Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik

Bachelor Thesis Computer Science

Scale invariant parameter spaces for ReLU networks

Christian Jestädt

25.3.2025

Reviewer

Prof. Dr. Philipp Hennig
Department of Computer Science
University of Tübingen

Supervisor

Nathaël Da Costa
University of Tübingen

Jestädt, Christian:
Scale invariant parameter spaces for ReLU networks
Bachelor Thesis Computer Science
Eberhard Karls Universität Tübingen
Period: 25.11.2024-25.3.2025

Abstract

Scale invariances in ReLU networks allow for parameter transformations, which leave the output of the network unchanged, but introduce redundancy in the parameter space, that can hinder optimization. This thesis presents two methods constructing a smaller parameter space, which is invariant to the scaling operation.
The $\mathcal{G}$-space is created using paths, which span the network from input to output. The weight fixing method fixes specific weights in the network in order to prevent rescaling. Both methods are implemented in Pytorch and evaluated on the MNIST and CIFAR-10 datasets.

Acknowledgements

I am grateful for the opportunity to write my bachelor thesis in the research group of Professor Philipp Hennig.
I would especially like to thank my supervisor Nathaël for his excellent support.
I would also like to thank Tobias for offering his help with the the codebase.

Contents

List of Figures

List of Tables

Notation

General notation & Neural Networks

$\mathbb{I}(...)$	Indicator function
$\mathbb{R}/\{M\}$	Real numbers excluding set M
x_l	Output of layer l
W_l	Weight matrix of layer l, $W_l \in \mathbb{R}^{n \times m}$
$W_{l,m,n}$	Entry at row m, column n in layer l
b_l	Bias vector of layer l
w	Single weight
L	Number of layers in the network
σ	ReLU activation function: $\sigma(x) = \max(0, x)$
μ	Learning rate
θ	Vector of all parameters, $\theta \in \mathbb{R}^m$
$\mathcal{L}$	Loss function (usually cross-entropy loss)
$\frac{\partial \mathcal{L}}{\partial w}$	Gradient of the loss with respect to weight w

$\mathcal{G}$-Space Specific Notation

v	Path value, defined as the product of weights along a path
V	$\mathcal{G}$-space
p	Path vector
$\mathcal{P}_0$	Set of basis paths
A	Structure matrix (paths as column vectors)
$r(w)$	Weight ratio
$R(p)$	Path ratio, also product of weight ratios along a path
m	Total number of parameters
n	Total number of paths
H	Number of hidden nodes
$d_{in}(W)$	Input dimension of a weight matrix W
S	Path factor matrix (captures fixed scaling contributions)

Chapter 1

Introduction

Scale invariance is a property of ReLU neural networks: It allows us to change the parameters of a network while leaving the output function unchanged. In particular, multiplying all incoming weights to a hidden node by a positive scalar and dividing all outgoing weights by the same scalar has no effect on the network's forward pass. These symmetries introduce additional directions in the parameter space, which can distort the geometry of the loss landscape and negatively impact training. A better understanding of these invariances can help us to design algorithms that are more robust and potentially improve training performance.

Other research Several researchers have studied the implications of scaling symmetries in ReLU networks. Many works have focused on optimizing over scale-invariant manifolds. Notable examples include Huang et al. [6], Badrinarayanan et al. [2], and Yi et al. [12], who explore projection-based or normalization-based techniques to address the issue indirectly.
Neyshabur et al. [9] propose a path norm to address this issue. Meng et al. [8] propose an optimization method that directly operates along similar invariant paths.
Most of these works however focus on building a specialized optimization rules or regularization techniques. To our knowledge, there is no existing implementation that explicitly constructs and operates on a scale invariant parameter space, independent of the optimizer or regularizer itself.
An advantage of such a parameter space is that it allows us to directly apply standard optimizers and study it's properties.

$\mathcal{G}$-space In this thesis, we construct the $\mathcal{G}$-space, a scale-invariant parameter space derived from paths through the network. Our work is primarily based on the $\mathcal{G}$-SGD method proposed by Meng et al. [8], which optimizes directly on path values that are invariant to scaling transformations.

We define explicit rules to map between the original weight space and the $\mathcal{G}$-space. Although $\mathcal{G}$-SGD is accompanied by a GitHub repository, we found the existing implementation restricted and too specific to SGD in particular. As a result, we developed our own implementation.

Weight fixing $\mathcal{G}$-SGD [8] introduces fixed skeleton weights, which already prevent rescaling within the network, leading to the idea of out second method: Directly fixing weights in the original parameter space to eliminate the scaling symmetries. More precisely, one outgoing weight per hidden node is fixed.

The project is accompanied by a Pytorch repository. It includes an implementation of the $\mathcal{G}$-space, the weight fixing method, the code to run all the presented tests and generate the visualizations and can be found here: github.com/chris-je/ScaleInvariantParameterSpaces.

Overview We begin by introducing the necessary notation for neural networks and formally describing the scaling symmetries.
We create the $\mathcal{G}$-space by extracting a scale-invariant parameter space from $\mathcal{G}$-SGD method and and extending it to support arbitrary values for the skeleton weights. We then present the weight fixing method as a simpler alternative.
In the experiment section, we compare training performance using the common benchmarking datasets MNIST and CIFAR-10. Also, we look at the properties of the fixed weights parameter space and give an outlook on how to improve this work.

1.1 Background

Before analyzing the effects of scaling symmetries, we first introduce the relevant notation and terminology.

1.1.1 Neural networks

We denote a neural network as sequence of layers, where each layer l consists of a weight matrix W_l, a bias term b_l and an activation function σ. A single weight should be denoted as w. The output of layer l can be computed as follows:

$$x_l = \sigma(W_l x_{l-1} + b_l) \tag{1.1}$$

Further, we define θ to be the set of parameters of the network and $y = f(x|\theta)$ to be the network function. The total number of layers is L and the total

number of parameters is m.
Unless stated otherwise, we assume fully connected layers.

1.1.2 CNN's

CNN's are a popular architecture in image recognition tasks. They reuse the small set of parameters - called kernel or filter - for each location on the input. Let $*$ denote the convolution operation, then the output of a single kernel can be written as:

$$x_l = \sigma(W_l * x_{l-1} + b_l) \tag{1.2}$$

Which we will extend to multiple input channels k and multiple output channels c. The output for layer l, output channel c can be written as:

$$x_l^c = \sigma\left(\sum_k W_l^{c,k} * x_{l-1}^k + b_l^c\right) \tag{1.3}$$

Pooling layers Pooling layers allow for a spatial reduction of the input size. Max pooling, for example, select the maximum selects the maximum value in each region (e.g. a 3×3 window), while average pooling computes the mean of all values in in that region. Let $z_1, z_2, ..., z_n$ denote the inputs in a single pooling region, then the output of a single pooling operation is:

$$\begin{aligned} x &= \max(z_1, z_2, ..., z_n) \\ x &= \operatorname{avg}(z_1, z_2, ..., z_n) \end{aligned} \tag{1.4}$$

1.1.3 ReLU Activation function

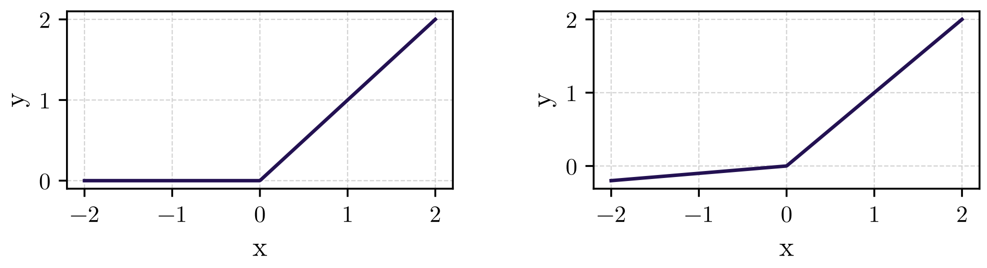
The Rectified Linear Unit (ReLU) is one of the most commonly used activation functions in deep learning. We define it as:

$$\operatorname{ReLU}(x) = \max(0, x) \tag{1.5}$$

And the leaky ReLU function as:

$$\operatorname{LeakyReLU}(x) = \max(0.1x, x) \tag{1.6}$$

If not mentioned otherwise, σ will explicitly refer to the ReLU activation function.



((a)) Output of the ReLU function. A clear cutoff below 0 is visible

((b)) Output of the LeakyReLU function. Below 0 the value is less inclined

Figure 1.1: Graphs of the ReLU and LeakyReLU function

1.1.4 Cross entropy loss

The cross entropy loss function is commonly used for classification tasks like image classification. We will explicitly use cross entropy loss, which we will refer to as $\mathcal{L}$.

$$\mathcal{L} = -\sum_i y_i \log(\hat{y}) \tag{1.7}$$

$y_i \in \{0, 1\}$ is the true label (1 for correct class) and
$\hat{y}_i \in [0, 1]$ is the predicted probability for class i.

1.1.5 Optimizers

Stochastic Gradient Descent (SGD)

SGD is a commonly used optimizer that updates the networks parameters in the direction of the negative gradient. The update rule for the parameters at iteration $t + 1$ is:

$$\theta^{t+1} = \theta^t - \mu \frac{\partial \mathcal{L}}{\partial \theta^t} \tag{1.8}$$

Adam

Adam is a more advanced optimizer, which adaptively changes the learning rate for each parameter based on an exponentially decaying average of past gradients s_i^t and squared gradients r_i^t. The update rule is:

$$\theta^{t+1} = \theta^t - \mu \cdot \frac{s_i^t}{\sqrt{r_i^t} + \varepsilon} \tag{1.9}$$

with:

$$\begin{aligned} s_i^t &= \beta_1 \cdot s_i^{t-1} + (1-\beta_1) \cdot \frac{\delta \mathcal{L}}{\partial w_i} \\ r_i^t &= \beta_2 \cdot s_i^{t-1} + (1-\beta_2) \cdot \left(\frac{\delta \mathcal{L}}{\partial w_i}\right)^2 \end{aligned} \tag{1.10}$$

1.2 Symmetries in neural networks

In ReLU networks, different parameter configurations can produce identical network outputs during a forward pass.
Given the parameters θ of a neural network, there exists a different set of parameters $\theta' \neq \theta$, such that the network computes the same function: $f(x|\theta) = f(x|\theta')$ for all inputs x. These redundancies are known as invariances. In the following, we describe two of the most common types of invariances.

1.2.1 Permutation Symmetry

Most neural networks, not only ReLU networks, possess a property called *permutation symmetry.* It can be seen as permutating or reordering nodes within a layer of a network along with their associated weights and biases.

