Efficient Deep Learning

Anders Øland

CMU-CS-24-147
August 2024

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

*Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.*

This research was sponsored by the Department of Interior under award number D11PC20068 and Adobe.

Keywords: deep learning, machine learning, neural networks, ai, efficiency, distributed learning, properties of hidden layers and representations, layer-wise learning, partition-wise learning, implicit regularization, interlayer regularization, data redundancy, data filtering, platonic projections, closed-form solution

For my father
Poul Emil Øland
(1945-2020)

Abstract

It is well-known that training deep neural networks is a computationally intensive process. Given the proven utility of deep learning, efficiency is thus an important concern. We study various aspects of deep networks with the goal of making the training process more efficient with respect to memory consumption and execution time. In doing so, we also discuss and contribute to various theoretical facets of the field.

An unavoidable component in dealing with very large models is the need for distributing the training over many, sometimes hundreds or thousands, computational devices. This usually induces a considerable communication overhead that increases the risk of under-utilization of the system resources. To address this, we show that the entropy of the weights decreases during training, which thus become highly compressible—allowing for a considerable reduction in said overhead.

It is common practice to use squashing functions, like the softmax, at the output layer of neural nets. We study the effect these functions have on the gradient signal and argue that they may contribute to the well-known vanishing gradient problem. To this end, we introduce non-squashing alternatives and provide evidence that suggests, that they improve the convergence rate.

Our main contribution is in layer-wise training of deep networks. First, we make various useful observations on the properties of hidden layers and representations. Motivated by those, we show that layer-wise training can match the results of full-model backprop, while considerably reducing the memory footprint of the training process. We discuss the effect of implicit interlayer regularization (AKA the implicit bias of depth) and introduce new conjectures on its theoretical origin. Based on these, we show that interlayer regularization can be simulated in a few simple steps. Additionally, we introduce partition-wise training, which may speed up the optimization process by allowing for larger batch sizes and improved model parallelism.

Finally, we take a look beyond gradient descent. Drawing on understanding gained on how and what neural nets learn, a novel solution to fitting multilayer perceptrons to training data is introduced. While it can outperform backpropagation with stochastic gradient descent on various toy problems, it tends to overfit and be capacity-hungry on more complex real data. We discuss why, and point to future ways of addressing this. This solution can be expressed in closed form, albeit we expect that it will evolve into a hybrid iterative approach. Also, we suspect that our method might be a substantially better candidate for training deep nets on quantum computers than backprop.

Acknowledgements

First and foremost, I want to express my deepest gratitude to my advisers Roger Dannenberg and Bhiksha Raj. I truly feel like I won the adviser lottery. Without their friendship, support, and mentorship over the years, this thesis would have never been completed. I came to CMU to work with Roger and Bhiksha on applying deep learning to problems in music production. We wanted to invent Computational Music Production (CMP) as a new field. Alas, we could not find the funding, and I spent most of my time on trying to get or synthesize good data—rather than actually doing any research. After a couple of years of frustration, I finally caved in and changed my research direction to methods for Efficient Deep Learning. While I am overall quite happy with the present thesis, I cannot say that I am 100% convinced that I made the right choice. A certain sadness still lingers that I gave up on my big project. Who knows where our CMP research would have been today if I had stuck with it? Furthermore, a special thank you must go to professors Rita Singh and Rich Stern for their help and advice throughout the years.

Next, I wish to declare my deep appreciation and love for my department. Besides providing an extraordinary academic environment, CSD has been a warm and welcoming home for me. I want to thank our wonderful staff. In particular, Deb Cavlovich, without whom I would have been utterly lost in countless situations. Also, a big shout-out to Catherine Copetas, Marcella Baker, Jenn Landefeld, and Matt Stewart. And to Srini Seshan, Karl Crary, and Dean Martial Hebert for allowing me to graduate after my rather extended leave of absence. Many professors have been a vital part of my CSD journey. To mention a few: Dave Eckhardt, Tuomas Sandholm, Dave Anderson, Stephen Brookes, Greg Kesden, and Mor Harchol-Balter. A special mention goes to Lenore & Manuel Blum, whose kindness and support has left a lasting impression.

From the Computer Music Group (and associates), I wish to thank Gus Xia, Shuqi Dai, Jorge Sarstre, Riccardo Schulz, Jesse Stiles, and Chris Donahue for their friendship and collaboration. From the many many members and friends of the MLSP group, I cannot possibly mention all. Still, I do want to express my gratitude and appreciation to Abelino "Abelito" Jimenez, Anurag Kumar, Nikolas Wolfe, Mark Harvilla, James Baker, Varun Gupta, and Mahmoud Alismail.

Among the many amazing fellow CMU students (graduate and undergraduate) I have encountered along the way, I must give a special shout-out to Christian Kroer (and Janine), Max Illfelder, Gaurav Marnek, Zeyu Jin, Aayush Bansal, Kevin Waugh, Ashique KhudaBukhsh, and Aram Ebtekar.

While at the IT University of Copenhagen, a few key people crossed my path. Most importantly, I owe a huge debt of gratitude to Joseph Kiniry for putting me on the path toward a Ph.D. and guiding me through the whole process of applying to positions in the US. I also wish to thank Dan Witzner, Rasmus Pagh, Lars Birkedal, Peter Sestoft, Kasper Videbæk—and, of course, my fellow A-Team members: Anders Høst and Anders Bech Mellson.

My Pittsburgh experience was particularly brightened by an eclectic bunch of wonderful people. First of all my Jitters crew: Matt Kessler, Wale Williams, Lucas Marsak, Harry Frazee, and Avi and Eli Wenger. And also, Gabriel Guzman, Frances Dannenberg, Caroline Jeselnik, my beautiful puppy Alfred, and the best neighbor ever, Denise Rousseau.

Perhaps my biggest regret is not finishing this thesis in time for my father to see me graduate. I know he would have been immensely proud. Fortunately, the rest of my family is still here to celebrate with me. From the bottom of my heart, thank you: mom, Henriette, Claus, Anton, Kaya, Benjamin, and Mariana. Additionally, I absolute must say thank you to my second family in Vanløse: Las, Tina, Ester, and Isak—for being a home away from home for me (and much more).

Finally, a big hug to Anders Christian Wolf Kølle for re-igniting my interest in machine learning at just the right time. And, of course, to ”firkløveret”: Monica Gregers Smith, Niels Bjørndahl, and Tommy Ahlers (you know why).

Last but not least, I want to express my gratitude to the members of my committee for their valuable time and advice.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The recent second (or third) coming of neural networks (NNs), now known as deep learning (DL), has undeniably yielded some impressive results in computer vision, natural language processing, generative AI, and multiple other fields. This has prompted some to name AI the fourth industrial revolution. Thus, massive amounts of effort, money, and energy are pouring into the field, with deep learning by far receiving most of the attention. This means, that a focus on efficiency is now more important than ever.

Training a single large neural network, on very large data, can potentially consume so much energy, that hundreds of tons of CO_2 will be released into the atmosphere (if the energy source is e.g. a coal power plant). Thus, AI as a whole, has a daunting potential for exacerbating the ongoing climate crisis. Another important aspect is that of cost. Due to the ever-increasing sizes of the models and the data, and the following need for very large compute, *training state-of-the-art neural nets have become prohibitively expensive.* This leaves the power of this disruptive technology in the hands of the few and wealthy. To their credit, some large AI-driven companies both contribute greatly to the research, and are prone to sharing their models and tools in an open source manner. However, it is still extremely hard (if not impossible) for smaller companies and actors in the field to compete. It simply not a level playing field, when the cost of training a single model can be in the tens of millions of dollars. In the long term, this poses a potential threat of monopolization—and even serious challenges to Western democracies. That is: too much power and influence in the hands of a few people (or corporations), that were *not* democratically elected.

While inference and deployment of AI models is also important, in this thesis, we focus on the *training* of deep networks. Our main focus is on new methods and new directions for deep learning rather than an in-depth analysis of energy and cost optimization. Our consideration of energy and cost will thus be limited to the consideration of memory consumption and execution time. The field is changing rapidly, both in terms of machine learning models and the hardware that implements them. We will present some new results that we believe will enable more cost- and energy-efficient training, but we are not claiming to offer complete solutions. Instead, we will focus on motivating the development of these methods and careful experiments to show that these methods work on interesting problems.

1.1.1 Our Focus

The past decade, or more, has seen an explosion in methods, architectures, and algorithms related to deep neural networks. Given the wide range of learning models and techniques, in this work we limit our main focus to strictly feed-forward (non-recurrent) models with fully-connected and/or convolutional layers trained with backpropagation [127] and stochastic gradient descent (SGD) with momentum. When relevant, we briefly expand our focus to include other methods or architectures, such as ADAM [81] and *transformers* [30, 118, 146]. Likewise, we limit our focus mainly to computer vision. We use well-known standard benchmarks such as MNIST, SVHN, CIFAR-10, CIFAR-100, and the ImageNet 2012 datasets [26, 28, 90, 114].

1.2 Chapters

In the following chapters, we study various aspects related to the efficiency of training deep neural networks. Below, we will briefly describe the contributions of each of the thesis chapters.

1.2.1 Speeding Up Distributed Learning (Ch. 3)

Large-scale distributed learning plays an ever-more increasing role in modern computing. However, whether using a compute cluster with thousands of nodes, or a single multi-GPU machine, communication is a very significant bottleneck. In this chapter, we explore the effects of applying quantization and encoding to the parameters of distributed models.

We show that, for a neural network, this can be done—without slowing down the convergence, or hurting the generalization of the model. In fact, in our experiments we were able to reduce the communication overhead by nearly an order of magnitude—while actually improving the generalization accuracy.

The underlying observation is, that the entropy of the weights decreases during training. Consequently, they become increasingly compressible as the training progresses. This can be exploited for quantization and Huffman coding [68].

1.2.2 Gradient Amplification (Ch. 4)

We show that saturating output activation functions, such as the softmax, impede learning on a number of standard classification tasks. Moreover, we present results showing that the utility of softmax does not stem from the normalization, as some have speculated. In fact, the normalization makes things worse. Rather, the advantage is in the exponentiation of error gradients. This exponential gradient amplification is shown to speed up convergence and improve generalization. To this end, we demonstrate faster convergence and better performance on diverse classification tasks: image classification using CIFAR-10 and ImageNet, and semantic segmentation using PASCAL VOC 2012. In the latter case, using the state-of-the-art neural network architecture, the model converged 33% faster with our method than with the standard softmax activation, and that with a slightly better performance to boot.

1.2.3 How & What Neural Networks Learn (Ch. 5)

In order to aid the development of new and more efficient methods for training deep neural networks, we observe and discuss some useful properties of hidden layers and representations. Historically, DNNs have largely been considered a black box technology. That is, we may know *how* to train them, but our understanding of *what* they learn (in the hidden layers) has been somewhat vague.

One common view is, that the hidden representations form a *feature hierarchy*. For convolutional neural nets (CNNs) trained on images, this means that the lowest layers capture simple features like lines, while upper layers capture more complex concepts (e.g. shapes or scenes). This may be true, but it is arguably not very useful with respect to developing new methods of training.

To this end, we illustrate how the classes gradually separate, layer by layer, in a trained neural net. This observation has multiple interesting consequences and interpretations that are indeed useful to our purpose. It means, that examples belonging to the same class will tend to cluster together, leading to the gradual forming of larger and larger regions that are *homogeneous*, or *pure*, with respect to the class label. This corresponds to a gradual *linearization*, which we can observe and quantify by solving linear systems of equations on the hidden representations.

Moreover, examples that lie near the centers of such pure regions are more likely to *redundant* with respect to correctly learning the decision boundary. As a consequence, we can filter them away before and during training. To do this, we use a simple k-nearest neighbors approach, and discard the examples whose k neighbors belong to its own class. In addition, there is an obvious interpretation of this in information theory.

Lastly, we describe our *correlation filter view* on neural nets. We explain how the fundamental process employed by the layers is that of *sorting and filtering* the input data. Examples are first sorted with respect to their correlation with the weight vectors, and then filtered by the activation functions—such that same-class examples line up next to each other, as much as possible, along each of the feature-dimensions. It is through this process that the layers gradually, from the bottom up, separate the classes.

1.2.4 Deep Layer-wise Learning (Ch. 6)

While the dominant trend in deep learning literature is training very large models on hundreds or thousands of GPUs, there is an increasingly important use-case for mobile and embedded systems. Such systems are typically significantly resource limited, both in compute and memory capacity. To address the latter, we propose a simple method for deep layer-wise learning (DLL). In brief, we simply extract an individual layer (or e.g. a residual block) from an architecture, attach a decision-layer (or block) on top, and train that subnetwork in the usual way with backpropagation and SGD. In the simplest case, we train each layer individually, from the bottom up, for n epochs.

DLL matches the results achieved with regular full-network backprop on a number of standard computer vision benchmarks, while significantly reducing the memory footprint during training (by 70% in most of our experiments). This constructive training scheme allows for the incremental addition of layers, such that the final architecture need not be known in advance; we hope, that this may lead to new dynamic methods for architecture search.

As a secondary contribution, we show that deep partition-wise learning (DPL) enables the use of both larger batch sizes, and better model-parallelization schemes. When training deep nets on the ImageNet dataset, we achieved speedups of 30-55% (279% in one severely memory-sensitive instance). We suspect, that this method could have a potentially big impact on the future of very large scale deep learning. If, for example, our results can be successfully replicated in other fields, like natural language processing. Even a 20% speedup on the training process could translate into millions of dollars (and tons of CO_2) saved.

Furthermore, we observe and analyze the effect of implicit interlayer regularization; i.e. that depth regularizes. This phenomenon presents a challenge to layer-wise training on certain benchmarks, because we train very shallow networks. However, we find that interlayer regularization can be efficiently simulated in a few simple steps. Our work thus has theoretical implications, by adding new understanding of how and what deep neural networks learn.

1.2.5 Beyond Gradient Descent: Platonic Projections (Ch. 7)

With our improved understanding of the inner workings of neural networks, gained in the previous chapters, we take a look beyond the confines of gradient descent and error-backpropagation. Thus, we develop a quite elegant, and surprisingly simple, way of fitting multilayer perceptrons (MLPs) to training data. By following the principle of gradual class separation, we show that computing reasonable targets for the hidden layers is easy. *Specifically, a good target is a representation in which the classes are already separated.*

For a supervised learning problem, given a dataset $\mathcal{D}(X, Y)$, we can simply construct such a target representation from the *augmented matrix*, $\left[\begin{array}{c|c} X & Y \end{array}\right]$. Consider a binary classification problem with $x \in \mathbb{R}^2$ and $y \in \{-1, +1\}$. By augmenting each example with its corresponding class label, y, we are embedding the two-dimensional input in a three-dimensional space in which the two classes are perfectly separated in the third dimension; i.e. by the $x_1 \times x_2$ plane.

Now, if the first hidden layer in our MLP has $n > 3$ features, we can simply *project* the augmented matrix up to n dimensions to obtain our target representation: $T_1 = \left[\begin{array}{c|c} X & Y \end{array}\right] P_1$, where $P_1 \in \mathbb{R}^{3\times n}$ is e.g. a dense random matrix. *Such a projection of the augmented matrix, is what we have named a **Platonic Projection**.* The method owes its name to Plato's famous "Allegory of The Cave", that provided the inspiration for a thought experiment that lead to the proposed method.

In order to fit the first layer of our MLP, we simply solve a linear system of equations, such that $W_1 = X^+ T_1$. Now, to solve the next layer, we repeat the process by letting $Z_1 = \sigma(XW_1)$, and computing $W_2 = Z_1^+ T_2$ where the new target, T_2, is the Platonic projection $T_2 = \left[\begin{array}{c|c} Z_1 & Y \end{array}\right] P_2$. We can repeat this process for the following layers as well. Thus, the complete solution can be expressed by the recurrence relation:

$$W_i = Z_{i-1}^+ T_i$$

where $T_i = \left[\begin{array}{c|c} Z_{i-1} & Y \end{array}\right] P_i$, $Z_0 = X$, and $Z_i = \sigma(Z_{i-1} W_i)$.

Obviously, the quality of the obtained solution depends heavily on the choice of the projection matrices, $P_i \in \mathbb{R}^{m\times n}$. For example, what happens when $m > n$, and the projection reduces the number of dimensions? Clearly, not just any random dense matrix, P_i, will guarantee that the classes are still separated. *Moreover, while this simple method can outperform backpropagation with stochastic gradient descent on various toy problems, it tends to overfit and be capacity-hungry on more complex real data.* These, and related issues, are discussed in the chapter. We offer some possible answers, but this method is still very much *work in progress*. So, to be clear: **we are not making any strong claims, that the proposed method is better than existing methods!** Merely, we see this as a promising new way forward in the pursuit of more efficient ways of training deep networks.

