{T}$,

$$\left\|\triangle\boldsymbol{w}^{(k+1)}\right\|-\left\|\triangle\boldsymbol{w}^{(k)}\right\|\leq\left\|\triangle\boldsymbol{w}^{(k+1)}-\triangle\boldsymbol{w}^{(k)}\right\|\leq(1+\widehat{L})\varepsilon_k\left\|\triangle\boldsymbol{w}^{(k)}\right\|+e(k,k+1),$$

which leads to

$$(1+(1+\widehat{L})\varepsilon_k)\left\|\triangle\boldsymbol{w}^{(k)}\right\|+e(k,k+1)\geq\left\|\triangle\boldsymbol{w}^{(k+1)}\right\|\geq\delta, \tag{A.13}$$

where the second inequality follows from (A.10). It follows from (A.13) that there exists a $\bar{\delta}_2 > 0$ such that for sufficiently large $k \in \mathcal{T}$,

$$\left\| \triangle \boldsymbol{w}^{(k)} \right\| \geq \frac{\delta - e(k,k+1)}{1 + (1+\widehat{L})\varepsilon_k} \geq \bar{\delta}_2 > 0. \tag{A.14}$$

Here, after assuming w.l.o.g. that (A.14) holds for all $k \in \mathcal{T}$ (in fact one can always restrict $\{\boldsymbol{w}^{(k)}\}_{k\in\mathcal{T}}$ to a proper subsequence).

The next step is showing that (A.12) is in contradiction with the convergence of $\{U(\boldsymbol{w}^{(k)})\}$. Invoking (A.6), for all $k \in \mathcal{T}$,

$$\begin{aligned} U(\boldsymbol{w}^{(k+1)}) - U(\boldsymbol{w}^{(k)}) &\leq -\varepsilon_k \left(\tau - \frac{L}{2}\varepsilon_k\right) \left\|\widehat{\boldsymbol{w}}^{(k)} - \boldsymbol{w}^{(k)}\right\|^2 + \varepsilon_k \delta \left\|\nabla f(\boldsymbol{w}^{(k)}) - \boldsymbol{v}^{(k)}\right\| \\ &\leq -\varepsilon_k \left(\tau - \frac{L}{2}\varepsilon_k - \frac{\left\|\nabla f(\boldsymbol{w}^{(k)}) - \boldsymbol{v}^{(k)}\right\|}{\delta}\right) \left\|\widehat{\boldsymbol{w}}^{(k)} - \boldsymbol{w}^{(k)}\right\|^2 \\ &\quad + \varepsilon_k \delta \left\|\nabla f(\boldsymbol{w}^{(k)}) - \boldsymbol{v}^{(k)}\right\|^2, \end{aligned} \tag{A.15}$$

and for $k < n < i_k$,

$$\begin{aligned} U(\boldsymbol{w}^{(n+1)}) - U(\boldsymbol{w}^{(n)}) &\leq -\varepsilon_n \left(\tau - \frac{L}{2}\varepsilon_n - \frac{\left\|\nabla f(\boldsymbol{w}^{(n)}) - \boldsymbol{v}^{(n)}\right\|}{\left\|\widehat{\boldsymbol{w}}^{(n)} - \boldsymbol{w}^{(n)}\right\|}\right) \left\|\widehat{\boldsymbol{w}}^{(n)} - \boldsymbol{w}^{(n)}\right\|^2 \\ &\leq -\varepsilon_n \left(\tau - \frac{L}{2}\varepsilon_n - \frac{\left\|\nabla f(\boldsymbol{w}^{(n)}) - \boldsymbol{v}^{(n)}\right\|}{\delta}\right) \left\|\widehat{\boldsymbol{w}}^{(n)} - \boldsymbol{w}^{(n)}\right\|^2, \end{aligned} \tag{A.16}$$

where the last inequality follows from (A.10). Adding (A.15) and (A.16) over $n = k+1,\dots,i_k-1$ and, for $k \in \mathcal{T}$ sufficiently large (so that $\tau - L\varepsilon_k/2 - \delta^{-1}\left\|\nabla f(\boldsymbol{w}^{(n)}) - \boldsymbol{v}^{(n)}\right\| \geq \widehat{\tau} > 0$ and $\left\|\nabla f(\boldsymbol{w}^{(k)}) - \boldsymbol{v}^{(k)}\right\| < \widehat{\tau}\bar{\delta}_2^2/\delta$), this results in

$$\begin{aligned} U(\boldsymbol{w}^{(i_k)}) - U(\boldsymbol{w}^{(k)}) &\overset{(a)}{\leq} -\widehat{\tau}\textstyle\sum_{n=k}^{i_k-1}\varepsilon_n \left\|\widehat{\boldsymbol{w}}^{(n)} - \boldsymbol{w}^{(n)}\right\|^2 + \varepsilon_k\delta\left\|\nabla f(\boldsymbol{w}^{(k)}) - \boldsymbol{v}^{(k)}\right\| \\ &\overset{(b)}{\leq} -\widehat{\tau}\bar{\delta}_2^2\textstyle\sum_{n=k+1}^{i_k-1}\varepsilon_n - \varepsilon_k\left(\widehat{\tau}\bar{\delta}_2^2 - \delta\left\|\nabla f(\boldsymbol{w}^{(k)}) - \boldsymbol{v}^{(k)}\right\|\right) \\ &\overset{(c)}{\leq} -\widehat{\tau}\bar{\delta}_2^2\textstyle\sum_{n=k+1}^{i_k-1}\varepsilon_n, \end{aligned} \tag{A.17}$$

where (a) follows from $\tau - L\varepsilon_k/2 - \delta^{-1}\left\|\nabla f(\boldsymbol{w}^{(n)}) - \boldsymbol{v}^{(n)}\right\| \geq \widehat{\tau} > 0$; (b) is due to (A.14); and in (c) $\left\|\nabla f(\boldsymbol{w}^{(k)}) - \boldsymbol{v}^{(k)}\right\| < \widehat{\tau}\bar{\delta}_2^2/\delta$ was used. Since $\{U(\boldsymbol{w}^{(k)})\}$ converges, it must be $\liminf_{\mathcal{T}\ni k\to\infty} \sum_{n=k+1}^{i_k-1}\varepsilon_n = 0$, which contradicts (A.12). Therefore, it must be $\limsup_{k\to\infty}\left\|\widehat{\boldsymbol{w}}^{(k)} - \boldsymbol{w}^{(k)}\right\| = 0$ w.p.1.

Finally, a proof is given that every limit point of the sequence $\left\{\boldsymbol{w}^{(k)}\right\}$ is a stationary solution of (5.29). Let $\boldsymbol{w}^\star$ be the limit point of the convergent subsequence $\left\{\boldsymbol{w}^{(k)}\right\}_{k\in\mathcal{T}}$. Taking the limit of (A.2a) over the index set $\mathcal{T}$ (and replacing w.l.o.g. y by $r(\boldsymbol{w})$)

$$\begin{aligned}&\lim_{\mathcal{T}\ni k\to\infty}(\boldsymbol{w}-\widehat{\boldsymbol{w}}^{(k)})^T(\boldsymbol{v}^{(k)}+\tau\left(\widehat{\boldsymbol{w}}^{(k)}-\boldsymbol{w}^{(k)}\right))+r(\boldsymbol{w})-r(\widehat{\boldsymbol{w}}^{(k)})\\&=(\boldsymbol{w}-\boldsymbol{w}^\star)^T\nabla f(\boldsymbol{w}^\star)+r(\boldsymbol{w})-r(\boldsymbol{w}^\star)\geq 0,\ \forall\boldsymbol{w}\in\mathbb{X},\end{aligned}$$

where the last equality follows from: i) $\lim_{k\to\infty}\left\|\nabla f(\boldsymbol{w}^{(k)})-\boldsymbol{v}^{(k)}\right\|=0$, and ii) $\lim_{k\to\infty}\left\|\widehat{\boldsymbol{w}}^{(k)}-\boldsymbol{w}^{(k)}\right\|=0$. This is the desired first-order optimality condition and $\boldsymbol{w}^\star$ is a stationary point of (5.29).