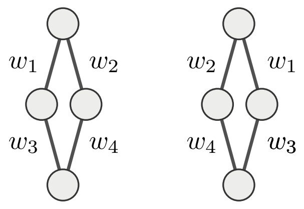


Figure 1.2: Visualization of the permutation of two middle nodes and their weights. Both networks compute the same function.

1.2.2 Scaling Symmetry

ReLU networks also have *scaling symmetries*, which allow us to rescale certain parameters while maintaining the network function. This is achieved by multiplying all ingoing weights to a hidden node by factor α and dividing all outgoing weights by α. As long as $\alpha > 0$, the network's forward pass remains unchanged.

This property stems from the fact that we can move any positive factor

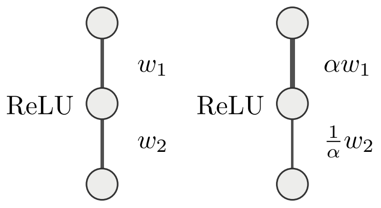


Figure 1.3: Visualization of a rescaling operation. Both networks compute the same function.

in and out of the ReLU function without affecting it's output (we further assume that $\alpha > 0$):

$$\mathrm{ReLU}(\alpha x) = \max(\alpha 0, \alpha x) = \alpha \ \max(0, x) = \alpha \mathrm{ReLU}(x) \tag{1.11}$$

Assume the output of layer i is given as:

$$x_i = \mathrm{ReLU}(W_i x_{i-1} + b_i) \tag{1.12}$$

We can multiply the the weight matrix by a factor and the output by it's inverse:

$$\frac{1}{\alpha}\mathrm{ReLU}(\alpha W_i y_{i-1} + \alpha b_i) = \mathrm{ReLU}(\frac{1}{\alpha}\alpha W_i y_{i-1} + \alpha b_i) = \mathrm{ReLU}(W_i y_{i-1} + \alpha b_i) \tag{1.13}$$

This only works for positive α, as the activation of the ReLU function will change for negative values. We can also write this in Matrix form, where D is a *monomial Matrix* with positive entries:

$$\begin{aligned} x_i &= D^{-1}\mathrm{ReLU}(DW_i x_{i-1} + Db_i) \\ x_i &= D^{-1}D \ \mathrm{ReLU}(W_i x_{i-1} + b_i) = \mathrm{ReLU}(W_i x_{i-1} + b_i) \end{aligned} \tag{1.14}$$

More general, the same holds for LeakyReLU as well:

$$x_i = \max(0.1 x_{i-1}, x_{i-1}) \tag{1.15}$$

$$x_i = \frac{1}{\alpha} \max(0.1 \alpha x_{i-1}, \alpha x_{i-1}) = \max(0.1 x_{i-1}, x_{i-1}) \tag{1.16}$$

As we can do this rescaling for every hidden node, we will assume the number of scale invariances to be equal to the number of hidden nodes.

Visualization

To visualize this effect, we construct a simple network and then add a scaling symmetry.

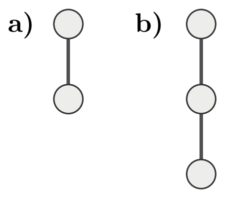
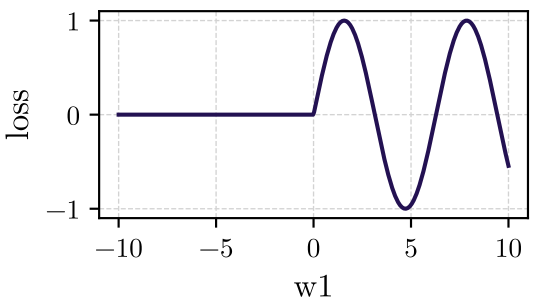
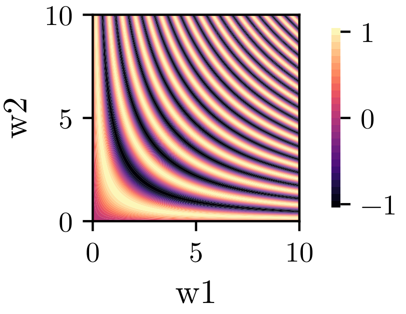


Figure 1.4: Differently sized networks **a, b** for our examples



((a)) Loss for network **a** wrt. the parameter w



((b)) Loss for network **b** wrt. both parameters

Figure 1.5: Comparison of loss for the 1- and 2- parameter networks

We start with network **a**, which consists of a single weight w_1 and a ReLU function at the end. Let y be the network output. The network function can be written as $y = \sigma(wx)$. For visualization purposes, we manually set the loss to be $\mathcal{L}(y) = \sin(y)$ and the input to be $x = 1$.
We can see that the ReLU function cuts off for inputs below 1. For positive values, we can see the typical sin function.

We extend our network by a second weight. The new network function of network **b** is:

$$y = w_2\sigma(w_1) \tag{1.17}$$

We just introduced a scale invariance, because we can rescale our parameters as follows:

$$y = \alpha w_2\sigma(\frac{1}{\alpha}w_1) \tag{1.18}$$

The new loss surface wrt. w_1 and w_2 has a clear and predictable pattern, giving us knowledge about the loss surface. The closer to 0 we get with one parameter, the more stretched the other parameter gets.

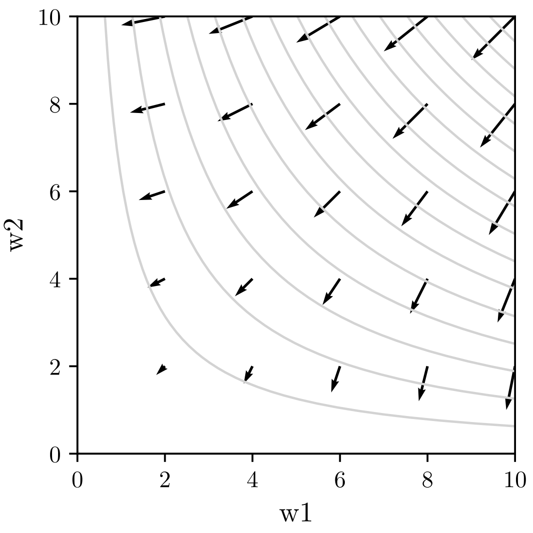


Figure 1.6: Invariant lines. Each point on such a line computes the same network function. The arrows display how the gradients of example **b** are scaled due to the scaling symmetry

Moving along a line as shown in *Figure 1.5* won't change the network output. But we can see that the gradient are scaled. Therefore, our network is invariant in the forward pass, but not invariant to training.

Why scaling symmetries matter

Scale invariances introduce several effects into the training of neural networks. First, they reduce the amount of effectively usable parameters. Although we optimize over all parameters, some directions of the parameter space lead to functionally equivalent networks. Therefore, we are training more parameters than theoretically necessary.

Second, scale invariances distort the magnitude of our gradient in multiple ways. While the gradient is always perpendicular to the scale invariant lines *Figure 1.5*, especially after a bigger step we might not end up perpendicular to the gradient due to the curvature of the symmetry. This effect reduces the effective step size for bigger steps because it introduces additional movement in the direction of the scale invariance.

Another effect is, that scaling the weights introduces a scaling of gradients. We look at a simple example network, where the output is $y = w_2 w_1 x$.

$$\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial y} \cdot \frac{\partial y}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial y} \cdot w_2 x \tag{1.19}$$

Now we scale the weights: $w'_1 = \alpha w_1$ and $w'_2 = \frac{1}{\alpha} w_2$. The loss wrt. y won't change because the output stays the same after rescaling.

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial w'_1} &= \frac{\partial \mathcal{L}}{\partial y} \cdot \frac{\partial y}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial y} \cdot w'_2 x \\ \frac{\partial \mathcal{L}}{\partial w'_1} &= \frac{\partial \mathcal{L}}{\partial y} \cdot w_2 x \frac{1}{\alpha} = \frac{\partial \mathcal{L}}{\partial w_1} \cdot \frac{1}{\alpha} \end{aligned} \tag{1.20}$$

By rescaling, we changed the gradient of w_1 by $\frac{1}{\alpha}$, w_2 will be changed inversely. Therefore, training in an unbalanced network introduces additional distortion of the gradients. The curvature of the loss surface could interfere with momentum-based optimizers, for example Adam, as well.
In practice, the data itself might limit us from exploring all directions of the parameter space or might introduce additional distortions.

Chapter 2

$\mathcal{G}$-space

2.1 Theoretical background

2.1.1 Introduction

In the attempt to remove the scaling symmetries and create a scale invariant parameter space, we present two methods to construct a scale invariant parameter space.
The $\mathcal{G}$-space method is based on $\mathcal{G}$-SGD by Meng et al. [8], a scale invariant version of SGD. We slightly change the notation in order to be consistent throughout the thesis. Also, we reconstruct the $\mathcal{G}$-space and compute the explicit update rules, as the original paper focuses only on SGD.

2.1.2 Paths

In $\mathcal{G}$-SGD, optimization is done on a so called path-space.
A path is a sequence of weights spanning from input to output through a series of hidden nodes (with one weight per layer). Let $w_1, w_2, ..., w_m$ denote all the weights in the network where m is the total number of weights. We define a path vector $p \in \{0,1\}^m$ such that its i-th entry is:

$$p_i = \begin{cases} 1, & \text{if } w_i \text{ is in path} \\ 0, & \text{otherwise} \end{cases} \tag{2.1}$$

Membership of a weight is defined as follows:

$$w_i \in p \Leftrightarrow p_i = 1 \tag{2.2}$$

We define the i-th hidden node as $\mathcal{N}_i$, the set of ingoing weights as $I(\mathcal{N}_i)$

and the set of outgoing weights as $O(\mathcal{N}_i)$. Let $\mathcal{P}$ be the set of all paths. For the path to be valid, if an ingoing weight is is the path, exactly one outgoing weight must be part of it too:

$$\sum_{w \in I(\mathcal{N})} \mathbb{I}(w \in p) = \sum_{w \in O(\mathcal{N})} \mathbb{I}(w \in p) \leq 1 \text{ for } \forall p \in \mathcal{P} \tag{2.3}$$

This condition ensures that the weights in all paths are connected through hidden nodes.
By collecting all possible path vectors, we create the structure matrix A. Each column corresponds to one path vector.

$$A = \begin{bmatrix} | & | & & | \\ p_1 & p_2 & \cdots & p_n \\ | & | & & | \end{bmatrix} \tag{2.4}$$

We define the path value v_p as product of all weights along the path p:

$$v_p = \prod_{w \in p} w \tag{2.5}$$

Theorem 2.1.1. *All paths values are inherently scale invariant, as rescaling cancels out. (Meng et al. [8], Lemma 9.1).*

We use those scale invariant paths values to later construct the scale invariant parameter space.