Our method, as presented here, can be expressed in closed form. However, we do expect that it will evolve into a hybrid iterative approach. This would both make it more practical, and likely play a role in addressing the overfitting. Finally, we suspect that our method might be a substantially better candidate for training deep nets on **quantum computers** than backprop.

Chapter 2

Neural Networks & Notation

A basic feed-forward neural network, AKA multi-layer perceptron (MLP), is a nested function consisting of a sequence of pairs of affine transformations and non-linear functions; each pair is referred to as a layer. For example, a two-layer MLP would be $y = f_2 \circ f_1 = f_2(f_1(x; W_1, b_1); W_2, b_2)$, where each layer f_i is $f_i(x; W_i, b_i, \sigma_i) = \sigma_i(xW_i + b_i)$. The matrices W_i and the vectors b_i are called the weights and the bias of a layer, respectively. The functions $\sigma_i(\cdot)$ are generally non-linear, and referred to as the activation function (AKA the non-linearity or squasher) of a layer. A NN with k layers is thus a $(k-1)$-nested function $y = f_k \circ f_{k-1} \circ \cdots \circ f_1$ that, when trained, learns some mapping from the set of inputs X to the target outputs Y.

Figure 2.1 shows a 3-4-2 MLP. That is, a two-layer neural network (we don't include the input layer in the count) with three inputs, one hidden layer with four neurons, and two outputs; we say that the width of the input layer is 3, the width of the hidden layer is 4, and the width of the output layer is 2. To this NN, a single input example x would be a 1×3 vector, W_1 a 3×4 matrix, b_1 a 1×4 vector, W_2 a 4×2 matrix, b_2 a 1×2 vector, and the output y a 1×2 vector.

It is standard notation in the literature to omit the bias term which is then assumed to be captured by adding an extra component of 1 to each input vector, such that $x = \{1, x_1, \ldots, x_n\}$. We will adopt this notation from here on. Thus, each layer is now defined by $f_i(x; W_i, \sigma_i) = \sigma_i(xW_i)$.

Definition 1 (Neural Network)**.** *A k-layer feed-forward neural network (NN) with layer widths, $d_i \in \mathbb{N}, i = 1 \ldots k$, is a nested function $f : \mathbb{R}^n \mapsto \mathbb{R}^p$ defined by the recurrence relation*

$$f_i = \sigma_i(\tau_i(f_{i-1})), f_0 = x$$

where $\sigma_i : \mathbb{R} \mapsto \mathbb{R}$ is generally non-linear, $\tau_i : \mathbb{R}^{d_{i-1}} \mapsto \mathbb{R}^{d_i}$ is a linear transformation, and $d_0 = n, d_k = p$.

For so-called fully-connected layers, τ_i is an affine transformation. In a convolutional neural network (CNN) the τ_i's can also be the convolution or pooling operations.

In our discussion, we shall be referring to the data representations in a NN:

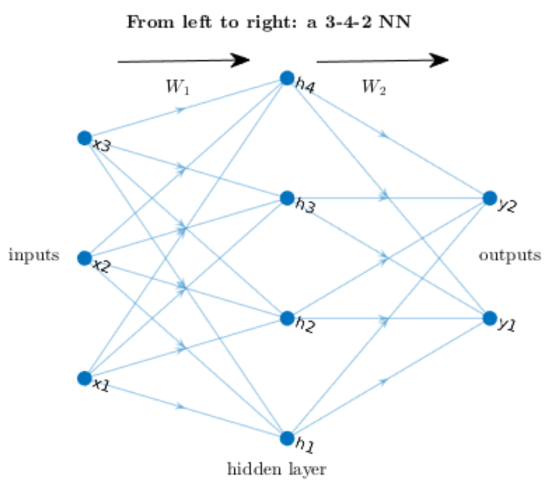


Figure 2.1: A 3-4-2 Feed-Forward Neural Network

Definition 2 (NN Representations)**.** *In a k-layer neural network, $f : X \mapsto \hat{Y}$, the representation $A_i = \tau_i(f_{i-1})$ is the preimage of σ_i, and the representation $Z_i = \sigma_i(A_i)$ is the image of $\sigma_i, i = 0 \ldots k$, where $Z_0 = A_0 = X$ and $Z_k = \hat{Y}$.*

Thus, for a NN we have the hidden representations (i.e. $0 < i < k$), $A_i = Z_{i-1}W_i$ and $Z_i = \sigma_i(A_i)$.

We will also be referring to the *architecture* of a neural network. That is, the set of all properties requd to sufficiently describe a particular network, such as the number of neurons and the activation functions for each layer. This is the mathematical equivalent to the Protocol Buffer- and YAML-based formats used by many deep learning software frameworks. For example, the architecture, Ψ, for our 3-4-2 MLP above would be $\Psi = (\{4, \sigma_1\}, \{2, \sigma_2\})$. Here, and for all NNs, the linear transformation associated with each layer is implied, namely the affine transformation. For more complex architectures, involving e.g. convolutions, pooling, or recurrences, it would need to be specified.

Definition 3 (Architecture)**.** *The architecture of a k-layer neural network is the tuple $\Psi = (\psi_1, \psi_2, \ldots, \psi_k)$, where each $\psi_i \in \Psi$ is the set of all properties associated with the i'th layer.*

We will not concern ourselves with listing all the possible layer types and their associated properties. We will simply assume that all relevant properties are included in any given configuration. Obviously, we will also need to talk about learning problems. Hence, for the sake of consistency and nomenclature, let us make clear what exactly we mean by that.

Definition 4 (Learning Problem)**.** *Given a dataset $\mathcal{D}(X, Y), X \in \mathbb{R}^{m \times n}, Y \in \mathbb{R}^{m \times p}$, and a loss function, $\mathcal{L}(Y, \hat{Y})$, the task of approximating a function $f : X \mapsto \hat{Y}$ such that $\mathcal{L}$ is minimized is referred to as a learning problem.*

We could say that the dataset $\mathcal{D}$ *captures* the learning problem. In the following, we will use the terms dataset and learning problem interchangeably.

Chapter 3

Speeding Up Distributed Learning

Large-scale distributed learning plays an ever-more increasing role in modern computing. However, whether using a compute cluster with thousands of nodes, or a single multi-GPU machine, the most significant bottleneck is that of *communication*. In particular, communication between parallel processes. In this work, we explore and compare different approaches to quantizing and encoding the parameters of distributed models. We show that, for gradient descent-based learning algorithms, this *can* be done - without slowing down the convergence, or hurting the generalization of the model. As a result, the time required for training may be reduced significantly.

With the sizes of today's datasets growing to giga- and petabytes, we see an increasing need for distributed solutions to learning problems. In particular, in the deep learning literature, we have seen the number of parameters in a single model reach the billions. This renders training on a single node infeasible – both with respect to memory requirements and execution time. It becomes necessary, and indeed imperative, to distribute the computation across many machines. However, distributed computing requires communication: data and model parameters must be exchanged between the worker machines that collaborate on the computation. This communication overhead can become a serious bottleneck and greatly slow down the learning.

Communication overhead arises both from the need to transfer *data* to the workers, and the need to share model parameters among them. In our work we focus on the latter, minimizing the communication overhead due to exchange of model parameters. Specifically, we focus on gradient-based methods in a *data-parallel* setting; *i.e.* where the dataset is distributed across workers, and each worker has a complete copy of the model. During training, each worker locally updates its own model, and the updated model parameters must periodically be synchronized among all of the workers.

The general approach to minimizing communication overhead has been to simply reduce the *number* of parameters that are exchanged, *e.g.* by having fewer parameters in the first place by sparsifying them, exchanging only some of the parameters, or by sending them less frequently [24, 27]. We take a different approach – rather than impoverishing the number of parameters exchanged, we reduce the communication overhead by reducing the *number of bits* used to transmit each parameter. We achieve this by quantizing the transmitted values. The Shannon entropy [132] of the parameters of the network varies with training epoch and the layer of the network

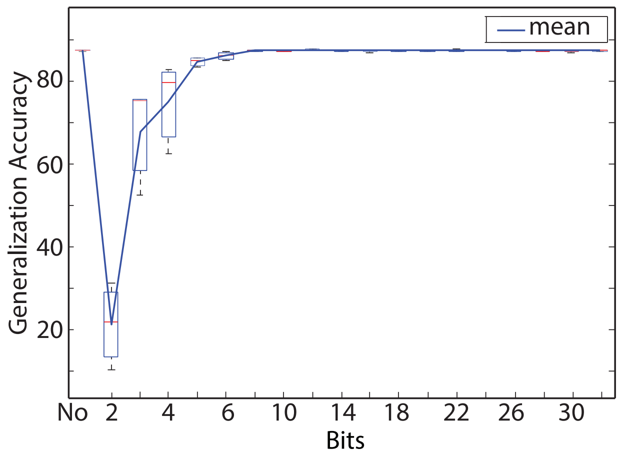


Figure 3.1: Generalization accuracy vs. quantization level of weights.

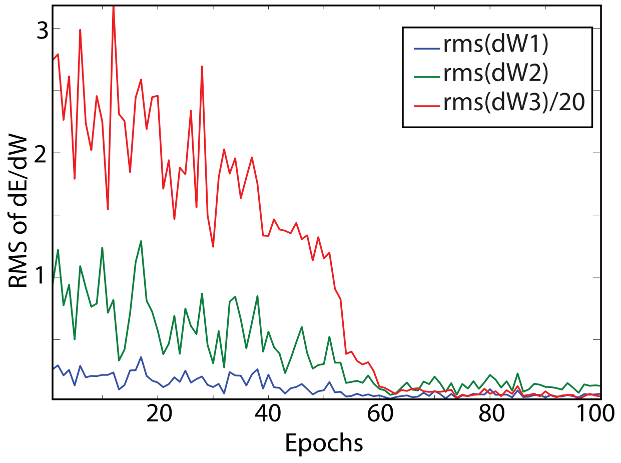


Figure 3.2: RMS of MSE derivative w.r.t. weights vs. training epoch.

that they represent. To take advantage of this, we vary the number of bits used to quantize any value dynamically, based on the observed entropy of the parameter values in any layer, at any epoch. Additional compression is obtained through Huffman coding [68] of the parameters prior to transmission. The added overhead of conveying the information about the quantization levels and Huffman code dictionaries, so that the machines receiving these communicated values can decode them, is insignificant compared to the actual number of bits needed to represent the parameters. We are thus able to achieve a compression of nearly an order of magnitude in the communication overhead, for no loss of generalization error in the trained network, as evaluated on a standard classification task.

Figure 3.1 shows the achieved classification accuracy as a function of the bit rate of the weights. When we go beyond 6-7 bits we do not gain anything in the precision of the learned classifier. In the process, we also achieve a second, somewhat surprising result. The introduction of quantization noise appears to have a beneficial effect on the training. When the parameters are quantized to a slightly higher bit rate than that required to maintain generalization error, the resulting network actually achieves lower generalization error than that obtained with unquantized transmission of parameters. Moreover, the training often appears to converge faster. In effect, we simultaneously achieve reduced communication overhead, improved generalization error, and faster convergence by quantizing the parameters for communication.

3.1 Parameter Compression

Our solution to reducing communication overhead is to compress parameters before transmission. In choosing a compression scheme, we must consider many factors including cost (complexity and execution time), memory usage, effectiveness (compression ratio), and impact on learning (the generalization error of the model). Obviously, the cost of encoding, sending, and decoding the data must be less than simply sending the data as is. Similarly, cutting down the cost of communication is meaningless if it affects the learning negatively, such that either the generalization error increases, or it takes considerably more iterations to converge (so that nothing is gained with respect to the total execution time). The model parameters to be transmitted are single- or double-precision floating point numbers. We find simple lossless compression of these parameters through algorithms such as LZW [157] to generally be ineffective. They achieve little, if any, reduction in the size of the data, at a significant computational cost. Instead, we utilize the robustness of model parameters to minor perturbation, particularly during training, to compose an inexpensive lossy compression scheme through quantization. Secondly, we utilize the relatively narrow range within which most parameter values lie to further compress the parameters through a lossless compression scheme

All illustrations are based on experiments on the MNIST dataset [88] with a deep neural network comprising 784 units in the input layer and three subsequent layers of size 392, 50 and 10 neurons respectively. The network is fully connected between any two consecutive layers. There are thus three sets of weights of size 784×392, 392×50 and 50×10 respectively. The activation functions of all neurons were tanh non-linearities.

3.2 Quantizing the Parameters

As a first try, we attempted to simply quantize all parameters to a fixed number of bits. To quantize the values to N bits, we find the largest and minimum values of all weights in each epoch, and evenly split the range up into 2^N bins. Each weight is quantized to the center of the bin that it falls into. In terms of communication, it will require only N bits to transmit any weight.

Figure 3.1 shows results obtained with quantization to various levels. The plot shows the classification accuracy obtained with the fully trained network, as a function of the number of bits used to quantize the weights. The boxes show the mean, median (red line) and standard deviation of the results obtained from 1500 runs of training with different initializations. The accuracy obtained with 8-bit quantization is comparable to that obtained without quantization. This compares very favorably with the 32 bits typically required to represent floating point numbers in IEEE format.

3.3 Dynamic Selection of Quantization Levels

As network training progresses, the derivative of the mean squared error (MSE) being minimized with respect to the weights converges towards a small value approaching zero, albeit in a somewhat noisy manner. Figure 3.2 shows the root-mean squared (RMS) value of the derivatives of the error with respect to network weights for each of the three layers in our network as a function of epoch. The derivatives for the third layer are scaled by 20 to fit in the plot. We note that the derivatives converge towards zero for all layers. Also, the derivatives are generally larger at higher layers of the network.

The derivatives are directly indicative of the degree of quantization that can be tolerated by the weights. When the derivatives are large, relatively small perturbations of weights can result in relatively large changes in the error. On the other hand, when the derivatives are small, the network is tolerant to larger perturbations of the weights. We also know that quantization to a larger number of bits results in smaller expected quantization error, while quantization to a fewer number of bits results in larger quantization error.

These observations together suggest that when the RMS value of the derivative is large, a larger number of bits are required to quantize the weights. On the other hand, at small RMS values of the derivative, a smaller number of quantization levels will suffice. This leads us to propose a *dynamic* selection of the number of bits to quantize the weights. In each epoch, for each layer, we will choose the number of bits to quantize the weights in that layer according to the RMS value of the derivative of the MSE with respect to the weights. Specifically, assuming the tolerance to changes in the MSE to be some constant T, and the RMS value of the derivatives to be G, the bin size ΔW that may be expected to perturb the objective to within T is given by $\Delta W \propto T/(G + c_0)$, where c_0 is a floor that enforces an upper bound on ΔW. Consequently, the number of bins that the weights are quantized to, $N_{bin} \propto \Delta W^{-1} \Rightarrow N_{bin} \propto (G + c_0)$. The number of bits required to quantize the weights comes out to $N = \log N_{bin} = \log(G + c_0) + c_1$. c_1 may be viewed as a floor on the number of bits used to quantize the weights. To ensure that N does not go below this value, c_0 must be set to 1.0.

3.4 Lossless Compression

The distribution of the weights is not uniform in any layer. Figure 3.3 shows the estimated entropy of the network weights in the various layer of the network, as a function of epoch. In each case, for N-bit quantization we have computed the entropy from the normalized histogram of the weights over the 2^N quantization bins. The four panels shows the entropy obtained with different levels of quantization of the weights. Expectedly, increasing the number of bits used to quantize the weights increases their entropy; however the overall trend of the entropy remains the same in all cases.

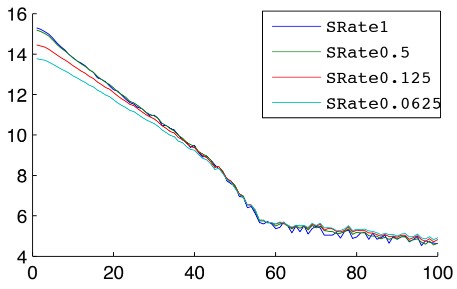
The entropy is generally less than N, the number of bits used to quantize the weights, and decreases with the epochs. This indicates that lossless Huffman coding of the quantized weights can result in significant compression of the weights.