Appendix B

ResNet-50 Structure after Pruning

In this section the resulting network structure of ResNet-50 after filter pruning is summarized. The structure of the unpruned ResNet-50 network is shown in Table B.1. It consists of 4 layers, where each layer contains a certain number of blocks. Each block contains three different convolutions. In the experiments from Section 6.3, the ResNet-50 model is trained on ImageNet-2012 using GSparsity. Table B.2 depicts the resulting pruned networks, which have been trained using GSparsity with $\mu \in \{0.02, 0.05, 0.07, 0.10\}$. Note that the filter height and width are removed from each cell in Table B.2 in order to save space.

Layers	Blocks	Conv1	Conv2	Conv3
Layer 1	Block 1	$1\times1,64$	$3\times3,64$	$1\times1,256$
	Block 2	$1\times1,64$	$3\times3,64$	$1\times1,256$
	Block 3	$1\times1,64$	$3\times3,64$	$1\times1,256$
Layer 2	Block 1	$1\times1,128$	$3\times3,128$	$1\times1,512$
	Block 2	$1\times1,128$	$3\times3,128$	$1\times1,512$
	Block 3	$1\times1,128$	$3\times3,128$	$1\times1,512$
	Block 4	$1\times1,128$	$3\times3,128$	$1\times1,512$
Layer 3	Block 1	$1\times1,256$	$3\times3,256$	$1\times1,1024$
	Block 2	$1\times1,256$	$3\times3,256$	$1\times1,1024$
	Block 3	$1\times1,256$	$3\times3,256$	$1\times1,1024$
	Block 4	$1\times1,256$	$3\times3,256$	$1\times1,1024$
	Block 5	$1\times1,256$	$3\times3,256$	$1\times1,1024$
	Block 6	$1\times1,256$	$3\times3,256$	$1\times1,1024$
Layer 4	Block 1	$1\times1,512$	$3\times3,512$	$1\times1,2048$
	Block 2	$1\times1,512$	$3\times3,512$	$1\times1,2048$
	Block 3	$1\times1,512$	$3\times3,512$	$1\times1,2048$

Table B.1.: *The structure of the unpruned ResNet-50 network that is used for training on ImageNet-2012. Each cell contains the filter height × filter height as well as the number of output channels.*

		$\mu = 0.02$			$\mu = 0.05$			$\mu = 0.07$			$\mu = 0.10$		
Layers	Blocks	Conv1	Conv2	Conv3	Conv1	Conv2	Conv3	Conv1	Conv2	Conv3	Conv1	Conv2	Conv3
Layer 1	Block 1	14	32	256	7	33	256	7	32	256	5	31	256
	Block 2	54	63	256	44	56	256	43	57	256	36	50	256
	Block 3	56	64	256	44	64	256	40	64	256	33	63	256
Layer 2	Block 1	72	127	512	42	110	512	32	102	512	24	88	512
	Block 2	14	51	512	9	45	512	8	50	512	6	40	512
	Block 3	60	93	512	34	72	512	32	58	512	30	54	512
	Block 4	97	124	512	59	121	512	51	118	512	41	116	512
Layer 3	Block 1	221	253	1024	161	238	1024	138	236	1024	124	222	1024
	Block 2	81	160	1024	51	130	1024	42	118	1024	39	105	1024
	Block 3	101	202	1024	64	176	1024	57	170	1024	52	161	1024
	Block 4	108	102	1024	61	160	1024	52	160	1024	40	143	1024
	Block 5	91	158	1024	54	130	1024	42	117	1024	34	100	1024
	Block 6	137	204	1024	92	178	1024	80	160	1024	72	150	1024
Layer 4	Block 1	512	512	2048	512	512	2048	500	512	2048	468	512	2048
	Block 2	461	510	2048	224	491	2048	172	464	2048	133	429	2048
	Block 3	44	70	2048	11	5	2048	10	4	2048	9	3	2048

Table B.2.: *Final structure of ResNet-50 after training with GSparsity on the ImageNet-2012 dataset. This Table depicts the resulting output channels of the different convolutions for four different values of* μ. *Please refer to Table B.1 for the structure of the unpruned network. The filter height and width have been omitted in this Table to save space.*

Appendix C

CIFAR-10 and CIFAR-100 on DARTS Search Space

Accuracy of the best performing architecture As previously mentioned, this thesis deviates from the commonly used practice of only reporting the average accuracy of the model with the best performance. In Table C.1 the results obtained from Table 6.4 are used to find the best performing architecture for each method on CIFAR-10 and CIFAR-100. On CIFAR-10, observe that PC-DARTS is able to reach the highest accuracy with 97.38 ± 0.08, followed closely by P-DARTS with 97.25 ± 0.07 and the proposed method GSparsity with 97.24 ± 0.03. On CIFAR-100, the proposed method is able to find the best performing architecture, which reaches an average accuracy of 84.04 ± 0.28, followed by P-DARTS with 83.62 ± 0.19.

	CIFAR-10 Accuracy	CIFAR-100 Accuracy
DARTS (2nd)	97.09 ± 0.09	81.05 ± 0.23
P-DARTS	97.25 ± 0.07	83.62 ± 0.19
PC-DARTS	97.38 ± 0.08	83.38 ± 0.20
DrNAS	96.98 ± 0.07	83.41 ± 0.18
GDAS	96.77 ± 0.11	81.41 ± 0.40
ISTA-NAS	96.86 ± 0.02	83.17 ± 0.17
GAEA	96.57 ± 0.04	80.34 ± 0.05
GSparsity (prop.)	97.24 ± 0.03	84.04 ± 0.28

Table C.1.: *Accuracy of the best architecture found by different NAS methods for the DARTS space. Each method has been run three times, and each of those three architectures that have been found have been evaluated three times. This table summarizes the performance of the best architecture out of the three that have been search by each method, contrary to Table 6.4, which shows the average accuracy of all three architectures.*

Pruning weights vs. pruning switches This paragraph investigates the approach where scaling factors (which act as a switch) are appended to the operations and then pruned (instead of weights). It turns out that the magnitudes of the scaling factors are very sensitive to the value of μ and they are either all active or all zero. For example,

- when $\mu = 3.66$, all scaling factors are active.
- when $\mu = 3.67$, all scaling factors are zero.
- when $\mu = 3.69$, all scaling factors are active.

Therefore, it is more beneficial to directly prune the weights, which is not as sensitive to changes in the value of μ. See Appendix E for an ablation study where weights are directly pruned.

Appendix D

Convergence of ProxSGD vs. SGD

This section compares the convergence of ProxSGD and SGD with $\ell_{2,1}$-regularization in the NAS setting. For this experiment the weight decay is set to a fixed value of $\mu = 50$ and a network architecture is searched using the proposed GSparsity method. For SGD the best performing hyperparameters are chosen after a variety of learning rates are searched. The momentum stays fixed at $\rho = 0.9$. The results are summarized in Figure D.1.

Observe that GSparsity outperforms SGD with a lower training objective and a higher validation accuracy. The biggest difference is observed in the cell architecture though, which is depicted in Figures D.2 and D.3 for SGD and Figure D.4 for ProxSGD. One can see that while GSparsity is able to converge to a sparse solution, with 44 non-zero operations in the normal cell and 11 operations in the reduction cell, SGD does not converge to a sparse solution. There are 96 remaining operations in the normal cell and 95 remaining operations in the reduction cell.