2.1.3 Paths not independent

Assuming a little example of two layers and 4 weights. We construct the structure matrix A consisting of all 4 paths. ([8] *Section 3.2 fig 1*)

$$A = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

The path values can be computed as $v_{p_1} = w_1 w_3$, $v_{p_2} = w_1 w_4$, $v_{p_3} = w_2 w_3$, $v_{p_4} = w_2 w_4$. We can see that they are dependent on each other because $v_{p_4} = \frac{v_2 v_3}{v_1}$.

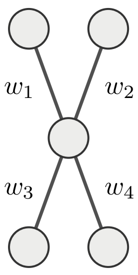


Figure 2.1: Example network with four neurons

Theorem 2.1.2. *Given a structure matrix A of a fully connected ReLU network, then $rank(A) = m - H$ where m is the number of weights and H is the number of hidden nodes. (Meng et al. [8], Lemma 9.1)*

Definition 2.1.1. *A set of paths $\mathcal{P}_0 = \{p_1, ..., p_{m-H}\}; \mathcal{P}_0 \subset \mathcal{P}$ is called a set of basis paths, if $p_1, ..., p_{m-H}$ construct a maximal linearly independent group of column vectors in structure matrix A.*

Definition 2.1.2. *The $\mathcal{G}$-space is defined as:*

$$V := \{v = (v_{p^1}, , v_{p^{m-H}}) : v \in (\mathbb{R}/\{0\})^{mH}\} \tag{2.6}$$

Therefore, the dimension of our $\mathcal{G}$-space matches the size of our scale-invariant parameter space.

2.1.4 Skeleton Method

To construct those path efficiently, Meng et al. [8] introduce the so called *skeleton method.*
Let $d_{in}(W)$ denote the input dimension and $d_{out}(W)$ the output dimension of layer matrix W. We define a subset of the weights called skeleton weights. They are typically on the diagonal of the weight matrix and are selected as follows:

First Layer (W_1):
For each column index $i_1 = 1, ..., d_{out}(W_1)$, select the element $W_1(i_1 \bmod d_{out}(W_1), i_1)$ as the skeleton weight for column i_1.

Intermediate Layer $(W_2, ..., W_{L-1})$:
For $W_2 - W_{L-1}$, select the diagonal elements of the weight matrix as skeleton weights. This assumes the network is straight in all hidden layers.

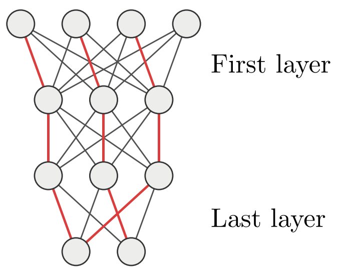


Figure 2.2: The weights marked with red are the skeleton weights. They form straight paths through the network. Every path with only red weights or one black weight is a basis path.

Last Layer (W_L): For each row index $i_{L1} = 1, ..., d_{in}(W_L)$, select the element $W_L(i_{L1}, i_{L1} \mod d_{in}(W))$ as the skeleton for that row.

The rest of the weights will be called *non-skeleton-weights.*

Basis path selection

Theorem 2.1.3. *We can select the basis paths $\mathcal{P}_0$ by selecting all paths with zero or one non-skeleton weight. (Meng et al. [8], Lemma 9.2)*

Definition 2.1.3. *We define paths only consisting of skeleton weight as skeleton path. All the other paths will be called non-skeleton-paths.*

This allows us to construct the basis paths efficiently.

Fixing skeleton weights

Given our ReLU network with H hidden nodes, we can fix H weights in place during training without losing expressiveness.
This reduces computational overhead and simplifies further equations significantly.

We fix all skeleton weights starting from the second layer, i.e. for layers $l = 2, ..., L$. The skeleton weights in the first layer are left free to change. This ensures one fixed outgoing weight per hidden node, H in total. This also ensures that each path remains free to change, as each path contains at least a free weight from the first layer.

Setting skeleton weights to 1

Meng et al. [8] set all fixed skeleton weights to 1. Since the path value is computed as the product of it's weights, this further reduces the number of non-trivial factors for the computation of v. Under those assumptions, each basis path value v can be computed only from the free weight from the first layer and at most one additional non-skeleton weight.

Chain rule inverse chain rule

We denote the set of all basis paths which contain weight w as:
$\mathcal{P}_0(w) := \{\forall v_p \in \mathcal{P}_0 : w \in p\}$

To compute the gradients in the path-space, Meng et al. [8] use the *inverse chain rule*:

$$\frac{\partial \mathcal{L}}{\partial w_p} = \sum_{v \in \mathcal{P}_0(w_p)} \frac{\partial \mathcal{L}}{\partial v} \frac{\partial v}{\partial w_p} \tag{2.7}$$

For the final computation, it is rearranged in order to yield the loss wrt. the path value. (See Appendix, *Chapter 6.2.2*).

Weight allocation

Our optimized path values can be projected back to weight values using the so called *weight allocation method.* Assume that at iteration $t+1$ we obtain an updated path value v^{t+1}.

Definition 2.1.4. *We define the path radio in iteration* $t+1$ *as* $R^{t+1}(p) = \frac{v^{t+1}}{v^t}$ *and the weight ratio at iteration* $t+1$ *as* $r^{t+1}(w) = \frac{w^{t+1}}{w^t}$

The path ratio is set equal to all the weight ratios along the path. $R(p) = \prod_{w \in p} r(w)$

We can update w at iteration $t+1$ as follows:

$$w^{t+1} = w^t \cdot r^{t+1}(w) \tag{2.8}$$

2.1.5 Grouping weights and paths

Now that we have the theoretical background, we can start by defining the explicit rules to convert to and from $\mathcal{G}$-space.
The transformation rule for a specific weight or path depends on its category, which is assigned according to the weight's location within the network.

Weight groups

We begin by assigning each weight to a group based on its layer and whether it is a skeleton weight. We assume that the skeleton selection method has already been applied to identify the relevant weights.

First-layer skeleton weight (w_j)
w_j are all skeleton weights from the first layer. They are called free skeleton weights as they are not fixed to any specific value. Each w_j appears in exactly one skeleton path and in multiple non-skeleton paths.

First-layer non-skeleton weight (w_h)
w_h are the non-skeleton weights from the first layer. Each w_h appears in one non-skeleton path.

Remaining non-skeleton weight (w_i)
w_i are all non-skeleton-weights from layer 2 to L. They are free to change and each w_i is part of exactly one basis path. Each w_i appears in one non-skeleton path.

Remaining skeleton weight (w_k)
w_k are the fixed skeleton weights (skeleton weight starting from the second layer). We won't ever update them and if not mentioned otherwise, they are set to 1.

Path groups

Similarly we split our paths into those groups:

Skeleton paths (p_j)
- Contain only skeleton weights
- Only contain one trainable weight (first layer): w_j
- All other weights will be $w_k = 1$
- Therefore the path-value can be easily computed: $v_j = \prod_{w \in p_j} w = w_j$

First-layer non-skeleton paths (p_h)

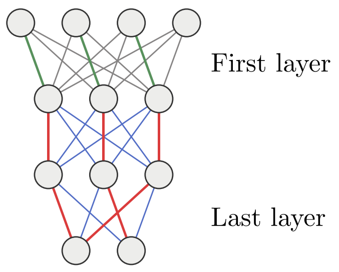


Figure 2.3: **green:** free skeleton weight w_j, **grey:** w_h, **red:** fixed skeleton weight w_k, **blue:** w_i

- Contain a non-skeleton weight in the first layer
- Their only trainable weight (first layer): w_h
- All other weights will be $w_k = 1$
- Therefore the path value can be computed as $v_h = \prod_{w \in v_h} w = w_h$

Remaining non-skeleton paths (p_i)
- We define the first layer skeleton weight of path p_i as $w_{j:p_i}$
- Contains two trainable weights:
- $w_{j:p}$ from the first layer and w_i
- All other weight are 1
- The path value can be computed as $v_i = \prod_{w \in v_i} w = w_i \cdot w_{j:p_i}$

2.2 Construct $\mathcal{G}$-space

Let V_j, V_h and V_i denote the sets of all path values v_j, v_h, v_i accordingly. We define the $\mathcal{G}$-space as concatenation of those sets:

$$V := \{V_j, V_h, V_i\} \tag{2.9}$$

A direct reconstruction of the parameter space θ from $\mathcal{G}$-space alone is not possible, e.g. we need to save the network structure as well to be able to correctly map back.

2.2.1 Explicit algorithm

The final update rules using are derived in *Chapter 6.2.2*.

Skeleton method

First, we need to run the skeleton method to categorize the weights into the different groups. It essentially selects the elements on the diagonal, but the first and last layer need special care for the case that they aren't diagonal. It is sufficient to run this algorithm once for every network.

Algorithm 1 Skeleton Method

```
1: Input: Neural network with weight matrices $\{W_1, W_2, \ldots, W_L\}$
2: Initialize mask[$l$] $\leftarrow \mathbf{0}$ for each layer $l$
3: for each layer $l = 1$ to $L$ do
4:     if $l = 1$ then                                          ▷ First layer
5:         for $i_1 = 1$ to $d_{\text{out}}(W_1)$ do
6:             $r \leftarrow i_1 \bmod d_{\text{out}}(W_1)$
7:             mask[1]$(r, i_1) \leftarrow 1$
8:         end for
9:     else if $l = L$ then                                     ▷ Last layer
10:        for $i = 1$ to $\min(d_{\text{in}}(W_L), d_{\text{out}}(W_L))$ do
11:            mask[$L$]$(i, i) \leftarrow 1$
12:        end for
13:    else                                                     ▷ Hidden layers
14:        for $i = 1$ to $d_{\text{in}}(W_l)$ do
15:            $c \leftarrow i \bmod d_{\text{out}}(W_l)$
16:            mask[$l$]$(i, c) \leftarrow 1$
17:        end for
18:    end if
19: end for
20: return mask
```

Table 2.1: Computation rules for path value and gradient

Path group	Path value	Path gradient
p_j	$v_j = w_j$	$\psi = \sum_{v \in \mathcal{P}_0(w_p)} \frac{\partial \mathcal{L}}{\partial v_i} \cdot w_i$ $\frac{\partial \mathcal{L}}{\partial v_j} = \frac{\partial \mathcal{L}}{\partial w_j} - \psi$
p_h	$v_h = w_h$	$\frac{\partial \mathcal{L}}{\partial v_h} = \frac{\partial \mathcal{L}}{\partial w_h}$
p_i	$v_i = w_{j:p_j} \cdot w_i$	$\frac{\partial \mathcal{L}}{\partial v_i} = \frac{\partial \mathcal{L}}{\partial w_i} \cdot \frac{1}{w_{j:p_i}}$

Forward transformation

The forward transformation allows us to transform our parameters from weight space into $\mathcal{G}$-space. It essentially iterates over all weights and computes the path value & gradient of them according to *Table 2.1*, depending on which category a weight belongs to. The new path value with it's gradient are then appended to the $\mathcal{G}$-space.

