Informationflow in Deep ReLU Networks

David Hartmann

February 19, 2025

Informationflow in Deep ReLU Networks

Dissertation
zur Erlangung des Grades

„Doktor der Naturwissenschaften“

am Fachbereich Physik, Mathematik und Informatik
der Johannes Gutenberg-Universität
in Mainz

David Hartmann
geb. in Mainz

Mainz, den 19. Februar 2025

Berichterstatter

Univ.-Prof. Dr. Michael Wand
Institut für Informatik, Johannes Gutenberg-Universität Mainz

Univ.-Prof. Dr. Ernst Althaus
Institut für Informatik, Johannes Gutenberg-Universität Mainz

Datum der mündlichen Prüfung
19. September 2025

Lizenz

Abstract

Deep learning has proven its effectiveness in large parts of the scientific world. Even large-scale applications, especially text-to-image or text-to-text processors with billions of parameters, consist at their core of simple linear algebra, stacked and separated by non-linear functions. One such so-called activation function, Rectified Linear Unit (ReLU), is defined as the maximum of its argument with zero, effectively discretizing space into one of two cases: greater or smaller than zero. These mechanisms; a continuous basis (using linear algebra) and a discrete choice (using ReLU) seem sufficient to induce representations capable of tackling tasks such as Autonomous Driving or passing the Turing Test.

This thesis aims to explore the propagation of information in training deep ReLU networks, moving beyond the perspective of a solely continuous optimization process. By switching back and forth between these two ideas, continuous and discrete interpretation of the very same process, this work aims to explore different instances of the same underlying question: How does information flow from the dataset using the learning scheme through a deep network? One way to answer this question is to observe what discrete decisions a deep network implicitly makes during training and inference, leading to one of the key contributions of this work, which is to examine the activation patterns and their changes during training, enabling the analysis of architectural and optimization choices in a unified model of the training process. Using these insights, the thesis introduces ActCooLR, a proof-of-concept learning rate scheduler based on the introduced transition model of activation pattern changes. A second way to approach the question is to adaptively enhance the optimization process by incorporating additional discrete decisions using a stochastic number system during training, and monitoring optimization for this increasing difficulty.

Zusammenfassung

Deep Learning hat sich in vielen Bereichen der Wissenschaft als effektive Methode bewährt. Selbst groß angelegte Anwendungen, insbesondere Text-zu-Bild- oder Text-zu-Text-Modelle mit Milliarden von Parametern, bestehen im Kern aus einfachen linearen Abbildungen, gestapelt und getrennt durch nicht-lineare Funktionen. Eine solche nicht-lineare Funktion, die sogenannte Rectified Linear Unit (ReLU), ist definiert als das Maximum ihres Arguments mit Null, was effektiv den Raum in einen von zwei Fällen diskretisiert: größer oder kleiner als Null. Für Aufgaben wie autonomes Fahren oder das Bestehen des Turing-Tests scheinen diese beiden Mechanismen auszureichen: eine kontinuierliche Basis (unter Verwendung Linearer Algebra) und eine Diskretisierung des Raumes (unter Verwendung von ReLU).
Diese Arbeit beschäftigt sich mit der Propagation von Information beim Training von tiefen ReLU-Netzwerken und betrachtet diese Propagation auch jenseits des Blickwinkels eines rein kontinuierlichen Optimierungsprozesses. Durch Hin- und Herwechseln zwischen diesen beiden Ideen – kontinuierlicher und diskreter Interpretation desselben Prozesses – exploriert diese Arbeit an verschiedenen Mechanismen dieselbe zugrunde liegende Frage: Wie fließt Information vom Datensatz mithilfe der Optimierung in ein tiefes Netzwerk? Eine Möglichkeit, diese Frage zu beantworten, besteht darin zu beobachten, welche diskreten Entscheidungen ein tiefes Netzwerk implizit während des Trainings und der Inferenz trifft. Dies führt zu einem der Hauptbeiträge dieser Arbeit: der Untersuchung der Aktivierungsmuster und ihrer Veränderungen während des Trainings, was die Analyse von Architektur- und Optimierungsalgorithmen in einem vereinheitlichten Modell des Trainingsprozesses ermöglicht. Mithilfe der dadurch gewonnenen Einsichten, stellt die Arbeit “ActCooLR” vor, einen Proof-of-Concept Learning Rate Scheduler, der das vorgestellte theoretische Modell der Aktivierungsmusteränderungen nutzt. Als weitere Möglichkeit, die Frage der Informationspropagation zu beantworten, werden zusätzliche diskrete Entscheidungen mithilfe eines adaptiven stochastischen Zahlensystems in das Netzwerk eingebaut, um den Optimierungsprozess unter der resultierenden erhöhten Schwierigkeit zu überwachen.

Acknowledgement

The Zentrum für Datenverarbeitung (ZDV) of Johannes Gutenberg University and the Mogon II Super Cluster are recognized for their valuable computational resources. Special thanks are extended to Lambda, Inc., for their generous support during the final stages of this journey.

Contribution Statement

Parts of this thesis have been published in the following papers, where I contributed significantly as the primary author and initiator of these works, including the experiments, explanations and theories they presented, while benefiting greatly from the valuable collaboration, discussions, and insights of my co-authors. Theorem 1, as an alternative to the qualitative model given by Hypothesis 2, has been sketched by my co-author M. Wand.

The peer-reviewed papers contained in this thesis,

- **Studying the Evolution of Neural Activation Patterns during Training of Feed-Forward ReLU Networks (2021)**
  David Hartmann, Daniel Franzen, Sebastian Brodehl
  Front. Artif. Intell. 4:642374.
  doi:10.3389/frai.2021.642374

- **Progressive Stochastic Binarization of Deep Networks**
  David Hartmann, Michael Wand
  2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing - NeurIPS Edition (EMC2-NIPS).
  doi:10.1109/EMC2-NIPS53020.2019.00015

as well as the manuscript,

- **ActCooLR – High-Level Learning Rate Schedules using Activation Pattern Temperature**
  David Hartmann, Sebastian Brodehl, Michael Wand
  https://openreview.net/forum?id=yqj6q_eNTJd,

have been extended and connected in this thesis. The Code and hyperparameters for all experiments contained in this work can be obtained from the following repositories:

- https://github.com/JGU-VC/activation-pattern-analysis
- https://github.com/JGU-VC/progressive_stochastic_binarization
- https://github.com/JGU-VC/actcoolr
- https://github.com/da-h/simple_relu_visualization

Contents

Glossary

Thesis-Specific

Terms that are defined in this thesis.

Activation Cell (AC) is the subset in the input space of a neural network that implicitly gets mapped using the same single affine transformation internally.
See Definition 2.

ActCooLR (Activation Cooling based Learning Rate Scheduler). This scheduler uses activation change cooling to manage learning rates during training.
See Section 3.3 and Appendix A.4.

Activation Pattern (AP). A binary vector representing the decision of ReLU activations for a single layer.
See Definition 1.

Activation Pattern Entropy (APE) measures the randomness of activation patterns in a network's layer under a given dataset.
See Equation (3.9).

Activation Pattern Temperature (APT) assesses the layer's functional change per training step, indicating the sensitivity of the model at a given learning rate.
See Definition 6.

Extrinsic Projection is a projection of intrinsic coordinates into extrinsic coordinates in the context of a given (fixed) network layer.
See Appendix A.2.

Global Activation Pattern (global AP). Concatenates activation patterns across all network layers.
See Definition 3.

Learning Rate Sensitivity Analysis is a method to assess the intrinsic (that is parameter-free) sensitivity of a neural network.
See Section 3.2.3.

Progressive Stochastic Binarization (PSB) is a bijective mapping of floating point weights to a stochastic number system that allows for dynamic control of precision during inference.
See Section 4.2.

Datasets

Well-known datasets that are mentioned in the context of this thesis.

CIFAR-100 is an image classification dataset consisting of 100 classes of 32x32 color images introduced by Krizhevsky [86].

CIFAR-10 is an image classification dataset consisting of 10 classes of 32x32 color images introduced by Krizhevsky [86].

FashionMNIST is constructed as a successor of MNIST [93] and is an image classification dataset consisting of 10 classe sof 28x28 grayscale images for image classification introduced by Xiao et al. [182]

ImageNet (ImageNet Large Scale Visual Recognition Challenge, also ILSVRC) is the largest dataset in this thesis – a benchmark for image classification featuring over a million images across 1000 classes, introduced by Russakovsky et al. [146].

Speech Commands is a speech classification dataset of short spoken utterances for command recognition, introduced by Warden [177].

Tiny ImageNet is a simplified version of the ImageNet dataset, consisting of fewer, smaller (64x64) and more homogeneous images, spanning 200 classes, introduced by Le and Yang [90].

Networks

Frequently mentioned neural networks throughout this thesis.

F_1 is a small network with five hidden layers with 512 neurons each. The simple structure enables the visualization of the internal activation cells (see Section 2.2.4). The setup is similar to that of Keskar et al. [80], with batch normalizations omitted in this work unless specified otherwise.

FixUp is an initialization scheme and model adaption of the ResNet architecture that requires no normalization. [191]

M5 is a compact convolutional network architecture tailored to speech classification tasks for use in raw audio processing. [35, 162]

PyramidNet is a variant of the ResNet architecture that employs a pyramid-shaped structure, gradiually increasing the number of feature maps with depth for more efficient feature extraction. [57]

ResNet is a easy-to-train convolutional network family that revolutionized deep-network featuring residual connections that enable very deep networks, introduced by He et al. [60, 62].

VGG is a deep convolutional network with a uniform architecture and multiple layers, known for its early success in image classification. [158]

Deep Learning

Terminology commonly used in the domain of deep learning.

Batch Normalization is a layer used in deep neural networks for normalizing outputs and stabilizing gradients. [73]

Cyclical Learning Rate Schedule (CyclicLR) adjusts learning rates in cycles, enhancing robustness against local minima. [159]

Learning Rate (η) is a critical parameter in stochastic gradient descent methods that controls the "speed of learning" by specifying the *step-size* towards a minimum of the loss function for each iteration.

Momentum (β) is an optimization technique that accelerates convergence by averaging gradients. [134]

One Cycle Learning Rate Schedule (OneCycleLR) is a variant of cyclical learning rate scheduling that only employs a single cycle. [160]

Parametric ReLU (PReLU) is an activation function that scales the negative values by a trainable parameter and maps all positive values to themselves. [61]

Rectified Linear Unit (ReLU) is an activation function that maps negative values to zero and all positive values to themselves. [124]

Residual Connection facilitates learning of input-to-output differences ("residuals") of "layer blocks", enabling to increase network depth. [60, 62]

Stochastic Gradient Descient (SGD) the standard optimization method for neural networks, computing gradients with random mini-batches.

Step Decay Learning Rate Schedule (StepDecayLR). Reduces the learning rate at specific steps in training to successively fine-tune the model.

Weight Decay (λ). Decays weights to prevent overfitting, often combined with momentum. Exists in two variants. [107]

Acronyms

AC See Glossary: **Activation Cell**

AP See Glossary: **Activation Pattern**

APE See Glossary: **Activation Pattern Entropy**

APT See Glossary: **Activation Pattern Temperature**

global AP See Glossary: **Global Activation Pattern**

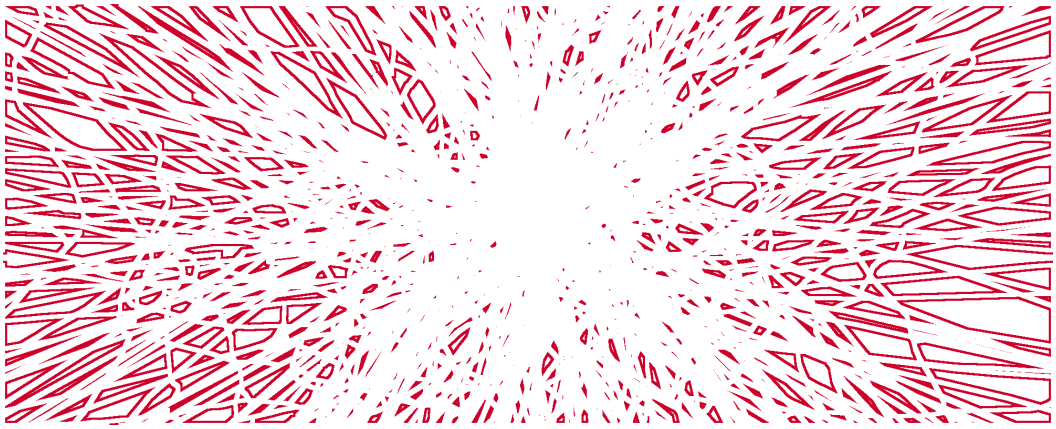
PReLU See Glossary: **Parametric ReLU**

PSB See Glossary: **Progressive Stochastic Binarization**

ReLU See Glossary: **Rectified Linear Unit**

SGD See Glossary: **Stochastic Gradient Descient**

1 Introduction



With neurons and layers to trace,
This thesis set out to embrace,
ReLU's sharp turn,
And patterns to learn,
Unraveling the network's pace!

— ChatGPT,
"Generate a limerick of the following text:"

Deep learning, often grouped under the umbrella term A.I. (Artificial Intelligence), has shown to match or exceed human performance in various tasks such as image recognition [40], natural language processing [21], reasoning benchmarks [38, 46, 178], and has paved the way [18] to become the cornerstone of modern technology that it is today [114]. Deep learning denotes a family of methods [92] that enable the transformation of arbitrary input data through a sequence of layers and non-linear transformations (*deep neural networks*) into a meaningful representation – here referring to a form of representation more tractable than the original raw data. The dynamics of how the noisy information of data propagates through the network and shapes the model variables that parametrize the internal representation remains a topic of research. Despite this, the performance of these models continues to improve annually, with the most consistent factors being increased computational resources and a larger amount of cleaner data. [92]

The title of this work, *Informationflow in Deep ReLU Networks*, emphasizes the idea to better understand how information propagates through deep neural networks during training. This works seeks to investigate the underlying mechanisms that govern this information flow in deep neural networks by analyzing the statistical properties of internal representations. Specifically, this thesis focuses on the role of Rectified Linear Unit (ReLU) activation functions in deep learning and examines how such a simple operation – computing the point-wise maximum with zero – enables networks to learn complex hierarchical representations.

Motivation and Objectives

The primary goal of this thesis is to clarify the described information flow, thereby enabling a better understanding of the entire training process. Emphasis lies on studying how the ReLU activation in deep networks shape the internal structures and to provide guidance for designing more effective architectures and optimization methods. At its core, this work examines aspects of deep learning methods by isolating their inherent core mechanisms and assessing their featured training behavior one-at-a-time. The underlying research strategy beyond the written aspect of this work is bifurcated. On one end of the spectrum lies the goal of studying the continuous part of optimization and how it influences the binary decision that the ReLU activation implicitly makes. On the other end of the spectrum, the research intends to study how these discrete aspects within the continuous process inform and influence the optimization process.
A unifying element is the ReLU activation function which enables a discussion ranging from the discrete decisions on classical deep network architectures for practical relevance and applicability of the study to more fundamental insights into the interaction between non-linear and linear aspects of deep learning, and includes also over simplified (bit-level) deep networks using alternative number systems.

The thesis primarily concentrates on analyzing the dynamic internal non-linear structures in deep neural networks (called activation patterns) to gain a better understanding of their training behavior, and ultimately contribute to the improvement of the stability and the consistency of training processes.

From Coarse to Fine, the Scope of this Work

The thesis starts with foundational context, a short introduction of deep networks, their optimization using stochastic gradient descient (SGD) and basic techniques that have enabled stable and quick training of deep networks on various tasks.
To frame the discussion, the thesis continues to highlight a central tension in deep learning: the interplay between non-linear and linear components. While optimization occurs on locally linear subspaces (via gradient-based methods), inference relies on non-linear activation functions to model complex patterns.
This duality, the direct continuous optimization versus implicit discrete optimization, serves as the core investigative theme of this work. The latter view is tackled practically by incorporating a number system with a control parameter steering between more continuous and more discrete calculations (bit-networks), introducing even more non-linearity into the process. The former view is explored by examining the role of activation functions such as ReLU in the traditional training setup, as outlined below:

The ReLU activation is a simple switch between two functional states, either acting as the identity or as the zero function for its input. The training process of networks with ReLU activations can be divided into two different training phases, with one being an initial non-linear phase where the network learns to extract meaningful features from the data, and a final more linear phase where the network fine-tunes its weights without much structural change. During the mostly non-linear initial phase, the network's weights change significantly and training can vary in terms of stability if not accounted for by a specific "warm-up" training phase.
In the linear phase, the changes to the weights are smaller and more akin to linear optimization as the optimization saturates and the network converges towards its final state. Empirical studies on the two described training phases exist and provided already some insights into the entire training process, however, especially the transition period between these two phases lacks an extensive discussion in the literature.

This work unifies the analysis for the entire training process, showing how it is influenced by factors such as the learning rate, activation functions, the network architecture, optimization techniques used, and thus enables also research on the transition period between these two states of optimization. The theoretical analysis of this training process, driven by experimental observations, provides a descriptive model for understanding the role of ReLU non-linearities in the training of deep networks, and leads to a constructive optimization method that utilizes the insights gained.

Overview of this Thesis

The remaining contents of this thesis outlined in more detail:

Chapter 2 – The Role of Relu: Starting with a short overview of basic deep learning concepts and techniques, the chapter proceeds to introduce the concepts of activation patterns and activation cells, representations of the non-linear structures used inside neural ReLU networks. Limiting input space to two input dimensions enables to visualize and to quantify how network parameters such as width, bias, depth, batch normalization, residual connections and training affect the implicit non-linear decisions made in a neural network.

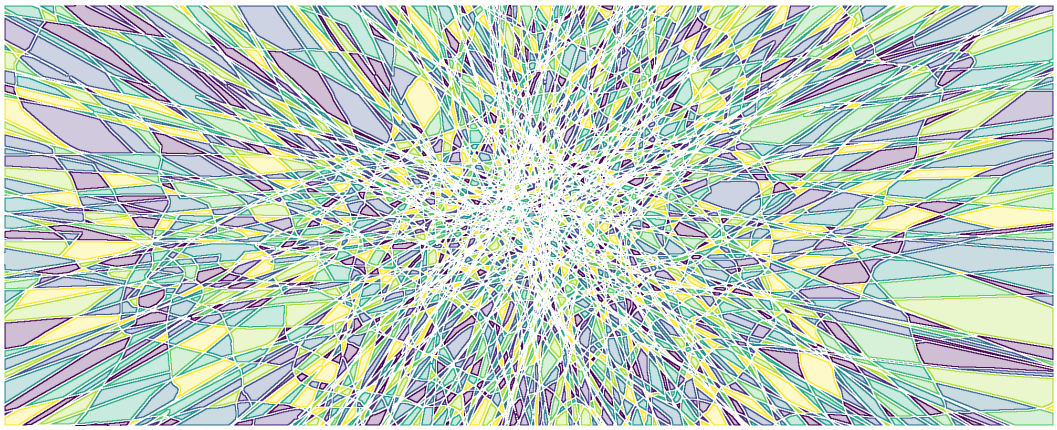
Chapter 3 – Activation Patterns During Training: Provides an in-depth analysis of activation patterns during training of deep networks. Using metrics that involve distributions and changes of activation patterns during training, this chapter explores the structural complexity and sensitivity of neural networks during optimization and connects the evolution of activation pattern distributions to hyperparameters and the effects of common architectural choices and training methods. For the primary optimization method of deep learning, SGD, the learning rate is a critical hyperparameter that controls the step size of a single optimization step. Exploring various learning schedules, this chapter derives a statistical model of activation pattern changes and develops an automatic learning rate scheduling algorithm to adapt the learning rate based on the network's state, adapting to changes in the training behavior and adhering to desired non-linear speed of search in function space.

Chapter 4 – Deep Networks of Lower Computational Complexity: This chapter covers deep learning models that employ discrete calculations. As a rather technical aspect of the idea of non-linear optimization, it focuses to assess wether deep learning also performs well if even more of the linear aspects are shifted towards non-linearity. By constructing a number system with a parameter that controls the level of "continuity" in calculations, the effect of that parameter on both training and evaluation results is investigated.

Chapter 5 – Conclusion: Provides the summary and overview of the findings, consolidating all themes covered that revolve around understanding and optimizing the training process of deep neural networks.

Appendix A: The appendix outlines ideas that, while promising, were not fully developed for inclusion into the main part of this thesis.

2 The Role of ReLU



For deep learning to be sound,
Functions non-linear must abound,
ReLU's a good pick,
For speed that is quick,
And network weights that are profound.

— ChatGPT,
"Generate a limerick of the following text:"

The core operations of deep learning are *linear mappings separated by non-linear functions*. The *extensions* of this operation; *convolutions*, *residual connections*, *intra-network normalizations* (see Section 2.1) to name a few, are the essential tricks that enabled deep learning based solutions in real-world scenarios. While the details of these core operations have changed over time, such as the choice of a specific non-linear function, this setup remains the building block of deep learning. One non-linearity stands out in simplicity and it's utility as it has "helped to obtain best results on several benchmark problems across many domains" [150], the *Rectified Linear Unit (ReLU)*-activation.

This thesis analyzes the role that ReLU-activations play in the function space of deep networks, investigates the expressivity they introduce, and, re-interpretes their locally linear optimization by observing effects of the non-linear parts of function space in isolation.

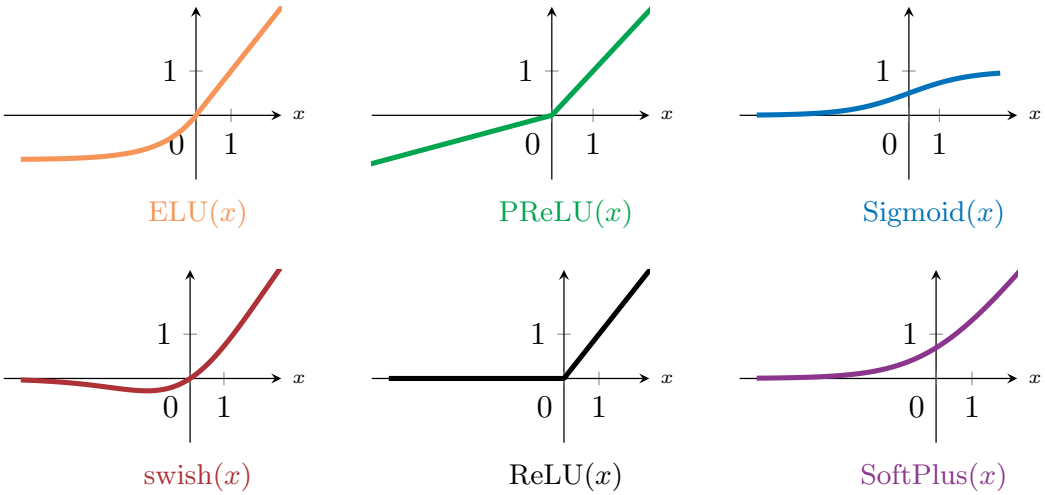


Fig. 2.1.: Comparative Visualization of activation functions with similar features as the ReLU activation. The graphs show the functions ELU, PReLU, Sigmoid, swish, ReLU and Softplus. The main differences include differentiability, limit behavior, and transition behavior.

Why concentrating on ReLU-activations?

The ReLU activation has played a significant role in numerous benchmark-breaking contests over the years. Probably the most notable appearance early on during the ImageNet Large Scale Visual Recognition Challenge in 2010 (referred to as ILSVRC-2010) where it was employed by Krizhevsky et al. [87]). ReLU has been *the* activation function of choice for years and has various successors nowadays (See Figure 2.1 for an overview of some variants). Some of these resemble ReLU as smoothened versions, specialized for efficient evaluation or targeting more stability during training. The "classical" form of the ReLU activation has become mostly obsolete in modern applications like large-scale language models. These models use smoother constructs such as SwiGLU [155], an activation function involving the swish activation (see Figure 2.1) in its gated form.

Yet, the simplicity of the basic form provides the main argument for a deeper inspection:

$$\mathrm{ReLU}\ (x) := \max(x, 0)\,.$$

The ReLU activation is defined as the identity function for positive values x and it is defined to map to zero for all negative values (see Figure 2.1). Since benchmark contests such as the mentioned ILSVRC-2010 competition have shown that training deep networks of high quality is possible with just this basic form of an activation function, a switch between the identity and zero, we take a closer look at ReLU.

Additionally, the ReLU activation simplifies the inspection of deep networks, as it reduces composites of linear functions with non-linear activations in-between to just a choice of a set of linear functions – more details about this in Section 2.2.
To conclude, if such a simple function, merely a switch between identity and zero, is enough for significant progress in the field of machine learning, and it simplifies analyses, an in-depth analysis of the role that this activation plays in deep learning could be a valuable endeavor. For instance, while the discreteness is often considered the main reason for a deep network's expressivity, it is also possibly the discreteness that may be a key factor for the training of networks – and if so, possibly relaxed versions such as those shown in Figure 2.1 use the same mechanisms implicitly.

Goals and Structure

The broad scope of this chapter is to identify implicit effects that the decisions induced by ReLU activations have on the expressivity of deep networks. To get a more comprehensive picture, this chapter aims to lay the ground for the base idea for this thesis, namely to consider the non-linear components of deep networks separated from their linear components. Thus, the first goal is to identify how these decisions impact the function space of deep networks structurally, that is, without taking the dynamics of optimization into account at first.

The contents of this chapter in more detail:

1. **Section 2.1 gives a short overview of the architectural features and optimization techniques used in the experiments shown in this thesis.** Effectively, a short tutorial on deep learning, this includes feed-forward networks, convolutions, residual connections, batch normalization, the datasets, the network architectures and the experimental setups used throughout this thesis. Accompanying the tutorial aspect, the section contains a short explanation of commonly used simple visualization techniques that aim to get insights into the inner working of neural networks.

2. **Section 2.2 visualizes the premise of implicitly handling nested decisions when training fully connected ReLU-networks.** After defining the concepts of activation patterns and activation cells, this section proceeds to visualize how fundamental network parameters such as width, depth, bias and architectural choices such as batch normalization affect the function family resulting from the ReLU activation.

2.1 A Very Short Tutorial on Deep Learning

Deep learning has emerged as the de-facto standard tool for numerous areas of research. Opinions on what single driver has fueled success in deep learning the most are unanimous on one point: computational power in the form of GPUs (as referenced by LeCun et al. [92], Raina et al. [139], and Pandey et al. [131]). Still, the transformation of deep learning from an experimental proof-of-concept technique applied to "lab-datasets" [93] into a practical and effective tool used in commercial real-world applications [33] can (apart from raw compute) be attributed to various components used together. Still used today in modern transformer and/or diffusion architectures, in one way or another, this section will introduce each of these parts in its simplest form.

Feed-forward Network

A feed-forward network $F : \mathbb{R}^{d_1} \to \mathbb{R}^{d_{L+1}}$, is a concatenation of layers $f_l : \mathbb{R}^{d_l} \to \mathbb{R}^{d_{l+1}}$, $1 \leq l \leq L$, separated by arbitrary non-linear functions σ, called *activation functions* (also referred to as *non-linearities*).
In this work, $F : \mathbb{R}^{d_1} \to \mathbb{R}^{d_{L+1}}$ is called a *feed-forward network*, if it is of the form:

$$x \mapsto F(x) := (f_L \circ \sigma \circ f_{L-1} \circ \sigma \circ \dots f_2 \circ \sigma \circ f_1)(x). \tag{2.1}$$

Here, f_i are called *hidden layers*, and f_L is called the *output layer*. The number of layers L is called the *depth* of the network, and the dimensions d_i are called the *width* of the network. The output of the network, $F(x)$, is called the *prediction* or *forward pass* of the network.

While layers f_l can vary in structure (e.g. nested architectures), parts of this thesis focus on a simpler case: a *simple feed-forward network* is defined as one where all its layers f_l are affine transformations,

$$x \mapsto f_l(x) := x^T W_l + b_l, \tag{2.2}$$

where the matrices $W_l \in \mathbb{R}^{d_l \times d_{l+1}}$ are called the *weights* and the vectors $b_{d_l} \in \mathbb{R}^{l+1}$ are called the *biases* of the network. Sometimes, weights and biases together are called filters, where the output dimension denotes the number of filters used in that layer. The components of the layer's outputs, $f_l^{(j)}, x \mapsto f_l(x) \cdot e_j$, (the vector

e_j denoting the standard unit vector with a 1 at its jth position) of any layer in a simple neural network are called *neurons* or *filters*[1].

Non-linearities, or activation functions, are univariate real-valued functions, $\sigma : \mathbb{R} \to \mathbb{R}$ with the property that they are non-linear. As described at the start of Chapter 2, this work focuses on the Rectified Linear Unit (ReLU) activation function, defined as:

$$\mathrm{ReLU} : \mathbb{R} \to \mathbb{R}_+, x \mapsto \mathrm{ReLU}(x) := \max(x, 0).$$

Also used in this work is an extension of the ReLU activation, called *parametric ReLU (PReLU)*. Unlike ReLU, which sets all negative values to zero, PReLU specifies the slope for negative inputs using a trainable parameter a. This modification enables the differentiation of the slope parameter, enhancing flexibility at the cost of additional weights,

$$\mathrm{PReLU}_a : \mathbb{R} \to \mathbb{R}, x \mapsto \mathrm{PReLU}(x) := \max(x, a \cdot x).$$

Optimization using Stochastic Gradient Descent

Randomly applying affine transformations and activation functions in a feed-forward network is only unlikely to result in a functional solution capable of solving the specific task at hand. What thus is needed is a way to choose which weights lead to such. The idea to accomplish this is to optimize a first guess by successively adapting the list of weights and biases of the model so that the resulting adapted function performs better in expectation compared to the previous "guess". To evaluate if the quality improved, a predefined quality measure Q, which is also called *loss* or *energy function* is used,

$$\underset{\text{weights, biases of } F}{\arg\min} \quad \mathbb{E}[Q(x, F(x))].$$

The most well-known and used optimization technique used in deep learning is *stochastic gradient descient (SGD)* and requires the Q to be differentiable. The toughest part of finding the optimum is the non-linear nature of the network F. Assuming Q is differentiable, we perform a *backward pass* to compute the gradients $\frac{\partial Q}{\partial w}$ of the loss Q with respect to each of the weights w in the network F. These gradients measure the local sensitivity of the loss to small changes of each of these weights. Gradient descent uses these gradients to improve the loss gradually (*gradient descent step*) by applying the rule

$$w \leftarrow w - \eta \frac{\partial Q}{\partial w}, \eta > 0 \tag{2.3}$$

[1] While the term *filters* is mostly used in the context of convolutional neural networks, this thesis uses both terms interchangeably

for each parameter w called *weight* and the hyperparameter η called learning rate. The remaining question is how to compute these gradients in practice, leading us from gradient descent to SGD; instead of computing the "global" gradients of the problem, i.e. for all data points x, we estimate these gradients for a subset of data points, called a *batch*. Smith et al. [161] have shown that the gradient noise introduced by the that way computed estimates lead to implicit regularization that may be responsible for the improved generalization performance of SGD compared to Gradient Descent (GD). One problem discussed in this work (in Section 3.3) and still subject of research as of today is how to choose the learning rate η best to obtain good performance.

Momentum: The optimization rule eq. (2.3) alone would not lead to a good optimization performance, yet. Two more tricks are employed to increase likelihood of success significantly. The first is *momentum*, which is remeniscient of the physically inspired motion of a heavy ball [134]. It's idea is to help the optimization in more shallow regions of the loss landscape by computing the *velocity* of each weight and applying it weighted by a parameter called *momentum*, $\beta \geq 0$. Thus, the rule above is adapted to

$$\begin{aligned} v &\leftarrow \beta v + \frac{\partial Q}{\partial \theta}, \\ w &\leftarrow w - \eta v. \end{aligned} \tag{2.4}$$

Weight Decay: The second technique is called *weight decay*. It's idea comes from penalizing large weights by adding a $\lambda \cdot l_2(w)$-regularization term to the loss, steered using the parameter $\lambda \geq 0$. This idea exists in two variations: weight decay that depends on the chosen learning rate η and weight decay that is detached from that learning rate η (Loshchilov and Hutter [107] provide a discussion). In practice this is implemented by either applying either of the following additional rules after the actual optimization step:[2]

$$\begin{aligned} \textbf{SGD / L2 Weight Decay:} \quad & w \leftarrow w - \eta \cdot (1 - \lambda) \cdot w \\ \textbf{SGDW / Decoupled Weight Decay:} \quad & w \leftarrow w - (1 - \lambda) \cdot w. \end{aligned} \tag{2.5}$$

Cross-Entropy Loss: A common example of a loss function, which is frequently used in the analyses of this work, is classification. In this case, each data point x is assigned to a class $y(x)$. The objective for the F is to predict the categorical $y(x)$ (without loss of generality, assumed to be an integer, allowing $y(x)$ to be used as an

[2] This is because the gradient of the l_2 loss is simply the weight itself. The factor 2 is implicitly included in the choice of the parameter λ.

index) based solely on x. To define this in a differentiable manner, the *cross-entropy loss* function can be employed:

$$\mathrm{CEQ}(F) = \mathbb{E}_{x\sim p}\left[-\log q_F(x)\right],$$

where $q_f(x)$ is the estimated probability based on F's output that x belongs to class $y(x)$. Let the network $F(x)$ returns as many dimensions as there are classes in the given problem. To interpret the output of F as a probability measure, we often use the *softmax* function to enable differentiation. This results in the following expression for the probability of the correct class $y(x)$ given the input x:

$$q(x) = \frac{\exp F(x)_{y(x)}}{\sum_i \exp F(x)_i}, \tag{2.6}$$

where $F(x)_i$ represents the i-th component of F's output vector.

Initialization: Optimization of such a feed-forward network becomes increasingly difficult with increasing number of layers. Early work has implemented greedy layer-wise initialization schemes [64] that required a pre-training phase using the to-be-trained dataset. A data-independent scheme has been introduced by He et al. [61] enabling training of deeper networks without the need to "gauge" the network on each new dataset. The idea is to prevent reducing or magnifying layer signals (or gradients). Specifically, requiring unit-variance of layer signals using the ReLU activation function leads to the initialization scheme

$$w_l \sim \mathcal{N}\left(0, 2/\mathrm{D}_l\right).$$

Similar schemes have been derived for other activation functions, too, and are pre-built into modern deep learning frameworks (see for example [28]).

Convolutional Layer & Convolutional Network

Convolutional neural networks, or short CNNs, are feed-forward networks that include *convolutional layers*. Their key idea is to apply the same set of feed-forward filters to different "local regions" of the input, effectively capturing spatial information of 2D input.[3] 2D input refers to data from input space $\mathbb{R}^d$ that can be arranged into a three-dimensional tensors with dimensions corresponding to width W, height H and channels C. Thus, the dimension of the input space has dimension given by $d = W \times H \times C$ for integers W, H, C. A convolutional layer is then defined as

[3]While CNNs can be defined for arbitrary many dimensions, we will concentrate here on two-Dimensional input data as an example.

a feed-forward layer that operates on local regions (subspaces) $\mathbb{R}^{w\times h\times C}$ with the mapping $f : \mathbb{R}^{W\times H\times C} \supset \mathbb{R}^{w\times h\times C} \to \mathbb{R}^{c'}$, where c' denotes the output dimension. This mapping is applied to subsets of the input space $\mathbb{R}^{W\times H\times C}$, where both of the following conditions are true:

- The extracted patches maintain the spatial structure of the original input,

$$\mathbb{R}^{W\times H\times C} \ni x \mapsto \hat{x} \in \mathbb{R}^{w\times h\times C}, \quad \text{where} \quad \hat{x}_{i,j,k} = x_{i+a,j+b,k},$$

  for offsets $a, b \in \mathbb{N}_0$.

- The result of $F(x)$, a vector with c' values for the input subset with offsets a, b is placed at the output position indexed by $(a, b, i), 0 \leq i \leq c'$, ensuring consistency with indexing conventions. This process alone would result in the output dimensions shrinking at each layer. To counteract this, optional *padding* adds zero-valued "helper dimensions" at the borders of input dimensions.

That way, a convolutional layer can be seen as a specialized feed-forward network that shares weights across different spatial locations. Note that the local regions often overlap[4] with neighboring copies.

Batch Normalization

Batch normalization is a specialized type of layer to ensure its output is normalized to have an expected mean of zero and standard deviation of one. It is assumed that batch normalization has been an important factor in the history of deep learning [69], since it enabled training deeper networks and provided a solution for both the vanishing and exploding gradient problem [73], thus reducing the potential optimization divergences.

It's definition is:

$$\mathrm{BN}_{\beta,\gamma}(x) := \frac{x - \mathbb{E}[x]}{\sqrt{\mathrm{Var}[x]}} \cdot \gamma + \beta,$$

where $\mathbb{E}$ denotes the mean, Var, the variance, and, $\gamma, \beta \in \mathbb{R}^d$ are trainable parameters. Typically, networks are used to compute results for multiple data points at the same time, called *batches*. These batches are used for the statistics computed in the mean and the variance used above. However, along these batch-wise statistics of mean and variance, two additional parameters μ and σ are measured to estimate the mean and variance over the full training dataset. In practice, this is achieved by

[4] Overlapping occurs only when the striding value is less than the window size (w, h). The striding value determines the index shift for neighbouring applications.

using exponential moving averages (*EMA*), by adapting a first guess $\mu_0 = \mathbb{E}_i[x]$ (of the first batch $i = 0$) and using the update rule

$$\mu_{i+1} \leftarrow \mu_i \cdot \alpha + (1 - \alpha) \cdot \mathbb{E}_i[x],$$

where α is a tuneable parameter, adjusting the trade-off between speed of convergence and accuracy of the value compared to the true mean and variance over the whole data base.

Using the batch-wise estimates as defined above is referred to batch normalization's *train* mode.
In test mode, unlike train mode, batch normalization utilizes the moving average measurements when calculated without any statistics as follows,

$$\mathrm{BN}(x) = \frac{x - \mu}{\sigma} \cdot \gamma + \beta,$$

ensuring predictable results independent from number of data points used and potential distribution shifts of samples used for computations. In the case of 2D inputs (of dimension $W \times H \times C$), batch normalization typically compute separate statistics and EMAs for each C component.

Note that placing linear layers (or convolutional layers) before batch normalization nullifies the effect of the linear layer's bias term, as batch normalization subtracts the mean from intermediate results. In this case, β takes the role of the effective bias term and thus will also be called bias in this thesis.
Note that various forms of batch normalization exist, some with their own name, which are determined by the dimension of the statistics used for estimates. Still in the 2D example, if the statistics measured has dimension C, the layer is called batch normalization. However, if the dimension is $W \cdot H$ it is called *InstanceNorm* [173] and computes statistics on a per-pixel level. GroupNorm [181] is another known variant, that measures statistics of dimension $c \leq C,\ C \mod c = 0$, but implicitly applies copies to all dimensions a also to the dimensions at $b \cdot c + a$. Transformer-architectures require a specialized version as instance dimensions typically differ in a single batch, called *LayerNorm* [175] computing the statistics in all dimensions together.

Many studies have examined the significance of the batch normalization layer in deep learning. We will show in Appendix A.1 that a bias value of about 0 (which is typically used in deep learning frameworks when using batch normalization) maximizes activation pattern entropy (APE) in the initialization only locally.

Tab. 2.1.: Summary of benchmark datasets utilized in this thesis, spanning visual and auditory modalities. Listed characteristics include training/test sample sizes, class diversity, and input dimensions. The data complexity ranges from low-resolution grayscale images to high-dimensional audio samples.

Dataset	Data Size (Train/Test)	Classes	Data Dimensions
FashionMNIST [182]	60,000 / 10,000	10	$28 \times 28 \times 1$ (gray scale images)
CIFAR-10 [86]	50,000 / 10,000	10	$32 \times 32 \times 3$ (RGB images)
CIFAR-100 [86]	50,000 / 10,000	100	$32 \times 32 \times 3$ (RGB images)
Tiny ImageNet [90]	100,000 / 10,000	200	$64 \times 64 \times 3$ (RGB images)
ImageNet [146]	$\sim$1.2 million / 50,000	1000	various dimensions (RGB images)
Speech Commands [177]	105,829 / 11,005	50	1×16000 (audio samples)

Residual Connections

Prior work has analyzed *residual connections* and explored their role in improving performance. [10, 129]. The technique that is also known under the term *skip-connections* increases training speeds and maximal depth of networks. First proposed by He et al. [60], their core idea is to mitigate information loss across networks layers via additive skip connections,

$$x \mapsto x + f(x),$$

where the network learns the *residual* to the original data. This way, in the event of a layer- or neuron-*collapse* that maps all inputs to nearly the same outputs, information can still reach subsequent layers.
This method effectively distributes information processing across multiple layers rather than isolating the said information within individual layers.

On the Problem of Visualizing Deep Networks

Deep learning models without activation functions do not perform as well as those with them. Consequently, it is a natural endeavor to seek a deeper understanding of the internal structures generated by these activations. The question is how these functions assist the network in identifying structures in complex tasks, such as recognizing a dog or a tennis ball in an image (Figure 2.2(a)). To gain insight into

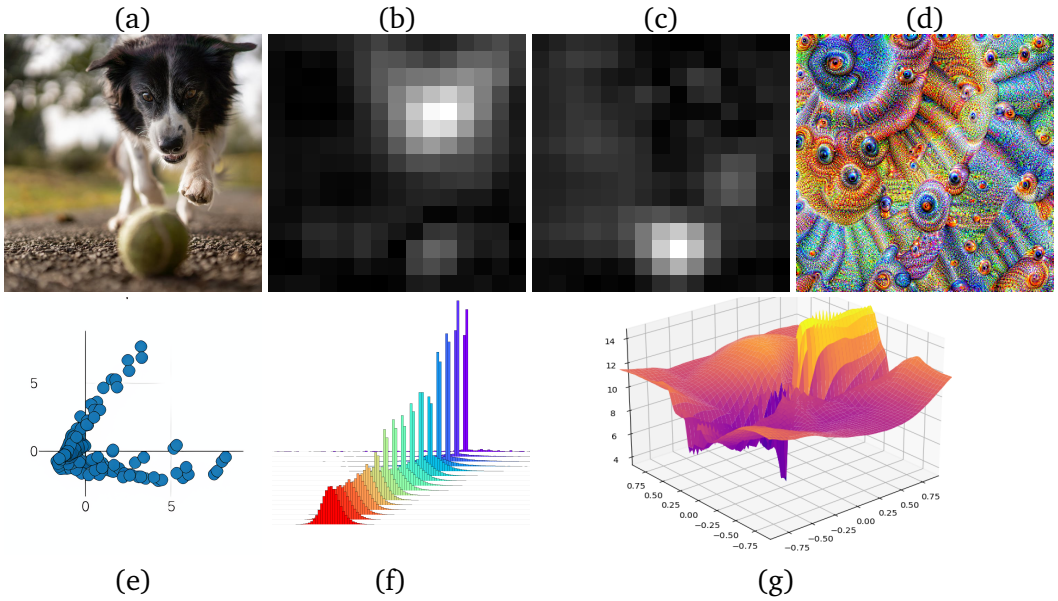


Fig. 2.2.: Using the example of a VGG-19 network, pre-trained on ImageNet, several simple neural network visualization techniques from the literature aim to visualize its inner workings: (a) tilt shot photo of a dog chasing a ball (CC0, Pixabay, `https://www.pexels.com/photo/tilt-shot-photo-of-dog-chasing-the-ball-1562983/`), (b) activation map for neuron #169 of the VGG's last convolutional layer (c) activation map for neuron #468 of the VGG's last convolutional layer, (d) generated deep dream of VGG's layer #27, (e) PCA of all neurons of the VGG's last convolutional layer, (f) weight distributions colored training steps, (g) loss landscape of two random directions for a VGG network.

the process of decision-making, it's helpful to begin with visualization techniques that offer an initial glimpse into the underlying computations.

Ironically, the visualization of deep neural networks is a complex task itself, particularly, when intending also to inspect the dynamics of training. The main problem lies in the ever-changing nature of a neural network. First, the network's effective computational graph may change with every input data because ReLU-activations stop data to succeeding layers depending on their sign, that is, negative numbers are mapped to zero.[5] The network's input typically has a canonical way of representation, for instance, pixels of an image, a time series window, or arbitrary data dimensions with predefined semantics, but succeeding layers in deep networks intrinsically do not provide such. Additionally, for deep networks to work effectively, they require large dimensions for intermediate results; a CNN, for instance, typically

[5]Technically, the computational graph does not change as each input is processed similarly. The meant changes are the neurons that change their activation states, depending on the given input. Results may become negative numbers, causing them to get deactivated in the computation. When this happens, the process can be seen as forming a new computational graph, as the zeroed values alter the subsequent calculations.

uses between 128 and 512 dimensions for internal representations.
One could, of course, use classical approaches for visualization to get a better understanding of inner mechanisms: For instance, traditional dimensionality reduction techniques, topological data analyses or interactive inspection using linking and brushing [105] have been used in the past to analyze how specific data points affect training, internal concepts in the network, and vice versa. Thus, early work on neural networks used *Principal Component Analysis* [132] to inspect interim results during training more systematically [152]. Figure 2.2(e) shows the PCA of an activation vector in a VGG-19 pre-trained on ImageNet for the input image shown in Figure 2.2(a). Nowadays, deep learning monitoring frameworks such as *Tensorboard* (part of the *Tensorflow* framework [113]) provide *t-distributed stochastic neighbor embedding* [111], or short, *t-SNE* as an inspection mechanism of the high-dimensional interim layers or weights.
However, the field of visualization still lacks interpretability, and this issue remains being researched in areas like *explainable AI* [148], which focuses on developing tools to automate the generation of explanations. The absence of semantic information is the biggest problem in working with the high-dimensionality of intermediate spaces utilized by deep networks as they often give only limited insights. Consequently, traditional dimensionality reduction techniques are less frequently employed in practical applications beyond monitoring.

One idea to circumvent the problem of registering possible semantics to interim results is to treat deep networks solely as "black boxes", focusing on how inputs get mapped to respective outputs or activations. One such approach, specialized to the task of image classification, are *class activation maps* that utilize the sliding-window nature of CNNs to determine which parts of an image weigh the most for the final decision of classifying the input. For instance, Figure 2.2(b) visualizes the class activation map for the label "dog" and Figure 2.2(c) the class activation map for the label "ball" for the input image shown in Figure 2.2(a). Similarly, it is also possible to visualize the interim layers by inspecting the pre-activation strengths of individual neurons in an image when using CNNs, showing which parts of an image fit the respective filter saved in the neuron least or most. (The reader is advised to check the interactive visualization by Olah et al. [128] for an informative example of this idea.) For instance, Figure 2.2(d) shows the pre-activation map for layer 49. One notable alternative concept is "deep dream" [121], an approach that projects the intermediate representation of a pre-trained network into its input. The core idea is to optimize the input image w.r.t. a single neuron or layer instead of optimizing the network weights w.r.t. the loss. This leads to a visualization of what the neuron has learned. (Figure 2.2(d) shows the deep dream of layer 27 of the same VGG-19 used

earlier.
The problem with the class activation approach or deep dream is that while it does show a network in action, and gives more information of their inner working beyond the "black box" view, the activations are rather importance maps of a single image, not revealing constructively what a network has actually stored, and neither what could be done to improve training or architectures. More constructive methods try to improve training stability by better understanding the structural and stochastic properties of a network, such as weight distributions (Figure 2.2(f)), embeddings of interim results (Figure 2.2(e)) during training, or, by checking loss landscape attributes such as smoothness before or during training (Figure 2.2(g)).

2.1.1 Datasets & Experimental Setups

Another driving factor in the effective evaluation of novel deep learning techniques has been the availability of benchmark datasets and standardized experimental setups. The datasets used, are publicly available and widely adopted by the research community, and the experimental setups include protocols, hyperparameter settings and evaluation metrics to enable comparisons of different techniques under consistent conditions. To improve comparability across the experiments in this thesis, the experiments employ the same datasets and setups as detailed in the following section.

Datasets

Among the dataset employed by the research community, vision tasks such as image classification have been particularly hotly contested, and websites such as `https://paperswithcode.com/sota/` have allowed researchers to see which innovations have truly led to performance breakthroughs on these tasks.
Besides toy datasets, this work also focuses on image classification to benchmark methods, in particular the datasets FashionMNIST [182], CIFAR-10 [86], CIFAR-100 [86], Tiny ImageNet [90] and ImageNet [146]. The Speech Commands [177] dataset serves as a validation of the constructive method presented in Section 3.3.2. Table 2.1 gives a high-level overview of each dataset's dimensions.

In more detail and sorted by increasing difficulty:
FashionMNIST is a dataset developed as a drop-in and more challenging replacement of MNIST [93], which has been found too easy and overused by several instances (see [182] for detailed critique on MNIST). The dataset contains of 60,000 gray

scale images of dimension 28×28, consisting of 10 classes of clothing items.
CIFAR-10 and CIFAR-100 [86] are collections of 60,000 color images of dimension $32 \times 32 \times 3$, also divided into 10 classes in the case of CIFAR-10 with 6,000 images per class, and, respectively 100 classes in the case of CIFAR-100 with 600 images per class. FashionMNIST, CIFAR-10 and CIFAR-100's each contain a predefined 10,000 image subset dedicated for testing.
Tiny ImageNet [90] is a smaller version of the ImageNet dataset (described next) consisting of 100,000 RGB images of dimension $64 \times 64 \times 3$. The data is divided into 200 classes with 500 training images, 50 validation images and 50 test images per class.
The ImageNet Large Scale Visual Recognition Challenge (ILSVRC) was a competition held each year from 2010 to 2017 providing a dataset commonly referred to as ImageNet [146]. We concentrate on one of the most used versions of the dataset, ILSVRC-2012, consisting of 1000 classes and a total of 1.2 million colored images of various dimensions. The validation dataset is composed of 150,000 images, while a separate, undisclosed test dataset is used to preserve the integrity of the actual challenge (which thus has not been used in this work). This thesis uses the validation dataset as the test dataset only evaluated finally, and for validation of hyperparameters, a random training dataset subset of 50,000 is employed. During training, a random crop-size of 224×224 serves as training augmentation, and for testing, images are rescaled to 256 pixels (shorter side), which is a standard practice in typical benchmark setups.
Finally, the Speech Commands dataset [177] comprises audio files with recordings of 35 different words employed as speech commands, with the objective of training voice interfaces. The class-unbalanced dataset consists of 105,829 audio samples (with a range of 1,557 to 4,052 audio samples per class). The audio files are 1-second long utterances of short words. The dataset consists of voice recordings from 2,618 speakers. These recordings have been encoded using linear 16-bit single-channel PCM values at a sampling rate of 16 KHz. The test subset consists of 4,890 recordings, the validation subset consists of 10,102 recordings.

Experimental Setups

The goal of this work is to get a basic understanding of core features used in deep learning. Thus, while plain convolutional networks have been replaced by ResNet architectures first, the Inception network family next, and lastly by more specialized "downstream" variants such as *Yolo v5* [77] or more modern transformer-based approaches such as ViT-G/14 [188].

This thesis will focus on the less complex versions of Convolutional Neural Networks (CNNs) and ResNets for two reasons: firstly, these models have demonstrated their effectiveness in achieving numerous breakthroughs; secondly, their consistent structure allows for a more straightforward evaluation of the impact of individual architectural features in subsequent chapters. The tools developed in this thesis are versatile and can be applied to various network types and datasets, as they only need a switch-like activation. Although certain sections may focus on the assumption that the switch-like activation follows an affine transformation, the monitoring tools described in Section 3.1 and Section 3.3 remain applicable nonetheless.

This thesis mainly uses CIFAR-10 and CIFAR-100 for base-line experiments, and validate on more complex datasets (ImageNet or Tiny ImageNet), a different modality (Speech Commands) or fully-connected instead of convolutional networks. Training data is split into 90% training and 10% validation set. The validation set is not included in any of the training results. Standard data augmentation (image flipping and random cropping) is applied during training. The experiments conducted focus on multiple architectures features. One is the choice of activation functions; the most commonly used piecewise linear activation functions for image classification today are ReLU [124] and PReLU [61], which are both compared in this work. As regularizers standard batch normalization [73], layer- and group-normalizations [181] are employed as described by the respective networks.

In terms of network architectures, the following are used in this thesis:

(i) *ToyNet* – stacked 3×3 convolutional layers, followed by global average pooling for base-line scenarios without residual connections.

(ii) F_1 – a fully-connected network with 5 hidden layers and 512 neurons each, using ReLU activations in-between.

(iii) VGG, introduced by Simonyan and Zisserman [158], comes in various configurations with differing numbers of layers, including 11, 13, 16, and 19-layer options. The architecture features extra max-pooling layers placed intermittently and a progressively increases the number of filters.

(iv) *ResNet* architecture in both variants (v1) [60] and (v2) [62] are tested, which first added residual connections to increase trainability of deeper networks.

(v) For ablation studies to gain insights into the influence of residual connections we define *ConvNet*, the same network architecture as ResNet, but with residual connections removed.

(vi) Two slightly more recent variants of ResNet that are also used in this thesis are PyramidNet [57], which uses a gradually increasing number of filters, and FixUp [191], which optimizes the different weight types in a network separately and changes the initialization scheme.

(vii) Section 4.2 additionally employs the networks Inception (v3) [170], MobileNet [67], Inception-Resnet (v2) [169], NasNetMobile [198], Xception [26], and Densenet [68].

(viii) *M5* [35] is used for the Speech Commands dataset.

The exact model definitions, hyperparameters and method implementations can be obtained from the provided GitHub repositories

- `https://github.com/JGU-VC/activation-pattern-analysis`
- `https://github.com/JGU-VC/progressive_stochastic_binarization`
- `https://github.com/JGU-VC/actcoolr`
- `https://github.com/da-h/simple_relu_visualization`.

2.2 Activation Patterns and Activation Cells

This section will introduce the concepts of activation patterns (APs) and activation cells (ACs). Although these definitions do not directly contribute to the explainability of networks, they will be instrumental in providing insights into the internal mechanisms and training behavior of deep networks throughout this thesis. The idea is to divide the input domain of a network with respect to a single layer.
This division implicitly creates what will be defined as ACs, revealing the number of distinct affine transformations a linear layer, followed by a Rectified Linear Unit (ReLU) activation ultimately represents.

This section begins by defining ACs and visualizing them for 2D inputs. The following analysis will then experimentally infer how the factors such as layer width, network depth, bias, and also training (for a simple experiment) influence their distribution in the input domain, and finally also discuss potential limitations of the method for more general cases.

2.2.1 Defining Activation Patterns and Cells

Let $F : \mathbb{R}^{d_1} \to \mathbb{R}^{d_{L+1}}$ be a simple feed-forward neural network in layer-wise form

$$F = f_L \circ \sigma \circ f_{L-1} \circ \sigma \dots f_2 \circ \sigma \circ f_1, \tag{2.7}$$

where f_l are the layer-blocks of the neural network (either only linear layers, or including normalization layers such as batch normalization)[6], and, σ denotes the activation function of the network. A *simple feed-forward network* refers to a simplified version which limits the layers f_l to linear layers of the form

$$f_l(x) := x^T W_l + b_l, \tag{2.8}$$

where the matrices $W_l \in \mathbb{R}^{d_l \times d_{l+1}}$ are called *weights* and the vectors $b_l \in \mathbb{R}^{d_{l+1}}$ *biases*.

The complexity of such a neural network F is significantly influenced by its activation function σ. As an illustration, removing the activation function reduces

[6] The approach described injects into the activation function of a network, and thus can also be applied to any kind of network structure beyond simple feed-forward networks as described above, involving also cross-connections, residual connections and recurrent networks – this definition uses merged methods denoted as f_l, representing linear methods followed by a ReLU activation for sake of simplicity.

the network F to a composition of affine transformations. Since compositions of affine transformations are affine transformations as well, increasing the depth of such a network alone would therefore add no complexity at all. For instance, in the two-layer case ($L = 2$, one can "squash" the activation-less network into an equivalent network consisting of only one layer ($L = 1$) using the combined weight matrix W' and bias vector b' as follows:

$$W_2(W_1x + b_1) + b_2 = \underbrace{W_2W_1}_{=:W'}x + \underbrace{W_2b_1 + b_2}_{=:b'}. \tag{2.9}$$

As explained in the discussion at the start of Chapter 2, the analysis will further set $\sigma = \mathrm{ReLU}$.[7] While the observation to be able to squash an arbitrarily deep network without activation functions (i.e. $\sigma = \mathrm{id}$) into an equivalent single linear layer is generally not true for networks with ReLU-activations, but it is still locally true:
The ReLU activation maps negative values to zero and passes through non-negative values unchanged[8]:

$$\mathrm{ReLU}\ (x) := \max(0, x) = \begin{cases} x, & \text{if} \quad x > 0 \\ 0, & \text{otherwise} \end{cases}. \tag{2.10}$$

Conceptually, the ReLU activation selectively applies either the identity function or the zero function. The ReLU-activation thus introduces in conjunction with linear layers a discrete decision into the computational graph of a network, implicitly partitioning the input space into discrete regions where the input space is mapped using the same functions.

The following definition uses this insight to reformulate the evaluation of the network F by using matrices called APs that mask out negative components using multiplication and formalizes how APs shape computation.

Definition 1. *Activation pattern (AP)*
Considering any fixed argument x, one can rewrite the ReLU-activations σ in $F(x)$, a feed-forward neural network in layer-wise form as follows:

$$\begin{aligned} F(x) &= f_L \circ \sigma \circ f_{L-1} \circ \sigma \ldots f_2 \circ \sigma \circ f_1 \\ &= f_L\left(A_{L-1}(x) \cdot f_{L-1}\left(A_2(x) \cdot \ldots f_2\left(A_1(x) \cdot f_1\left(x\right)\right)\right)\right), \end{aligned} \tag{2.11}$$

[7] The described method can be adjusted to work with any general activation function. Functions that have a standard discrete choice, such as parametric ReLU (PReLU), can be directly utilized. For functions without a standard discrete choice, the visualization can still be adapted by either implementing a threshold value or using multiple buckets.
[8] We stretch the notation by processing vector-valued function arguments component-wise implicitly.

where $A_l(x) := \operatorname{diag}\left(A_l^{(1)}(x), \ldots, A_l^{(d_l)}(x)\right) \in \{0,1\}^{d_l \times d_l}$ *and*

$$A_l^{(i)}(x) = \begin{cases} 1, & \text{if } (f_l \circ \mathrm{ReLU} \circ f_{l-1} \circ \ldots f_2 \circ \mathrm{ReLU} \circ f_1)(x) > 0 \\ 0, & \text{otherwise} \end{cases}.$$

The diagonal matrix $A_l(x)$ *describes the ReLU-decisions the neural network makes for the input* x *and is called* activation pattern (AP) on layer l.
We will use the notation $A_{\theta,l}$ *to indicate the APs for the network* F *with weights* θ*. Of course, the matrices* $A_l(x)$ *depend both on* x *as well as the structure and weights of* F*, but since* F *is fixed in all equations in this thesis,* F *is omitted from the notation.*

The previous definition can be directly used to define the implicit division of input space into subsets that contain only points that result in the same APs.