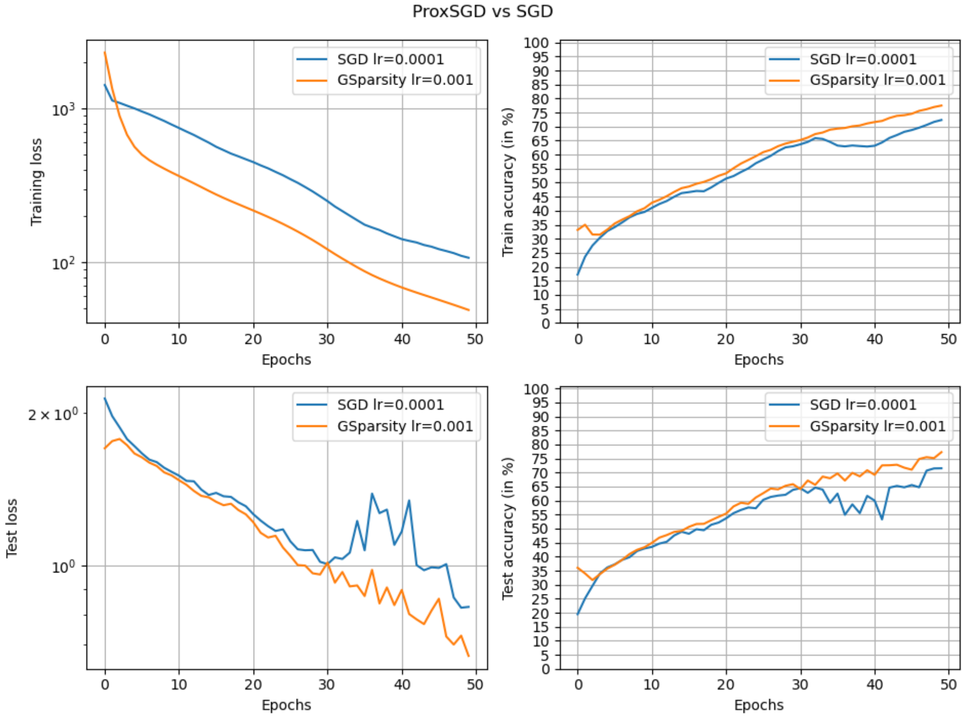


Figure D.1.: *Neural architecture search using SGD and ProxSGD (denoted by GSparsity).*

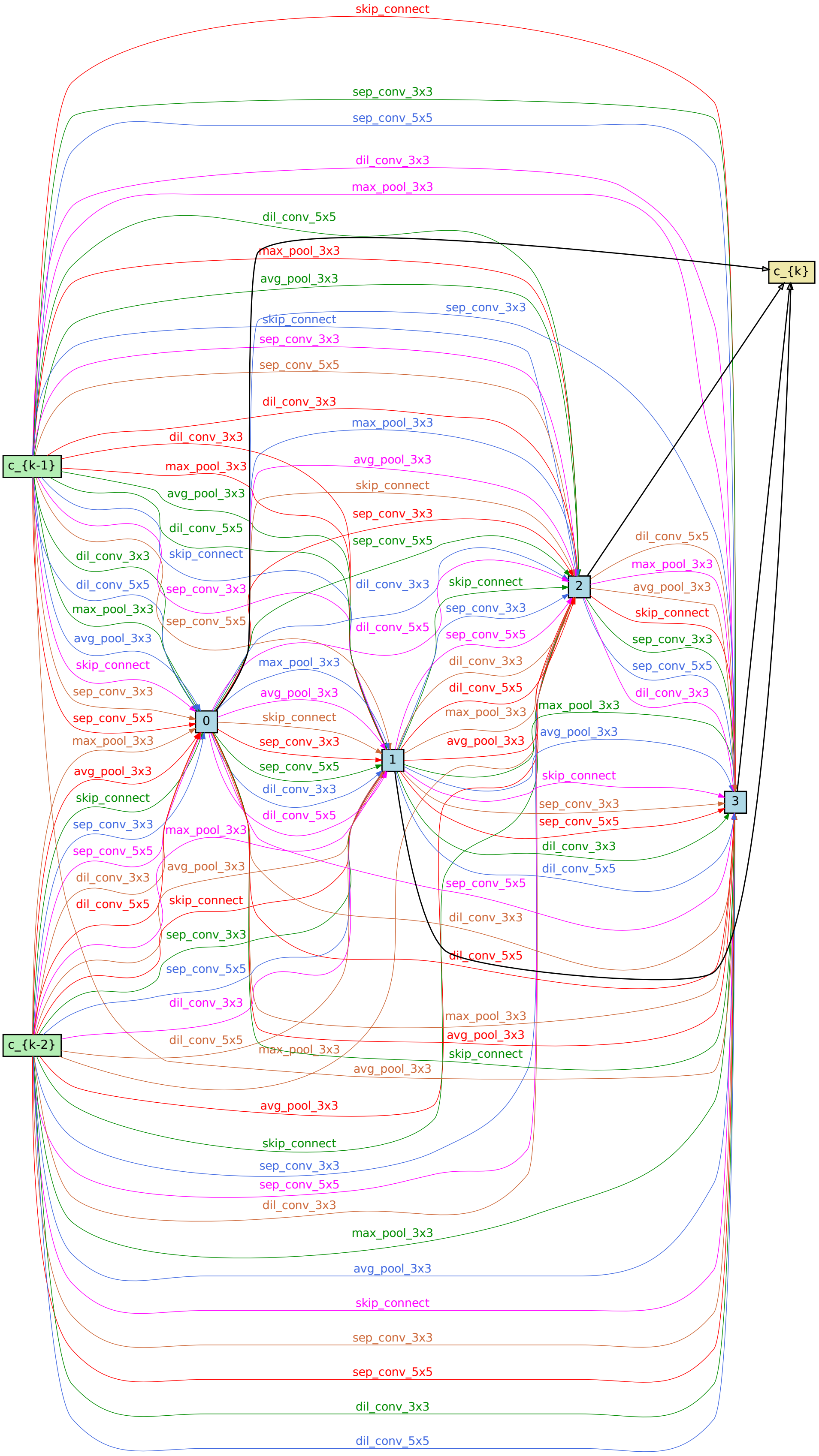


Figure D.2.: *Normal cell for $\mu = 50$ trained with SGD. There are 96 non-zero operations in this cell after pruning.*