Algorithm 2 $\mathcal{G}$-space - Forward transformation

```
1: Input: Neural network with weight matrices $\{W_1, W_2, \dots, W_L\}$
2: Input: Skeleton masks: mask
3: Initialize lists: $V_j \leftarrow [\,]$, $V_h \leftarrow [\,]$, $V_i \leftarrow [\,]$
4: for each layer $l = 1$ to $L$ do
5:     if $l = 1$ then                                         ▷ First layer
6:         for each element $w \in W_1$ do
7:             if mask[1]$(r, c) = 1$ then
8:                 Append $w$ to $V_j$
9:             else
10:                Append $w$ to $V_h$
11:            end if
12:        end for
13:    else                                                    ▷ Hidden and last layers
14:        for each element $w = W_l(r, c)$ where mask[$l$]$(r, c) = 0$ do
15:            Append $w \cdot V_j[c]$ to $V_i$
16:        end for
17:    end if
18: end for
19: $V \leftarrow$ concatenate$(V_j, V_h, V_i)$
20: return $V$
```

Backward transformation

To transform back to the weight space, we need the $\mathcal{G}$-space, the skeleton masks and our network which we want to update.
Theoretically the mask can be re-computed and we would theoretically only need parts of the network.
Essentially we map back from $\mathcal{G}$-space to weight space, only that we skip w_k and that we need to include the additional factor for w_i.

Table 2.2: Computation rules for the updated weights

Weight group	**Weight value**
w_j	$w_j^{t+1} = v_j^{t+1}$
w_h	$w_h^{t+1} = v_h^{t+1}$
w_i	$w_i^{t+1} = \frac{v_i^{t+1}}{w_{j:p_i}^{t+1}}$
w_k	$w_k^{t+1} = w_k; w_k \neq 0$

Algorithm 3 $\mathcal{G}$-space - Backward transformation

```
1: Input: Transformed vector $V$, weight matrices $\{W_1, \ldots, W_L\}$, masks $\{\text{mask}[1], \ldots, \text{mask}[L]\}$
2: Split $V$ into $(v_j, v_h, v_i)$ based on original sizes
3: Initialize index trackers: $j \leftarrow 0$, $h \leftarrow 0$, $i \leftarrow 0$
4: for each layer $l = 1$ to $L$ do
5:     if $l = 1$ then                                  ▷ First layer
6:         for each element $(r, c)$ in $W_1$ do
7:             if mask$[1](r, c) = 1$ then
8:                 $W_1(r, c) \leftarrow v_j[j]$
9:                 $j \leftarrow j + 1$
10:            else
11:                $W_1(r, c) \leftarrow v_h[h]$
12:                $h \leftarrow h + 1$
13:            end if
14:        end for
15:    else                                             ▷ Hidden and output layers
16:        for each element $(r, c)$ in $W_l$ where mask$[l](r, c) = 0$ do
17:            $W_l(r, c) \leftarrow v_i[\text{counter}]$
18:            i $\leftarrow i + 1$
19:        end for
20:    end if
21: end for
22: return updated weights $\{W_1, \ldots, W_L\}$
```

In practice, we do not iterate over every single weight but rather over entire layers. Element-wise operations are done on entire weight matrices.

2.2.2 Extension - Already trained networks

The presented method is not inherently able to deal with already trained networks, as setting the skeleton weight to 1 changes the network function. We introduce a method to allow for arbitrary w_k while ensuring efficient computation. This will still allow us to set the skeleton weights w_k to an arbitrary number as long as they are still fixed. We introduce a constant path factor $w_{s:p}$:

$$w_{s:p} = \prod_{w_k \in p} w_k \tag{2.10}$$

Which is the product of all fixed skeleton weights w_k on the path. Until now this factor was 1 so we were able to ignore it.

Table 2.3: Update rules for the case, that we don't set $w_k = 1$. Changes of this extension are annotated with $*$.

Path Group	Path value	Path gradient
p_j	$v_j = w_j \cdot \underbrace{w_{s:p}}_{*}$	$\psi = \sum_{v \in \mathcal{P}_0(w_p)} \frac{\partial \mathcal{L}}{\partial v_i} \cdot w_i \underbrace{w_{s:p}}_{*}$ $\frac{\partial \mathcal{L}}{\partial v_j} = (\frac{\partial \mathcal{L}}{\partial w_j} - \psi)\frac{1}{w_{s:p}}$
p_h	$v_h = w_h \cdot \underbrace{w_{s:p}}_{*}$	$\frac{\partial \mathcal{L}}{\partial v_h} = \frac{\partial \mathcal{L}}{\partial w_h} \cdot \frac{1}{\underbrace{w_{s:p}}_{*}}$
p_i	$v_i = w_j \cdot w_i \cdot \underbrace{w_{s:p}}_{*}$	$\frac{\partial \mathcal{L}}{\partial v_i} = \frac{\partial \mathcal{L}}{\partial w_i} \cdot \frac{1}{\underbrace{w_{s:p}}_{*} w_j}$

Let $w^{t+1\prime}$ be the weight for the case that all w_k are set to 1. The new update rule can be written as $w^{t+1} = \frac{w^{t+1\prime}}{w_{s:p}}$.

Computation of $w_{s:p}$

To allow for easy lookup, we create a matrix S_l for each layer l, which contains the constant path factors $w_{s:p}$. The location in S at which we can lookup $w_{s:p}$ will be the same location, at which the weights w_j of path v_j, w_i of path v_i and w_h of path v_h are located in the layer.

To compute S, we first generate two help-matrices per layer, containing the product of all previous and later layer weights respectively:

$$\begin{aligned} F_2 &= W_2, \\ F_l &= W_l \cdot F_{l-1}, \quad l = 3, \dots, L, \\ B_L &= W_L, \\ B_l &= W_l \cdot F_{l+1}, \quad l = L-1, \dots, 2, \end{aligned} \tag{2.11}$$

Because all w_k (except for last layer) lie on the diagonal, we have the product of all w_k up to layer l on the diagonal of matrix F. For this computation, the last layer is simply repeated until it is square.
We compute S_l as the product of all w_k before layer l and all w_k after the layer:

$$\begin{aligned} S_l(r,c) &= F_{l-1}(c,c) \cdot B_{l+1}(r,r), l = 3, ..., L-1 \\ S_L(r,c) &= F_{l-1}(c,c) \\ S_2(r,c) &= B_3(r,r) \\ S_1(r,c) &= B_2(r,r) \end{aligned} \tag{2.12}$$

2.2.3 Extension - Biases

Meng et al. [8] never mention biases. We simply append them to the $\mathcal{G}-space$, where $\boldsymbol{b}$ is the vector of all biases:

$$V := \{V_j, V_h, V_i, \boldsymbol{b}\}$$

2.2.4 Python implementation

Our *GSpace* class implements the forward and backward transformation. We provide two abstractions to allow an easy use:

Optimizer encapsulation

The $\mathcal{G}$-Space optimizer (*GOptmizer*) can be initialized with another optimizer as parameter. It computes the $\mathcal{G}$-space, which is then passed to the given optimizer. In the following example, *optimizer* will run Adam on the $\mathcal{G}$-space:

```
model_parameters = list(model.parameters())
```

```
optimizer_class = optim.Adam
learning_rate = 0.05
optimizer = GOptmizer(model_parameters, optimizer_class, learning_rate)
```

For the optimizer class we pass, currently no other parameters than *learning_rate* are supported.

Model encapsulation

Our inplementation also allows for the encapsulation of whole models. The model itself won't be altered in any way, as *GModel* will mostly just redirect function calls to the model. But *GModel* introduces a few additional methods. A model can be encapsulated as follows:

```
gmodel = GModel(model)
# Calling parameters() will now return the $\mathcal{G}$-space
gspace = gmodel.parameters()
```

We currently can't track whether the gradients of the original model have changed, therefore we need to call *update_gradients()* before working on the gradients of the $\mathcal{G}$-space:

```
loss.backward()
# call update gradients after the backward pass
# and before the optimizer step
gmodel.update_gradients()
optimizer.step()
```

Unit tests

The created $\mathcal{G}$-space is proven to behave the same as the original implementation when used with SGD. Also, the parameter space created has a size of exactly $m - H$. Calling $f^{-1}f(\theta) = \theta$ is proven to be correct. It is also proven that the skeleton weights don't update during training.

2.2.5 Limitations

The current implementation introduces some heavy restrictions. Due to the *skeleton method*, our networks need to be straight in every hidden laqyer. Also, currently only linear layers are supported. No other layer type may exist in the network.

Chapter 3

Weight fixing method

3.0.1 Introduction

Given a hidden node $\mathcal{N}$ in a ReLU network, fixing a single in- or outgoing weight is sufficient to prevent positive rescaling on this node. Let the fixed weight be w_k and let it be an outgoing weight of the node: $w_k \in O(\mathcal{N})$.
Assuming a rescaling operation, where all ingoing weights into node $\mathcal{N}$ are multiplied by α and all outgoing weights by $1/\alpha$ for $\alpha > 0$: $w' = \frac{1}{\alpha} \cdot w$ for $\forall w \in O(\mathcal{N})$ but this is not possible as w_k is fixed.
In this section we introduce an algorithm that aims to remove scale invariances based on this principle - without requiring transformation into path space and back. Our method works directly on a subset of the parameter space. The idea is to fix one outgoing weight per hidden node in order to prevent the network from rescaling.

3.0.2 Basic fixed weight selection

The fixed weight selection of this algorithm is inspired from $\mathcal{G}$-SGD [8] skeleton method.
Fixed weights in straight parts of the network are selected in a similar way by choosing the weights on the diagonal of W.

But in this case we allow the network to get bigger or smaller:
No matter how the size of the layer changes, we select exactly one outgoing weight per hidden layer, the exact amount of assumed scale invariances. The remaining parameter space has the same size as the $\mathcal{G}$-space: $m - H$.

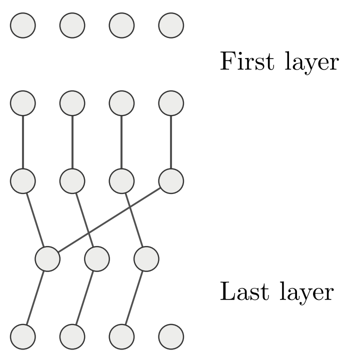


Figure 3.1: Marked in grey: the fixed weights. We can see that no fixed weights are selected in the first layer. For the other layers it is exactly one outgoing weight per hidden node.