Definition 2. *Activation cells (ACs)*
Let $F : \mathbb{R}^{d_1} \to \mathbb{R}^{d_{L+1}}$ *be a neural network given by Equation* (2.7)*. Consider any layer* $1 \leq l \leq L$ *of* F*. An activation cell (AC) is defined as the region of the input space that gets mapped to the same AP,*

$$\mathcal{A}(x) := \left\{ y \in \mathbb{R}^{d_1} \mid A_l(y) = A_l(x) \right\}.$$

To simplify notation, ACs may be denoted using $\mathcal{A}_k$*, where* k *is an index variable, throughout this work.*

A definition that appears similar to Definition 1 has been also used by Croce et al. [34] and analyzed further by Hanin and Rolnick [59]. The key difference is that the proposed APs are defined across the whole network instead of only a single layer. Their notion of APs also implies ACs that they denote "activation regions", and, will be referred to global activation patterns (global APs) in this thesis, defined as follows:

Definition 3. *Global activation pattern (global AP)*
In addition to Definition 1, the concatenated vector of all ReLU-decision in the network F*, given by*

$$A(x) := \left(A_1^{(1)}(x), \ldots, A_l(x)^{(d_L)}\right) \in \{0,1\}^{\sum_{\text{layer } l} d_l} \tag{2.12}$$

is called the global activation pattern (global AP) *for input* x.

The layer-wise APs (Definition 1 which will be further denoted solely by APs) are the main focus in this thesis and can be obtained from global APs (Definition 3) by

ignoring the parts of the vector that represent all preceding layer activations. That way, the cells stemming from global APs, which are at the basis of the analysis of Hanin and Rolnick [59], are simply subsets of the ACs defined above. While these two definitions may appear alike, they function quite differently in practice.

2.2.2 Properties of Activation Cells

This thesis primary focus is to effectively count the APs and extract meaningful information about in the network using these measurements. Additionally, ACs offer valuable insights into the conceptual structure of the model, and they will be used in this thesis to analyze how different architectural components, such as width, height, depth, bias, and batch normalization layers, influence them in 2D input spaces. On a more abstract level, the difference appears most obvious when looking at the ACs stemming from the two different types of APs.

The number of unique linear APs on a given layer of a fixed network F has an obvious combinatorial upper bound, namely $2^{\sum_l d_l}$, since every activation can be either negative or positive, mapping to zero or keeping the value unchanged. Of these decisions, there are $\sum_l d_l$, the total sum of dimensions used in the computation, many.
However, even in the first activation layer, this upper bound cannot be realized, unless the input dimensions of the first linear layer is greater than its output dimension, $d_1 \geq d_2$. To illustrate this with as a simple example, imagine a scenario of three filters in two dimensions, which can only result in 7 ACs, although there are $2^3 = 8$ combinations of sign outcomes.

Proposition 1. *For 2^d APs in d dimensions to exist for a single hidden linear layer with D filters requires $d \geq \mathrm{D}$.*

Proof. Achieving 2^d APs with a linear layer would require $\mathrm{D} \cdot 2^d$ unique linear inequalities to hold (each activation requires a unique mapping to its respective signed outcome). However, there are only $d \cdot 2^d$ variables to choose from as each input dimension and each pattern results in one free variable. Therefore, the system of equation can only be solved if $d \geq \mathrm{D}$. □

The regions divided by the decision boundaries, that also implicitly define APs, have been already thoroughly analyzed in the literature. Most importantly: In the general case D hyperplanes in d dimensions result in a total of $\sum_{n=0}^{d} \binom{\mathrm{D}}{n}$ connected regions

([164], Proposition 2.4.) which approximates to 2^{D} for $\mathrm{D} \leq d$ and $\frac{\mathrm{D}^d}{d!}$ for $\mathrm{D} \gg d$ [59].

Definition 2 differs from the ACs analyzed Hanin and Rolnick [59] in the space of the interim activations (i.e. those in layers denoted with the index $l \neq 1$). While their understanding is to separate space into the distinct affine transformations of the whole network, Definition 2 concentrates on the action of a single layer. As can be obtained from the next observation, one direct implication is that, unlike the global APs used in Hanin and Rolnick [59], the layer-wise ACs defined in Definition 2 are *not* always convex.

Proposition 2. *(Non-convexity of ACs)*
ACs are always convex for the first layer ($l = 1$), but they may become non-convex for succeeding layers ($l > 1$).

Proof. For the case $l = 1$, note that Definition 2 matches the definition used in Raghu et al. [137]. Thus, the reader is referred to their proof of convexity of ACs (Theorem 2 in [137]).
For the latter case, $l > 1$, a simple counterexample provides the rest of the proof: Consider a neural network $F : \mathbb{R}^2 \to \mathbb{R}^1$ in layer-wise notation (see Equation (2.7)) with two layers:

$$F(x, y) := \mathbb{1}_1 \cdot \mathrm{ReLU}\left(\begin{pmatrix} -1 & -1 \end{pmatrix} \cdot \mathrm{ReLU}\left(\mathbb{1}_2 \cdot \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \end{pmatrix}\right) + 1\right),$$

where $\mathbb{1}_d$ denotes the identity matrix in d dimensions. First, rewrite the definition into its component-wise form:

$$F(x, y) = \max\left(0, -\max\left(0, x + 1\right) - \max\left(0, y + 1\right) + 1\right).$$

It follows, there are two ACs in the second layer,

$$\mathcal{A}_1 := \{(x, y) \in \mathbb{R}^2 \mid -\max\left(0, x + 1\right) - \max\left(0, y + 1\right) + 1 > 0\}$$
$$\mathcal{A}_0 := \{(x, y) \in \mathbb{R}^2 \mid -\max\left(0, x + 1\right) - \max\left(0, y + 1\right) + 1 \leq 0\}.$$

Now consider the points $(1, -3)$ and $(-3, 1)$. Both points are elements of A_0:

$$F(1, -3) = \max(0, -2 - 0 + 1) = \max(0, -1) = 0,$$
$$F(-3, 1) = \max(0, -0 - 2 + 1) = \max(0, -1) = 0.$$

Thus, for A_0 to be convex, all points in the set connecting these two points directly,

$$\left\{(x, y) \in \mathbb{R}^2 \mid y = x + 2 \text{ and } -3 \leq x \leq 1\right\},$$

would have to be also elements of A_0. However, the point $(-1, -1)$ is not:

$$F(-1, -1) = \max(0, -0 - 0 + 1) = \max(0, 1) = 1,$$

thus $(-1, -1) \in A_1$. We have found a non-convex activation cell, A_0. □

2.2.3 Visualization of Activation Cells

As discussed in Section 2.2.1 the visualization of the internal decision-making process of a deep network has its own set of challenges: One issue is that networks use high dimensional spaces for intermediate calculations, which are difficult to visualize effectively. Second, the structure of a network changes implicitly, as the activations depend on the input. The last section has introduced the concept of ACs, the separation of input space into disjoint subsets that are mapped by the respective layer using a different (fixed) linear mapping which will help us in this section to get qualitative results in the special case of a 2D data domain. This section uses this concept and circumvents both of the aforementioned issues related to visualizing interim representations by revealing how hyperparameters of a neural network influence the number of functions the network implicitly utilizes for processing the input space.

Activation Cell Visualization Algorithm

Assuming a 2D input domain and a ReLU activation function, ACs can be used in the following way to visualize any layer of arbitrary interim dimension in a neural network. As shown in Figure 2.3 the algorithm is visually represented in a schematic diagram. As a first step, the 2D-input space is sampled by a predefined grid. After restructuring the sampled 2D-positions (x- and y-axis) the data is reshaped to a flattened grid-tensor of dimension $(w \cdot h) \times 2$. This tensor can be efficiently processed by the neural network. The next step is the forward pass of the network, evaluating the interim results up until layer l, in particular retrieving the APs $A_l((x, y))$ (defined in Definition 1) for each input point on the grid. The next objective is to assign colors to the sampled points in the input domain, ensuring that identical AP values always correspond to the same color. However, the total number of potential APs

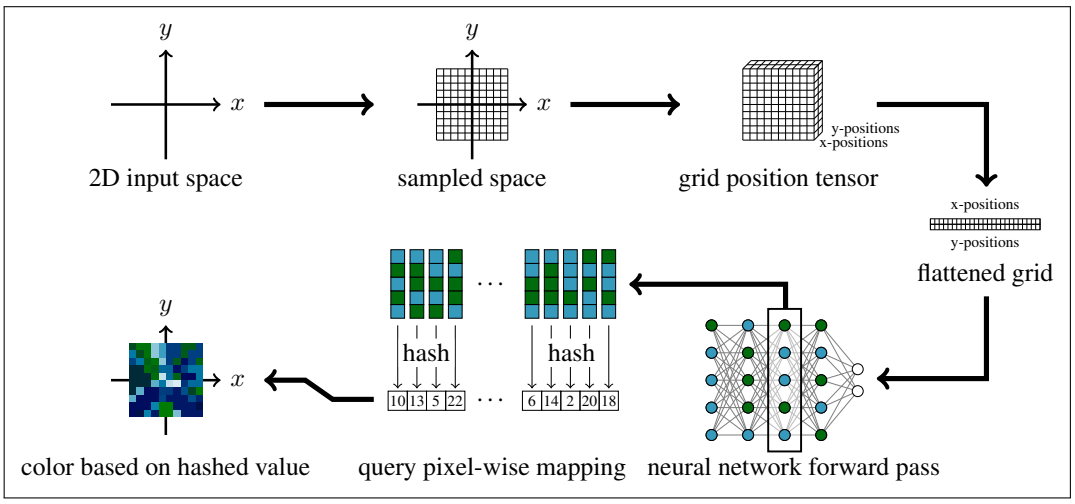


Fig. 2.3.: Schematic representation of how to transform the 2D input space into a bitmap image showing the ACs of a neural network layer efficiently. First, sample the 2D space using an equidistant grid. Then, construct a tensor of the respective coordinates in 2D space given by the constructed grid and reorganize the result in order for a neural network to process each coordinate. Finally, save the resulting activations from the neural network, hash these vectors and map the values back to their original position on the sampled grid. At the cost of a fixed resolution, the efficiency comes from using samples to extract activations instead of constructing the decision boundaries from the internal parameters of the neural network.

increases exponentially with the interim dimension (there are 2^{d_l} potential patterns in a vector space with d dimensions). Thus, to decrease memory usage and control the number of colors, the APs are then hashed using the binary hash function shown in Algorithm 1 (which can be efficiently computed on a GPU). The assignment of a color is determined using a straightforward modulo operation. Finally, the colors assigned to all points on the 2D grid are then plotted as a $w \times h$ image again.

While this enables to visualize the decision boundaries for networks with two-dimensional input dimension, arbitrary input dimensions would be of course more challenging to visualize using this technique. A simple extension is to sample two dimensions and applying the same technique in the projected plane, similar in spirit to Li et al. [98] (also visualized in Figure 2.2(g). We will evaluate this idea at the end of this section.

2.2.4 Analysis of Uniform-Width Feed-Forward Networks

Using Figure 2.3, the following paragraphs analyze separately the effect of the network parameters **width**, **bias-values** and **depth**, additional **batch normalization** layers and also the effect of **training** on the FashionMNIST dataset.

Algorithm 1 Hashing a list of activations, $\text{hash}\,(A)$

Input: activation list $A \in \{0,1\}^{n \times d}$

// Reinterpret Binary-Vector to Integer Vector
$A \leftarrow$ **reinterpret**$(A, \text{int64})$, thus $A \in \text{int}_{64}^{n \times \lceil d/64 \rceil}$
$h \leftarrow 0 \in \text{int}_{64}^{n}$

// Define three arbitrary, but large primes
let offset $= 1610612741, P = 805306457, Q = 402653189$

// hash integer vector row-wise (parallelizable)
for $i = 1$ **to** n **do**
  $h_i \leftarrow$ offset
  for $j = 1$ **to** $\lceil d/64 \rceil$ **do**
    $h_i \leftarrow (A_{ij} \cdot Q)\ \text{XOR}\ (h_i \cdot P)$
  end for
end for
Output: list of hashed activations $h \in \text{int}_{64}^{n}$

The analysis starts with the inspection of simple feed-forward networks; L layers with linear filters f_l (see Equation (2.8)) with D interim dimensions in each layer and ReLU-activations in-between.
Initialized using the *Kaiming Normal initialization* [61], the networks expect the input distribution to be uniformly normal distributed. The choice of domain and sample size is best chosen to fit most data points of any normal distributed dataset. In this section, we will present visualizations of the activated cells using a grid with 500×500 uniformly distributed sample points within the range $[-5, 5]^2$, as this configuration has been found to provide the best fit in the shown setups.

Width: The first parameter to analyze is the effect of varying network widths (i.e. number of filters per linear layer) on the separation of the input space. To consider the effect isolated from other parameters, the network used consists only of a single hidden layer, and a "default" bias, i.e. a uniformly sampled value. Figure 2.4 the visualization technique applied to three choices of filters, from left to right using $\mathrm{D} = 16$, $\mathrm{D} = 64$ and $\mathrm{D} = 256$ filters. The separations of input space shows that the number of filters directly affects the number of cells. As expected, the number of cells increases in an exponential-like manner as the number of filters rises. Indeed, the maximum number of activations for D filters is 2^{D}, but the number of actually measured ACs grows much slower than expected, which complies with the observations of Hanin and Rolnick [59]. Figure 2.5 shows this discrepancy by plotting the measured number of cells in the domain and the theoretical maximum 2^{D}. The difference from the estimation $\mathrm{D}^2/2!$ of [59] comes from the restriction

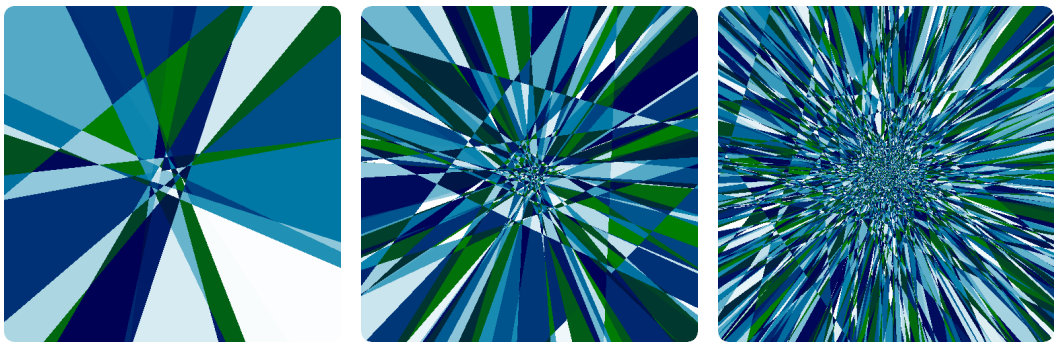


Fig. 2.4.: Visualizations of the ACs for a single linear layer followed by a ReLU activation. From left to right, the images show the ACs for 16 filters, 64 filters, and 256 filters. The ACs are convex, and the number of lines separating them corresponds to the number of filters used. One can observe that while the number of filters quadruples, the number of ACs increases exponentially.

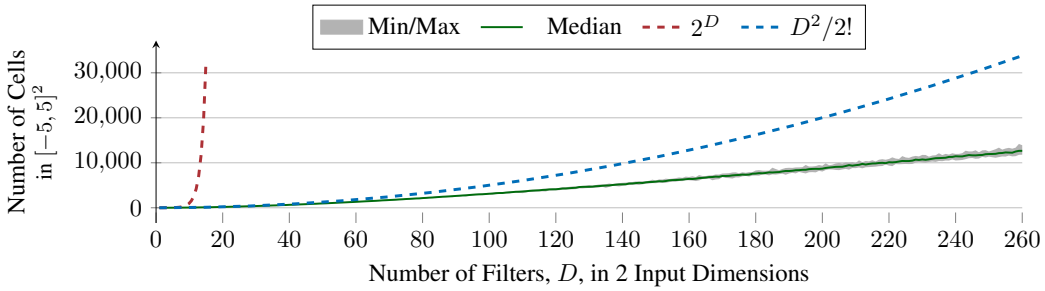


Fig. 2.5.: This graph compares the combinatorial number of possible binary APs of a single hidden-layer network with D many filters in $d = 2$ input dimensions, namely 2^{D} in a single neural network, to the number of expected hyperplanes in two dimensions, $\sum_{k=0}^{d} \binom{\mathrm{D}}{k} \approx \mathrm{D}^d/d!$ (for $\mathrm{D} \gg d$), to the mean number of measured ACs in the domain $[-5,5]^2$. The combinatorial maximum of 2^{D} holds as an approximation for $\mathrm{D} \leq d$. Using a subspace compared to counting all possible ACs in the whole domain $\mathbb{R}^2$ leads to a discrepancy in the expected number of ACs known from the literature.

on the fixed subset $[-5,5]^2$ instead of counting the ACs in the whole input domain $\mathbb{R}^2$.

Bias: The next experiment considers the effect of the bias on the ACs. The goal is to view how the bias influences the number of cells in isolation by solely adjusting the bias values – a discussion using more realistic example with uniformly sampled biases follows later in Figure 2.12. Thus, instead of sampling the bias as before, Figure 2.6 visualizes the ACs for 128 filters in a single hidden layer, with one of three fixed bias-values, $b_i \in \{0, 1/4, 1\}$, corresponding to the sub-figures from left to right). A constant zero-bias setup (left plot) reveals a unique structure (compared to all other choices) where all ACs intersect in the image center, i.e. the input space origin ($0 \in \mathbb{R}^{d_1}$). The reason for this observation can be simply explained by examining when sign changes happen, which in turn cause changes of APs. A sign change

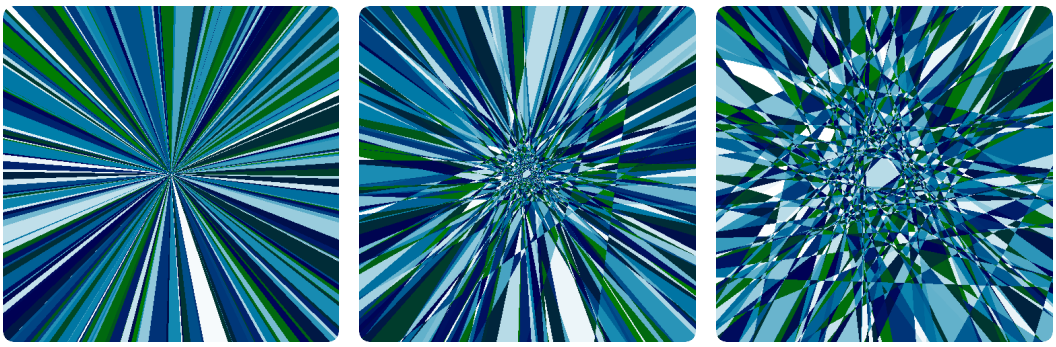


Fig. 2.6.: Visualizations of the ACs for a single linear layer with 128 filters followed by a ReLU activation. From left to right, the images show the ACs for three varying but constant bias values, bias vector set to zero, bias components set to 0.25, and bias components set to 1.0. For a zero bias, all ACs meet in the domain center; for increasing bias values, new ACs appear and increase in size. For a non-zero bias vector, a AC exists around the origin of the domain.

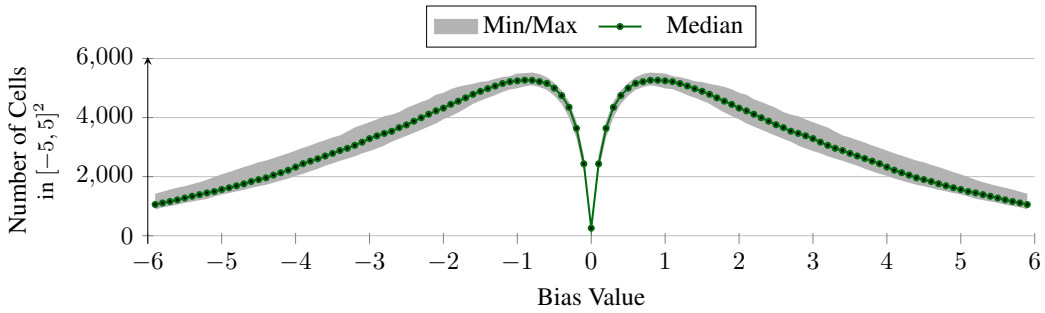


Fig. 2.7.: This graph relates the chosen bias value for otherwise constant linear layer properties followed by a ReLU activation to the number of ACs in the domain $[-5,5]^2$. For a bias value of zero, the number of ACs is smallest; then, for increasing magnitudes, the number of ACs grows to a maximal value at about 0.8. After that, the number of cells shrinks due to the restriction of the measurement to a fixed domain and the ACs becoming too large or appearing out-of-scope. The symmetric nature of this graph (positive and negative bias values provide similar results) is due to the symmetry that ACs provide by counting for each additional filter how many patterns it splits, which is equal if all signs are flipped, for instance.

occurs for any filter in $Wx + b$ whenever x crosses a decision boundary, which all intersect in the origin without a bias ($b = 0$). (Unless the vector x is orthogonal to that filter in the matrix W in which case x sits on the decision boundary of that filter.)

The other two figures (middle and right) show the effect for a non-zero bias: Most interestingly, the decision boundaries do not show same distance to the origin. This observation reveals that the bias values work on a different scales, depending on the magnitude of the weights of the respective filter-components. Assuming a scenario where a single filter is enlarged by a factor of 10. In that case, changing the bias without an according increase of magnitude affects the input space solely one tenth

the amount compared to the default case without the scaling.
Another interesting observation is that there exist a cell in the center around the origin for any non-zero bias vector. Assuming a strict positive bias vector, $b_i > 0, \forall i$, this central cell would represent the identity branch for all ReLU-activations in that layer. Conversely, the cell's contents are mapped using the zero function for strictly negative bias vectors, $b_i < 0, \forall i$. Typically only positive biases are used for initializing neural networks (both in the case of simple networks but also when using batch normalization). Thus, the bias magnitude influences the size of the central "identity" cell that represents the fixed linear mapping for the data points near the origin.
As with network width, it seems that the bias also affects the number of ACs. At least, Figure 2.7 visualizes the relation of a constant bias magnitude to the number of measured ACs in the subdomain $[-5, 5]^2$ and shows that the number of cells first grows to a maximal value at $b_i \approx 0.9$. However, while a bias magnitude of 0 does effectively limit the number of possible patterns, the AC count in the whole input domain actually remains constant for any magnitude $b_i \neq 0$ (see Hanin and Rolnick [59]). The reason for this observed difference can be inferred from Figure 2.6: limiting the sampling of the subdomain to $[-5, 5]^2$ results in some cells outside that subdomain (for large biases) or too small in size (for small biases) to be missed in the measurement.
Seemingly a worse strategy to measure number of cells on a subdomain of input space instead of the full input domain, this is precisely the goal of this experiment: while the whole domain contains a constant number of ACs, what the network will *actually* encounter during training and testing is *not* uniformly sampled from the whole domain, but typically normalized to have zero mean and a unit standard deviation. Thus, smaller cells are also more unlikely to be "found" by a random example in the dataset, making the visualization and this graph a valuable tool already to estimate the number of ACs that a network actually encounters in practice. This idea of sampling existing data points (instead of considering the whole domain) and checking the actually encountered APs for the network (instead of computing what it theoretically could do) will be used multiple times in this thesis to inspect the information flow of a network in more detail.

Batch normalization: A batch normalization layer is typically placed right before the ReLU-activation and computes the batch-wise mean and standard deviation to normalize interim results (just as dataset normalization does to the whole dataset), meaning this layer requires actual data to flow through the network. Crucially, this data must be separate from the data grid created for visualization purposes. If not, altering the grid parameters for visualization would affect the outcomes. Thus, to visualize the ACs of a network that includes batch normalization layers requires

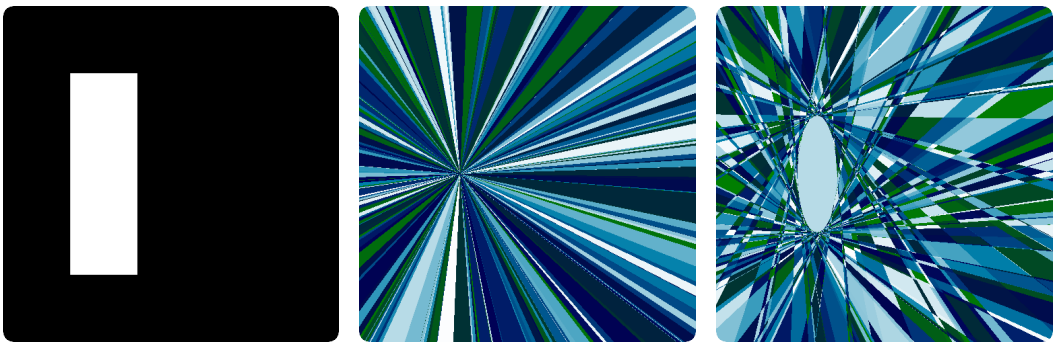


Fig. 2.8.: The choice of the dataset may affect the distribution of ACs for some choices of deep networks. This visualization shows that an off-center dataset affects the ACs when using a batch normalization layer between the linear layer and the activation layer. The left visualization shows the dataset used (all white pixels living in a subspace of the evaluated domain). The visualizations on the right show the effect of batch normalization in *train mode* with (middle) zero bias and (right) batch normalization's trainable additive parameter (effectively bias) set to 1.0. The dataset mean moves the point where all ACs meet for the zero bias, and the "middle" cell for non-zero bias values. The right plot also shows that the standard deviation of the dataset transforms the distribution of ACs as well.

employing a dataset. In this case the experiment uses a simple one, consisting of points drawn uniformly from the subset $[-3, -1] \times [-3, 3]$ as shown in Figure 2.8(a). By first testing the simple case of omitting a bias term, $b = 0$, one can see in Figure 2.8(b) that the ACs align themselves with the mean of the example dataset.[9] Re-introducing a bias term (Figure 2.8(c)) shows that the ACs even adapt according to the axis-aligned standard deviation of the dataset. Again, the AC around the dataset mean is the cell that represents the "identity" branch of all ReLU-activation components, effectively passing the filter results directly into the next layer.

Depth: The earlier mentioned parameters did not change the *shapes* of the ACs. For networks consisting of a single hidden layer, the active cells defined in Definition 2 are equivalent to those described by Hanin and Rolnick [59]. They have demonstrated that their definition results in convex cells. For multi-layers networks, however, the two definitions do differ. While the definition of [59] ensured that the ACs of all layers remain convex, the definition given in Definition 2 does not always result in convex shapes (see Proposition 2). We can also see this in Figure 2.9 that visualizes the AC in the first, the second, and the eighth layer of a simple eight-layer neural network with 16 filters per layer, no batch normalization, and using uniformly sampled biases. While the first layers left image has only convex cells, already the

[9] During normalization, batch normalization subtracts the mean of the batch. This makes the bias-term b of the filter layer obsolete. However, as the normalization layer introduces a trainable parameter called β which is added additively right before the ReLU-activation, we name *that* parameter the "bias" in this context.

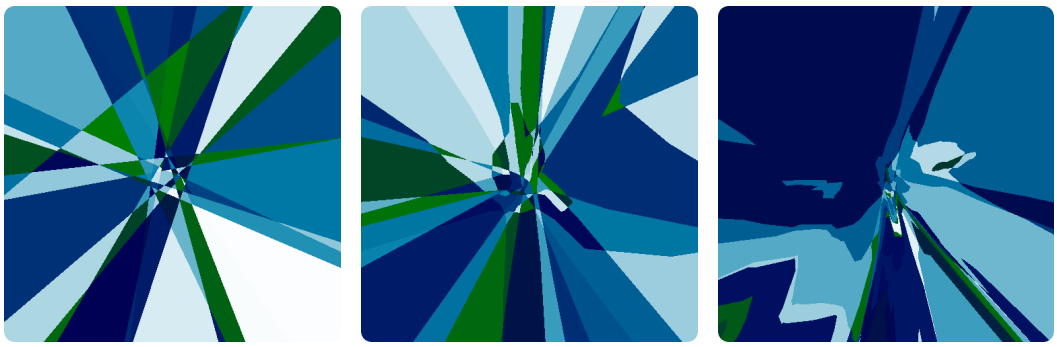


Fig. 2.9.: The definition of APs does not always lead to convex sets as shown in this Figure. This visualization shows the resulting ACs for a linear 8-layer ReLU network, randomly initialized using He-initialization and a fixed bias of 1 for all bias dimensions. The AC visualization is evaluated at layer 1 (left), layer 2 (middle), and layer 8 (right). Only layer 1 consists of only convex cells and non-convex ACs already appear layer 2, and become increasingly complex with every successive layer.

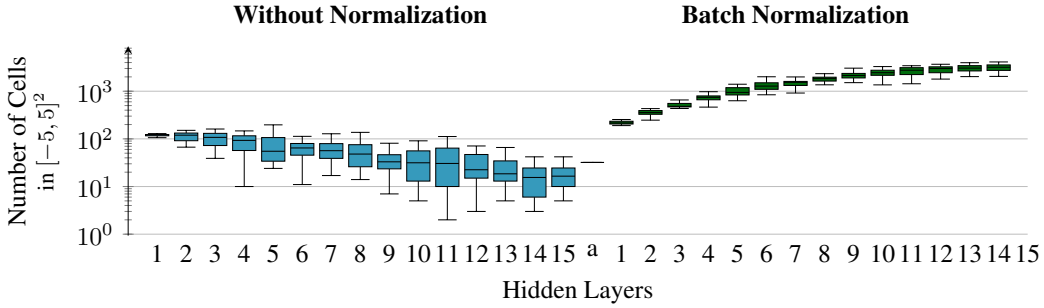
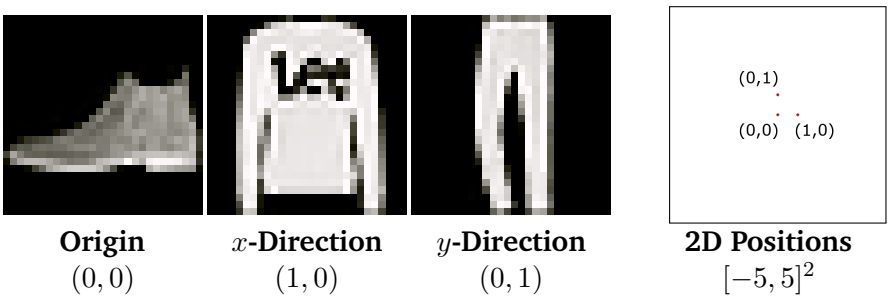


Fig. 2.10.: This graph compares the number of measured ACs for a simple neural network consisting of only linear layers, ReLU activations with and without batch normalization for varying layer count. The box-plots each visualize the median line, the lower/upper quartiles, and the lower/upper whiskers (quartile $\pm\, 1.5 \cdot \mathrm{IQR}$) over 25 seeds. While the number of global APs increases with network depth, the number ACs may decrease with network depth in a fixed domain. Batch normalization layers change this behavior – in that case, the number of ACs increases monotonically.

second layer (middle image in Figure 2.9) contains non-convex cells in the given domain, and in the eighth layer (right image), there is even a cell contained in another cell (the dark blue cell that takes up about one fourth of the whole input domain contains a small cell). Both observations, the existence of a much larger in deeper layers, and the existence of non-convex cells, are contrary to the findings of Hanin and Rolnick [59]. We will discuss these differences in more detail at the end of this section. Similarly to the other experiments conducted in this section, Figure 2.10 shows the relation of the number of measured ACs to the depth of the layer employed for the measurement. The box-plot visualizes the median, lower quartile, upper quartile, lower and upper whisker of the measured number of ACs for



Origin $(0,0)$ | x**-Direction** $(1,0)$ | y**-Direction** $(0,1)$ | **2D Positions** $[-5,5]^2$

Fig. 2.11.: To visualize a 2D-slice of the FashionMNIST dataset (784 input dimensions) three random images from the dataset are sampled to span a hyperplane to apply the 2D AC visualization technique as done in previous figures. Left image is used as the origin and the two other examples are used to define the x-direction (middle image) and the y-direction (right image). The subspace spanned by these images is defined as the linear interpolation of these three data points. The 2D plot on the right visualizes the positions the three images will have in the following AC visualizations.

repeated experiments over 25 seeds. All layers use the same number of filters, are each randomly initialized using He initialization and for the visualization the grid size is once again defined within the domain of $[-5,5]$. The experiment is conducted in two variants: with and without batch normalization layers.
The results differ noticeably for the networks with batch normalization compared to the networks without the normalization layers. While the number of measured ACs decreases for the network without normalization layers, the number increases instead in the case batch normalization is used.
To summarize, the deeper the layer the measurement takes place, the more complex the form and appearance of the AC may be. For instance, while the expected number of ACs decreases for deeper layers for simple networks, batch normalization counters that effect in this case. Chapter 3 will provide other methods to analyze similar effects showing that other architectural features and training schemes beyond network depth also influences the number of ACs measured in a deep network.

The visualizations until now have shown the network features “width”, “bias”, “” and “depth” in isolation and for a fixed network, but they have not yet considered a setup that also includes a real-world dataset and the training of such using gradient-based optimization. The next visualization will show the Adam Optimizer (with default settings), using default settings, optimizing a small network for a total of 100 epochs. This process includes two learning rate reductions by a factor of 10, occurring at epochs 50 and 75, respectively.

Network Training (F_1 on FashionMNIST): The setup consists of the network F_1 (similar to [80]), a fully connected network consisting of five ReLU-activation layers

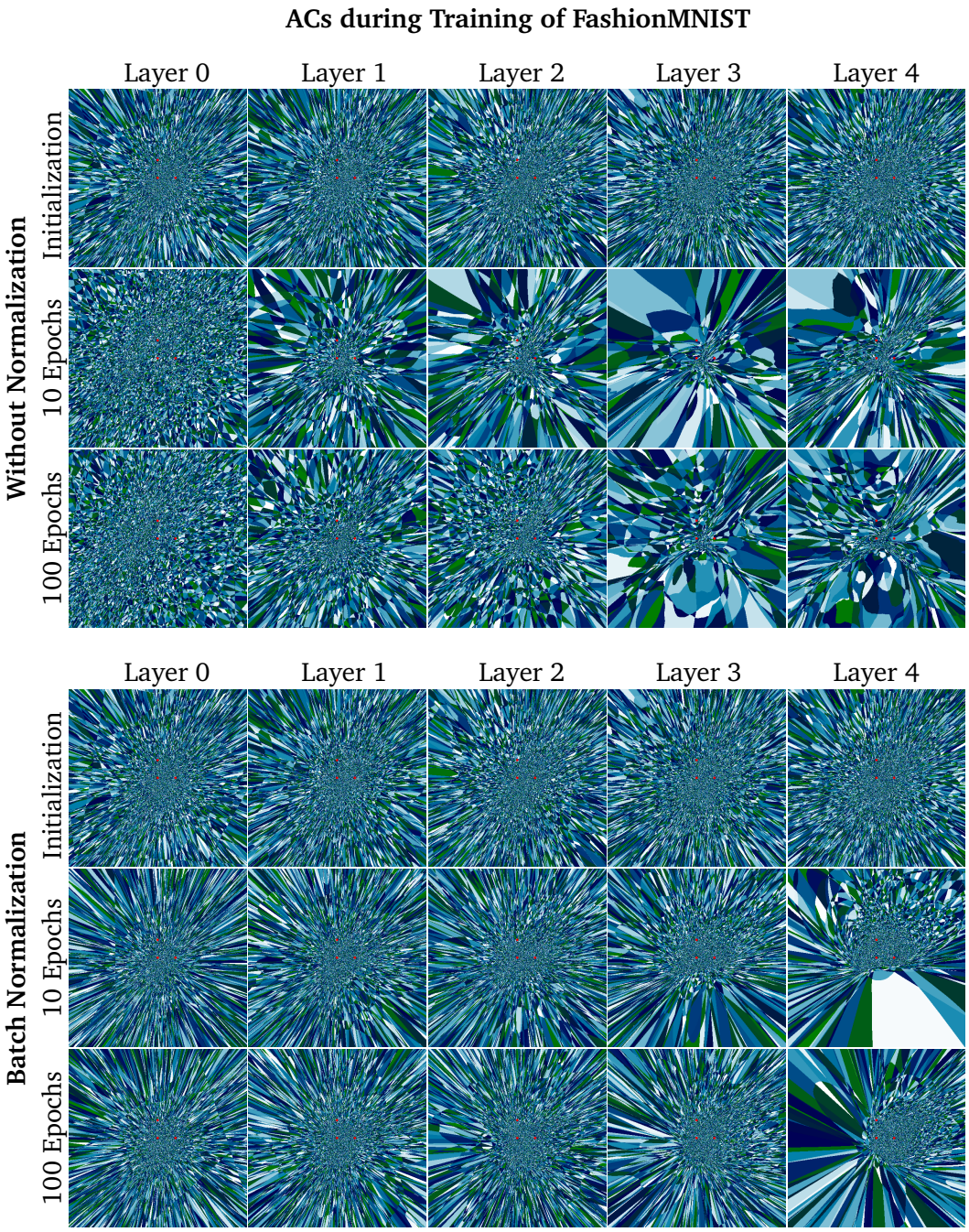


Fig. 2.12.: This table illustrates the training process of the model F_1 trained on FashionMNIST, by showing the ACs of each of its five activation layers at initialization, after 10 epochs of training, and after 100 epochs of training. The top three rows show the network without any normalization layers, the bottom three rows show the network with batch normalization layers before the ReLU activations. The three red dots indicate the positions of the three images from the dataset used to span the 784-dimensional input space of FashionMNIST in order to enable the visualization of the ACs on a 2D plane. Key observations are: (1) The network with batch normalization layers covers the space around the red dots consistently across all layers and training time. (2) Omitting the normalization layer starts with a dense distribution of ACs at the center as well, leading throughout the training to a sparser distribution, but (3) still densely covering the space around the red dots with ACs.

but without normalization layers, to train to classify FashionMNIST. As above, the experiment is conducted using two network variations: with and without batch normalization layers. The dataset consists of 60.000 gray scale images of size 28×28. Therefore, a 2D visualization as for example in the previous experiment using two-dimensional input spaces cannot be applied directly. Instead, the strategy is to choose three fixed images to form a 2D coordinate system to represent a 2D slice through the 784 dimensional inputs of the network. The three example instances shown in Figure 2.11) span the two-dimensional linear subspace of the whole input space.

As these three data points span a 2D plane, one can apply the same approach of visualizing ACs as in the toy-dataset examples before. For the visualizations of this experiment, the origin for that subspace is placed in the image centers, the second data point is then placed at the coordinate $(1, 0)$ in that subspace, and the third data point is placed at the coordinate $(0, 1)$ as shown on the right picture of Figure 2.11.[10]

The result is shown in Figure 2.12, which visualizes the ACs of all five activation layers of F_1 at different points during training: before training (Initialization), after 10 epochs of training, i.e. 10-fold re-iteration of the full dataset consisting of 930 batches[11], and the state after training has been completed (100 Epochs). Each of the images shows the positions of the three images (origin $(0, 0)$, $(0, 1)$, and $(1, 0)$ in the image using red pixels. To ensure consistency with previous results and to better visualize the areas within the domain that corresponds to standard normal distributed data, the next plots will show the extended subspace $[-5, 5]^2$, just as in the previous experiments.

The figure shows that training can have different effects on different kinds of networks. In this case, batch normalization layers ensure that the number of ACs is high throughout training. Without batch normalization layers, the network remains effective in densely covering the surrounding space with ACs. This means that in these densely covered areas, the linear layer combined with the activation layer achieves a higher potential expressivity compared to areas such as in the corners of the $[-5, 5]$ space, where only a limited number ACs exist.

This figure also implies that the initialization of both a network with batch normalization and a network without batch normalization results in the same coverage in terms of number of ACs – at least within the subdomain $[-1, 1]^2$, where most data points are likely to reside, assuming sufficient many data points and interim dimensions are used. This example illustrates that the choice of the visualization

[10] Of course, as any three data points will result in different visualizations, qualitative results can thus only be obtained by inspecting many such tuples. Chapter 3 will introduce a more consistent way of analyzing activation patterns.

[11] The training dataset consists of 50.000 images, for validation 600 images have been extracted beforehand for this experiment.

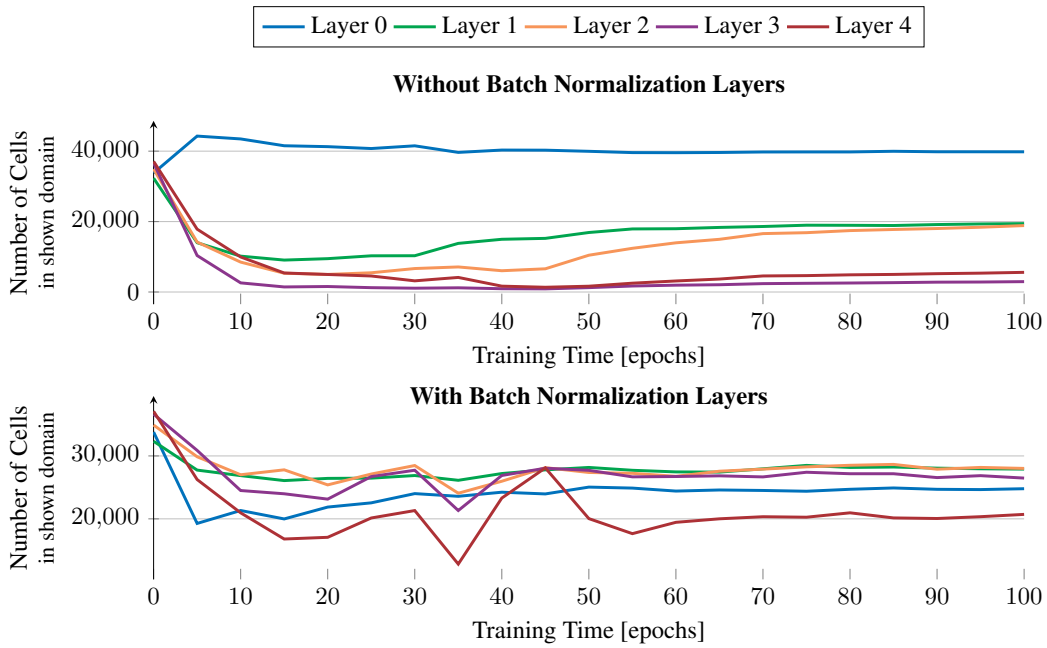


Fig. 2.13.: Accompanying Figure 2.12, this figure counts the number of ACs of the experiment visualized Figure 2.12. In addition to the three shown states (Initialization, 10 epochs, 100 epochs, i.e. after training), this graph shows the layer-wise number of measured ACs in the domain $[-5, 5]^2$ after every 5 epochs of training with and without batch normalization.
Qualitatively observable in Figure 2.12, this graph shows a different behavior of training when comparing the same network with or without batch normalization. Especially, the number of ACs does not increase during training in this experiment. Note that this graph changes with the chosen subdomain as discussed in the main text. For instance, in the case of an even smaller subdomain, $[-1, 1]^2$, the number of ACs would increase throughout training when used with a batch normalization layer.

plane that is used changes the number of measured ACs, i.e. the number of affine transformations a linear layer followed by a ReLU-activation may represent.
While the centers of the plots all have a higher density of ACs (for instance the regions around the three red dots), the space in the corners of the visualizations are rather sparsely covered by ACs for the network without batch normalization layers, but appear more dense for the network with batch normalization layers before ReLU-activations. This shows that data-dependent measurement of APs (as done in this section) may reveal different properties, compared to the data-independent technique by Hanin and Rolnick [59], who have symbolically derived the (global) ACs for a given network state, but for the whole input domain.
The next section discusses the differences of these two approaches and resulting qualitative different conclusions of data-dependend and data-independent AP analysis.

Differences to the ACs in Hanin and Rolnick [59]

ACs have been analyzed in the literature already (e.g. [34, 59]). The idea of ACs (or, "activation regions") is to separate input space into disjoint regions of points that are mapped by the network (global ACs) or by the layer (ACs) by the same affine transformation. Hanin and Rolnick [59] have observed that the number of distinct activations typically increases during training. Mathematically, they have derived an upper bound for the expected number of activation regions in a ReLU network.
However, their analysis relies on global APs, which differ from APs that focus solely on a single activation layer. Furthermore, their approach considers patterns in the entire domain, whereas this thesis examines the patterns on a fixed set of data samples (i.e. sampled from a fixed domain and closed domain in this section). Therefore there is a significant distinction between their findings and the observations from the experiments conducted in this thesis.

There is also a major technical difference that limits comparisons in non-default experimental setups: In contrast to the definition of APs (Definition 1), the global APs[12] used by Hanin and Rolnick [59] contains the activations of all neurons of preceding layers in the AP vector. The practical difference with this approach is that except for fully connected networks, this approach cannot be used to analyze the arbitrary ReLU networks: For instance, convolutional networks cannot be analyzed using global APs, because of three key reasons. Firstly, the shape of input data varies across data samples (image dimensions). Secondly, the shape of interim tensors also changes across the network for a single input image due to striding and pooling operations. Lastly, the outputs of convolutional networks are typically computed using a sliding window of neurons.
The resulting challenge stems from the question of how to handle the repeated application of the same neurons for different parts of the image. The AP vector, which represents the activation of the convolved filter, would have to reflect the re-application by containing some neurons multiple times, but with potentially different activations at different parts of the image. Furthermore, maintaining comparability of AP vectors across different input images would limit the choice of datasets for analyses to such that contain only images of the same size.
Instead, the layer-wise view to measure APs enables to view every application of the filter set as a new data sample. Other architectural features may also complicate the analysis of global APs: For instance, batch normalization layers require to compute the mean and standard deviation of the interim results, effectively changing succeeding calculations depending on the data that is used to compute the statistics.

[12] Note that Hanin and Rolnick [59] called global APs just "APs ".

This last point, however, could be solved by examining only using the "evaluation" mode of the neural network with fixed statistics, or, by choosing representative batches for the analysis.

Of course, the definition given by Hanin and Rolnick [59] has a different goal, which is to get the number of affine transformations that describe a whole neural network – they found that the number of APs rises slower than expected with as network depth increases. This method involves observing the whole input space, instead of limiting the measurement to a fixed domain, as done in the previous section, yielding different results overall: For instance, they found, scaling all bias values by the same number does not change the total number of ACs (see Lemma 6 in [59]). Similarly, deep and wide networks may also provide the same expressivity (in terms of global APs) under certain conditions.
In contrast, the definition of AP given in this thesis (as defined in Definition 2) focuses on the action of a single layer in isolation, splitting the input space into ACs with the property, that this *single linear layer* followed by ReLU transforms input data using a given set of affine transformations (identifiable by the AP in that domain). While the definition in [59] results in ACs that are convex, Definition 2 yields different results: Figure 2.9 has visualized ACs that are not convex. Being additionally limited to a subset of the whole input domain (the earlier plots showed the 2D-domain $[-5, 5]^2$), the number of ACs is much smaller than the upper bound that has mathematically been derived by Hanin and Rolnick [59] (a comparison of the discrepancy is shown in Figure 2.4). Similarly, increasing the bias values reduces the number of measured ACs additionally (see Figure 2.6) as ACs "move out of scope" for the given domain subset.
However, similarities exist as well – similar to the observations by [59], the number of APs first decreases quickly and recovers only slowly as training proceeds, see Figure 2.13. This effect will be discussed in more detail in Chapter 3.

Limitations of Sampling based Measurements

The data-dependent analysis described in this section also has its own set of limitations, which can be best visualized by examining the results depicted in Figure 2.7, as discussed next. The limitations stem from counting the ACs on a subdomain using a fixed grid of uniformly spaced data samples, instead of working with the whole input domain as Hanin and Rolnick [59] did. For instance, a constant bias value of 0 does have a minimal number of ACs for both techniques. When considering global APs on the whole domain and non-zero biases, the actual bias value does not change the number of ACs. This observation differs from the data-dependent

sampling-based technique used in this section. As shown in Figure 2.7, the number of ACs increases with the bias until a bias-value of about 0.9. This is a side-effect of ACs being effectively too small to be "seen" with the sampling-based approach. It is important to note that using the subdomain $[-5, 5]^2$ a reasonable choice, if most of the data points fall within this domain. Increasing the bias value to a larger value, say 10, would lead to the ACs move "out of sight", giving the impression that fewer cells exist overall. Although the analysis is primarily suited for "reasonable" values of biases, filters and sampling setups of the sampling grid, the method is capable of providing a qualitative representation of the network's implicit separation of input space. The following section will explore that this approach may initially appear inferior, but has the potential of ultimately prove beneficial for practical applications.

2.2.5 Discussion of Activation Patterns and Activation Cells

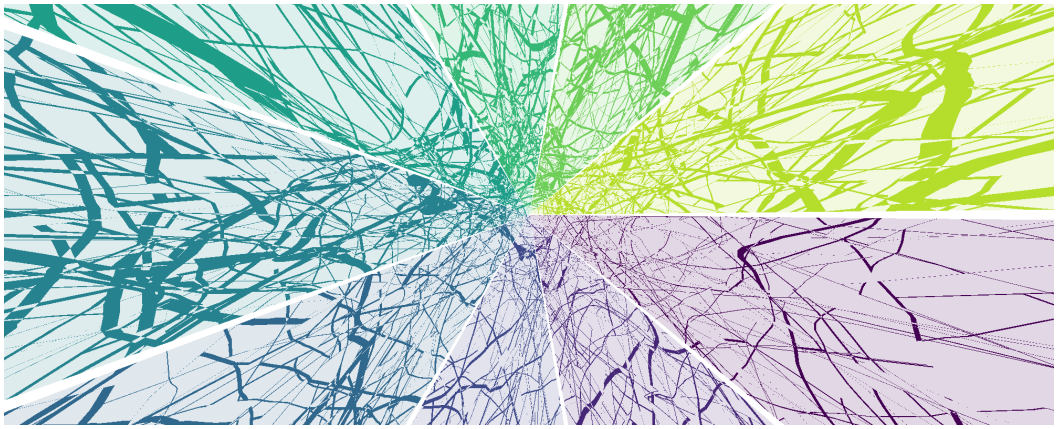
An in-depth analysis the internal representations of a neural network is a challenging task due to the high-dimensional interim spaces used for computations. This section has introduced activation patterns (APs) (as detailed in Definition 2) and utilized them to define activation cells (ACs), the subsets of *input space* that represents the same affine transformation in a given linear layer followed by a Rectified Linear Unit (ReLU) activation . Effectively, this allows to inspect the discrete decisions the network makes internally.
In more detail, when using ReLU-activations, the local coordinates of every layer are each multiplied by 0 or 1, depending on their sign. From a combinatorial point of view there are 2^{D} possible combinations of such filter output vectors for D filters as each local coordinate can result in a positive or a negative number. This is where the complexity of neural networks comes from: because, omitting the activation functions would result in a single AC that just map between the coordinate systems of successive layers, and thus, the whole network could be replaced by a single global such linear coordinate transform.

The APs of a neural network thus hint at the amount of non-linearity the network uses. The previous analysis used this idea to get a simple possibility to visualize interim results of toy datasets in 2D and get a qualitative understanding of the internal structures formed in a simple multi-layer network that uses ReLU-activations in-between. Most importantly, in practice, the maximal number of binary decisions with D filters, 2^{D}, is never achieved – the analysis showed that when restricting the domain to a sampled subspace, the number of binary decisions changes substantially with network parameters: width, depth, bias and architectural choices such as batch

normalization. In contrast to the definition of global activation patterns (global APs) provided by Hanin and Rolnick [59], the experiments provided earlier examined the APs of each layer individually. The rationale for this choice is as follows: while global APs are a great tool to inspect the *theoretical capability* of a neural network, this work focuses on the *actually observed* APs given a dataset. Thus, the definition of APs in this work aims to illustrate what the network actually *computes* during training, rather than showing what it *could* do structurally. Not having the requirement of actually deriving all ACs symbolically in the input space will additionally come in handy in the next chapters: Chapter 3 will build histograms of all activations that take place during training and infer information about the training process using the patterns that are observed. Such an endeavor would be time-consuming to do in higher dimensions without restricting the analysis to a set of sampled data points as the shape of the cells would become increasingly hard to compute for high-dimensional input spaces.

To conclude, instead of measuring the total number of possible ACs, this work focuses on ACs that actually appear for a given dataset. The next chapters will show that sampling APs only using the given data points makes this idea a feasible tool for practical uses, like architecture design (Section 3.1), monitoring during training (Section 3.1 and Section 3.1), learning rate scheduling (Section 3.3), and, as a potential outlook entropy regularization (Appendix A.1).

3 Activation Patterns During Training



In neural networks' training we pry,
Where unanswered questions still lie,
With layers so deep,
Complex patterns they keep,
Discrete optimization, we apply?

— ChatGPT,
"Generate a limerick of the following text:"

The training process of neural networks continues to present an enigmatic aspect of study with many details still not fully understood. Despite initial breakthroughs, essential questions from early studies continue to resurface in contemporary analyses. For example, the phenomenon of self-organization of patterns during training has been observed early in the literature [85, 87, 45, 138]. Another open question concerns the optimal choice of hyperparameters and particularly their course during training (called *scheduling*) [166, 82, 108]. Even though practical guidance exists (as shown in [160]), constructive derivations for a good choice of such remains unattained. The observations of self-organization of patterns have sparked research aimed at better understanding the training process of deep networks [157], and, potentially even gaining some insight into the process of "learning process in general" [16].

However, “understanding the learning process of deep learning” is a challenge too general to grasp as a single research question, thus, research topics have emerged to gain insights by focusing only on a single specific component of the full process. Most recognized topics are the *Bias-Variance Trade-Off* [49, 13, 126], the effect of over-parameterization on generalization [189], and optimization landscapes and their role in network performance [27, 98]. The subtopics linked to in this chapter (each explained in detail in the next section) are much more narrow in scope: *The Early Phase of Training*, *Model Complexity Measures* and *Learning Rate Scheduling*.

This chapter starts by proposing a hypothesis, and proceeds to test it through a combination of the named research topics and the definitions of activation cells (ACs) and activation patterns (APs) provided in Definition 1.

Hypothesis 1. *With the real-valued weight optimization of stochastic gradient descient (SGD), the training process also implicitly performs a discrete optimization of the activation patterns (APs). This discrete view of training can be used to analyze the network complexity and training phases, and it can also be applied constructively to control training dynamics via learning rate scheduling.*

The last chapter already gave some hints: Chapter 2 showed that the switch-like nature of the Rectified Linear Unit (ReLU) activation results in effectively splitting the input space into domains of equal activation denoted ACs. The network uses these cells’ affine transformation to map input into output space. Thus, the deep network can be *viewed* as a discrete choice algorithm that transforms data in each layer by mapping it first to its AC and then applying the corresponding affine transformation, which in turn alters the data representation basis. The analysis has already indicated that training affects that AC-landscape, but did not inspect the dynamics of such training in detail.
As an outlook, Chapter 4 in contrast will seek to train binarized networks that do not use linear regressors internally at all. The linear part of training thus will be reduced to a minimum by utilizing stochastic numbers, where the forward pass is a completely discretized process using only addition of binary numbers and simple bit shifts, i.e. multiplying or dividing by powers of two. Thus, model space used for optimization can be completely discretized in this example.

What remains unanswered in Hypothesis 1 is a link between the discrete components of the model and its optimization.

Goals and Structure

The goal is to provide evidence that Hypothesis 1 is true. To do that, we examine the named research topics through the lens of APs and ACs defined in Chapter 2.

The contents of this chapter in more detail:

- **Section 3.1 uses the concept of APs to define measures reflecting the dynamics of training and network complexity.** In order to inspect the non-linear function space a neural network operates on, these measures utilize histograms of APs evaluated across the whole dataset. The goal is to validate the insights on the topic of early and late phases of training known from the literature.
- **Section 3.2 inspects the training process further by defining the *activation pattern temperature (APT)*, a measure of functional change in the network during a single step of optimization.** By analyzing this cheap-to-compute measure for several architectures and training methods, further links to previous findings in the literature can be inferred. This measure connects the non-linear aspect of optimization to the learning rate used for training.
- **Section 3.3 uses the APT to define a practically feasible learning rate scheduler.** This scheduler takes sensitivity changes during training automatically into account. By defining a simple monotonic decrease of APT throughout training, the "wave-like" appearance of One Cycle learning rate schedules can be deducted.

3.1 Studying the Evolution of Neural Activation Patterns during Training of Feed-Forward ReLU Networks

This section is based on the peer-reviewed article:
Studying the Evolution of Neural Activation Patterns during Training of Feed-Forward ReLU Networks (2021)
David Hartmann, Daniel Franzen, Sebastian Brodehl
Front. Artif. Intell. 4:642374.
doi:10.3389/frai.2021.642374

Stochastic Gradient Descent (SGD) does not directly optimize the discrete decisions in Rectified Linear Unit (ReLU) networks. It cannot, as the gradient of ReLU is either 1 or 0,

$$\mathrm{ReLU}\ (x) = \max\left(0, x\right), \quad \frac{\partial\,\mathrm{ReLU}\ (x)}{\partial x} = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{otherwise}^{1} \end{cases},$$

completely ignoring the distance to the decision boundary at zero.
Thus, on a small scale, a single training step using SGD either changes a data point to cross a ReLU decision boundary, or the update keeps the data point on the same side of the decision boundary. On the other hand, the ability of deep neural networks to form powerful emergent representations of complex patterns in data hinges upon the acquired ability to select suitable activations for different inputs.
Section 2.2 visualizes these decision boundaries for two-dimensional datasets for a fixed neural network state. This section aims to study the implicit *discrete decisions* made during optimization of ReLU networks using SGD.
The broader goal is to analyze the qualitative evolution of the expressivity of deep networks during training by measuring the distribution of activation patterns (APs) seen in the whole training dataset. In more detail, this section aims to answer the following specific questions regarding the training process of deep ReLU-networks.

How much information about the initialization is retained during training?
The training of a neural network begins with its initialization. The widely debated theory known as the "Lottery Ticket Hypothesis" [44, 165, 165] suggested that a network may already contain a subset of weights sufficient enough to form a sub-network with similar performance. Although the hypothesis was later found to be problematic in the conclusions it drew ([165]), the idea has sparked renewed interest in finding better initialization methods, and established discussions towards connecting the initialization to the rest of the training process. Adding to that

[1] While the gradient is not well-defined in zero, most deep learning frameworks assign a gradient of 0 at zero.

discussion, this section investigates a similar idea in the scope of non-linear functions by measuring the proportion of APs that differ from the network's initial state and how the choice of network architecture influences that percentage. The results will show a consistent (but not complete) replacement of the initial APs after only a few training steps. Though, *some* initial patterns persist throughout the entire training process, which aligns with the findings of Li et al. [97], suggesting that training occurs within a smaller subspace.

Regarding the convergence of APs in a network, where and when does structure form first?
The emergence of structure and features in a neural network has been a fascinating topic throughout deep learning history, especially since convolutional neural networks achieved considerable leaps in performance on the ImageNet dataset (e.g., [87]). When and where feature changes happen during training in the network can be answered using histograms of APs: In all experiments of this section, the non-linear structure converges *bottom-up* (i.e., lower layers stabilize first). Noteworthy, the experiments will show both PyramidNet [57] and ResNet with FixUp initialization [191], which are variants of this network family known for better performance, also provide a faster convergence of APs.