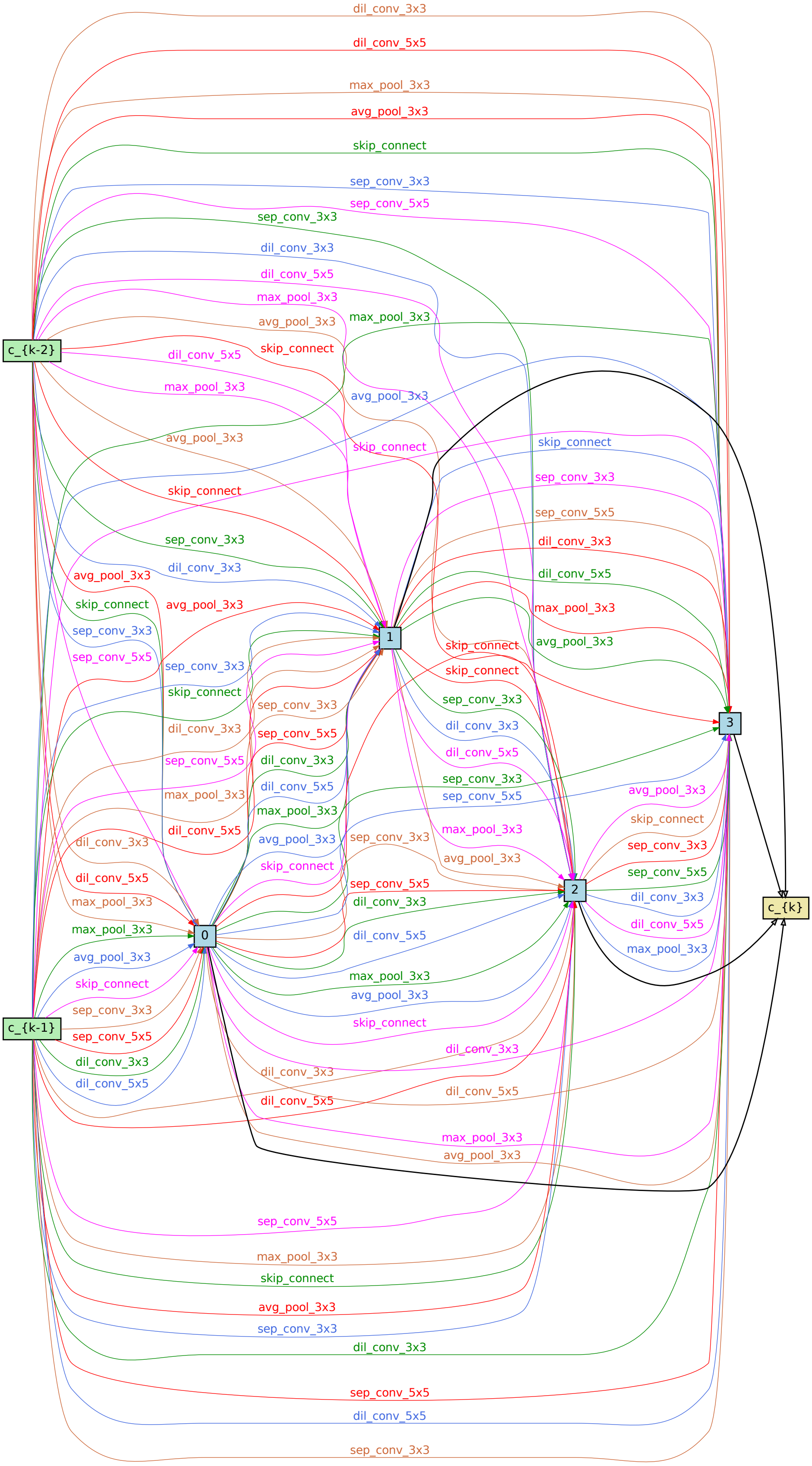
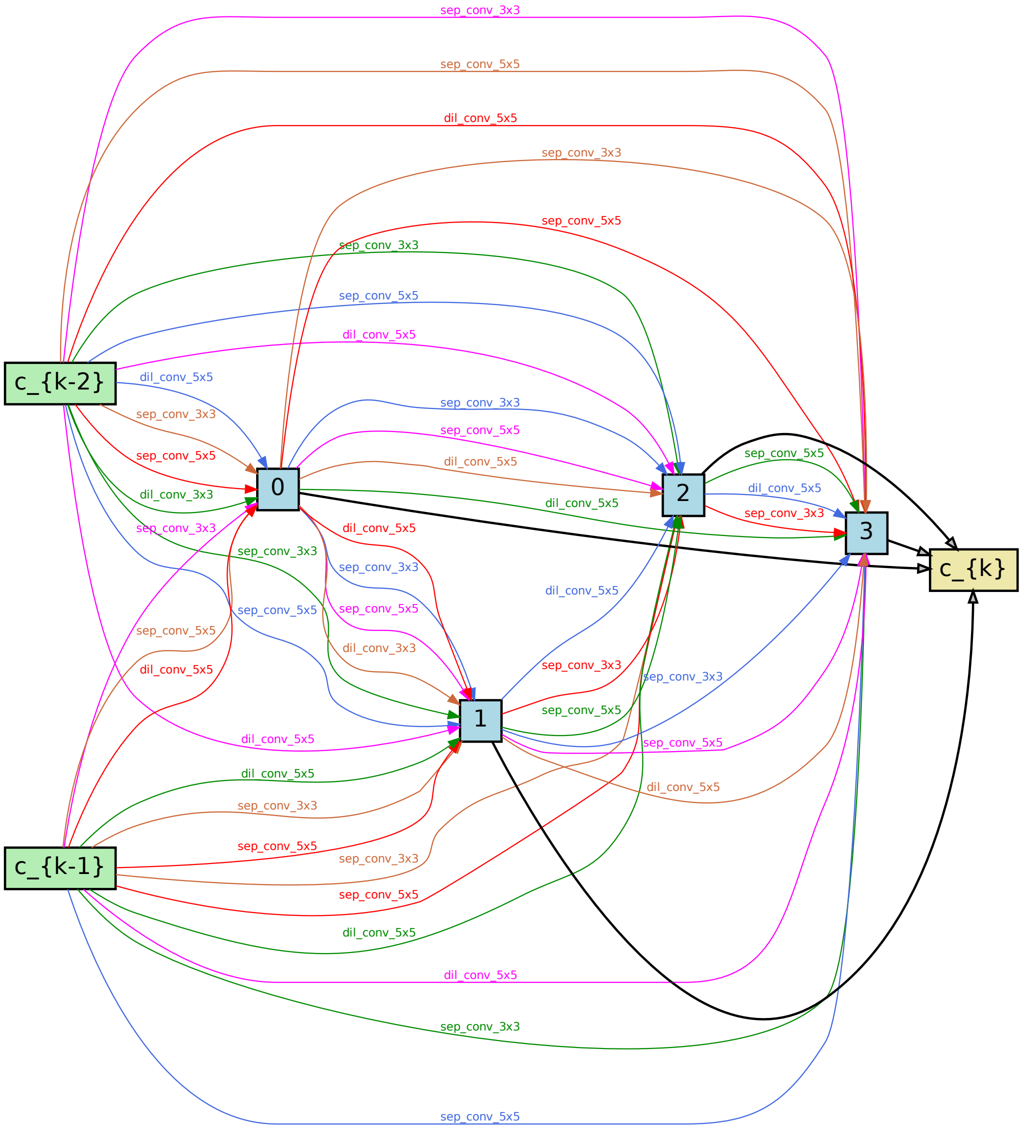
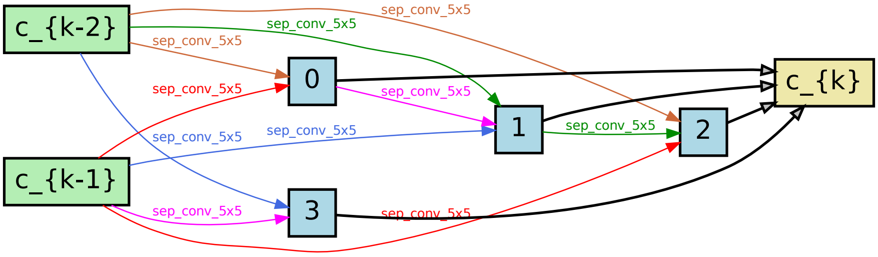


Figure D.3.: *Reduction cell for $\mu = 50$ trained with SGD. There are 95 non-zero operations in this cell after pruning.*



(a) *Normal cell with 44 non-zero operations.*



(b) *Reduction cell with 11 non-zero operations.*

Figure D.4.: *Normal and reduction cell for $\mu = 50$ trained with ProxSGD.*

Appendix E

GSparsity and NAS: Ablation Study

In order to see the effect of the regularization parameter μ on the structure of the final network, the found architecture for $\mu \in [0.1, 1, 10, 50, 100, 200, 500, 1000]$ is depicted in Figures (E.1)-(E.11). During training of each model with GSparsity, the learning rate has been kept at a fixed value of $\varepsilon = 0.001$. One can observe that the number of non-zero operations decreases monotonically with the value of μ. For $\mu = 0.1$, the number of non-zero operations in the normal cell is 98, and in the reduction cell there are 64 operations. On the other hand, for a very large value of μ the network prunes all of the operations in both cells, as can be seen in Figure E.11 for $\mu = 1000$. For small values of μ, the resulting number of operations is more sensitive for changes in its value. For example, going from $\mu = 0.1$ to $\mu = 1$, a change of only $\Delta\mu = 0.9$, the total number of non-zero operations is reduced by 42. But going from $\mu = 100$ to $\mu = 200$ only reduces the number of non-zero operations by 10.