3.0.3 Fixed weight selection

Given weight matrix $W_l \in \mathbb{R}^{r \times c}$ for layer l.
Let $W_{l,r,c}$ denote the weight at position r, c. The set of fixed weights S_l for layers $\{2, ..., L\}$ is selected as follows:

$$S = \{W_{i,k,k \ mod \ c} | k = 1, ..., r\}$$

In *Chapter 6.1.1*, we show that the number of effective scale invariances is less than one per hidden node, because of inner dependencies. Because this effect is not easily predictable, we chose to assume exactly one scale invariance per hidden node.

Setting fixed weights to 1

Similar to the $\mathcal{G}$-space, we can also chose whether we want to fix our weights to any specific value or keep them as they are. Keeping the weights also allows us to use already trained networks. Scaling the fixed weight introduces additional scaling of the gradients

3.0.4 Reparametrization

We created an algorithm [appendix] which allows to reparametrize the network in order to have 1 as value for every fixed weight without changing the network function.
An improvement of the algorithm *Chapter 6.3.1* selects the fixed weights in a way, which allows for minimal changes during reparametrization.

3.0.5 CNN's

CNN's are also scale invariant - they possess one scale invariant parameter per output channel [8]. To remove these scaling symmetries, we fix one weight in one kernel per output channel to 1.

Pooling layers Max pooling and average pooling layers - very common in CNN's - are capable of routing through scale invariances. Although pooling layers do not have any parameters themselves, they partially preserve the existing scaling properties. Let $\alpha > 0$ be our scaling factor and let n be the size of the pooling filter. When we scale all the input of a single pooling operation by α, the output will be scaled by α as well.

$$\max(\alpha x_1, \alpha x_2, ...) = \alpha \max(x_1, x_2, ...)$$

$$\text{avg}(\alpha x_1, \alpha x_2, ..., x_n) = \alpha \text{avg}(x_1, x_2, ..., x_n)$$

When we apply the pooling in a non-overlapping manner, each pooling operation receives a completely different subset of inputs. For example: $\text{pool}_1(x_1, x_2, ..., x_n)$ and $\text{pool}_2(x_{n+1,n+2}, ..., x_{2n})$ Therefore, we can scale the input for each pooling operation by a separate factor. Transmitting one scale invariance per channel and pooling operation to the next layer. When we apply pooling in an overlapping fashion, we have to scale all ingoing x of the channel by the same factor, effectively reducing the number of scale invariances transmitted to the next layer to 1 per channel.

3.0.6 Pytorch implementation

In the Pytorch implementation, we use hooks to set the gradients of the fixed weights to 0. This allows to continue working on a modified version of the original parameter space with zero additional runtime during optimization.

This implementation is way more flexible than $\mathcal{G}$-Space as it allows non-straight networks. Further, the implementation automatically selects invariant layers (currently Linear and Conv2D are supported), therefore our network may contain arbitrary layer types.
The following snippet will fix the scale invariant parameters of the model:

```
FixModel(model, fixed_factor)
```

fixed_factor can either be a boolean or a float. When it is a boolean, it will either fix the weights without changing them (False) or setting them to 1

(True). For any float, it will set the fixed weights to the given number.
Using regularizers on the fixed model may still update the value of the fixed weights.

Chapter 4

Experiments

4.1 $\mathcal{G}$-space

In our first experiments, we look at the learning behavior of the $\mathcal{G}$-space. Specifically, we compare the speed and quality of learning when using $\mathcal{G}$-SGD and $\mathcal{G}$-Adam to SGD and Adam. For $\mathcal{G}$-SGD and $\mathcal{G}$-Adam we use the default Pytorch implementations, but we pass them the $\mathcal{G}$-space instead of the weight space.

We conduct the experiments on the MNIST and CIFAR-10 datasets. MNIST is a large dataset consisting of 60.000 training and 10.000 test images of handwritten digits. Each image is grayscale and has a size of 28×28 pixels. The CIFAR-10 dataset consists of 50.000 training and 10.000 test images across 10 different categories, including examples such as "deer", "ship" and "truck".

To determine the best learning rate, we use grid search over $\mu \in 10^{-\alpha}, \alpha \in \{1, ..., 5\}$.
Since the effectiveness of different presented measures heavily depends on the learning rate, we decided against the use of gradient decay. We chose the learning rate that gives the best validation performance at any epoch. Unless stated otherwise, we use a batch size of 64. Each configuration is run 10 times and the mean is used.

MNIST The network used for our experiments with MNIST is a fully connected neural network with four hidden layers, each with 64 units. The input is a 784-dimensional vector which is the flattened image and the output are 10 logits. Each hidden layer has a ReLU activation function. The network is bias-free.

CIFAR-10 The CIFAR-10 network follows a similar architecture. The input is a $32 \times 32 \times 3 = 3072$ dimensional flattened RGB image. The network consists of five hidden layers of 64 units each. The network concludes with a final layer with 10 output classes. As with the MNIST model, all layers are bias-free and ReLU is used in all hidden layers.

Our model architectures for MNIST and CIFAR-10 were taken from the official codebase of [8].

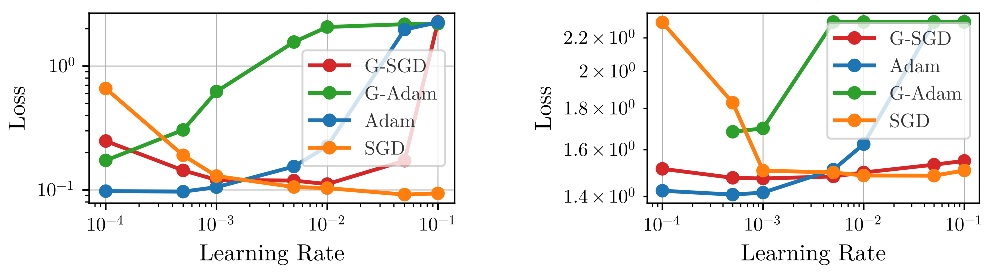


Figure 4.1: Comparison of validation loss and learning rate for MNIST (left) and CIFAR-10 (right)

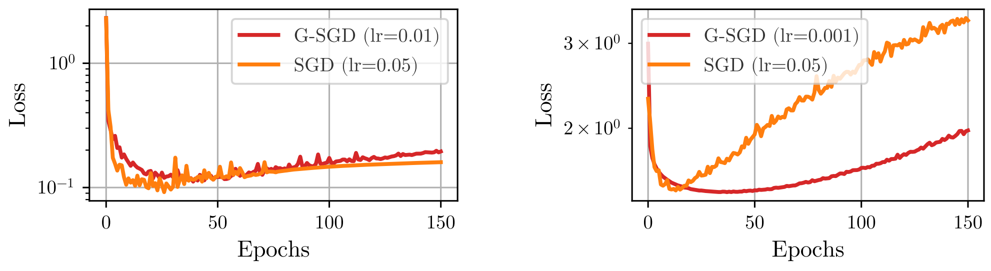


Figure 4.2: Comparison of validation losses for the learning rate with the best overall result in all epochs. MNIST (left) and CIFAR-10 (right)

Discussion

In *Figure 4.1* and *Figure 4.2*, we observe that both $\mathcal{G}$-space based methods do not significantly improve the loss compared to their counterparts in the parameter space. Notably, G-Adam consistently underperforms relative to the other optimizers for both MNIST and CIFAR-10. The concept of first and second momentum does not appear to translate well to the path space.

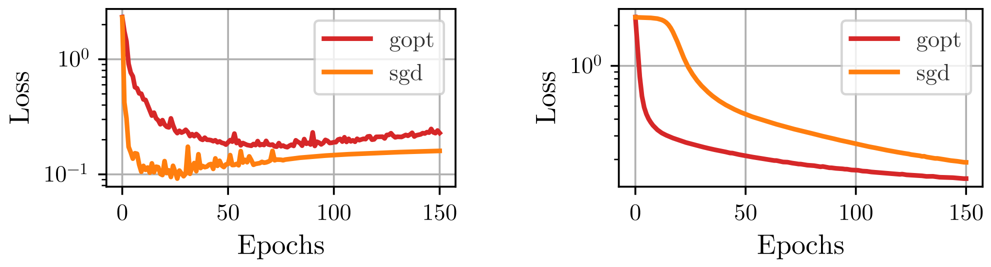


Figure 4.3: Comparison of validation losses for learning rates 0.05 (left) and 0.0005 (right) for MNIST

$\mathcal{G}$-SGD seems to be less sensitive to the choice of learning rate. While standard Adam only performs well for smaller learning rates and SGD for larger ones, $\mathcal{G}$-SGD seems to be stable accross a broader range. In *Figure 4.1*, we can see that $\mathcal{G}$-SGD converges much faster than standard SGD for smaller choices of learning rates.
Ultimately however, Adam still performs better than our $\mathcal{G}$-space based methods.

4.2 Weight fixing

4.2.1 General tests

In the next experiment, we will also look at the learning behavior, but in this case of the weight fixing method. The experimental setup is the same as for $\mathcal{G}$-space. We will compare either default

Fixed weights are set to one if not mentioned otherwise.

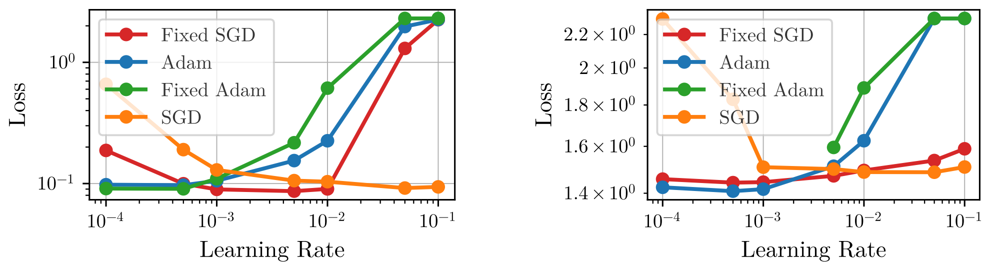


Figure 4.4: Performance comparison of Adam and SGD for in fixed weight space and normal parameter space for different learning rates and MNIST (left) and CIFAR (right).

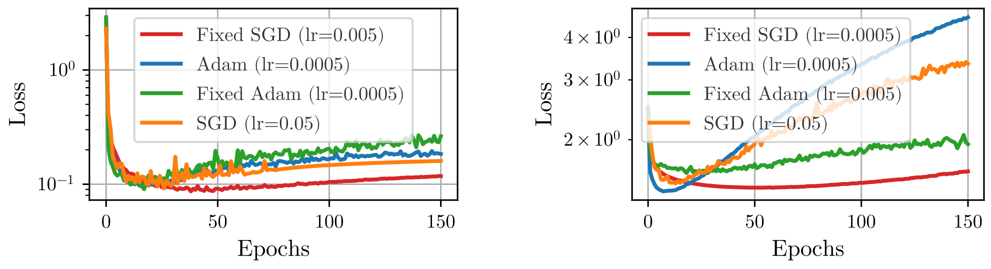


Figure 4.5: Performance comparison of Adam and SGD in fixed weight space and normal parameter space for MNIST (left) and CIFAR (right).