How does the functional expressivity evolve during training?
Layer-wise AP distributions act as proxies for network "linearity": For instance, the network uses much of its potential expressivity if the distribution is heavy-tailed, meaning many different APs are used. In contrast, the case of sparsely occurring APs for the whole dataset is easier to approximate with only a few affine transformations. The experiments shown in this section indicate that the expressivity increases with deeper layers upon a network's initialization, and, during training, the expressivity depends on architectural choices: For instance, Residual networks remain in a higher expressivity state during training compared to networks without residual connections. Neural networks without residual connections can experience a sudden decline in expressivity during the first few training steps, recovering only slowly with further training.

The remaining part of this section is structured as follows: Section 3.1.1 lists related work on the topic of training phases in deep learning. Section 3.1.2 introduces the framework to measure distributions of APs efficiently and defines measures that take advantage of the technique. Section 3.1.3 explains the architectures and methods used for the analysis. Section 3.1.4, Section 3.1.5 and Section 3.1.6 provide analyses targeted to answer the questions stated above. The results and their relevance in the context of related work is discussed in Section 3.1.7.

3.1.1 Related Work on Training Phases

Driven by the goal of better understanding generalization, the training process of deep network training is subject to a large body of work. One approach to analyzing that training dynamics is to separate the whole process into phases of distinct characteristics. This section is closely related to the discussion how to classify these phases of training [91, 45, 3] and extends its ideas:
By analyzing the complete training process and investigating multiple architectural choices like activation functions, ResNet variants, and learning rate schedulers the goal is to verify the observations made in the literature.
An early variant of training phases is the work of Bengio et al. [16], who showed that increasing the intrinsic complexity of data during training can help to improve generalization performance, indicating that networks possibly learn differently in early or later stages of training.
One can also argue that earlier training techniques that increase noise during training to improve performance are instances of that very same idea – noteworthy examples might be direct noise injection [127, 196], or increasing noise levels of dropout-layers [163, 122]).

Indeed, several studies identify *two distinct training phases*: Deep networks train low-frequency features first, yielding low generalization error, and continue to learn high-frequency features in a second training phase that is more susceptible to overfitting [138, 79]. Other studies have made similar observations that also take the effect of the learning rate into account [91, 96, 101].
Nevertheless, succeeding studies also note the named "experiments suggest that this [(two training phases)] is not the complete picture" [125]. While some agree on two phases, others show evidence of *three training phases* instead of only two [45, 96].
To summarize their discussion, there appears to be consensus in the literature that at the beginning of the training, the activations of a network mainly perform a random walk [45, 65].
The other widely accepted fact relates to the end of the training, where momentum becomes increasingly important [91] as gradients directions simplify [56] and the loss landscape flattens [11]. The end of training enables a deeper study more easily, because only few activations typically change on every optimization step. Hoffer et al. [65] have shown that this comes from the loss landscape getting smoother the farther the weights travel from initialization.

This section will utilize APs defined in Definition 1 to add to that discussion.
The closest related work that also takes APs into account is probably that of Montúfar et al. [120] and Hanin and Rolnick [59], where the authors analyze the capabilities

of neural networks in terms of expressivity. They show that neural networks use much fewer APs during training than theoretically possible. In contrast to this section, they did not take the input distribution into account but analyzed the whole theoretical input space. Additionally, their work provide upper and lower bounds for APs of the static model in general – instead, the observations of this chapter consider the dynamical changes of APs during training in practice.

3.1.2 Notation, Measures & Efficient Counting

APs can be utilized to analyze structural changes that occur during training of ReLU networks. The approach is similar to the experiments described in Section 2.2, where APs were the focus of investigation for a fixed input domain. But in contrast, this section focuses on collecting *all* APs that occur when the entire training dataset is transformed through the network.

This approach enables a qualitative understanding of the network's expressivity, as the set of APs measured represents the number of distinct affine transformations used to transform the entire dataset. (Conversely, APs that only exist for data samples far outside the expected distribution are thus ignored by design). Therefore, the *cardinality of the set of APs* indicates how "linear" the layer-ReLU combination is, given the input dataset. This and other measures that use APs, their distribution, or their state at certain training stages, are defined as outlined below.

Notation

A short recap of the definition of APs from Section 2.2: The following definitions consider deep networks with piecewise-linear activation functions $\sigma(\cdot)$, specifically $\sigma = \mathrm{ReLU}$,

$$F : \mathbb{R}^{d_1} \to \mathbb{R}^{d_{L+1}}.$$

We split the forward-pass of neural network at a specific layer $l \leq L$ into the part before, F_l, and the part after, G_l, the activation function σ:

$$F(x) = G_l\left(\sigma\left(F_l\left(x\right)\right), x\right). \tag{3.1}$$

It's worth noting that the G_l function takes the input implicitly twice, which allows for more complex networks, for instance those including residual connections, to be examined.

Definition 4. *Activation Pattern Count*
Section 2.2 has introduced the AP on layer l of input x *as diagonal matrices*

$$A_l(x) := (\delta_{>0} \circ F_l)(x) := \operatorname{diag}(a_j) \quad \textit{where} \quad a_j = \begin{cases} 1, & \textit{if } F_l(x)_j > 0 \\ 0, & \textit{else} \end{cases}. \tag{3.2}$$

In addition to that, the count of an AP $A \in \operatorname{Diag}_{d_l}(\{0,1\})$ on layer l under the finite input dataset Ω *is defined as the number of its occurrences over the whole training set,*

$$c_l(A) := |\{x \mid A_l(x) = A, x \in \Omega\}|. \tag{3.3}$$

The maximal number of theoretically possible APs on a particular layer l is $N_l := 2^{d_l}$, where d_l is the output dimension of F_l.
Knowing that number, all activations can be listed in a vector containing the number of occurrences of each activation: Let $(A_{l,1}, A_{l,2}, \ldots, A_{l,N_l})$ be an arbitrary but fixed ordering of all possible APs on layer l. Then, the occurrence vector of all APs on layer l under the input dataset Ω is

$$c_l := (c_l(A_{l,1}),\, c_l(A_{l,2}),\, \ldots,\, c_l(A_{l,N_l})) \in \mathbb{N}_0^{N_l}. \tag{3.4}$$

The notation $c^{(i)}$ denotes the i-th component of the activation count vector c.

Set Measures

The following measures use just the information if an AP exists in a given activation state c.

Cardinatilty of the Activation Pattern Set: As motivated before, the cardinality of the pattern set gives a broad view of how many distinct functions transform the incoming data. The measure is similar in spirit to the "number of regions" shown in Section 2.2. The *total (distinct) pattern count* is thus defined as

$$\operatorname{total}(c) := \sum_i \delta_{>0}(c_i), \tag{3.5}$$

where $\delta_{>0}(x)$ is 1 if $x > 0$ and 0 otherwise.
While there are N_l potential APs, as observed in Section 2.2, much fewer APs can be measured in practice for a typical neural network.

Activation Pattern Changes: Related work that provided experiments regarding structural work in the early training phase of neural networks measured sign changes

and magnitude changes of weights and their gradients (see Section 3.1.1). AP counts do not consider weights directly, but can still be utilized to quantify explicit changes in a network's structure. The *AP changes* does so by defining the relative cardinality of the symmetric difference of the AP counts at two network states.
In more detail, this metric specifies the number of patterns that are present at training step $s+1$ but have not been present in the preceding step s and vice versa, the sign specifies which of these two happened more often: Let $c_{l,s}$ denote the AP count on layer l of a particular activation state after s training steps.

$$\text{changes}\,(c_s, c_{s+1}) := \frac{\sum_i \delta_{>0}(c_s^i) - \delta_{>0}(c_{s+1}^i)}{\text{total}(c)}. \tag{3.6}$$

Using Similarity of AP Sets to Estimate Non-linear Convergence: Similar in spirit to the AP changes, set-distances can be used to check for AP overlaps between two sets of activation states. The idea is to measure the extent two such sets intersect, taking the value 0 if the two sets of measured APs X and Y do not intersect and the value 1 if they are equal.
The *Jaccard Index* of the two pattern sets, $JI\,(X,Y) = \frac{X \cap Y}{X \cup Y}$, would reflect this idea. However, as counts for each set-element (AP) do exist, the definition below uses the *weighted Jaccard Index* that takes the number of patterns into account:

$$J_{\mathcal{W}}(c, c') := \frac{\sum_i \min(c_i, c_i')}{\sum_i \max(c_i, c_i')} \in [0, 1], \tag{3.7}$$

The measures equals 0 if the two occurrence vectors c and c' have no patterns in common, and it equals 1 if their occurrences match.
Two applications of this idea are of particular interest for this analysis: First, the distance of the current training state to the state of initialization estimates the number of initialization patterns, still present during any given point of training: The measure is defined as taking the Jaccard Index of the current training state s and the initialization state s_{init} (before training),

$$J_{\mathcal{W}}(c_s, c_{s_{\text{init}}}).$$

Second, the convergence of the APs can be measured by computing the similarity to the APs of the final training step,

$$J_{\mathcal{W}}(c_s, c_{s_{\text{final}}}),$$

where $c_{s_{\text{final}}}$ denotes the last AP count vector after training has completed. To retrieve the final APs in practice, the experiments involve to first train the network, save

the AP count vector of the final network state and restarting training with the same seed. This measure represents the convergence at which the non-linear part of the network reaches its final structure.

Distribution Measures

The next measures take the distribution of the APs into account.

Most frequent pattern count: The *frequency of the most frequent pattern* is simply the fraction

$$\operatorname{maxFreq}(c) := \frac{\max_i c_i}{\sum_i c_i}. \tag{3.8}$$

In case the number is high, the layer only uses either only a few patterns altogether or the layer uses a single AP is used very often, potentially reflecting the size of the central AP (see Figure 2.6); in other words, the layer is primarily using linear computations except for some data points.

Activation Pattern Entropy: The most frequent pattern count may be misleading in some cases. To illustrate this, consider the special case of few distinct available APs, each distributed uniformly. The most frequent pattern count may be higher in this case, yet the measure gives no intuition about the distribution overall.
The following definition of the activation pattern entropy (APE) on a given layer addresses this by taking the distribution of the APs into account:

$$H(c) := -\sum_i \frac{c_i}{N_l} \cdot \log_2 \frac{c_i}{N_l}. \tag{3.9}$$

Like the pattern set's cardinality, the APE indicates how many functions the network uses to process the dataset up until a given layer but is highest if all obtainable patterns appear equally frequent and lowest if only one pattern exists.
Thus, if the APE in the last layer of a network is very low, the whole network can be approximated with only a few linear maps. The network uses the most non-linear capabilities to compute the data if the number is the highest.

Relative APE: The APE is not a measure suited to compare different layers in a neural network. For instance, a linear layer with d filters followed by a ReLU-activation can combinatorically use at most 2^d APs, limiting the value for the APE to a maximal value of d. Consequently, one combination of linear layer and a ReLU activation with few filters but equally distributed APs could yield a smaller value of APE than another such combination with many filters of skewed distribution. Additionally, the number of data points equals the total APs. A training dataset consisting of n

Algorithm 2 $\text{count}_{\text{act}}$
This function uses a hash table to count the occurrences of all unique APs on the GPU. Inputs are the $n \times d$ activation tensor, the hash list size, and the data type of the hashlist.

```
import torch
import np

def get_hist_torch(act, hashlist_size=30000, dtype=torch.int32):
    assert act.dim() == 2 # shape: (num patterns, num filters)
    n = act.shape[0]
    ones = torch.ones(1, device=act.device, dtype=dtype).expand(n)
    counts = torch.zeros(hashlist_size, dtype=dtype, device=act.dev:
    hashes = hash(act) # hash activation patterns to key
    key = hashes % hashlist_size
    counts.index_add_(0, key.view(-1), ones)
    return key, counts
```

examples, for instance, thus also limits the entropy to a maximum value of $\log_2(n)$. Therefore, to achieve comparability in-between layers and in-between experiments, APE is normalized as follows, denoting the new measure *relative APE*,

$$H_r(c) := \frac{H(c)}{\min\left(d, \log_2(\sum_i c_i)\right)}. \tag{3.10}$$

Efficient Activation Pattern Counting

Measuring or allocating the full AP count vector is not feasible in real-world scenarios. For instance, while 16 filters would require only about 8 kilobytes of memory on the system, already 32 filters would require about 0.5 gigabytes of memory to store all possible pattern vectors in memory, and a more actually used number of filters of 64 filters would require already about 2.25 exabyte of memory. The better alternative would be to use a dictionary, as the number of data points in the dataset is the limiting factor of the number of measured APs. However, this technique would significantly hinder the measurement due to the need to either implement the dynamic dictionary on the GPU, or by copying the APs found during training to the host machine's RAM, at the cost of slowing down training considerably.
Thus, instead of listing all possible patterns, this thesis uses hash lists of fixed size for every layer. Algorithm 2 presents the code used to query a hash list of AP counts efficiently on the GPU. Occupancies of the hash lists are also given in the analyses to enable discussion regarding the validity of the results.

3.1.3 Experimental Setup

To analyze the emergence of non-linear structure given by APs in a neural network, the following section will inspect the evolution of the measures given in the previous section, Section 3.1.4. To maintain comparability while looking for answers to the questions given in the introduction of Section 3.1 all experiments use shared settings.
We measure the APs that occur when training typical convolutional neural networks to classify the CIFAR-10 dataset [86] and the Tiny ImageNet dataset [90].
While listing all potential APs (to initialize such a vector of counts) directly is infeasible, this problem can be solved by using a hash list of the found occurrences. Thus, during the training process, the Algorithm 2 retrieves the AP count vector on a per-layer basis. More specifically, *all* APs that occur during the training of neural networks are taken into account. To ensure non-invasiveness of the measurements, model parameters and optimization parameters are frozen after each optimization step of training. Following this, the AP count vector (Equation (3.3)) are retrieved on a per-layer basis.

To reduce hash-key collisions as much as possible, the hash list sizes on each layer are increased by hand, fitting the maximal capacity of 80 GB GPU-RAM to fit the GPU memory, while also maintaining a low occupancy for layers that undergo many different patterns. Due to the memory limits the occupancy of the layers are balanced while keeping the occupancy as low as possible. For most shown experiments, the occupancy of the layers remains below 10%.

PyramidNet-20, ResNet-20, ConvNet-20 and ToyNet-20

The research in different activation functions has been vast; the most commonly used piecewise linear activation functions for image classification are ReLU [124] and PReLU [61], which are both checked in the following experiments. Architecture-wise, plain convolutional neural networks have been replaced by ResNet variations [62, 169]. Also, several research papers have already focused on why residual connections result in higher-quality networks [10, 129]. Two slightly more recent variants of ResNets are PyramidNet [57], which uses a gradually increasing number of filters, and FixUp [191], which optimizes the different weight types in a network separately and changes the initialization scheme. The experiments will compare these architectural decisions in terms of APE and AP convergence.

The ResNet-20 model in its v2-CIFAR-Variant as described by He et al. [62] is used as a baseline. Three architectural features used in that ResNet architecture are tested for their behaviour through the lens of APs, thus, the experiments denote the following variants of the network: First, the structural effect of the residual connections (also known as *skip-connections*) are to be examined. In more detail, let *ConvNet-20* denote a variant of ResNet-20 with it's residual connections removed.
Second, ResNet-20 increases filter counts with layer depth (16 filters in the first convolutional block[2], 32 filters in the second block, and 64 filters in the last block respectively). allowing more "high-level" features than "low-level" features. To test the effect that architectural feature has on structural changes of training dynamics, let *ToyNet-20* denote a variant of ConvNet-20, but with equal filter counts (32 filters) throughout the whole network.
In addition to these variants, some experiments will also include PyramidNet-20[3] [57], that has shown increased performance by introducing a smoother increase of filters with layer depth, and FixUp-20, an adaption to 20 layers according to the original FixUp-Network suggested by Zhang et al. [191].

Batch normalization layers in these named networks are set to their dynamic *training* mode of (i.e., the layers will use the actual mean and variance of the batch results for normalization). The reason is to mimic the internal structures that also appear during training.

The four network variants use average pooling and a classification layer each as the final layers – pooling layers are only implicitly captured by experiments, and in this case with average pooling layers at the end of the network, the effect of average pooling is thus not captured at all.
The reason is simple: the analysis takes only results preceding activation layers into account to measure the number of affine transformations a linear layer-ReLU combination "applies" to data – average pooling as well as the final classification layer do not change these measures, thus, have been excluded from the analysis.

[2] In this context, a convolutional block denotes the side-branch of the ResNet architecture – that is the sequence of convolutional layers, normalization layers, and non-linear activation functions – excluding the residual connection.
[3] The definition in the original paper employs 200 layers, but employs a formula how filter sizes increase. Thus, to enable comparability with the other architectures, a 20 layer variant is implicitly defined by their formula, which calculates the filter size based on the depth of the respective layer.

3.1.4 Structural Changes in the Early Phase of Training

The first experiment tracks the AP distributions during the first 3000 iterations of the training process on the respective dataset using the ResNet-20, ConvNet-20, and ToyNet-20 architectures. Each architecture uses the same hyperparameters that have been optimized for ResNet-20 achieving a test accuracy of about 91% without additional data. In each training step and layer, the total number of patterns are evaluated, the number of pattern changes, the weight of the most frequent pattern, the APE, and the weighted Jaccard Similarity to the pattern occurrences at the initialization of the network.

The idea of this experiment is to inspect the training of convolutional neural networks through the lens of APs. As a start into the topic some observations aim to extend the learning from Section 2.2 and previous work on the early phase of neural network training; Previous works have already identified the early phase of training to have training behavior that differs from the rest of training process (see Section 3.1.1). The goal is to validate the observations from the literature on the early phase of training: These include rapid sign changes, converging gradient magnitudes, the alignment of momentum that indicates a more linear rather than non-linear learning phase, and the finding that corrupt data (bad batches) might cause irrevocable damage. Additional goal is to analyze how much of the structure of neural networks given by its random initialization is maintained throughout training.
Figure 3.1 shows the outcomes of each metric, one per column, across all layers which are plotted along the y-axis, and evaluated over the entire training dataset at each of the first 3,000 training steps, shown on the x-axis. The following text will go through each column individually.

Total Pattern Count: In Figure 3.1(i), we observe that the *absolute number of patterns per layer* differs by several orders of magnitude for each architecture. While the number of patterns ranges from maximally 22 million patterns in the first convolutional block to about 2 million patterns in the last block of the ToyNet in the initialization state of the network, the number of distinct patterns drops significantly after the first step to maximally 3 million patterns. The number of patterns remains constant in the last convolutional block. The ConvNet-20 and ResNet-20 models behave differently: the first blocks in both models have the fewest distinct APs (due to only 16 filters per layer instead of 32 in the first layer), and the middle blocks have the most APs in the later stages of training. Measuring the *total (distinct) pattern count* validates the observations in Section 2.2 and the work of Hanin and Rolnick [59], which have shown that ReLU activations are rather sparsely distributed. According to their observation the number of "activation regions" (or APs in our

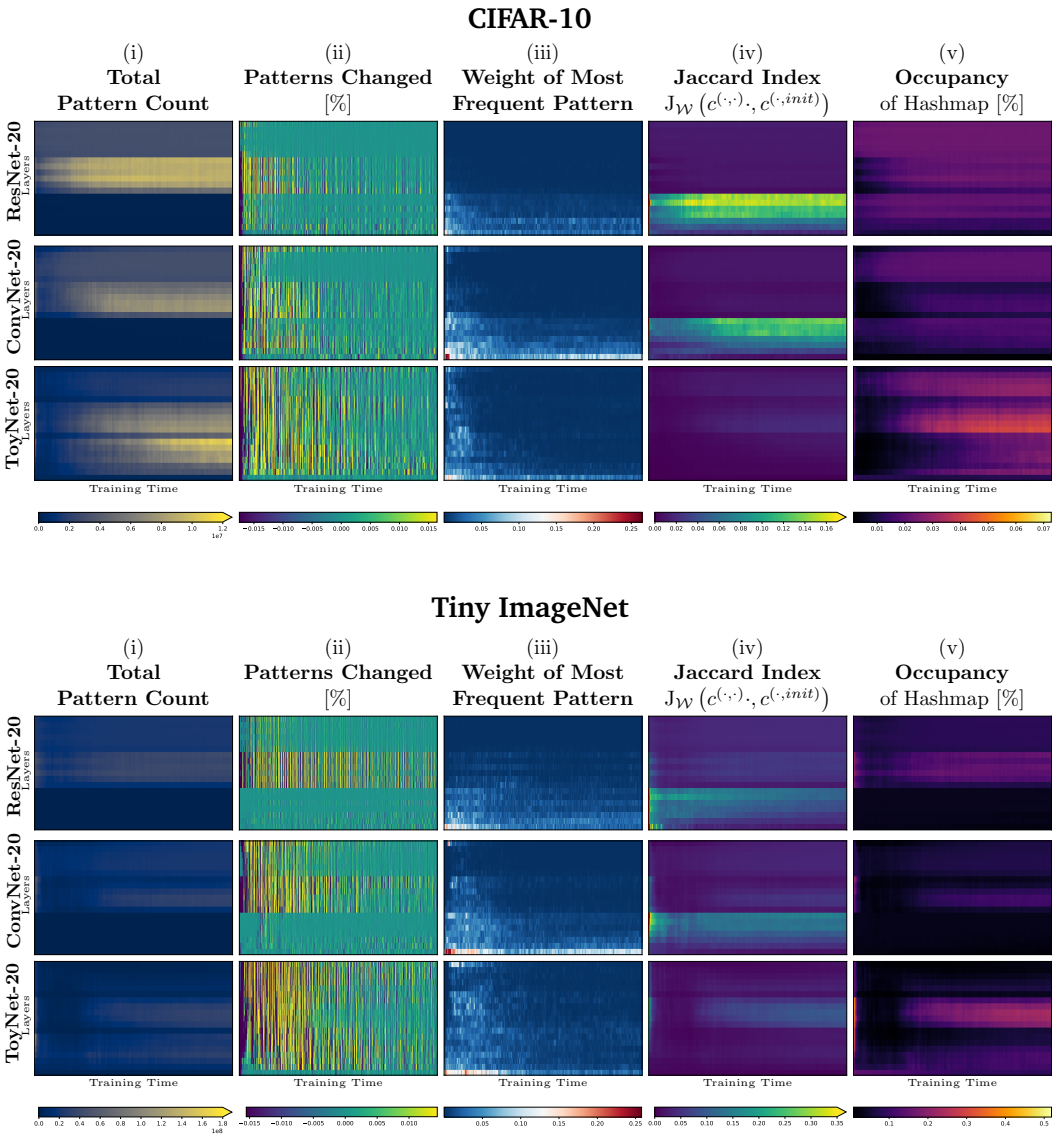


Fig. 3.1.: The early training phase of ToyNet-20, ConvNet-20, and ResNet-20 (first 3000 iterations) on CIFAR-10 (top figure group) and Tiny ImageNet (bottom figure group). The top row of each individual plot represents the last convolutional layer before the linear classification layer. The bottom row respectively corresponds to the first convolutional layer. The columns, from left to right, show the measurements: (i) the size of the AP set, relative to the maximum size seen during training; (ii) the change of size of this pattern set, truncated for better visibility; (iii) the relative frequency of the most frequent AP; (iv) the weighted Jaccard Similarity comparing the current APs to those directly after network initialization, and (v) the hash map occupancy used. Red pixels indicate outliers, visualized as out-of-range values to maintain the plot's readability and proper color range scaling.

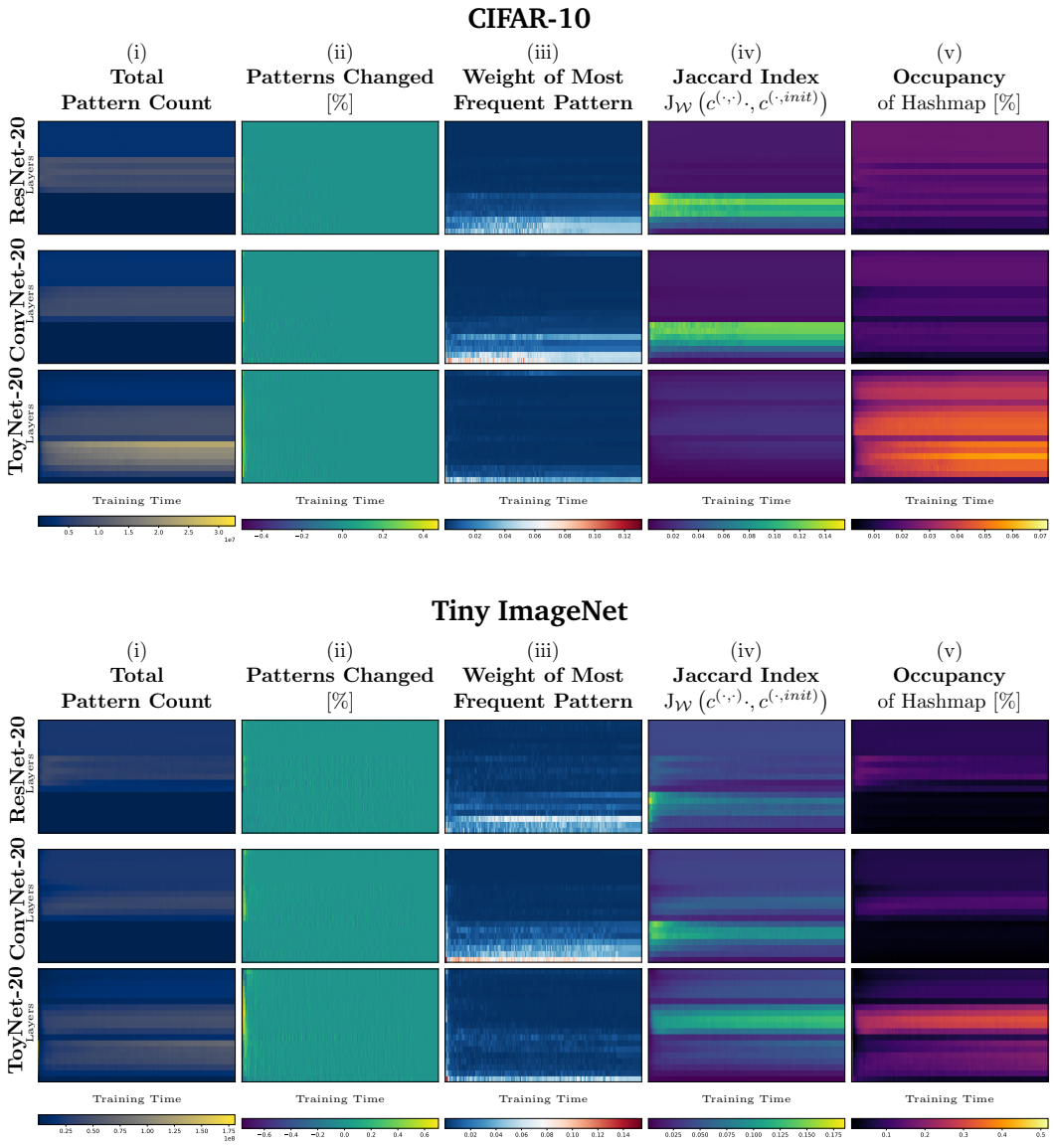


Fig. 3.2.: The full training phase consisting of 200 epochs for CIFAR-10 (top figure group) and 80 epochs for Tiny ImageNet (bottom figure group) of ToyNet-20 (top rows), ConvNet-20 (middle rows), and ResNet-20 (bottom rows). The plots correspond to those in fig. 3.1, showing from left to right the measures: total pattern count, patterns changed, weight of most frequent pattern, the Jaccard index between the current network state and the initial network state, and the hash map occupancy.

notation) also drops here first after initialization and increases slowly with training again.

Patterns Changed: Figure 3.1(ii) shows how many pattern changes occur at each layer for each training step. In accordance with related work that analyzed sign changes of weights or pre-activations [45, 137], most pattern changes occur during the first few steps in all experiments tested in this section.
Interestingly, during training, pattern changes may strike from the lower to the top layers simultaneously – vertical nearly solid lines at some steps during training indicate changes throughout the entire network, suggesting significant changes happened in the calculation tree in lower layers. In particular, these strikes often happen in pairs: new patterns first appear and then approximately the same amount of patterns disappear again or vice versa, this might be evidence for unfavorable batches for training. These strikes appear in all networks types tested. For ToyNet, most APs disappear from the pattern sets in the first training steps. The ResNet, in contrast, shows a more stable distribution over pattern set changes. This potentially gives a hint why ResNet architectures are easier to train.

Most Frequent Pattern Weight: Figure 3.1(iii) shows the weight of the most frequent pattern per layer and training step, observed in the whole dataset. A high-density pattern would hint at a non-uniform distribution of APs. The most frequent patterns arise for the ToyNet variant, for example, only in the first hundred training iterations or the first activation layer. In contrast, the ResNet network has the fewest high-density patterns in the first few hundred steps during training throughout the network. Remarkably, for the ConvNet variant, some layers have patterns that account for up to 25% of all observations, indicating possibly unnecessary structures inside the network. However, these high-density patterns occur only intermediately – after a few hundred training iterations, the most observed patterns in all layers, except for the first layer, only account for fewer than about 5% of all pattern evaluations. They even account for fewer than 0.001% of all pattern evaluations in deeper layers, giving potential new insight for pruning methods (such as [192]) that take combinatorial frequencies of activations into account.

Jaccard Index (Similarity with the Initialization State): Next, we inspect if and to what extent the APs of the initialization remain during training. The goal is to determine what kind of architectural features may utilize the initialization structure instead of using it only as a starting point and changing its nature completely. Thus, Figure 3.1(iv) shows the weighted Jaccard Index between the pattern occurrences of the initialization c_0 (for each layer) and the current training step. In the first few steps, the similarity to the initialization is unsurprisingly highest. Initially, the highest

similarity to the initialization states can be observed in the first few steps of training, which is not surprising. However, it then drops to 0% for the ToyNet architecture and to about 10% for the ResNet architecture within only few optimization steps. Compared to the other two analyzed models, the ToyNet-20 model has the fewest overlap of APs to the initialization. For this architecture, the similarity first drops to about 3%, and, during training, this number increases again to 10% in the later stages of training. This can be seen in Figure 3.2 that shows the same experiment, but "zoomed-out": instead of focusing on only the first 3000 optimization steps, this experiment illustrates the training behavior for the full training process over 200 epochs for CIFAR-10 and 80 epochs for Tiny ImageNet. The increase from a 3% similarity to a 10% similarity towards the end of training indicates a rediscovery of some patterns that have already been present at the initialization. Despite having a similar architecture, ConvNet-20 and ResNet-20 behave differently than the ToyNet-20 model.

In conclusion, while the pattern set of the initialization is mostly lost within a few training steps, ResNet-20 and ConvNet-20 "rediscover" a good proportion of the initialization within the first 3000 iterations. While patterns that emerge near the input of a network tend to represent structurally simpler features from what we know from literature [187], their "rediscovery" of seem to depend on the architecture used.

3.1.5 Architecture-Specific Entropy Curves

How does functional expressivity evolves over time? The APs of a network directly relate to the number of affine transformations that can be used to describe the neural network layer and the dataset used. By inspecting these patterns during optimization, the previous section demonstrated that structural changes occur during training. The next experiments aims to relate the respective layers in each architecture to the expressivity emerging at a particular moment in training time - the measure that rates the expressivity of an activation layer in this sense is the relative APE: A low expressivity layer (i.e., having low APE) corresponds to being more easily approximated by only a single linear layer. Easily in this context means that using the "most used affine transformation" would result in a low approximation error. In contrast, a high expressivity layer (i.e., having high APE) behaves more like a hash table in the sense that it has enough expressive power to map a unique affine mapping to each input data point.

Experimental Setup: Again, the three network variants ResNet-20, ConvNet-20 (ResNet-20 without residual connections), and ToyNet-20 (ResNet-20 without resid-

ual connections and constant filter sizes) are to be examined while being trained on CIFAR-10. A separate experiments is conducted for Tiny ImageNet. Figure 3.3 shows the mean relative APE (y-axis) for each architecture and for each layer (x-axis). The columns in the figure represent the three ResNet variants: ResNet-20, ConvNet-20, and ToyNet-20. Each row corresponds to a specific point in training: (i) the initialization state of the networks, (ii) after a single training iteration of SGD, (iii) after 40 training iterations, yet still at the beginning of the early phase of training (see Section 3.1.1), (iv) after 2 epochs of training, (v) after 70 epochs of training. Each architecture is evaluated in multiple variants: with $2, 3, 4, \ldots, 10$ convolutional layers per convolutional block, each of indicated in the figure by a different color.
As deep learning experiments are inherently stochastic, with random initialization of the weights and random order of training examples, one can expect a certain variance of results for different seeds. However, this is not the case: Figure 3.3 visualizes the *mean* relative APE of 25 training runs of each network, the lighter colored hoses around the graphs indicate the single standard deviation of the 25 measurements using different random seeds but equal parameters otherwise.
The same experiment, but trained on a different dataset, Tiny ImageNet, is shown in Figure 3.5. Additionally, for both sets of experiments, there exists plots showing the hash map occupancy for the respective measurements. The following paragraph will solely describe the figure in the early phase of training, and the final training phase, the discussion of the observations follows in Section 3.1.7.

Observations: For all network variants, the relative APE increases with deeper layers. Its value ranges from about 0.35 (less then half of the maximal possible expressivity) to about 1 (the maximal expressivity for the respective layers).
The ToyNet variant has the most evident increase of APE in the initialization, the ConvNet runs into saturation at the end of the second block, which increase again within the next block. After only a few steps of training, the APE decreases across all network types. Notably, the ToyNet variants experience the most significant decrease in absolute terms, while the ResNet variants show the least decrease.
At step 40 of training (third row), the APE further drops for all network types, specifically for ConvNet and ToyNet, showing the most significant standard deviation. Especially deeper networks have a more extensive spread in their measurements than shallower nets in this setup. Noticeable is the characteristic "zig-zag" structure for the ResNet variants that starts with the initialization and becomes more apparent throughout training. Specifically for this variant, layers before residual connections have lower APE than layers after residual connections. As the plot transitions from shallower to deeper layers, there is a noticeable global upward trend in the APE measure.

In the final stage of training (last row in the figure), the standard deviation of APE decreased again to a similar level as in the initialization phase. The zig-zag pattern of the ResNet variants has strengthened slowly in the first block, while in the last block it has saturated to a near-maximal APE again. The ConvNet variants and in particular the ToyNet variants show a saturating increase of APE in the first block, a slight decrease of APE in the second block, and a decrease of APE in the last block. Regarding the three convolutional blocks of the network variants, the levels of APE increase from shallower blocks to deeper blocks.
To summarize, the results suggest that from the three variants tested, the ResNet architecture maintains its expressivity throughout training. In contrast, the missing residual connection (ConvNet and ToyNet) result in a drastic loss of expressivity in the first few training steps. With training, however, this expressivity slowly increases towards their initial magnitude. The second feature of ResNet, the increasing number of filters, results in high-expressive final layers. In the case of ToyNet, which does not include increasing filter sizes, the APE of the final layers decreases throughout training. This might be important for the cascading nature of deep networks, as they combine features of increasing complexity until the final layers combinatorically accumulate many partial results for a final decision.

Section 3.1.7 discusses the observations above and compares them with previous experiments in this chapter, offering a broader perspective on the overall results.

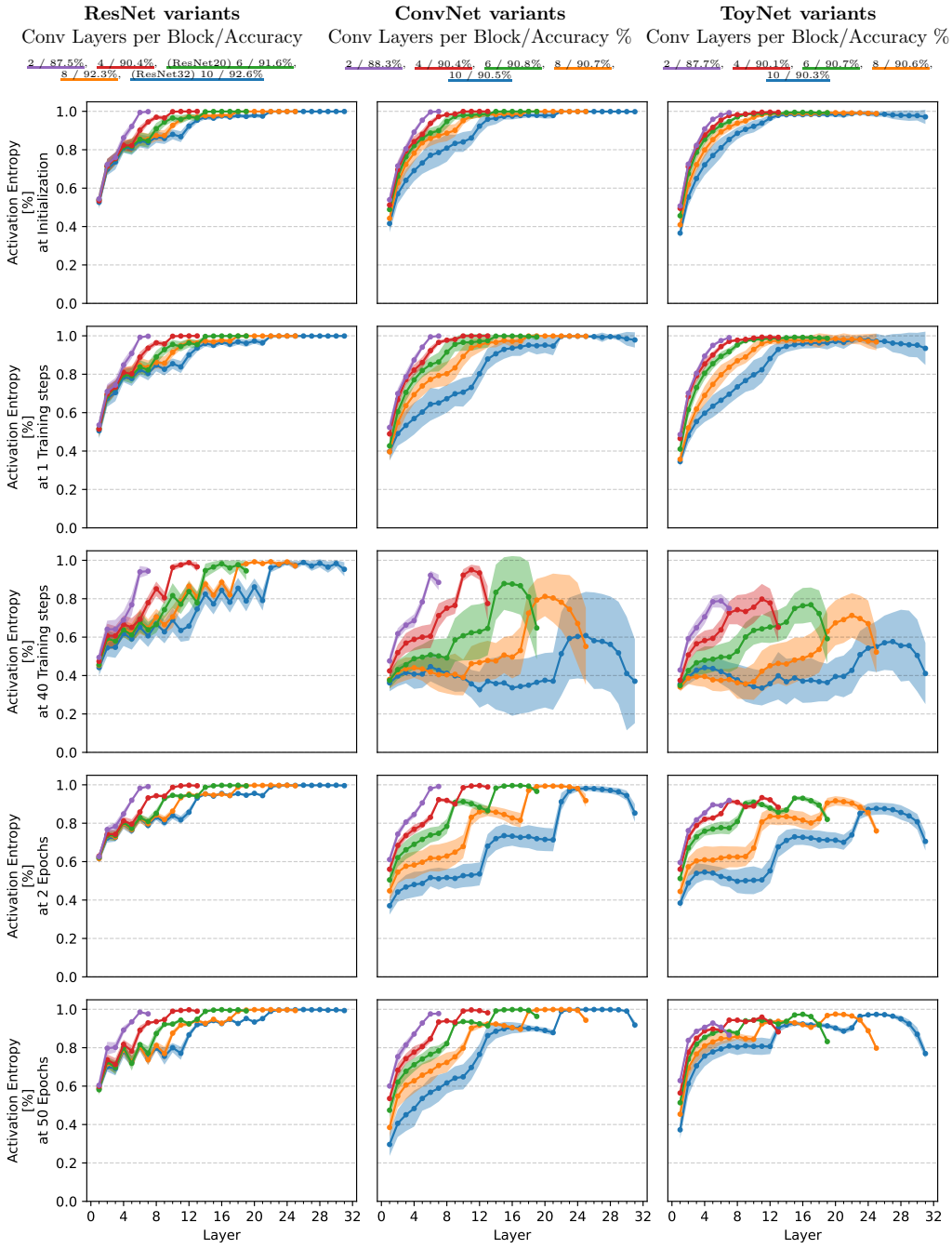


Fig. 3.3.: Mean relative APE for ResNets and ConvNets of various depth, measured for an ensemble of 25 training runs per network type. The measurements were taken separately for each layer at different points of the training process: (i) at initialization; (ii) after a single steps of SGD; (iii) after 40 steps of SGD; (iv) after two epochs; and (iv) right after the first learning rate drop at epoch 50. The shaded area indicates the variance spanning one standard deviation in each direction. The ResNet variants use the architecture proposed for CIFAR-10 by [60] and pre-activation residual connections [62], with a varying number of convolutional layers at each layer block, indicated by the different colors. The ConvNets networks use the same basic architecture but without the residual connections. ToyNet additionally does not increase number of filters.

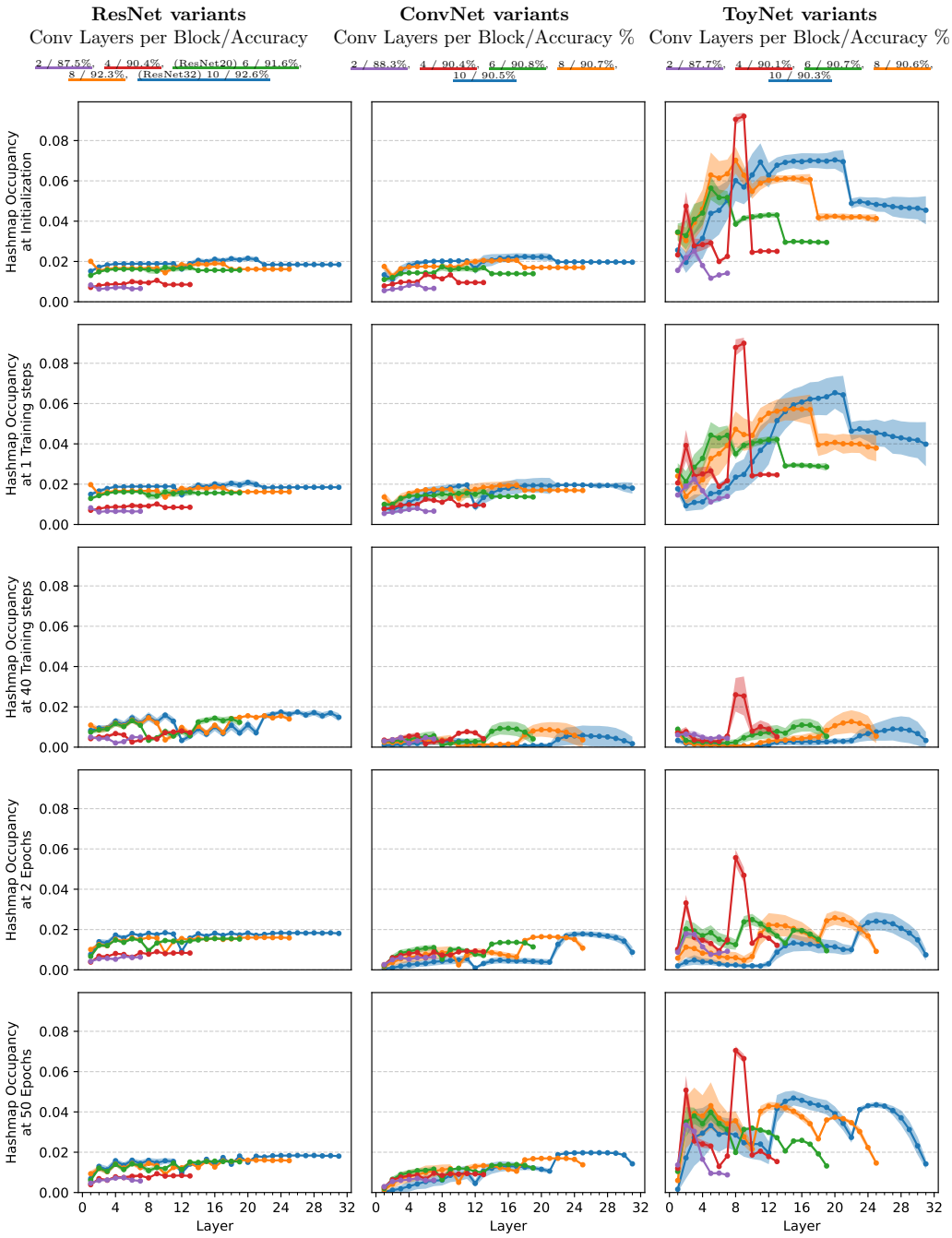


Fig. 3.4.: Companion plot for Figure 3.3: This plot shows the mean occupancy of the hash maps used to compute the activation entropies.

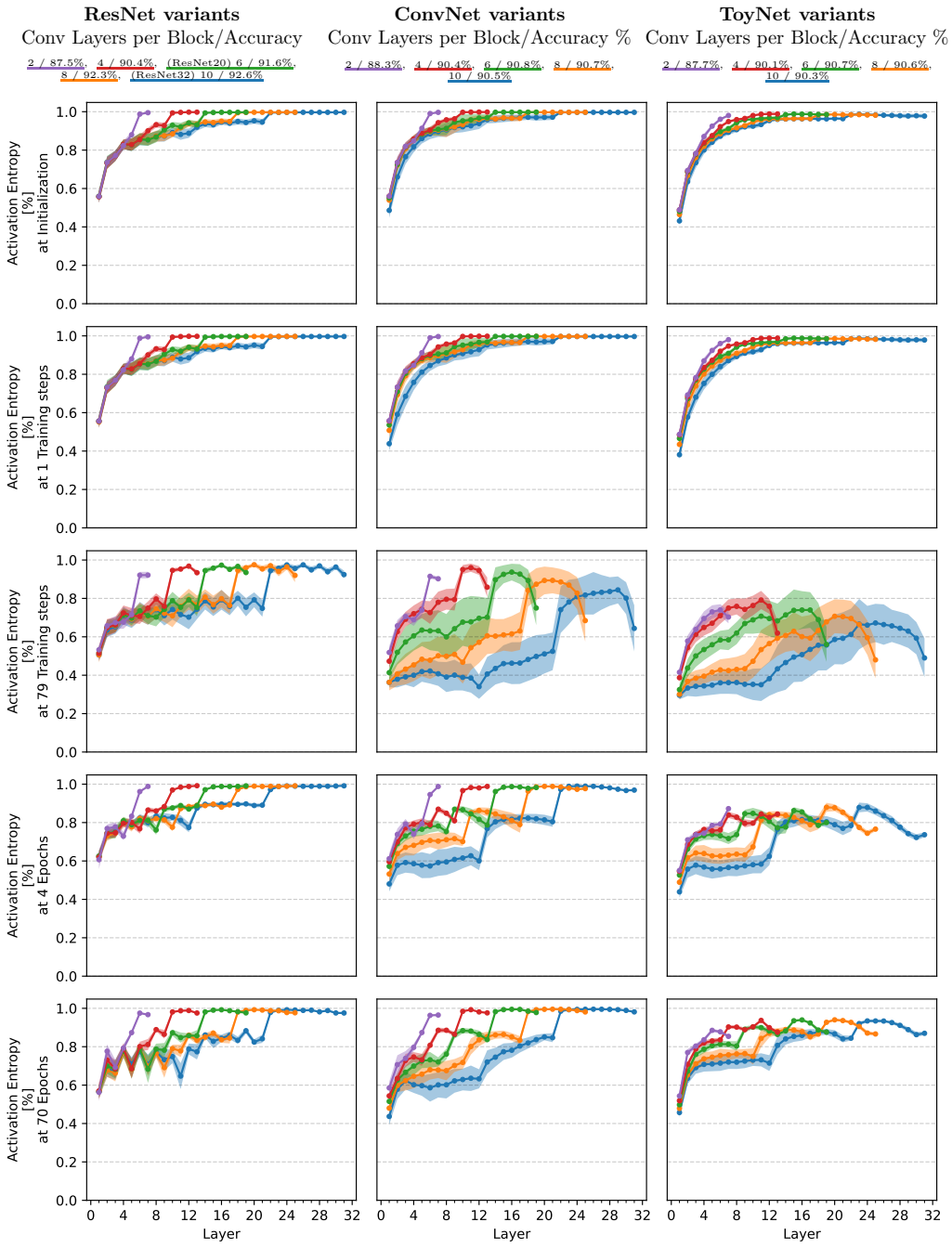


Fig. 3.5.: Same experiment as Figure 3.3, but trained on Tiny ImageNet.

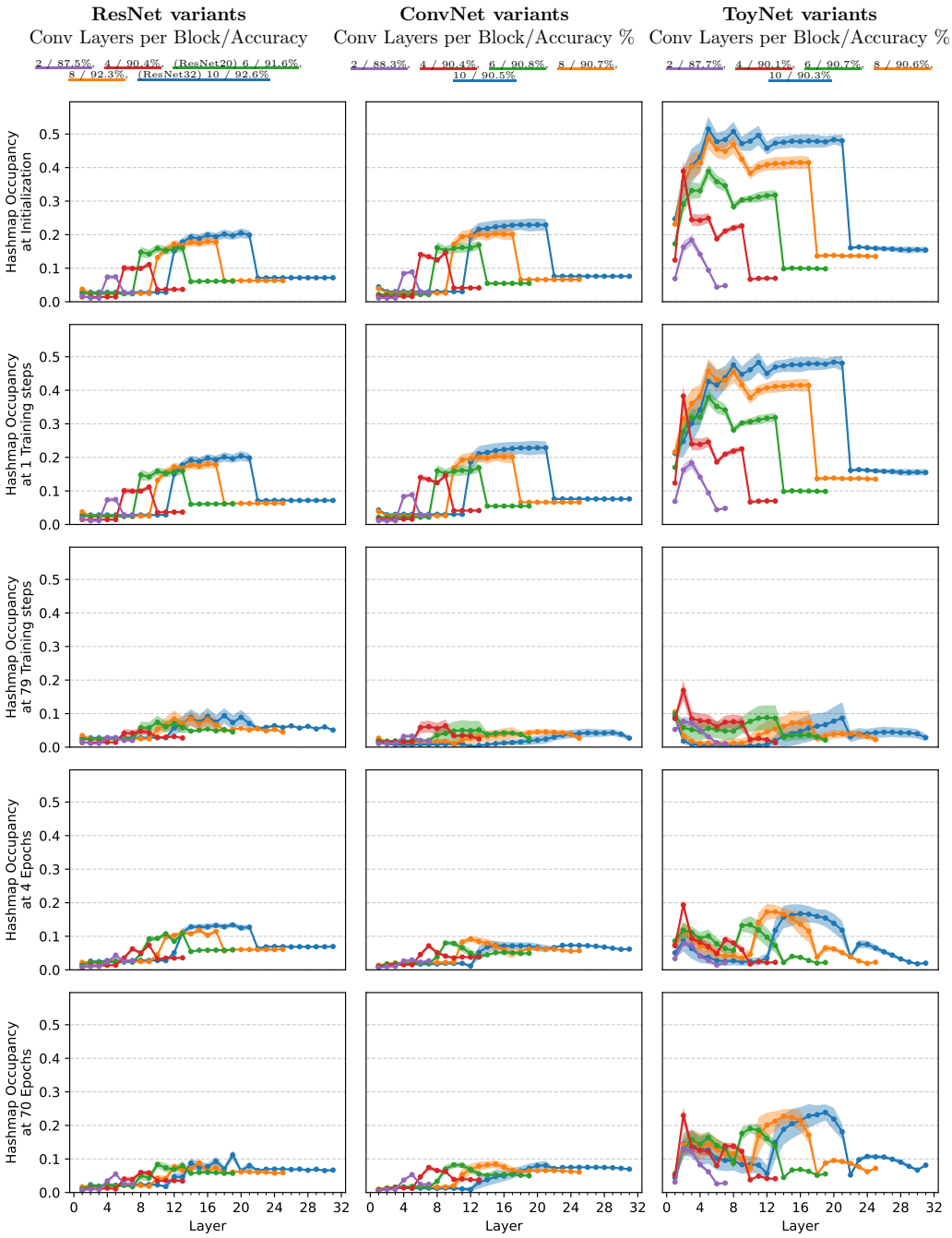


Fig. 3.6.: Companion plot for Figure 3.5: This plot shows the mean occupancy of the hash maps used to compute the activation entropies.

3.1.6 Structural Convergence of Architectures & Training Methods

The next applies two of the measures defined in Section 3.1.4 during training of several methods that are known to improve image classification performance.

Setup: In detail, for small ResNet networks the effect of ReLU [124], parametric ReLU (PReLU) [61], cyclical learning rate schedules (multiple and one cycle, Smith [159]) is evaluated. The architectures used are ResNet-20, ConvNet-20, PyramidNet and FixUp [191]. The measures used are the APE and the Jaccard similarity of the current training step to the final network state, $J_{\mathcal{W}}\left(c^{(\cdot,\cdot)}, c^{(\cdot,\text{final})}\right)$. The APE gives an intuition of how much of its non-linear capabilities a network uses per layer. The second measure, the Jaccard similarity to the final network state, represents the activation-wise convergence of the networks per layer. As described by Hanin and Rolnick [59], APs appear only sparsely – still the number of potential filters rises exponentially as the network depth increases. As before, the occupancies of the layer-wise hash lists are also collected alongside the measurements to validate the measurements in terms of possible collisions. To maintain comparability with ResNet and ConvNet, PyramidNet and FixUp have been adapted to use 20 layers. This results in fewer activation layers for PyramidNet, thus fewer rows in the resulting heatmap plots, as the measures target activation layers exclusively. For the setups labeled "StepDecayLR", a step decay learning rate schedule is used, decaying the learning rate after precisely the first half of the entire training time by a factor of 10, and then again by a factor of 10 at the start of the last quarter of the training. As in the previous section, the following text starts by describing the results in detail. It begins by discussing the outcomes for the APE measurements, followed by the outcomes for the Jaccard Index. The observations are then analyzed and presented in a broader context in Section 3.1.7.

Observations for the APE: Across all tested network variants the APE increases with increasing network depth. In the initialization state of training, APE achieves larger values (as described in Section 3.1.5), but just as the total pattern count in the previous experiments (illustrated in Figure 3.1), drops for some layers after only a few steps of training. The layers that show a reduction in entropy in the early training phase slowly regenerate entropy as the training proceeds. With the learning rate reductions at 50% and 75% of training, the APE in the last layer briefly rises again. Exception is the PyramidNet-20 network, which is discussed later in more detail.
Remarkably, the cyclical learning rate schedules, "CyclicLR" [159] and "OneCycleLR" [160], dampens the effect of APE drop in the early phase of training. Nevertheless, the initial decrease of APE still occurs in the middle layers of the networks.

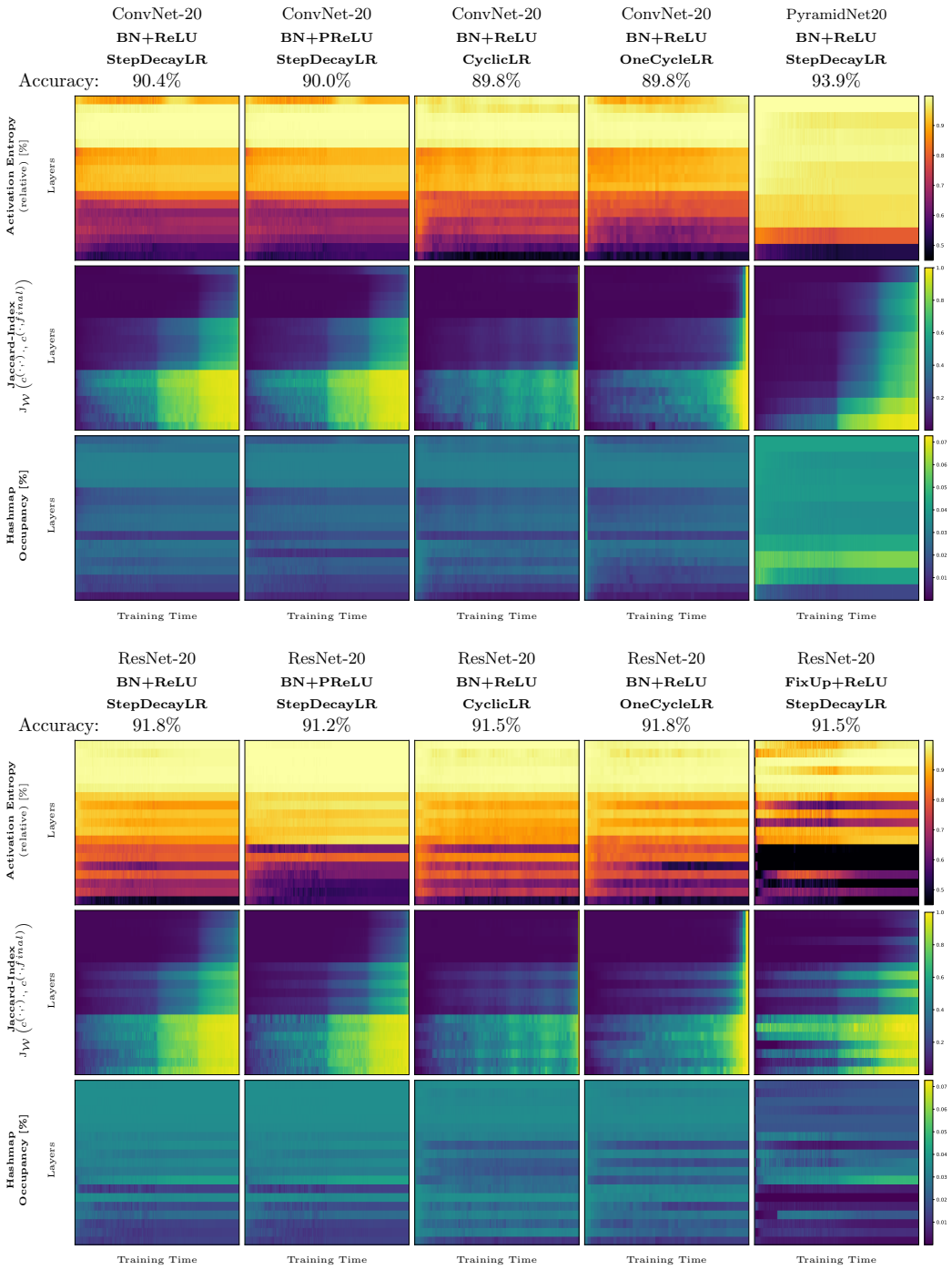


Fig. 3.7.: Complete training runs for several architectures (ConvNet-20, ResNet-20, PyramidNet-20, FixUp) and learning rate schedulers (CyclicLR, and OneCycleLR). Each plot shows a measure based on AP distributions throughout the whole training (x-axis) for all activation layers in each network (y-axis). The first row shows the APE, the second row shows the Jaccard index between the current network state during training and the final network state, and the third row shows the hash list occupancy used to compute the distributions efficiently.

From the named methods, “PReLU” dampens the drop of APE most in the middle and the third convolutional block in the case of the ResNet-20 network. The network shows a more significant reduction in entropy in the first block. The ConvNet-20 network has a similar course of APE, but shows the same characteristics additionally also in its last convolutional layer. The “PReLU” method noticeable inverts the zig-zag pattern of the ResNet-20 architecture (previously described in Section 3.1.4) in the second convolutional block. FixUp, which (in contrast to the other approaches) works without batch normalization layers, shows a distinct behavior: Except for only a few layers in the second convolutional block, the first and third convolutional blocks consistently measured the smallest value of APE compared to the other methods throughout training. In contrast, PyramidNet-20 that increases number of filters linearly as the network depth increases, has the most gradually increasing and over time most stable entropy curve: Its value rises quickly, with a low APE value at the beginning of training. Only two activation layers, those at position 6 and 9 measure a decrease of APE with progressing training. Despite their completely different behavior compared to the other methods, FixUp and PyramidNet provide a higher accuracy for the classification task on CIFAR-10 and Tiny ImageNet.