3.5 Bits Required to Quantize the Weights

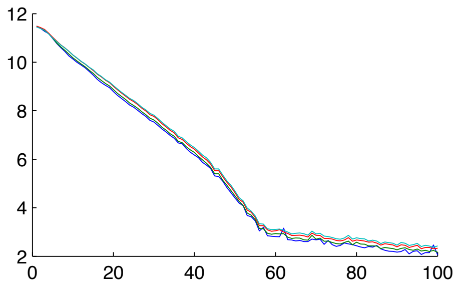
Comparing Figure 3.3 and Figure 3.2 we note that the entropy is well correlated with the RMS value of the gradient. Empirically, we find the normalized correlation between $\log(G + c_0)$ for $c_0 = 1$ and the entropy of the weights to be greater than 0.75 at all times, and frequently higher than 0.95, particularly in the higher layers.

This correlation is sufficient for us to use the entropy of the weights as a proxy for the RMS value in determining the desired quantization. Thus we arrive at the following estimate for the number of bits required to quantize the weights in the k^{th} level of the network in the e^{th} training epoch: $N_{k,e} = H_{k,e} + c$, where $H_{k,e}$ is the entropy of the weights in the k^{th} layer, in epoch e, and c is a floor on the number of bits to be used.

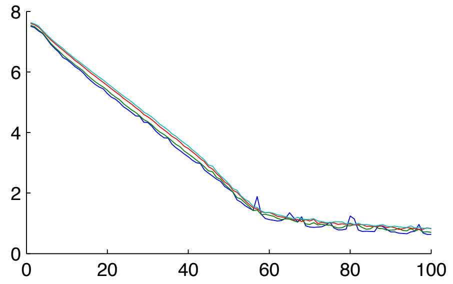
The above formula appears self-referential at first glance: the entropy estimate $H_{k,e}$ itself depends on the number of bits used to quantize the weights as noted earlier. To resolve this, we compute $H_{k,e}$ from a preliminary quantization of the weights to M bits. Typically, it is sufficient



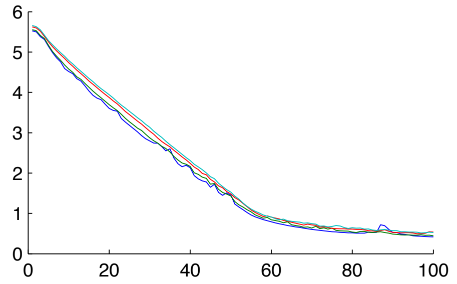
(a) 65,536 bin / 16 bits.



(b) 4,096 bins / 12 bits.



(c) 256 bins / 8 bits.



(d) 64 bins / 6 bits.

Figure 3.3: Computing the entropy using different sample ratios and number of histogram bins (bits). The sample ratio (denoted SRate in the legend), representing the fraction of the total set of weights that was used to estimate the entropy, has very little affect. Lowering the number of bins merely results in a shift of the entire curve.

to estimate $H_{k,e}$ from only a small sample of the complete set of weights. Empirically, we have found that using as few as 3% of the total set of weights can provide us with reliable estimates of the entropy of the weights in any layer. M is a parameter we can choose. c must be empirically determined, and now also accounts for the fact that $H_{k,e}$ depends on M, since increasing M by a factor of 2^K will increase $H_{k,e}$ by approximately K bits. On the other hand, decreasing M results in smoother, and potentially more robust estimates of the entropy as seen from Figure 3.3.

3.6 An Algorithm for Dynamic Compression of Weights

Algorithm 1 describes our final algorithm for compresing the network weights. The output of the algorithm is a set of packaged weights, where the package contains all the necessary information required to decode the quantized weights to their actual values. Algorithm 2 describes the algorithm used to decode the packaged weights to real numbers that can be used by the receiving node. *HuffmanCodebook*, *Encode* and *Decode* are standard algorithms for computing the Huffman codebook, encoding a sequence of bits with a given Huffman code, and decoding a stream of Huffman codes.

The parameters F, M, and c must be empirically determined. F is the fraction of weights that are used to obtain the preliminary estimate of entropy, H. M is the size in bits of the initial quantizer used to obtain this preliminary estimate. c is the quantization floor. Empirically, we have that $F = 0.03$, $M = 4$, and $c = 6$ provide excellent results. We present experiments that support this conclusion in the next section.

3.7 Experiments

We ran experiments to investigate the proposed method and demonstrate its validity. The experimental setup was that described in Section 2. Experiments were run on the 784-392-50-10 neural network described in Section 3, on the MNIST database. In total, we ran the Backpropagation algorithm about 1,500 times, while saving all the parameters of the NN after each of the 100 epochs.

3.8 Establishing Optimal Parameter Values

The proposed algorithm has three parameters: (a) the sample ratio F, which determines the fraction of the weights we use to arrive at H, the preliminary estimate of entropy, (b) the quantization floor c, and (c) M, the bit size used for the preliminary entropy estimate. We investigate the setting of each of these.

Estimating F: Figure 3.3 shows the entropy estimates obtained with different sample rates in each panel. The entropy estimates do not vary much with sample rate. We therefore set $F = 0.03$, *i.e.* 3%. For larger networks an even lower value of F may suffice.

Estimating M and c: The optimal values of M and c are closely coupled. First we establish the *ceiling* on the number of bits needed, through fixed-size quantization. Figure 3.1 shows the generalization accuracies obtained with fixed-size quantization at every quantization level between 2 and 30 bits. We are able to recover baseline results at 8 bits, which gives us a 75% reduction in communication (in single-precision) by quantization alone. We therefore consider a ceiling of 10 bits to provide room for variation.

Ideally, we must explore the entire range of (M, c) values to establish the optimal setting. Instead, we present a summary that only considers marginal variation of each of the variables, while assuring the reader that the conclusions noted generalize to the larger search.

In general, we can expect the dynamically assigned quantization size not to exceed the bit-rate ceiling. The maximum value of the preliminary entropy estimate obtained from M-bit quantization is M. Thus, we expect $c + M$ to be no greater than the ceiling. We use this to first establish an effective value for M. Figure 3.4a shows the generalization performance obtained on the test set when $c + M = 10$. Each point on the plot represents the aggregate statistics of 50 trials run with different initializations. We have varied the floor c from 2 to 10 bits. A ceiling of 10 and a floor of 10 (represented by "10:10" in the figure) represents fixed quantization to 10 bits. We observe that we obtain the best results with $M = 4$, representing a quantization floor of $c = 6$ bits. Figure 3.4b shows the performance obtained at different values of the floor c, at $M = 4$. At $M = 4, c = 5$ we obtain performance just marginally below baseline, giving us a "sweet spot" in terms of compression. It is interesting to note that the performance with 9:5, representing $M = 4$ is actually superior to that obtained with 10:5 representing $M = 5$.

Thus, we establish $c = 5$, $M = 4$, since this results in only minor loss of performance. Figure 3.5 shows the average bit-rate per parameter as a function of epoch. This represents an average bit rate of 6.8 bits/parameter over all training epochs, and an overall compression rate of 4.70.

Finally, we apply the final stage of Huffman coding to the quantized weights. Figure 3.5 also shows the average bit-rate obtained with Huffman coding. This results in a significant reduction

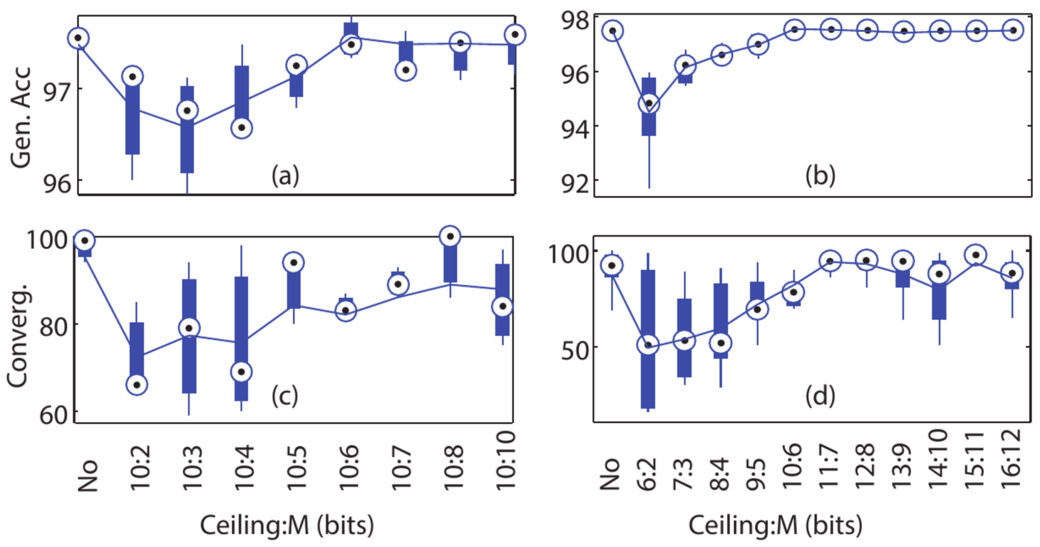


Figure 3.4: **Left:** Generalization accuracies (a), and convergence (c) for fixed quantization ceiling of 10 bits with varying floor. **Right:** Generalization accuracies (b), and convergence (d) for a fixed quantization *range* of 4 bits with increasing floor. **All:** The left-most value is the baseline, and the dots are medians and the line plots the mean.

of bit-rate at every stage, and an overall average bit-rate of 3.55 bits/parameter, representing an overall compression rate of 9.01 with respect to the baseline with no compression.

3.9 Going beyond the Baseline

Figure 3.4 reveals an interesting fact. The generalization error obtained with $c = 6, M = 4$ is actually *superior* to the baseline. These results are consistent over a large number of runs of the experiment. The mean bit rate at this setting is 3.78 bits/parameter, representing an overall compression of 8.47.

Figures 3.4c and 3.4d show the number of iterations required for the training to achieve convergence, where we define converegence as the peaking of generalization accuracy. In all cases compression of the weights results in faster convergence. At $c = 6, M = 4$, convergence is achieved in 20% fewer iterations than the baseline. Considering both the gains from parameter compression and the faster convergence, we obtain an effective overall reduction of communication of a factor of 10.32, while also improving generalization error.

3.10 Conclusion & Discussion

As the size of neural networks and the data used to train them increase, not only will more parameters need to be communicated, they will be done so over increasingly greater numbers of machines. Gains such as those we report will become very important. The computational

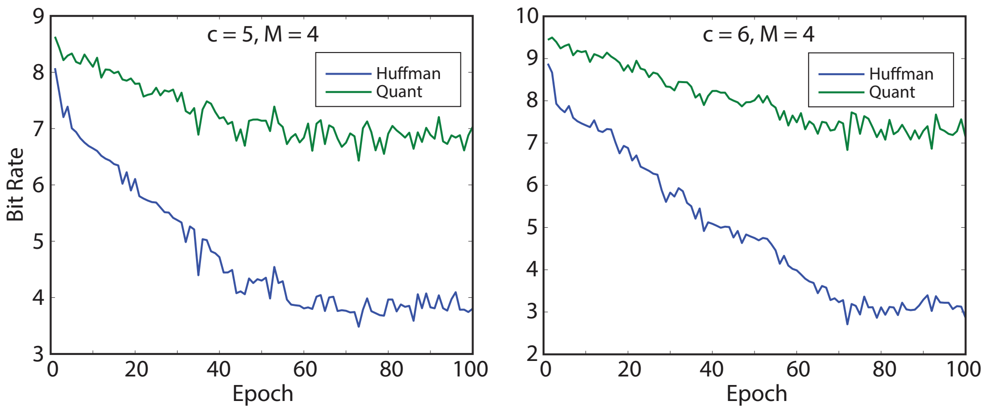


Figure 3.5: Bit-rate as a function of epoch with raw and Huffman coded quantized values. **Left:** at $c = 5, M = 4$. **Right:** at $c = 6, M = 4$.

overhead of compression is miniscule compared to the actual transmission in these scenarios. Although we have presented the compression scheme in the context of neural networks, it will apply to gradient-descent based distributed optimimzation in general. Moreover, it can also be combined with approaches that only transmit a subset of parameters, for additive gains in compression.

While our results are very promising, the actual numbers reported must still be considered with caution. A part of our current work is validating these results on much larger corpora. We are also investingating extensions to *model-parallel* formalisms [24].

The improved generalization results we observe are, perhaps, not so surprising, since the effect of quantization is to introduce quantization noise. It is well-known that noise may help a learning algorithm escape from local minima [51], and has been used in de-noising Auto-Encoders (DAEs) [147], and other variants such as dropout [55] and DropConnect [148].

Algorithm 1 Weight Compression Algorithm

```
1: Input: Weights $\mathcal{W}_l$, $l = 1..L$ for each of the $L$ layers in the net, $F$, $M$, $c$.
2: for layer $l = 1..L$ do
3:     Find $w^l_{max} = \max_{w \in \mathcal{W}_l} w$ and $w^l_{min} = \min_{w \in \mathcal{W}_l} w$.
4:     Entropy $H$ = Estimate Entropy($\mathcal{W}, F, w_{min}, w_{max}, M$)
5:     Compute the number of bits: $N = H + c$.
6:     Quantize $\mathcal{W}_l$: $\mathcal{Q}_l$ = Quantize($\mathcal{W}_l, w^l_{max}, w^l_{min}, N$).
7:     Estimate Probability $\mathbf{P}_l$ = Estimate Distribution($\mathcal{Q}_l, N$)
8:     Huffman Codebook $\mathcal{C}_l$ = HuffmanCodebook($\mathbf{P}_l$).
9:     Encode $\mathcal{Q}_l$: $\mathcal{E}_l = Encode(\mathcal{Q}_l, \mathcal{C}_l)$.
10:    Package $\mathcal{P}_l = [\mathcal{E}_l, w^l_{min}, w^l_{max}, \mathbf{P}_l, N]$.
11: end for
12: Output: $\mathcal{P}_l$, $l = 1..L$.
13:
1: function QUANTIZE($\mathcal{W}, w_{min}, w_{max}, K$)
2:     Return $\mathcal{Q} = \{ \left\lfloor \frac{2^N (w_{max} - w)}{w_{max} - w_{min}} \right\rfloor \forall w \in \mathcal{W} \}$
3: end function
4:
1: function ESTIMATE DISTRIBUTION($\mathcal{Q}, K$)
2:     Compute a $2^K$-bin histogram $h(i), i = 1..2^K$ from $\mathcal{Q}$.
3:     Normalize the histogram : $p(i) = \frac{h(i)}{\sum_i h(i)}$.
4:     Return $\mathbf{P} = \{p(i),\ i = 1..2^K\}$
5: end function
6:
1: function ESTIMATE ENTROPY($\mathcal{W}, F, w_{min}, w_{max}, M$)
2:     Select a random subset $\mathcal{W}_{sel}$ of $\mathcal{W}$ of size $F|\mathcal{W}|$.
3:     Quantize $\mathcal{W}_{sel}$: $\mathcal{Q}_{sel}$ = Quantize($\mathcal{W}_{sel}, w_{min}, w_{max}, M$))
4:     Estimate P = Estimate Distribution($\mathcal{Q}_{sel}, M$)
5:     Return $H = -\sum_{i=1}^{2^M} p(i) \log p(i)$.
6: end function
```

Algorithm 2 Weight De-Compression Algorithm

```
1: Input: $\mathcal{E}_l, w^l_{min}, w^l_{max}, \mathbf{P}_l, N$
2: Huffman Codebook $\mathcal{C}_l$ = HuffmanCodebook($\mathbf{P}_l$).
3: Decode $\mathcal{Q}_l = Decode(\mathcal{E}_l, \mathcal{C}_l)$.
4: $\mathcal{W}_l = \{w_{min} + \frac{(w_{max} - w_{min})(i + 0.5)}{2^N} \forall I \in \mathcal{Q}_l\}$.
5: Output: $\mathcal{W}_l$
```

Chapter 4

Gradient Amplification

It is well-known that saturating activation functions can impede learning in neural nets. When they saturate they produce very small gradients (e.g. in the flat sections of the sigmoid), that can cause the optimizer to get stuck (see e.g. [91]). Rectified linear units do not have this problem, and their success can in part be considered evidence of the severity of the problem of saturation. However, it is still commonplace to use a saturating activation function at the output layer, namely the softmax – even though applying the the softmax does not expand the space of functions that the NN can represent. This is done, in part, so that we may perceive the outputs as probabilities, although strictly speaking they are not. In this work, we show that using non-saturating output activation functions can improve learning on a number of standard computer vision tasks. Moreover, we present results showing that the utility of the softmax does not stem from the normalization, as some have speculated [39, 79]. Rather, the advantage is in the exponentiation of error gradients. This exponential gradient amplification is shown to speed up convergence and improve generalization.