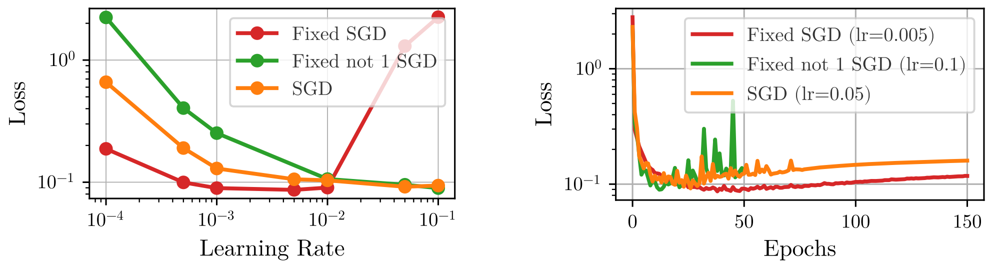


Figure 4.6: Performance comparison of Fixing our weights to one (Fixed) and fixing them without setting them to one (Fixed not 1) for MNIST. On the right, the learning rate with the best overall result is shown

Discussion

In *Figure 4.4*, we can observe that Fixed SGD performs slightly better than standard SGD, while Fixed Adam shows only a minor drop in performance compared to regular Adam.
Moreover, Fixed SGD seems to overfit less stand standard SGD. By reducing our parameter space - fixing one weight per hidden node - we effectively remove more parameters than there are scale invariances (*Chapter 6.1.1*). This reduction in the effective dimensionality of the parameter space could be one reason for the reduced overfitting.
However, when the weights are not set to 1, the weight fixing method tends to perform worse than standard SGD. As seen in *Figure 4.6 (right side)*, Fixed SGD without setting weights to one behaves like a more unstable variant of SGD.
While the weight fixing method shows a slight performance improvement over standard SGD, this benefit seems to result more from setting the weights to a specific value, rather than from removing scaling symmetries.

4.2.2 Reparametrization

In section 3.0.4 we proposed a reparametrization method, which allows to rescale all fixed weights to a value of 1 without changing the network function.

Initialization strategies like Kaiming, which are used by Pytorch by default, normally initialize the weights below 1. Scaling up each layer in order to have 1 on the fixed weight leads to large scaling factors which accumulate over all layers, leading to numerical instability.
Using the advanced fixed weight selection worked successfully for the forward pass, the scaling factors were small enough to lead to numerically stable and equivalent networks.
However, the rescaling also introduced large changes in the gradient scaling (explained in 1.2.2; why scaling symmetries matter). This leads to an unbalanced network which was numerically unstable during training due to exploding gradients.

Discussion

This rescaling method did not meet our expectations. By rescaling, we artificially made the network more unbalaned - and unbalanced networks mostly perform worse [9].
This was also a hint that the weight fixing method is not capable of compensating for (largely) unbalanced networks. Further, this led to the idea, that it might be worth to do the opposite - to balance out the network before weight fixing is applied. However, most initialization strategies already initialize networks relatively balanced.

4.2.3 Correlation learning rate and scaling value

In this experiment we want to observe how the choice of fixed weight influences the network performance and how it depends on the learning rate. To compare the effect for different choices of both values for the fixed weights b and learning rates μ, a 2D grid was chosen.

Here, each pixel corresponds to the median values of 5 runs. Some values, especially for high μ and high b lead to exploding gradients and the network reached ‘NaN‘ within iterations. The validation loss of those runs is treated as infinity. The final color is the output after 20 iterations.

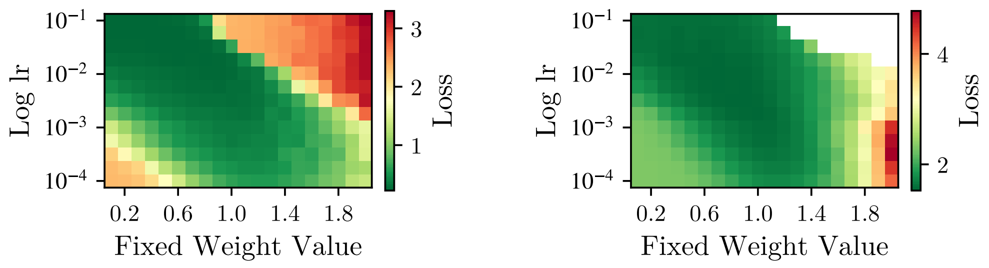


Figure 4.7: Median validation loss vs value of fixed weights for MNIST (left) and CIFAR-10 (right)

Discussion

A clear correlation between the network performance, learning rate and fixed weight scale can be seen. Further, there seems to be no clear advantage of setting the b to any specific value. Computation of those grids took multiple days each. Therefore it was not possible to run the experiment for the 150 epochs we did in the other experiments.

4.2.4 Analyzing the loss surface of weight fixing

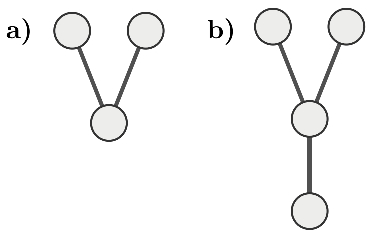


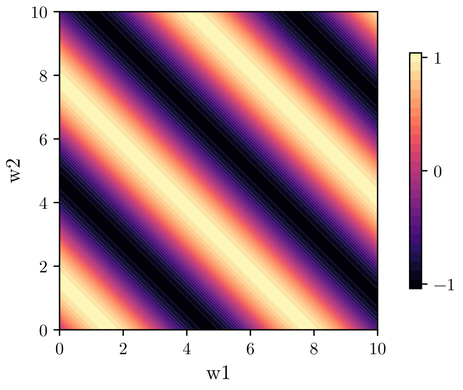
Figure 4.8: Networks **a** and **b** for our examples.

In this experiment we investigate how the loss surface changes when we change the value of a fixed parameter.
We use a simple network with two inputs and a single output, considering only positive weights. For simplicity, we fix the inputs to $x_1 = x_2 = 1$ and we define the loss function for visualization as $\mathcal{L}(y) = \sin(y)$.

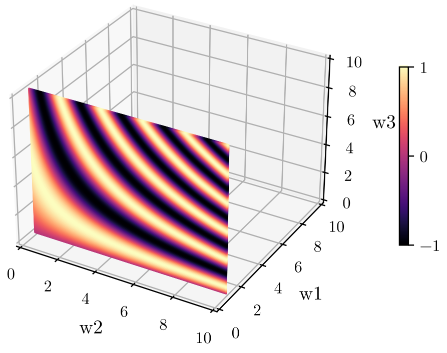
We first examine network **a**, which consists of a single layer and therefore can't posses any form of scale invariance. The output is defined as:

$$y_a = \sigma(w_1 + w_2)$$

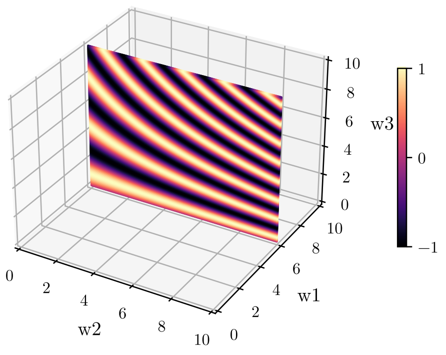
Plotting the loss with respect to w_1 and w_2, we observe the stripe pattern of the $\sin(w_1 + w_2)$ function.



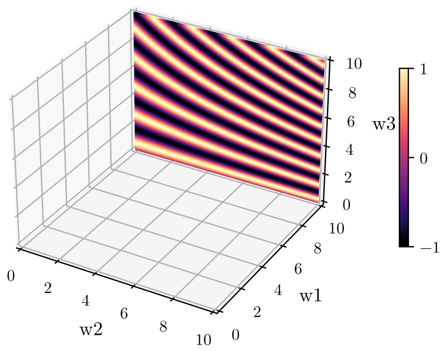
((a)) Loss surface for network **a**. The explicit loss function is $\sin(w_1 + w_2)$



((b)) Loss surface for network **b** for a $w_1 = 1$.The explicit loss function is $\sin(w_3(1 + w_2))$



((c)) Loss surface for network **b** for a $w_1 = 6$.The explicit loss function is $\sin(w_3(6 + w_2))$



((d)) Loss surface for network **b** for a $w_1 = 10$.The explicit loss function is $\sin(w_3(10 + w_2))$

Figure 4.9: Loss surfaces for the network **a** and the network **b**, where w_1 is fixed in place

Next, we add a second layer to create network **b**, adding a hidden layer and therefore introducing a scale invariance. The updated network function becomes:

$$y_b = w_3\sigma(w_1 + w_2)$$

Because of the introduced scale invariance, the effective parameter space has the same dimensionality as before. Ideally, our scale invariant parameter space should give us the undistorted parameter space from **a** back. However, we now explore how the parameter space of the weight fixing method behaves.
Instead of fixing the newly introduced weight w_3 - which would essentially revert the network back to **a** - we fix w_1. Fixing w_1 constrains the parameter space to a two-dimensional plane spanned by w_2 and w_3. In the figure we visualize the loss surface of network **b** for various fixed values of w_1.

Discussion

We can see that for w_1 close to 0, we get the same loss surface as in *figure 1.5*. This is because w_1 is effectively inactive, reducing the network function to $y = w_3(0 + w_2) = w_3 w_2$, the same as in *Section 1.2.2; example* **b**. As we increase the value of w_1, the loss surface pattern changes and seems to compress, especially for w_2 (our other weight in the same layer) close to 0.
In the previous example, we have shown that the weight fixing method has a clear correlation to the learning rate, which matches our observation from this experiment. However, additional deformation is introduced as well.
The exact behavior - especially in higher dimensions - is yet to study.

4.3 Training comparison $\mathcal{G}$-space and Weight fixing

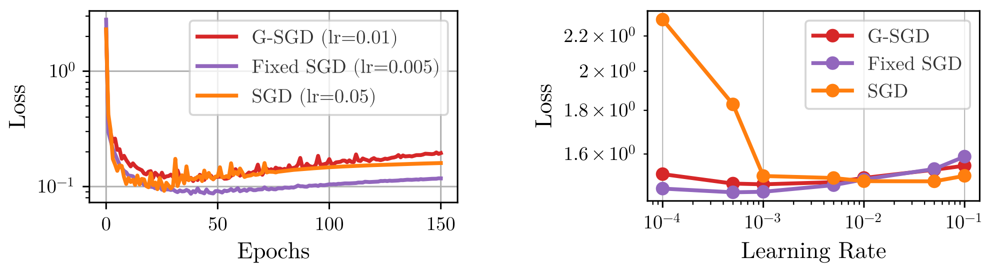


Figure 4.10: Comparison of SGD on the default weight space, on the $\mathcal{G}$-space and on the fixed weight space for MNIST. Fixed weights are set to one.