Observations for the Jaccard Similarity: Next, we describe the methods based on the Jaccard similarity to the final network state. The Jaccard similarity of the current network state to the final network state gives an intuition for the convergence of the APs: A large value indicates that a big proportion of patterns have already settled (as it measures the distance to the final state). Conversely, a low value implies that the majority of structural changes in the network are still occurring at a later stage in the training process. As expected, in all training setups all layers start with low similarity to the final network state.
For ConvNet using ReLU activation, after just a few epochs, the activations in the first convolutional block have already a similarity of about 50%. This means that about 50% of all patterns (weighted by their occurrence count) are already exactly those of the trained (“final”) network state. But, up until the first learning rate reduction at epoch 100, the second block reaches only a similarity of about 17%. The last layer does not increase significantly during that phase, starting at about 5% to 6%, the layer reaches a similarity of 6% to 7%. With an additional decrease of the learning rate at epoch 100, the similarity further increases within a few steps to about 97% in the first block, about 60% in the second block, and about 30% in the last block. In the very last few epochs, all layers converge to a similarity of 100% (as expected by construction of the measure).
The other training setups differ as follows: All ResNets show a “zig-zag” pattern, just like in the APE plots (Figure 3.3 and Figure 3.5). Compared to the entropy,

however, the high and low points in its course depends on the method used; while the similarity of the convolutions in the second convolutional block reach their maximum for the experiment denoted "ResNet-20, BN+PReLU, StepDecayLR"[4] in the layers that directly succeed a residual connection, the similarity of the same network trained with ReLU instead of PReLU, denoted "ResNet-20, BN+ReLU, StepDecayLR", is flipped across layers after or before residual connections. The training runs that use a cyclical learning rate schedule show that a "wave-like" learning rate scheduling is also reflected in the measured similarity throughout training, with a larger similarity whenever the learning rate is lower and vice-versa. This indicates that larger learning rates can lead to deviations in function space without destroying information beyond recovery. The single training runs that use the cycle learning rate scheduler, denoted by "OneCycleLR", start to converge very late compared to the other methods. However, these runs also show the most uniform convergence across all layers and converge the fastest compared to the other methods. Regarding uniformity among the tested architectures, "PyramidNet-20" is the network type with has the most uniform increase of Jaccard similarity, both layer-wise and training time-wise. The first layer of PyramidNet-20 has the most similar pattern sets throughout training compared to the final network states' pattern set despite having the lowest APE. The similarity of the subsequent layers behaves similarly but with lower similarity values, only the last layer does not converge as quickly as the other layers. From the tested architectures, the FixUp variant of ResNet is probably the most distinct in terms of the behavior of Jaccard similarity (as it is also for APE). Compared to the other methods, FixUp manages to reach some of the final APs for some layers (5, 6, 7, 10, 12, and 16) much earlier in training than the other methods. The remaining layers have approximately the same behavior as other "default" training setup denoted "ResNet-20, ReLU, StepDecayLR".

3.1.7 Discussion of the Evolution of Activation Patterns

Stochastic gradient descent (SGD) [143] does not take the decision boundaries of Rectified Linear Unit (ReLU) activations directly into account – a gradient descent step may result in crossing that boundary for a single update, or it doesn't. Which case occurs is somewhat random and depends on factors like the learning rate, the used optimizer and the exact data distribution, just to name a few. The following paragraphs will review the results described earlier, shedding some light on the questions asked at the beginning of this chapter.

[4] This denotes a ResNet-20 with batch normalization layers and PReLU activations, trained with a step decay learning rate schedule.

Methodical Considerations and General Findings

The Activation pattern entropy (APE) relates to the internal complexity of a neural network throughout training. This measure indicates qualitatively how many distinct affine transformations are required to fully describe a deep network used for inference of a given, finite dataset. If a network uses only a few distinct affine transformations (i.e. activation patterns (APs)), this corresponds to a regression task that rather blends linearly between data dimensions. On the other hand, a network that uses the maximal number of APs comes closer to applying a distinct linear mapping for every element in the dataset.
The experiments (e.g., Figure 3.7) visualize the extent of expressivity throughout training, and they illustrate that the exact behavior depends on the architectural choices used for the networks. For instance, the tested ResNet architectures and residual connections in particular, indicate an alternating scheme, using fewer patterns (meaning they are "more linear") in layers that succeed the residual connections directly and thus use more patterns in every other layer in-between. Removing architectural features (such as residual connections) and increasing filter sizes (ToyNet) results in APE valleys during the early and later training phases, in other words, sudden reductions of expressivity. Methods that reduce training speed (parametric ReLU (PReLU), OneCycleLR, CyclicLR) have countered this effect again in the shown experiments.

Initialization (and the first steps of training)

The most widely adopted method of initializing deep neural networks that use ReLU-activations today is [61], commonly referred to as He initialization. Its idea is that a very precisely scaled Gaussian distribution results in i.i.d. random activations per neuron, i.e., maximally entropic APs. In theory, the condition of independent input dimensions may be correct for specifically designed datasets. However, real-world data rarely meets the assumption of independent dimensions. For example, pixels in photographs are often correlated. But even for uncorrelated data, the assumption is even less likely to hold true for deeper layers due to the mixing of inputs throughout the network. In support of this, one can name the numerous works that have developed successful methods for training even without adhering to this assumption, for example, the exponential learning rate scheduler [102], OneCycleLR [160], and fixed-update initialization [191]. The strict increase of APE in the initialization phase (see Section 3.1.5) can be seen as a validation that the possibly unfulfilled condition of "i.i.d." pre-activations, which was used to prove He-initialization does

not impact the initialization and thus performance negatively. Previous works have given many reasons why the ResNet-Architecture is beneficial to training [10, 129]. Our measures give a new indicator why this might be the case.
In Figure 3.2, we have measured the Jaccard similarity to the initialization state over the entire training time. The two models, ConvNet-20 and ToyNet-20, lose most APs of the initialization, but regain a larger subset of patterns from the initialization again during the rest of the training process. In contrast, the ResNet architecture shows a less steep decrease, and, in some layers, the architecture manages to maintain the same level of shared APs of the initialization over extended periods of training. The analysis in this work, however, does not cover whether the APs of initialization are meaningful for the network's performance and thus, this remains an open question for future work.

The Early Phase of Training

The early phase of training has been identified in several works to be distinct from the rest of training. Goodfellow and Vinyals [52] already described that gradient magnitudes increase substantially during the first ≈ 10 steps of training. This section identified that during that phase, the APE drops in almost all layers (see Figure 3.3 and Figure 3.1). A similar reduction of expressivity and a subsequent recovery phase of APE has also been previously observed by Hanin and Rolnick [59] in terms of the *theoretical* total number of activations over the whole input domain. The experiments in this section verify that theoretical change in activations is also applicable to more realistic configurations of architectures & methods.
Frankle et al. [45] describe the first 500 steps of training as a phase that experiences a substantial "rapid motion" in weight space. The experiments in this section revealed similar characteristics (see Figure 3.1): Most APs of the initialization state are replaced the first few steps of training, indicating that further work on the initialization could improve training speeds. Additionally the pattern changes, especially those of the middle layers of the tested networks, increase significantly in the first 500 steps. At the same time the most frequent patterns increases in weight before the weight decreases again as the pattern changes begins to even out. One possible explanation is that these rapid structural changes accommodate a more efficient representation of the input data, initially relying on a few dominant features (the most frequent patterns), but diversifying the feature representations as training progresses. This aligns with the current understanding of how networks learn, which to learn low-frequency features first and gradually build up complexity afterwards [138, 184].

In the next training phase, described by Gur-Ari et al. [56] to end at about training step 700, the Hessian Eigenspectrum changes; the gradients are guided to a more confined subspace. Also, the direction of the momentum starts to align in that phase (in the example of VGG-networks). Similarly, the experiments illustrated in Figure 3.1 showed that the amount of pattern changes and the number of total patterns stabilize for most layers, implying that less structural changes occur. After that (starting at about training step 2000 in this setup), Frankle et al. [45] characterizes training to have a constant magnitude spectrum of gradients and a slow increase of weight magnitudes. The metrics defined in Section 3.1.4 reflect these observations as the magnitudes of pattern changes also converge in the entire network to a hitherto minimum value.

The pattern changes also reveal which training steps change the whole structure the most, indicating that some batches incur more substantial changes for a single optimization step. Sudden strikes of pattern changes throughout the whole network are visible in Figure 3.1, first removing many patterns in all layers at the same time and then adding patterns back in an approximately equal amount with the subsequent optimization step. Achille et al. [3] has shown that unfavorable data early in training can corrupt the whole training process resulting in irreparable damage to the network state. By diagnosing examples based on their learnability and ambiguity, Swayamdipta et al. [167] have shown that data example difficulty and training variance can vary significantly across a single dataset. One can argue that measures such as the number pattern changes for a single step of training could help to identify such counter-productive data points or batches, since they have a more significant impact on the network's structure.

Convergence Speeds of Methods

The experiment visualized in Figure 3.2 demonstrates the convergence rate of APs throughout training. The findings indicate that networks converge first in the lower layers and from there, bottom to top. More specifically, the first convolutional block converges much faster compared to the rest of the architectures. For the final layers, the similarity between the training states and the final network states reach only a Jaccard Similarity of about 30% at the end of training, meaning that the remaining patterns are still not converged, and thus will change significantly in the last few optimization steps. This hints at especially the APs of deeper layers being very noisy. The best behavior in terms of fastest convergence among the tested setups have shown PyramidNet and FixUp. PyramidNet has shown to converge more uniformly across all layers, whereas FixUp converges much faster in the early stages of training for certain layers.

Conclusion

Non-linearities are directly responsible for the high expressivity of deep networks. As a prominent example, the ReLU non-linearity uses a simple decision boundary to break the linear character of calculations. Surprisingly, although optimization using stochastic gradient descent does not take that decision boundary directly into account, the decisions still change during training due to overshooting their boundaries if gradient descent steps are large enough. This section analyzed these discrete changes in function space by reinterpreting the continuous optimization as a discrete search of emerging structures. In more detail, the goal has been to analyze where, when, and how quickly such structures arise during training. Unsurprisingly, given the magnitude of the problem, the experimental analysis does not provide an overall model of the implicit algorithm of discrete optimization. Instead, the value and main contribution of this chapter lie in its ability link common architectural features and prominent training techniques to the qualitative changes in the discrete part optimization.

The answer to the question which layers learn first in a network is much more nuanced than just bottom-up or top-down. In more detail, the experiments reveal that ResNets tend to retain certain patterns of the initialization better during training, unlike similar architectures without residual connections, which replace the pattern set of the initialization almost entirely within only few steps of training. The experiments also validate several effects observed in the literature, including the analyses of Frankle et al. [45], Leclerc and Madry [91], Hanin and Rolnick [59], and Achille et al. [3]. Further experiments showed that the network's expressivity, or APE, follows an architecture- and training method-specific curve during training, often dropping after only the first few training steps. This behavior has been tested against several methods and features; PReLU, ResNet residual connections, learning rate schedulers, and ResNet variants (PyramidNet and FixUp), of which some particularly countered that effect by maintaining higher expressivity throughout training and boosting the convergence speeds of non-linear structures in the networks.

The findings of this section on the non-linear convergence, expressivity, and initialization of neural network structures could lead to the development of more efficient training methods, architectures, or monitoring tools, potentially providing new avenues towards for understanding the optimization of neural networks in general.

3.2 Activation Pattern Temperature

This section is based on the manuscript:
ActCooLR – High-Level Learning Rate Schedules using Activation Pattern Temperature
David Hartmann, Sebastian Brodehl, Michael Wand
`https://openreview.net/forum?id=yqj6q_eNTJd`

Despite the huge success of deep networks, their training behavior is still under-explored. While the last section has discussed the literature on training phases, a holistic understanding to explain the strong performance of stochastic gradient descient (SGD) is still lacking. As the non-linearities are not used during the computation of the gradients used for optimization, how does SGD manages to find good solutions that generalize to unseen data at all?
To get one step closer to an answer, this section will take a look at what a single optimization step does to the activation patterns (APs) of a network by combining the notions of APs and learning rates used for training.

As stated by Bengio [14], the *initial* learning rate of SGD is "[t]he single most important hyperparameter, and one should always make sure that has been tuned". It is considered to steer the amount of noise that regularizes the optimization [19, 75]. Research spans from practical recommendations, such as best practice learning rate schedules of distinct forms [14] to theoretical models that unveil the implicit regularization of SGD that depends on the learning rate [11, 161]. For instance, many training procedures include a *warm-up* phase into the learning rate schedules to adapt training to numerical limitations as well as the distinct behavior of the initial training phase compared to the rest of training [53, 54, 109].
As discussed previously (in Section 3.1.1), more recent studies divide the whole training process into phases of distinct characteristics - but even the actual number of regimes or phases is still discussed, most commonly described as two or three phases ([45, 91, 96, 101, 125]).

This section aims to combine the notions of AP and learning rates. As a central tool, this section proposes a new layer-wise measure of the learning progress, *Activation Pattern Temperature (APT)*. The key idea is to focus on the "implicit" part of optimizing deep networks, which is the fitting of their non-linear compositions – the APs. Therefore, the measure considers network changes not based on the step-size in the parameter space, but rather by examining the APs to get insights into changes in the discrete function space. The idea is to count the non-linear changes of *APs* to assess the amount by which the non-linear part of network, specifically the layer-wise

decomposition of feature maps into piecewise linear regions, changed.
Effectively, this idea describes a learning rate in the non-linear function space of APs of any network. Unlike the original learning rate and other simple differential measures in parameter space, the measure is independent of linear reparametrizations of the model and provides no additional hyperparameters.

Through the lens of this measure, the training of Rectified Linear Unit (ReLU)-networks using the commonly used *OneCycleLR* scheduler [160] reveals a simple behavior, namely an approximately linear decrease of APT during training (see Section 3.2.2). By observing the relation of the APT and the learning rate used at any point in training Section 3.2.3 discuss a two-parameter model that accurately predicts the number of AP changes for the next optimization step at any point in training.

3.2.1 Related Work on Complexity Measures

Driven by the goal to better understand generalization and training, complexity measures have been suggested in the literature.
Even though these often consider static network states (and in contrast the herein defined measure takes a dynamic training step into account), the notion of both ideas is similar: to get a better understanding of the random fluctuations in the output of a network.

There are several studies on the correlation between complexity- and norm-based measures – a wide variety of work introduces sharpness-based measures that give mathematical characterizations of the loss landscape, promising a deeper understanding of the training phases, stability of training, and generalization of deep networks (see [76] for a general discussion). In particular, whether generalization improvements come from the flatness of the loss landscape has been discussed both affirmatively and negatively [39, 161]. Nevertheless, sharpness- or curvature-based methods have been utilized to improve generalization in practice [145, 43].

Numerous works have included additional regularization into the training process; For instance, the angle between the momentum vector and the local gradient has been used to construct a statistical test to determine convergence [89]. Value and statistics of the loss itself have also been used for regularization during training, either by relaxing the softmax loss [123] or by adapting the gradients in order to make constant progress on the loss [144]. Lastly, Raghu et al. [136] proposed a method to measure the layer-wise complexity of a network by computing the

Singular Value Decomposition of the activations. This method is similar in spirit to the idea analyzed in this section, but it measures the intrinsic dimensions in a stationary fashion, excluding the training process in the measure itself.

Most related to this section in terms of interpreting network activations to probe training behavior are [136, 53]. In these works, the authors propose a new technique for comparing network representations using Singular Value Decomposition and Canonical Correlation Analysis. Other works focusing on the analysis of the weights of a network are inherently limited, since the weights exhibit several invariances, such as permutation and scaling, as Gotmare et al. [53] note.
In contrast to the named studies, this section focuses on the discrete APs rather than the continuous view of values, weights, gradients, and the similar. In contrast to other measures, the measure defined in this section is hyperparameter-free and is by construction invariant from linear network re-parametrizations.
It avoids the complexity of rescaling due to surrounding layers, which plague many continuous measures, by solely focusing on the discrete activations of a ReLU network.

3.2.2 Activation Pattern Temperature

The learning rate is the step-size used during optimization of a network that controls the amount by which the computed gradient is applied to the weights on every iteration. Depending on the norm of the actual weights, the same learning rate can result in different "effective" step sizes, which is why the term *effective learning rate* was coined in the literature [147].

This section makes a similar observation for the step size in function space: The approach is based on the notion that the non-linearity is what enables the expressivity of deep networks. The non-linearity of whole networks is thus encoded in how data affects the switch between those discrete states in an orchestrated way.
Our core idea is to track the change of an AP by comparing corresponding outputs of ReLU layers for the same input data before and after an optimization step. The neg-log-likelihood of these changes, defined below as the *Activation Pattern Temperature (APT)*, quantifies a non-linear equivalent of a "step-size" in the training progress.

Definition of the APT

As in Definition 1 the ReLU-layers are inspected independently. Formally[5], the study applies to any network F_θ with parameters θ with at least one ReLU-layer σ_l, for $l = 0, \ldots, L$. First, the evaluation $F_\theta(x)$ on a data point x is decomposed into

$$F_\theta(x) = G_\theta\left(\sigma_l\left(F_{l,\theta}(x)\right), x\right), \tag{3.11}$$

where $F_{l,\theta}(x)$ and $G_\theta(x)$ are the part before and after the ReLU-layer to be inspected. This decomposition first maps data using a black-box function $F_{l,\theta}$ to their pre-activations of dimension d before the ReLU-layer to be inspected, $F_{l,\theta}(x) \in \mathbb{R}^d$. Then the network applies the ReLU-activation σ, and proceeds the evaluation using the remaining (black-box) part G_θ of the network. Note that (as before) G_θ enables arbitrary architectural choices such as skip connections, by employing x as G's second argument.

To recite Definition 1 in the context of optimization, and to add the parameters θ to the notation, the *APs* for the ReLU layer σ_l and parameters θ are denoted in the following as

$$A_{\theta,l}(x) := \left(\operatorname{sign} \circ \sigma_l \circ F_{l,\theta}\right)(x) \in \{0,1\}^d. \tag{3.12}$$

Modelling the Activation Changes during Training: Training is performed in discrete steps $t = 0, 1, 2, \ldots$ At each step, input data is sampled from the dataset, then these are used by the optimizer to compute a new set of parameters θ_{t+1} (possibly using multiple samples in form of a data batch implicitly). The first step, θ_0 is determined by the network's initialization, and all subsequent steps are computed by (re-)applying the rule called an optimization step

$$\theta_{t+1} \leftarrow \operatorname{Opt}\left(\theta_t, \eta_t\right), \tag{3.13}$$

under loss Q. The optimization step adapts the parameters θ under a step size η_t, also called *learning rate*. For instance, in the case of SGD without momentum, the optimization step is more specifically given by

$$\operatorname{Opt}(\theta_t, \eta_t) = \operatorname{SGD}(\theta_t, \eta_t) := \theta_t + \eta_t \nabla_{\theta_t} Q(F_{\theta_t}, X), \tag{3.14}$$

where $\nabla_{\theta_t} Q$ denotes the gradients, which are through the loss Q informed of the ground-truth information that is used for a training objective, for instance image classification.

[5] Note that notations are adapted slightly in this section to also including the current state θ

For all notations and equations that follow, we neither need the loss Q, nor the ground truth labels, and thus we will omit these as well as the specific implementation of Opt. By doing so we implicitly assume Opt to contain the forward and backward passes of the network F_θ as well as the loss computations.

Definition 5. *Activation Pattern Distance*
To simplify notations further, we assume in this section that data points are random variables $x \sim \mathrm{Data}$ sampled from a dataset. This enables to compute the random variable D that encapsulates an AP change given a fixed learning rate η as follows

$$D_\theta(\eta) := \left(A_\theta(X) \neq A_{\mathrm{Opt}(\theta,\eta)}(X)\right), \tag{3.15}$$

or simply $D(\eta)$ in case the parameter set θ is clear from context.

Definition 6. *Activation Pattern Temperature*
The Activation Pattern Temperature (APT) *is defined as the log-probability of the event that no AP has changed for a fixed learning rate η,*

$$T(\eta) := -\log\left(1 - P\left(D(\eta)\right)\right), \tag{3.16}$$

where $P(D(\eta))$ denotes the probability distribution of the binary event that an AP has not changed.

Abstractly speaking, Equation (3.16) describes the speed at which information flows from the input X through the non-linear part of a network.

This measure is directly linked to the learning rate and is otherwise parameter-free. It specifies the amount of non-linear change in a network layer. Its lower bound is 0, stating that no non-linear change occurred and only the linear parts of the network could have been changed during that step. The temperature approaches infinity if all activations have changed during a single optimization step. The rationale of using the logarithmic version of the probability is to reflect the learning rate needed to have all APs changed. Considering that the activation cells (ACs) become larger the farther away they are from the origin (see Section 2.2), and considering that normal distributed data points can also occur in such distant cells, for them to "change" from one cell to another one requires a larger change of weights for that event to occur. In essence, for infinite many data points the learning rate has to approach infinity to make sure that all APs are changed. This will be discussed in more detail in Section 3.2.3. In more detail, Section 3.2.3 will derive an analytical representation of Equation (3.16) and show empirically how well that model fits practice.

Implementation: To measure $T(\eta)$ at a given model state θ_t, two forward-passes of a network have to be computed. The first one is to record the APs of the current training step. After that, the network is adapted according to the optimizer and choice of learning rate η. Finally, a second forwand-pass for *the same input data* is needed to check if there has been a sign change for any recorded AP. While the ACs could differ in size (and they do regarding the ACs outside of the expected domain, see Section 2.2), small batch sizes for the measurement give already good estimates of the probability $P(D(\eta))$, enabling this measurement for practical applications. To reduce costs further, this measure is only evaluated when necessary during training. For instance, Section 3.3 will use the measure in practice to propose a new learning rate scheduler adapting weights based on the discrete sensitivity of the network state.
An additional note regarding convolutional layers: the convolutions are handled as "re-using" the same layer for multiple inputs per image. Effectively this improves statistical power as each pixel of the convolution output acts as a separate example, yielding more accurate estimates for the APT.

Some of the properties of $P(D)$ and T can be explained analytically.

Observation 1. *First, by definition, for parameters $\theta \in \mathbb{R}^{d_w}$ and a fixed network F_θ and fixed data distribution $P(x)$, we have:*

- $P(D_\theta(\eta)) \geq 0$
- $P(D_\theta(0)) = 0$
- $P(D_\theta(\eta)) = P(D_{\mathrm{Opt}(\theta,\eta)}(-\eta))$

This also holds for the empirical version of the measure, denoted $\hat{D}$, and implies that the temperature $T(\eta) \geq 0$, the measured temperature $\hat{T}(\eta) \geq 0$ and that $T(0) = \hat{T}(0) = 0$. If the learning rate is zero and the optimizer does not change any parameters, no non-linear change can occur as well.

First Observations using APT during Training

As it counts the number of AP changes in a network for a given learning rate, the measure presents itself as a non-linear variant of an optimization step size. Thus, before going into the details of what tools the APT enables, a first step is to evaluate it on some examples.

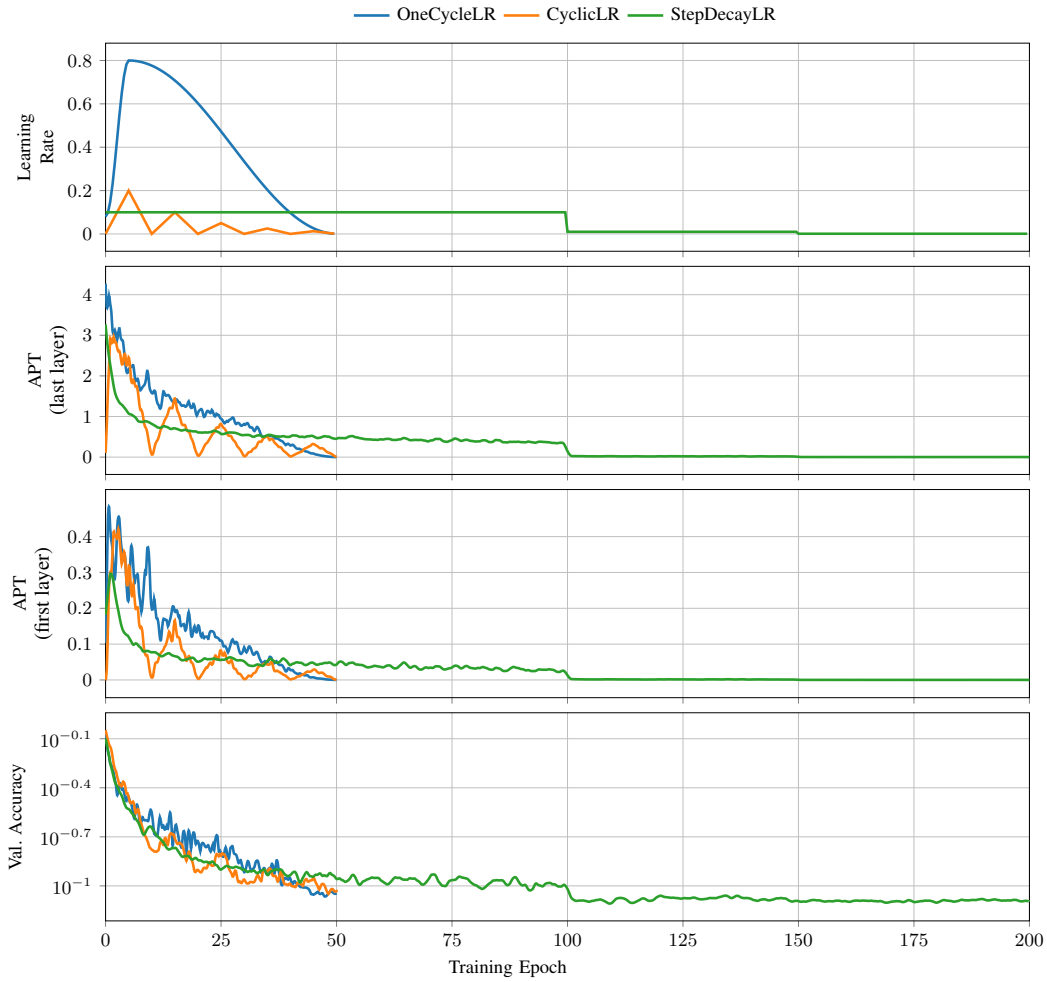


Fig. 3.8.: Learning rates, validation accuracy & temperatures of the first and last ReLU-layer over the course of training of ResNet-32 on the CIFAR-10 dataset. The used learning rate schedules are: StepDecayLR, OneCycleLR and CyclicLR.

A baseline experiment to consider is the training of ResNet-32 (CIFAR-Variant, [62]) trained on CIFAR-10, using three different learning rate schedules: *StepDecayLR*, *OneCycleLR* and *CyclicLR*.[6] Figure 3.8 shows the course of the learning rate (first row), the corresponding activation temperature of the last ReLU-layer (second row), the activation temperature of the first ReLU-layer (third row), and the top-1 validation accuracy (fourth row) for each learning rate schedule used.

Over the course of training (x-axis), APT reflects changes of the learning rate n the shown experiments (APT declines, whenever learning rate declines). Other observations are that the APT starts strictly greater than zero and the actual value

[6]Please note that the variants used in this section are essentially similar to those outlined in Section 3.1.3, with the sole difference that they use more layers per convolutional layer block. The choice has been made to emphasize the effects seen in the experiments and accompanying figures.

depends on the architecture, the layer, as well as the training hyperparameters used. Towards the end of the training process, the APT decreases and approaches zero. Most importantly, OneCycleLR (blue curves), shows approximately a linear decrease of APT.

Notably at the start of training with step decay learning rate scheduling, the temperature decreases despite a constant learning rate. This means that for a fixed setting, the course of APT does not solely depend on the magnitude of the learning rate. Consequently the APT thus contains hidden variables responsible for its course.
This is in comply with previous work that studied the early phase of training in more detail (see Section 3.1.1). This observation could also shed some light on the initial phase of training and possibly explain the requirement of using a warm-up schedule in common training schemes beyond numerical instabilities.

Per-Layer Temperature

The APT is a per-layer measure, thus similar to Section 3.1.6, the value of this measure over the course of the whole training can be best shown using a two-dimensional heatmap plot, where the x-axis represents the training time, the y-value the respective layer of the model and the color the value of the APT. For instance, Figure 3.9 illustrates the APT of ResNet-32 and ConvNet-32 both being trained using the two learning rate schedules OneCycleLR and CyclicLR each. As in Figure 3.1, the APT decreases monotonically (ignoring the noise) on all layers for ResNet-32. This is not the case for ConvNet-32, where after a sudden drop in value after the first few training steps, the APT peaks again at about epoch 4. However, this effect disappears for the second scheduler tested, CyclicLR, for both networks.
In each network of this experimental setup, the APT decreases (or "cools down") from bottom to top, in accordance to related work ([136], see also Section 3.1.1). This cool-down can be observed most visibly towards the end of training.
Also, architectural choices such as the number of filters affect the resulting temperatures. For instance, the number of filters used are clearly distinguishable in the plot as the change in color occurs with the start of a new convolutional block, differing only in the number of filters from the previous convolutional block. During experimenting, striding did not increase the temperature noticeably.
However, the variance of the noise in the measurements did change – this is probably a result of fewer applications of convolutional filters per layer due to striding.
Other architectural choices affect the temperature profile as well: activations succeeding a residual connection have a slightly higher temperature, reminiscent of

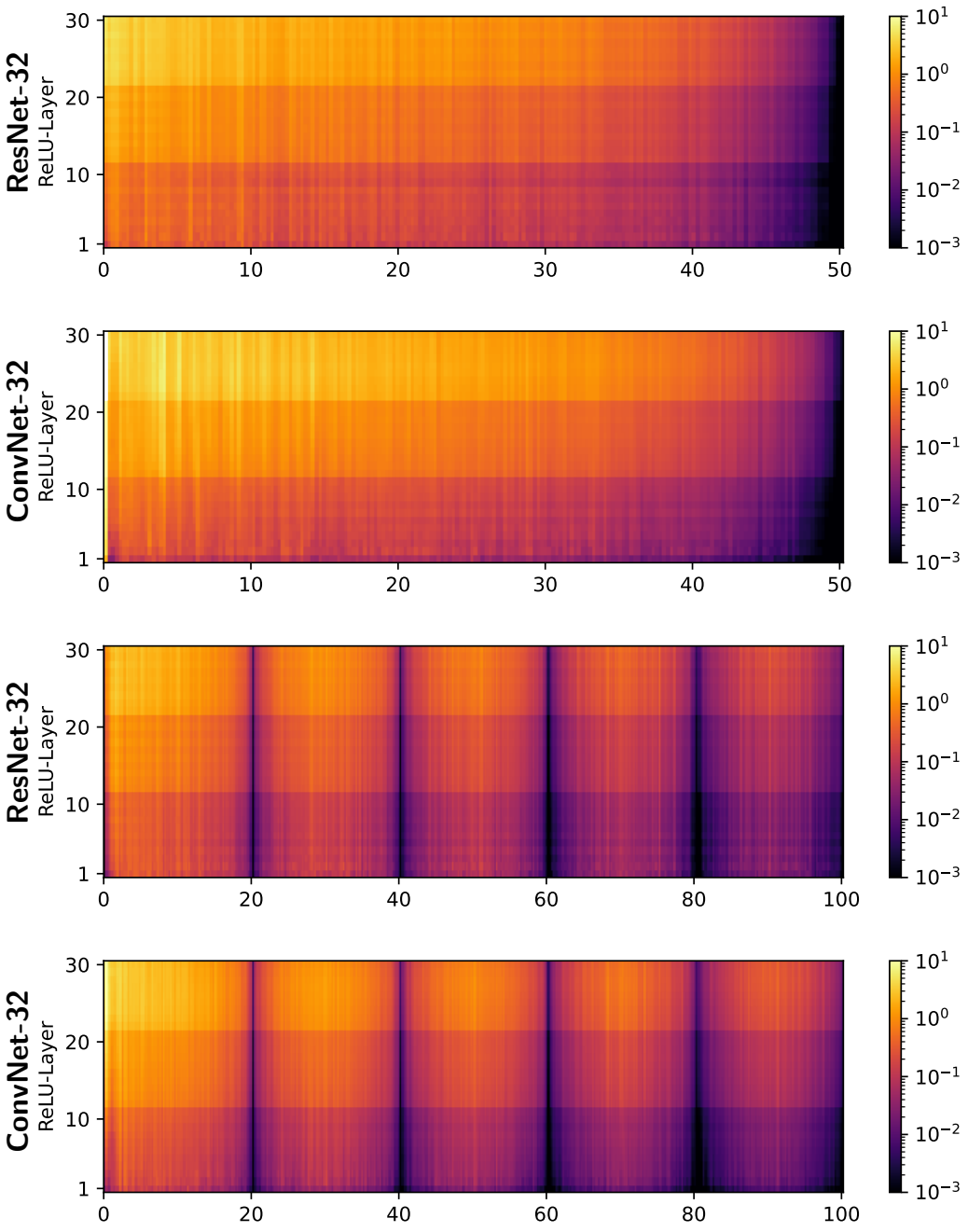


Fig. 3.9.: Per-layer APT of AP change for two network models: ResNet-32 and ConvNet-32 trained using two learning rate schemes each (see Section 3.1.3): OneCycleLR (top two rows), CyclicLR (bottom two rows).

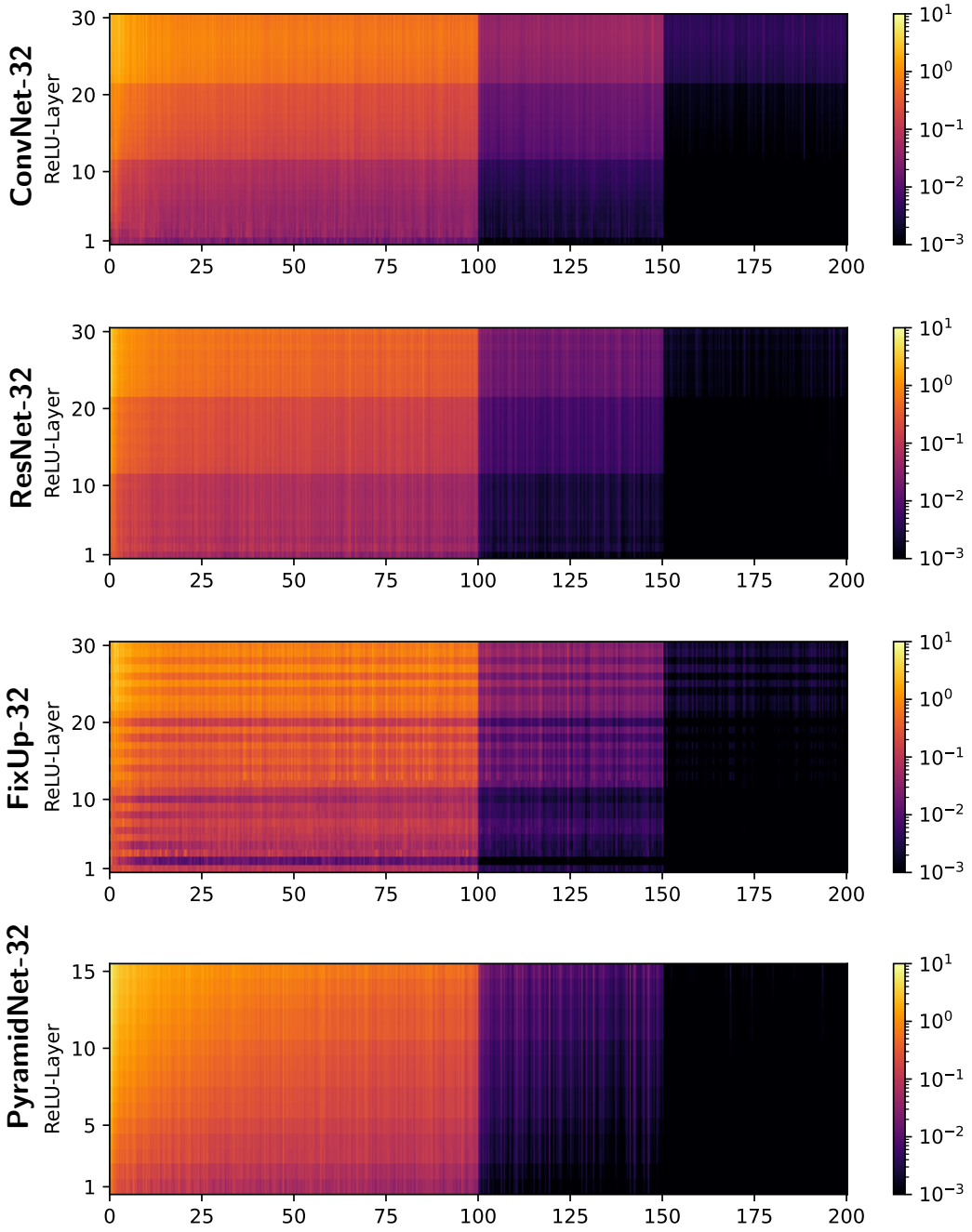


Fig. 3.10.: Per-layer APT of AP change for four ResNet variants (defined in Section 3.1.3), from top to bottom: ConvNet-32, ResNet-32, FixUp-32, PyramidNet-32. Training is done using step decay learning rate scheduling with a decay factor of 10 in Epochs 100 and 150. While the learning rate is constant, the APT decreases slowly, most visibly for PyramidNet-32.

studies that showed the relation of residual connections and the smoothness of loss landscapes [98].

Most importantly, the chosen learning rate for optimization does not yield similar APTs throughout the networks.
As network depth increases, the APT increases as well and as mentioned before the APT makes discrete jumps between the convolutional blocks (i.e. where the filter sizes change). These jumps are dependent on the network architecture used. For instance, Section 3.2.2 illustrates the layer-wise APT measurement applied four architecture variants trained with step decay learning rate scheduling: ConvNet-32 (without residual connections), ResNet-32 (with residual connections), PyramidNet-32 (linear increasing filter size) and FixUp-32 (fixed update initialization).
In contrast to ConvNet-32, the other variants exhibit a more consistent decreasing APT profile as the learning rate decreases. Viewed layer-wise there are also noticeable differences in the architectures – for instance, PyramidNet increases the APT more uniformly from lower to higher layers.

3.2.3 Learning Rate Sensitivity Analysis

While the APT, which is the proportion of AP changes, is influenced by the learning rate used for a gradient descent step by definition, it is not clear yet how these two concepts are connected in detail. This section aims to address this connection by deriving a two-variable model that demonstrates predictive power: given any learning rate and the two scalar variables representing the current state of a layer during training, the model estimates the probability that an arbitrary AP changes in the next optimization step.

To derive the model, this section begins by investing experimentally the sensitivity of the non-linear part of the model at any point in training is. This section thus suggests the following analysis method denoted the *Learning Rate Sensitivity Analysis*.

Analysis Method: The goal is to study the APT from a learning rate-invariant viewpoint while a network is being trained. This is accomplished by inspecting the APT for an wide range of learning rates at the same training step.
Before training, the predefined range of learning rates is uniformly sampled on a logarithmic scale and saved. A setting proven effective in practice is a range of 10^{-4} to 10^{2}, providing a total of consisting 50 sampled learning rates in total. Then, for the initialization and each epoch during training, the network is fixed and re-evaluated to assess the APT across the list of learning rates. Finally, this APT

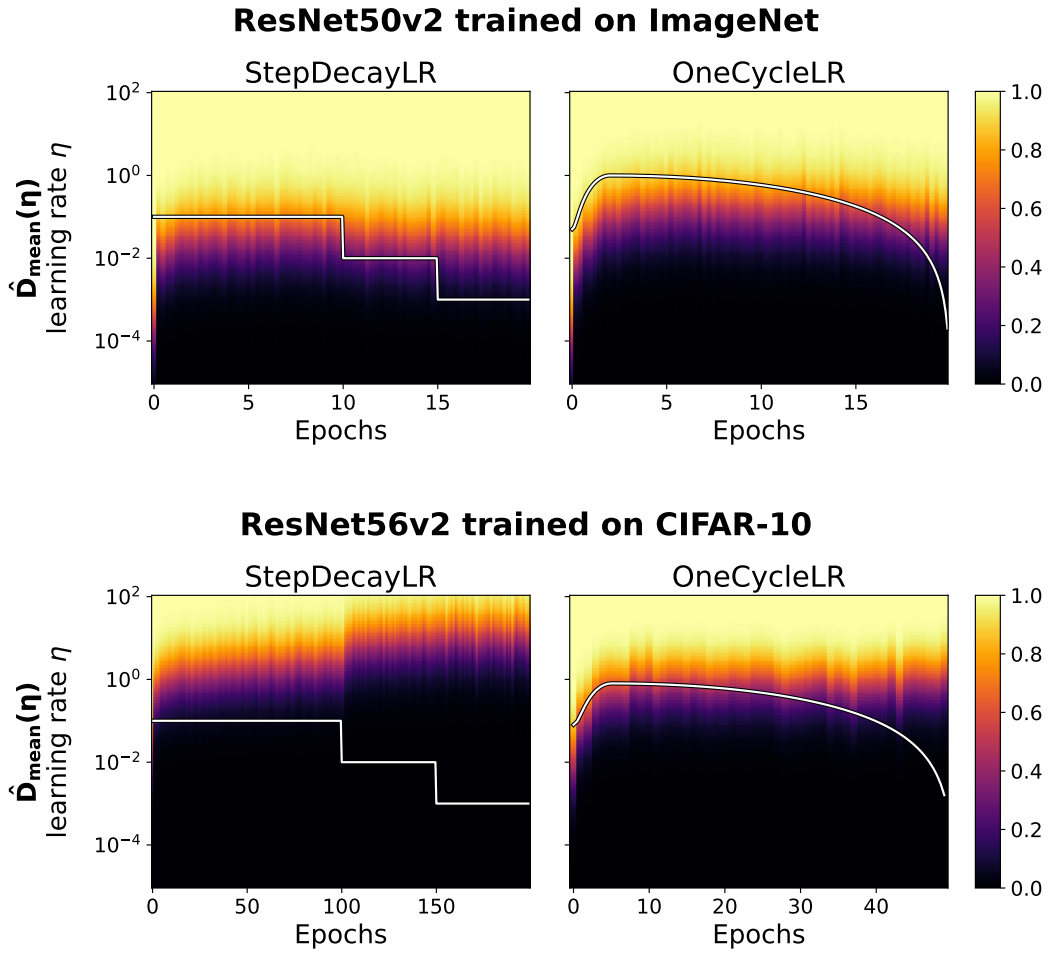


Fig. 3.11.: Learning Rate Sensitivity Analysis: For each point in training time of ResNet-56 on CIFAR-10 and ResNet-50 on ImageNet both trained with OneCycleLR and CyclicLR, the probability of change of an AP is measured for a theoretical choice of a learning rate. The actual learning rate used for training is visualized at any point in training by the white line.

is visualized as a heatmap plot with the training time on the x-axis and the used learning rate for the measurement on the y-axis, the color corresponds to the value of the measurement.

It's worth noting that the previous section also inspected heatmap plots, however, they differed as follows: In contrast to the previous section, where the y-axes in the heatmap plots depicted the network layers used for the measurement, this section uses the y-axis to specify the learning rate used to measure the APT. Thus, as the APT is a layer-wise measure, the subsequent experiments will combine the different layer-wise measures by taking the average over all layers, $\hat{D}_{\text{mean}}(\eta)$ unless specified otherwise.

First Observations: The following experiment, as illustrated in Figure 3.11, demonstrates the method for two different architectures on different datasets and two different learning rate schedules each: ResNet-50 on ImageNet, and ResNet-56 on CIFAR-10. Both networks are trained using the step decay learning rate schedule (a total of 90 epochs and 200 epochs respectively) and One Cycle learning rate schedule (a total of 20 epochs and 50 epochs respectively) each.
The top rows of each heatmap plot show the probability of AP change $D(\eta)$ for each network and for each learning rate schedule averaged over all layers of the respective networks, ranging from black (row APT) to yellow (high APT) The white lines indicate the learning rate, which actually has been used for training.
The results shown in Figure 3.11 show that after a short initialization phase of few initial epochs, only minor changes occur in the probability profile. Furthermore, the probability profiles of ResNet-50 (ImageNet) show a phase in which, from a learning rate-independent perspective, the networks "cool" down slowly, independently of the choice of learning rate the probabilities decreases. It is worth noting that APT curves such as those presented in Figure 3.8 represent logarithmically scaled cross-sections through the profiles shown in Figure 3.11 at the actually used learning rate curve indicated by the white lines.
Most notably, the One Cycle learning rate schedule uses learning rates in a higher probability regime (near the vertex) for a longer period of time during training. In contrast, step decay learning rate schedule transitions into low-probability regime more quickly as seen for "ResNet-50 on ImageNet", or even start lower, as observed for "ResNet-56 on CIFAR-10". The following section aims to model this variation in probability profiles.

Gaussian Transition Model

Typically, weight initialization is based on the work of He et al. [61], which specifies the initial distribution of the weights in such a way that the output variance of interim results *after* the ReLU-activation remain constant for any network depth. This distribution changes after only few steps in training, thus techniques like batch normalization are used to mitigate drift by normalizing layer results online during training. Still, as shown below the probability of a change of APs can be modeled as the cumulative distribution function of a normal distribution as a first approximation for the whole training process. The phase diagrams in Figure 3.11 show a smooth transition from $D \approx 0$ to $D \approx 1$ over a narrow transition boundary. Thus, this section discusses the postulated hypothesis to model this transition as a cumulative Gaussian (erf) transition with parameters μ and σ for mean and standard deviation.

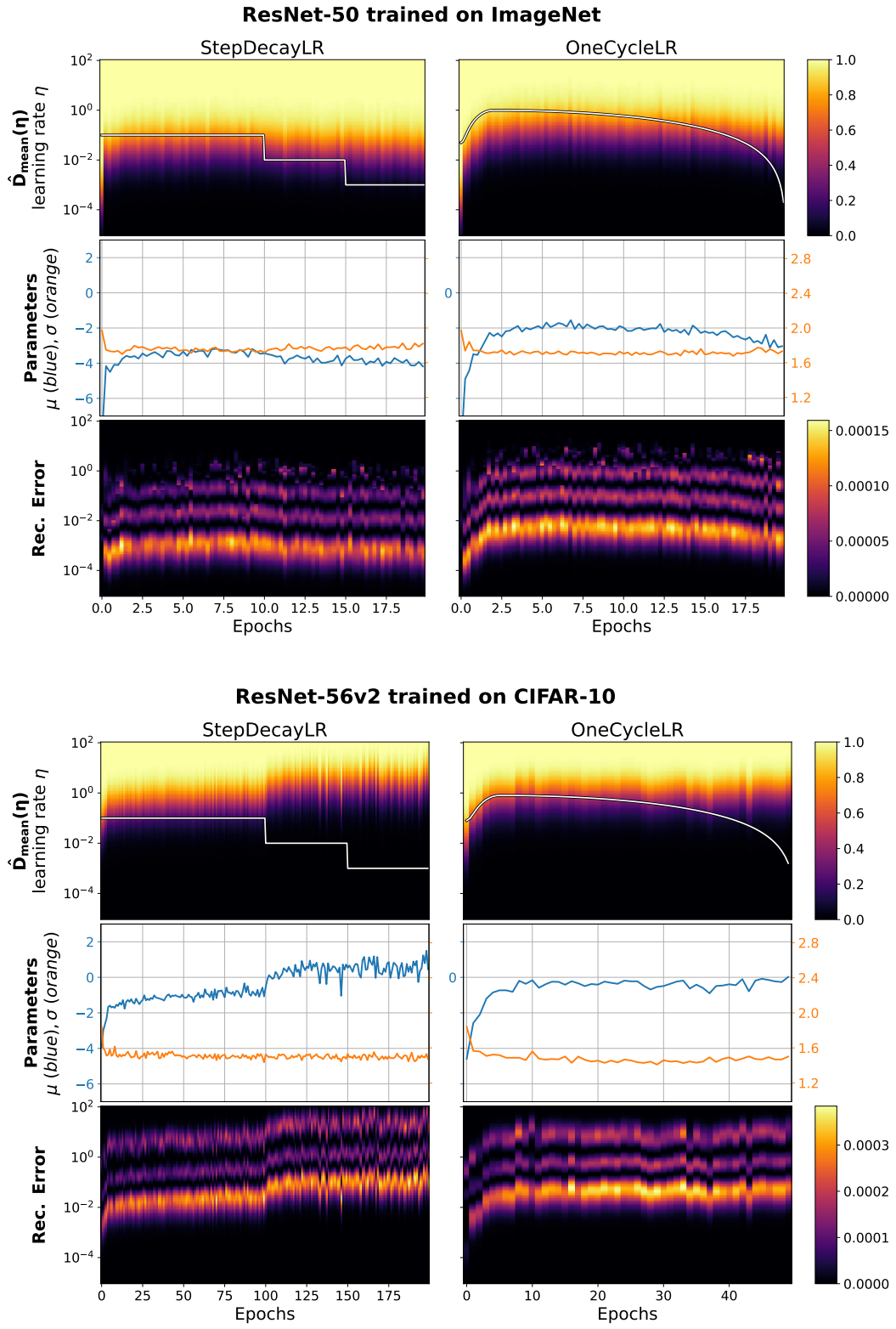


Fig. 3.12.: Accompanying Figure 3.11, the learning rate sensitivity analysis plot (first row) shows also the non-linear least squares fits of the parameters μ and σ for the alternative model of $D(\eta)$ given in Equation (3.17) (second row) and the reconstruction error of the fit (third row).

Hypothesis 2. *The probability that an activation pattern changes after a single update step with learning rate η is given by*

$$\Pr\left(A_{\theta_t}(X) \neq A_{\mathrm{Opt}(\theta_t)}(X)\right) = \frac{1}{2} \cdot \left(1 + \mathrm{erf}\left(\frac{\log \eta - \mu}{\sigma \cdot \sqrt{2}}\right)\right), \tag{3.17}$$

where the data is represented by the random variable X.

Note that the parameters $\mu, \sigma \in \mathbb{R}$ depend on the training process, the choice of architecture, and the layer.

The following section will look at the mathematical derivations that underpin the coherence and validity of this and a second model. But first, we employ the model to reconstruct the results shown in Figure 3.11, showing that Equation (3.17) actually fits in practice. In more detail, for a fixed network state (at training time t), a non-linear least squares optimization fits the model parameters to the data shown in Figure 3.11. Figure 3.12 shows the resulting curves for μ and σ for the same experiment that has been shown in Figure 3.11. The quadratic reconstruction-error of the model is shown in the third row against ground truth in the first row of Figure 3.12.

The most important observation is that favorably only small drifts of the parameter σ occur that defines the width of the Gaussian distribution used in the model. In contrast, the course of μ seems to be especially affected by the learning rate used for training. Section 3.2.4 will go into this observation in more detail and compare this to different architectures and training parameters – the model values depend on the layer the temperature has been measured in, the dataset and architecture used and varies also with the random seed of the initialization of weights. Section 3.3.2 will use this observation to define a learning rate schedule based on this notion to steer the probability of AP changes instead.

Can we Do better than erf?

The transition via an error-function is an ad-hoc approximation that empirically works well for the overall training process and averages over several layers. However, the phase diagrams in Figure 3.11 could be represented by any S-curved smooth transition function from $D \approx 0$ to $D \approx 1$ instead. Nonetheless, at least at initialization, a more precise characterization for a single layer with large enough width can be derived mathematically.

Theorem 1. *Assume a ReLU network f is at initialization, with weights and biases initialized i.i.d. from Gaussian normal distributions with mean zero and variances $\sigma_w, \sigma_b \in \mathcal{O}(1/d_{l-1})$, respectively. Further, assume that the gradients of the parameters also follow a normal distribution with mean zero and $\sigma_{\Delta w}, \sigma_{\Delta b}$, and the inputs $x = (x_1, .., x_{d_{l-1}})$ are d_{l-1} independent random variables with finite mean and variance, and independent of the weights and their gradients. Further, assume that the gradients do not vanish for any data point x and all ACs have approximately the same size. For layer l in the limit of width networks with large enough d_l, d_{l-1} the probability of AP change then approaches:*

$$D_l(\eta) = 1 - \left[1 - \frac{1}{\pi} \arctan\left(\frac{\sigma_g}{\sigma_c}\eta\right)\right]^{d_l}. \tag{3.18}$$

For $\sigma_b = \sigma_{\Delta b} = 0$ (i.e., no biases used), this is also the probability over the data distribution.

Proof. Consider a layer l at initialization. Let the entries of the weight matrix $W^{(l)}$ be denoted by $w_{i,j}^{(l)}$, $i = 1 \dots d_l$, $j = 1 \dots d_{l-1}$ and the entries of the bias vector $b^{(l)}$ by $b_i^{(l)}$, $i = 1 \dots d_l$. Pre-activations y are then functions of their input x from the layer below:
Let $y_i : \mathbb{R}^{d_{l-1}} \to \mathbb{R}$ be the pre-activation of the i-th hidden neuron in layer l:

$$y_i^{(l)}(x) = \sum_{j=1}^{d_{l-1}} w_{ij} x_j + b_i.$$

The activation of hidden neuron i in layer l, $a_i^{(l)}$, is given by

$$a_i^{(l)} = \text{sign}\left(y_i^{(l)}\right).$$

Before the gradient step, the activation of neuron i switches at a random hyperplane $x \mapsto w_i^T x + b_i$ with normal vector $w_i := (w_{i_1}, \dots, w_{i_{d_{l-1}}})$. With ΔW as (generalized[7]) gradient vector on all weights, let Δw_i denote the entries of ΔW corresponding to the weights in the normal vector w_i, and Δb_i the gradient of the i-th bias. The resulting plane equation then becomes

$$\begin{aligned} y_i^{(l)}(x; W - \eta\Delta W) &= (w_i - \eta\Delta w_i)^T x + b_i - \eta\Delta b_i \\ &= w_i^T x + b_i - \eta((\Delta w_i)^T x + \Delta b_i). \end{aligned}$$

[7] ΔW might be result of any optimizer, including aspects like momentum; for this reason it is denoted by ΔW and not by ∇W.

The transition points at which an activation $a_i^{(l)}$ changes (with varying η) satisfy the condition:

$$\underbrace{w_i^T x + b_i}_{=: c_i,\ \text{initial value}} = \eta \underbrace{\left[((\Delta w_i)^T x + \Delta b_i)\right]}_{=: g_i,\ \text{gradient}}.$$

This can be simplified to $c_i = \eta \cdot g_i$, where c_i and g_i are (in the limit, for large d_{l-1}) normally distributed random variables with zero mean (because both are linear combinations of independent, such normally-distributed random variables) and standard deviations denoted by σ_c and σ_g.

Under the assumption that the parameters and the gradients are i.i.d. normally distributed, and that the inputs are independent (also from the other parameters) with finite mean and variance σ_{x_i}, we can compute the standard deviations as:

$$\begin{aligned}
\text{Var}[w_{ij}x_j] &= \mathbb{E}[w_{ij}^2 \cdot x_j^2] - \mathbb{E}[w_{ij}x_j]^2 = \sigma_w^2 \sigma_{x_i}^2, \text{ and thus} && (3.19)\\
\sigma_c^2 &= \sigma_w^2 \sum_{i=1}^{n} \sigma_{x_i}^2 + \sigma_b^2, \text{ and} && (3.20)\\
\sigma_g^2 &= \sigma_{\Delta w}^2 \sum_{i=1}^{n} \sigma_{x_i}^2 + \sigma_{\Delta b}^2. && (3.21)
\end{aligned}$$

The likelihood of activation $a_i^{(l)}$ changing now corresponds to finding a value $\eta > 0$ for which[8] $\eta > \frac{c_i}{g_i}$.
First, we model the effect of w_i being a random variable, and fix g_i (i.e., fix $\Delta w_i, \Delta b_i$):

$$\begin{aligned}
\Pr\left(a_i^{(l)} \text{ unchanged } \mid g_i\right) &= \Pr\left(\eta < \frac{c_i}{g_i} \text{ and } c_i \cdot g_i > 0\right) && (3.22)\\
&= 1 - \int_0^{\eta} \mathcal{N}_{\sigma_c,0}(|g_i| \cdot \eta)\ \mathrm{d}\eta && (3.23)\\
&= 1 - \left(\Phi\left(\frac{|g_i| \cdot \eta}{\sigma_c}\right) - \Phi\left(0\right)\right) && (3.24)\\
&= 1.5 - \Phi\left(\frac{|g_i| \cdot \eta}{\sigma_c}\right) && (3.25)\\
\left[&= 1 - \frac{1}{2}\,\text{erf}\left(\frac{|g_i| \cdot \eta}{\sigma_c\sqrt{2}}\right)\right], && (3.26)
\end{aligned}$$

[8] Division by g_i: We assume that the gradient does not vanish, thus $g_i \neq 0$. This also ensures a properly defined distribution in Section 3.2.3.

using the properties of the *error function* erf and the *normal cumulative distribution function* Φ:

$$\begin{aligned} \operatorname{erf}(x) &:= \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} \mathrm{d}t \text{ and} && (3.27) \\ \Phi(x) := \int_{-\infty}^{x} \mathcal{N}_{0,1}(t)dt &= \frac{1}{2}\left(1 + \operatorname{erf}\left(\frac{x}{\sqrt{2}}\right)\right), \text{ i.e.,} && (3.28) \\ \int_0^x \mathcal{N}_{0,1}(t)dt &= \frac{1}{2}\operatorname{erf}\left(\frac{x}{\sqrt{2}}\right). && (3.29) \end{aligned}$$

Taking the random nature of g_i into account, we obtain:

$$\begin{aligned} \Pr\left(a_i^{(l)}(x) \text{ unchanged }\right) &= \int_{\mathbb{R}} \Pr\left(a_i^{(l)}(x) \text{ unchanged } \middle|\, g\right) \Pr(g)dg && (3.30) \\ &= \int_{\mathbb{R}} \mathcal{N}_{0,\sigma_g}(g)\left[1.5 - \Phi\left(\frac{|g|\cdot\eta}{\sigma_c}\right)\right] dg && (3.31) \\ &= \int_{\mathbb{R}} \frac{1}{\sigma_g}\mathcal{N}_{0,1}\left(\frac{g}{\sigma_g}\right)\left[1.5 - \Phi\left(\frac{|g|\cdot\eta}{\sigma_c}\right)\right] dg && (3.32) \\ &= 2\int_0^{\infty} \frac{1}{\sigma_g}\mathcal{N}_{0,1}\left(\frac{|g|}{\sigma_g}\right)\left[1.5 - \Phi\left(\frac{|g|\cdot\eta}{\sigma_c}\right)\right] dg && (3.33) \\ &= 1.5 - \frac{2}{\sigma_g}\int_0^{\infty} \mathcal{N}_{0,1}\left(\frac{g}{\sigma_g}\right)\Phi\left(\frac{g\cdot\eta}{\sigma_c}\right) dg && (3.34) \\ &= 1.5 - \frac{2}{\sigma_g}\left(\frac{\sigma_g}{2\pi}\left(\frac{\pi}{2} + \arctan\left(\frac{\sigma_g}{\sigma_c}\eta\right)\right)\right) && (3.35) \\ &= 1 - \frac{1}{\pi}\arctan\left(\frac{\sigma_g}{\sigma_c}\eta\right). && (3.36) \end{aligned}$$

Here, we have used the identity given by Owen [130].

$$\int_0^{\infty} \mathcal{N}_{0,1}(ax)\Phi(bx)\mathrm{d}x = \frac{1}{2\pi|a|}\left(\frac{\pi}{2} + \arctan\left(\frac{b}{|a|}\right)\right). \quad (3.37)$$

In order to change the AP on layer l, it is sufficient to change one single activation $a_i^{(l)}$. This means, the likelihood of the *AP* not changing is

$$\begin{aligned} D_l(\eta) &= 1 - \prod_{i=1}^{d_l} \Pr\left(a_i^{(l)}(x) \text{ unchanged}\right) && (3.38) \\ &= 1 - \left[1 - \frac{1}{\pi}\arctan\left(\frac{\sigma_g}{\sigma_c}\eta\right)\right]^{d_l}. && (3.39) \end{aligned}$$

Data distribution: Taking the data distribution into account requires to integrate the probability over $p(x)$. If we assume that the variances of the bias, σ_b, and

the gradients of the bias, $\sigma_{\Delta b}$, are zero, then is the ratio $\frac{\sigma_g}{\sigma_c}$ independent of the distribution of x, as seen by taking the ratio of Equations 3.20, 3.21. Correspondingly, Equation 3.39 is the final result over the whole data distribution when no biases are used. □

Empirical Distance Fitting Quality

The earlier sections have mentioned two models for $D(\eta)$. This section discusses the prerequisites of each and also tests the models in practice:

$$D_l(\eta) \stackrel{?}{=} 1 - \left[1 - \frac{1}{2}\operatorname{erf}\left(\frac{\sigma}{\sqrt{2}}\eta\right)\right]^{\mu} \tag{3.40}$$

$$D_l(\eta) \stackrel{?}{=} 1 - \left[1 - \frac{1}{\pi}\arctan\left(\sigma\eta\right)\right]^{-\mu} \tag{3.41}$$

$$D_l(\eta) \stackrel{?}{=} \frac{1}{2} \cdot \left(1 + \operatorname{erf}\left(\frac{\mu + \log\eta}{\sigma \cdot \sqrt{2}}\right)\right). \tag{3.42}$$

(The parameter μ in the second model has been adapted by a minus sign to align with the orientation of the measure in subsequent plots.)