4.1 Squashers & Saturation

Historically, output squashers, such as the logistic sigmoid and tanh functions, have been used as a simple way of reducing the impact of outliers on the learned model. For example, if you fit a model to a small dataset with a good amount of outliers, those outliers can produce very large error gradients that will push the model towards a hypothesis that favors said outliers, leading to poor generalization. Squashing the output will reduce those large error gradients, and thus reduce the negative influence of the outliers on the learned model. However, if you have a small dataset, you should not use a neural network in the first place—other methods are likely to work better. And if you have a large dataset, the impact of any outliers will be minuscule. Therefore, the outlier argument is not very relevant in the context of deep learning. What is relevant, however, is that squashing functions saturate, resulting in very small gradients, appearing in the error surface as infinite flat plateaus, that slow down learning, and even cause the optimizer to get stuck [38, 91]. This observation was part of the motivation behind applying the now popular ReLU activation (rectified linear units) to convolutional neural nets [76, 83, 112]. Surely, the massive success of ReLUs (and other related variants) speaks to the importance of avoiding saturating activations. Thus, it is interesting to investigate if there is an advantage to using non-saturating activations at the output-layer as well.

4.2 The Softmax Function

The softmax function [16], $y_j = \frac{\exp(x_j)}{\sum_i \exp(x_i)}$, is the de facto standard activation used for multi-class classification with one-hot target vectors. When the normalization term (the denominator) grows large, the output goes towards zero, and thus the function saturates. The original motivation behind the softmax function was not dealing with outliers, but rather to treat the outputs of a NN as probabilities conditioned on the inputs. As useful as this is, we must remember that in most cases the outputs of the softmax would actually *not* be true probabilities. To claim that outputs are probabilities, we must assume a within-class Gaussian distribution of the data, made in the derivation of the function [16]. In practice, we say that the outputs may be interpreted as probabilities, as they lie in the range $[0; 1]$ and sum to unity [14, 15]. However, if these are sufficient criteria for calling outputs probabilities, then the normalization might just as well be applied after training, which would not make the probabilistic interpretation any less correct. This way, we can avoid the problem of saturation during training, while still treating the outputs are probabilities (in case that is relevant to the given application). Another potential drawback of the normalization is that it bounds the function at both ends s.t. $f : \mathcal{R} \rightarrow [0, 1]$. Consequently, when we apply it at the output layer, s.t. $y = f(x)$, where the error gradient (or "error delta") $\frac{\nabla \mathcal{L}(t,y)}{\nabla x} = y - t$, and $t \in \{0, 1\}$, we effectively bound the gradients too, which affects all the previous layers during back-propagation.

4.3 The Main Idea

The first gradient that we compute for backpropagation is the derivative of the loss with respect to the input end of the output-layer activation function. As pointed out by Bishop [14, 15], for all the canonical combinations of activation + loss (i.e. linear + mean squared error, sigmoid + cross-entropy, and softmax + multi-class cross-entropy), that gradient is computed in exactly the same way. Namely, $\frac{\nabla \mathcal{L}(t,y)}{\nabla x} = y - t$, where y is the NN output and t is the target (x is the input to out the activation function of the output layer). Thus, as mentioned above, this gradient is bounded when $y = \mathrm{softmax}(x)$ or $y = \mathrm{sigmoid}(x)$. The main motivation for this work is to investigate whether the observed benefits of using non-saturating hidden activations, such as the ReLU, will also apply at the output layer. We take the view, that we can choose to *amplify* the error signal being backpropagated from the output layer—by our choice of activation and loss function. Or rather, by our *choice of gradient!* For example, if we use linear outputs with the mean squared error (MSE) instead of the softmax + cross-entropy (CE) for classification, then we can view it as a kind of *gradient amplification*. In both cases, the gradient is $y - t$, but this value can become much larger when $y = x$ than when $y = \mathrm{softmax}(x)$ —because the softmax is bounded and saturates.

Table 4.1 shows what happened when we first applied this view on real data; the MNIST

Output Activation	Error	Convergence
Sigmoid	1.8	98.5
Tanh	1.7	95.0
Linear	1.7	**73.5**

Table 4.1: Median results (20 trials) on MNIST for a 392-50-10 NN with ReLUs in the hidden layers; final classification error & no. of epochs needed to converge.

dataset [88]. Training a simple three-layer NN (fully connected) with ReLUs in the hidden layers, we compared the median results obtained over twenty trials with sigmoid, tanh, and linear output activations. The learning rate was fixed, and carefully tuned for each setting, and neither dropout [55], batch normalization [72], nor weight decay was used. The NN trained for 100 epochs, and the point of convergence is set to be the epoch where the minimum classification error was observed. This experiment was repeated multiple times with other hidden activations, and weight initialization schemes, and they all gave the same result: with linear output activations, the rate of convergence is reduced by approximately 25 percent (and some moderate improvements in generalization was observed as well). Note, that the softmax is not included in the table as it performed poorly on this specific problem.

4.4 Gradient Amplification

The softmax does have one possible advantage over linear output with respect to amplifying error gradients: namely, the exponentiation of the outputs. The exponential function is monotonic, so it does not change anything with respect to the one-hot classification, but large errors will be amplified. As the magnitude of the gradient term, $|\exp(x) - t|$, grows faster than $|x - t|$, this allows the optimizer to take bigger steps towards a minimum. An intuitive interpretation of this would be that when we are confident about an error, we can take an exponentially larger step towards minimizing that error. The idea bears some resemblance to momentum, where we gradually speed things up when the directions of the error gradients are consistent.

4.5 Exponential Amplification

If exponentiation of error gradients is good, and saturation is bad, it follows that using an "unnormalized" softmax, so to speak, should yield an improvement. That is, simply use exponential outputs, $y = \alpha \exp(x)$, but keep computing the error gradients as $\frac{\nabla \mathcal{L}(y,t)}{\nabla x} = \alpha \exp(x) - t = y - t$. For now, we can think of it as an exponential output activation with an incorrect gradient formulation imposed on it, i.e. the canonical $y - t$ (we will derive the loss that it minimizes below). As seen in Figure 4.1a, this simple change does in fact lead to a consistent boost in performance. The result was obtained on the CIFAR-10 data [82], with a 5-layer CNN; four convolutional layers followed by an affine output layer with linear outputs and exponential gradient amplification (exp-GA), and batch normalization in all layers. We set $\alpha = 0.1$, which has worked well in all our experiments. To further boost the non-linear interaction between the outputs and the targets, we used larger target values, $t \in \{0, 16\}$ instead of $t \in \{0, 1\}$. As can be seen in the histograms of Figure 4.2 (from a different experiment), this produces much larger gradients. The deltas are roughly in $[-6, 10]$, as opposed to the bounded errors of the softmax, that are in $[-1, 1]$. The idea of picking better target values is not new. To reduce the risk of saturating with logistic units LeCun et al. [91] recommend choosing targets at the point of the maximum of the second derivative.

Another potential advantage of the exponentiation is that $\exp(x)$ asymptotically approaches zero towards negative infinity. This is especially advantageous with one-hot target vectors, because we do not care about exact output values as long as the correct class has the largest value. Hence, we can mostly ignore any negative errors in outputs for the negative classes. This can be seen as a relaxation of the optimization problem, where we are essentially trying to solve an inequality for the negative classes instead of an exact equality. With linear activations and the MSE loss, the optimizer would always try to push the outputs for the negative classes towards zero. This can lead to situations where an otherwise correct output (i.e. the maximum value belongs to the node representing the target class) for a given input, x_i, leads to a weight update that renders the output incorrect the next time that x_i is seen; this is in exchange for the mean output for the negative classes being slightly closer to zero than on the previous iteration. This is undesirable, but we can avoid the problem by using exp-GA.

4.6 Sparse Cross-Entropy Loss

When we change the activation function, but *not* the way we compute the gradient, we implicitly change the loss function. The loss that is minimized by exp-GA turns out to be quite interesting: it is exactly the multi-class cross-entropy with an $L1$ penalty on the activations. As the $L1$-norm induces sparsity, we will call it the *sparse cross-entropy loss*: $\mathcal{L}(t, y) = \sum_{m=1}^{M} \sum_{k=1}^{K} (|y_{mk}| - t_{mk} \ln y_{mk}) = \sum_{m=1}^{M} (\|y_m\|_1 - \sum_{k=1}^{K} t_{mk} \ln y_{mk})$, where t_{mk} and y_{mk} denote the k'th component in the m'th example of the target and output vectors, respectively. Imposing sparsity on the output makes a lot of sense, because our one-hot targets are in fact sparse. To verify the correctness of this loss, we can take the partial derivative with respect to the input to the activation; it should be $y - t$. Assuming that $\alpha > 0$, then the activation $y = \alpha e^x$ is non-negative, and we can ignore the absolute value in the $L1$-term. The loss for a single output node does not depend on the other output nodes, so we can ignore the summations and simply consider the loss for a single node:

$$\begin{aligned}
\frac{\partial}{\partial x}(|y| - t \ln y) &= \frac{\partial}{\partial x}(\alpha e^x - t \ln(\alpha e^x)) && \text{(assuming } \alpha > 0) \\
&= \alpha e^x - t \frac{\partial}{\partial x} \ln(\alpha e^x) \\
&= \alpha e^x - t && \text{(because } \frac{\partial}{\partial x} \ln(\alpha e^x) = 1) \\
&= y - t
\end{aligned}$$

To be clear, implementing exp-GA thus requires that we use exponential output units. If we use a software package that applies automatic differentiation, then we must minimize the sparse cross-entropy loss. If not, we need to make sure that the output layer gradient is computed correctly, as $y - t$.

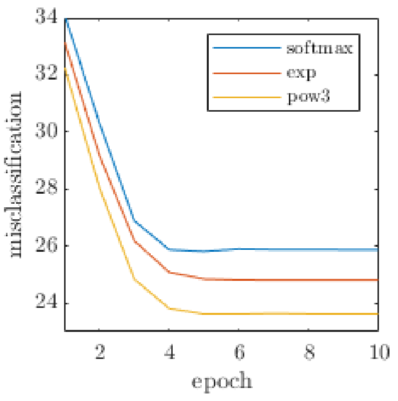
4.7 Cubic Amplification

Although we can often ignore large negative outputs that yield large negative error deltas, it is not clear that we can safely ignore all of them. This raises the question whether we may further boost performance by also allowing for the amplification of large *negative errors*. For this, we use a simple polynomial, $y = \alpha x^3 + \beta$, and again we want the canonical gradient expression, $y - t$; let's call this pow3-GA. This polynomial has a flat section centered around zero where the gradients will be relatively small. Similar to the left tail of the exponential function, for outputs that land in that section, the optimizer will thus not push that hard for strict equality. Taking another look at Figure 4.1a, we see that this does indeed yield a better result; following exactly the same trend as observed with exp-GA, that the error drops significantly faster than with the softmax. In this first experiment, we set $\alpha = 0.001$, $\beta = 0.4$, and use target values $t \in \{0, 10\}$.

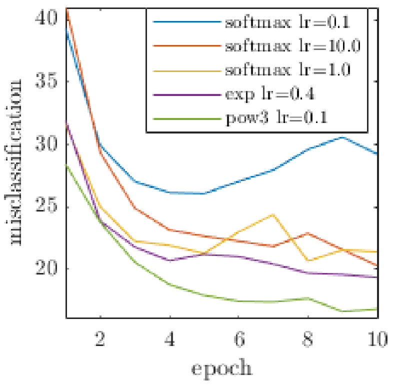
4.8 Negative Correlation Loss

The loss that we are minimizing with pow3-GA is not quite as clean and intriguing as what we saw for exp-GA. We can view it as a *negative correlation loss* with some penalty terms: $\mathcal{L}(t, y) = \sum_{m=1}^{M} \sum_{k=1}^{K} (\frac{1}{4}\alpha y_{mk}^4 + \beta y - t_{mk} y_{mk}), \alpha > 0, \beta \geq 0$, where $y = x$, and t_{mk} and y_{mk} denote the k'th component in the m'th example of the target and output vectors, respectively. The last term in the summation, $-t_{mk} y_{mk}$, measures the negative correlation between the target, t, and the *linear* output activation, $y = x$. This is meaningful, and it is essentially the cross-entropy without the logarithm. The term, $+\beta y$, is reminiscent of the $L1$-norm, with the big difference that it promotes larger negative y's. This too is meaningful, especially for the negative classes, insofar that we again view it as solving an inequality instead of a strict equality. The first term in the summation, $\frac{1}{4}\alpha y_{mk}^4$, acts much like the $L2$-norm but with a stronger and faster growing penalty on the magnitude of y (depending on α of course). Again, we verify the correctness of the loss by taking the partial derivative with respect to the input to the activation, and again we can ignore the summations and simply consider the loss for a single node:

$$
\begin{aligned}
\frac{\partial}{\partial x}(\frac{1}{4}\alpha x^4 + \beta x - tx) &= \frac{1}{4}\alpha \frac{\mathrm{d}}{\mathrm{d}x} x^4 + \frac{\partial}{\partial x}(\beta x - tx) \\
&= \alpha x^3 + \frac{\partial}{\partial x}(\beta x - tx) \\
&= \alpha x^3 + \beta \frac{\mathrm{d}}{\mathrm{d}x} x - t \frac{\mathrm{d}}{\mathrm{d}x} x \\
&= \alpha x^3 + \beta - t
\end{aligned}
$$



(a) Median misclassification rates (20 trials) for a 5-layer CNN with softmax, exponential, and cubic outputs.



(b) Misclassification rates for a 10-layer CNN with softmax (for three different learning rates), exponential, and cubic outputs.

Figure 4.1: Top-1 misclassification rates on CIFAR-10.

Which is the gradient expression that what we wanted. To clarify, implementing pow3-GA with automatic differentiation, we must use *linear* output activations, $y = x$, and optimize for the negative correlation loss. With manual differentiation, the simplest approach is to use cubic activations, $y = \alpha x^3 + \beta$, and compute the output layer gradient as $y - t$.

4.9 Experiments

We now study the performance and behavior of gradient amplification on the task of image-classification on CIFAR-10/100 [82], and the pixel-level task of semantic segmentation on the PASCAL VOC 2012 dataset [31].

4.10 All Convolutional Net

In this experiment, our purpose is not to get state-of-the-art results but to further study the behaviour of our method. We look at the first ten epochs of training with an (almost) all convolutional network with ten layers; following the principle presented in [137], but with batch normalization, and the average pooling layer replaced by a fully-connected one. The latter was done to make computation more deterministic, so as to allow for better evaluation of the effects of changing various parameters. Note that pooling involves atomic operations on the GPU, which can result in relatively large variance in output. For this experiment, we used a fixed learning rate and carefully tuned it with the purpose of getting the best result within ten epochs. We use the same α and β values as in our previous experiment, but this time we use different target values. $t \in \{0, 6\}$ produced better results for exp-GA. With pow3-GA it seemed a good idea to try negative target values for the negative classes since the function is not bounded at the lower end; we saw a significant improvement when using $t \in \{-2, 10\}$.

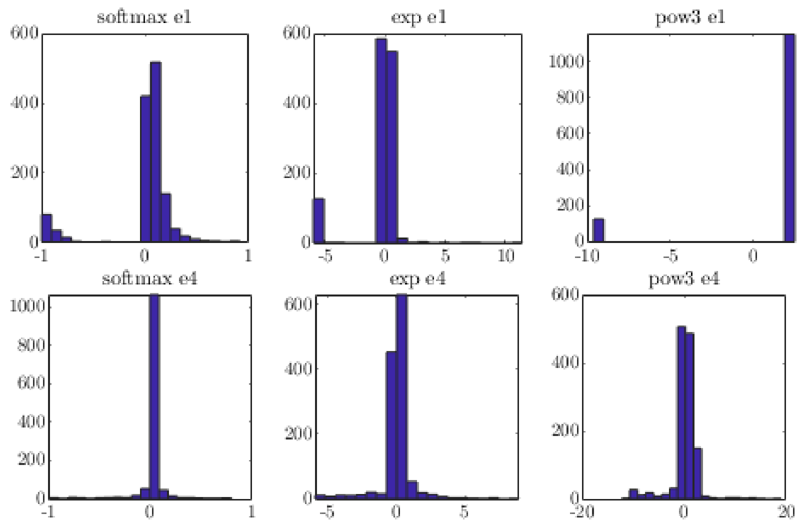


Figure 4.2: **Top:** The distribution of output layer error gradients for softmax, linear with exponential amplification, and linear with cubic amplification at the start of epoch 1; training a 10-layer CNN on CIFAR-10. **Bottom**: same, but for epoch 4.