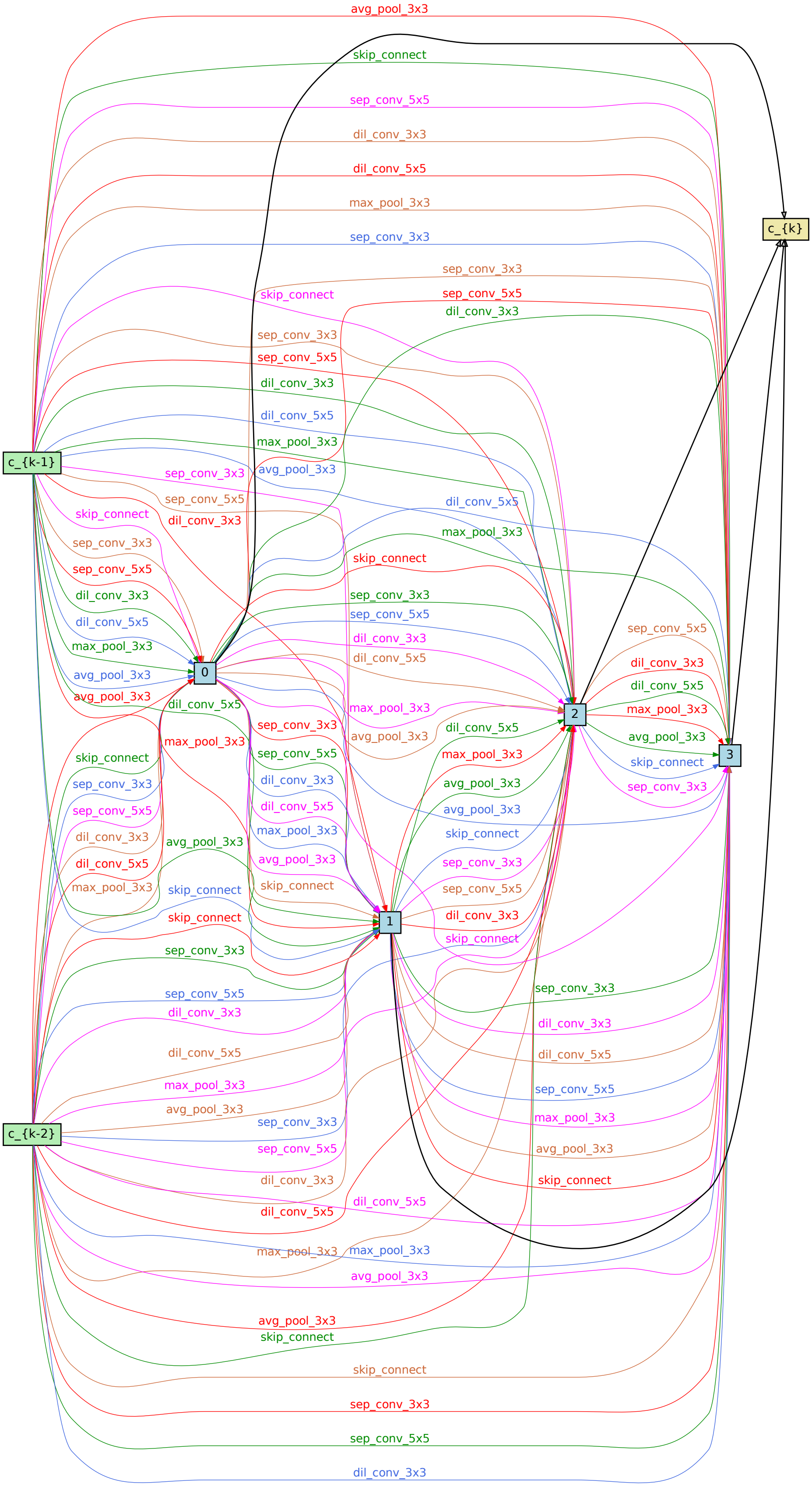


Figure E.1.: *Resulting normal cell after training with GSparsity with* $\mu = 0.1$ *and fixed* $\varepsilon = 0.001$. *All 98 operations are still present in this cell after pruning.*

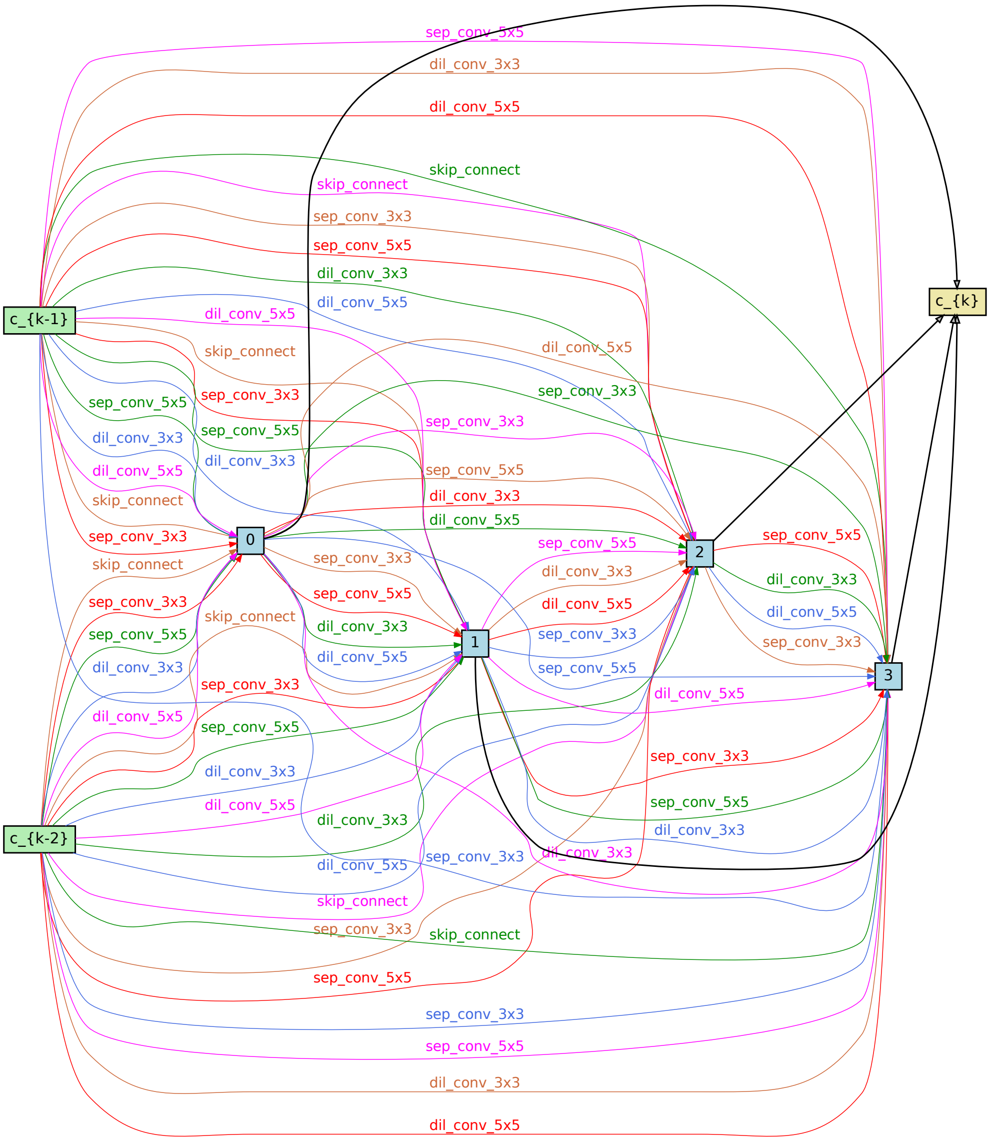


Figure E.2.: *Resulting reduction cell after training with GSparsity with* $\mu = 0.1$ *and fixed* $\varepsilon = 0.001$*. There are 64 non-zero operations in this cell after pruning.*