The previous section has given a mathematical understanding of where the error function of Hypothesis 2 may come from, yielding the first model given above. Adding the prerequisite that gradients are normal distributed leads to a different model (second equation). However, both these models are results of strong assumptions: independent Gaussian distributions of weights and data, independent data dimensions, non-vanishing gradients – and most importantly, equal size of ACs. The last assumption is used in the first step of the derivations in Theorem 1 by assuming that the value η is independent of the AP to compute the probability of change for. Section 2.2 has shown that this is, however, not true in general – ACs further away from the origin become larger (see for instance Figure 2.12). Thus, the third model (third row), which is also used in the remaining parts of this chapter, scales the learning rate η logarithmically to model this change in size.

Figure 3.12 has evaluated the reconstruction error of the third model, Equation (3.42). In comparison, the reconstruction of the second model, Equation (3.41), is shown in Figure 3.13 and illustrates a much higher (order of magnitude) higher maximal reconstruction error as the model utilizing the logarithm. Using the three model variants given above to characterize the soft phase transition with only two parameters (mean and variance of the underlying Gaussian of that model), the plot

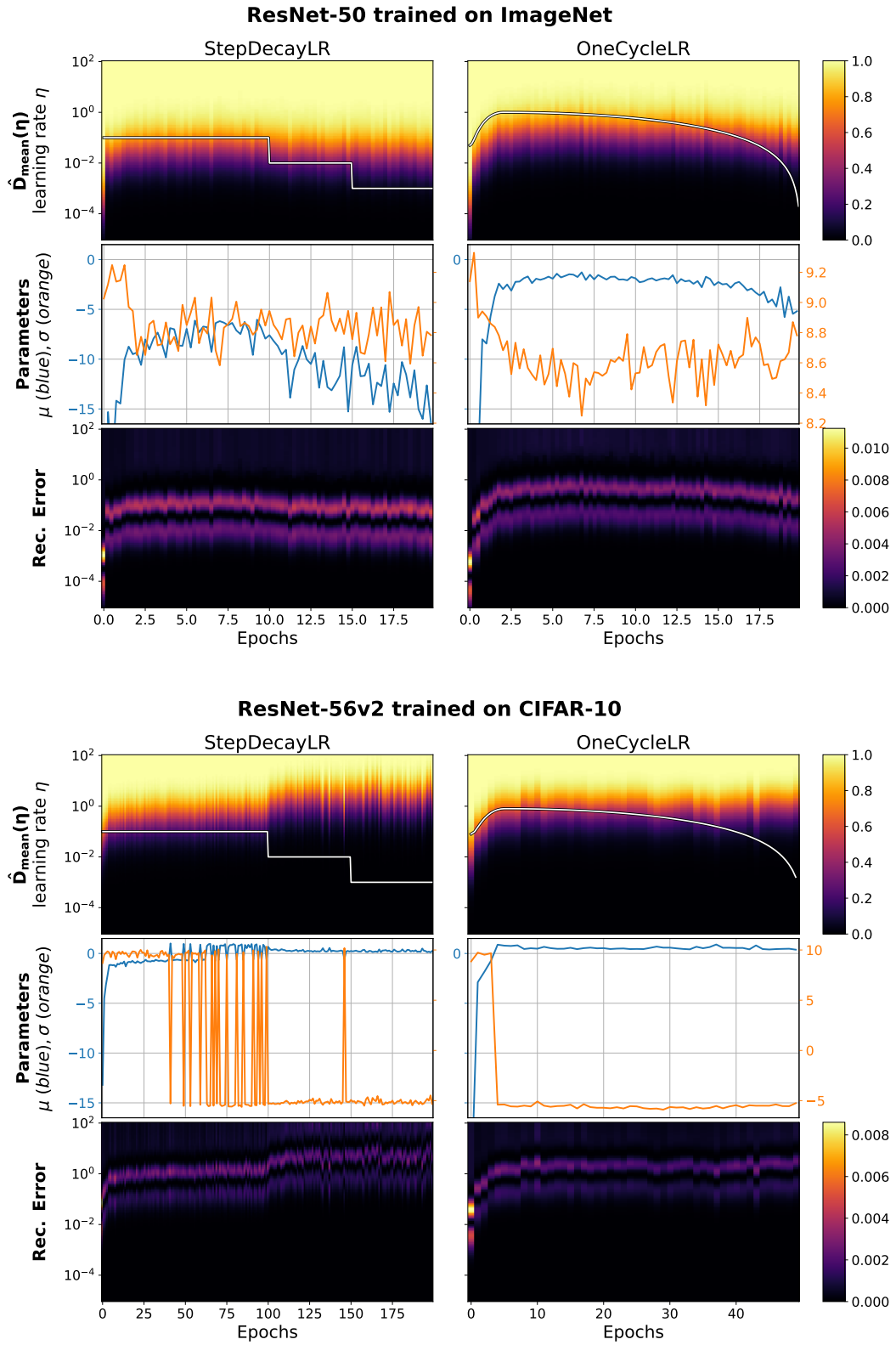


Fig. 3.13.: Accompanying Figure 3.11 and Figure 3.12, the learning rate sensitivity analysis plot (first row) shows also the non-linear least squares fits of the parameters μ and σ for the model given in Section 3.2.3 (second row) and the reconstruction error of the fit (third row).

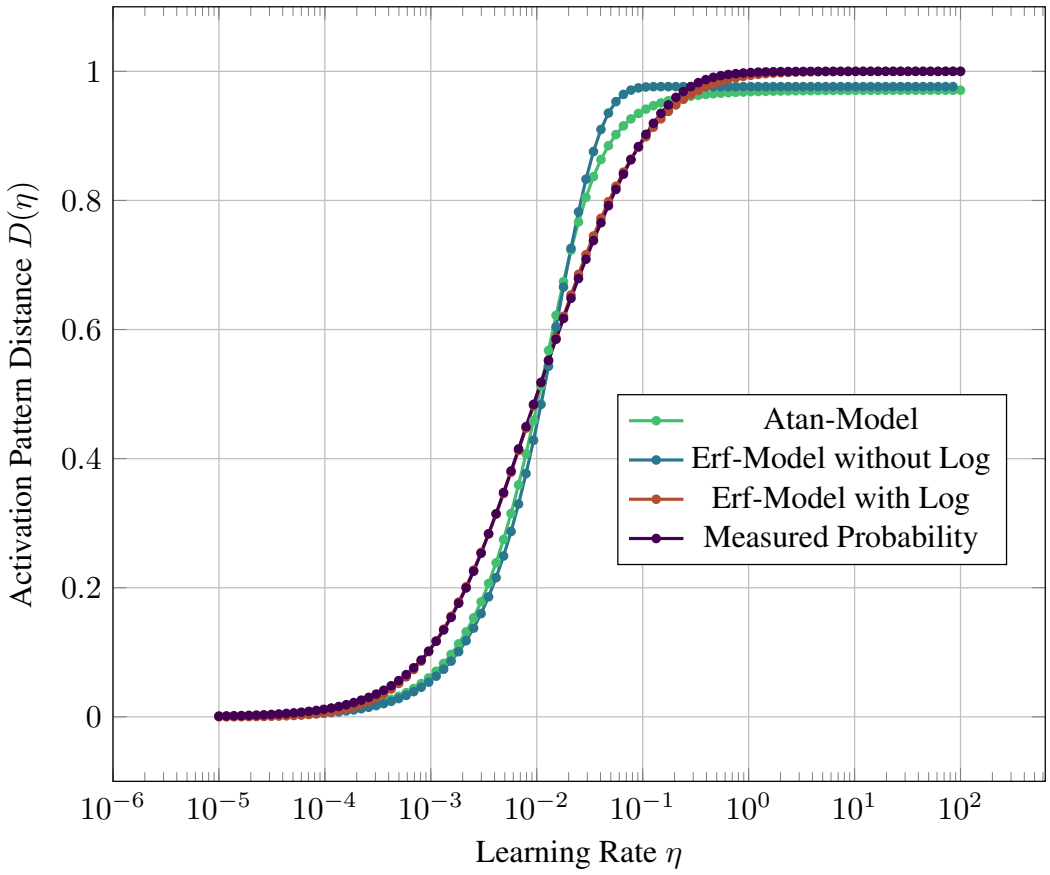


Fig. 3.14.: Comparison of least squares fits of the three suggested models of $D(\eta)$ tested against the initialization state of the experiment visualized Figure 3.11. The best model in terms of qualitative reconstruction error is the one given by Equation (3.17).

depicted in Figure 3.14 shows the best least squares fits for all three variants and the actual measured probability for the initialization of the ResNet-56 and the first step of optimization to classify the CIFAR-10 dataset (the same experiment that has been utilized before). During initialization, most prerequisite of the derivation of model in row one and two should be met. However, in practice, the fit using the logarithm to compensate for the changing size of ACs yields best fitting results. The rest of this thesis thus assumes that Hypothesis 2 is true, and utilizes this model for all subsequent experiments, as well as for developing a learning rate scheduler as described in Section 3.3.

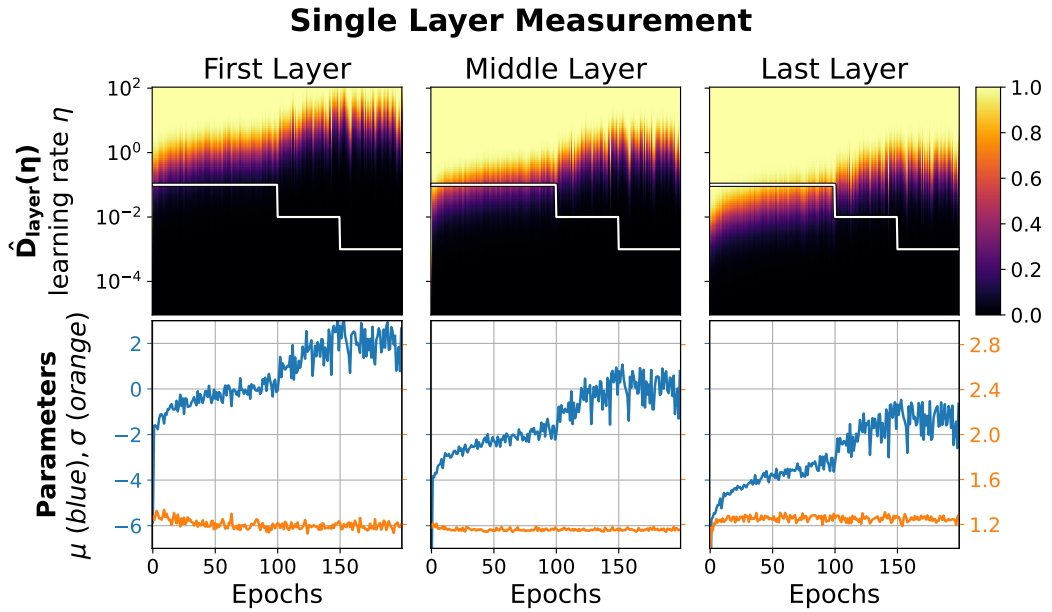


Fig. 3.15.: Activation pattern distance $\hat{D}(t)$ over training time t showing per-layer temperature.

3.2.4 Understanding the Training Process with the APT

Deep learning offers a wide variety of architectures, training methods and settings to choose from in order to train a model. For instance, the choice of hyperparameters for training or how to schedule the learning rate over the course of the entire training time remains a matter of trial-and-error (see Section 3.1.1 and Section 3.2.1 for a discussion of related work). In this section, we will examine various network features and training schemes through the lens of the APT to get a better understanding of their impact on training behavior.

In order to isolate the different effects taking part in training deep networks, the individual experiments are kept as simple as possible. In the following, image recognition on CIFAR-10 is used as benchmark, applied to different CNNs. The experiments consists of two parts:
First, measuring phase-diagrams similar to Figure 3.11 that show the training process of different CNNs over time with markedly different architectural features and optimization techniques typically used for training. The parametrization invariant measurements suggest in most cases quite clearly a connection between training dynamics and known first-order network-sensitivity effects like vanishing and exploding gradients.
Second, by mapping the model parameters given in Hypothesis 2 to any point in

training allows to analyze absolute and relative sensitivity shifts during training separately.

The phase-diagrams depicted in Figure 3.15 to Figure 3.20 show the mean probability of AP change (denoted "distance" in the following), $D(\eta, t)$ at each training step t for a range of learning rates η. The white line in each plot indicates which learning rate has actually been used for training – training uses a simple step decay learning rate schedule and batch normalization unless stated otherwise. Unlike Figure 3.12, training uses no weight decay or momentum (using "plain" SGD), unless noted otherwise to view their effect in isolation. Below each resulting phase-diagram, estimates of μ and σ of the Gaussian transition model are shown, computed using batch-wise least squares fitting of the model given in Hypothesis 2 to all measured values.

Interpretation: Low values of μ correspond to high sensitivity (phase transition occurring already at low learning rates, larger yellow area overall in the following diagrams), and high values indicate low sensitivity to changes. States with high values of μ are denoted "colder" and states with low values of μ are denoted "hotter" in the following. (Think of μ as the amount of work required to effect a significant and meaningful change in the network). On the other hand, σ captures how abrupt the transition occurs. The most prominent feature, of a soft and monotonic phase transition has already been explained analytically in the previous Section Section 3.2.3 and is clearly visible in all scenarios examined in the following.

Per-Layer Measurements: Figure 3.9 has already revealed that during training, the layers change APs with different probability – deeper layers have higher change rates compared to those near the input of the network. Figure 3.15 validates this result by inspecting the probability landscape for different layers in isolation, but this time independently of the learning rate chosen for training. Compared to the earlier mentioned plot, the figure's y-axis represents learning rates instead of network depth. The three heatmap plots show the same network evaluated in the first layer, a middle layer and the last layer of a ResNet-18 model without residual connections. Notably, the first layer is generally "colder" than the last layer for the full training cycle (note how μ is shifted vertically), hinting to different layers learning at different paces. Also the course of the transition spreads (σ) differs for the last layer where it increases first, while in contrast it decreases in the first and middle layers.

Residual Connections: Figure 3.16 shows the mean probability across all layers for ResNet-56 with and without residual connections enabled. As expected, without residual connections the width of the transition phase increases, as the probability levels increase with layer depth. Residual connections reduce this variance and

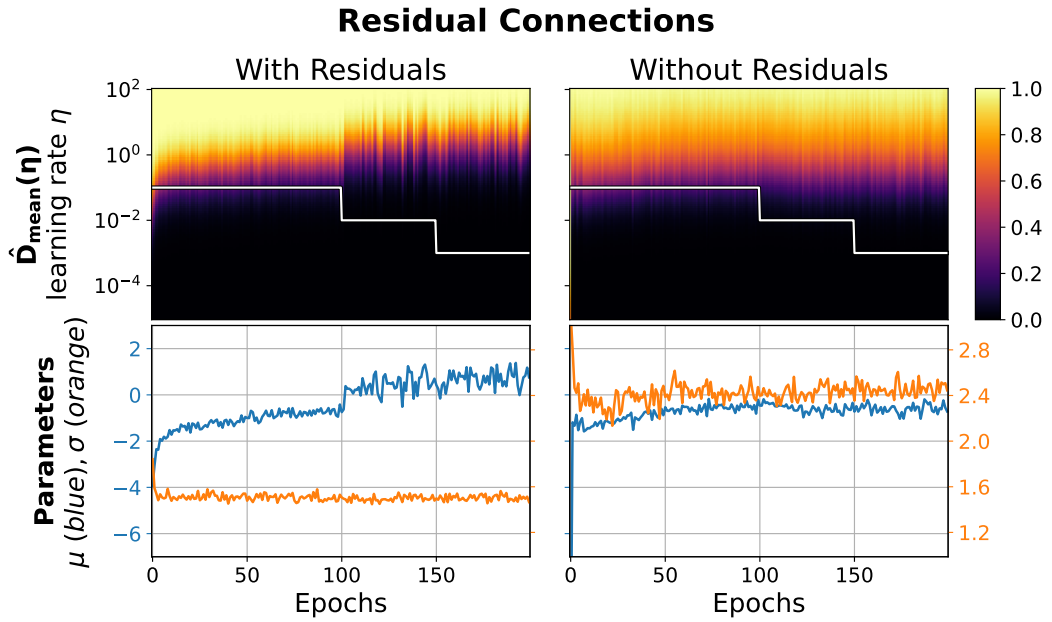


Fig. 3.16.: Activation pattern distance $\hat{D}_{\text{mean}}(t)$ over training time t for networks with and without residual connections.

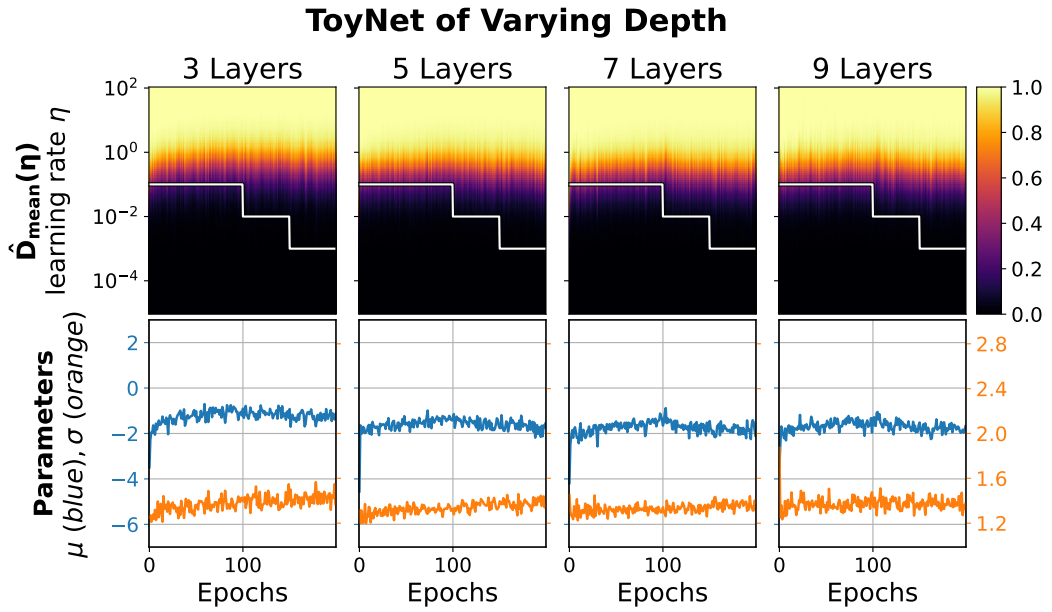


Fig. 3.17.: Activation pattern distance $\hat{D}_{\text{mean}}(t)$ over training time t for varying depths of ToyNet without batch normalization and with 32 filters in each layer.

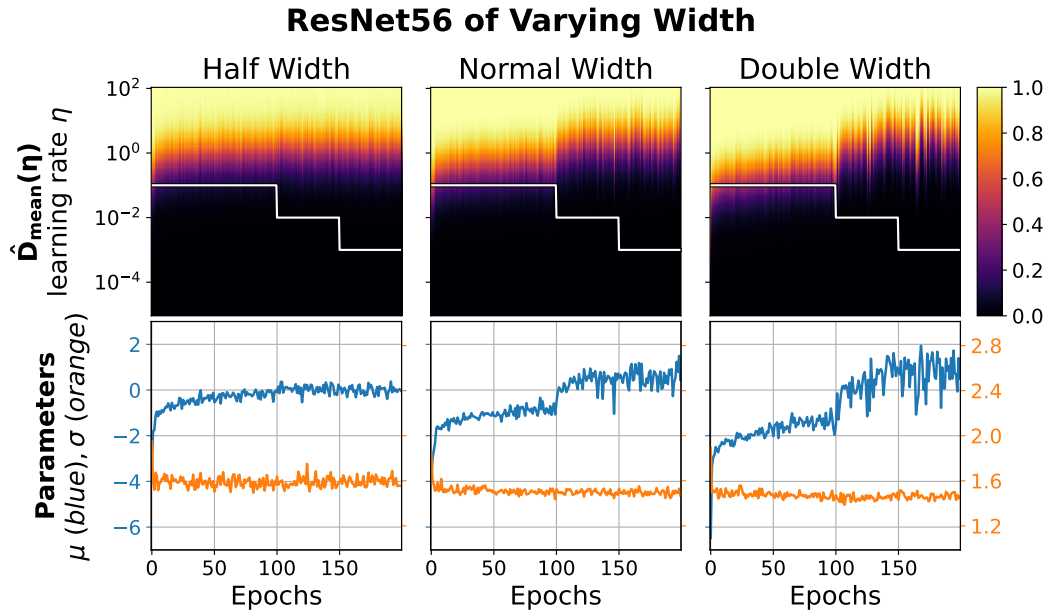


Fig. 3.18.: Activation pattern distance $\hat{D}_{\text{mean}}(t)$ over training time t for varying widths in ResNet-56.

stabilize σ across training time. Interestingly, when residual connections are not present, changes in the learning rate have no impact on the model's sensitivity, in contrast to a model that does utilize residual connections. The next experiments will also focus on the mean probability across all layers.

Network Width & Network Depth: As shown in Figure 3.17, also the mean sensitivity of a network increases with depth, however only moderately; the figure shows a simple ToyNet, where no other architectural features may blur the effect. In contrast as shown in Figure 3.18, network width increases the sensitivity more substantially, as the likelihood of AP changes increases (the measure D is not invariant to the number of filters; Equation (3.41) shows an exponential dependence at initialization). Also σ decreases with width in accordance to that model, Equation (3.41) also predicts that the transition curve becomes steeper with increasing number of filters.

Early Divergence Detection & Normalization-less Training While the sensitivity generally decreases for all networks during the early training epochs, networks with batch normalization, however, behave qualitatively different from un-normalized ones. We know from literature that it is batch normalization that causes exponentially exploding gradients at initialization [185], but it is also its implicit regularization that subsequently equalizes gradient magnitudes early in training [8]. The networks without batch normalization do not experience this initial excursion; on the contrary,

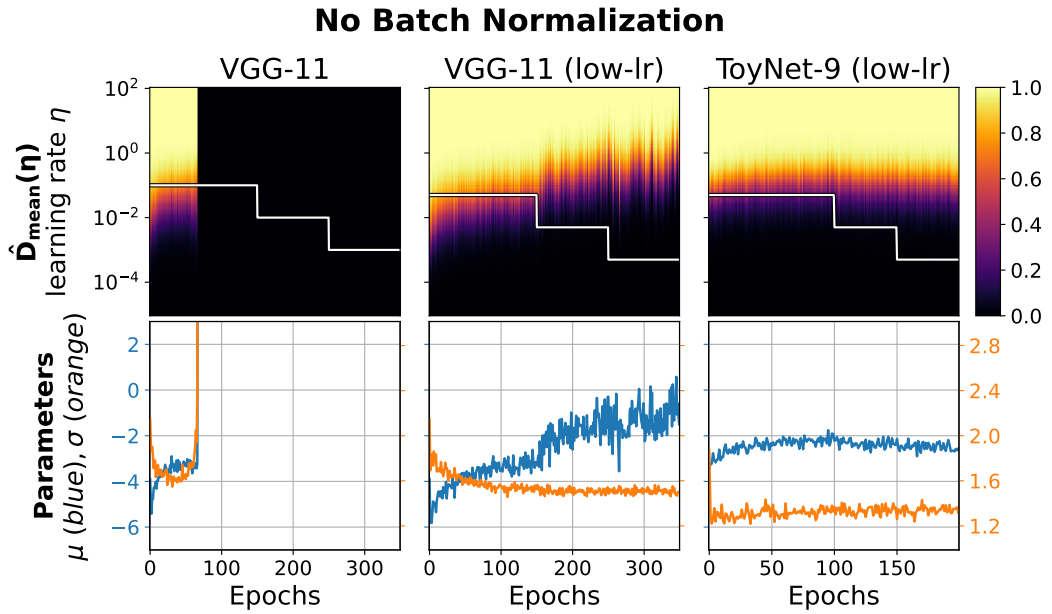


Fig. 3.19.: Activation pattern distance $\hat{D}_{\text{mean}}(t)$ over training time t for networks with batch normalization turned off.

the VGG-network without batch normalization Figure 3.19 even starts with reduced sensitivity, probably due to vanishing gradients. Same goes for the ToyNet without batch normalization shown in Figure 3.17. Accordingly, different warm-up strategies are required with and without batch normalization. Additionally, to the sensitivities shown for low learning rates (to prevent divergence), Figure 3.19 also contains a VGG-11 that does diverge (left plot), which is foreshadowed by the divergence of σ. Experience has shown that this value has always fallen in the long term - and a rapid rise seems to imply a direct divergence (see also the experiment visualized in Figure 3.21).

Correlated Gradients & Momentum: A further prominent feature is that the value of μ performs discontinuous jumps at changes of the learning rate (see for instance Figure 3.21). This is explained and related to changes in smoothness of the loss landscape when step size is decreased [98]: As the step size decreases, the gradient directions may shift abruptly, pointing towards local minima with different noise properties in the surrounding loss landscape. Consequently, the new direction also affects the sensitivity of AP changes. The effect is obscured in the layer-average for the un-normalized ResNet models, but clearly visible in the layer-wise plots Figure 3.15, but also mean sensitivity plots (e.g. in Figure 3.18). The general tendency of a decrease in sensitivity (growing μ) in all plots would also be consistent with the observation that smoothness of the local loss landscape increases during training [115, 11, 65]. A similar effect is caused by momentum, where high values

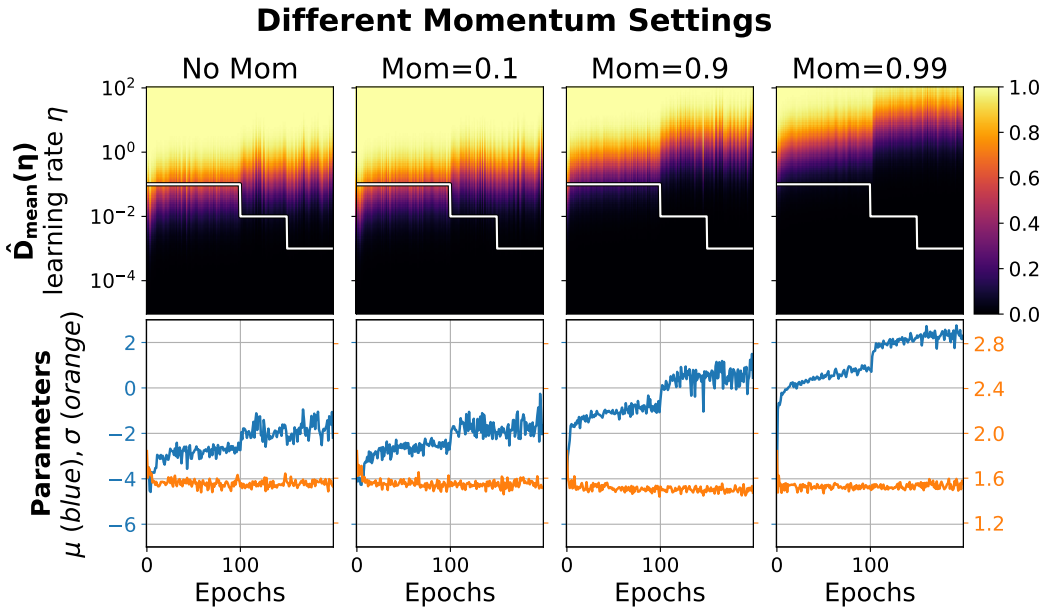


Fig. 3.20.: Activation pattern distance $\hat{D}_{\text{mean}}(t)$ over training time t with varying momentum values.

correlate gradients especially towards the end of the training [91], and which leads to a visible decrease in sensitivity, especially for larger momentum values as can be seen in Figure 3.20.

Weight-Decay Two types of weight-decay exist: either by adding a l_2-regularization term of the weights to the loss to be minimized (denoted l_2-weight decay), or, by adapting the optimizer (denoted SGDW, or SGD-weight decay) to reduce weights by a constant factor with each optimization step. For a constant learning rate both result in the same update, however, as learning rates are scheduled during training, both types of weight-decay play a different role, which has been described already by Loshchilov and Hutter [107].

Figure 3.21 shows that SGD weight decay increases sensitivity in a batch normalized ResNet (analyzed also by Li and Arora [102] and Zhang et al. [190]) as expected, since constantly decreasing weight-magnitudes lead to larger effective steps for the same learning rate η . In terms of APs, a decrease of weight magnitudes decreases AC sizes (as illustrated in Figure 2.6), explaining the effect. At smaller learning rates, thus more weight dampening is performed, increasing sensitivity. Lower weight decay-values reduce the effect. A too large weight decay-value is counter-productive for optimization, because the resulting decay magnitudes exceed that of the loss gradients, which may even lead to divergence of the system (see the column labeled "sgd10^{-3}" in Figure 3.21). No such effects are observed with l_2-Regularization variant, where dampening is independent of step size; here the correlation effect of

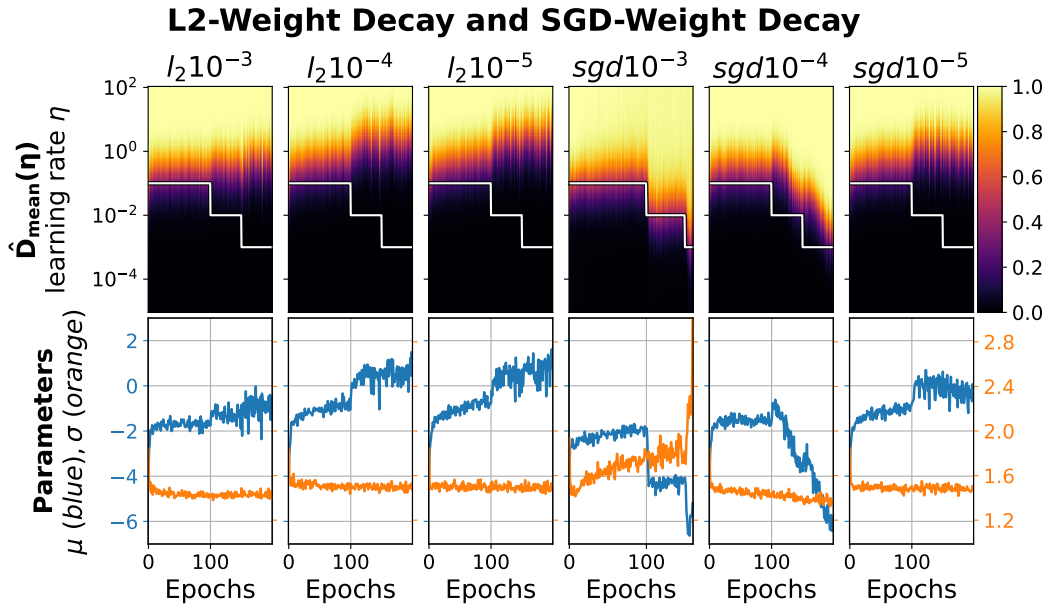


Fig. 3.21.: Activation pattern distance $\hat{D}_{\text{mean}}(t)$ over training time t with varying levels of weight decay.

smaller step sizes is still visible, and it increases when reducing weight-decay, as this effectively leads to smaller steps.

3.2.5 Discussion of the Activation Pattern Temperature

Scheduling the learning rate (LR) in deep neural networks is a complex problem, with significant effects on the achievable training time and generalization performance. Practically proven SOTA schemes, such as One Cycle learning rate schedules, offer substantial gains over baseline methods, but the reason for the success is not well understood. Differences of resulting model accuracies have led to research on the topic of training phases (see Section 3.1.1 and Section 3.2.1 for a discussion). This section has reviewed training dynamics of Rectified Linear Unit (ReLU) networks through the lens of the activation pattern (AP) framework, which measures changes in the non-linear behavior of the network. The ideas have led to a new measure, AP distance (and its log-scaled version the Activation Pattern Temperature (APT)) for comparing the function computed by the layers in a ReLU network for different learning rate choices. This measures are characterized by the likelihood that APs will differ for the same data used before and after a single training step. Both measures can be computed quickly, at the cost of one additional forward pass. In contrast to the activation pattern entropy (APE) analyzed in Section 3.1.5, this approach concentrates on the dynamics during training.

This section has suggested an effective model that connects the network state to the learning rate, thereby providing a predictive view of how the non-linear behavior of the ReLU network changes over time. On a conceptual level, the observations of the structure of how APs change provides new insights into the complex training dynamics of deep networks such as; changes in linear dampening or amplification due to varying exploding or vanishing gradient effects, smoothness of the loss landscape and, effects of training techniques such as weight-decay.
In a practical context, the cost-effectiveness of using this measure as a monitoring tool during training makes it a viable option for early detection of divergences. Overall, the behavior described by earlier work on network sensitivity or training behavior are also qualitatively described by the APT. The large number of qualitatively correct predictions make it plausible that the measure can effectively capture these effects, and can thereby in the next section, Section 3.3, be automatically compensated for using a constructive algorithm.

3.3 ActCooLR – High-Level Learning Rate Schedules using Activation Pattern Temperature

This section is based on the manuscript:
ActCooLR – High-Level Learning Rate Schedules using Activation Pattern Temperature
David Hartmann, Sebastian Brodehl, Michael Wand
`https://openreview.net/forum?id=yqj6q_eNTJd`

The previous section decoupled the learning rate used for optimization from a network's intrinsic sensitivity during training. The idea is to re-evaluate the same training step (same data, without applying the gradients) and measuring the proportion of an activation pattern (AP) changing during that step for a range of learning rates. Learning rate schedules can be seen as methods that "choose" a path through that transition spectrum that ranges from "no updates" (no change of APs) to "full modification" (all APs changed). This transition landscape is however not static, but changes during training time with architectural choices, network depth, optimizer hyperparameters (see Section 3.2.4) and even learning rate schedules themselves (see Figure 3.11).
This section verifies these observations by constructing a method that performs successful learning rate scheduling solely based on the discrete information of current and desired AP changes in the network. Functional profiles start at high temperatures and gradually decrease until the APs stabilize, at which point optimization converges to a linear regime (see Figure 3.8). As the proof-of-concept activation pattern temperature (APT) scheduler matches the performance of previously hand-tuned learning rate schedules, it could open up the research of effective and efficient, hyperparameter-free automatic learning rate-schedules.

3.3.1 Related work on Hyperparameter Schedules

Hyperparameter choice has a strong effect on (generalization) performance and convergence speed. From a broader point of view, optimization has been an active field of research for decades, not only in the context of neural networks. (See [151] for a general overview and [51, 119] for an introductory text in the context of neural networks, specifically discussing stochastic gradient descient (SGD) [143].) Especially recent works have pushed state-of-the-art performance by evolving specific niches of optimization techniques like faster converging learning rate schedules, better network initializations, or activation functions:

Economically, under fixed training budgets, hyperparameter choice often leads to a trade-off [25, 180]. In cases of small batch sizes, one crucial invariant control parameter has been shown to be the ratio of learning rate and batch size [54, 65]. Further studies prove that SGD itself has an implicit bias toward flat regions, reinforced by large learning rates [11, 161]. Additionally, there is strong empirical evidence that large initial learning rates can help with generalization in over-parameterized networks [96, 101]. However, training of large networks requires a "warm-up" phase to prevent divergence of deeper layers [53] in the early phase of training (as discussed in Section 3.1.1). While most schedules let the learning rate approach zero toward the end of the training, especially the course of the learning rate in the middle of the training process has not yet been analyzed extensively.

Learning rate scheduling is considered to be directly linked to generalization performance [65, 75, 76, 91, 101]. For instance, a cyclical learning rate schedule enables to train networks with a good "anytime performance" [108] and the implicit learning rate schedules that are built into adaptive optimizers such as the Adam Optimizer [82] are topic of current research [4, 109]. However, specific research in learning rate schedules is sparse. Although, correctly tuned, OneCycleLR achieves the same accuracy using order-of-magnitude fewer training iterations [160], "large models in NLP and vision use schedules which can be easily resumed" [95], using "clipped" cosine decay [22] or exponential decay [171] learning rate scheduling. Research has been also done in automatic and adaptive hyperparameter tuning. For instance, automatic tuning of decay time for the exponential decay schedule, [89, 95], and meta-networks designed to predict the learning rate based on learned typical training courses ([66], or more specifically, [36, 183]) have been tested. Another approach includes the learning rate into the optimization process by deriving the loss w.r.t. the learning rate as well [12].

Well-established techniques, such as weight decay have been a point of discussion related to the generalization gap between SGD and adaptive optimizers [107, 180]. Also, it has been shown to affect the learning rate scheduling directly when used in conjunction with batch normalization: every weight decay step increases the effective learning rate by a multiplicative factor for a constant learning rate schedule [102, 190].

Several papers discuss the use of statistics to guide optimization methods [89, 183], but only use the training and validation losses to approximate the training behavior. Moldovan et al. [118] use the so-called transfer entropy between network nodes to guide back-propagation. Other works use statistics for feature extraction [42], feature pooling [176] or network compression [179]. Related to that is the research

on the distribution of activations, which often treats all neurons as independent stochastic variables and has proven helpful for derivations of initialization schemes and methods to help with training [147, 73, 61, 50].
In contrast to these works, our emphasis lies on the study of the discrete distribution of layer-wise APs in a deep neural network, explicitly analyzing their internal non-linear structures.

The most similar of the listed methods to the presented method in this section from an optimization perspective is probably that of Roos et al. [145], where successive training steps are used to estimate the change of curvature of the loss function to adapt the learning rate automatically at the cost of multiple re-evaluations of the loss to estimate the local neighborhood.

3.3.2 Activation Cooling based Learning Rate Scheduler

The learning rate sensitivity analyses conducted in Section 3.2.3 reveal that learning rate schedules potentially affect the network's global sensitivity, and thus, a learning rate scheduler also implicitly has to follow that change over training time – this holds true for OneCycleLR as well as a constant learning rate. On the other hand, the APT measures the actual non-linear change of a network that represents the convergence of a network's APs directly, taking the value 0 precisely when no APs change across one single optimization step for the entire data distribution. Thus, the canonical extension of the learning rate sensitivity analysis experiments are to use that knowledge to automatically adapt to changes in the network's intrinsic sensitivity.

The following section outlines a basic, prototypical automatic learning rate scheduler that prescribes temperatures over training time. The focus at this point is in understanding the training process; yet to validate to be a feasible tool in practice since a learning rate scheme needs to have a low computational overhead.
Importantly, the following text will explain how the algorithm works; to get a more structured representation of the full algorithm that also covers corner cases and tricks that makes the algorithm more robust in practice, see Appendix A.4.

Efficient Probing of μ and σ

The goal of the scheduler is to determine learning rates that impose a specified temperature profile. This has to be done online, adapting the learning rate η to the

current network state. Thus, the key ingredient for efficient automatic scheduling is a quick approximation of the soft phase-transition via Equation (3.17), as it allows to predict learning rates from only two parameters μ and σ.
The problem with the model given in Equation (3.17) is that the model parameters σ and μ are not stationary, but change during training, and additionally are also dependent on choice of architecture, training techniques and the history of used learning rates throughout training (see Section 3.2.3). Thus, the first problem to tackle in order to use the model its efficient approximation by estimating μ and σ using only few samples.

Efficient Initial Model Fit: The equation given in Equation (3.17) models the probability of AP changes for a single optimizer step given a learning rate η. As the model describes a monotonically increasing transition from 0 (no changes) to 1 (all APs changed) one option would be to predefine a fixed list of learning rates to measure the actual probability on, and then, to least squares fit the model parameters μ and σ to these results. This would be a viable solution, as only three sample points are enough to get good approximations for μ and σ, if chosen learning rates are chosen so that resulting measured probabilities fill the range of $[25\%, 75\%]$ of changed APs and assuming noise of measurements is low enough. However, as shown in the plots of Section 3.2.3, the variance of the learning rates used for learning rate sensitivity analysis plots varies by several orders of magnitudes in-between architectures, but also in the very same training run (see for example Figure 3.19). Thus instead, a more efficient approach is to do bisection search to estimate a learning rate achieving a probability above 75%, and, another bisection search to estimate a learning rate achieving a probability below 25% of changed APs.
Importantly the model' closed form formula depends logarithmically on the learning rate for a fixed network state. Thus, the bisect search works also on a logarithmic scale, multiplying or dividing the last learning rate guess by a factor of 2. When the requested range of probabilities is measured, all estimates computed on the way can be re-used to fit the model.

Corner Cases: Note that this technique still has some cases requiring special treatments: consider measured probabilities of 1.0 or 0.0 – which both are not achievable using the model as it only returns 0 and 1 for the limits $\log \eta \to \pm\infty$ due to the logarithm and error function being involved. To alleviate numerical instabilities of fitting the model near the measured probabilities of 0 (no APs changed), the learning rate is capped to a minimal learning rate of $1e-10$, whenever it would drop below this threshold. While the model can be extended to allow probabilities near zero as just explained, a probability of 1 would correspond to an infinite learning rate.

In practice this special case can (and does) occur, since the probability of changed activations is only sampled using a finite batch that can happen to change all APs at once. Thus, these measurements are excluded from the list and filled by further bisection search calls in case less than three points are in the list of measurements.

Online Model Update: During training a first guess can be easily obtained by reusing the previous model parameters μ_{prev} and σ_{prev} as follows. First, rearranging Equation (3.17) gives a formula to estimate learning rates given a probability, and given the two model parameters:

$$\eta = \exp\left(\sigma \cdot \sqrt{2} \cdot \operatorname{erf}^{-1}\left(2 \cdot P_\eta - 1\right) - \mu\right), \tag{3.43}$$

where P_η denotes the probability given the used learning rate η. This formula can be used to estimate learning rates that (as a first guess) surround the probabilities 0.25, 0.5 and 0.75. As mentioned above, these (additional) measurements can be used to improve the model-fit after the bisection search.
To reduce the probing size for the model during training further, probability measurements are only done every 8 steps[9] an observation from Section 3.2.3 can be used: the model parameter μ undergoes a lot of noise and is susceptible to training behaviors, but the parameter σ changes on a much slower pace. Thus, another re-arrangement of Equation (3.17) can be utilized,

$$\mu = \sigma \cdot \sqrt{2} \cdot \operatorname{erf}^{-1}\left(2 \cdot P_\eta - 1\right) - \log \eta, \tag{3.44}$$

to "correct" the parameter μ on smaller time scales. To prevent over-fitting of the model parameters, the more expensive bisection based model initialization is also used when the correction surpasses 2.5 μ-units more than 4 times since the last model update, trying to "ignore" the expected variance of μ itself.[10]

Optimization using ActCooLR

Assuming up-to-date model parameters for the current state of AP probability exists, for example using the method described above, the next step is to define a temperature curve, a drop-in replacement to a learning rate schedule, but taking sensitivity changes of the training process itself into account automatically. As a design decision, we have the option to measure temperatures at every layer, and using adaptive learning rates, even to specify them layer-wise. For simplicity the

[9] This value gave good results in terms of balancing stability and speed on the experiments given later in this section.
[10] Again, this number is chosen to reduce probing costs for the set of experiments in this section.

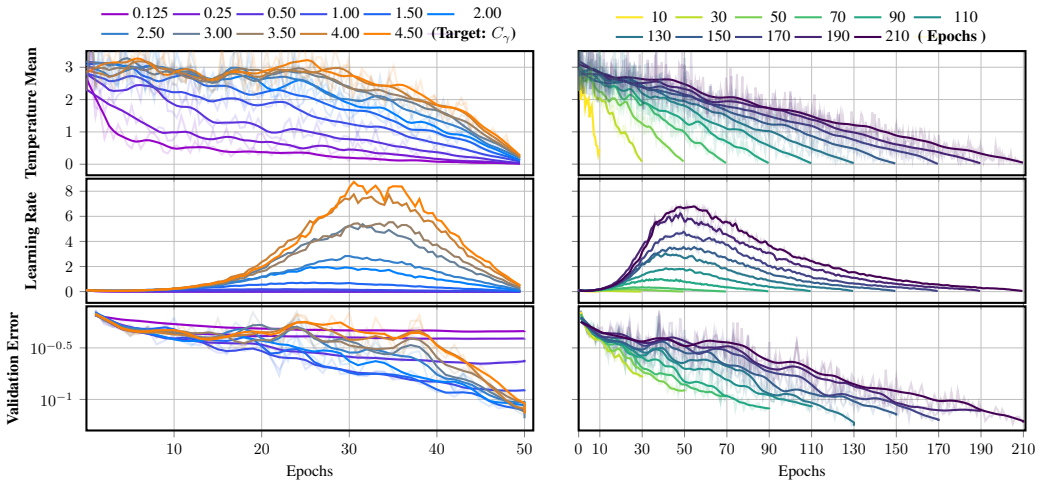


Fig. 3.22.: Training a ResNet-32 on CIFAR-10 with "ActCooLR". Left: varying γ. Right: varying number of epochs ($\gamma = 1$). We use $T_{\text{start}} = 3.17$, i.e. $D = 89\%$. Top row: measured mean temperature, middle: learning rates, bottom: top-1 validation error.

fine-grained adaptation are left for future work and instead the proof-of-concept generally operates with a global learning rate (similar to OneCycleLR) and use the mean temperature over all layers for control.

As the fast learning rate scheduler OneCycleLR has an approximately linear temperature decrease (see Figure 3.8), the temperature schedule suggested in this section is very simple, starts at value T_{start} and decreases to zero over t_{max} optimization steps according to:

$$T(t) = T_{\text{start}} \cdot (1 - t/t_{\text{max}})^{\gamma},\ \gamma \geq 1. \tag{3.45}$$

The parameter γ allows to bend the temperature curve as to spend more time early on at higher temperatures. For example, $\gamma > 1$, $\gamma = 1$ corresponds to a linear decrease of T, i.e., and exponential decrease of D. Empirically, $\gamma = 4.0$ yields good results in most cases, but may be tuned as well to increase performance.
Given model parameters μ and σ, and a target temperature T_t, to actually determine the learning rate required to achieve that temperature, Equation (3.43) can be used again by using the currently estimated values for P_lambda, σ and μ. In analogy to the fact that the layers are getting "colder" due to the steady decrease of their temperature, this learning rate adaption technique is called "ActCooLR".

The main parameter that needs to be tuned is the length of the training (as with other learning rate schedules as well) and the start temperature T_{start}. Figure 3.22 shows the effect of the selected target temperature curve (top row) on the learning rate (middle row) and the validation error (bottom row) for a ResNet-32 model trained

Tab. 3.1.: Test errors for Figure 3.22

Epochs	Test Error (Top-1)
10	37.88%
30	18.33%
50	11.87%
70	10.59%
90	9.09%
110	8.09%
130	7.50%
150	7.28%
170	7.18%
190	7.15%
210	6.41%

on CIFAR-10. For comparison, the Figure shows a variation of training lengths for the same temperature curve parameters and also for a variation of choices of γ when training length remains constant. Most notably, all settings show the same "One Cycle-like" behavior of increasing the learning rate first, and then reducing it with an exponential-like decay. The γ value explicitly controls the point in time at which the maximum is reached. The linear temperature schedule, $\gamma = 1$, resembles OneCycleLR the most in shape and position of the maximal value. However, in absolute terms, the learning rate used in a stable training run with "ActCooLR" is much higher – in this examples learning rates surpassing a value of 8 can be observed, while OneCycleLR typically has it's maximum at a value of about 0.8 in this experimental setup.

In Table 3.1 (showing the performance of the schedules shown in Figure 3.22(b)) the disadvantage of the proposed schedule is evident: the validation error remains high for the most number of epochs during training, converging only very late compared to methods like CyclicLR or StepDecayLR. For instance, in contrast to cyclical learning rate schedules, ActCooLR does not show a good anytime performance. On the one hand, cyclic temperatures could also work, but need to be evaluated separately, thus leaving this as topic of future research.

Comparisons with other Methods

Table 3.2 shows the results of automatic LR-scheduling in benchmark scenarios. All hyperparameters and details about the architecture are provided in Appendix. Several networks trained on classification tasks are listed; every experiment is repeated

Tab. 3.2.: Automatic LR-Scheduling Results.

Dataset	Network	Training Length	Method	Top-1 Accuracy
FashionMNIST [182]	**F_1**	15 Epochs	StepDecayLR	**90.6%**
		15 Epochs	**ActCooLR**	89.8%
Speech Commands [177]	**M5** [35]	50 Epochs	StepDecayLR	**86.0%**
		50 Epochs	OneCycleLR	82.7%
		50 Epochs	**ActCooLR**	85.7%
CIFAR-10 [86]	**VGG-16** [158]	50 Epochs	StepDecayLR	90.6%
		50 Epochs	OneCycleLR	90.3%
		50 Epochs	**ActCooLR**	**92.5%**
		200 Epochs	StepDecayLR	91.3%
		200 Epochs	ABEL [95]	92.9%
		200 Epochs	OneCycleLR	91.7%
		200 Epochs	**ActCooLR**	**93.0%**
CIFAR-10 [86]	**VGG-19** (w/o Norm.)	50 Epochs	StepDecayLR	88.6%
		50 Epochs	OneCycleLR	90.2%
		50 Epochs	**ActCooLR**	**90.4%**
Tiny ImageNet [90]	**PyramidNet-41** [57]	20 Epochs	StepDecayLR	52.7%
		20 Epochs	OneCycleLR	54.7%
		20 Epochs	**ActCooLR**	**55.5%**
ImageNet [146]	**ResNet-50 (v2)** [62]	20 Epochs	StepDecayLR	68.2%
		20 Epochs	OneCycleLR	**72.4%**
		20 Epochs	**ActCooLR**	70.9%

three times and the mean is reported. Hyperparameters are optimized manually on validation (not test) data for all methods. For ActCooLR, generally large, constant values of momentum are used and for T_{start}, starting with a high-value, and lowering until training does not diverge. Weight decay is set to zero. For OneCycleLR, the LR-range test is used according to the original paper [160], along with a momentum schedule that is both used and manually tuned for optimal results. For VGG-16, this approach leads to divergence and a maximum learning rate was set manually. For step-decay, step reduction is always set to a multiplicative factor of 0.1 per step; otherwise parameters were tuned manually (the LR-range test constitutes a good starting guess for learning rates as well). Results of ABEL are taken from [95].

As a main result, test accuracies for ActCooLR are comparable to OneCycleLR (sometimes slightly better and sometimes slightly worse than OneCycleLR). In general, short training periods on difficult datasets are unfavourable to base-line step-decay, enlarging the gap. The base-line method is only competitive on F_1/FashionMNIST and M5/Speech Commands, but only at a narrow margin. In general, ActCooLR consistently performs close to the best results. However, it does outperforms both StepDecayLR and OneCycleLR on certain experimental setups, most notably for the harder-to-train network architecture VGG.

3.3.3 Discussion: Connecting Learning Rates and Activation Patterns

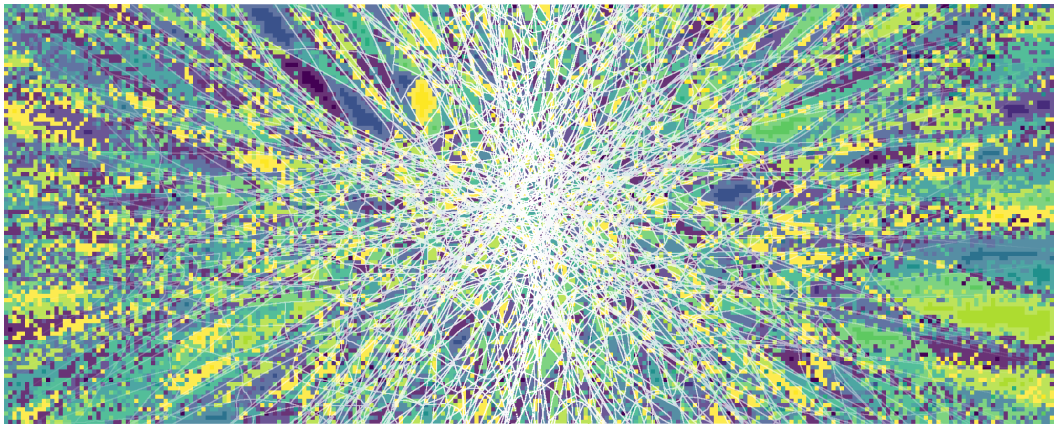
Scheduling the learning rate in deep neural networks is a complex problem, with significant effects on the achievable training time and generalization performance. Practically proven SOTA schemes, such as OneCycleLR, offer substantial gains over baseline methods, but the reason for the success is not very well understood. The key hypothesis of this section is that solely tracking the changes to the non-linear behavior provides enough information needed for adapting step-size control automatically. The experiments support this view, at least for the tested task of classification on image as well as audio benchmark datasets. Specifically, performing a simple linear decrease in activation pattern temperature (APT) already yields a learning rate-scheduler with performance comparable to OneCycleLR, and bending the temperature curve towards staying longer in the high-temperature regime at the beginning appears to improve generalization performance slightly (at least for the training time scales examined). The simple Gaussian two-parameter model of Equation (3.17) approximates the temperature well empirically and leads to a simpler and more efficient understanding of the training process in terms choosing a learning rate. This observation has led to an automatic learning rate-scheduling algorithm that replaces hand crafting the learning rate schedules for each new training setup.

The most important result is, however, probably on the conceptual side: the rather complex learning rate-curve of a cyclical learning rate schedule (including OneCycleLR) appears to just correspond to an annealing of the APT. This is reminiscent of simulated annealing methods which use a very similar strategy in order to solve combinatorial optimization problems. The logarithmic temperature measure has an analogous form to the temperature in the Boltzmann-distribution of a Markov-Chain-Monte-Carlo (MCMC) optimizer used there. In this sense, the method suggests that a good learning rate-scheduler for stochastic gradient descient (SGD), also performs on the discrete, non-linear network components, a process very similar to simulated annealing as introduced by Kirkpatrick et al. [84]. More concretely, Figure 3.11 shows a smooth transition between a linear training regime, with low probability of non-linear changes and a high-temperature, presumably chaotic, regime for high learning rates. By controlling the APT, training can be steered within the band of non-linear, but not chaotic learning automatically, and only converge into the purely linear regime at the end, thereby plausibly obtaining a quicker convergence. Changes of sensitivity at early training steps (see for instance Figure 3.12) are handled automatically, and provides a plausible explanation for the utility of the

initial warm-up phase used in the One Cycle learning rate schedule scheme. It is also interesting that the phase boundary width changes only slowly after that.

Limitations and Future Work: The problem of choosing the learning rate curve arguably has been shifted to a more abstract problem; choosing the temperature curve. It is clear that the mean temperature needs to cool down to a value of zero, specifying explicitly that the network shall converge. This temperature is trivially given by a learning rate of 0. From our experience and also in comply with the observations made in Section 3.2.2, training generally longer on higher temperatures ($\gamma > 1$) achieves favorable performance compared to a faster reduction of temperature ($\gamma < 1$). The case of a too big γ decreases the amount of time the optimization takes place in a linear learning regime considerably, resulting in a worse performance. Many previous work has analyzed the positive effects of large initial learning rates (see Section 3.2.1 for a discussion) – supporting the idea of starting with a large temperature as well. However, despite some promising results with the simple temperature schedule suggested earlier, the overall optimal shape of that temperature curve is not yet clear, especially as the sensitivity of the training process changes with the choice of a learning rate scheduler. I believe that finding temperature curves with theoretical bounds is a promising direction for future work to better understand the internal effects. Particularly interesting is that the method enables training on very large learning rates (far beyond typical choices) without divergence. Nonetheless, a broader study on a large corpus of models and architectures, as well as examining applications beyond image recognition and feed-forward CNNs, is an important next step for future work. Further, a predictive theoretical model of the statistical dynamics of activation patterns (APs) under parameter trajectories and exploring a closer connection of SGD and MCMC optimization would be interesting avenues for future work.

4 Deep Networks of Lower Computational Complexity



> *In AI's deep realm, we seek to refine,*
> *Streamline the complex, in less space align.*
> *Weights turn discrete,*
> *Efficiency sweet,*
> *In balancing simplicity, it's a new design.*
>
> **— ChatGPT,**
> **"Generate a limerick of the following text:"**

In light of the widespread use of deep networks in various practical applications, extensive research has been directed towards streamlining these networks. This includes areas such as network quantization, the creation of architectures optimized for specialized hardware, and even the innovation of simplified, hardware-friendly versions of gradient descent algorithms. Striving towards computationally simpler networks without sacrificing expressivity and generalization performance aligns with a much broader goal: to distill the essential mechanisms of learning and to gain a better understanding of the amount of information required to be propagated from the data through a model into its target response.

Goals and Structure

This chapter focuses on understanding where the transition point of such simplification is. The techniques discussed in the literature try to push this transition point towards more simplified networks, often based on finding ways to define gradients on discrete spaces – either by imposing a gradients onto discretized values, or by slowly increasing regularization towards quantized states. While these present potential ways to train discretized networks from scratch or using a fine-tuning approach, Section 4.2 suggests an alternative approach to analyze where this discretization breaks.

The contents of this chapter in more detail:

1. **Section 4.1 gives an overview over binarization and quantization schemes known from literature on this topic.** In detail, this section will describe the work done on quantization and binarization schemes mentioned in Section 4.1 and give an overview over the related work on hardware optimized architectures.

2. **Section 4.2 presents a stochastic number system that enables the conversion of any neural network into a neural network that uses only additions of integers and bit shifts, which are conceptually much simpler operations compared to floating point multiplications.** The idea is to use accumulations of stochastic additions of bit-shifted integers to compute the weighted mean of this operation. The section will show that this approach achieves the same performance using a simpler number system as a drop-in replacement for floating point operations.

4.1 Lowering Computational Costs – Techniques & Related Work

With the advent of deep networks in practical applications, large amounts of work has been done towards simplifying deep networks: simplified evaluation using quantization, binarization and hardware-optimized architectures. A core challenge in applying deep learning to discrete spaces is that these spaces typically cannot handle fractions at a fine level and thus, lack fine-grained differentiability required for standard gradient-based optimization. This section summarizes the literature for each of these sub-topics, but before that the next paragraph gives a short introduction into the topic.