Figure 4.1b shows how the classification error evolved during training. For softmax, we show results from trying three different learning rates to ensure that our choice of 1.0 really is a good one. We note that the overall trend is the same as for the 5-layer CNN; for the first 2-5 epochs, the error rates drop significantly faster with GA than with the softmax. The histograms in Figure 4.2 show the distribution of the output error deltas for the first batch of epoch 1 and epoch 4. The larger target values used for GA are clearly reflected; resulting in sharper distributions clustered around the negated target values. This is of course most significant on the first iteration, but the trend is still very clear in the fourth (and tenth) epoch. This amplification of the output errors has a significant effect on the gradient signals received in the hidden layers during backpropagation. Figure 4.3 shows this effect very nicely via the root mean square (RMS) of the gradients. With exp-GA, the RMS of the hidden layer gradients is an order of magnitude higher than with the softmax; for pow3-GA it is more than two orders of magnitude. Interestingly, the hidden-layer RMS-gradients recorded for pow3-GA grow rapidly from the second epoch and onwards. A similar trend is seen for exp-GA, albeit less dramatically, and for the softmax there is only a slight upwards trend, and only in the top layers. This amplification correlates well with the error rates (see Figure 4.1b); the softmax gets stuck early on, and the linear activations with gradient amplification continue to learn through all ten epochs. All in all, this seems to indicate that gradient amplification may help alleviate the infamous problem of vanishing gradients [39, 57, 58] in deep neural networks.

4.11 The VGG Architecture

GA was tested on CIFAR-10 with the well-known VGG architecture [134] with 13 and 19 layers; VGG-13 and VGG-19, respectively. The learning rate and weight-decay were tuned very carefully. We tried hundreds of different settings and picked the ones that performed best over

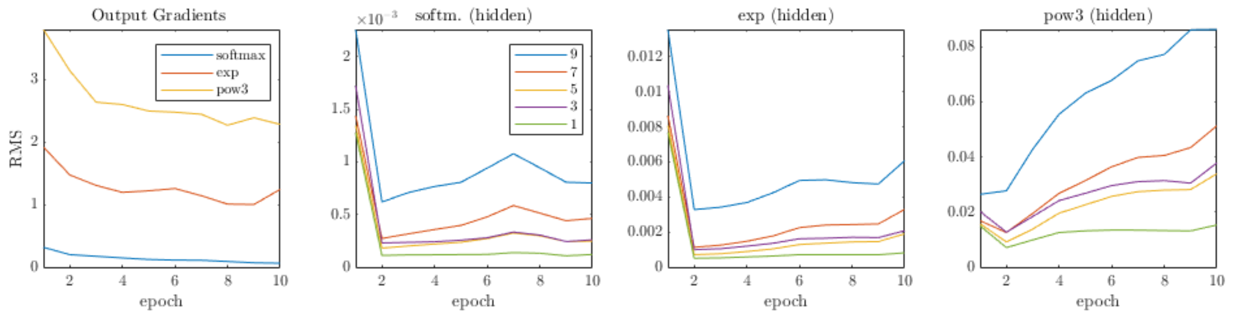
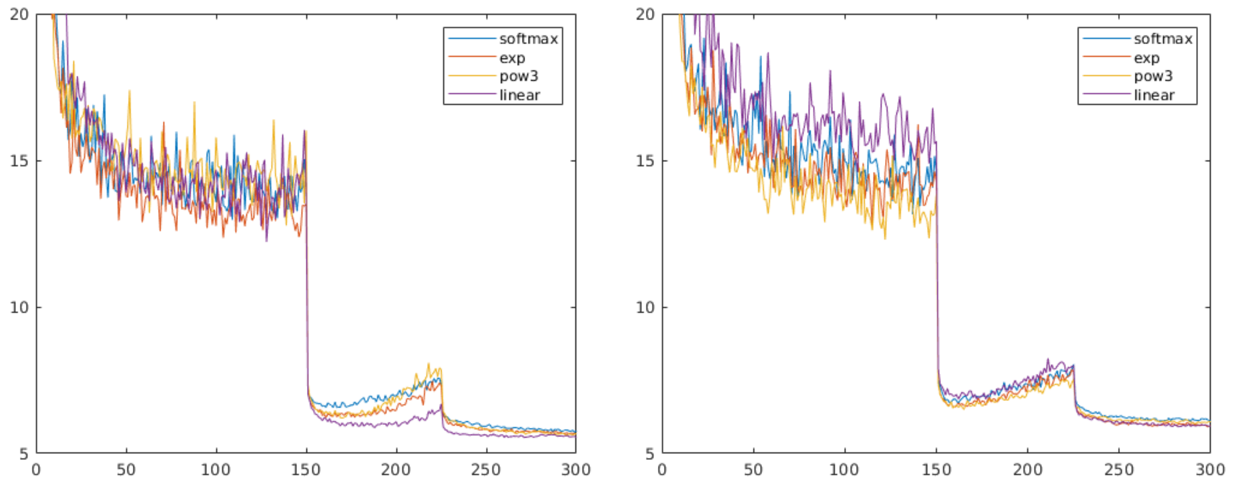


Figure 4.3: RMS of error gradients over ten epochs of training a 10-layer CNN on CIFAR-10. **Left:** output layer. **Rest:** every second hidden layer. Note the upwards trend from epoch 2 onwards in the hidden layers for GA.



(a) Median misclassification rates (8 trials) for VGG-13 with softmax, exponential, cubic, and linear outputs.

(b) Median misclassification rates (8 trials) for VGG-19 with softmax, exponential, cubic, and linear outputs.

Figure 4.4: Classification errors on CIFAR-10 with the VGG architecture.

multiple different random intializations. The learning rate was divided by 10 after 150 and 225 epochs. This learning rate schedule is common in the literature, and using other schedules did not significantly affect the results.

As shown in Figure 4.4, the benefit of GA is no longer clearly seen from the very beginning of training, but rather halfway through, or near the end. For VGG-13 both exp-GA and linear outputs overtake the softmax right after the first reduction of the learning rate (epoch 150), and for VGG-19 it is after the second reduction (epoch 225). This is because GA can cause overfitting, if applied too aggressively. So, we reduced the learning rate and used smaller target values. It actually does makes some sense for the benefit of GA to kick in later, as this is also when you would expect the softmax to saturate more.

Table 4.2 summarizes our results and hyper-parameter settings for the VGG architecture with batch normalization. On both the 13- and the 19-layer model, the softmax with the cross-entropy loss gets the worst median top-1 misclassification rate (over 8 trials) of all. Interestingly, for this architecture linear activations with the mean squared error yield the best results.

Architecture	**Activation + Loss**	**Top-1 (Median)**	**LR**	**WD**	**Tval**	α	β
VGG-13	Softmax+CE	5.68	0.05	0.0005	$\{0, 1\}$	—	—
	exp-GA	5.61	0.01	0.03	$\{0, 2\}$	0.5	—
	pow3-GA	5.55	0.1	0.003	$\{0, 2\}$	0.001	0.0
	Linear+MSE	**5.47**	0.01	0.03	$\{-1, 1\}$	—	—
VGG-19	Softmax+CE	6.05	0.05	0.0005	$\{0, 1\}$	—	—
	exp-GA	5.82	0.01	0.03	$\{0, 2\}$	0.5	—
	pow3-GA	5.93	0.07	0.0003	$\{0, 2\}$	0.001	0.0
	Linear+MSE	**5.79**	0.01	0.03	$\{-1, 1\}$	—	—

Table 4.2: VGG-19 architecture. CIFAR-10 median top-1 misclassification rate (8 trials) & hyper-parameter settings; learning rate (LR), weight decay (WD), target values (Tval), and GA-parameters, α & β.

4.12 PyramidNet

GA was tested on CIFAR-10 and CIFAR-100 with the PyramidNet architecture, Han et al. (2017) [43], with 55 and 110 layers. The widening factor, denoted by α in [43], was set to 48. We used the same learning rate, weight decay, momentum, and schedule as Han et al. Note, that this is the same standard schedule that we used earlier; the learning rate was divided by 10 after 150 and 225 epochs.

Table 4.3 summarizes our results and hyper-parameter settings for the PyramidNet architecture. Neither exp-GA or linear+MSE did well in this setting, but pow3-GA yielded a good improvement over the softmax+CE on both datasets. For the 55-layer net trained on CIFAR-100, pow3-GA converged more than 100 epochs sooner than softmax+CE. Somewhat surprisingly, the learning rates for pow3-GA had to be 1-2 orders of magnitude higher than what we would usually expect for DNNs.

Although more experiments should be conducted before making any final conclusions, one interesting observation comes from our experiments on the CIFAR-10/100 datasets. We hypothesize that the newer (and deeper) architectures have become significantly better at dealing with the vanishing gradient problem; in particular, deep residual networks such as PyramidNet. As a consequence, the problem of saturation at the output layer is less outspoken, and only the most extreme type of gradient amplification has a noticeable effect.

Architecture	Activation + Loss	Top-1	LR	WD	Tval	α	β
PyramidNet-55	Softmax+CE	6.02	0.1	0.0001	$\{0, 1\}$	—	—
	exp-GA	**5.51**	0.4	0.0001	$\{0, 2\}$	1.0	—
	pow3-GA	5.66	7.5	0.000003	$\{-1, 1\}$	0.0007	0.0
PyramidNet-110	Softmax+CE	4.98	0.1	0.0001	$\{0, 1\}$	—	—
	exp-GA	**4.67**	0.2	0.0001	$\{0, 3\}$	0.5	—
	pow3-GA	4.71	5.0	0.000003	$\{-1, 1\}$	0.0007	0.0

Table 4.3: PyramidNet architecture. CIFAR-10 top-1 misclassification rate & hyper-parameter settings; learning rate (LR), weight decay (WD), target values (Tval), and GA-parameters, α & β.

Architecture	Activation + Loss	Top-1	LR	WD	Tval	α	β
PyramidNet-55	Softmax+CE	25.23	0.5	0.0001	$\{0, 1\}$	—	—
	exp-GA	29.17	0.3	0.00005	$\{0, 2\}$	1.0	—
	pow3-GA	**24.21**	20.0	0.000001	$\{-1, 1\}$	0.001	0.0
PyramidNet-110	Softmax+CE	21.70	0.5	0.0001	$\{0, 1\}$	—	—
	exp-GA	25.51	0.3	0.00005	$\{0, 2\}$	0.5	—
	pow3-GA	**21.41**	16.0	0.000001	$\{-1, 1\}$	0.001	0.0

Table 4.4: PyramidNet architecture. CIFAR-100 top-1 misclassification rate & hyper-parameter settings; learning rate (LR), weight decay (WD), target values (Tval), and GA-parameters, α & β.

4.13 Semantic Segmentation

We now evaluate our method for the pixel-level classification task of semantic segmentation. The goal in semantic segmentation is to determine class labels for every single pixel in an image. Prior work [7, 45, 96] in this direction use a fully convolutional network with the standard softmax and multi-class cross-entropy loss for optmization. In this experimwent, we use the PixelNet architecture [7]. This model uses a VGG-16 [134] architecture (pre-trained on ImageNet) followed by a multi-layer perceptron that is used to do per-pixel inference over hypercolumn descriptors [45]. It has achieved state-of-the-art performance for various pixel-level tasks such as semantic segmentation, surface normal estimation, and boundary detection.

We evaluate our findings on the heavily benchmarked Pascal VOC 2012 dataset. Similar to prior work [7, 45, 96], we make use of additional labels collected on 8498 images by Hariharan et al. [44]. We keep a small set of 100 images for validation to analyze convergence, and use the same settings as used for analysis in [7]: a single scale 224×224 image is used as input to train

	bg	aero	bike	bird	boat	bottle	bus	car	cat	chair	cow	table	dog	horse	mbike	person	plant	sheep	sofa	train	tv	**mIoU**
E-40	92.3	**87.9**	**43.2**	73.6	54.6	**68.8**	83.9	**82.2**	77.1	**27.7**	62.0	50.8	**74.6**	**75.5**	**80.6**	**78.7**	47.4	**74.0**	43.2	**76.0**	**60.1**	**67.3**
P-40	92.3	86.0	39.1	**74.1**	49.4	66.3	**84.7**	79.7	77.4	26.4	63.2	51.6	71.4	74.7	79.8	76.6	45.1	70.6	47.5	71.8	59.9	66.1
S-40	91.9	84.9	38.5	66.8	54.0	63.4	79.8	72.9	72.7	25.4	63.6	55.4	68.2	72.7	75.5	76.2	46.7	71.6	42.8	71.2	58.4	64.4
S-60	**92.4**	86.7	39.8	72.4	**58.0**	65.6	82.9	78.9	**77.8**	26.6	**66.1**	**59.2**	71.6	74.2	77.5	77.1	**49.3**	73.8	**45.7**	73.9	58.4	67.1

Table 4.5: **Evaluation on Pascal VOC-2012 for Semantic Segmentation:** We found our analysis consistent for the pixel-level task of semantic segmentation. With only 40 epochs, our formulation exceeds the performance using Softmax+Cross-Entropy for 60 epochs. **E** deonotes exp-GA+mean-squared-error; **P** denotes pow3-GA+mean-squared-error; and **S** denotes softmax+cross-entropy-loss.

the model. All the hyper-parameters are kept constant except the initial learning rate[1]. We report results on the Pascal VOC-2012 test set (evaluated on the PASCAL server) using the standard metrics of region intersection over union (**IoU**) averaged over classes (higher is better).

Table 4.5 shows our results (both per-class and **mIoU**) for GA and the standard softmax. We observe that the model trained using *exp-GA* converged after 40 epochs, whereas the *softmax* model converged after 60 epochs. As seen in Table 4.5, our method provides **33%** faster convergence, while yielding a slightly better performance (**E-40** vs. **S-60**). We see a significant **3%** boost in the first 40 epochs with exp-GA (**E-40** vs. **S-40**).

Additionally, recent work [110, 145, 149] in the computer vision community have formulated regression problems such as depth and surface normals estimation, and trajectory estimation, in a classification paradigm, in hope of easier optimization and better performance. From these experiments, we however infer that it is likely not the *softmax+cross-entropy* that boosts the performance. Rather, it is the use of one-hot encoding of the target vectors.

4.14 Further Analysis

We can take a slightly more theoretical view on gradient amplification, by reasoning about second-order properties of the error surfaces induced by exp-GA and pow3-GA. This is typically done with the Hessian matrix, H, of second derivatives, which tells us something about the rate of change in the error for a single step of gradient descent. To keep things simple, we will consider only the case of a single output activation, i.e. a single dimension, so we do not need the full Hessian, $\frac{d^2}{dx^2}f$ will do. We will look at $\frac{\partial^2 E}{\partial x^2}$, where E is the sum-of-squares error, $E(y,t) = \frac{1}{2}\sum_i \|y-t\|^2$. For our purpose we can simply ignore the summation in E, and just analyze $\frac{\partial^2 E}{\partial x^2}$ for a single example, (x,t). Let us start by comparing the Hessians for linear, softmax, exponential, and cubic activations.

For a linear activation, $y = x$, the Hessian is simply

$$H_{linear} = \frac{\partial^2 \|y-t\|^2}{\partial x^2} = 1 \tag{4.1}$$

[1]The initial $lr = 1{\times}10^{-3}$ for softmax, $lr = 1{\times}10^{-4}$ for exp-GA, and $lr = 5{\times}10^{-5}$ for pow3-GA. Lowering the learning rate for softmax deteriorates the performance.