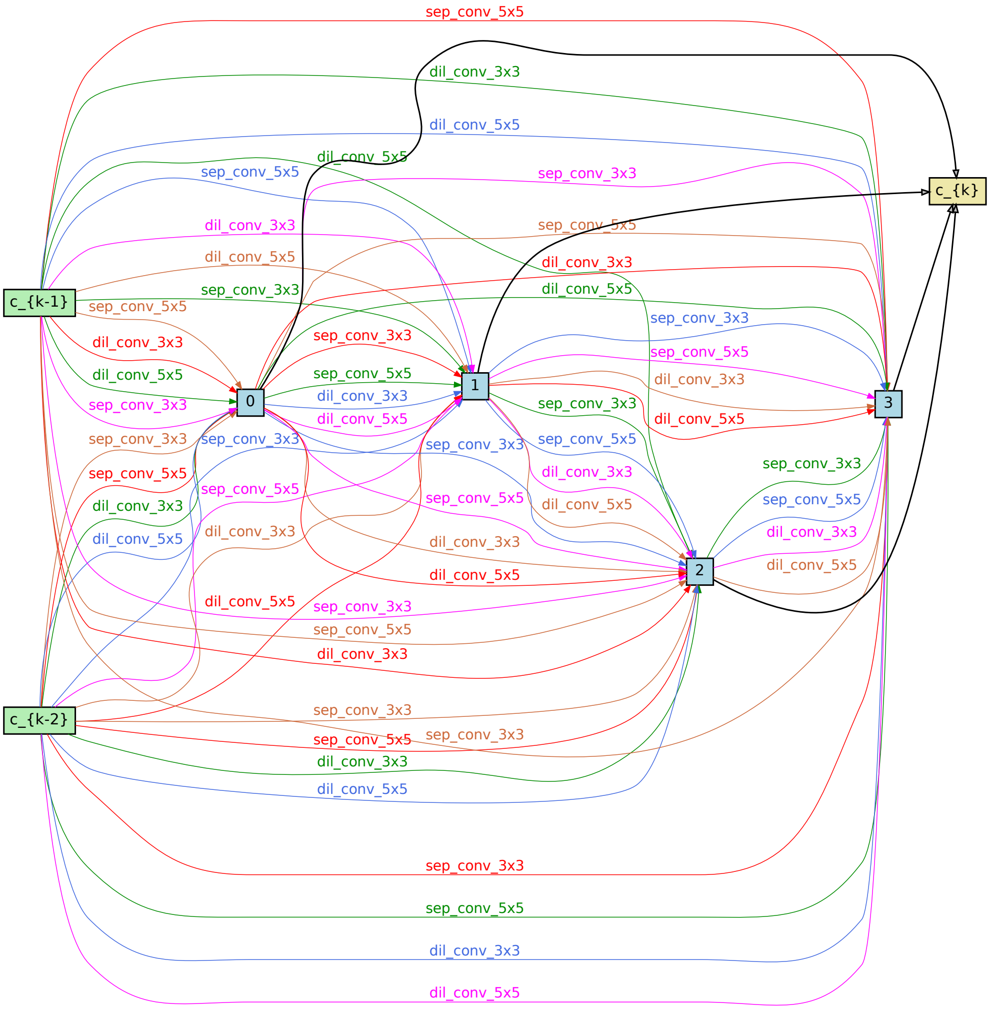


Figure E.3.: *Resulting normal cell after training with GSparsity with* $\mu = 1$ *and fixed* $\varepsilon = 0.001$. *There are 56 non-zero operations in this cell after pruning.*

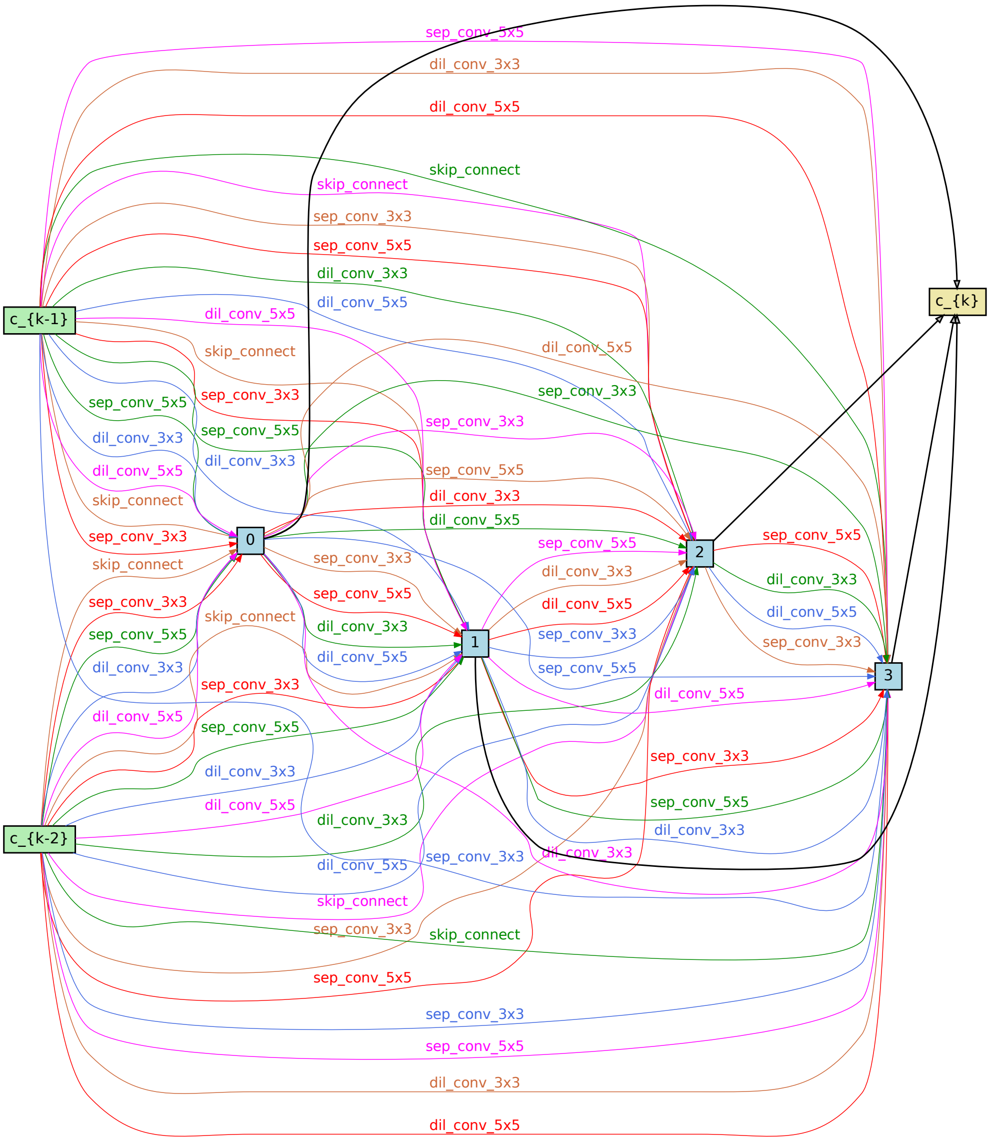


Figure E.4.: *Resulting reduction cell after training with GSparsity with* $\mu = 1$ *and fixed* $\varepsilon = 0.001$. *There are 64 non-zero operations in this cell after pruning.*