Straight-Through Estimator

A simple trick to circumvent the problem of searching in discrete spaces directly in order to retrieve a quantized network in the forward pass is to use the *straight-through estimator* [15]: Let $q : \mathbb{R} \to \{v_1, v_2, \ldots, v_n\} \subset \mathbb{R}$ be a quantization function, mapping real-valued numbers to quantized values from a set of predefined states v_i. To enable stochastic gradient descient (SGD) on a network involving such a quantization function requires to define the gradient of that method. The simplest such gradient is to pass the gradient through to the input,

$$\frac{\partial q}{\partial x}(x) := 1. \tag{4.1}$$

Several versions of this idea exist, for instance, clipping gradients if their magnitude r becomes too large to prevent the model to perform too big jumps in discretized parameter space during optimization,

$$\frac{\partial q}{\partial x}(x) := \begin{cases} 1, & r < 1 \\ 0 & \text{otherwise} \end{cases}. \tag{4.2}$$

Gumbel-Max-Trick

The straight-through estimator works only if the quantized values to choose from relate to the underlying un-quantized representation. A different technique is the Gumbel-Max-Trick [72] that allows also to optimize the probability distribution of random variables directly, which effectively enables gradients on arbitrary categorical sets.

The trick works as follows: Let $D \sim \text{Discrete}(\alpha_1, \ldots, \alpha_n), \alpha_i \in (0, \infty)$ be a discrete random variable sampled according to their weights α_i, that is

$$P(D = k) = \frac{\alpha_k}{\sum_{i=1}^{n} \alpha_i}.$$

By reformulating the sampling of D with the probability distribution $P(D = k)$, it becomes possible to decouple randomness from parameters, enabling gradients to be computed directly at the weight-level,

$$D = \arg\max_k \left(\log \alpha_k - \log(-log U_k)\right), \quad \text{where} \quad U_i \sim \text{Uniform}(0, 1) \text{ i.i.d.}.$$

Quantization

By reducing the number of bits per operand, and avoiding expensive floating-point units by utilizing number formats with fixed scaling, computational costs can be brought down considerably [32, 55, 194]. However, direct quantization of full-precision pre-trained models lead to significant loss of accuracy in deep networks. As a result, most techniques today for large language models or diffusion models apply quantization during training or as fine-tuning of pre-trained full-precision models [99, 110].

One approach increases the significance of low-precision weights by rescaling intermediate results globally before and after the weight-multiplication adaptively [32]. Alternatively, it is possible to determine optimal global scaling factors for pre-trained models by estimating the statistics of incoming data [103].

A second idea uses an iterative discretize-and-retrain technique. Combining this approach and pruning results in a technique that produces networks with a significantly smaller memory footprint [58]. Similar in spirit one can also constrain the filters to shift-operations that are much easier to implement on hardware [193].

The most similar idea to the stochastic number system defined in Section 4.2 uses the local re-parametrization trick to include stochastic components in a network without losing the ability to estimate gradients [83]. For instance, this technique enables integrating a continuous relaxation of discrete random variables into deep networks; stress on weights is increased gradually during full training or fine-tuning to encourage weights to align with a given discrete grid [112]. Closely related is also the idea to sample normal distributed pre-activations during training instead of discrete random weights [154, 133].[1]

[1] The sum of i.i.d random variables tends towards a normal distribution due to the central limit theorem.

Binarization

The limit case of quantization is binarization, where either weights [140] or even weights and activations [71] are constrained to one of two possible values. This reduces costs dramatically, eliminating multiplications (binary weights) or even simplifying all arithmetic operations to logical operations (full binarization) at the cost of considerable decrease of accuracy [6].
Many variants use the *straight-through estimator* [15] discussed earlier that enable gradient descent optimization on discretized values. Fully binarized network architectures with binarized training processes exist [31, 71], however implemented naively show a significant performance gap compared to full-precision networks [197].
Further work concentrates on different number representations. As in quantization, global real-valued scaling factors of binary weights can be used to represent numbers with greater variance [140]. Another approach learns number representations with binary coefficients by means of a sum of multiple globally scaled binary weights [104]. It has been shown that *ResNets* are well suited for purely binary weights, if implemented with full-precision residual connections [106]. More recent hardware support the `bfloat16` number format that increases the dynamic range with more bits used for the exponent at the cost of fewer bits used for the mantissa, leading to a more optimized number format for deep learning applications [78].

Hardware Optimized Architectures

The application to optimize networks for the use on embedded or mobile systems has led to optimized architectures that allow for a trade off between latency and accuracy [67, 149]. Or, to go into low-level optimizations; discussions about better numerical representations exist [63].
Ternary quantization[2] [195], two-bit quantization [116] and integer only multiplications [74][3] are related interpretations of the restrictions of binarization that also allow for better implementation on hardware.
Concrete implementations prove that networks can indeed be implemented on hardware for faster inference [88, 117]. And, publications that discuss binary neural networks on *FPGAs* [100], improved inference on CPUs using SIMD-Instructions [174] and stochastic binary neural networks for near-sensor computing [94] and speed-optimized networks such as the *Yolo family* [77] for fast object detection emphasize the interest for such techniques.

[2]Ternary weights are usually constrained to the values -1, 0, or 1.
[3]The authors use 8-bits weights, 32bit biases and fixed-point numbers for intermediate results.

4.2 Progressive Stochastic Binarization of Deep Networks

This section is based on the peer-reviewed article:
Progressive Stochastic Binarization of Deep Networks
David Hartmann, Michael Wand
2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing - NeurIPS Edition (EMC2-NIPS).
doi: 10.1109/EMC2-NIPS53020.2019.00015

The last section has discussed techniques used in literature to enable to train networks with increasingly smaller number systems, while still keeping information and gradients intact. The goal of this section is to find a way to have both: small number systems while having the option to refine results when needed.
In order to do that, this section will discuss stochastic multiplications as the base operation in deep networks, convert networks that use the new operations, or, train these networks from scratch and discuss ways to refine results. Because of the equivalence in the limit of the introduced stochastic multiplication method, called *progressive stochastic binarization (PSB)*, and floating point multiplications, this results gives us an intuition of the stability that the information flow in deep networks have at a varying degree of discretization.

Stochastic multiplications
A multiplication $w \cdot x$ of a number $w \in [0, 1]$ and a number $x \in \mathbb{R}$ can be estimated stochastically by $B_w \cdot x$ where $B_w \in \{0, 1\}$ is a Bernoulli random variable with probability $\Pr(B_w = 1) = w$, and can be written as

$$\mathbb{E}[B_w \cdot x] = w \cdot x.$$

Thus, we substitute all multiplications $w \cdot x$ in the network's scalar products,

$$w \mapsto B_w \tag{4.3}$$

and obtain a statistical approximation of the filter results.

Value-based importance Sampling
A huge drawback of networks consisting of only binary weights (or weights between 0 and 1) is the limited range of possible values. A typical observation when applying this technique in practice is that the probabilities gather around the borders 0 and 1 [37]. This reduces the amount of obtainable information of each gradient descent step, as gradients that point outside the weight-domain $[0, 1]$ are discarded (various

other techniques exist, that try to reduce that problem [104, 186, 37].
Instead, the strategy is to use an encoding that allows to represent any filter in a given domain. We can therefore split each weight into an exponent $e \in \mathbb{Z}$, a sign $s \in \{-1, 1\}$ and a mantissa $p \in [0, 1]$ and reformulate any weight w as

$$w \mapsto \overline{w} := s \cdot 2^e \cdot (B_p + 1), \tag{4.4}$$

where

$$s := \text{sign}\,(w)\,, \tag{4.5}$$
$$e := \lfloor \log_2 |w| \rfloor \text{ and} \tag{4.6}$$
$$p := \frac{|w|}{2^e} - 1. \tag{4.7}$$

Building on top of work of others [74], who have already shown that intermediate results can be converted to fixed point integers, the idea of value-based importance sampling enables to restrict the whole network to work only with bit-shift operations of fixed-point integers as intermediate results. Technically, one would choose stochastically the bit-shift $\cdot 2^{e+1}$ with probability p and $\cdot 2^e$ with probability $1 - p$ (See Figure 4.1 (a) and (b)). By construction, the mean of the representation equals to w,

$$\mathbb{E}[\overline{w}] = s \cdot 2^e \cdot \left(\frac{|w|}{2^e} - 1 + 1 \right) = s \cdot |w| = w.$$

Capacitors

The multiplication scheme shown above (Equation (4.4)) has a high variance (See Section 4.2.1).
To reduce the variance, one could draw multiple network samples and average the outcome of every sample. However, deep networks are non-linear[4]; therefore, the statistical estimates obtained from plugging Bernoulli samples into Equation (4.4) are not consistent for multi-layer networks that include non-linearities. (A later experiment in Section 4.2.5 will show that in some cases the technique can be used anyways). Especially gradients w.r.t. the parameters of a probabilistic variable are hard to estimate [112]. The training process can be augmented to account for this [47], or to reduce the variance of the gradients significantly [186], but even then inference can suffer from high variance of intermediate results.[5]

[4] This is an important difference to traditional statistical well-known integration approaches, such as Monte-Carlo light-transport in computer graphics [29], where end-results can be averaged due to linearity

[5] This is intuitively clear: binary samples at the input of a highly-nonlinear computational network lead to strongly varying outputs

The solution is to statistically average layer-wise, over multiple linear combinations *before* applying the non-linearity. This step is in the following denoted as the *capacitor* step, in allusion to the analog component. Implementation costs are not significantly impacted, assuming that computations can be ordered accordingly, which is the case for wide networks operating on, for example, many pixels, independent samples or activations in parallel.

Putting it all together, Equation (4.4) includes multiple samples as follows:

$$w \mapsto \overline{w}_n := s \cdot 2^e \cdot (B_{n,p}/n + 1), \tag{4.8}$$

where n denotes the sample size, and $B_{n,p}$ the n-fold binomial distribution with probability p.
In practice, one would continue to sample from Bernoulli distributions, for example by rewriting the multiplication to hardware-optimized operations, with sample sizes that are powers of two, $n' = 2^n$

$$x \cdot \overline{w}_{2^n} = s \cdot \left(\sum_{i=1}^{2^n} x \ll (e + B_p) \right) \gg n, \tag{4.9}$$

where $\ll k$ denotes the left shift $\cdot 2^k$ and $\gg k$ denotes the right shift operation $\cdot / 2^k$.

4.2.1 Properties of the Accumulated Samples

The single-sampled case $\overline{w}$ is first to consider. Even though, the mean of $\overline{w}$ equals to w, the value varies significantly for large values of w (Figure 4.1 (c)),

$$Var(\overline{w}) = 3 \cdot 2^e \cdot |w| - 2^{2e+1} - w^2 \leq \frac{w^2}{8}.$$

With multiple samples n, the bound decreases anti-proportionally,

$$Var(\overline{w}_n) \leq \frac{w^2}{n \cdot 8}. \tag{4.10}$$

Thus, sampling from this number-system leads to high variances between two shifts, i.e. it is locally maximal whenever $p = 1/2$. For instance, the representation for $w = 3$ is $e = 1$ and $p = 1/2$. Weights in these regions have high variances (see upper bound above). However, the experiments in Section 4.2.2 and Section 4.2.3 using this number system show, that this does not turn out to be a problem in practice.

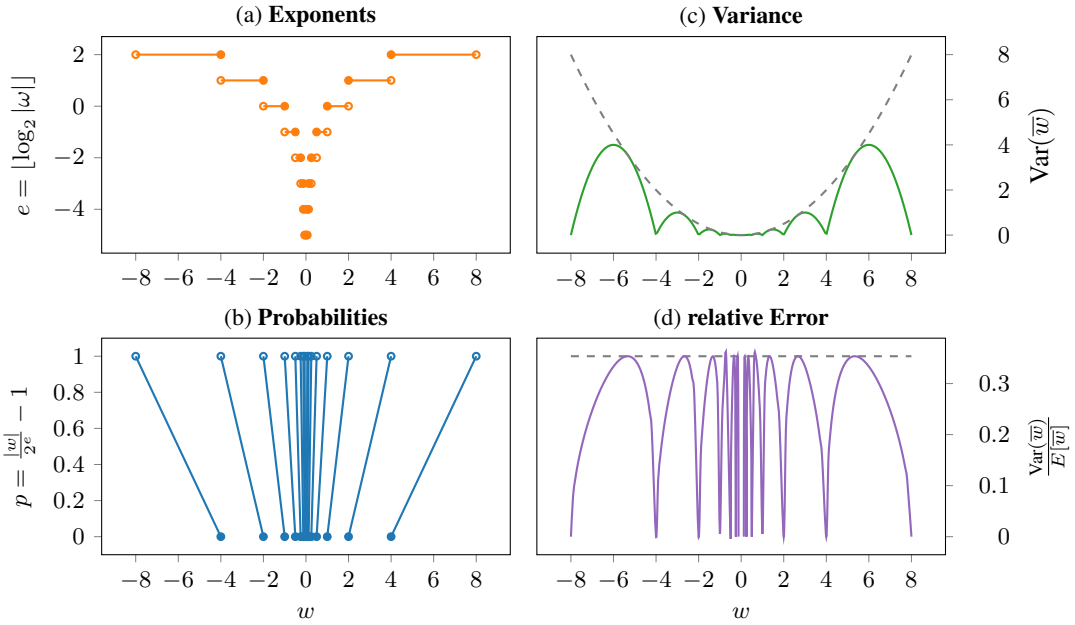


Fig. 4.1.: Figure (a) and (b) visualize given an arbitrary target value (x-axis) the values (y-axis) of the exponent and of the probability of the stochastic multiplication. Figure (b) and (c) show the variance and the relative error. In practice, the values near 0 are only hypothetical; too many shifts of integers result always in the number 0.

More interestingly, it follows that the *relative variance* of the number system has an upper bound (Figure 4.1 (d)):

$$\frac{\sigma_{\overline{w}_s}}{|\mathbb{E}[\overline{w}_s]|} \leq \frac{1}{\sqrt{s \cdot 8}}. \tag{4.11}$$

Note that some cases, like $p = 1/2$, or more generally $p = m \cdot 2^{-k}, m = 0 \dots k$ can be computed deterministically and exact, with a similar approach as Equation (4.4): instead of sampling, add the surrounding bit-shifted versions of the multiplications and divide by the total number of "deterministically chosen samples". For instance, multiplying with $w = 3$ can be accomplished by adding the products with 2 and 4 and dividing the result by 2. To check the effect of sampling, these special cases (despite being null-sets in the space of real-valued numbers anyways) are not considered in the following.

To evaluate accuracy-vs-sampling trade-offs in practice, simulated capacitors as defined in Equation (4.8) are implemented instead, sampling the corresponding filters directly as Binomial distributions. Intermediate results are quantized to 16-bit integers.

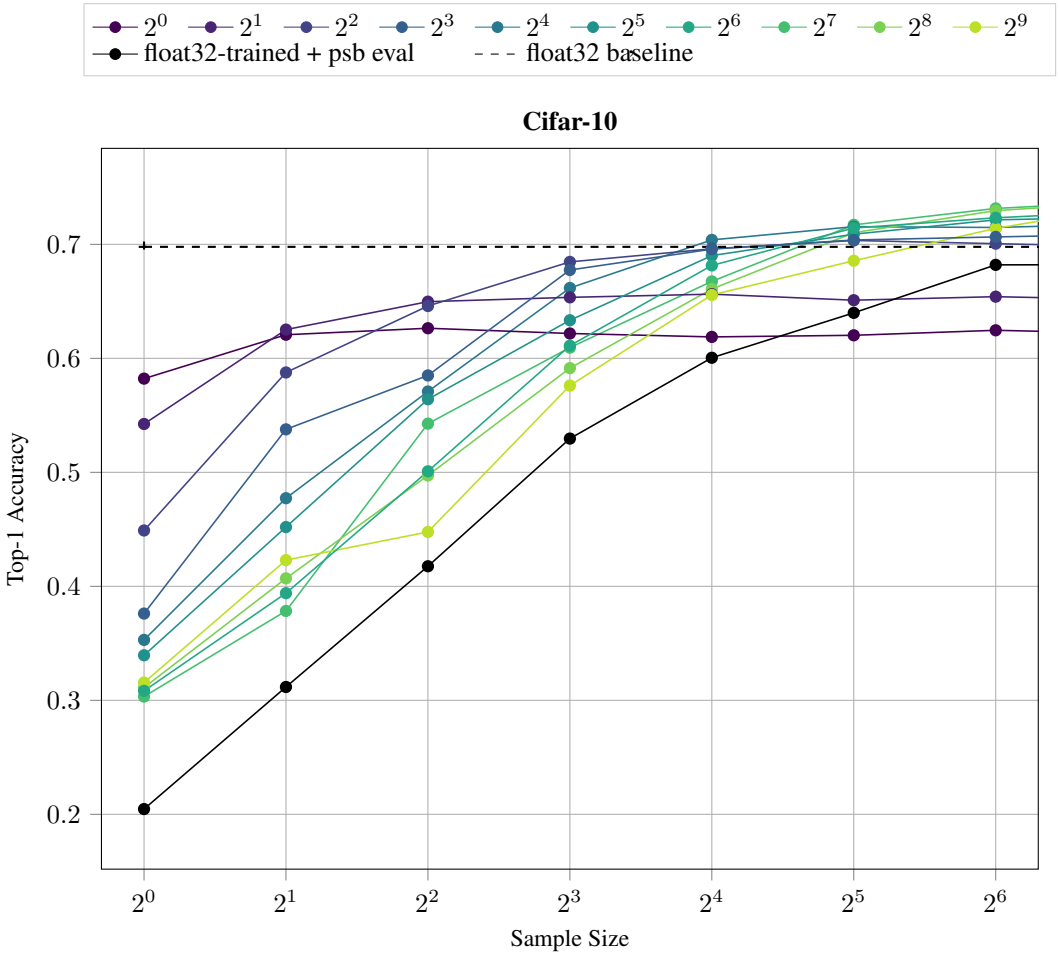


Fig. 4.2.: Training PSB from scratch on CIFAR-10, with training specific to various sample sizes, evaluated at different sampling levels. Dashed black line: floating-point accuracy; solid black line: In-Place-PSB (no retraining).

4.2.2 Training PSB from Scratch vs In-Line PSB

As a first test of the number system defined in Equation (4.8) serves a small convolutional network trained on CIFAR-10 (eight layers of 3×3 convolutions followed by batch normalization and Rectified Linear Unit (ReLU) activation). The idea is to test Equation (4.4) as a drop-in replacement for all weight multiplications: First, by comparing the quality of a full precision `float32`-trained network with the replaced stochastic multiplication for various sampling sizes. This serves as a baseline to check how much quality is potentially lost compared to a network that uses the stochastic multiplication also during training – effectively learning to mitigate the variance of the stochastic multiplications.
Figure 4.2 compares the results of the model trained with floating point arithmetic to the same model trained with Equation (4.4).

The full-precision trained network looses significant performance when replacing the multiplications directly: the performances drops to about 20% with a single sample, about 30% for two samples, about 40% for four samples, about 55% for eight samples, and 32 samples result to about 90% of the performance of the original `float32` model.

However, when trained directly with the number system given in Equation (4.4), the performance increases, even at lower sample sizes. First, for a given sample size ranging from 2^1 to 2^5 networks are trained. After that, as with the `float32` model, the experiment checks how much the performance changes if a different sample size is used compared to the one used for training itself. Results show higher accuracies when trained directly with PSB, compared to converting `float32`-weights. This is in accordance with other quantization schemes. As expected, for a given fixed evaluation sample size, it is also the network that has been trained with that sample size that is best. (For instance, the best model of the family that can be evaluated with only $2^0 = 1$ sample is also the model that used only a single sample for training). However, the contrary is not true in this experiment: the model trained with $2^1 = 2$ samples improves quality-wise if evaluated with more than two samples. This indicates that while the forward pass is noisy, averaged over many gradient descent steps the backward pass into the real-valued probability p, exponent e and sign s still serves as a good approximation.

4.2.3 Models that Scale Well Stochastically

The previous experiment has shown how much performance is lost, when using a small `float32`-pretrained convolutional network using in-place PSB for evaluation. The experiment missed two aspects: first, how much of that performance gap is caused by the network architecture, and second, was that gap specific to the CIFAR-10 dataset. The next experiment will use on ILSVRC (ImageNet) pre-trained models and evaluate these without retraining using PSB as an in-place replacement multiplication scheme. Several image classification architectures are used; especially *Inception*- or *ResNet*-Types [60, 62, 169], which provide easily foldable batch normalization (BN) layers reducing stochasticity by merging batch normalization layers into the linear layers that come before it, and thus, resulting in fewer stochastic multiplications.

Figure 4.3 shows the named models, each evaluated using PSB using $1, 2, 4, 8, 16$ and 32 samples per multiplication. In most cases, PSB yields half of the accuracy of an un-binarized network with only four samples. With increasing sample size the results

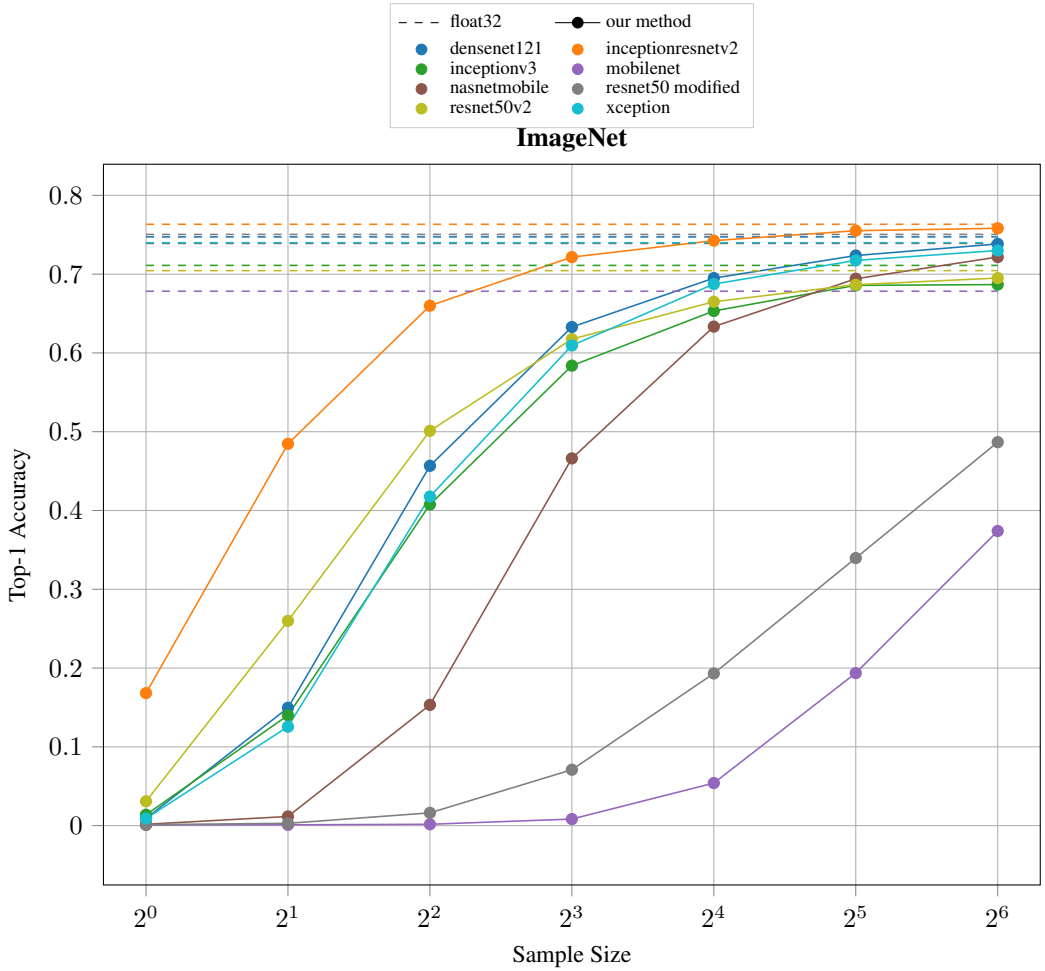


Fig. 4.3.: Pretrained ImageNet models after binarizing using In-Place-PSB with different sample sizes. Dashed lines: original floating-point performance. No retraining is used in any ImageNet test.

approach full precision. One of two exceptions is *MobileNet* [168] that uses separable convolutions with intermediate batch normalization layers. These problems are in line with previous work that combines MobileNet and quantization [156]. To emphasize the requirement of avoiding direct successive stochastic multiplications, a modified ResNet model, *ResNet-50 modified*, is used with BN layers *after* each residual connection (named "BN after addition" in the original ResNet paper [60]). Here, data in the residual connection is multiplied multiple times, leading to a large total variance and worse performance when used in conjunction with PSB (gray line). A small architectural modification (ResNetv2 variant, yellow line) avoids this issue directly.

A comparison with other quantization methods from literature applied to ResNet-18 [60] is listed in Table 4.1. However, when comparing these methods it is important

Tab. 4.1.: Classification Top 1-Accuracy for ResNet-18 trained on ImageNet for a selection of quantization and binarization methods.

Method	Fine-Tuning	Precision Weights, Activations	Accuracy **Top-1** [%]
LSQ [41]	✓	$4, 4$	70.9
DoReFa [194, 168]	✓	$4, 4$	68.1
INQ [193, 168]	✓	$2, \cdot$	66.6
BWN [140, 168]	✓	$1, \cdot$	60.8
XNOR-Net [140, 168]	✓	$1, 1$	51.2
ABC-Net [104]	✓	$5, 5$	65.0
ABC-Net [104]	✓	$1, 1$	42.0
Baseline		float32	69.7
PSB (ours)	×	2^6 samples	69.2
PSB (ours)	×	2^5 samples	68.2
PSB (ours)	×	2^4 samples	67.1
PSB (ours)	×	2^3 samples	63.0
PSB (ours)	×	2^2 samples	54.7

to note: The cited methods require retraining or training from scratch to achieve comparable results in a low-precision setting, whereas PSB does not require any fine-tuning of the pre-trained models, yet retraining improves the results further (see Section 4.2.2).

4.2.4 Weight Pruning & Discretization of Probabilities

The implementation of the idea so far is not yet ready for actual use on efficient hardware. A crucial step is to enable storage of the probabilities themselves as quantized values as well, the other step would be to reduce the number of multiplications in the network overall.

Starting with the reduction of multiplications overall, one can use pruning to disable multiplications with small numbers. Again, the focus is on the ResNet-18 network, evaluated first with `float32` and with PSB of 2^4 samples to get the baseline numbers without pruning enabled and shown in Table 4.2. Thus, as a first modification the table shows the accuracy with pruning enabled, first by removing 25% and then even 50% of all the weights closest to zero. As shown in Table 4.2, pruning of 25% has only moderate influence.

Next modification is to also reduce the memory footprint of weights by quantizing the probabilities of the numbersystem in Table 4.2 regularly in its domain, $p \in [0, 1)$,

Tab. 4.2.: Classification error for a `float32`-pretrained ResNet-18. Note: `psb`k refers to PSB with 2^k-fold sampling, `psb`j/k refers to PSB with 2^j-fold base sampling and 2^{k-j} fold additional refinement.

Experiment		**Precision**	**Top-1 Accuracy** [%]
Baseline		float32	69.7
		psb5	68.2
		psb4	67.1
		psb2	54.7
+ Pruning	25%	float32	69.0
		psb4	65.8
	50%	float32	41.5
		psb4	35.3
+ Discrete p-values	4-bit	psb4	66.7
	2-bit	psb4	62.7
	1-bit	psb4	31.3
+ Computational Attention	Random 37%	psb1/5	44.7
	Entropy-based Selection	**psb1/5**	**57.1**
	Random 76%	psb2/5	65.7
	Entropy-based Selection + Margin	**psb2/5**	**67.7**
= Combined		**psb1/5**	**57.4**
		psb2/5	**67.8**

to 4, 2 and 1 bits.[6] The accuracy drops slightly with discretized probabilities in a stochastic setting. A significant drop exists for the discrete case of 1-bit probabilities. Using 16-bits fixed point numbers for intermediate results in the rest of the network, 4-bit exponents and 4-bit probabilities should suffice most cases of the PSB number scheme.

4.2.5 Computational Attention

Replacing floating point multiplication by sampling allows (at least in theory) for a fine-granular control of the precision of a model at the cost of additional sampling costs. However, as computations in one layer in neural networks depend on the results of earlier layers, this control has to be specified beforehand.
This section explores this, by testing if this control can be deferred to a later stage. The idea is to evaluate a network in a low-precision mode first and use a heuristic

[6] More elaborate techniques could be used here. For instance, if re-training is used a straight-through estimator could be used to quantize p itself, enabling gradients to optimize the quantized version of p instead of p itself.

on the low-precision, high-variance results to determine which parts of a network may require higher precision intermediate results.

A network type that is particularly well-suited for this idea are convolutional neural networks, as their receptive field for output-pixels is easier to calculate and detached from values not in that receptive field. As a baseline, this section uses the ResNet-18 model pre-trained on ImageNet. It has worked reasonably well regarding scalability using sampling and because of it's final average pooling layer its receptive field is easier to keep track of. Table 4.2 shows that due to the bound approximation error of PSB relation to the default floating point mode, 2^5 samples already achieves a similar testing performance. The goal is now to optimize the overall computational attention by evaluating the network with varying sample precisions in the image plane.

Algorithm Description

First, interesting regions are obtained by using the network in a low-precision mode a the full image, as shown in Figure 4.4(a), and to refine the results on these interesting regions with more samples. In detail, to reduce costs substantially, an initial low-precision mode forward pass is used to determine a rough estimate of the classification. The next step is to choose which regions to refine. While this could be also solved by using the variance of the output in theory, using only 1 or 2 samples in practice is however not enough to compute precise estimates of such statistics. Another way would be to rely on the probabilities p used for the computations, as many $p = 1/2$ probabilities would increase the variance of interim results. However, such a mechanism would have to take the exponents of the multiplications and the data distribution of the previous layers into account – a probability of $1/2$ may result in a maximal variance given all other parts are fixed, but a small exponent or a small input could negate this effect again. Thus, instead of relying on the statistics internally, or the multiplications themselves, the heuristics that works well in practice to determine the part of the image where results are "uncertain" by the original neural network already uses the per-pixel class logits *before* the average pooling layer. More specifically, we estimate the estimated per-pixel class entropy, h_{xy}, of the image in the last convolutional layer, defined by

$$h_{xy} \approx -\sum_c \frac{e^{a_{xyc}}}{\sum_j e^{a_{xyc}}} \cdot \log\left(\frac{e^{a_{xyc}}}{\sum_j e^{a_{xyc}}}\right), \tag{4.12}$$

where a_{xyc} denotes the activation of the last layer in the pixel (x, y) and channel c. It measures the level of uncertainty in the class prediction – the pixels with a

high uncertainty are thus exactly the results that need a higher precision to prevent wrong classifications.
While this uncertainty measure is a more expensive computation, it only needs to be evaluated on the last layer of the network, having only a coarse resolution. Using the per-pixel entropy, a global threshold T then determines the regions of highest uncertainty, $h_{xy} > T$, to estimate the interesting regions. As the value of the threshold implicitly determines the proportion of re-evaluated parts of the image, it also specifies the trade-off between precision and computational costs. Figure 4.4(a) shows two applications of the computational attention mechanism applied to the ResNet-18 model, one using a threshold of a mean image uncertainty, $T = \text{mean}_{xy}\, h_{xy}$, (middle image) and one that increases the regions of interest by a margin of one pixel to ensure that interesting regions are covered completely (right image).

The reason why this methods works well becomes clear when looking at the errors of the network using PSB compared to it's `float32` version. Figure 4.4(b) and (c) show the absolute, $|x_{\texttt{psb5}} - x_{\texttt{float32}}|$ and relative errors, $\left|\frac{x_{\texttt{psb5}} - x_{\texttt{float32}}}{x_{\texttt{float32}}}\right|$ of layers 0, 8 and 16 of an on ImageNet pre-trained ResNet-18, evaluated using PSB with 2^5 samples. While the highest absolute errors happen in uniform colored or over-illuminated areas of the image for lower layers, absolute errors in the last layers rather accumulate in the areas of class-specific features, such as the tail of the cat or it's face in the used example image shown in Figure 4.4(a). In contrast, relative errors appear mostly unrelated to the content of the image. The critical point to note is that the most noticeable errors in lower layers do not appear to significantly impact later layers' calculations, ensuring the overall prediction remains stable. This strongly suggests that the computational attention mechanism is effective in practical applications.

Computational Attention on ImageNet

The following section evaluates the computational attention mechanism on the ImageNet dataset. However, it is important to note that the benefits of the mechanism on ImageNet show only a lower bound of the potential benefits of the mechanism. This is due to the limited picture size of the ImageNet data points, as the images in the dataset only show the object of interest in the center of the image, with only less proportion of backgrounds.
Two selection approaches are used to determine the interesting regions for the image classification task: first, using the per-pixel entropy of the class logits as described above, and second, with an additional margin of one pixel around the selected

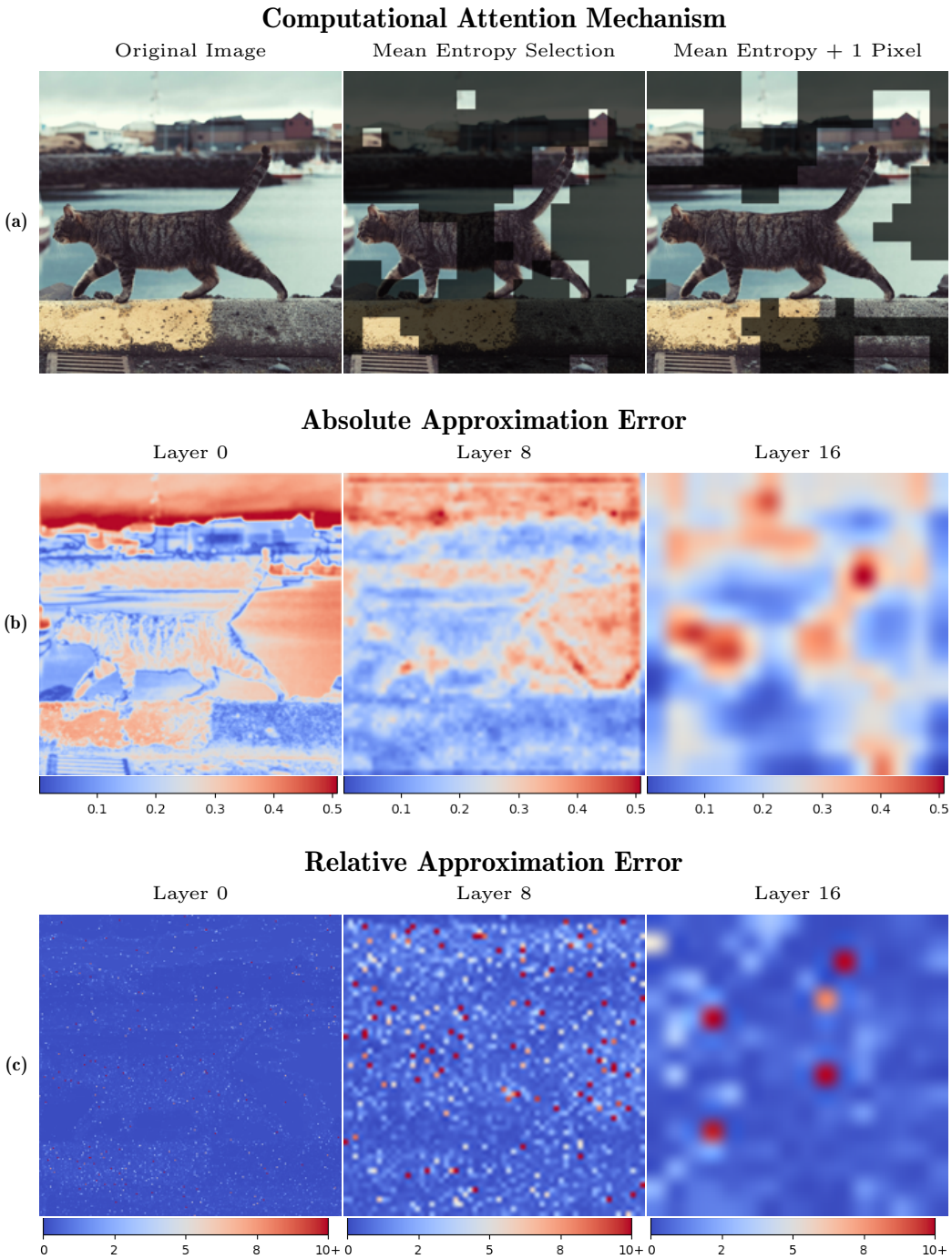


Fig. 4.4.: Visualization of computational attention using ResNet-18. (a) From left to right: the Original image (CC0-like Unsplash License, source: `https://unsplash.com/de/fotos/getigerte-katze-auf-felsvorsprung-AL2-t0GrSko`), interesting parts according to the computational attention mechanism, and an additional one-pixel margin around the computational attention selection. (b) Absolute Errors of PSB with 2^5 samples on Layers 0, 8 and 16 compared to `float32`. (c) Relative Errors of PSB with 2^5 samples on Layers 0, 8 and 16 compared to `float32`. Errors are averaged over 100 independent repetitions of `psb5` inference. While highest absolute errors happen in uniform colored areas in lower layers, errors in last layers accumulate most at the class-specific features (here the tail of the cat and it's face). Relative errors on the other hand seem mostly unrelated to the content of the image.

region of interest. The latter version results in larger portions of the receptive field of the selected pixels to be included for refinement. In the case of ImageNet, the threshold of $T = \text{mean}_{xy}\, h_{xy}$ covers about 34% of a random image, while the additional margin of one pixel around the interesting regions results in about 76% of the images being selected to be refined. These computed regions of interest are then used as scaled masks to enhance the precision of all filters individually in a second forward pass by identifying the receptive fields of interesting regions throughout the network.

The result shown in Table 4.2, "+ attention") chooses between different choices for high- and low-precision mode of stochastic inference for both selection variants ("entropy" and "entropy + margin"): Compared are two choices for a base sample size (1 sample and 2 samples) and fixed the number of refine-ment samples of 2^5 samples, thus the names **psb1/5** and **psb2/5**. For comparison with a baseline, the results for a random selection of masked areas with the same proportion of interesting to non-interesting proportions is also shown in Table 4.2. The result indicate that the uncertainty heuristic using the per-pixel entropy of class logits correctly helps to identify the parts of the image that need a more precise evaluation.

Of course, the choice of a specific base and refinement sample size represents a trade-off between accuracy and computational costs.

Figure 4.5 visualizes this trade-off using the same heuristics as above; the mean entropy threshold with a coverage of 34% and a second version with an additional margin with a coverage of 76%. By evaluating all pair-wise combinations of base sampling sizes and refinement sampling sizes, and, how they would affect accuracy and computational cost savings, the table shows both, the loss of accuracy and the reduction of computational costs for a given choice of base sample size and refinement sample size. Again, the baseline of randomly selecting interesting regions with the same coverage is shown for comparison. The results show that the random selection fails to identify the interesting regions resulting in poorer performance at the same computational costs compared to the entropy based selection, especially for the low base sample size regime – where reduction of computational expenses become most evident.

4.2.6 Discussion of Progressive Stochastic Binarization

This section has presented a novel number system in the context of evaluating deep networks. By using stochastic additions any pre-trained network with `float32` weights can be converted into a stochastic binarized version with tunable precision at runtime. A heuristic can be applied in addition to optimize computational costs

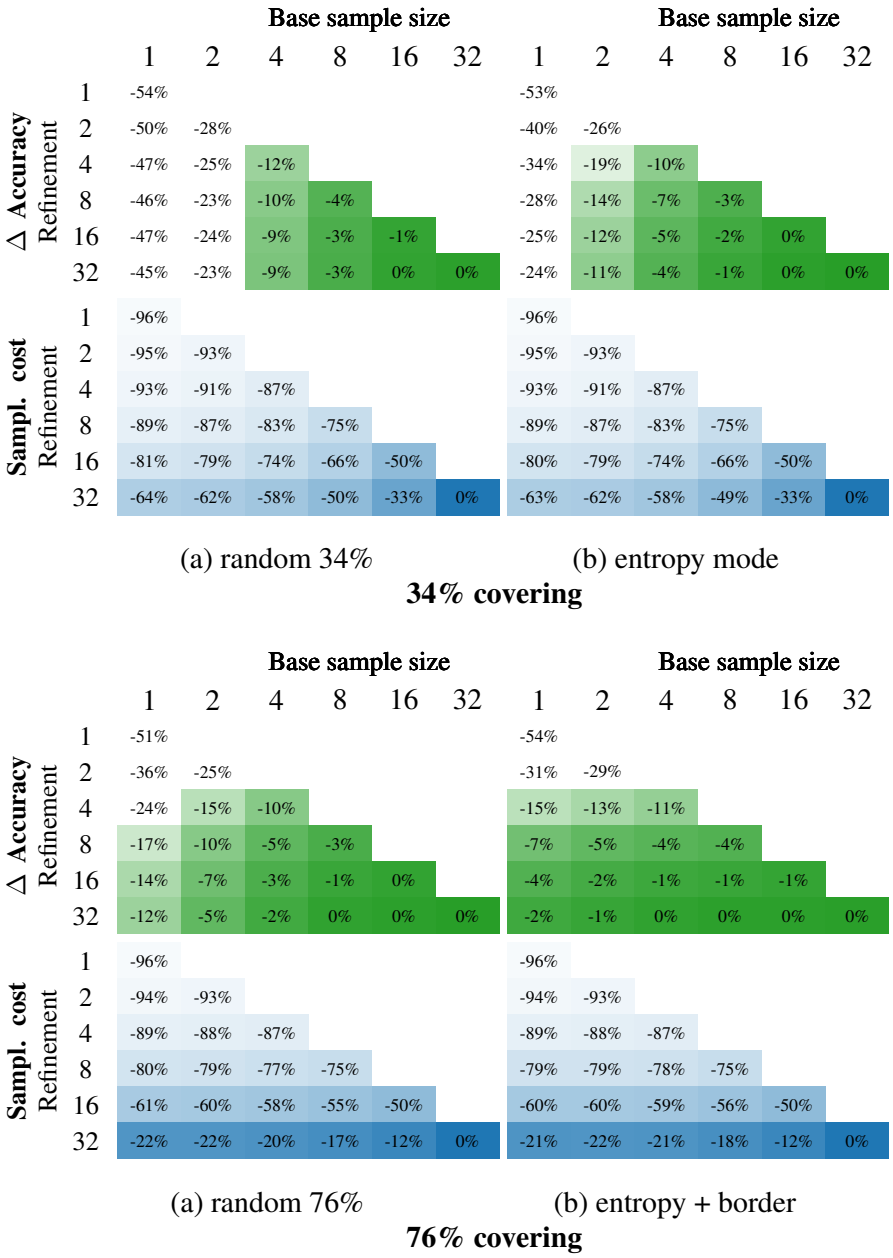


Fig. 4.5.: Reduced expenses when using computational attention — Shown are the difference in accuracy (upper row) and the reduced sampling costs (lower row) for a given choice of base sample size for a first rough estimate using ResNet-18 evaluated on ImageNet. Interesting regions found using these rough estimates are refined with the help of a selection mechanism: here the discussed entropy-based selection mechanism and a baseline of random selection. The upper table shows the results for these two selection mechanisms that both cover about 34% of the test set images (random selection on the left, entropy based selection on the right). The lower table shows results for the same mechanisms with an increased coverage of about 76% of interesting regions in the images (random selection on the left, entropy based selection with margin on the right).

by estimating which sections of a convolutional neural network trained for image classification might require calculation of higher precision.

Progressive Stochastic Binarization (PSB) In Context of Related Work on Network Quantization

The method is an in-place quantization approach, i.e., additional fine-tuning of the model is not required as all weights can be transformed bijectively into their new represenation. Compared to other quantization methods, the quality can be chosen at run-time.
The method suggests stochastic binarization that has the potential of closing the performance gap of binary neural networks at the cost of re-sampling. In contrast to other binarization techniques that require further hyperparameter tuning or add even more hyperparameters [17], this method changes the number representations bijectively and thus, same hyperparameters apply. Noteworthy is that the method reevaluates stochastic weights to even out the randomness, while other methods use a single pass during inference. However, the experiments show that only few accumulations are sufficient to achieve similar performances like previous methods. In practice, accumulations can be rolled out and computed in parallel.
The presented method uses only shift-operations on small integer numbers. For each such shift-instruction one random bit decides which of two shift-operations is used for accumulation. Although promising for hardware-implementations, it is yet to be shown if hardware architectures have a substantial benefit from these rather simple instructions compared to usual floating point multiplications.
Other work has shown that controlled noise injection into gradient estimates can benefit network performance [127, 127]. And, additional noise in each layer has become a widely used technique for Generative Adversarial Networks [20]. Such tasks could thus potentially benefit from a number representation that already incorporates stochasticity. For instance, future work could implement the gradient computation itself using a similar stochastic binarization technique.

On the Relation to Alternative Network Design

A strongly related topic to PSB is stochastic computation (SC) [48, 5]. Here, numbers are approximated by sequences of binary samples whose average correspond to the intended number. This idea has recently been applied to hardware implementations of deep networks [81, 142, 7]. Similarly, stochastic quantization (see above) has also been proven to be a valuable tool to reduce the negative impact of reduced

representational efforts.
Closely related to SC is the idea of Sum-Product-Networks (SPN), a familiy of probabilistic graphical models that encode joint distributions of their input random variables [135]. SPNs are equivalent to arithmetic circuits and thus, are also promising candidates for computational efficient networks. They have been used successfully for various typical deep learning tasks, such as natural language processing [24], image classification [153] and image segmentation [141].
The biggest difference of PSB to SC and SPNs is the interpretation of data-streams and intermediate results. SC and SPNs interprets data mostly as probabilities that produce random data streams. PSB in contrast, uses fixed-point numbers for incoming data and intermediate results, while only weights are random variables. This reduces the variance of intermediate results in practice (see Section 4.2.3).

Conclusion

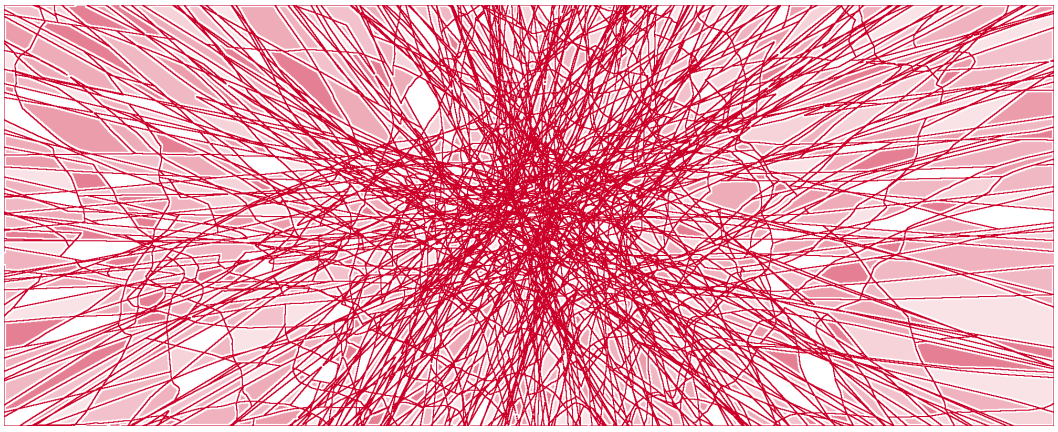
In practice, this type of computation can only become interesting, if specialized hardware existed that could take advantage of stochasticity, probable capable of distributing sampling based on the expected variances of the number system Figure 4.1.[7] Additionally, for the computational savings to be effective, the ratio of interesting classes in an image to the size of the image has to be small. This technique, thus, might be effective for applications involving panoramic or 360-degree imagery. This is particularly relevant in the context of self-driving cars, where such applications often have a high demand for speed and power savings, thus requiring a binary neural network. Future work would, however, first need to identify how well the low-precision mode actually is to determine all interesting parts in a scene.

In the context of this work (disregarding practical feasibility), this technique shows that binarization is not a theoretical problem per se – by incorporating sampling information can flow through the network at varying sample sizes. The surprising part is that the number of activation cells (the subdomains that are mapped by the network using the same affine transformation globally) rise exponentially with the number of filters. In other words, they become smaller when a model uses more filters, and thus, should be more susceptible to noise. However, as shown in this section, mid- or lower-sampling modes don't interfere with the overall result of a network, especially when focusing "computationally" on the "important" part of the input data.

[7] As a side-note, the simulated stochastic multiplications used in the experiments have not used true random numbers, but only a RNG using a mersenne-twister algorithm. Thus, precomputed random numbers lists or RNG would be viable choices if this were implemented on specialized hardware.

Conclusion

5



> *There once was a thesis so keen,*
> *With ReLUs that sliced through the scene,*
> *APT took the measure,*
> *Discretely, with pleasure,*
> *Revealing what training might mean.*
>
> **— ChatGPT,**
> **"Generate a limerick of the following text:"**

This thesis set out to explore how information propagates from raw data through the training process into the model parameters. The goal was to examine the internal mechanisms that govern learning flow in deep networks, with a specific focus on the ReLU activation function which was widely adopted as a successful component in major achievements in the history of deep learning research. A remarkable property of the ReLU activation is that the employed continuous optimization of weights using stochastic gradient descient (SGD) does not "see" the activation boundaries. In this sense, discrete optimization happens implicitly, serving as the underlying theme of this work. In this regard, the question is how the implicit discrete aspect of training can be observed, and whether the same discrete aspect also informs the continuous part of optimization.

5.1 Summary

Activation patterns (APs) are a direct way to measure which of the two decisions – pass-through or stopping – a ReLU activation has made for a given input and thus already provides some insight into the activation's role in the process. The subsets in input space that result in the same APs are referred to as activation cells (ACs), and offer a way to visualize and analyze the internal structures of the network, revealing how hyperparameters of an architecture like width (number of filters), depth (number of layers), bias (offset of filters) and architectural features influence these structures.

Network training is still subject of research as the training dynamics are not yet fully understood. Research discussed multiple training regimes, with the network transitioning from initially learning the broad structure of the data to refining its weights in a more linear fashion as it converges. By efficiently recording statistics of APs during training, this thesis presents new techniques that enable to view the training process through a unifying lens. In an extensive study of the evolution of APs during training, this thesis offers insights into how quickly different layers converge, how architectural choices or optimization methods impact training stability, and how expressivity of networks evolve over training time.

As a result, the experiments revealed the following characteristics in the tested comparisons of architectures and training schemes: (1) the initial set of APs is mostly replaced within the first epoch of training, but features such as residual connections enable to retain more information from the initialization throughout training compared to networks without this architectural feature (2) most architectures converge bottom-up, with the exception of highly-tuned architectures where layers converge more simultaneously, (3) during training, the expressivity changes depending on the architecture, but the expressivity of a network in the initialization state generally increases as network depth increases, but often drops soon after that – again and in contrast, residual networks maintain higher expressivity throughout training, (4) expressivity of non-residual networks can recover slowly with further training, (5) learning rate schedules, such as One Cycle learning rate schedules, adapt the learning rate over the course of training to fade linearly between exploration and exploitation of non-linear structures.

Tangentially related, and probably more important, is the introduced concept of learning rate sensitivity analyses that connect the learning rate with the proportion of changes in APs. The experiments revealed a soft phase transition observed across various architectures and schemes, enabling automatic learning rate scheduling. In exploring the connection between discrete and continuous aspects of optimiza-

tion and by implementing a stochastic number system that can dynamically switch back and forth between continuous and discrete calculations, the thesis finishes with a discussion of how shifting linearity even more towards non-linearity impacts performance.

5.2 Contributions

The primary focus of this thesis is to efficiently count the activation patterns (APs) during training and to extract meaningful information about the network using these measurements. The three most notable contributions are probably the *Activation pattern entropy (APE)*, the *Activation Pattern Temperature (APT)* and the probabilistic model connected to the APT.
While the former is a measure of expressivity, the network's potential in terms of how many affine transformation it represents, the latter is a measure of non-linear distance. Akin to the learning rate, a critical hyperparameter that controls the step size of the weight updates during training, the APT captures the functional change in a network layer, focusing solely on the non-linear aspects of optimization. In detail, the APT represents the proportion of changed APs in a network layer for a single gradient descent step using the *same* batch of data.
Observing the APT for a range of learning rates (the technique has been referred to as learning rate sensitivity analysis in this thesis) leads to the third major contribution of this thesis: the development of a probabilistic Gaussian transition model to estimate the relationship between learning rates and AP changes. By explicitly modeling the distribution of AP changes, this thesis unifies previously observed aspects of training phases in a single model, which fills the gap in the literature regarding the transition period between the initial, preferably non-linear phase and the final, preferably linear phase of training. The sensitivity analysis reflects how architectural features (such as batch normalization and residual connections), optimization features (such as weight decay and momentum) and learning rate scheduling affect training, thus modeling the entire training process in a single unified framework. This thesis introduces ActCooLR, a proof-of-concept learning rate scheduler, based on the presented transition model that adjusts the learning rate based on the network's sensitivity automatically. The performance of ActCooLR matches the performance of hand-tuned schedules like the OneCycleLR learning rate schedule, thus validating the presented findings.
These combined topics – from learning rate scheduling and effects of architectural choices to the theoretical modeling and empirical evaluation of models and training

behavior towards better understanding and improving the training process of deep networks – highlight the importance of AP analysis in general and the contributions of this thesis in particular.

5.3 Limitations

This thesis provides a theoretical model for statistical dynamics during training; however its findings do not fully explain how the optimal temperature curve of APT looks like. The transition model has demonstrated its predictive capability through experiments and by constructing a proof-of-concept learning rate scheduler. However, the model does not align perfectly with the observed measurements of APT, which required to implement mechanisms in ActCooLR to account for this mismatch. The derivation of the model has shown where these differences might stem from, indicating that further efforts should be directed towards creating a more suitable transition model of the APT. Furthermore, the analysis primary focuses only on certain network types (residual networks, convolutional neural networks, and feed-forward networks) for image and audio classification. To enhance the scope, the study should incorporate alternative network types such as transformers or recurrent neural networks and explore additional problem domains such as natural language processing or reinforcement learning.

5.4 Future Directions & Final Remarks

Taking APs into account provides a principled approach to set hyperparameters or to inspect new model types. Insights gained from this research can lead to downstream applications like finding more stream-lined training approaches and potentially guiding the design of more effective architectures. From a practical side, the transition model has shown to be a promising tool for monitoring training that may help identifying early training issues reliably. For instance, the transition spread has shown to be rather constant throughout training. Based on my experience, early divergence can be detected by checking it recurrently for inconsistent behavior. However, the theoretical analysis could be extended towards a better-fitting model, not only improving stability of the fit, but also improving the performance of the ActCooLR learning rate scheduler beyond its current use as a proof-of-concept training scheme. A better fitting model could, for instance, fully characterize the entire training process, potentially enabling to pre-calculate the optimal learning

rate offline, just by observing the initialization of the model and its first few training steps. In addition, the APT is a layer-wise measure wich serves also as a candidate for adaptive per-layer learning rate scheduling. Furthermore, the study should be replicated with smoother variants of Rectified Linear Unit (ReLU), which are more commonly used in practice today. This could be achieved, for example, by using ReLU for the AP analysis and the smoothened version for the model inference, since the transition points of the smoothened versions are often also placed in the origin. However, since it is not clear how to interpret the results for an activation function without a sharp decision boundary, this aspect is deferred for exploration in future research.

The research reveals that especially hyperparameter tuning requires new approaches due to the dynamic interactions during training. Resembling discrete optimization using Markov-Chain Monte-Carlo methods, this observed interaction between linear and non-linear optimization suggests further theoretical exploration. In the best case, both aspects can be tuned independently, explicitly deciding on the amount of exploration versus the amount of exploitation during training.

Bibliography

[1] *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. IEEE Computer Society, 2018.

[2] *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[3] Alessandro Achille, Matteo Rovere, and Stefano Soatto. "Critical Learning Periods in Deep Networks". In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019 (cit. on pp. 48, 73, 74).

[4] Naman Agarwal, Rohan Anil, Elad Hazan, Tomer Koren, and Cyril Zhang. "Disentangling Adaptive Gradient Methods from Learning Rates". In: *CoRR* abs/2002.11803 (2020). arXiv: 2002.11803 (cit. on p. 105).

[5] Armin Alaghi and John P. Hayes. "Fast and accurate computation using stochastic circuits". In: *Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014*. Ed. by Gerhard P. Fettweis and Wolfgang Nebel. European Design and Automation Association, 2014, pp. 1–4 (cit. on p. 134).

[6] Milad Alizadeh, Javier Fernandez-Marques, Nicholas D. Lane, and Yarin Gal. "An Empirical Study of Binary Neural Networks' Optimisation". In: *International Conference on Learning Representations*. 2019 (cit. on p. 119).

[7] Arash Ardakani, François Leduc-Primeau, Naoya Onizawa, Takahiro Hanyu, and Warren J. Gross. "VLSI Implementation of Deep Neural Network Using Integral Stochastic Computing". In: *IEEE Trans. VLSI Syst.* 25.10 (2017), pp. 2688–2699 (cit. on p. 134).

[8] Sanjeev Arora, Zhiyuan Li, and Kaifeng Lyu. "Theoretical Analysis of Auto Rate-Tuning by Batch Normalization". In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019 (cit. on p. 99).

[9] Philip Bachman, R Devon Hjelm, and William Buchwalter. "Learning representations by maximizing mutual information across views". In: *Advances in Neural Information Processing Systems*. 2019, pp. 15509–15519 (cit. on p. 179).

[10] David Balduzzi, Marcus Frean, Lennox Leary, et al. "The Shattered Gradients Problem: If resnets are the answer, then what is the question?" In: *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*. Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. PMLR, 2017, pp. 342–350 (cit. on pp. 14, 54, 72).

[11]David G. T. Barrett and Benoit Dherin. “Implicit Gradient Regularization”. In: *CoRR* abs/2009.11162 (2020). arXiv: 2009.11162 (cit. on pp. 48, 75, 100, 105).

[12]Atilim Gunes Baydin, Robert Cornish, David Martínez-Rubio, Mark Schmidt, and Frank Wood. “Online Learning Rate Adaptation with Hypergradient Descent”. In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018 (cit. on p. 105).

[13]Mikhail Belkin, Daniel Hsu, Siyuan Ma, and Soumik Mandal. “Reconciling modern machine learning and the bias-variance trade-off”. In: *CoRR* abs/1812.11118 (2018). arXiv: 1812.11118 (cit. on p. 44).

[14]Yoshua Bengio. “Practical Recommendations for Gradient-Based Training of Deep Architectures”. In: *Neural Networks: Tricks of the Trade - Second Edition*. Ed. by Grégoire Montavon, Genevieve B. Orr, and Klaus-Robert Müller. Vol. 7700. Lecture Notes in Computer Science. Springer, 2012, pp. 437–478 (cit. on p. 75).

[15]Yoshua Bengio, Nicholas Léonard, and Aaron C. Courville. “Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation”. In: *CoRR* abs/1308.3432 (2013). arXiv: 1308.3432 (cit. on pp. 117, 119).