Discussion

We observe that $\mathcal{G}$-SGD and Fixed SGD behave quite similarly: both perform well across a broad range of learning rates and achieve comparable loss values,

with weight fixing showing a slight advantage.
This similarity raises the question whether the positive effect of $\mathcal{G}$-SGD observed by Meng et al. [8] truly is a result of operating in scale invariant space, or if it may instead be a result of scaling the skeleton weights to one. Unfortunately, we were not able to carry out the experiments on the fixed path scale for $\mathcal{G}$-space. Such an experiment would have been valuable as it would allow to test the performance of $\mathcal{G}$-SGD without the need to set the skeleton weights.
Additionally, the proportion of hidden nodes to the total number of weights is smaller in networks with larger layers. It remains unclear whether scale invariances play a significant enough role in larger networks to justify working in scale-invariant parameter spaces.

Chapter 5

Conclusion and Outlook

5.1 Conclusion

We presented two approaches for constructing scale invariant parameter spaces for ReLU networks. The first, called $\mathcal{G}$-space, is derived from $\mathcal{G}$-SGD by Meng et al. [8] and builds the parameter space based on paths through the network. The second, the weight fixing method, defines a subspace of the original parameter space by fixing specific weights. Both approaches produce parameter spaces of the same dimensionality, with the goal of improving training dynamics and allowing further experimentation.
The $\mathcal{G}$-space implementation enables encapsulation of either the optimizer or the model, allowing the rest of the code to work with the $\mathcal{G}$-space without further changes. The weight fixing method in contrast modifies the gradients directly by setting them to zero.
Experiments on MNIST and CIFAR-10 revealed that $\mathcal{G}$-SGD did not significantly improve over standard SGD, while Adam performed notably worse on the $\mathcal{G}$-space. Fixed SGD produced similar results to $\mathcal{G}$-SGD. Both Fixed SGD and $\mathcal{G}$-SGD showed greater robustness against the learning rate choice, converging faster than standard SGD for small learning rates.
A slight performance improvement was observed for Fixed SGD, but only when setting specific network parameters to 1 - similar to the $\mathcal{G}$-space. Much of the performance variation appears to stem from the choice of the fixed parameter values than the invariances themselves. Fixing weights without setting them to 1 led to results similar, but slightly worse than standard SGD.

5.2 Outlook

The $\mathcal{G}$-SGD implementation is currently limited to networks composed of fully connected linear layers with straight hidden layers. An important step for the

future is to adapt the skeleton method to support non-straight hidden layers - if feasible. Further extending the implementation to CNN's would also improve the applicability of the $\mathcal{G}$-space. Additionally, it might be interesting to conduct experiments using the proposed constant path scale. It allows skeleton weight to take on arbitrary values. This would help isolate the effects of the true scale invariance from those introduced by setting the weight to one.
Another potential research area is to explore the behavior of regularization in scale invariant parameter spaces.
Both methods presented in the thesis should be tested on larger and more complex network architectures to better understand the practical impact of scale invariances on real-world deep learning applications.
Finally, scale invariant parameter spaces might be useful beyond training. For instance, methods like Laplace approximations could benefit from the additional structural knowledge of the loss landscape.

Chapter 6

Appendix

6.1 Scaling symmetries

6.1.1 Less than one scale invariance per hidden node

Here we show that the number of effective scale invariances is less than the number of hidden nodes, due to inner dependencies.
Given a multi-layer ReLU network:

$$y = \sigma(W\sigma(Wx))$$

We can rescale the network as follows:

$$y = \sigma(D\ W\sigma(D^{-1}W))$$

Same works over multiple layers when using a scalar, because the scalar is commutative:

$$y = \alpha\ \sigma(W\sigma(\frac{1}{\alpha}\ W))$$

$$y = \sigma(\alpha W\sigma(\frac{1}{\alpha}\ W))$$

$$y = \sigma(W\alpha\sigma(\frac{1}{\alpha}\ W))$$

$$y = \sigma(W\sigma(\alpha\frac{1}{\alpha}\ W))$$

The same is not true for matrices, except if all the weights in between are diagonal (or more generally monomial), because matrices are not commutative: DW is not necessarily equal to WD. In our network, W is only monomial, if paths are not connected to each other.
The bigger our layer size, the more influence the other weights have and the less pronounced the scale invariance after a few layers gets.

6.2 $\mathcal{G}$-space

6.2.1 Scale invariance of paths

In this proof we show that all paths are inherently scale invariant.
Let the weights w_k and w_f be part of the path p.
Let w_k be the ingoing and w_f be outgoing weight of hidden node $\mathcal{N}$: $w_k \in I(\mathcal{N}); w_f \in O(\mathcal{N})$.
We can compute the path value as $v_p = ... \cdot w_k \cdot w_f \cdot ...$
Assuming a rescaling operation, where all ingoing weights into node $\mathcal{N}$ are multiplied by α and all outgoing weights by $1/\alpha$ for $\alpha > 0$. The new path value v' after rescaling will be invariant to the rescaling operation: $v'_p = ... \cdot \alpha w_k \cdot \frac{1}{\alpha} w_f = ... \cdot w_k \cdot w_f \cdot ... = v_p$.

6.2.2 Update rules - forward transformation

The computation of the path values has already been explained in *Chapter 2.1.2*. Therefore, we will only derive the gradients here. We use the inverse chain rule as proposed by Meng et al. [8].
We assume, that the fixed weights (w_k) are set to 1.

Gradient of v_h:

$$\frac{\partial \mathcal{L}}{\partial w_h} = \sum_{v_p \in \mathcal{P}_0(w_h)} \frac{\partial \mathcal{L}}{\partial v_p} \frac{\partial v_p}{\partial w_h}$$

Only a single path (v_h) crosses through w_h, therefore we can simplify the sum to:

$$\frac{\partial \mathcal{L}}{\partial v_h} \frac{\partial v_h}{\partial w_h} = \frac{\partial \mathcal{L}}{\partial w_h}$$

$v_h = w_h$ and $\frac{\partial v}{\partial w} = \frac{v}{w}$:

$$\frac{\partial \mathcal{L}}{\partial v_h} \frac{\partial v_h}{\partial w_h} = \frac{\partial \mathcal{L}}{\partial v_h} \frac{v_h}{w_h} = \frac{\partial \mathcal{L}}{\partial v_h} = \frac{\partial \mathcal{L}}{\partial w_h}$$

Gradient of v_i:

$$\frac{\partial \mathcal{L}}{\partial w_i} = \sum_{v_p \in \mathcal{P}_0(w_i)} \frac{\partial \mathcal{L}}{\partial v_p} \frac{\partial v_p}{\partial w_i}$$

Only a single path (v_i) crosses through w_i, therefore we can simplify the sum to:

$$\frac{\partial \mathcal{L}}{\partial v_i} \frac{\partial v_i}{\partial w_i} = \frac{\partial \mathcal{L}}{\partial w_i}$$

$\frac{\partial v}{\partial w} = \frac{v}{w}$:

$$\frac{\partial \mathcal{L}}{\partial v_i}\frac{v_i}{w_i} = \frac{\partial \mathcal{L}}{\partial w_i}$$
$$\frac{\partial \mathcal{L}}{\partial v_i} = \frac{\partial \mathcal{L}}{\partial w_i}\frac{w_i}{v_i}$$

$v_i = w_i w_j$, therefore:

$$\frac{\partial \mathcal{L}}{\partial v_i} = \frac{\partial \mathcal{L}}{\partial w_i}\frac{w_i}{v_i} = \frac{\partial \mathcal{L}}{\partial w_i}\frac{w_i}{w_i w_j} = \frac{\partial \mathcal{L}}{\partial w_i}\frac{1}{w_j}$$

Gradient of v_j: We use the inverse chain rule:

$$\frac{\partial \mathcal{L}}{\partial w_j} = \sum_{v_p \in \mathcal{P}_0(w_p)} \frac{\partial \mathcal{L}}{\partial v_p}\frac{v_p}{w_j}$$

The paths through w_j are v_i and a single v_j, so we can split those up:

$$\frac{\partial \mathcal{L}}{\partial w_j} = \underbrace{\frac{\partial \mathcal{L}}{\partial v_j}\frac{v_j}{w_j}}_{*} + \sum_{w_j \in \mathcal{P}_0(w_j)} \frac{\partial \mathcal{L}}{\partial v_i}\frac{v_i}{w_j}$$

We can rearrange to $*$:

$$\frac{\partial \mathcal{L}}{\partial v_j}\frac{v_j}{w_j}\frac{\partial \mathcal{L}}{\partial w_j} - \underbrace{\sum_{v_i \in \mathcal{P}_0(w_j)} \frac{\partial \mathcal{L}}{\partial v_i}\frac{v_i}{w_j}}_{\psi}$$

We define the sum as ψ. Also $v_i = w_i w_j$, therefore:

$$\psi = \sum_{v_i \in \mathcal{P}_0(w_j)} \frac{\partial \mathcal{L}}{\partial v_i}\frac{v_i}{w_j} = \sum_{v_i \in \mathcal{P}_0(w_j)} \frac{\partial \mathcal{L}}{\partial v_i} w_i$$

$v_j = w_j$, therefore $\frac{v_j}{w_j} = 1$ and we can summarize the gradient of v_j as:

$$\frac{\partial \mathcal{L}}{\partial v_j}\frac{v_j}{w_j} = \frac{\partial \mathcal{L}}{\partial v_j} = \frac{\partial \mathcal{L}}{\partial w_j} - \psi$$

6.2.3 Update rules - backward transformation

We will derive the formulas to transform from path-space back to weight-space given the new path values v^{t+1} for iteration $t + 1$ and the old weight values w^t. We assume, that the fixed weights (w_k) are set to 1.