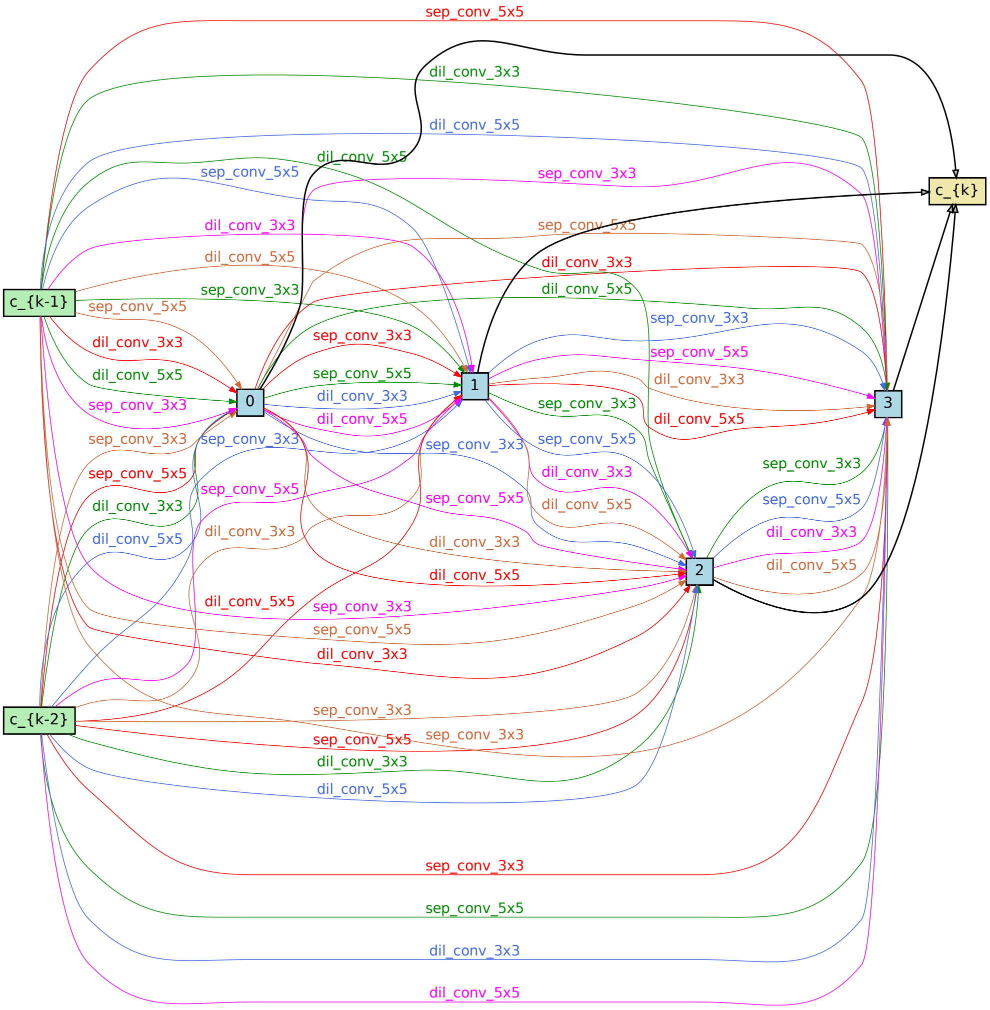


Figure E.5.: *Resulting normal cell after training with GSparsity with* $\mu = 10$ *and fixed* $\varepsilon = 0.001$. *There are 56 non-zero operations in this cell after pruning.*

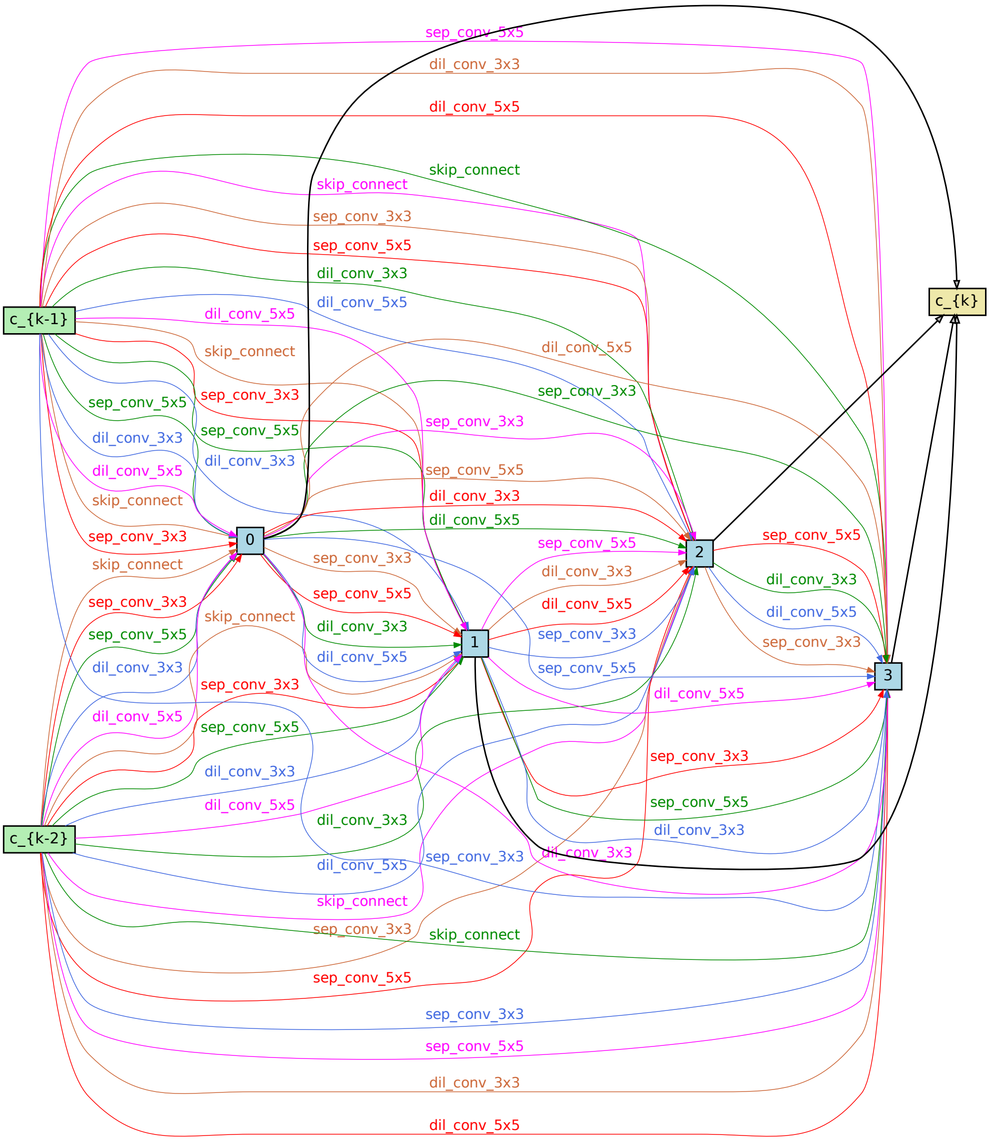
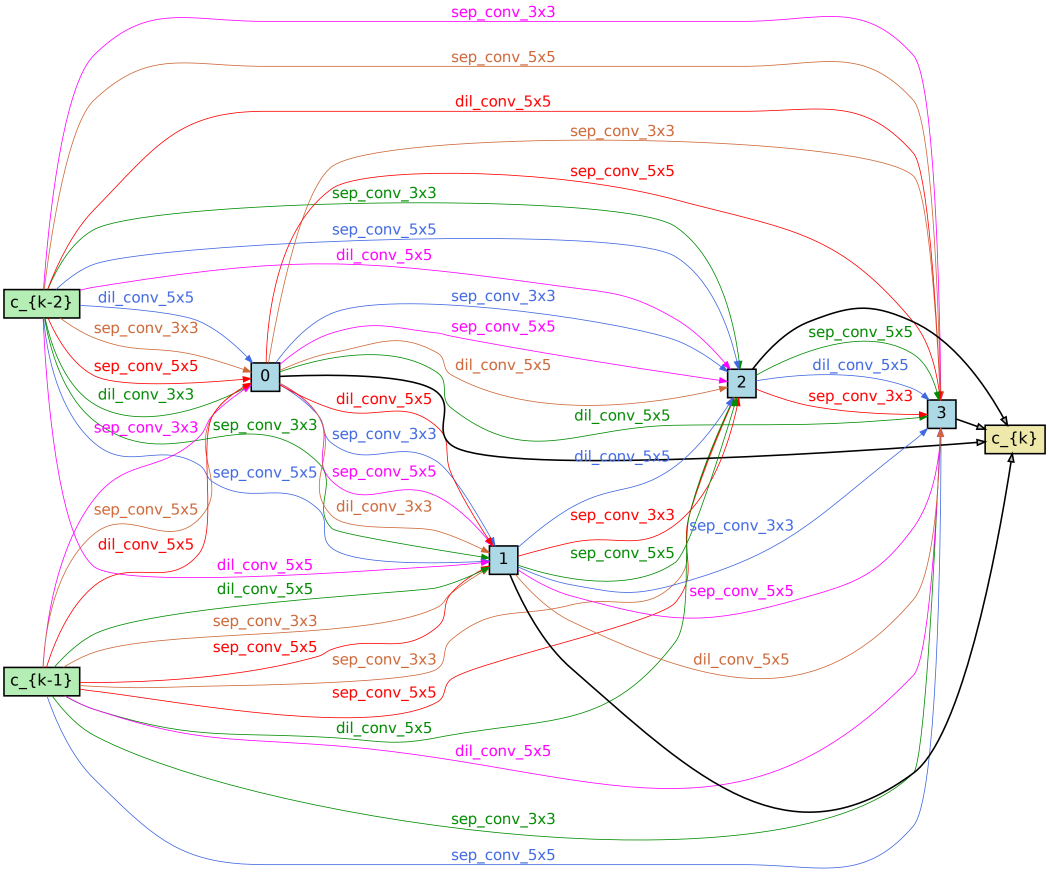
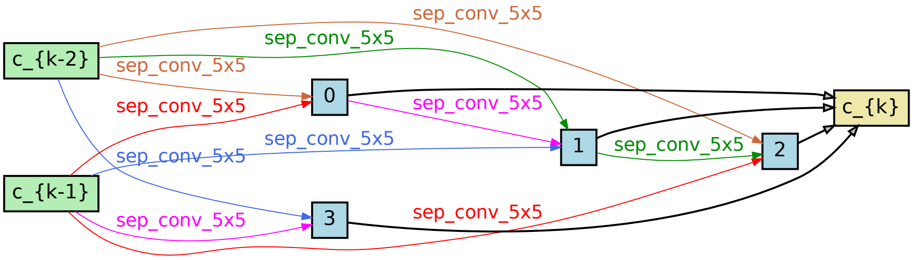