[16]Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. “Curriculum learning”. In: *Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009, Montreal, Quebec, Canada, June 14-18, 2009*. Ed. by Andrea Pohoreckyj Danyluk, Léon Bottou, and Michael L. Littman. Vol. 382. ACM International Conference Proceeding Series. ACM, 2009, pp. 41–48 (cit. on pp. 43, 48).

[17]Joseph Bethge, Haojin Yang, Christian Bartz, and Christoph Meinel. “Learning to Train a Binary Neural Network”. In: *CoRR* abs/1809.10463 (2018). arXiv: 1809.10463 (cit. on p. 134).

[18]Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, et al. “On the Opportunities and Risks of Foundation Models”. In: *CoRR* abs/2108.07258 (2021). arXiv: 2108.07258 (cit. on p. 1).

[19]Léon Bottou. “Online learning and stochastic approximations”. In: *On-line learning in neural networks* 17.9 (1998), p. 142 (cit. on p. 75).

[20]Andrew Brock, Jeff Donahue, and Karen Simonyan. “Large Scale GAN Training for High Fidelity Natural Image Synthesis”. In: *CoRR* abs/1809.11096 (2018). arXiv: 1809.11096 (cit. on p. 134).

[21]Tom B. Brown, Benjamin Mann, Nick Ryder, et al. “Language Models are Few-Shot Learners”. In: *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*. Ed. by Hugo Larochelle, Marc’Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin. 2020 (cit. on p. 1).

[22]Tom B. Brown, Benjamin Mann, Nick Ryder, et al. “Language Models are Few-Shot Learners”. In: *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*. Ed. by Hugo Larochelle, Marc’Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin. 2020 (cit. on p. 105).

[23]Yanshuai Cao, Gavin Weiguang Ding, Kry Yik-Chau Lui, and Ruitong Huang. “Improving GAN Training via Binarized Representation Entropy (BRE) Regularization”. In: 2018 (cit. on p. 179).

[24]Wei-Chen Cheng, Stanley Kok, Hoai Vu Pham, Hai Leong Chieu, and Kian Ming Adam Chai. “Language modeling with sum-product networks”. In: *INTERSPEECH 2014, 15th Annual Conference of the International Speech Communication Association, Singapore, September 14-18, 2014*. Ed. by Haizhou Li, Helen M. Meng, Bin Ma, Engsiong Chng, and Lei Xie. ISCA, 2014, pp. 2098–2102 (cit. on p. 135).

[25]Dami Choi, Christopher J. Shallue, Zachary Nado, et al. “On Empirical Comparisons of Optimizers for Deep Learning”. In: *CoRR* abs/1910.05446 (2019). arXiv: 1910.05446 (cit. on p. 105).

[26]François Chollet. “Xception: Deep Learning with Depthwise Separable Convolutions”. In: *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*. IEEE Computer Society, 2017, pp. 1800–1807 (cit. on p. 20).

[27]Anna Choromanska, Mikael Henaff, Michaël Mathieu, Gérard Ben Arous, and Yann LeCun. “The Loss Surfaces of Multilayer Networks”. In: *Proceedings of the Eighteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2015, San Diego, California, USA, May 9-12, 2015*. Ed. by Guy Lebanon and S. V. N. Vishwanathan. Vol. 38. JMLR Workshop and Conference Proceedings. JMLR.org, 2015 (cit. on p. 44).

[28]PyTorch Contributors. *PyTorch documentation - torch.nn.init*. https://pytorch.org/docs/2.4/nn.init.html#torch.nn.init.calculate_gain. [Online; accessed 18-August-2024]. 2023 (cit. on p. 11).

[29]Robert L. Cook, Thomas K. Porter, and Loren C. Carpenter. “Distributed ray tracing”. In: *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1984, Minneapolis, Minnesota, USA, July 23-27, 1984*. Ed. by Hank Christiansen. ACM, 1984, pp. 137–145 (cit. on p. 121).

[30]Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett, eds. *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*. 2015.

[31]Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. “BinaryConnect: Training Deep Neural Networks with binary weights during propagations”. In: *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*. Ed. by Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett. 2015, pp. 3123–3131 (cit. on p. 119).

[32]Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. “Training deep neural networks with low precision multiplications”. In: *International Conference on Learning Representations Workshop* (2015) (cit. on p. 118).

[33]Marian Croak and Jeff Dean. *A decade in deep learning, and what’s next*. `https://blog.google/technology/ai/decade-deep-learning-and-whats-next/`. [Online; accessed 18-August-2024]. 2021 (cit. on p. 8).

[34]Francesco Croce, Maksym Andriushchenko, and Matthias Hein. “Provable Robustness of ReLU networks via Maximization of Linear Regions”. In: *The 22nd International Conference on Artificial Intelligence and Statistics, AISTATS 2019, 16-18 April 2019, Naha, Okinawa, Japan*. Ed. by Kamalika Chaudhuri and Masashi Sugiyama. Vol. 89. Proceedings of Machine Learning Research. PMLR, 2019, pp. 2057–2066 (cit. on pp. 23, 38).

[35]Wei Dai, Chia Dai, Shuhui Qu, Juncheng Li, and Samarjit Das. “Very deep convolutional neural networks for raw waveforms”. In: *2017 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2017, New Orleans, LA, USA, March 5-9, 2017*. IEEE, 2017, pp. 421–425 (cit. on pp. xvi, 20, 111).

[36]Christian Daniel, Jonathan Taylor, and Sebastian Nowozin. “Learning Step Size Controllers for Robust Neural Network Training”. In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*. Ed. by Dale Schuurmans and Michael P. Wellman. AAAI Press, 2016, pp. 1519–1525 (cit. on p. 105).

[37]Sajad Darabi, Mouloud Belbahri, Matthieu Courbariaux, and Vahid Partovi Nia. “BNN+: Improved Binary Network Training”. In: *CoRR* abs/1812.11800 (2018). arXiv: `1812.11800` (cit. on pp. 120, 121).

[38]DeepSeek-AI, Daya Guo, Dejian Yang, et al. *DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning*. 2025. arXiv: `2501.12948 [cs.CL]` (cit. on p. 1).

[39]Laurent Dinh, Razvan Pascanu, Samy Bengio, and Yoshua Bengio. “Sharp Minima Can Generalize For Deep Nets”. In: *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*. Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. PMLR, 2017, pp. 1019–1028 (cit. on p. 76).

[40]Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, et al. “An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale”. In: *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021 (cit. on p. 1).

[41]Steven K. Esser, Jeffrey L. McKinstry, Deepika Bablani, Rathinakumar Appuswamy, and Dharmendra S. Modha. “Learned Step Size Quantization”. In: *CoRR* abs/1902.08153 (2019). arXiv: `1902.08153` (cit. on p. 127).

[42]Alex Finnegan and Jun S. Song. “Maximum entropy methods for extracting the learned features of deep neural networks”. In: *PLoS Comput. Biol.* 13.10 (2017) (cit. on p. 105).

[43]Pierre Foret, Ariel Kleiner, Hossein Mobahi, and Behnam Neyshabur. “Sharpness-Aware Minimization for Efficiently Improving Generalization”. In: *CoRR* abs/2010.01412 (2020). arXiv: 2010.01412 (cit. on p. 76).

[44]Jonathan Frankle and Michael Carbin. “The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks”. In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019 (cit. on p. 46).

[45]Jonathan Frankle, David J. Schwab, and Ari S. Morcos. “The Early Phase of Neural Network Training”. In: *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020 (cit. on pp. 43, 48, 59, 72–75).

[46]Daniel Franzen, Jan Disselhoff, and David Hartmann. *The LLM ARChitect: Solving ARC-AGI Is A Matter of Perspective*. https://github.com/da-fr/arc-prize-2024/blob/main/the_architects.pdf. 2024 (cit. on p. 1).

[47]Michael C Fu. “Gradient estimation”. In: *Handbooks in operations research and management science* 13 (2006), pp. 575–616 (cit. on p. 121).

[48]Brian R Gaines. “Stochastic computing systems”. In: *Advances in Information Systems Science: Volume 2*. Springer, 1969, pp. 37–172 (cit. on p. 134).

[49]Stuart Geman, Elie Bienenstock, and René Doursat. “Neural Networks and the Bias/Variance Dilemma”. In: *Neural Comput.* 4.1 (1992), pp. 1–58 (cit. on p. 44).

[50]Xavier Glorot and Yoshua Bengio. “Understanding the difficulty of training deep feedforward neural networks”. In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2010, Chia Laguna Resort, Sardinia, Italy, May 13-15, 2010*. Ed. by Yee Whye Teh and D. Mike Titterington. Vol. 9. JMLR Proceedings. JMLR.org, 2010, pp. 249–256 (cit. on p. 106).

[51]Ian J. Goodfellow, Yoshua Bengio, and Aaron C. Courville. *Deep Learning*. Adaptive computation and machine learning. MIT Press, 2016 (cit. on p. 104).

[52]Ian J. Goodfellow and Oriol Vinyals. “Qualitatively characterizing neural network optimization problems”. In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2015 (cit. on p. 72).

[53]Akhilesh Gotmare, Nitish Shirish Keskar, Caiming Xiong, and Richard Socher. “A Closer Look at Deep Learning Heuristics: Learning rate restarts, Warmup and Distillation”. In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019 (cit. on pp. 75, 77, 105).

[54]Priya Goyal, Piotr Dollár, Ross B. Girshick, et al. “Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour”. In: *CoRR* abs/1706.02677 (2017). arXiv: 1706.02677 (cit. on pp. 75, 105).

[55]Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. "Deep Learning with Limited Numerical Precision". In: *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*. Ed. by Francis R. Bach and David M. Blei. Vol. 37. JMLR Workshop and Conference Proceedings. JMLR.org, 2015, pp. 1737–1746 (cit. on p. 118).

[56]Guy Gur-Ari, Daniel A. Roberts, and Ethan Dyer. "Gradient Descent Happens in a Tiny Subspace". In: *CoRR* abs/1812.04754 (2018). arXiv: 1812.04754 (cit. on pp. 48, 73).

[57]Dongyoon Han, Jiwhan Kim, and Junmo Kim. "Deep Pyramidal Residual Networks". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*. IEEE Computer Society, 2017, pp. 6307–6315 (cit. on pp. xvi, 20, 47, 54, 55, 111).

[58]Song Han, Huizi Mao, and William J Dally. "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding". In: *International Conference on Learning Representations (ICLR)* (2016) (cit. on p. 118).

[59]Boris Hanin and David Rolnick. "Deep ReLU Networks Have Surprisingly Few Activation Patterns". In: *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*. Ed. by Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, et al. 2019, pp. 359–368 (cit. on pp. 23–25, 28, 31–33, 37–39, 41, 48, 56, 67, 72, 74).

[60]Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 2016, pp. 770–778 (cit. on pp. xvi, xvii, 14, 19, 63, 125, 126).

[61]Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". In: *2015 IEEE International Conference on Computer Vision, ICCV 2015, Santiago, Chile, December 7-13, 2015*. IEEE Computer Society, 2015, pp. 1026–1034 (cit. on pp. xvii, 11, 19, 28, 54, 67, 71, 87, 106, 201, 203).

[62]Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Identity Mappings in Deep Residual Networks". In: *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part IV*. Ed. by Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling. Vol. 9908. Lecture Notes in Computer Science. Springer, 2016, pp. 630–645 (cit. on pp. xvi, xvii, 19, 54, 55, 63, 81, 111, 125).

[63]Parker Hill, Babak Zamirai, Shengshuo Lu, et al. "Rethinking Numerical Representations for Deep Neural Networks". In: *CoRR* abs/1808.02513 (2018). arXiv: 1808.02513 (cit. on p. 119).

[64]Geoffrey E Hinton and Ruslan R Salakhutdinov. "Reducing the dimensionality of data with neural networks". In: *science* 313.5786 (2006), pp. 504–507 (cit. on p. 11).

[65] Elad Hoffer, Itay Hubara, and Daniel Soudry. "Train longer, generalize better: closing the generalization gap in large batch training of neural networks". In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. Ed. by Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, et al. 2017, pp. 1731–1741 (cit. on pp. 48, 100, 105).

[66] Timothy M. Hospedales, Antreas Antoniou, Paul Micaelli, and Amos J. Storkey. "Meta-Learning in Neural Networks: A Survey". In: *CoRR* abs/2004.05439 (2020). arXiv: 2004.05439 (cit. on p. 105).

[67] Andrew G. Howard, Menglong Zhu, Bo Chen, et al. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications". In: *CoRR* abs/1704.04861 (2017). arXiv: 1704.04861 (cit. on pp. 20, 119).

[68] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. "Densely Connected Convolutional Networks". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*. IEEE Computer Society, 2017, pp. 2261–2269 (cit. on p. 20).

[69] Lei Huang, Jie Qin, Yi Zhou, et al. "Normalization Techniques in Training DNNs: Methodology, Analysis and Application". In: *IEEE Trans. Pattern Anal. Mach. Intell.* 45.8 (2023), pp. 10173–10196 (cit. on p. 12).

[70] Lei Huang, Dawei Yang, Bo Lang, and Jia Deng. "Decorrelated Batch Normalization". In: *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. Computer Vision Foundation / IEEE Computer Society, 2018, pp. 791–800 (cit. on p. 179).

[71] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. "Binarized Neural Networks". In: *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*. Ed. by Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett. 2016, pp. 4107–4115 (cit. on p. 119).

[72] Iris A. M. Huijben, Wouter Kool, Max B. Paulus, and Ruud J. G. van Sloun. "A Review of the Gumbel-max Trick and its Extensions for Discrete Stochasticity in Machine Learning". In: *IEEE Trans. Pattern Anal. Mach. Intell.* 45.2 (2023), pp. 1353–1371 (cit. on p. 117).

[73] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*. Ed. by Francis R. Bach and David M. Blei. Vol. 37. JMLR Workshop and Conference Proceedings. JMLR.org, 2015, pp. 448–456 (cit. on pp. xvii, 12, 19, 106).

[74]Benoit Jacob, Skirmantas Kligys, Bo Chen, et al. “Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference”. In: *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. IEEE Computer Society, 2018, pp. 2704–2713 (cit. on pp. 119, 121).

[75]Stanislaw Jastrzebski, Zachary Kenton, Devansh Arpit, et al. “Three Factors Influencing Minima in SGD”. In: *CoRR* abs/1711.04623 (2017). arXiv: 1711.04623 (cit. on pp. 75, 105).

[76]Yiding Jiang, Behnam Neyshabur, Hossein Mobahi, Dilip Krishnan, and Samy Bengio. “Fantastic Generalization Measures and Where to Find Them”. In: *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020 (cit. on pp. 76, 105).

[77]Glenn Jocher. *YOLOv5 by Ultralytics*. Version 7.0. May 2020 (cit. on pp. 18, 119).

[78]Dhiraj D. Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, et al. “A Study of BFLOAT16 for Deep Learning Training”. In: *CoRR* abs/1905.12322 (2019). arXiv: 1905.12322 (cit. on p. 119).

[79]Dimitris Kalimeris, Gal Kaplun, Preetum Nakkiran, et al. “SGD on Neural Networks Learns Functions of Increasing Complexity”. In: *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. Ed. by Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, et al. 2019, pp. 3491–3501 (cit. on p. 48).

[80]Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. “On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima”. In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017 (cit. on pp. xvi, 34).

[81]Kyounghoon Kim, Jungki Kim, Joonsang Yu, et al. “Dynamic energy-accuracy trade-off using stochastic computing in deep neural networks”. In: *Proceedings of the 53rd Annual Design Automation Conference, DAC 2016, Austin, TX, USA, June 5-9, 2016*. ACM, 2016, 124:1–124:6 (cit. on p. 134).

[82]Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2015 (cit. on pp. 43, 105).

[83]Diederik P. Kingma, Tim Salimans, and Max Welling. “Variational Dropout and the Local Reparameterization Trick”. In: *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*. Ed. by Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett. 2015, pp. 2575–2583 (cit. on p. 118).

[84] S Kirkpatrick, C D Gelatt Jr, and M P Vecchi. “Optimization by simulated annealing”. en. In: *Science* 220.4598 (May 1983), pp. 671–680 (cit. on p. 112).

[85] Teuvo Kohonen. “The self-organizing map”. In: *Proc. IEEE* 78.9 (1990), pp. 1464–1480 (cit. on p. 43).

[86] Alex Krizhevsky. *Learning Multiple Layers of Features from Tiny Images*. 2009 (cit. on pp. xvi, 14, 17, 18, 54, 111).

[87] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*. Ed. by Peter L. Bartlett, Fernando C. N. Pereira, Christopher J. C. Burges, Léon Bottou, and Kilian Q. Weinberger. 2012, pp. 1106–1114 (cit. on pp. 6, 43, 47).

[88] Griffin Lacey, Graham W. Taylor, and Shawki Areibi. “Deep Learning on FPGAs: Past, Present, and Future”. In: *CoRR* abs/1602.04283 (2016). arXiv: 1602.04283 (cit. on p. 119).

[89] Hunter Lang, Lin Xiao, and Pengchuan Zhang. “Using Statistics to Automate Stochastic Optimization”. In: *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. Ed. by Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, et al. 2019, pp. 9536–9546 (cit. on pp. 76, 105).

[90] Ya Le and Xuan S. Yang. “Tiny ImageNet Visual Recognition Challenge”. In: 2015 (cit. on pp. xvi, 14, 17, 18, 54, 111).

[91] Guillaume Leclerc and Aleksander Madry. “The Two Regimes of Deep Network Training”. In: *CoRR* abs/2002.10376 (2020). arXiv: 2002.10376 (cit. on pp. 48, 74, 75, 101, 105).

[92] Yann LeCun, Yoshua Bengio, and Geoffrey E. Hinton. “Deep learning”. In: *Nat.* 521.7553 (2015), pp. 436–444 (cit. on pp. 1, 8).

[93] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. “Gradient-based learning applied to document recognition”. In: *Proc. IEEE* 86.11 (1998), pp. 2278–2324 (cit. on pp. xvi, 8, 17).

[94] Vincent T. Lee, Armin Alaghi, John P. Hayes, Visvesh Sathe, and Luis Ceze. “Energy-efficient hybrid stochastic-binary neural networks for near-sensor computing”. In: *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017*. Ed. by David Atienza and Giorgio Di Natale. IEEE, 2017, pp. 13–18 (cit. on p. 119).

[95] Aitor Lewkowycz. “How to decay your learning rate”. In: *CoRR* abs/2103.12682 (2021). arXiv: 2103.12682 (cit. on pp. 105, 111).

[96] Aitor Lewkowycz, Yasaman Bahri, Ethan Dyer, Jascha Sohl-Dickstein, and Guy Gur-Ari. “The large learning rate phase of deep learning: the catapult mechanism”. In: *CoRR* abs/2003.02218 (2020). arXiv: 2003.02218 (cit. on pp. 48, 75, 105).

[97] Chunyuan Li, Heerad Farkhoor, Rosanne Liu, and Jason Yosinski. "Measuring the Intrinsic Dimension of Objective Landscapes". In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018 (cit. on p. 47).

[98] Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. "Visualizing the Loss Landscape of Neural Nets". In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. Ed. by Samy Bengio, Hanna M. Wallach, Hugo Larochelle, et al. 2018, pp. 6391–6401 (cit. on pp. 27, 44, 85, 100).

[99] Xiuyu Li, Yijiang Liu, Long Lian, et al. "Q-Diffusion: Quantizing Diffusion Models". In: *IEEE/CVF International Conference on Computer Vision, ICCV 2023, Paris, France, October 1-6, 2023*. IEEE, 2023, pp. 17489–17499 (cit. on p. 118).

[100] Yixing Li, Zichuan Liu, Kai Xu, Hao Yu, and Fengbo Ren. "A GPU-Outperforming FPGA Accelerator Architecture for Binary Convolutional Neural Networks". In: *JETC* 14.2 (2018), 18:1–18:16 (cit. on p. 119).

[101] Yuanzhi Li, Colin Wei, and Tengyu Ma. "Towards Explaining the Regularization Effect of Initial Large Learning Rate in Training Neural Networks". In: *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. Ed. by Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, et al. 2019, pp. 11669–11680 (cit. on pp. 48, 75, 105).

[102] Zhiyuan Li and Sanjeev Arora. "An Exponential Learning Rate Schedule for Deep Learning". In: *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020 (cit. on pp. 71, 101, 105).

[103] Darryl Dexu Lin, Sachin S. Talathi, and V. Sreekanth Annapureddy. "Fixed Point Quantization of Deep Convolutional Networks". In: *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*. Ed. by Maria-Florina Balcan and Kilian Q. Weinberger. Vol. 48. JMLR Workshop and Conference Proceedings. JMLR.org, 2016, pp. 2849–2858 (cit. on p. 118).

[104] Xiaofan Lin, Cong Zhao, and Wei Pan. "Towards Accurate Binary Convolutional Neural Network". In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*. Ed. by Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, et al. 2017, pp. 344–352 (cit. on pp. 119, 121, 127).

[105] Shusen Liu, Dan Maljovec, Bei Wang, Peer-Timo Bremer, and Valerio Pascucci. "Visualizing High-Dimensional Data: Advances in the Past Decade". In: *IEEE Trans. Vis. Comput. Graph.* 23.3 (2017), pp. 1249–1268 (cit. on p. 16).

[106] Zechun Liu, Baoyuan Wu, Wenhan Luo, et al. “Bi-Real Net: Enhancing the Performance of 1-Bit CNNs with Improved Representational Capability and Advanced Training Algorithm”. In: *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part XV*. Ed. by Vittorio Ferrari, Martial Hebert, Cristian Sminchisescu, and Yair Weiss. Vol. 11219. Lecture Notes in Computer Science. Springer, 2018, pp. 747–763 (cit. on p. 119).

[107] Ilya Loshchilov and Frank Hutter. “Decoupled Weight Decay Regularization”. In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019 (cit. on pp. xvii, 10, 101, 105).

[108] Ilya Loshchilov and Frank Hutter. “SGDR: Stochastic Gradient Descent with Warm Restarts”. In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017 (cit. on pp. 43, 105).

[109] Jerry Ma and Denis Yarats. “On the adequacy of untuned warmup for adaptive optimization”. In: *CoRR* abs/1910.04209 (2019). arXiv: 1910.04209 (cit. on pp. 75, 105).

[110] Shuming Ma, Hongyu Wang, Lingxiao Ma, et al. “The Era of 1-bit LLMs: All Large Language Models are in 1.58 Bits”. In: *CoRR* abs/2402.17764 (2024). arXiv: 2402.17764 (cit. on p. 118).

[111] Laurens van der Maaten and Geoffrey Hinton. “Visualizing Data using t-SNE”. In: *Journal of Machine Learning Research* 9.86 (2008), pp. 2579–2605 (cit. on p. 16).

[112] Chris J. Maddison, Andriy Mnih, and Yee Whye Teh. “The Concrete Distribution: A Continuous Relaxation of Discrete Random Variables”. In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017 (cit. on pp. 118, 121).

[113] Martín Abadi, Ashish Agarwal, Paul Barham, et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015 (cit. on p. 16).

[114] Nestor Maslej, Loredana Fattorini, C. Raymond Perrault, et al. “Artificial Intelligence Index Report 2024”. In: *CoRR* abs/2405.19522 (2024). arXiv: 2405.19522 (cit. on p. 1).

[115] Christian H. X. Ali Mehmeti-Göpel, David Hartmann, and Michael Wand. “Ringing ReLUs: Harmonic Distortion Analysis of Nonlinear Feedforward Networks”. In: *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021 (cit. on p. 100).

[116] Wenjia Meng, Zonghua Gu, Ming Zhang, and Zhaohui Wu. “Two-Bit Networks for Deep Learning on Resource-Constrained Embedded Devices”. In: *CoRR* abs/1701.00485 (2017). arXiv: 1701.00485 (cit. on p. 119).

[117] Sparsh Mittal. “A survey of FPGA-based accelerators for convolutional neural networks”. In: *Neural computing and applications* (2018), pp. 1–31 (cit. on p. 119).

[118]Adrian Moldovan, Angel Cataron, and Razvan Andonie. “Learning in Feedforward Neural Networks Accelerated by Transfer Entropy”. In: *Entropy* 22.1 (2020), p. 102 (cit. on p. 105).

[119]Grégoire Montavon, Genevieve B. Orr, and Klaus-Robert Müller, eds. *Neural Networks: Tricks of the Trade - Second Edition*. Vol. 7700. Lecture Notes in Computer Science. Springer, 2012 (cit. on p. 104).

[120]Guido Montúfar, Razvan Pascanu, KyungHyun Cho, and Yoshua Bengio. “On the Number of Linear Regions of Deep Neural Networks”. In: *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*. Ed. by Zoubin Ghahramani, Max Welling, Corinna Cortes, Neil D. Lawrence, and Kilian Q. Weinberger. 2014, pp. 2924–2932 (cit. on p. 48).

[121]Alexander Mordvintsev, Christopher Olah, and Mike Tyka. “Inceptionism: Going deeper into neural networks”. In: (2015) (cit. on p. 16).

[122]Pietro Morerio, Jacopo Cavazza, Riccardo Volpi, René Vidal, and Vittorio Murino. “Curriculum Dropout”. In: *IEEE International Conference on Computer Vision, ICCV 2017, Venice, Italy, October 22-29, 2017*. IEEE Computer Society, 2017, pp. 3564–3572 (cit. on p. 48).

[123]Rafael Müller, Simon Kornblith, and Geoffrey E. Hinton. “When does label smoothing help?” In: *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. Ed. by Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, et al. 2019, pp. 4696–4705 (cit. on p. 76).

[124]Vinod Nair and Geoffrey E. Hinton. “Rectified Linear Units Improve Restricted Boltzmann Machines”. In: *Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel*. Ed. by Johannes Fürnkranz and Thorsten Joachims. Omnipress, 2010, pp. 807–814 (cit. on pp. xvii, 19, 54, 67).

[125]Preetum Nakkiran, Gal Kaplun, Yamini Bansal, et al. “Deep Double Descent: Where Bigger Models and More Data Hurt”. In: *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020 (cit. on pp. 48, 75).

[126]Brady Neal, Sarthak Mittal, Aristide Baratin, et al. “A Modern Take on the Bias-Variance Tradeoff in Neural Networks”. In: *CoRR* abs/1810.08591 (2018). arXiv: 1810.08591 (cit. on p. 44).

[127]Arvind Neelakantan, Luke Vilnis, Quoc V. Le, et al. “Adding Gradient Noise Improves Learning for Very Deep Networks”. In: *CoRR* abs/1511.06807 (2015). arXiv: 1511.06807 (cit. on pp. 48, 134).

[128]Chris Olah, Alexander Mordvintsev, and Ludwig Schubert. “Feature Visualization”. In: *Distill* (2017). https://distill.pub/2017/feature-visualization (cit. on p. 16).

[129]A. Emin Orhan and Xaq Pitkow. “Skip Connections Eliminate Singularities”. In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018 (cit. on pp. 14, 54, 72).

[130]D. B. Owen. “A table of normal integrals”. In: *Communications in Statistics - Simulation and Computation* 9.4 (1980), pp. 389–419. eprint: https://doi.org/10.1080/03610918008812164 (cit. on p. 92).

[131]Mohit Pandey, Michael Fernández, Francesco Gentile, et al. “The transformational role of GPU computing and deep learning in drug discovery”. In: *Nat. Mach. Intell.* 4.3 (2022), pp. 211–221 (cit. on p. 8).

[132]Karl Pearson. “LIII. On lines and planes of closest fit to systems of points in space”. In: *Lond. Edinb. Dublin Philos. Mag. J. Sci.* 2.11 (Nov. 1901), pp. 559–572 (cit. on p. 16).

[133]Jorn W. T. Peters and Max Welling. “Probabilistic Binary Neural Networks”. In: *CoRR* abs/1809.03368 (2018). arXiv: 1809.03368 (cit. on p. 118).

[134]Boris Polyak. “Some methods of speeding up the convergence of iteration methods”. In: *Ussr Computational Mathematics and Mathematical Physics* 4 (1964), pp. 1–17 (cit. on pp. xvii, 10).

[135]Hoifung Poon and Pedro M. Domingos. “Sum-product networks: A new deep architecture”. In: *IEEE International Conference on Computer Vision Workshops, ICCV 2011 Workshops, Barcelona, Spain, November 6-13, 2011*. IEEE Computer Society, 2011, pp. 689–690 (cit. on p. 135).

[136]Maithra Raghu, Justin Gilmer, Jason Yosinski, and Jascha Sohl-Dickstein. “SVCCA: Singular Vector Canonical Correlation Analysis for Deep Learning Dynamics and Interpretability”. In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*. Ed. by Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, et al. 2017, pp. 6076–6085 (cit. on pp. 76, 77, 82).

[137]Maithra Raghu, Ben Poole, Jon M. Kleinberg, Surya Ganguli, and Jascha Sohl-Dickstein. “On the Expressive Power of Deep Neural Networks”. In: *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*. Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. PMLR, 2017, pp. 2847–2854 (cit. on pp. 25, 59).

[138]Nasim Rahaman, Aristide Baratin, Devansh Arpit, et al. “On the Spectral Bias of Neural Networks”. In: *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, 2019, pp. 5301–5310 (cit. on pp. 43, 48, 72).

[139]Rajat Raina, Anand Madhavan, and Andrew Y. Ng. “Large-scale deep unsupervised learning using graphics processors”. In: *Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009, Montreal, Quebec, Canada, June 14-18, 2009*. Ed. by Andrea Pohoreckyj Danyluk, Léon Bottou, and Michael L. Littman. Vol. 382. ACM International Conference Proceeding Series. ACM, 2009, pp. 873–880 (cit. on p. 8).

[140]Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. “XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks”. In: *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part IV*. Ed. by Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling. Vol. 9908. Lecture Notes in Computer Science. Springer, 2016, pp. 525–542 (cit. on pp. 119, 127).

[141]Fabian Rathke, Mattia Desana, and Christoph Schnörr. “Locally Adaptive Probabilistic Models for Global Segmentation of Pathological OCT Scans”. In: *Medical Image Computing and Computer Assisted Intervention - MICCAI 2017 - 20th International Conference, Quebec City, QC, Canada, September 11-13, 2017, Proceedings, Part I*. Ed. by Maxime Descoteaux, Lena Maier-Hein, Alfred M. Franz, et al. Vol. 10433. Lecture Notes in Computer Science. Springer, 2017, pp. 177–184 (cit. on p. 135).

[142]Ao Ren, Zhe Li, Caiwen Ding, et al. “SC-DCNN: Highly-Scalable Deep Convolutional Neural Network using Stochastic Computing”. In: *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi’an, China, April 8-12, 2017*. Ed. by Yunji Chen, Olivier Temam, and John Carter. ACM, 2017, pp. 405–418 (cit. on p. 134).

[143]Herbert Robbins and Sutton Monro. “A Stochastic Approximation Method”. In: *The Annals of Mathematical Statistics* 22.3 (1951), pp. 400–407 (cit. on pp. 70, 104).

[144]Michal Rolinek and Georg Martius. “L4: Practical loss-based stepsize adaptation for deep learning”. In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. Ed. by Samy Bengio, Hanna M. Wallach, Hugo Larochelle, et al. 2018, pp. 6434–6444 (cit. on p. 76).

[145]Filip de Roos, Carl Jidling, Adrian Wills, Thomas B. Schön, and Philipp Hennig. “A Probabilistically Motivated Learning Rate Adaptation for Stochastic Optimization”. In: *CoRR* abs/2102.10880 (2021). arXiv: 2102.10880 (cit. on pp. 76, 106).

[146]Olga Russakovsky, Jia Deng, Hao Su, et al. “ImageNet Large Scale Visual Recognition Challenge”. In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252 (cit. on pp. xvi, 14, 17, 18, 111).

[147]Tim Salimans and Diederik P. Kingma. “Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks”. In: *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*. Ed. by Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett. 2016, p. 901 (cit. on pp. 77, 106).

[148] Wojciech Samek, Thomas Wiegand, and Klaus-Robert Müller. "Explainable Artificial Intelligence: Understanding, Visualizing and Interpreting Deep Learning Models". In: *CoRR* abs/1708.08296 (2017). arXiv: 1708.08296 (cit. on p. 16).

[149] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. "MobileNetV2: Inverted Residuals and Linear Bottlenecks". In: *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. IEEE Computer Society, 2018, pp. 4510–4520 (cit. on p. 119).

[150] Jürgen Schmidhuber. "Deep learning in neural networks: An overview". In: *Neural Networks* 61 (2015), pp. 85–117 (cit. on p. 5).

[151] Johannes Josef Schneider and Scott Kirkpatrick. *Stochastic optimization*. Scientific computation. Springer, 2006 (cit. on p. 104).

[152] Bernhard Schölkopf, Alexander J. Smola, and Klaus-Robert Müller. "Nonlinear Component Analysis as a Kernel Eigenvalue Problem". In: *Neural Comput.* 10.5 (1998), pp. 1299–1319 (cit. on p. 16).

[153] Bruno Massoni Sguerra and Fábio Gagliardi Cozman. "Image Classification Using Sum-Product Networks for Autonomous Flight of Micro Aerial Vehicles". In: *5th Brazilian Conference on Intelligent Systems, BRACIS 2016, Recife, Brazil, October 9-12, 2016*. IEEE Computer Society, 2016, pp. 139–144 (cit. on p. 135).

[154] Oran Shayer, Dan Levi, and Ethan Fetaya. "Learning Discrete Weights Using the Local Reparameterization Trick". In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018 (cit. on p. 118).

[155] Noam Shazeer. "GLU Variants Improve Transformer". In: *CoRR* abs/2002.05202 (2020). arXiv: 2002.05202 (cit. on p. 6).

[156] Tao Sheng, Chen Feng, Shaojie Zhuo, et al. "A Quantization-Friendly Separable Convolution for MobileNets". In: *CoRR* abs/1803.08607 (2018). arXiv: 1803.08607 (cit. on p. 126).

[157] Ravid Shwartz-Ziv and Naftali Tishby. "Opening the Black Box of Deep Neural Networks via Information". In: *CoRR* abs/1703.00810 (2017). arXiv: 1703.00810 (cit. on p. 43).

[158] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2015 (cit. on pp. xvi, 19, 111, 200).

[159] Leslie N. Smith. "Cyclical Learning Rates for Training Neural Networks". In: *2017 IEEE Winter Conference on Applications of Computer Vision, WACV 2017, Santa Rosa, CA, USA, March 24-31, 2017*. IEEE Computer Society, 2017, pp. 464–472 (cit. on pp. xvii, 67).

[160]Leslie N. Smith and Nicholay Topin. “Super-Convergence: Very Fast Training of Residual Networks Using Large Learning Rates”. In: *CoRR* abs/1708.07120 (2017). arXiv: 1708.07120 (cit. on pp. xvii, 43, 67, 71, 76, 105, 111).

[161]Samuel L. Smith, Benoit Dherin, David G. T. Barrett, and Soham De. “On the Origin of Implicit Regularization in Stochastic Gradient Descent”. In: *CoRR* abs/2101.12176 (2021). arXiv: 2101.12176 (cit. on pp. 10, 75, 76, 105).

[162]*Speech Command Classification with torchaudio - PyTorch Tutorials 1.13.1+cu117 documentation — pytorch.org*. https://pytorch.org/tutorials/intermediate/speech_command_classification_with_torchaudio_tutorial.html. [Accessed 18-02-2025] (cit. on p. xvi).

[163]Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. “Dropout: a simple way to prevent neural networks from overfitting”. In: *J. Mach. Learn. Res.* 15.1 (2014), pp. 1929–1958 (cit. on p. 48).

[164]Richard P. Stanley. “An Introduction to Hyperplane Arrangements”. In: 2007 (cit. on p. 25).

[165]Jingtong Su, Yihang Chen, Tianle Cai, et al. “Sanity-Checking Pruning Methods: Random Tickets can Win the Jackpot”. In: *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*. Ed. by Hugo Larochelle, Marc’Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin. 2020 (cit. on p. 46).

[166]Ilya Sutskever, James Martens, George E. Dahl, and Geoffrey E. Hinton. “On the importance of initialization and momentum in deep learning”. In: *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*. Vol. 28. JMLR Workshop and Conference Proceedings. JMLR.org, 2013, pp. 1139–1147 (cit. on p. 43).

[167]Swabha Swayamdipta, Roy Schwartz, Nicholas Lourie, et al. “Dataset Cartography: Mapping and Diagnosing Datasets with Training Dynamics”. In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*. Ed. by Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu. Association for Computational Linguistics, 2020, pp. 9275–9293 (cit. on p. 73).

[168]Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. “Efficient Processing of Deep Neural Networks: A Tutorial and Survey”. In: *Proceedings of the IEEE* 105.12 (2017), pp. 2295–2329 (cit. on pp. 126, 127).

[169]Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A. Alemi. “Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning”. In: *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*. Ed. by Satinder P. Singh and Shaul Markovitch. AAAI Press, 2017, pp. 4278–4284 (cit. on pp. 20, 54, 125).

[170] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. “Rethinking the Inception Architecture for Computer Vision”. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 2016, pp. 2818–2826 (cit. on p. 20).

[171] Mingxing Tan and Quoc V. Le. “EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks”. In: *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, 2019, pp. 6105–6114 (cit. on p. 105).

[172] Asher Trockman and J. Zico Kolter. “Orthogonalizing Convolutional Layers with the Cayley Transform”. In: *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021 (cit. on p. 189).

[173] Dmitry Ulyanov, Andrea Vedaldi, and Victor S. Lempitsky. “Instance Normalization: The Missing Ingredient for Fast Stylization”. In: *CoRR* abs/1607.08022 (2016). arXiv: 1607.08022 (cit. on p. 13).

[174] Vincent Vanhoucke, Andrew Senior, and Mark Z Mao. “Improving the speed of neural networks on CPUs”. In: *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*. Vol. 1. Citeseer. 2011, p. 4 (cit. on p. 119).

[175] Ashish Vaswani, Noam Shazeer, Niki Parmar, et al. “Attention Is All You Need”. In: *CoRR* abs/1706.03762 (2017). arXiv: 1706.03762 (cit. on p. 13).

[176] Weitao Wan, Jiansheng Chen, Tianpeng Li, et al. “Information Entropy Based Feature Pooling for Convolutional Neural Networks”. In: *2019 IEEE/CVF International Conference on Computer Vision, ICCV 2019, Seoul, Korea (South), October 27 - November 2, 2019*. IEEE, 2019, pp. 3404–3413 (cit. on p. 105).

[177] Pete Warden. “Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition”. In: *CoRR* abs/1804.03209 (2018). arXiv: 1804.03209 (cit. on pp. xvi, 14, 17, 18, 111).

[178] Jason Wei, Xuezhi Wang, Dale Schuurmans, et al. “Chain-of-Thought Prompting Elicits Reasoning in Large Language Models”. In: *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*. Ed. by Sanmi Koyejo, S. Mohamed, A. Agarwal, et al. 2022 (cit. on p. 1).

[179] Simon Wiedemann, Arturo Marbán, Klaus-Robert Müller, and Wojciech Samek. “Entropy-Constrained Training of Deep Neural Networks”. In: *International Joint Conference on Neural Networks, IJCNN 2019 Budapest, Hungary, July 14-19, 2019*. IEEE, 2019, pp. 1–8 (cit. on p. 105).

[180] Ashia C. Wilson, Rebecca Roelofs, Mitchell Stern, Nati Srebro, and Benjamin Recht. "The Marginal Value of Adaptive Gradient Methods in Machine Learning". In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. Ed. by Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, et al. 2017, pp. 4148–4158 (cit. on p. 105).

[181] Yuxin Wu and Kaiming He. "Group Normalization". In: *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part XIII*. Ed. by Vittorio Ferrari, Martial Hebert, Cristian Sminchisescu, and Yair Weiss. Vol. 11217. Lecture Notes in Computer Science. Springer, 2018, pp. 3–19 (cit. on pp. 13, 19).

[182] Han Xiao, Kashif Rasul, and Roland Vollgraf. "Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms". In: *CoRR* abs/1708.07747 (2017). arXiv: 1708.07747 (cit. on pp. xvi, 14, 17, 111).

[183] Zhen Xu, Andrew M. Dai, Jonas Kemp, and Luke Metz. "Learning an Adaptive Learning Rate Schedule". In: *CoRR* abs/1909.09712 (2019). arXiv: 1909.09712 (cit. on p. 105).

[184] Zhi-Qin John Xu, Yaoyu Zhang, Tao Luo, Yanyang Xiao, and Zheng Ma. "Frequency Principle: Fourier Analysis Sheds Light on Deep Neural Networks". In: *CoRR* abs/1901.06523 (2019). arXiv: 1901.06523 (cit. on p. 72).

[185] Greg Yang, Jeffrey Pennington, Vinay Rao, Jascha Sohl-Dickstein, and Samuel S. Schoenholz. "A Mean Field Theory of Batch Normalization". In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019 (cit. on p. 99).

[186] Mingzhang Yin and Mingyuan Zhou. "ARM: Augment-REINFORCE-Merge Gradient for Stochastic Binary Networks". In: *International Conference on Learning Representations*. 2019 (cit. on p. 121).

[187] Matthew D. Zeiler and Rob Fergus. "Visualizing and Understanding Convolutional Networks". In: *Computer Vision - ECCV 2014 - 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part I*. Ed. by David J. Fleet, Tomás Pajdla, Bernt Schiele, and Tinne Tuytelaars. Vol. 8689. Lecture Notes in Computer Science. Springer, 2014, pp. 818–833 (cit. on p. 60).

[188] Xiaohua Zhai, Alexander Kolesnikov, Neil Houlsby, and Lucas Beyer. "Scaling Vision Transformers". In: *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2022, New Orleans, LA, USA, June 18-24, 2022*. IEEE, 2022, pp. 1204–1213 (cit. on p. 18).

[189] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. "Understanding deep learning requires rethinking generalization". In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017 (cit. on p. 44).

[190] Guodong Zhang, Chaoqi Wang, Bowen Xu, and Roger B. Grosse. "Three Mechanisms of Weight Decay Regularization". In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019 (cit. on pp. 101, 105).

[191] Hongyi Zhang, Yann N. Dauphin, and Tengyu Ma. "Fixup Initialization: Residual Learning Without Normalization". In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019 (cit. on pp. xvi, 20, 47, 54, 55, 67, 71).

[192] Kaiqi Zhao, Animesh Jain, and Ming Zhao. "Adaptive Activation-based Structured Pruning". In: *CoRR* abs/2201.10520 (2022). arXiv: 2201.10520 (cit. on p. 59).

[193] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. "Incremental Network Quantization: Towards Lossless CNNs with Low-precision Weights". In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017 (cit. on pp. 118, 127).

[194] Shuchang Zhou, Zekun Ni, Xinyu Zhou, et al. "DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients". In: *CoRR* abs/1606.06160 (2016). arXiv: 1606.06160 (cit. on pp. 118, 127).

[195] Chenzhuo Zhu, Song Han, Huizi Mao, and William J. Dally. "Trained Ternary Quantization". In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017 (cit. on p. 119).

[196] Zhanxing Zhu, Jingfeng Wu, Bing Yu, Lei Wu, and Jinwen Ma. "The Anisotropic Noise in Stochastic Gradient Descent: Its Behavior of Escaping from Sharp Minima and Regularization Effects". In: *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, 2019, pp. 7654–7663 (cit. on p. 48).

[197] Bohan Zhuang, Chunhua Shen, Mingkui Tan, Lingqiao Liu, and Ian Reid. "Rethinking Binary Neural Network for Accurate Image Classification and Semantic Segmentation". In: *CoRR* abs/1811.10413 (2018). arXiv: 1808.02513 (cit. on p. 119).

[198] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. "Learning Transferable Architectures for Scalable Image Recognition". In: *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. IEEE Computer Society, 2018, pp. 8697–8710 (cit. on p. 20).

List of Figures

List of Tables

List of Algorithms

Appendix

A

A PhD thesis involves extensive experimentation, resulting in a wealth of material that may not be included in the final main manuscript. Three of the incomplete yet noteworthy ideas found their place here in the Appendix.

The contents of the Appendix in more detail:

1. **Appendix A.1 ties in with the APE and discusses ways to regularize that entropy easily by defining gradients on activation pattern counts.** This gradient makes it possible to directly control the in Section 3.1.5 defined complexity of a network by means of a regularizer.

2. **Appendix A.2 aims to discuss how a ReLU activation folds input space.** As the linear part of a deep network can not change higher-order relations of data, we can calculate the linear change of basis out of the global transformation under certain conditions. To enable such observations, this section introduces a metric of deep network mappings that ignores all possible affine transformations and aims to infer the ReLU transformation through the lens of this metric.

3. **Appendix A.3 re-implements the implicit mechanism identified in Appendix A.2 by transforming data explicitly:** Instead of relying on coordinate system changes, the proposed method abstracts from that and applies approximated ReLU-specific effects directly to the global coordinates of the data space.

4. **Appendix A.4** gives a structured view of the ActCooLR algorithm used for the experiments in Section 3.3.

A.1 On Defining Gradients on Activation Patterns

Numerous studies have examined the significance of the batch normalization layer in deep learning. This section explores a possible issue that batch normalization may cause and provides a straightforward solution using defined gradients on APs to address it. However, it is crucial to mention that the newly introduced regularization technique, designed to take the place of batch normalization did not increase the performance of the resulting networks in the conducted experiments (not shown). Nonetheless, defining a gradient in the discrete space of APs appears to be a fascinating area of study that hasn't been tackled, yet, and is thus discussed here.

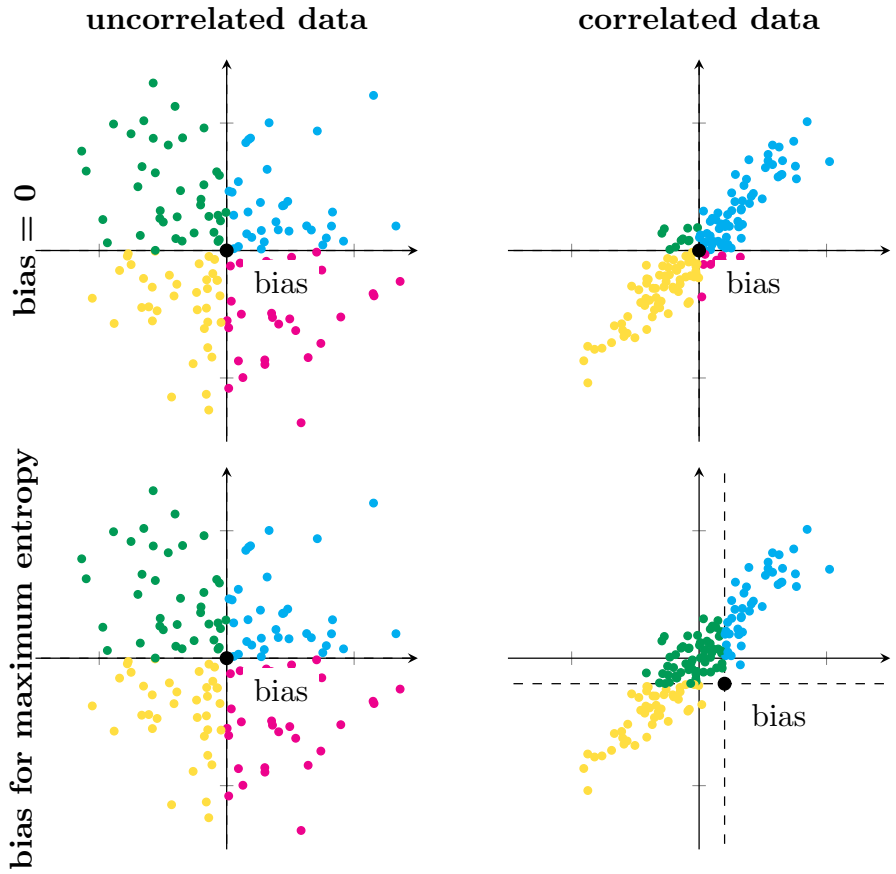


Fig. A.1.: Batch normalization normalizes the batch statistics of each feature to mean zero and standard deviation of 1. It includes two trainable (scaling/shift) parameters, which in practice usually deviate only slightly from 1/0. For uncorrelated Gaussian noise, this maximizes APE (left column) but already for linearly correlated data this no longer holds (right column).

A.1.1 Activation Pattern Entropy under Batch Normalization

Batch normalization implementations typically first re-normalizes data to have standard normalized dimensions. In addition, two trainable parameters may scale and shift the data. As an illustration, let's consider a toy network that represents a batch normalization layer at initialization (w.l.o.g. we omit the scaling factor):
Figure A.1 shows a simple two-dimensional dataset (with standard deviation 1 and mean 0) in two variations: on the left is a version with independent input dimensions, on the right a version with linearly dependent input dimensions (and additional noise). The two rows in the Figure represent different choices for the bias of the batch normalization layer and illustrates that only for a very specific value of stochastic dependence of the data dimensions, batch normalization ensures highest possible activation pattern entropy. The degree of stochastic dependence that batch normalization fits best depends on the initial value chosen for the bias, which is typically set to zero.

Section 3.1.5 has analyzed the APE during training and shows that in practice where data or interim dimensions are not stochastically independent, neither batch normalization, nor initializer methods ensure high entropic activations throughout the training process. (However, residual connections have significantly helped to increase the APE).
The reason for this discrepancy is that batch normalization assumes all input features to be statistically independent. This assumption is not always true for real data as shown in the experiments in Section 3.1.5 and prior work confirms that batch normalization whitens features only sub-optimally [70], i.e. also does not achieve the maximal amount of APs.

This section proposes a new regularization technique that directly optimizes for APE by defining gradients on the APs. In order to directly optimize this value, we define a derivative for the counting operation to optimize for higher APE and name the overall optimization process *discrete activation vanishing entropy* normalization.

A.1.2 Regularizing the Activation Pattern Entropy

In order to study the effect of APE (or the lack thereof) on the training of deep networks, we now devise a regularizer that aims at maximizing APE. In general, measuring entropy of non-trivial data is a hard problem. A major issue is complexity: A binary activation vectors of k bits can assume 2^k different patterns. This complexity issue can be avoided by using approximations, such as second order approximations [23] or variational bounds [9].

Algorithm 3 applyDeriv

This function defines the gradient for the first input w.r.t. the third input. It does so by remapping the gradients Δcounts (given by the autograd computation that uses the counts) to the corresponding input

Forward Input: counts, ids_backward, $X \in R^{n \times d}$
save for backward ids_backward, X
Forward Output: counts

Backward Input: $\Delta counts$
$\Delta X_i = X \cdot (\Delta counts)_{ids\_backward_i}$ **for** $i = 1, \ldots, n$
Backward Output: ΔX

Algorithm 4 count_act

This function uses a hash-table to quickly count the occurrences and backward-indices of all APs. Depending on the data dimensions one could also use radix-sort as a prefix sum as an alternative, more GPU-friendly implementation.

Input: activation list $A \in \{0,1\}^{n \times d}$
let $D = \{0,1\}^d \to 2^{\{1,\ldots,n\}}$ be an empty hash table
let counts, ids_backward two lists
for $i = 1$ **to** n **do**
  if $A_i \notin D$ **then**
    $D(A_i) \leftarrow$ **len**(counts)
    append 1 **to** counts
  else
    increase $\text{counts}_{D(A_i)}$ **by** 1
  end if
  append $D(A_i)$ **to** ids_backward
end for
Output: counts, ids_backward

To optimize directly for activations with higher entropy, one can define the individual directions based on the measured AP. The obstacle is the discontinuous nature of activations; in order to integrate this discrete optimization problem smoothly with continuous, gradient-based optimization, the trick is to define a relaxed, probabilistic histogram that yields useful gradients.

Measuring as a first step is done just as in as in Section 3.1.5; APs are measured during training by hashing activations and counting APs efficiently on the GPU. Then, given the AP count vector c, the APE can be calculated simply using the formula

$$H(c) := -\sum_i \frac{c_i}{N} \cdot \log_2 \frac{c_i}{N}. \tag{A.1}$$

Algorithm 5 Activation Count Layer
(as being defined in any autograd system)

Input: data coming from previous layer $X \in \mathbb{R}^{n \times d}$

// query all unique activation patterns, together with their corresponding indices in the data

$\text{counts}, \text{ids\_backward} = \text{count\_act}(\text{relu}(\text{sign}(X)))$

// use counts to define gradient on X

$\text{counts} = \text{applyDeriv}(\text{counts}, \text{ids\_backward}, \tilde{X})$

$\text{total} = \sum_{c \in counts} c$

$H = -\sum_{c \in counts} \frac{c}{total} \cdot \log \frac{c}{total}$

Output: autograd derivable activation Entropy H

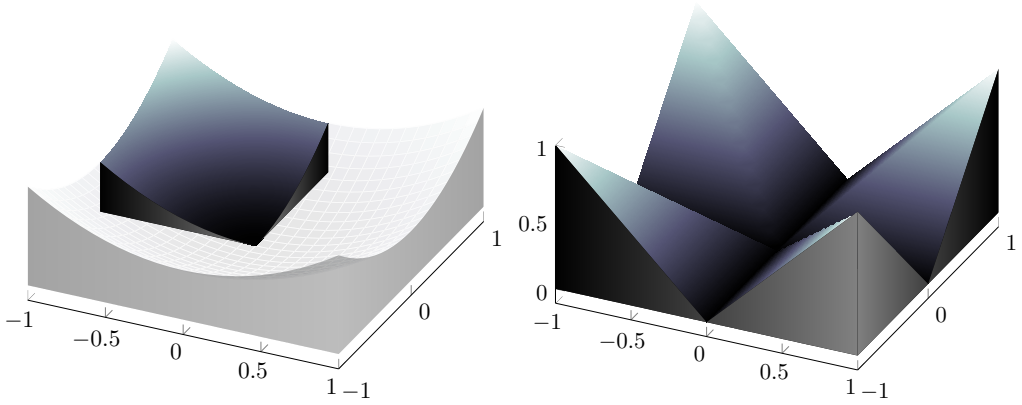
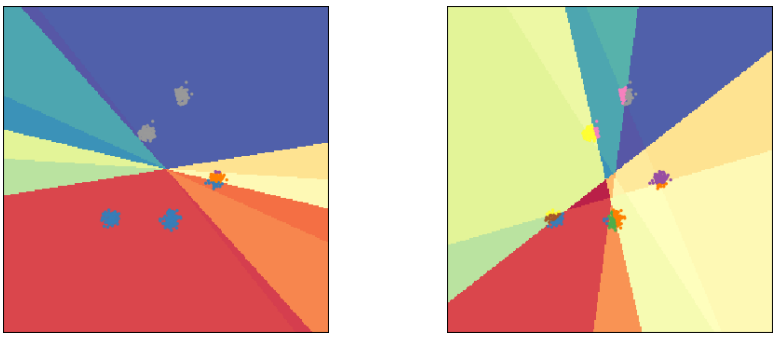


Fig. A.2.: Relaxed potential function: Counting points (as part of a histogram) has a derivative of 0 w.r.t. the data points. Thus, we define a relaxed potential function for a point being in a bin. The image shows the potential we use for the partial derivatives $\frac{\partial count}{\partial data}$ for a two dimensional space that has four possible activations in this case.

The derivative for the layer that is used to compute the count vector c, given by $y(x) = \text{ReLU}\,(x^T W + b)$, is then formally given by

$$\frac{\partial H(c)}{\partial y} = \sum_i \frac{\partial H}{c_i} \frac{\partial c_i}{\partial y}. \tag{A.2}$$

The first factor in the sum can be evaluated using standard automatic differentiation. However, the term $\frac{\partial c_i}{\partial y}$ is not defined, since the counting operation is not differentiable. This gradient specifies the direction a point y needs to move towards in order to decrease the count for that activation pattern. As an alternative view, we can find a differentiable mapping g that maps y so that $\frac{\partial g}{\partial y}$ behaves like $\frac{\partial c_i}{\partial y}$ should. Two such relaxed "counting" functions g worked well in practice and are illustrated in Figure A.2:



(a) He-init & Batch Normalization (b) He-init & Entropy Normalization

Fig. A.3.: A toy dataset consisting of five normal distributed point clusters in a single-layer network optimized by (a) He initialized weights + batch normalized pre-activations and (right) He init. and entropy maximization. Colors indicate relu-hyperplanes.

Option 1: The piecewise linear triangle function:

$$g_{(linear)}(y) := \max\left(0, \min_i |y_i|\right) \tag{A.3}$$

Option 2: A quadratic basis

$$g_{(quad)}(y) := \frac{1}{2}\sum_i y_i^2 \tag{A.4}$$

In total, the entropy be implemented in autograd systems as shown in Algorithm 3 and Algorithm 5. To ensure correct derivative routing, activation ids must be tracked during counting (see Algorithm 4).

Figure A.2 illustrates the two relaxation functions' visualization. In practical applications, both functions appear to be useful, with the quadratic variant delivering marginally superior outcomes, albeit at the cost of heightened sensitivity to parameter choices (regularization strength and learning rate). Figure A.3 visualizes a toy example optimized using the suggested histogram normalization (right) as compared to the outcome achieved with batch normalization (left).

A.2 ReLU's Implicit Subspace Folding

Chapter 2 has identified disjoint subsets of input space, where a neural network applies effectively the same constant mapping. These subsets have been introduced under the name ACs, and, the mappings on these have been identified as affine transformations.
The goal for this section is to understand how these transformations help a neural network to solve a specific task such as classification or regression on a high-dimensional input space. To achieve this goal, the computations of deep networks are going to be interpreted as deformations of input space. Viewed this way, the network performs a non-injective folding of the input space.

A.2.1 Intrinsic and Extrinsic Coordinates

The primary task to comprehend the described folding of input space involves inspecting a simple network with just a single hidden layer. The case of more layers unfolds as simply recursively re-iterating the technique of the single-layer case.
Thus, let F be a simple neural network (as defined in Equation (2.7)) consisting of a single hidden layer defined by the tuple (W, b), and followed by a ReLU-activation. Additionally, assume there exists a final linear layer (typically used for downstream tasks such as classification) defined by the tuple (V, c)). In total, let the network be given as

$$F = \text{ReLU}\left(x^T \cdot W + b\right) \cdot V + c, \tag{A.5}$$

with an input dimension d_1, interim dimension d_2 and output dimension d_3.

Before starting to deform input space, a short recap of Section 2.2: (The visualization in Figure A.4 may help with following along.[1]) The neural network F first applies the affine transformation, $y = x \cdot W + b$, mapping from input space data into a higher-dimensional space (This mapping is shown in Figure A.4(i)). The coordinate system in input space are further called *extrinsic coordinates* and the coordinate system of the interim representation y *intrinsic coordinates*. In these intrinsic coordinates the network then performs the non-linear ReLU-operation, mapping all negative components to zero (shown in Figure A.4(ii)). Finally, the results of the ReLU-activation are then mapped using a second linear layer into a second intrinsic space

[1] To simplify the visualization, the dataset lives in a rectangular subspace of the input domain, $[-c, c]^2 \subset \mathbb{R}^3$, and the network shown in Figure A.4 has three filters in the first layer, and two filters in the final/second layer, i.e. $d_1 = 2, d_2 = 3, d_3 = 2$. The input space is three-dimensional, $d_1 = 3$. The network weights are chosen using *He-initialization*, but for visualization's sake normalized to have unit-norm.