Re-writing the softmax activation as $y = \frac{e^x}{s}$, where $s = \sum_i e^{x_i}$ is a proxy for the normalization term, we get

$$H_{softmax} = \frac{\partial^2 \| \frac{e^x}{s} - t \|^2}{\partial x^2} \qquad = \frac{e^{2x}}{s^2} - \frac{e^x}{s}(t - \frac{e^x}{s}) \tag{4.2}$$

and for exponential and cubic activations we have,

$$H_{exp} = \frac{\partial^2 \| e^x - t \|^2}{\partial x^2} \qquad = e^{2x} - e^x(t - e^x) \tag{4.3}$$

$$H_{pow3} = \frac{\partial^2 \| x^3 - t \|^2}{\partial x^2} \qquad = 9x^4 - 6x(-x^3 + t) \tag{4.4}$$

If we consider the situation where x is near some local minimum, we know that the error surface will be locally convex around that point. This means that $H \approx 0$, and that the first and second term in each of the above Hessians will be approximately equal (i.e. they cancel each other out). Thus, we will ignore the second terms, and simply compare the growth of all the first terms, as we move x away from that local optimum. Now it becomes immediately evident that (locally) $H_{pow3} > H_{exp} > H_{softmax} > H_{linear}$, because as $x \to \pm\infty$ we get $9x^4 > e^{2x} > \frac{e^{2x}}{s^2} > 1$ for all $s > 1$. Unsurprisingly, it all depends on the magnitude of the normalization term of the softmax, $s = \sum_i e^{x_i}$. If s is very small $H_{softmax}$ will blow up, so we need to assert the probability of that happening. At the onset of training, it is reasonable to assume that the input to the softmax will be evenly distributed around zero. Thus, half of the x_i's are positive, guaranteeing that $s > 1$ as $\forall x > 0, e^x > 1$. To see what happens later, we can consider a numerical example for one thousand classes. Even when the model is trained well, such that the x_i's for the 999 negative classes are likely to be negative and contribute very little to s as $e^{x_i} \ll 1$—it still takes only one single $x_i \geq 0$ to make $s > 1$ (likely to be the one for the positive class). It seems reasonable to claim that this will probably be the case most of the time.

To back up this claim, we take a look at the actual x_i's recorded during training of the 10-layer CNN from our CIFAR-10 experiment in the previous section. Figure 4.5 shows how the normalization term, s, of the softmax actually behaved. It starts out with a value of 2,342 and increases monotonically from there.

However, we need to remember that for GA the Hessians are a little different, as we are just amplifying the error gradients, $y - t$. Thus, the second derivatives are just the derivatives of those deltas, with $H_{exp-GA} = e^x$, $H_{pow3-GA} = 3x^2$, and $H_{softmax} = 1$ (with the multi-class cross-entropy loss)—which only adds to our point that GA can minimize the error faster than the softmax.

4.15 Conclusion

Our results suggest fundamental changes to our default approach to deep network training, and to our perception of the omnipresent softmax function. Allowing output activations to saturate, and thus bounding their derivatives, comes at a price. However, by specifically addressing the issue of

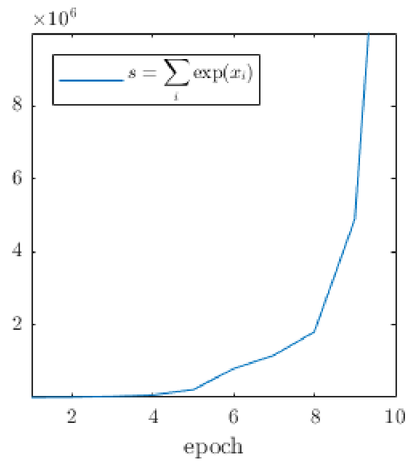


Figure 4.5: Magnitude of the softmax normalization term, $s = \sum_i e^{x_i}$, recorded for a 10-layer CNN trained on CIFAR-10. It starts out with $s = 2,342$ and blows up from there.

saturation, and choosing unbounded output error derivatives, we get a consistent improvement in DNN training. For all of the four datasets, MNIST, CIFAR-10, CIFAR-100, and Pascal VOC, and all of the architectures, we were able to outperform softmax+CE. The test errors are consistently lower, and the training can converge 25-33% faster. In the case of semantic segmentation with the PixelNet architecture, this saved a full two days of computation, and lead to an improvement of the state-of-the-art.

This all comes at no other cost than a few minutes of coding. The only drawback is the introduction of some new hyper-parameters, α, β, and the target values. However, these have been relatively easy to choose, and we do not expect that a lot of fine-tuning is required in the general case.

Chapter 5

How & What Neural Networks Learn

5.1 Introduction

Understanding how and why neural networks work so well has turned out to be quite non-trivial. One thing which has certainly contributed to the overall fascination with neural nets is that they are, however loosely, inspired by how the brain works. This fact has been an essential part of the classical approach to NN training. An approach that has certainly led to some important and impressive results. However, this brain analogy has perhaps also contributed somewhat negatively. Firstly, it has to some extent shrouded neural nets in a veil of mystery—after all, we still do not completely understand how the brain actually works (and maybe we never will?). Secondly, it has inadvertently locked the entire field into a brain/backprop paradigm; with a rather fixed set of underlying assumptions on how things should be done. Notwithstanding the overwhelming success of this paradigm, and more recent improvements such as ReLUs (first used by Fukushima in [35, 36], later popularized by [48, 112]), batch normalization [72], dropout [56], residual networks [48], and transformers [146], the core approach to NN training has not evolved much since 1986. Hence, so far, many important questions remain unanswered. Thus, the recent success of neural nets, the era of deep learning, can mostly be accredited to the availability of more data, as well as the access to more powerful computers. In our context, the biggest concern may be the absence of alternative training algorithms. In particular, algorithms that are more *efficient* than backprop + SGD.

In this chapter, we will first describe three main classical views on deep learning. Next, we will observe and discuss some properties of the hidden layers & representations of deep neural networks, and present an alternative view on DNN training. The aim is to establish useful insights that may aid in finding new ways of training them.

5.2 Three Classical Views on DNNs

Historically, the approach to training and understanding neural networks has been dominated by three main views. In this section, we will describe these three views in some detail.

5.2.1 The Brain/Connectionist View

Neural networks owe their name to their structural similarity with the biological brain. Specifically, that they consist of many simple computational units (neurons) that are interconnected. These connections (or synapses, if you will) are weighted, meaning that any connection between two neurons can be considered strong or weak. Thus, a *trained* neural net may be seen as satisfying a known property of the brain, namely that: “neurons that fire together, wire together”. This property was first described by Donald O. Hebb in 1949 [49], and lead to the so-called Hebbian learning paradigm (to which the backpropagation algorithm does *not* belong, by the way). In the 1980s, the idea of having many connected units that compute in parallel was referred to as “parallel distributed processing”. McClelland & Rumelhart presented this view in their seminal book series of the same name [103, 104, 126]. Later, the view was re-branded as *connectionism*. The idea stems back to historical work by McCulloch & Pitts [105], Rosenblatt [124, 125], and Hebb [49]. Thus, structurally and algorithmically, the main link between artificial neural networks and biological ones, is that they fall under the connectionist view. In the case of affine, or fully connected, layers this is pretty much the only link.

However, in the case convolutional neural nets, it is a different story. They were first introduced by Fukushima in 1980, under the name “Neocognitron”, and were explicitly designed to have a “structure similar to the hierarchy model of the visual nervous system proposed by Hubel and Wiesel” [36]. This model of the mamallian visual system, was described in [65, 66, 67] by Hubel and Wiesel (1959, 1962, 1968), who were later awarded with the Nobel prize. A convolutional network captures three important properties of the primary visual cortex, an area in the brain also referred to as V1:

1. Like V1, a convolutional layer is organized in a spatial map.
2. The features of a convolutional layer are spatially localized; corresponding to the *simple cells* of V1.
3. Pooling layers make CNNs invariant to small shifts in the position of detected features; roughly corresponding to the *complex cells* of V1.

For a more detailed account of the connection between CNNs and neuroscience, see Goodfellow et al. (2016) [39], p. 353-359.

5.2.2 The Feature Hierarchy View

Perhaps the most interesting aspect, in our context, of the CNN/brain link is *what* they learn. As it turns out, there is a remarkable similarity between the representations learned by a CNN, and those of the visual cortex. Figure 5.1, due to Manassi et al. (2013) [101], illustrates the hierarchical, feedforward visual processing in the four regions of the visual cortex. The authors state:

> “Stimuli are processed in a series of visual areas. V1 neurons are most sensitive to low-level features, such as edges and lines. In higher visual areas, like V4 and IT, receptive fields are larger, and neurons are sensitive to complex features, such as shapes and objects. Responses of high-level neurons are fully determined by the

neural firing of lower-level neurons. For example, the neural firing to a square is determined by the neural firing for two vertical and two horizontal lines."

This hierarchy of features is exactly what we can observe in the layers of a convolutional net. In fact, Cichy et al. (2016) [21] showed that when "tuned to the statistics of real-world visual recognition", CNNs capture "the stages of human visual processing in both time and space from early visual areas towards the dorsal and ventral streams."

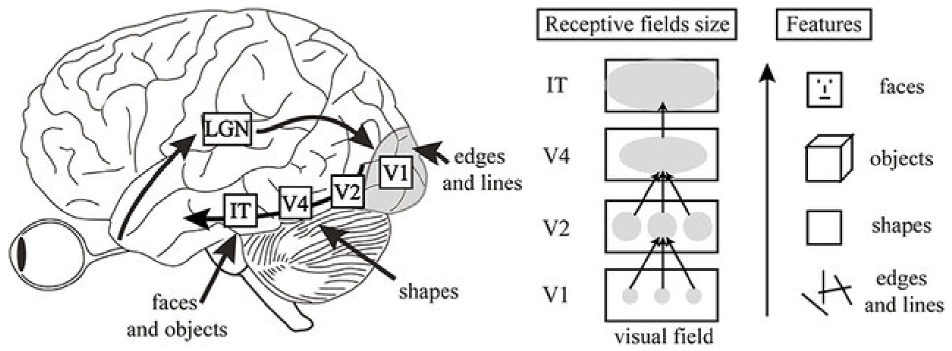


Figure 5.1: Hierarchical, feedforward visual processing in the visual cortex. This figure from Manassi et al. (2013) [101]. Visual stimuli passes from the retina through the optical nerve and, via the lateral geniculate nucleus (LGN) region, to the back of the head where the visual cortex is located. It enters at V1 and is fed forward to V2, V4, and IT (the inferotemporal cortex). As depicted, higher and higher level features are detected as the stimuli is fed from region to region.

Figure 5.2 shows a stick figure illustration of what such a hierarchy of features might look like in a convolutional neural network. The CNN, f, maps the input images to the first hidden representation which detects simple lines. The next layer detects linear combinations of lines, i.e. simple shapes, and the next one objects composed by said shapes. The last hidden layer detects combinations of the objects, i.e. scenes, and the final output is a class label. Note, that all neural nets contain layers that compute linear combinations of the features from the previous layer. Thus, the feature hierarchy view is valid for all types of NNs—only the direct link with the visual cortex is strictly limited to CNNs.

Visualizing the actual features from a CNN is not completely straightforward. Nevertheless, Zeiler et al. (2014) [155] produced some interesting ones by using the technique of deconvolution (see their paper for details). As can be seen in Figure 5.3, the features they visualized correspond very well with the conceptual ones depicted by our stick figures in Figure 5.2.

An interesting observation from Figure 5.3 is that the first layer learns Gabor-like filters. In 2D, a Gabor filter is essentially a Gaussian kernel multiplied by a sinusoidal wave. These filters are localized with respect to spatial location, frequency, and rotation (unlike e.g. a Fourier basis which is not localized in space); see Goodfellow et al. (2016), p. 357-359, for an explanation of Gabor functions and their connection to machine learning. Gabor filters are similar to the features employed by the simple cells of V1. Neural nets are not alone in learning such functions, when applied to natural images. Actually, a very wide range of machine learning algorithms are

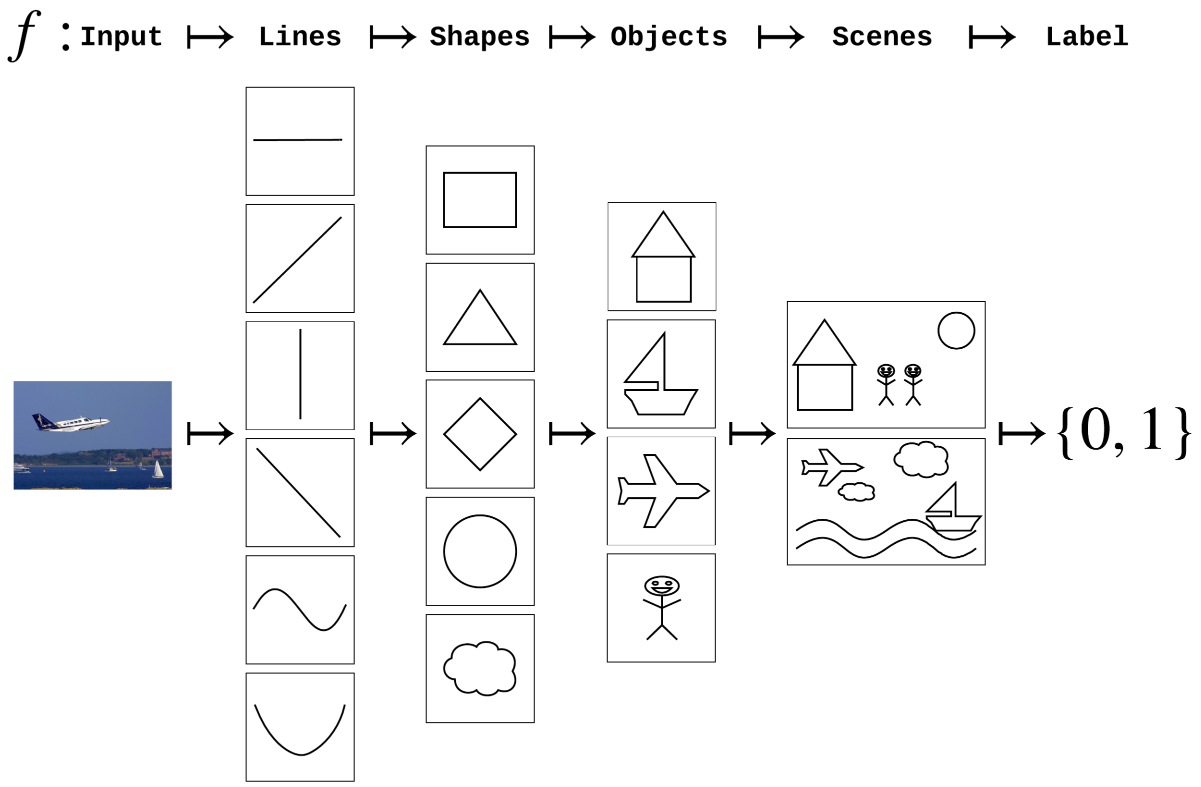


Figure 5.2: Stick figure illustration of the feature hierarchy view on DNNs.

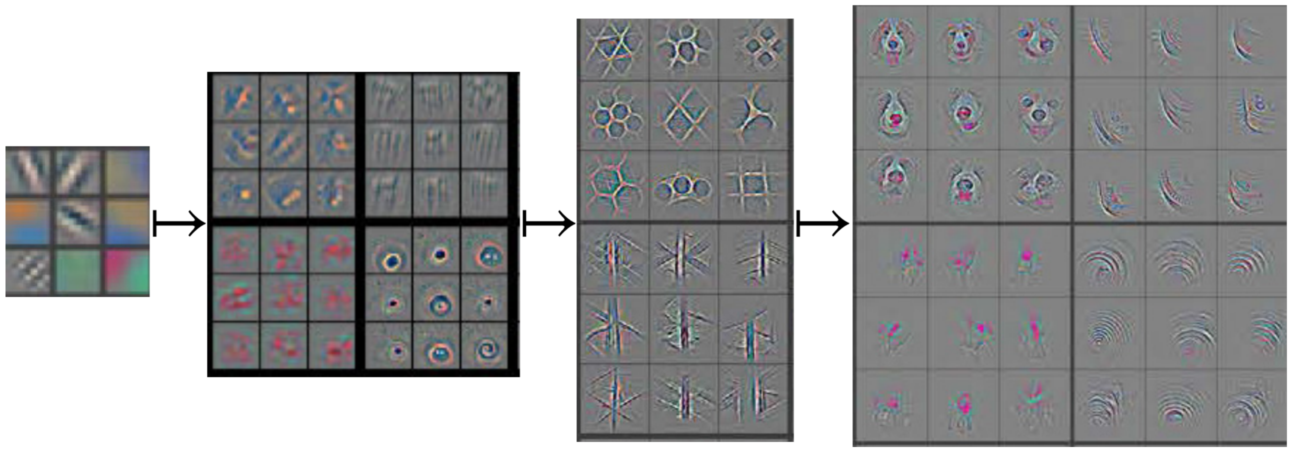


Figure 5.3: ConvNet features visualized using deconvolution. This figure adapted from Zeiler et al. (2014)[155]. Showing a subset of the features from the first four conv-layers (left to right). The first layer detects Gabor-like features, while the last layer detects much more complex shapes such a dog faces. The CNN was trained with the ImageNet 2012 dataset.

known to do that. Figure 5.4 shows an example of a Gabor filter bank: the same feature rotated 180 degrees in 64 steps.