First-layer skeleton weight (w_j): We have only one weight in the path w_j (which is not 1), therefore: $R(v_j) = \prod_{w_j \in p_j} r(w_j) = r(w_j)$

$$w_j^{t+1} = w_j^t \cdot r(w_j) = w_j^t \cdot R(w_j)$$
$$w_j^{t+1} = w_j^t \cdot \frac{v_j^{t+1}}{v_j^t}$$

Because $v_j = w_j$:

$$w_j^{t+1} = w_j^t \cdot \frac{v_j^{t+1}}{w_j^t}$$
$$w_j^{t+1} = v_j^{t+1}$$

First-layer non-skeleton weight (w_h) Computing w_h^{t+1} is trivial as it works the same as w_j. The weight ratio for every other weight than w_h weight in p_h is 0, therefore $R(v_h) = r(w_h)$ and $w_h = v_h$:

$$w_h^{t+1} = w_h^t \cdot r(w_h) = w_h^t \cdot R(w_h)$$
$$w_h^{t+1} = w_h^t \cdot \frac{v_h^{t+1}}{v_h^t}$$
$$w_h^{t+1} = v_h^{t+1}$$

Other non-skeleton weight (w_i) $R(v_i) = \prod_{i \in p_i} r_i = r(w_j) \cdot r(w_i)$

$r(w_i) = \frac{R(v_i)}{r(w_{j:p_i})}$

$$w_i^{t+1} = w_i^t \cdot r(w_i)$$

Insert $r(w_i)$:

$$w_i^{t+1} = w_i^t \cdot \frac{v_i^{t+1}}{v_i^t} \cdot \frac{w_{j:p_i}^t}{w_{j:p_i}^{t+1}}$$

Insert $R(v_i)$:

$$w_i^{t+1} = w_i^t \cdot \frac{v_i^{t+1}}{v_i^t} \cdot \frac{w_{j:p_i}^t}{w_{j:p_i}^{t+1}}$$

$v_i = w_i \cdot w_j$:

$$w_i^{t+1} = \frac{v_i^{t+1}}{w_{j:p_i}^t} \cdot \frac{w_{j:p_i}^t}{w_{j:p_i}^{t+1}}$$
$$w_i^{t+1} = \frac{v_i^{t+1}}{w_{j:p_i}^{t+1}}$$

6.2.4 Derivation constant path factor

Under the assumption that we don't set our skeleton weights w_k to 1, we need to compute the new update rules. We define the product of all w_k of the path p as $w_{s:p}$.

Path values:

Assuming v_p would be the path value for the case that $w_k = 1$, then the updated path value v'_p can be computed as follows:

$$v'_p = \underbrace{\prod_{w_j, w_i, w_h \in p} w}_{v_p} \cdot \underbrace{\prod_{w_k \in p} w_k}_{w_{s:p}}$$

$$v'_p = v_p \cdot w_{s:p}$$

Gradients

We take the base formulas from *Chapter 6.2.2*.

Gradient of v'_h: $v_h = w_h w_{s:p}$, therefore:

$$\frac{\partial \mathcal{L}}{\partial v_h} \frac{v_h}{w_h} = \frac{\partial \mathcal{L}}{\partial v_h} \cdot w_{s:p} = \frac{\partial \mathcal{L}}{\partial w_h}$$

$$\frac{\partial \mathcal{L}}{\partial v_h} = \frac{\partial \mathcal{L}}{\partial w_h} \cdot \frac{1}{\underbrace{w_{s:p}}_{\text{new factor}}}$$

Gradient of v_i: $v_i = w_i w_j w_{s:p}$, therefore:

$$\frac{\partial \mathcal{L}}{\partial v_i} \frac{v_i}{w_i} = \frac{\partial \mathcal{L}}{\partial v_i} \cdot w_{s:p} w_j = \frac{\partial \mathcal{L}}{\partial w_i}$$

$$\frac{\partial \mathcal{L}}{\partial v_i} = \frac{\partial \mathcal{L}}{\partial w_i} \frac{1}{\underbrace{w_{s:p}}_{\text{new factor}} \quad w_j}$$

Gradient of v_j: $v_j = w_j w_{s:p}$, therefore:

$$\frac{\partial \mathcal{L}}{\partial v_j} \frac{v_j}{w_j} = \frac{\partial \mathcal{L}}{\partial w_j} - \psi$$

$$\frac{\partial \mathcal{L}}{\partial v_i} = (\frac{\partial \mathcal{L}}{\partial w_i} - \psi) \underbrace{\frac{1}{w_{s:p}}}_{\text{new factor}}$$

New ψ:

$$\psi = \sum_{v_i \in \mathcal{P}_0(w_j)} \frac{\partial \mathcal{L}}{\partial v_i} \frac{v_i}{w_j} = \sum_{v_i \in \mathcal{P}_0(w_j)} \frac{\partial \mathcal{L}}{\partial v_i} \cdot \underbrace{w_{s:p}}_{\text{new factor}} w_i$$

Weight updates:

For each w_k, it's value stays fixed. Therefore, it's weight ratio remains constant at $r(w_k) = 1$. As a result, the values of w_k don't matter for the path ratio:

$$R(v_p) = \prod_{w \in p} r(w) = \prod_{w_j, w_i, w_h \in p} w \cdot \underbrace{\prod_{w_k \in p} w_k}_{=1}$$

Insert R in weight update:

General update rule:

$$w^{t+1} = w^t \cdot \frac{v^{t+1}}{v^t}$$

For each weight category:

$$w_j^{t+1} = w_j^t \cdot \frac{v_j^{t+1}}{v_j^t} = w_j^t \cdot \frac{v_j^{t+1}}{w_j^t w_{s:p}^t} = \frac{v_j^{t+1}}{w_{s:p}^t}$$
$$w_h^{t+1} = w_h^t \cdot \frac{v_h^{t+1}}{v_h^t} = w_h^t \cdot \frac{v_h^{t+1}}{w_h^t w_{s:p}^t} = \frac{v_h^{t+1}}{w_{s:p}^t}$$
$$w_i^{t+1} = w_i^t \cdot \frac{v_i^{t+1}}{v_i^t} = w_i^t \cdot \frac{v_i^{t+1}}{w_i^t w_j^t w_{s:p}^t} = \frac{v_i^{t+1}}{w_j^t w_{s:p}^t}$$

6.3 Weight fixing method

6.3.1 Reparametrization

To adapt the weight-fixing method for the use with already trained networks while still setting the fixed weights to 1, we introduce this adaptive rescaling algorithm.
It starts in the last layer and propagates further and further forward:
For each hidden node $\mathcal{N}$, rescale the in- and outgoing weights such that the outgoing fixed weight is one.

Because the fixed weights are on the diagonal, we rescale such that W_l' has ones on the diagonal. More formally:

We start in the last layer by multiplying D_{L-1}^{-1} to the right of the weight matrix:

$$y_L' = \sigma(\underbrace{W_L D_{L-1}^{-1}}_{W_L'} y_{L-1} + b_L)$$

For the other layers we additionally multiply D_l from the left:

$$y_l = \sigma(W_l y_{l-1} + b_l)$$

$$y_l' = \sigma(D_l W_l D_{l-1}^{-1} y_{l-1}' + D_l b_l)$$
$$y_l' = \sigma(\underbrace{D_l W_l D_{l-1}^{-1}}_{W_l'} y_{l-1}' + D_l b_l)$$

Proof of equivariance

The following proof shows, that multiplying D_l^{-1} from the right to a weight in layer l and D_l from the left of the weight in layer $i-1$ will in fact result in the same network function. We start with

$$y_i' = \sigma(W_i D_i^{-1} y_{i-1}' + b_i)$$

$$y_{i-1}' = \sigma(D_i W_{i-1} y_{i-2} + D_i b_{i-1})$$

We replace y_{i-1}' by it's definition:

$$y_i = \sigma(W_i D_i^{-1} \underbrace{\sigma(D_i W_{i-1} y_{i-2} + D_i b_{i-1})}_{y_{i-1}'} + b_i)$$

We can move D_{i-1}^{-1} into the ReLU function:

$$y_i = \sigma(W_i \sigma(D_i^{-1}(D_i W_{i-1} y_{i-2} + D_i b_{i-1})) + b_i)$$

$$y_i = \sigma(W_i \sigma(D_i^{-1} D_i W_{i-1} y_{i-2} + D_i^{-1} D_i b_{i-1}) + b_i)$$

And it will cancel out:

$$y_i = \sigma(W_i \underbrace{\sigma(W_{i-1} y_{i-2} + b_{i-1})}_{y_{i-1}} + b_i)$$

$$y_i' = \sigma(W_i y_{i-1} + b_i)$$
$$y_i' = y_i$$

Computation of D

We iterate through all layers $\{L-1, L-2, ..., 2, 1\}$ backwards. The matrix D and it's inverse D^{-1} can be computed as follows:

$$D_{l-1}^{-1} = |diag(\frac{1}{D_l W_l})|$$

$$D_{l-1} = |diag(D_l W_l)|$$

where $|M|$ is the element wise absolute value function.

Advanced skeleton selection

The presented rescaling algorithm can lead to numerical instabilities. The following algorithm selects the fixed weights in such a way, that minimizes their distance to 1 or -1. We use the Hungarian algorithm to determine the optimal permutation of our default mask:
We first construct a cost matrix C, where each entry represents the minimum absolute distance of a weight to 1 or -1:

$$C_{i,j} = \min(|DW_{i,j} - 1|, |DW_{i,j} + 1|)$$

Hungarian algorithm to find minimum cost-pairs between input and output. Optimize for:

$$\min_{\sigma} \sum_{i=1}^{n} C_{i,\sigma(i)}$$

Here, σ is a bijective mapping:

$$\sigma : \{1, ..., n\} \to \{1, ..., n\}$$

$\mathcal{M}$ contains the matching (input, output) pairs which will be used as fixed weights:

$$\mathcal{M} = \{(i, \sigma(i)) | i \in \{1, ..., n\}\}$$

Bibliography

[1] P.-A. Absil, R. Mahony, and R. Sepulchre. *Optimization Algorithms on Matrix Manifolds.* Princeton University Press, Princeton, NJ, 2008.

[2] Vijay Badrinarayanan, Bamdev Mishra, and Roberto Cipolla. Symmetry-invariant optimization in deep networks, 2015.

[3] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, 2016. http://www.deeplearningbook.org.

[5] J. Elisenda Grigsby, Kathryn Lindsey, and David Rolnick. Hidden symmetries of relu networks, 2023.

[6] Lei Huang, Xianglong Liu, Bo Lang, and Bo Li. Projection based weight normalization for deep neural networks, 2017.

[7] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10 (canadian institute for advanced research). 2009.

[8] Qi Meng, Shuxin Zheng, Huishuai Zhang, Wei Chen, Zhi-Ming Ma, and Tie-Yan Liu. $\mathcal{G}$-sgd: Optimizing relu neural networks in its positively scale-invariant space, 2021.

[9] Behnam Neyshabur, Ruslan Salakhutdinov, and Nathan Srebro. Path-sgd: Path-normalized optimization in deep neural networks, 2015.

[10] Mary Phuong and Christoph H. Lampert. Functional vs. parametric equivalence of re{lu} networks. In *International Conference on Learning Representations*, 2020.

[11] Eli Stevens, Luca Antiga, and Thomas Viehmann. *Deep Learning with PyTorch.* 2020.

[12] Mingyang Yi. Accelerating training of batch normalization: A manifold perspective, 2022.

Selbständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Bachelorarbeit wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Ort, Datum	Unterschrift