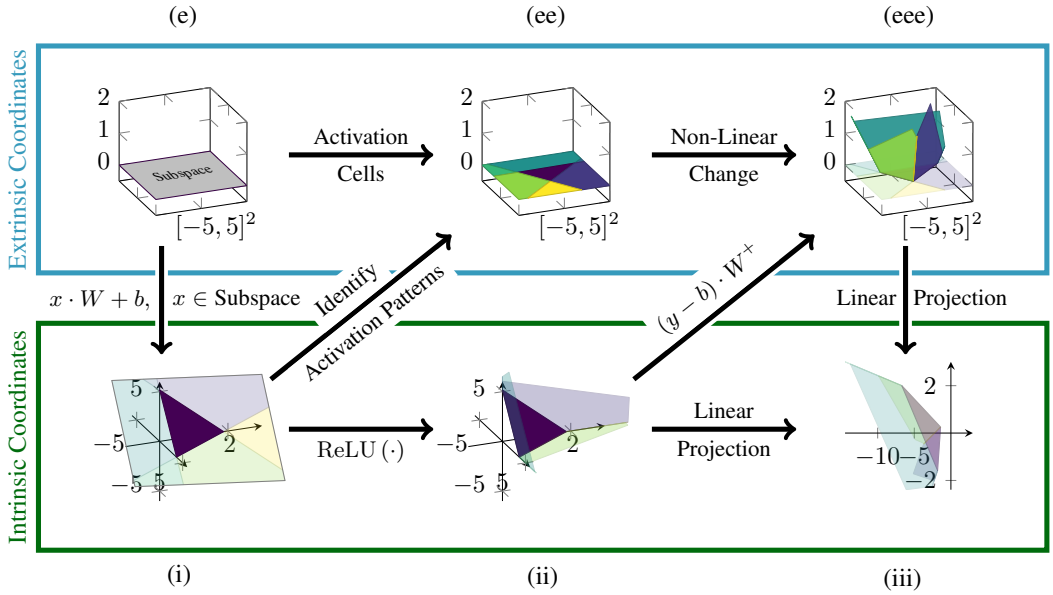


Fig. A.4.: This schematic visualization shows the computation of a single linear layer followed by a ReLU-activation and a final linear output layer afterward using two views: intrinsic view and extrinsic view. In the **intrinsic view**, the input space (e) gets first projected into the coordinate system of the linear layer (i), followed by the ReLU-activation that maps negative values to 0 (ii), finally, those non-linearly mapped coordinates are then projected into an output layer (iii). In the **extrinsic view**, the input space (e) is implicitly separated into ACs (ee) that are non-linearly transformed (eee) such that another output layer maps the coordinates into the same points that have been found using the intrinsic view (iii). The operation shown from (i) to (ee) represent the algorithm that has been used in the AC analysis in Section 2.2. This section (Appendix A.2) aims to better understand the transformation from (ii) to (eee). The overall goal is to then use this knowledge to construct an alternative kind of network in Appendix A.3 that approximates the step (ee) to (eee) directly without the requirement of computing interim results explicitly.

(shown in Figure A.4). In this last coordinate system, axes are typically given a semantic annotation – for instance the likelihood of the respective classes of the dataset in case of a classification task – in order to then define a loss function on.

Section 2.2 has already identified the different resulting affine transformations that are implicitly performed due to using a combination of a linear layer and a succeeding ReLU-activation. The domains of those distinct affine transformations have been denoted as *ACs*, which are disjoint subsets in input space. To compute these, the intrinsic coordinates before the ReLU-activation (Figure A.4(i)) can be used to identify the AP, the sign vector that describes whether the components of the interim results after the ReLU-activation are active or not. Note that the partitioning into ACs occurs in extrinsic coordinates, that is, the same space in which the dataset resides (Figure A.4(e)).

Each AP a corresponds to an AC A_a, serving as the domain for a distinct affine transformation implicitly defined by the neural network. Let the affine transformation of the AC A_a be further denoted as

$$f_a : A_a \to f_a(\mathbb{R}^{d_3}),\ f_a(x) \mapsto x \cdot V_a + c_a, \tag{A.6}$$

where the pair (V_a, c_a) is chosen s.t. $F(x) = f_a(x) \,\forall\, x \in A_a$.

Having previously identified *where* the function that the neural network F computes differs, the focus now shifts to the *kind* of transformation it applies to the respective regions. The idea is to identify transformations of the ACs that would simulate the action of ReLU in the original coordinate system, the extrinsic coordinates. A good start is to cancel the affine transformation applied by the first linear layer as it is only a change of basis.

Definition 7. *The* extrinsic projection *back into input space of input x for a single linear layer followed by a ReLU-activation,*

$$y(x) = \mathit{relu}\left(x \cdot W + b\right),$$

is defined as the projection

$$\hat{x} = \left(y\left(x\right) - b\right) W^{+},$$

where W^+ is the pseudo-inverse of W, i.e. it has the property $W = WW^{+}W$. Formally, the coordinates $y(x)$ are named intrinsic coordinates *and the coordinate system of $\hat{x}$ and x* extrinsic coordinates.

The goal of Definition 7 was to re-create the effect of the non-linearity in input space. However, the approach is not exact: first it maps from one space to another, applies the non-linearity by zeroing components of the result and then maps back, which is in most cases *not* a lossless operation.
It follows a discussion about when and what information is lost in this process of "re-creating" the effect of the non-linearity using the definition above. But, to define what the term "information loss" actually means in this context, an additional linear layer needs to be postulated in the following. In practice such a linear layer after a non-linearity is typically used[2], justifying such an approach. The idea is to define a new representations as lossless, if for every choice of that final layer in data space,

[2]This last layer is typically used to map the interim data into a final vector space with a predefined semantic. This space is usually used to define a loss to be optimized for, for instance, targeted to solve a classification or regression task and thus it's output is required to coin the notion of information loss of a projection of space.

there exists another final layer in the "re-created" space that maps to the same results.

Definition 8. *Two representations $x \in \mathbb{R}^d$ and $y(x) : \mathbb{R}^d \to \mathbb{R}^{d'}$ are* equal w.r.t. any affine transformation *if for every affine transformation (V, c), there exists a "counter" affine transformation (V', c') that satisfies*

$$x \cdot V + c = y(x) \cdot V' + c' \quad \forall x.$$

A direct consequence of the definition of $\hat{x}$ is that it maps the "identity"-AC (if it exists) always to itself through the lens of any affine transformation:

Lemma 1. *Assume the "identity" activation cell A_1 exists, i.e. the cell that corresponds to the AP consisting of only ones, $a(x) = (1, \ldots, 1)$, and W^+ is the pseudo-inverse of W.*
For any choice (W, b, V, c), there exists a affine transformation given by the pair $(V', c') = (W \cdot V, c)$ s.t.

$$\hat{x}^T \cdot V' + c' = y\,(x) \cdot V + c, \quad \forall x \in \mathcal{A}_1.$$

Proof. In the case $x \in \mathcal{A}_1$, ReLU behaves like the identity function, thus

$$\begin{aligned}
\hat{x}^T \cdot V' + c' &= (y(x) - b)W^+WV + bV + c \\
&= (xW + b - b)W^+WV + bV + c \\
&= xWW^+WV + bV + c \\
&= xWV + bV + c \\
&= y(x) \cdot V + c.
\end{aligned}$$

□

However, a loss of information becomes imminent if requiring the result of only one *additional* AC to be retained. In this case, information can be retained through the extrinsic projection if the possible choices of the bias term b are restricted.

Lemma 2. *Again, let A_1 be the "identity" activation cell, i.e. assume it exists, and assume W^+ is the pseudo-inverse of W. Additionally, assume the "zero" activation cell exists as well A_0, i.e. the cell that corresponds to the AP consisting of only zeros, $a(x) = (0, \ldots, 0)$.*

For any choice of (W, b, V, c)*, satisfying* $b = \hat{b}W$*, there exists a affine transformation given by the pair* $(V', c') = (W \cdot V, bW^{+}WV + c)$ *s.t.*

$$\hat{x}^T \cdot V' + c' = y\left(x\right) \cdot V + c, \quad \forall x \in A_1 \cup A_0.$$

Proof. First consider the case $x \in \mathcal{A}_1$. In this case, ReLU behaves like the identity function,

$$\begin{aligned}
\hat{x}^T \cdot V' + c' &= (y(x) - b)W^{+}WV + bW^{+}WV + c \\
&= (xW + b - b)W^{+}WV + bW^{+}WV + c \\
&= xWW^{+}WV + \hat{b}WW^{+}WV + c \\
&= xWV + \hat{b}WV + c \\
&= y(x) \cdot V + c.
\end{aligned}$$

Next, consider the case $x \in \mathcal{A}_0$. In this case ReLU zeros all components,

$$\begin{aligned}
\hat{x}^T \cdot V' + c' &= (y(x) - b)W^{+}WV + bW^{+}WV + c \\
&= (0 - b)W^{+}WV + \hat{b}WW^{+}WV + c \\
&= 0 + c \\
&= y(x)V + c.
\end{aligned}$$

□

Thus, in the general case, the extrinsic projection $\hat{x}$ does not represent the same information as $y(x)$, yet. But under what circumstances *does* it contain the same information? A straightforward case is obtainable directly from the definition.

Lemma 3. *For any choice of* (W, b, V, c) *where the weight matrix* W *has linearly independent columns there exists a affine transformation given by the pair* $(V', c') = (W \cdot V, bV + c)$ *s.t.*

$$\hat{x}^T \cdot V' + c' = y\left(x\right) \cdot V + c, \quad \forall x \in \mathbb{R}^{d_1}.$$

Proof. Assuming that W has linearly independent columns results in W^{+} being the left inverse of W, which directly shows the above:

$$\begin{aligned}
\hat{x}^T \cdot V' + c' &= \hat{x}^T WV + bV + c \\
&= \left(y\left(x\right) - b\right) W^{+}WV + bV + c \\
&= \left(y\left(x\right) - b\right) V + bV + c \\
&= y\left(x\right) V + c.
\end{aligned}$$

□

The utility of this case is limited: While it ensures equality w.r.t. any affine transformation, the assumption of linearly independent columns in W (i.e. the number of filters of the linear layer) means that the number of filters cannot exceed the number of input dimensions for the extrinsic projection to contain the same information as the intermediate results in a deep network using ReLU-activations.
What information is missing in the general case then?

Intuition of information being lost

W.L.o.G (by including homogeneous coordinates in x implicitly), consider the case without any bias, $b = 0$. The ReLU-activation will be reformulated below as a multiplication with the data-dependent AP given in matrix notation, $A(x) = \mathrm{diag}\,(a(x))$. Using again the counter affine transformation $(V', c') = (WV, c)$, one can obtain the following equation:

$$\begin{aligned} \hat{x}^T \cdot V' + c' &= y\,(x)\, W^+ WV + c \\ &= xWA(x)W^+ WV + c. \end{aligned} \tag{A.7}$$

For the pseudo-inverse property to apply, and thus having equality of representations, the matrix $A(x)$ would need to commute with the product W^+W. (Lemma 3 has solved this by adding $W^+W = \mathrm{id}$ as a requirement). In the general case this is not true; while W^+W and $A(x)$ are both symmetric, their multiplication is not:

$$A(x)W^+W = A(x)^T(W^+W)^T = (W^+WA(x))^T \neq (W^+WA(x)).$$

This would, however, be required for the pseudo-inverse property to be applicable. While the left-hand side zeroes the rows of W^+W, the right-hand side of the inequality zeroes the columns of W^+W, showing qualitatively what information is lost.

The Multiple-Layer Case

What the text above has shown so far is how to transform intrinsic coordinates into extrinsic ones for a single linear layer followed by a ReLU-activation.
The next step is to apply the same technique to multiple such layer-activation combinations.

Definition 9. *(Extrinsic Projection of l Layers)*
For simplicity's sake, the method is described for homogeneous coordinates (i.e. $b = 0$) and uses the matrix notation of a ReLU activation a layer k defined in Definition 1, $A_l(x)$. (Note that the matrices $A_l(x)$ depend also on W_l). Assume the network F consists of n linear layers defined by the filter matrices $W_l, 1 \leq k \leq n$, each followed by a ReLU activation and a final affine transformation defined by the matrix V:

$$F(x) = xW_1A_1(x)W_2A_2(x)\dots W_nA_n(x)V. \tag{A.8}$$

The intrinsic coordinates on layer l are then given by

$$y_l(x) = xW_1A_1(x)W_2A_2(x)\dots W_l. \tag{A.9}$$

Applying one extrinsic projection step, $y_l(x)W_l^+$, as described in the previous section would give us the extrinsic projection into the space of layer $l-1$. Thus, the projection needs to be applied for every layer that changes the coordinate system, step by step until reaching the first layer, and finally the input space again. The extrinsic view for layer l is thus given by

$$\hat{x}_l = y(x)W_l^+\dots W_2^+W_1^+. \tag{A.10}$$

Note that the projection matrix $W_l^+\dots W_2^+W_1^+$ is independent of x.

Batch Normalization, Residual Connections and Convolutional Layers

Batch normalization (see Section 2.1) normalizes data using the batch mean (or a learned offset in evaluation mode) and the batch standard deviation (or a learned scaling in evaluation mode). The batch normalization layer can be easily included in the extrinsic projection scheme: when projecting back into the input space those coordinate changes just need to be taken into account, for instance by including those into the weight matrices directly. The same applies to residual connections. Typically these identity operations are only defined for a subset of the intermediate dimensions, thus, only a subset of filters need to reflect this by adding a binary diagonal matrix with ones at those components, where the identity operation would take place due to the residual connection.
Convolutional operations are harder to reconcile with the idea of inverting the intrinsic coordinate system using the pseudo-inverse of the weight matrix. While the convolution uses the transposed convolution for gradient computation, the operation is not the inverse of the convolution in the general case. While efficient orthogonalizations of the convolution operation do exist [172], Appendix A.3 will

use approximated versions of this transformation, leaving the transposed convolution as a sufficiently precise choice in practice.

Summary

This section has shown: under the assumption that the input space is a subset of the output space (e.g. by assuming the dataset is embedded in a higher-dimensional space as a plane with all unused coordinates set to 0), the extrinsic projection serves as a *view* of the inner working of a deep feed-forward network using ReLU-activations. Thus, instead of analyzing the interim results of the network, its interim results can be reformulated as non-linear deformations of the input space. The role of the extrinsic projection is to inverse the change of coordinates into intrinsic coordinates, but to still keep the essence of what the ReLU activation does.
In this back-projected view, a single ReLU-activation affects the overall deformation only for points which have interim results that are zeroed. Shown in Lemma 1 the identity cell is always mapped to the identity mapping under the extrinsic projection. This is different to the intrinsic coordinates, where also points of the identity cell are skewed using a affine transformation.

A.2.2 Extrinsic Coordinates of a Toy Example

The idea of the following experiment is to observe the training process through the lens of the extrinsic projection defined in the last section. As the extrinsic projection converts intrinsic coordinates into extrinsic ones, toy-Datasets living in two dimensions simplify the task of visualization.

Experimental Setup

Figure A.5 shows the two-dimensional classification toy dataset named *checkboard dataset*. The dataset consists of four classes, each arranged in a checkboard-like manner. Thus, the target of this dataset is to learn the points according to the classification grid, meaning a network needs to learn a representation that makes it easy for a linear classifier to separate the respective four classes. For instance Figure A.5(b)) visualizes an example of such an “optimal” representation. The loss to minimize is simply cross-entropy with softmax (see Section 2.1). The dataset is already normalized to have zero mean and unit-standard deviation, thus no dataset normalization is required in this experiment and use also no augmentation. To

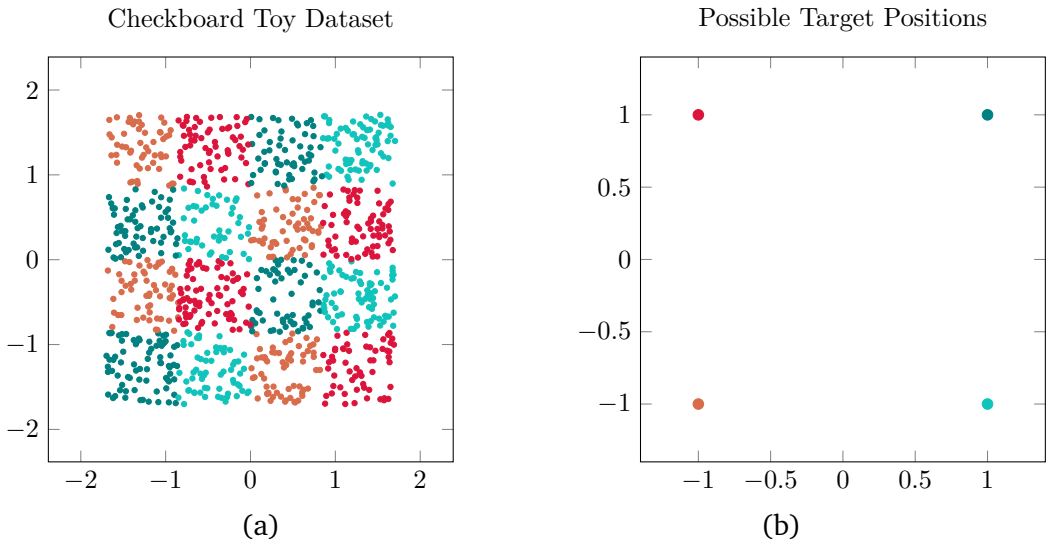


Fig. A.5.: To visualize implicit deformations in input space easily using the extrinsic projection, the dataset dimension needs to be low-dimensional. Of particular interest are two-dimensional toy datasets in, as they allow to visualize the deformation on a sheet of paper. The shown toy dataset is name *checkboard dataset* (a) are randomly sampled points in a 2D boundary. All points are assigned to one of four classes (yellow, red, teal and lightblue), assembling together a checkboard pattern. The domain is chosen in a way that the dataset as a whole has zero mean and a unit standard deviation. The target positions (b) shows a possible goal positions in two dimensions for a linear layer with four filters to classify each point to it's respective classes.

generate the dataset, sample points in the dataset domain $[-\sqrt{3}, \sqrt{3}]$ (the domain is chosen to have unit standard deviation) are generated and the class of each point (x, y) is given by:

$$c = \lfloor \frac{x+y}{\sqrt{3}} \cdot 4 \rfloor \mod 4. \tag{A.11}$$

The training dataset consists of 200 batches with 250 points each. For the final accuracy, separate dataset of the same size is sampled. The visualizations of the extrinsic projection uses a third small dataset, consisting of 1000 fixed points to show the implicit deformation throughout training. The networks to inspect using the extrinsic projection is a simple feed-forward network as defined in Equation (2.7) without any batch normalization layers. The optimizer used is plain SGD without momentum or weight decay, applying a fixed learning rate of 0.1 over 10 epochs.

Two variants of this experiment are considered in the following:

- **(1) The number of filters inside the network ensures equality w.r.t. any affine transformation**:

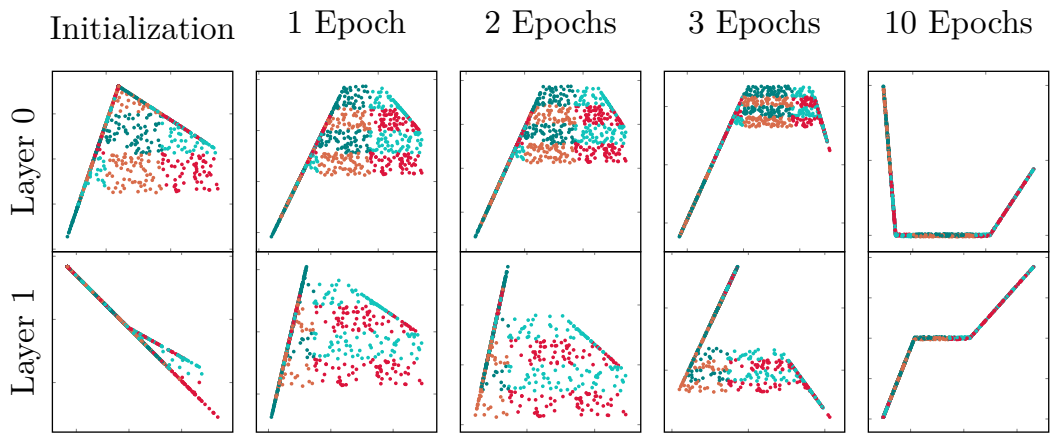


Fig. A.6.: A lossless extrinsic projection (i.e. one that does not implicitly remove unseen data) requires in two-dimensional datasets only at most two filters per layer. This figure shows the training of a two-layer neural network with two filters in each layer and a final four-filter linear classification layer through the lens of extrinsic projection. Test data points are back-projected using the neural network at four stages of training: at initialization, after 1 epoch, after 2 epochs, after 3 epochs, and after 10 epochs of training with 200 batches with 250 training data points per epoch. The network visualized is only able to separate two sets of classes (yellow and green against red and blue). The deformation used is a folding in two dimensions, effectively squashing a sub-spaces onto a line.

This holds when the filters of the weight matrices are linearly independent (see Lemma 3). For two-dimensional datasets, only 1 or 2 filters per layer can be used.

- **(2) Arbitrary number of filters per layer**:
  While this case provides insights into the training, the resulting plots do not exhibit the same information the network has internally, but hide additional dimensions. (This is in contrast to the first case, where the plots shown can be used to recompute the internal representations from the 2D-positions on the sheet of paper).

Experiment (1): equality w.r.t. any affine transformation

The first experiment uses a simple two-layer fully-connected network as defied in Equation (2.7) with two filters in each layer. According to Lemma 3, if the columns of W are linearly independent, equality w.r.t. any transformation is ensured.[3] In this example, the extrinsic projection serves as a true representation of what the

[3]Technically, the filters could become dependent during training or be sampled dependent already, however, re-sampling solves that as the probability of such sampling is sufficiently small in this experimental setup.

ReLU-activation effectively does to the input dataset in the neural network's hidden layers.
Figure A.6 visualizes the training process of the neural network for the two-dimensional toy dataset described above. The rows in the plot represent the respective two layers of the neural network, each back-projected using Definition 9. The columns represent the deformations at different stages of training: at step 0 (initialization, i.e. before any training), after one, after two, after three and after ten epochs of training respectively.
The experiment shows that the implicit deformations made by the network in this case are to first compress the first and last column of the dataset horizontally and then to compress the remaining columns vertically. Note that the "zeroing" of the ReLU-activation implicitly accomplishes the compression in the resulting back-projected view. In this example, the second layer learns to "bend" one of the "arms" in the $\sqcup$-shaped result down to a $\sqrt{\ }$-shaped result, which then can then be used to distinguish two sets of classes using a single dividing diagonal line ($/$) in data of the bottom right plot.
The extrinsic projection shows what kind of "deformation" operations are possible with only two filters in a two-dimensional space. Also, this experiment shows that two filters with two layers are unfortunately not sufficient to classify the checkboard dataset successfully. However, the network manages to distinguish two of the classes from the other two classes; the test accuracy denotes 48.9% after training for ten epochs.

While more layers could potentially lead to a better classification, even with only two filters, its optimization is much harder; The visualization shows already that the zeroing squishes large parts of the dataset. The problem is that these squished points will always have zero gradients as well, similar to the well-known "dead-neuron" effect, but only for some points in the dataset instead of for specific neurons. Unless countering this, a network with two-filter but more layers will quickly lead to all points being mapped to a single position without any possibility of optimization. This is due to the "zeroing" that also removes any gradient for negative values.

Experiment (2): Projected view of 50 Filters per Layer

The last experiment has revealed the mechanisms that take place inside intermediate representations of a two-filter deep network. The experiment was chosen in a way that the information visualized and the information used internally by the network where identical. Constrained by being capable of only visualizing two dimensions at once on paper, the same cannot be done for more than two filters per layer, i.e.

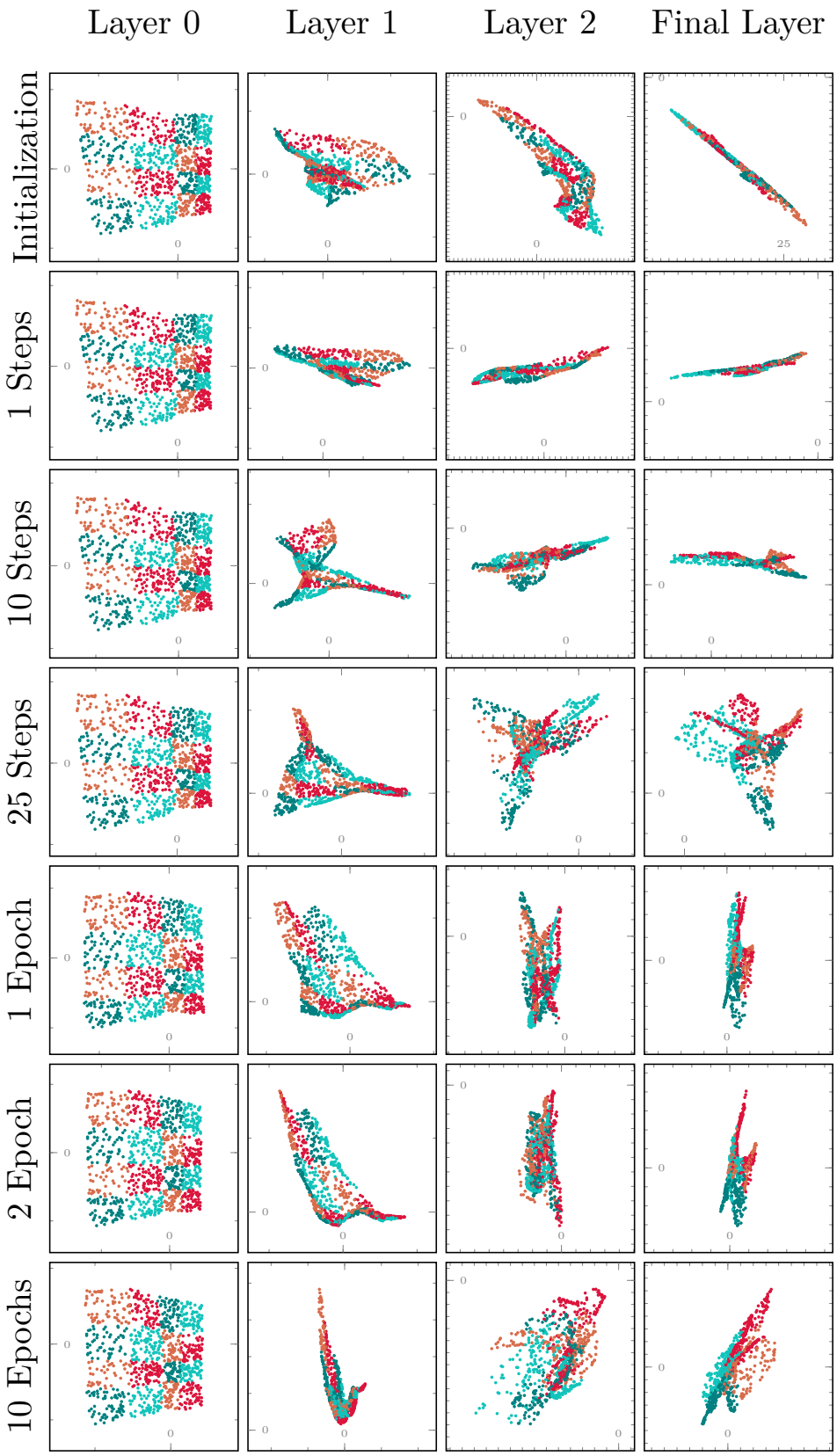


Fig. A.7.: Back-projected layer-wise results of the 2d-dataset for a three hidden layer network with 50 filters each. The rows show the course of the deformation over the course of training the network for 10 epochs at various training steps. The columns of the figure indicate the extrinsic projections of the first, the second, the third and the final classification layer of the network.

a network with 50 filters or more on each layer. Apply the extrinsic projection in this case, implicitly assumes the input space is also 50 dimensional but uses only the first 2 dimensions, due to the requirement of the filters being linearly independent for the extrinsic projection to be used in a lossless fashion. The result would be a view that does not show the exact information that the network has anymore. One thing, that still is true, is that the "identity"-cell, i.e. the AC that is not affected by any ReLU-activation (see Lemma 2), is projected back correctly. Or in other words: the extrinsic projection returns a projection of the exact same two dimensions that we started with.

Through the lens of the extrinsic projection training of a three-ReLU-layer network with 50 filters each is to be inspected. As in the two-filter example, an additional linear layer on top the network maps to four dimensions (due to the number of classes). Figure A.7 shows the extrinsic projection for various stages in training: the initialization of the network (before training), after a single step of training, after ten steps, after 25 steps, after one epoch, after two epochs and finally, after ten epochs of training.
The implicit folding mechanism in this case unfolds as follows: The first layer shows only little deformation of the initial plane (at least in the projection of the two initial dimensions that can be seen). The second layer implicitly rotates the dataset and shows a non-linear "untwisting" of the dataset plane during training and performs a folding of the lower row of classes towards the end of training. The third layer, harder to comprehend, successively collects points of the same class near each other, at least in the projected view shown. While the final layer does not contain any ReLU-activation, it's extrinsic projection can still be considered, implicitly assuming it to be a zeroing operation as well as it effectively chooses four of the 50 dimensions (final column). During training, we see in that last layer that the network is able to arrange the data implicitly in such a way that the respective classes are overlapping when projected down into the final two-dimensional space again. The network achieves a test accuracy of about 93.8% after ten epochs of training, also shown by the back-projected representation in that the classes are distinguished using four linear filters pointing in the direction of the four classes.[4]

[4]Note that we used a simple optimization technique; a constant learning rate without any learning rate decay, no weight decay and also no momentum. The goal of this experiment is to purely show the type of optimization the ReLU-activation implicitly applies to the input space.

A.2.3 Discussion of the Extrinsic Projection

This section has defined the *extrinsic projection* of a linear layer followed by a Rectified Linear Unit (ReLU)-activation. It is a projection of the result of such that retains information, namely when the filters used in the linear layer are linearly independent of each other. Through the lens of this projection, one can view the "zeroing" of ReLU as a deformation of the input space, leaving points as they are if their components are not zeroed. This deformation is *combinatoric* as each dimension that is affected by ReLU affects multiple activation cells (ACs) (that is, the domain in which the same combination of discrete decision occur in the ReLU-activation) and thus changes multiple of the global linear mappings a neural network applies implicitly at once. This deformation is *non-injective*, as multiple values can be mapped to the same spot (unsurprisingly, as ReLU maps all negative values to zero). The deformation of one such non-injective mapping can be seen in the schematic visualization in Figure A.4 – for instance, consider the ACs that are mapped to an edge.

The extrinsic projection presented in this section is a theoretical tool to examine the implicit deformation of a network of the input space: In order to compute the extrinsic projection of any layer l of the network, the intrinsic coordinates $y_l(x)$ of the original network are still needed. This is due to the computation of the ACs given by the matrices $A_l(x)$ require the interim results of x after layer l, $y_l(x)$. The next section, Appendix A.3 will utilize the idea to deform input space using a trick resembling the kernel-trick to construct a new network layer that applies an approximated version of that transformation directly in input space. Effectively, the network performs computations directly in the extrinsic coordinates, and thus, the resulting network type is denoted *extrinsic network*.

A.3 Extrinsic Neural Networks

Appendix A.2 has provided the observation that a reformulation allows to view the operation done by a linear layer followed by a ReLU-activation as a deformation of input space under some conditions. Although seldom met in practice, the key condition required was that the input space's dimension be greater, or equal to, the output space's dimension. This section will expand on that idea and inspect if the condition can be ignored at the cost of approximating the deformation. The idea of the herein defined *extrinsic layer* is to construct a similar operation that applies the ReLU deformation directly to data in input space.
Creating such a layer could have multiple benefits: First, the number of parameters is not limited by the number of interim dimensions for extrinsic layers, as all operations done in high-dimensional space are implicitly projected back into the original space. Second, deforming input space rather than computing new local coordinates in each layer is inherently more stable: the early experiments demonstrate that no batch normalization layers are required and gradients flow more freely throughout the network. The drawback, however, is that deforming input space increases the difficulty of initializing weights optimally.

Recap of the last Section

ACs have been defined in Definition 2 as the disjoint sets where a network applies the same affine transformation. In the context of a given linear network layer in a network f, they can be described using the product of the weight matrix[5], W and the diagonal activation matrix $A(x)$ representing the action of the ReLU-activation on that layer. Restricted to one such cell, the linear layer followed by a ReLU-activation has been reformulated using the extrinsic projection in Equation (A.7) to a function that lives in input space of the whole network. (In this section the last affine transformation that has been named V earlier will be ignored for now, inspecting only the extrinsic projection in isolation):

$$\hat{x} := \text{ReLU}\,(x^T W)W^+ = x^T W A(x) W^+ \tag{A.12}$$

Assuming that W^+ is the true inverse of W, the extrinsic projection changes the coordinate system (named *intrinsic coordinates*), zeroes the by ReLU affected dimensions, and then reverses the change of coordinates back into the original coordinate system (named *extrinsic coordinates*).

[5] Again, this section assumes that the bias is zero, i.e. represented by homogeneous coordinates. How this works out in practice, will be discussed later in this section.

A.3.1 Extrinsic ReLU Layer

As the matrix $A(x)$ is a binary-diagonal matrix, with a 1 in exactly in the dimensions where the ReLU-activation passes data through (positive values), and a 0 otherwise, using the Identity matrix I, this equation can be rewritten to look more like a deformation of x:

$$\begin{aligned}\hat{x} &= x^T W A(x) W^+ \\ &= x + \underbrace{x^T W (A(x) - I) W^+}_{=:dx}\end{aligned} \tag{A.13}$$

Approximating the Deformation

Note that this holds only if $WW^+ = \mathrm{id}$ (see Appendix A.2), which can only be true in general if the input space's dimension is greater or equal to the output space's dimension. However, in practice the input dimension is usually smaller than the output spaces' dimension, which is the number of filters for intermediate layers. Thus, as in the general case $WW^+ \neq \mathrm{id}$ either due to the condition named, or due to W^+ not being the right inverse of W, the resulting deformation given by dx will not result in the same operation that ReLU activations when viewed through the lens of the extrinsic projection (Appendix A.2). One could project W onto the closest matrix $\arg\min_W ||WW^+ - I||$ either during training, or, as part of the initialization, but this step did not provide additional performance benefits. Instead, the idea is to let the deformation be part of the optimization process, removing the pseudo-inverse overall and replacing it with a transposed version of the filter matrix,

$$x \leftarrow x + ds = x + x^T W (A(x) - I) W^T.$$

Thus, detached from the intrinsic linear layer followed by a ReLU activation, the extrinsic idea performs only an approximation of the deformation.

In detail, the deformation dx that the extrinsic projection applies to a point x in input space performs the following steps:

1. Compute the separation of input space into activation cells using the combination of matrix W and ReLU function. $A(x) := \mathrm{ReLU}\left(\mathrm{sign}(x^T \cdot W)\right)$ is the 0-1 vector that represents the activation cell the data point x is contained in. Note that $A(x)$ depends on the results of the previous layer, i.e. constructing the extrinsic deformation perfectly would require to compute the intrinsic network anyways. Instead, another approximation takes place here: considering the density in which activation cells exist for realistic choices of networks (see

Figure 2.12), instead of computing activation cells of higher order that only result from deeper network computations, only a single layer will suffice the separation into activation cells, namely exactly the same that is used for the deformation itself. Note that the gradients do not flow into $A(x)$, but affect the activations through W.

2. Given a separation, $A(x)$, defines then the direction in which a data point is shifted. Note that the number of potential directions is the number of activation cells there are in the specific setup (see Section 2.2.4). This direction is linear in each activation cell but globally non-linear.

3. The *amount* of movement in the direction of dimension i can be determined by inspecting the value $(x^T W)_i$. The movement occurs if $\hat{A}(x) = 1$ and does not occur if $\hat{A}(x) = 0$.

Constructing the Extrinsic Layer

Before defining the extrinsic layer that approximates the back projected view of the ReLU activation, there is one additional mechanism to consider: the bias term. As shown in Section 2.2.4 the bias term plays a crucial role in the number of cells that an otherwise static network with a single hidden layer may have. When observed through the lens of the deformation framework, the extrinsic projection, the bias term has two roles

- It performs finer control over the activation cell separator function that – including the bias b and in Cartesian coordinates – can also be written as $A(x) := \text{ReLU}\left(\text{sign}(x^T \cdot W + b)\right)$.

- A large bias value may pass data near the value zero through to the next layer, without being affected by the ReLU activation (see Section 2.2.4). This effectively performs a "light" form of a residual connection to a subset of the dataset as data is still scaled by the weight matrices.

All these functions will be mimicked as follows: To enable the same number of activation cells the separator function $A(x)$ has with a bias term, the layer defined below increases input space by one dimension, effectively using homogeneous coordinates where the values of the new dimension are set to 1. Note that this additional dimension will be used when stacking multiple such layers as part of the deformation process. The residual connections idea is already included in the deformation idea by design, as stacking multiple such layers effectively only adds new deformations on top of the original data x. What is missing, however, is to steer

the "amount" of deformation, i.e. the residual connection in this network design. Thus, the definition below adds two types of *scale* parameters to the deformation vector dx: one to steer the amount of movement per cell, the other to steer the global amount of movement of the whole deformation of the layer.

In total, the *extrinsic linear layer with ReLU activation* is defined as follows:

Definition 10. *(Extrinsic Linear Layer with ReLU Activation) Given the AC* $A(x) := \mathrm{ReLU}\left(\mathrm{sign}(x^T \cdot W)\right)$ *of the input* x*, the layer computes as:*

$$\begin{aligned} \mathrm{ELRA}_{W,s,s'} &: \mathbb{R}^n \to \mathbb{R}^{n+1}, \\ \mathrm{ELRA}_{W,s,s'}(x) &:= [x,1] + [x,1]W \cdot \left(\tilde{A}\left(x\right) - I\right) \cdot \mathrm{diag}\left(s\right) \cdot W^T \cdot \mathrm{diag}\left(s'\right), \end{aligned}$$

where $W \in \mathbb{R}^{n+1,m}$ *is the filter matrix,* W^T *is it's transposed version,* $s \in \mathbb{R}^m$ *is the* intrinsic shift*,* $s' \in \mathbb{R}^{n+1}$ *is the* extrinsic shift*, and,* $[x,1]$ *denotes the addition of a homogeneous coordinate dimension to the data input.*

Potential benefits of this layer: The benefit of this layer is that the number of parameters in intermediate layers doesn't scale quadratically with the number of filters, as it happens with intrinsic networks. Say, a network with filters $W_l \in \mathbb{R}^{d_l, d_{l+1}}$ has at least

$$\sum_i d_i \cdot d_{l+1}$$

Parameters. Early Networks with large intermediate dimensions (such as VGG introduced by Simonyan and Zisserman [158]) were limited by layers that map between high-dimensional spaces. In addition to that, techniques like bias-vectors or batch normalization require additional parameters (exponential moving average, exponential moving scaling, learned scaling, learned bias) that scale only linearly. In the case of a simple feed-forward network with batch normalization this results in a total number of

$$\sum_i d_i \cdot d_{l+1} + d_{l+1} \cdot 4$$

parameters.

For the extrinsic layer described above, the number of parameters is, in every layer depending on the input space's dimension,

$$d_0 + d_{l+1} + \sum_i d_0 \cdot (l + d_{l+1}).$$

The first term describes the number of the parameters used for the weight matrices W_l, the second term counts the number of the "inner" shift parameter s, and, the last

term counts the number of the "outer" shift parameter s'. (Batch normalization is not needed when using this layer, because data-set normalization already transforms the data-set already into a form that can be handled by the network, and, intermediate representations are only locally moved versions of that.)

As an example, a 1000-layer convolutional network with 64 filters in each layer, 3 input channels, 3×3 kernels, and 1000 output channels (to classify the images of the ImageNet dataset) requires $(3 \cdot 3 \cdot 3) \cdot 64 + 64 \cdot 1000 + \sum_{i=1}^{999} 64 \cdot 64$ weight parameters and $4 \cdot \left(1000 + \sum_{i=0}^{999} 64\right)$ additional parameters used by batch normalization, a total of $4.153.536 + 259.744 = 4.413.280$ parameters.

In contrast, the extrinsic counterpart of this network with the same number of filters, 64 per layer uses $1.845.208$ parameters according to the formula given above. The difference of this number increases more if number of filter increases, or, number of input dimensions decreases.

Limitations of Optimizing in Input Space

Optimizing in Input Space comes with a limitation that cannot be solved without further tools. The idea of moving points in their original space layer by layer collides with the preconditions required for linear layers with a ReLU activation to work well. Section 2.2.4 has shown how the number of weights in a linear layer followed by a ReLU activation separates space into smaller parts. Appendix A.2 has analyzed how this separation is responsible for how many distinct linear functions being used implicitly to map data. That this mechanism works well is based on the fact that input data is more or less normal-distributed across all layers (see [61]). Otherwise, much less of the number of distinct linear functions would be used for the internal mapping. Now, this section has presented a layer that utilizes this separation to move data in input space in a non-linear way. The result of this mechanism is that the distribution of data becomes structured, resulting in more points landing in some ACs, and fewer points landing in other ACs. It follows, that the expressivity, the number of distinct linear functions used becomes smaller with every additional deformation. While in typical neural networks with ReLU activations, minor deviations from the standard normal-distribution can be compensated by batch normalization, but in this new framework, this technique wouldn't help much. Appendix A.1 suggests a simple technique that aims to overcome this problem using activation pattern entropy (APE) maximization.

A.3.2 Extrinsic Network on a Toy Example

The network used in Appendix A.2 to classify the points of the checkboard dataset (see Figure A.5) was a 3 layer fully connected network without normalization, 50 filters per layer and a classification layer with 4 filters to classify each of the four classes.
Appendix A.2 has used the extrinsic projection to visualized how this network *deforms* the input space implicitly. The following experiment adapts the same network to an extrinsic one. To accomplish this, first all linear layers followed by a ReLU activation are replaced by the extrinsic layer given in Definition 10. In order to visualize the kind of deformations this layer can apply, no additional dimension is added to the input space for every layer. Instead, the network is only allowed to deform the 2D input space. Doing that effectively results in a network that does not have a bias term. The number of filter has been adapted so that both networks (the intrinsic 50 filter ReLU network of Figure A.7) and this extrinsic network have approximately the same number of parameters. Thus, the number filters chosen are 600, resulting in a total of

$$((\underbrace{2}_{\text{input dim}} + \underbrace{1}_{\text{intrinsic shift}}) \cdot \underbrace{600}_{\text{number of filters}} + \underbrace{2}_{\text{extrinsic shift}}) \cdot \underbrace{3}_{\text{number of layers}} = 5400$$

parameters. In contrast, the intrinsic 50 filter ReLU network of Figure A.7 has

$$\begin{aligned} &(\underbrace{2}_{\text{input dim}} + \underbrace{1}_{\text{bias term}}) \cdot \underbrace{50}_{\text{number of filters}} \\ +&(\underbrace{50}_{\text{number of filters}} + \underbrace{1}_{\text{bias term}}) \cdot \underbrace{50}_{\text{number of filters}} \cdot \underbrace{2}_{\text{number of hidden layers}} \\ +&(\underbrace{50}_{\text{number of filters}} + \underbrace{1}_{\text{bias term classifier}}) \cdot \underbrace{4}_{\text{number of classifier filters}} = 5454 \end{aligned}$$

parameters.

The resulting training progress is shown in Figure A.8. Note that each layer is only able to use the data of the previous layer and applies one deformation to it. This means, no hidden dimensions are used in this experiment besides the dimensions showed in the visualization, unlike in the ReLU-case shown earlier in Figure A.7 where 50 dimensions have been hidden in the visualization due to being limited to 2D space on a piece of paper. The resulting extrinsic deformation is capable of creating a *non-injective* map that effectively "merges" parts of the input space over the course of 10 epochs. Even though the network defined can only deform input space, the optimization algorithm is able to find a good solution to the problem.

Thus, the input space of the checkboard dataset is sufficient to simulate a small neural network, i.e. more dimensions are not needed for the operations that lead to a solution in this case.

A.3.3 Discussion of Extrinsic Layers

Still, the question remains whether extrinsic networks provide a more or less detailed internal represenation compared to intrinsic networks.
As the number of parameters rises quadratically for intrinsic Rectified Linear Unit (ReLU) networks with increasing numbers of filters the cost benefit (memory wise) would be substantial. However, the operation that the extrinsic variant applies to input space is just an approximation to it's intrinsic counterpart.
The question is, if the number of filters, i.e. the effective number of activation patterns (APs) is the driving force behind the quality of a network, or, the number of parameters overall. As the number of cells may rise exponentially, the resulting calculation may be still similar to that of a intrinsic network if many filters and many layers are used instead, with only small "step-sizes" (i.e. shift values). This fact could mitigate that the activation cells of deep networks become more complex, the deeper the network gets.
On the plus side for the presented extrinsic layer is surely it's stability, optimization-wise. Because it is, by construction, a localized residual operation ($x \leftarrow x + dx$), gradients can flow through the whole network – the same idea that has lead to the success of the ResNet model family.
Major drawback of the method is that it is unclear how to utilize the full potential of a randomly chosen normal-distributed filter-matrix. The intrinsic version could assume, both by clever initialization [61], and, by additionally using batch normalization layers, that the intermediate results are again normal-distributed. This is not true anymore, when moving data points through space, and batch normalization would not help to resolve this limitation. Thus, a different approach needs to be developed that makes sure that activation distributions are of high variance. A method similar to the one presented in Appendix A.1 might be a future direction to handle this or similar problems.

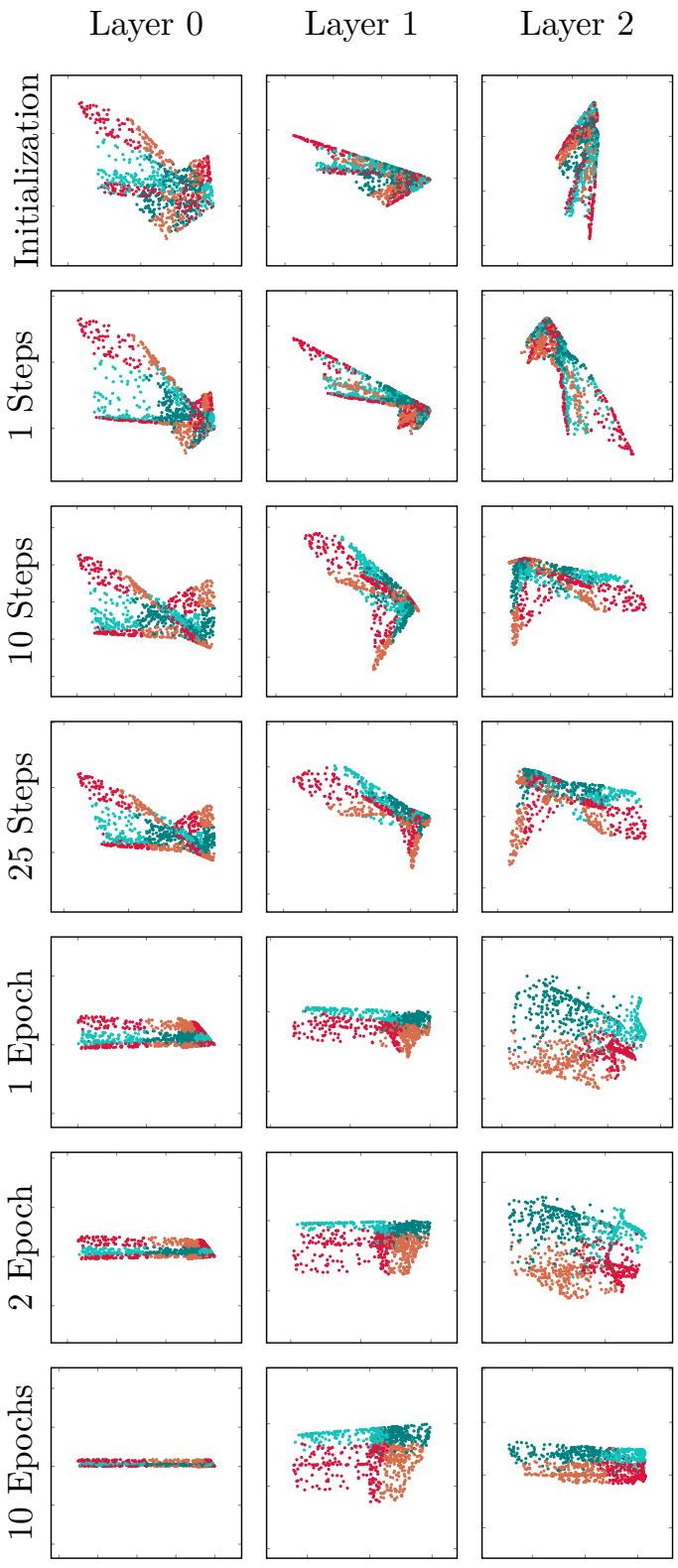


Fig. A.8.: The rows show the course of the deformation over the course of training the network for 10 epochs at several training steps. The columns of the figure indicate the extrinsic projections of the first, the second, the third and the final classification layer of the network.

Parameter	Default	Description
$\eta_{\min}$	10^{-10}	Minimum feasible learning rate η.[6]
remeasure_init	$\{0.25, 0.5, 0.75\}$	Initial probabilities for re-measurements.[7]
initlrs	$\{10^{\ell} : \ell \in \text{linspace}(-4, 2, 25)\}$	Fallback η-candidates.[8]
error_num	5	Number of consecutive large-error events before forcing a full re-fit.
error_muthresh	2.5	Threshold for μ change in a single step to count as large error.
remeasure_jump	2	Multiplicative factor used in the iterative $\_search\_\eta$ procedure.
max_while_loop	100	Upper bound on the number of search iterations.
probing_step_size	8	Carries out an extra verification of the measured probability this given number of optimization steps.

Tab. A.1.: Hyperparameters and Default Values.

A.4 ActCooLR Algorithm

This section provides a more compact and structured representation of the ActCooLR algorithm used for the experiments in Section 3.3. The algorithm is a heuristic for efficiently updating the parameters of the model given in Equation (3.17) that predicts the probability of a AP change and uses that model to define a learning rate scheduler for the training of a neural network.
The challenge that the algorithm addresses is the efficient initialization and updating of the model parameters by minimizing forward passes. The algorithm has two modes of updates: a full update in case of network initialization, frequent contradictory model predictions, and a fast update during training by updating only the parameter μ.

Model Definition:
The following model relates the learning rate η to the probability of a change of a random activation pattern a in a network layer. The model is fully defined by the parameters σ and μ and analysed in Section 3.3.

(a) **Forward model (probability as function of η):**

$$P(\eta;\, \sigma, \mu) \;=\; \frac{1}{2}\Big[1 + \operatorname{erf}\Big(\frac{\log(\eta) - \mu}{\sigma\sqrt{2}}\Big)\Big].$$

(b) **Inverse model (η from probability):**

$$\eta(P;\,\sigma,\mu) \;=\; \exp\Big(\sigma\sqrt{2}\,\mathrm{erfinv}(2P-1) \;+\; \mu\Big).$$

(c) **Partial inverse (solving for μ):**

$$\mu \;=\; \log(\eta) \;-\; \sigma\sqrt{2}\,\mathrm{erfinv}(2P-1).$$

Algorithm Components:

The hyperparameters of the model are listed in Table A.1. The algorithm consists of the following components:

1. **Initialize model parameters:** This is only for a first guess. Working well in practice is the choice:

   $$\sigma \;\leftarrow\; 1, \quad \mu \;\leftarrow\; -1, \quad \eta \;\leftarrow\; \text{undefined initially.}$$

2. **Activation Pattern Temperature (APT) Measurement**
   To estimate the probability of a change in the AP, we need to perform a forward-pass of the model twice: once with the current network state (to get the current APs) and once after the optimization step has been completed. The difference in the APs is then used to estimate the probability of a change.
   When performing a measurement for a learning rate that is not intended to be used for optimization, we need to perform the forward pass twice. However, as a forward pass needs to happen with every optimization step anyways, we can get a measurement (to be used to check if the model is still valid) if we perform only one additional forward pass after the optimization using the same batch. To optimize the total number of additional forward passes further, we perform this additional only every $\mathrm{probing\_step\_size}$ optimization steps.

3. **Bisection-like Search** $\_search\_\eta$
   Upon initialization, and whenever we need to refine the model, we use a bisection-like search to gather additional (η, P)-pairs to cover the outer bounds of the probabilities given by $\mathrm{remeasure\_init}$.
   Suppose we have:

- An initial guess η_{near} (which can be inferred from the initial model parameters and the inverse model formula given above).
- A measured probability P_{near} at η_{near}.
- A target probability P_{exp} that we wish to approach from above/below, most probably the minimum or maximum of remeasure_init.
- A direction ($\text{where} \in \{\min, \max\}$) indicating whether we are bounding the probability from below or above.
- Though not a pure bisection, we repeatedly multiply or divide η by remeasure_jump. This "expand/shrink" approach helps bracket the region where η yields probabilities near P_{exp}. The break condition stops once we cross the desired bound.

1: **Input** *Initial guess* η_{near} with probability P_{near}
2: Initialize *list_lr* $\leftarrow []$, *list_prob* $\leftarrow []$
3: **if** $P_{\text{near}} \neq 0.0 \wedge P_{\text{near}} \neq 1.0$ **then**
4: *list_lr*.append(η_{near})
5: *list_prob*.append(P_{near})
6: **end if**
7: **for** $k = 1 \ldots \text{max\_while\_loop}$ **do**
8: **if** $\eta_{\text{near}} < \eta_{\min}$ **or** $\eta_{\text{near}} > 10^{300}$ **then break**
9: Adjust η_{near} by remeasure_jump :

$$\eta_{\text{near}} \leftarrow \begin{cases} \eta_{\text{near}} \cdot \text{remeasure\_jump}, & \text{if } (P_{\text{near}} \leq P_{\text{exp}} \ \wedge \ P_{\text{near}} \leq P_{\text{exp}}), \\ \eta_{\text{near}} / \text{remeasure\_jump}, & \text{otherwise.} \end{cases}$$

10: $P_{\text{near}} \leftarrow \text{model.measure}(\eta_{\text{near}})$
11: *list_lr*.append(η_{near})
12: *list_prob*.append(P_{near})
13: Check break condition :

$$\text{if } \Big((P_{\text{near}} \leq P_{\text{exp}}) \wedge (\text{where} = \max)\Big) \vee \Big((P_{\text{near}} \geq P_{\text{exp}}) \wedge (\text{where} = \min)\Big),$$

14: **end for**
15: **return** (*list_lr*, *list_prob*)

4. **Model Update (Fit σ, μ)**:
This step is triggered when the model is considered stale, uncertain or upon initialization. To give an overview of this step: First, we retrieve a (η_i, P_i) pairs, either by using the bisection-search of the probabilities given by remeasure_init or if that fails by using the initlrs learning rate grid. Then, we fit the model

parameters σ, μ to these pairs.
The total step of the algorithm is as follows:

(i) *Collect new measurements:* For each $p \in \text{remeasure\_init}$,

$$\eta_i \;=\; \eta(p;\, \sigma, \mu), \quad P_i \;=\; \text{model.measure}(\eta_i).$$

Optionally, include the previous $(\eta_{\text{last}}, P_{\text{last}})$ from the last update step.

(ii) *Remove invalid:* Discard any (η, P) pairs for which $P \le \epsilon$ or $P \ge 1 - \epsilon$ (with, e.g., $\epsilon = 10^{-10}$). This avoids extreme values that lead to $\text{erfinv}(\pm 1)$ and maintains stable $\log(\eta)$.

If none remain, we need to evaluate revert to a global measurement grid initlrs to gather new measurement pairs (η, P).

(iii) *Ensure boundary measurements exist:* (This improves the quality of the model fit significantly.)
Let

$$P_{\min} \;=\; \min_i P_i, \quad P_{\max} \;=\; \max_i P_i.$$

Use $\_search\_\eta(\cdot)$ near the min or max to gather additional (η, P) points for the borders of remeasure_init. Concatenate all valid data.

(iv) *Fit new σ, μ:* Solve

$$(\sigma^*, \mu^*) \;=\; \text{fit}(\{\eta_i\}, \{P_i\}, \text{optional initial guess})$$

(e.g., regression or MLE).
A practical and stable approach involves employing Singular Value Decomposition (SVD) to solve the system of linear equations in a least-squares sense, truncating the singular values at a threshold of 10.
Finally, $\sigma \leftarrow \sigma^*, \mu \leftarrow \mu^*$.

5. **Probability/Temperature Scheduling:** (Choosing η Before Optimization Steps).

- Suppose we want to use a linear schedule of the probability,

$$P_{\text{wish}}(t) \;=\; \left(1 - \frac{t}{T}\right) \quad \text{for step } t, \text{ total } T.$$

Instead of solving the inverse model for $P_{\text{wish}}(t)$, we solve for a slightly smaller probability, for example $0.99 \cdot P_{\text{wish}}(t)$. This prevents aiming for $P_{\text{wish}} = 1.0$ (where $\text{erfinv}(1)$ diverges) and keeps us below the extreme tail.

- **Compute** η.

$$\eta \;=\; \exp\Big(\sigma\sqrt{2}\,\mathrm{erfinv}\big(2\,P_{\text{wish}}(t)-1\big)+\mu\Big).$$

  If η is above, say, 10^{10} or NaN, treat that as out-of-range and exit the program. This indicates a diverging training run.

- If the learning rate η is less than the minimum learning rate $\eta_{\min}$, set it to $\eta_{\min}$. This is because a learning rate of zero is not meaningful, as it would prevent the network from being optimized.

6. **After Forward Pass (Incremental μ Update)**: Suppose after a training step, we measure $(\eta_{\text{now}}, P_{\text{now}})$. (This is done every $\mathrm{probing\_step\_size}$ steps to optimize the number of forward passes.)

   (i) If $P_{\text{now}} \in \{0.0, 1.0\}$, discard that data and trigger a full model update (Step 4). These extremes break the normal approximation and primarily occur because of insufficient data utilized per batch for measurements or an inadequate number of filters in the network.

   (ii) Otherwise, compute the new μ. (This assumes that the model's σ changes slower than μ, which has been shown in the main text to be the case.)

$$\mu_{\text{new}} \;=\; \log(\eta_{\text{now}}) \;-\; \sigma\sqrt{2}\,\mathrm{erfinv}\big(2\,P_{\text{now}}-1\big).$$

   If $|\mu_{\text{new}} - \mu| > \mathrm{error\_muthresh}$, increment a counter.

   (iii) Once the counter hits $\mathrm{error\_num}$, do a full model update (Step 4). Otherwise, set $\mu \leftarrow \mu_{\text{new}}$.

   This "fast path" update quickly re-centers μ if the measured probability was close to the model's prediction, without re-fitting σ.

Colophon

This thesis was typeset with LaTeX 2$_\varepsilon$. It uses the *Clean Thesis* style developed by Ricardo Langner. The design of the *Clean Thesis* style is inspired by user guide documents from Apple Inc.

Download the *Clean Thesis* style at `http://cleanthesis.der-ric.de/`.

Declaration

I hereby declare that I have written the present thesis independently and without use of other than the indicated means. I also declare that to the best of my knowledge all passages taken from published and unpublished sources have been referenced. The thesis has not been submitted for evaluation to any other examining authority, nor has it been published in any form whatsoever. I duly noted the Regulations for Good Scientific Practice and Dealing with Scientific Misconduct.

Mainz, February 19, 2025

David Hartmann