In image analysis, Gabor filters are commonly used for edge detection and texture analysis.

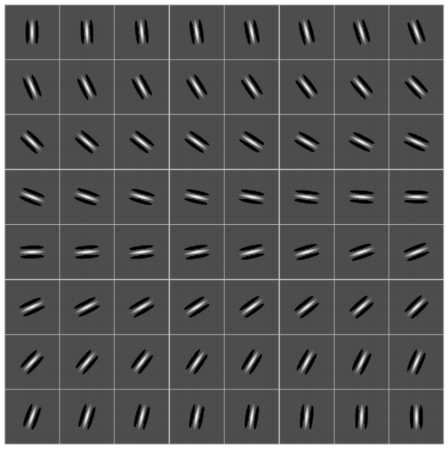


Figure 5.4: Example of a Gabor filter bank. The black and white pixels indicate large negative, and a large positive weights, respectively. The background color indicates a zero weight. Here we see the same feature rotated 180 degrees in 64 steps.

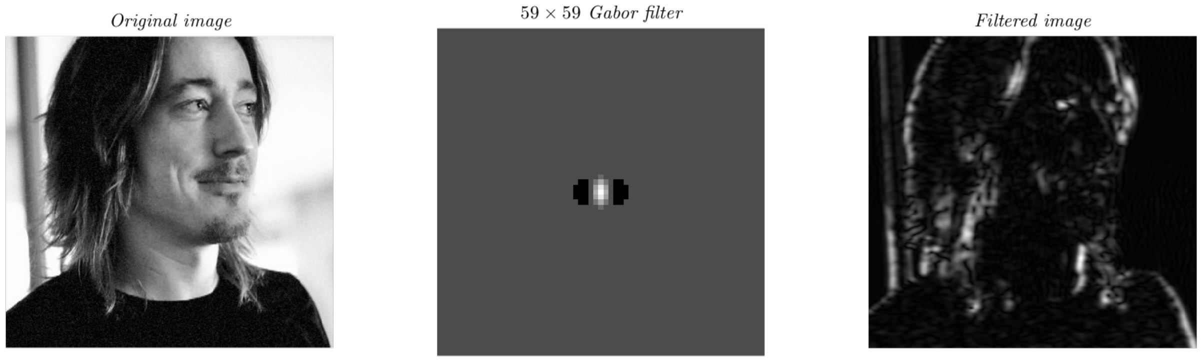


Figure 5.5: Example of edge detection using a Gabor filter. The original image (left) is convolved with the Gabor kernel (middle), which produces the image on the right: the edges are clearly enhanced.

An edge in an image is characterized by a sharp transition in color along some direction—which is exactly what the Gabor filter detects (and looks like). In Figure 5.5 we can see what this may look like. A two-dimensional convolution of the original image with the Gabor filter produces an image where edges are clearly enhanced, while everything else is mostly filtered away.

5.2.3 The Backprop View

Backpropagation [127] is of course an *algorithm*. However, implicitly, it also represents a *view* on what it means to train a neural net, and what the hidden representations should be. In practice, backprop is the only algorithm currently being used for training DNNs. After each forward pass, the algorithm computes the error gradients at the output layer. These gradients are then propagated backwards, layer by layer, using the chain rule of calculus. For each layer, the weights are updated such that they take a small step in the direction of their respective negated error gradients. This process is repeated, usually thousands of times, until the algorithm converges. Thus, the following properties can be said to characterize the underlying view of the backprop algorithm:

1. To learn the weights of the ith layer, we *must* use precise information about the representations and error gradients of all the succeeding, jth layers, $j > i$.
2. All layers must be trained in concert.
3. Training requires differentiability.
4. Training is an iterative process.

The tremendous success of backprop speaks for itself. For roughly 40 years, it has been *the* go-to algorithm for training neural networks. Over the past 10-15 years or so, in the era of deep learning, it has been the driving method behind significant breakthroughs in computer vision, speech recognition, natural language processing and many other fields. Hence, empirically, the properties above have been validated many times over. So, to say that they are wrong would not make much sense. Nonetheless, in the following, we will show that they are at least not *exclusively* correct—it all depends on how you view the problem. In fact, when we change our view on neural network learning, all of those four properties can be made redundant. We will show how in the following chapters on *deep layer-wise learning* (Chapter 6), and *Platonic Projections* (Chapter 7).

5.3 *What* Neural Networks Learn

Although the classical views on DNNs have certainly inspired and resulted in a multitude of important work in the field, there is a need for alternate ones. Mostly because the classical views do not immediately offer any useful and directly applicable insights (well, apart from that running backprop is generally a pretty good idea). In this section, we offer some of our own observations and views on the properties of hidden layers and representation—that we find are indeed *useful.*

We should note that of course multiple other theoretical contributions have been made to the field of deep learning in recent years. In our opinion, the most useful one is that of Tishby et al. [143] which frames deep learning in the context of information theory via the information bottleneck principle. Another one of note is through spline theory [6], and more recently authors have shown correspondence between neural networks and quantum field theory [34, 42, 71].

5.3.1 A Discrete Algebraic Cross-Fade From X to Y

A k-layer neural net is a sequential map between $k + 1$ representations

$$f : X \mapsto Z_1 \mapsto Z_2 \mapsto \cdots \mapsto Z_{k-1} \mapsto Z_k = \hat{Y} \tag{5.1}$$

In a manner of speaking, assuming $\hat{Y} \approx Y$, we can view this as a discrete algebraic cross-fade between the two *signals* X and Y. With every step forward in the map, less of X and more of Y

is "mixed" into Z_i. Or, to put it more mathematically, and relating back to Equation 5.1, we may say that *ideally* this will hold for sufficiently large k (and sufficient capacity in the network etc.)

$$\{\|Y - Z_i Z_i^{+} Y\|\}_{i=1}^{k} \to 0, i \to k \tag{5.2}$$

Empirically, we have shown the validity of this equation multiple times. That is, for classification problems we have shown that the *least squares distance* (LSD),

$$\frac{1}{m}\|Y - Z_i Z_i^{+} Y\|_F^2 \tag{5.3}$$

decreases monotonically in i in a trained network. For example, we recorded the LSD for each of the hidden layers, while training a four-layer MLP on the MNIST dataset (using backprop + SGD). As can be seen in Figure 5.6, all the hidden layers move towards the target (in a least squares sense), and end up satisfying Equation 5.2.

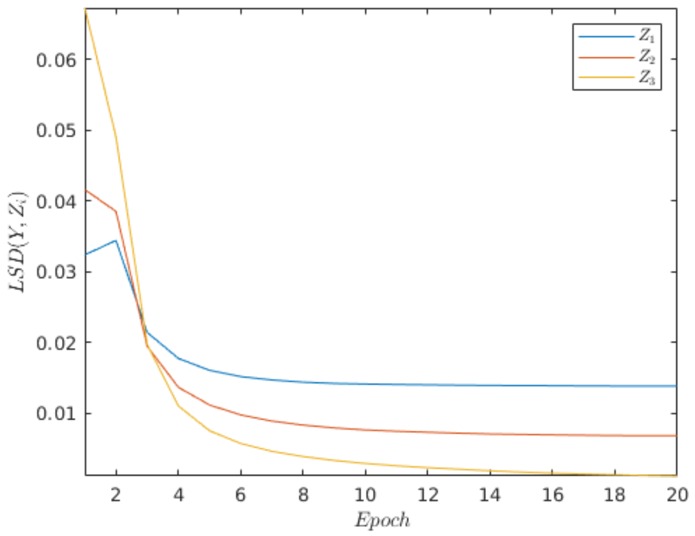


Figure 5.6: The least squares distance (LSD) from each of the hidden representations, Z_i, to the target, Y, recorded in a 784-392-196-50-10 MLP while being trained on the MNIST data.

A related observation was made in [155], only for transfer learning. That is, they showed that the features of a model pretrained on the ImageNet data become gradually more discriminative, layer by layer, for the Caltech-101 and Caltech-256 datasets.

5.3.2 Gradual Linearization

In thinking about what this all means for the hidden representations, our Z_i's, as we step *backwards* from the output towards the input layer, some useful intuitions arise. **Thus, with each step backwards in the sequential map, f:**

1. Y becomes less and less planar, and more and more bent in the Z_i's.
2. The decision boundary becomes more and more wiggly in Z_i space.

3. The Z_i's move away from Y (in target space), and towards X—in a least squares sense.

In summary, the goal of NN training is to learn a sequence of hidden representations, $Z_i, i = 1, \dots, (k-1)$, such that at the final Z_{k-1}, the learning problem, $\mathcal{D}(Z_{k-1}, Y)$, has become (sufficiently) linear. Figure 5.7 is a grossly simplified depiction of what it might look like, for a binary classification problem, if we could visualize the decision boundaries in each Z_i space of the network.

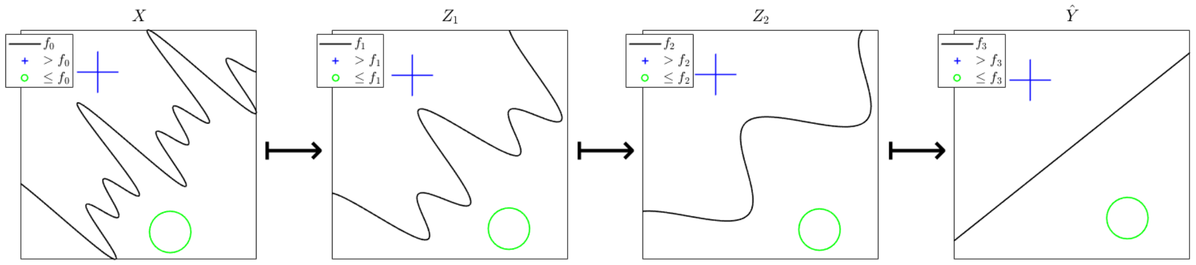


Figure 5.7: The decision boundary, depicted as f_i, becomes more and more linear (or less and less complex), as we map forward through the network.

5.3.3 Gradual Class Separability

Another way of describing what Equation 5.2 implies is in terms of class separability. Or, conversely, in terms of class overlap. The goal of classification with neural networks is to produce a representation, Z_{k-1}, where each class can be separated from all the other classes by a single hyperplane. This separation happens gradually through the layers, as depicted in Figure 5.8.

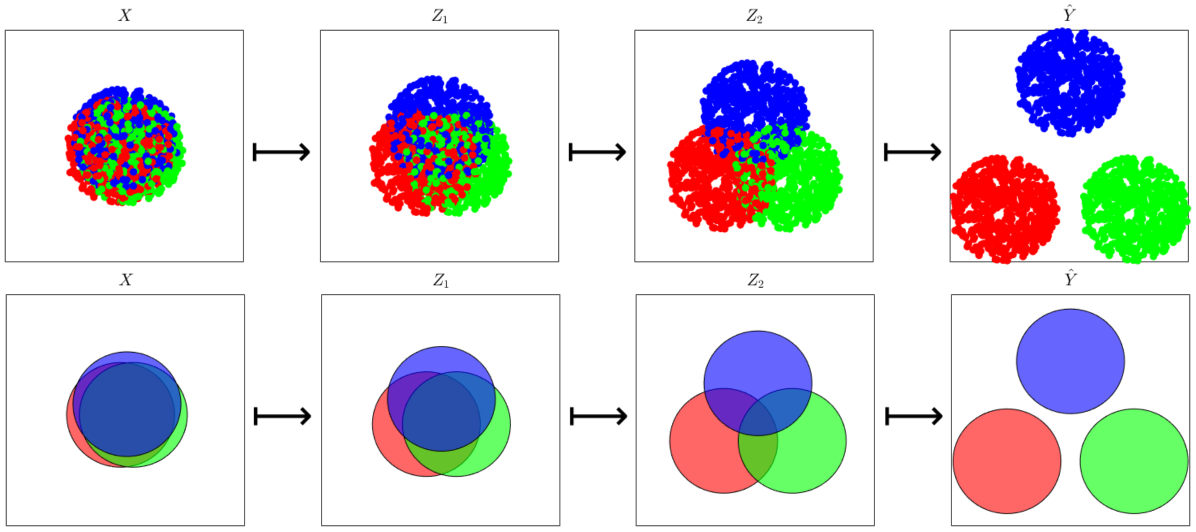


Figure 5.8: Cartoon illustration of how the classes gradually separate, layer by layer. Clearly, the critical regions are those where the classes overlap. Viewing the situation as a Venn diagram, the goal of learning must then be to minimize the summed cardinality of all set intersections.

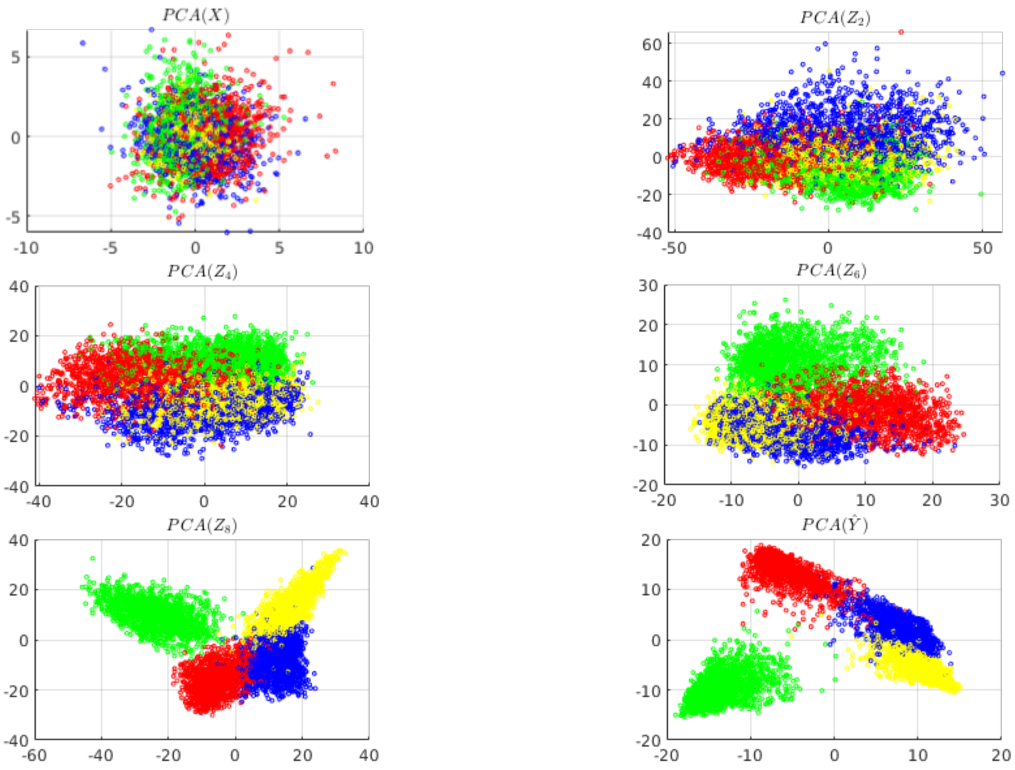


Figure 5.9: The classes gradually becoming more separated through the layers of a 10-layer CNN. Showing the two first principal components of the input, output, and hidden layers 2, 4, 6, and 8.

This process can be shown on real data too. We trained a 10-layer CNN on the CIFAR-10 dataset, and then visualized the hidden representations. To avoid clutter, we used only four of the classes, which each were plotted with individual colors. As these representations are in very high dimensions, and cannot be visualized directly in 2D, we plotted them along their first two principal components. As one can see in Figure 5.9, the result shows that the classes gradually become more and more separated (or disentangled), as we step forward from layer to layer. This literally means, that the decision boundary (or Y in Z_i space) becomes more and more planar, and consequently, that the $LSD(Y, Z_i) = \frac{1}{m}\|Y - Z_i Z_i^+ Y\|_F^2$ must decrease as i increases.

Note, that multiple other authors have also observed this gradual class separability (and linearization), such as [10, 73, 100, 111, 119, 150, 155].

5.3.4 The Rubik's Cube View

A visually pleasing way of thinking about neural nets is, what we have dubbed, The Rubik's Cube View. The key observation underlying it is that it is all about *binning* the data. **Concretely, for each layer, the goal is to bin the data into as few bins as possible—with the constraint that each bin is *pure* (or homogeneous) with respect to the class label.** This view is thus exactly in line with the view of Tishby et al. [133, 143]. Whereas they arrived at it through information theory, we arrived at the same conclusion through algebra, geometry, and empirical observations.