Figure E.6.: *Resulting reduction cell after training with GSparsity with* $\mu = 10$ *and fixed* $\varepsilon = 0.001$. *There are 64 non-zero operations in this cell after pruning.*

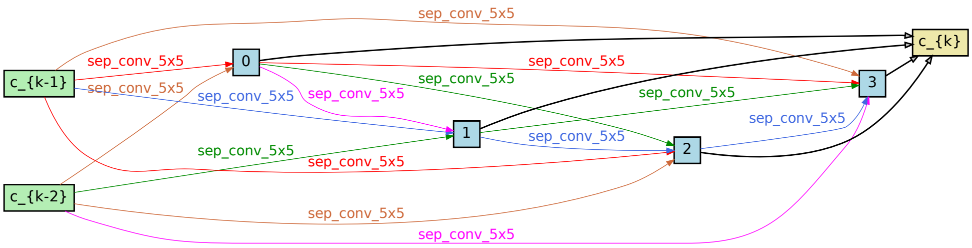


(a) *Normal cell with* $\mu = 50$*. There are 44 non-zero operations in this cell after pruning.*

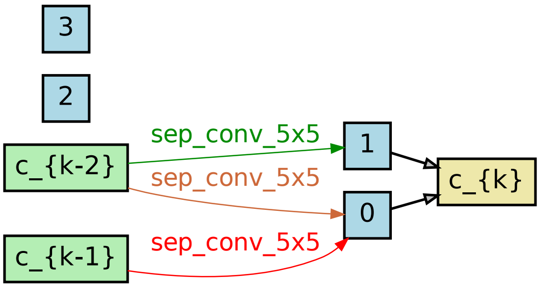


(b) *Reduction cell with* $\mu = 50$*. There are 11 non-zero operations in this cell after pruning.*

Figure E.7.: *Resulting network structure after training with GSparsity with* $\mu = 50$ *and fixed* $\varepsilon = 0.001$.

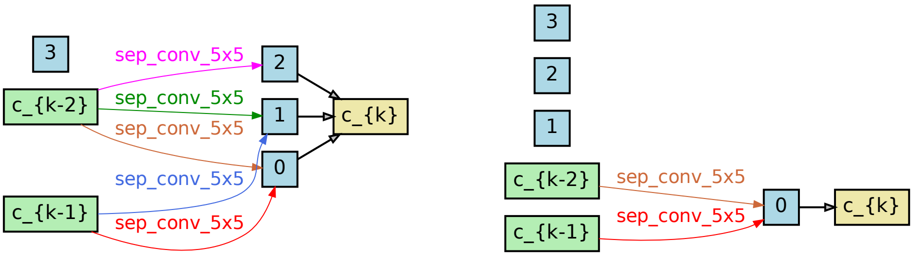


(a) *Normal cell with* $\mu = 100$*. There are 14 non-zero operations in this cell after pruning.*



(b) *Reduction cell with* $\mu = 100$*. There are 3 non-zero operations in this cell after pruning.*

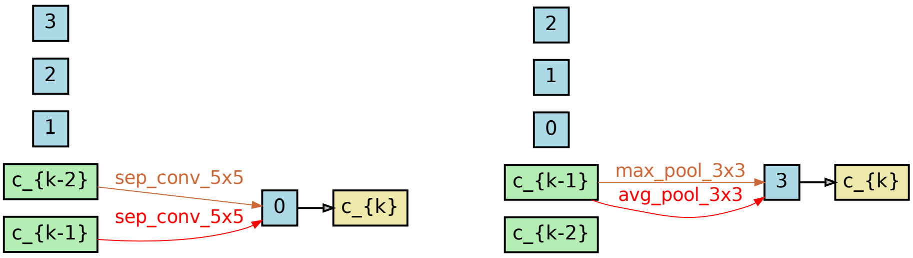
Figure E.8.: *Resulting network structure after training with GSparsity with* $\mu = 100$ *and fixed* $\varepsilon = 0.001$.



(a) *Normal cell with* $\mu = 200$*. There are 5 non-zero operations in this cell after pruning.*

(b) *Reduction cell with* $\mu = 200$*. There are 2 non-zero operations in this cell after pruning.*

Figure E.9.: *Resulting network structure after training with GSparsity with* $\mu = 200$ *and fixed* $\varepsilon = 0.001$.



(a) *Normal cell with* $\mu = 500$. *There are 2 non-zero operations in this cell after pruning.*

(b) *Reduction cell with* $\mu = 500$. *There are 2 non-zero operations in this cell after pruning.*

Figure E.10.: *Resulting network structure after training with GSparsity with* $\mu = 500$ *and fixed* $\varepsilon = 0.001$.

c_{k}
3
2
1
0
c_{k-1}
c_{k-2}
c_{k}
3
2
1
0
c_{k-1}
c_{k-2}

(a) *Normal cell with* $\mu = 1000$. *There are 0 non-zero operations in this cell after pruning.*

(b) *Reduction cell with* $\mu = 1000$. *There are 0 non-zero operations in this cell after pruning.*

Figure E.11.: *Resulting network structure after training with GSparsity with* $\mu = 1000$ *and fixed* $\varepsilon = 0.001$.

Appendix F

Publications in the context of this work

- **Gradvis: Visualization and second order analysis of optimization surfaces during the training of deep neural networks**
  A. Chatzimichailidis, J. Keuper, F.J. Pfreundt, N.R. Gauger
  In *IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC) (pp. 66-74).* IEEE (2019)

- **Combating Mode Collapse in GAN Training: An Empirical Analysis using Hessian Eigenvalues.**
  R. Durall, A. Chatzimichailidis, P. Labus, J. Keuper
  In *Proceedings of the 16th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications, VISIGRAPP 2021, Volume 4: VISAPP. (pp. 211-218)* (2021)

- **ProxSGD: Training Structured Neural Networks under Regularization and Constraints**
  Y. Yang, Y. Yuan, A. Chatzimichailidis, RJG van Sloun, L. Lei, S. Chatzinotas
  In *International Conference on Learning Representations.* (2019)

- **Group Sparsity: A Unified Framework for Network Pruning and Neural Architecture Search**
  A. Chatzimichailidis, A. Zela, S. Shalini, P. Labus, J. Keuper, F. Hutter, Y. Yang
  In *CVPR2021-NAS: Computer Society Conference on Computer Vision and Pattern Recognition: Workshop on Neural Architecture Search.* (2021)

- **GSparsity: Unifying Network Pruning and Neural Architecture Search by Group Sparsity**
  A. Chatzimichailidis, A. Zela, J. Keuper, Y. Yang
  In *International Conference on Automated Machine Learning (Late-Breaking Workshop).* (2022)

The author also contributed to

- **MSM: Multi-stage Multicuts for Scalable Image Clustering**
  K. Ho, A. Chatzimichailidis, M. Keuper, J. Keuper
  In *International Conference on High Performance Computing Workshop: Machine Learning on HPC Systems (MLHPCS) (pp. 267-284)* (2021)

Appendix G

Curriculum Vitae

Work Experience

2021–today	**Research Scientist** Fraunhofer ITWM Department for High Performance Computing, GreenByIT Group

Education

2018–2021	**PhD Student** TU Kaiserslautern
2016–2017	**Master of Science in Physics** ETH Zürich Master Thesis: *Studies on time dependent activity distributions for the SAFIR project* Supervisor: *Prof. Dr. Günther Dissertori*
2013–2016	**Bachelor of Science in Physics** ETH Zürich