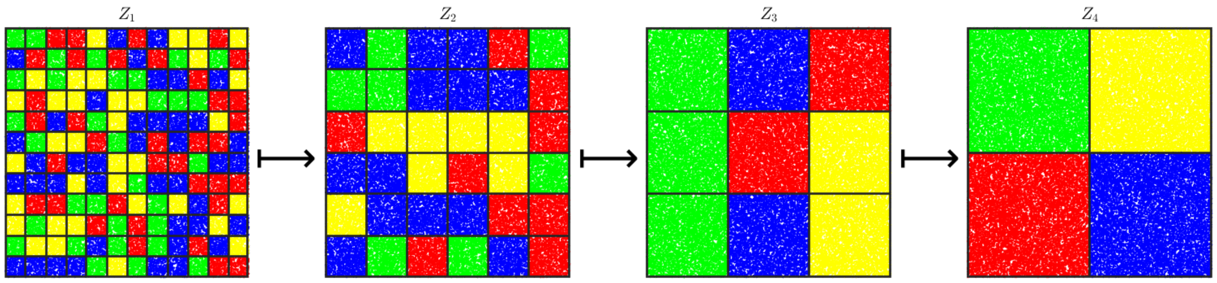


Figure 5.10: Conceptual illustration of the Rubik's cube view on deep learning. The job of any layer is to map the classes into fewer bins (linear regions) than were present in the previous layer.

Figures 5.8 and 5.9 illustrate our observation quite clearly: as the classes separate, samples tend to be surrounded mostly by samples from their own class. Visually, this is somewhat similar to solving a Rubik's cube, where the sides gradually become more and more "pure" in color, as you solve it; as depicted in Figure 5.10.

Pure Voronoi Maps

The squares in each of the Z_i depicted in Figure 5.10 (as 2D Rubik's cubes) form a Voronoi map; see Definition 5.

Definition 5 (Discrete Euclidean Voronoi Map)**.** *Let X be a finite discrete set of points in $\mathbb{R}^n$, and let $C = (c_1, c_2, \ldots, c_k), c_i \in \mathbb{R}^n$, be a tuple of centroids. The Voronoi map of C on X, denoted $V(C) \in X$, is the tuple $V^*(C) = (v_1, v_2, \ldots, v_k)$ of Voronoi cells (or regions), where each cell is given by*

$$v_i = \{x \in X | \forall j \neq i . \|c_i - x\|_2 \leq \|c_j - x\|_2\}$$

In fact, the squares form a *Pure Voronoi map*; see Definition 6. That is, a Voronoi map with the added constraint that the cells be pure (homogeneous) with respect to the class label. We could also call it a pure partitioning (or tessellation) on $\mathcal{D}(X, Y)$.

Definition 6 (Pure Voronoi Map)**.** *Let $\mathcal{D}(X, Y)$ be a dataset of pairs, $(x, y) \in \mathcal{D}$, let $C = ((c_1, y_1), (c_2, y_2), \ldots, (c_k, y_k)), c_i \in \mathbb{R}^n$, be a tuple of **labelled** centroids, and let $\ell(c_i) = y_i$ denote the label of c_i. A pure Voronoi map on $\mathcal{D}$, denoted $V(\mathcal{D}) = V(X; Y)$, is the tuple $V(C) = (v_1, v_2, \ldots, v_k)$ of pure Voronoi cells (or regions), where each cell is given by*

$$v_i = \{(x, y) \in \mathcal{D} | y = \ell(c_i) \wedge \forall j \neq i . \|c_i - x\|_2 \leq \|c_j - x\|_2\}$$

As we shall see below, pure Voronoi maps provide a useful way of reasoning about neural networks, their hidden representations, and the complexity of learning problems.

Pure Entropy

If we consider the cell-wise distribution of samples in $V(\mathcal{D})$ a probability distribution, we can compute the entropy of it. **The probability of each cell is its frequency**

$$p(v_i) = \frac{|v_i|}{|\mathcal{D}|}$$

Thus, we simply define the pure entropy as follows:

Definition 7 (Pure Entropy)**.** *Let $V(\mathcal{D})$ be a pure Voronoi map (Definition 6). The pure entropy, denoted $H(V(\mathcal{D}))$, is the Shannon entropy over the probability distribution of the cells in $V(\mathcal{D})$*

$$H(V(\mathcal{D})) = -\sum_{v \in V(\mathcal{D})} p(v) \log p(v), \text{ where } p(v) = \frac{|v|}{|\mathcal{D}|}$$

We will be using the notation, $H^*(\mathcal{D}) = H^*(X;Y)$, when making statements about a given dataset (or representation). The asterisk indicates that the pure Voronoi map used to compute the pure entropy is considered *optimal*. That is, $V^*(\mathcal{D})$, is minimal with respect to the number of Voronoi regions, and the probability distribution over those regions is as sharp as possible—such that it minimizes $H(V(\mathcal{D}))$. *This assumption of optimality is useful, in that we cannot always expect to know, or even be able to compute, the correct $V(\mathcal{D})$.*

Figures 5.11 & 5.12 show the pure Voronoi maps (and corresponding entropies) on simple binary classification problems. Please note, that the maps were computed with a naive, and very inefficient, recursive algorithm that does not scale to large data—and does not guarantee that the map is optimal. Regardless, the depicted results are still meaningful and correctly illustrate the concept.

Now, we are ready to state **an important inequality, that captures the Rubik's cube view.** For a classification problem, or any $D(X,Y)$ where $H(X) > H(Y)$, the following should hold for a well-trained neural net

$$H^*(X;Y) > H^*(Z_1;Y) > \cdots > H^*(Z_{k-1};Y) > H^*(Z_k;Y) \tag{5.4}$$

The restriction to $H(X) > H(Y)$ is probably important here, and we may think of it as representing *many-to-one* relationships; i.e. $f : X \mapsto Y$ is *injective*. In practice, we suspect that the inequality holds for most well-trained neural nets. But, at least in theory, it should *not* hold for autoencoders, and probably many generative models (e.g. for deep fakes); because $H(X) = H(Y)$, representing *one-to-one* relationships, i.e. $f : X \mapsto Y$ is *bijective*.

Thus, for autoencoders, or any $D(X,Y)$ where $H(X) = H(Y)$, the inequality becomes an equality (ideally)

$$H^*(X;Y) = H^*(Z_1;Y) = \cdots = H^*(Z_{k-1};Y) = H^*(Z_k;Y) \tag{5.5}$$

For such problems, the best the DNN can do is to extract a *minimal sufficient statistic* for X in the encoder part of the network. This essentially boils down to getting rid of redundant dimensions.

5.3.5 Information Theory

Taking a view from information theory, in the vein of Tishby et al. [133, 143], we can state a number of things. Of course, this all requires that we assume that all the representations, X, Z_i, and Y, in Equation 5.1 are random variables. And that the sequential map itself is a Markov chain. *This assumption is probably debatable, but we'll play along.*

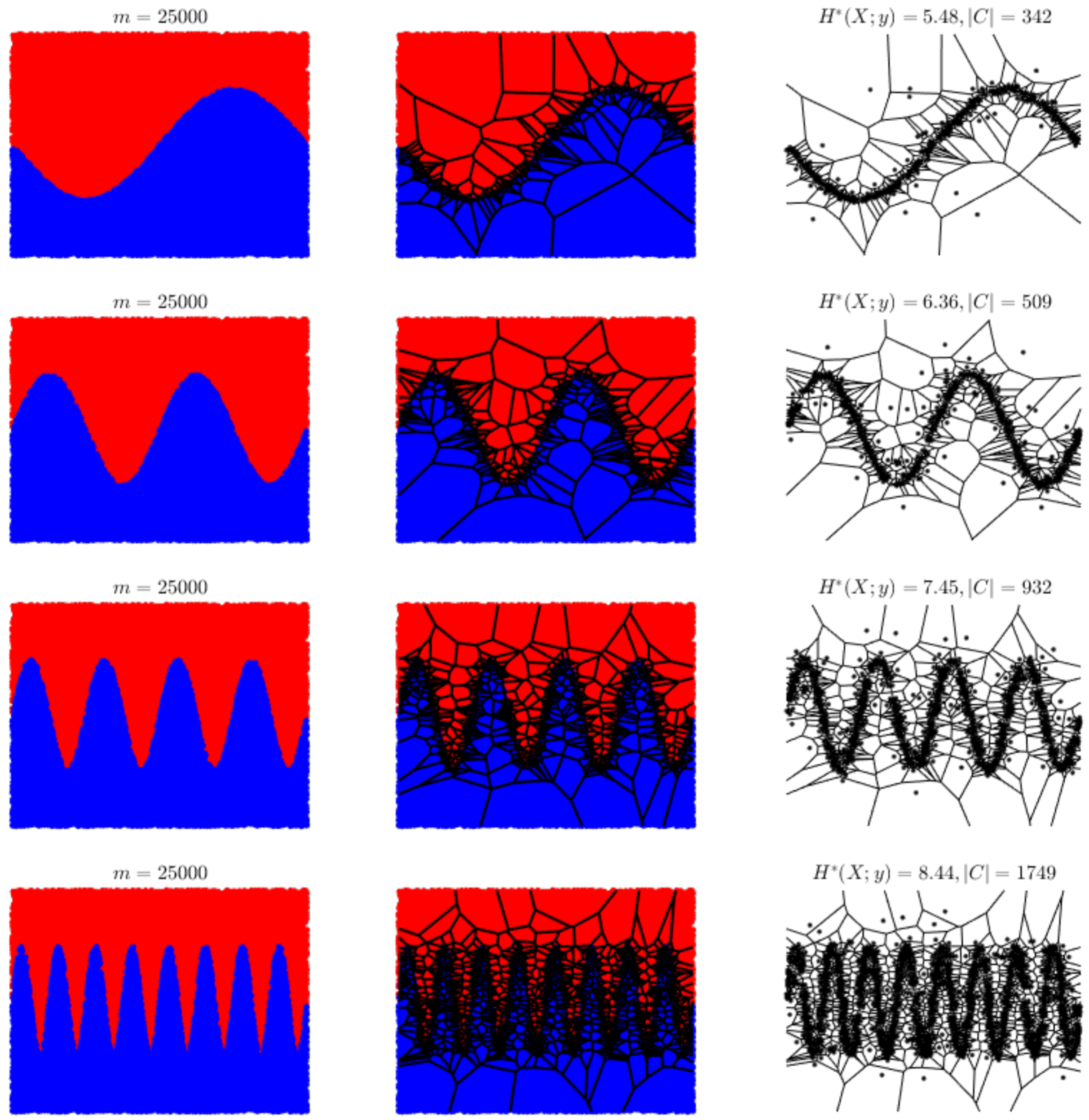


Figure 5.11: **The pure (but not optimal) Voronoi maps & entropies for sinusoidal decision boundaries of frequencies 1, 2, 4, and 8.** The number of necessary centroids, $|C|$, increases with the frequency of variation, and so does the pure entropy, $H^*(X;y)$. Thus, capturing the complexity of the learning problems. **Left:** the dataset, $\mathcal{D}(X,y)$. **Middle:** the pure Voronoi map, $V(\mathcal{D})$, on top of $\mathcal{D}$. **Right:** $V(\mathcal{D})$ shown with the centroids.

Mutual Information

Firstly, as pointed out in [143], due to the data processing inequality (DPI), we have

$$I(Y;X) \geq I(Y;Z_1) \geq I(Y;Z_2) \geq \cdots \geq I(Y;\hat{Y}) \tag{5.6}$$

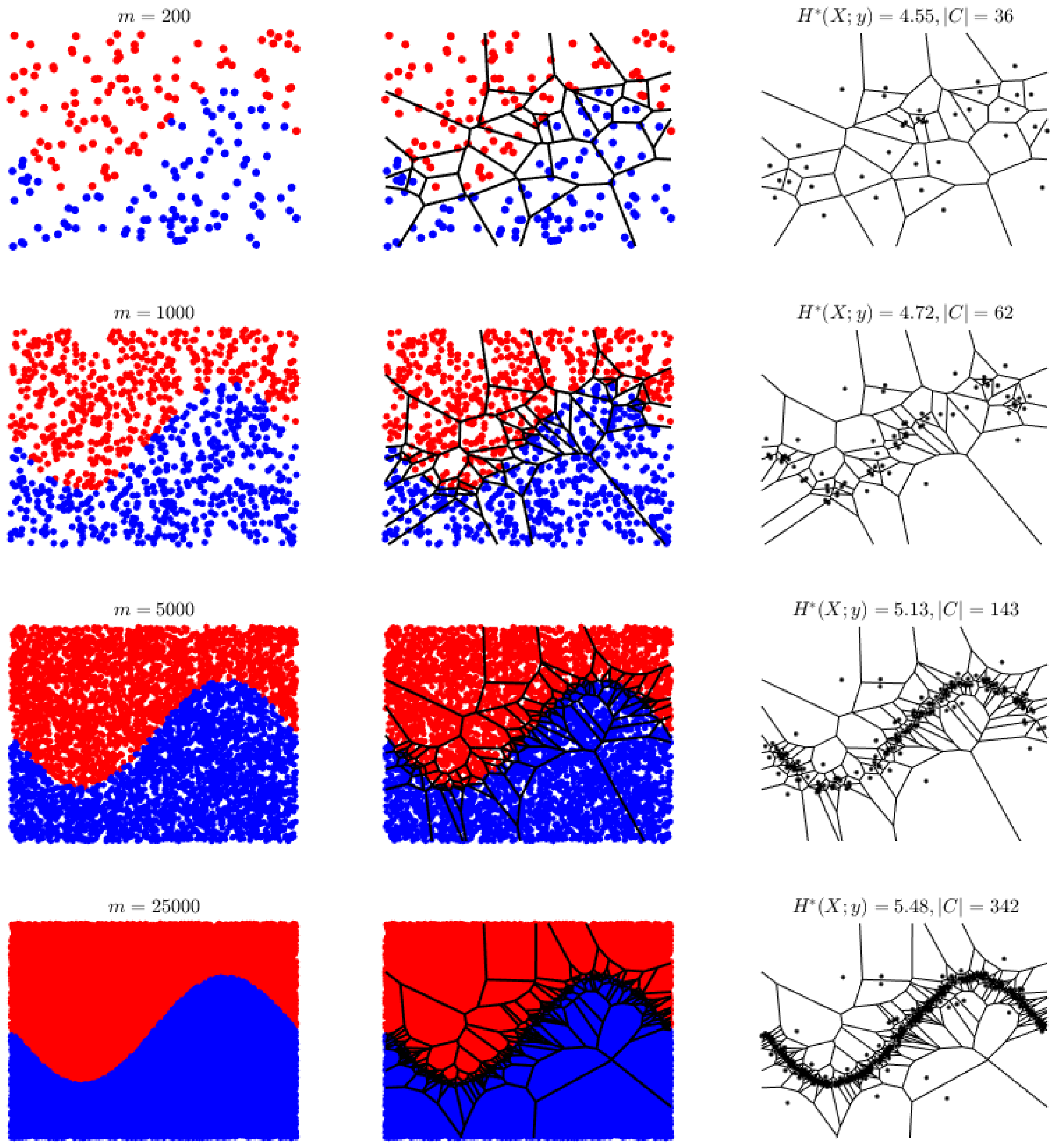


Figure 5.12: **Illustrating the effect of sample density on the pure Voronoi map & entropy for a sinusoidal decision boundary of frequency 1.** The number of necessary centroids, $|C|$, increases with the number of samples, m, and so does the pure entropy, $H^*(X; y)$. As one would expect, the centroids center near the decision boundary. **Left:** the dataset, $\mathcal{D}(X, y)$. **Middle:** the pure Voronoi map, $V(\mathcal{D})$, on top of $\mathcal{D}$. **Right:** $V(\mathcal{D})$ shown with the centroids.

Which says that any information lost about Y in Z_i can never be recovered in any subsequent $Z_{j>i}$

Conditional Entropy

Secondly, we can apply the same argument as for Equation 5.4. Namely, that for a classification problem, or any $D(X, Y)$ where $H(X) > H(Y)$, the following should hold for a well-trained neural net

$$H(X) > H(Y|X) > H(Y|Z_1) > \cdots > H(Y|Z_{k-1}) > H(Y|Z_k) \approx H(Y) \tag{5.7}$$

This is notably similar to Equation 5.4. In other words, **the pure entropy is a proxy for the conditional entropy**:

$$H^*(X;Y) \simeq H(Y|X) \tag{5.8}$$

Representational Entropy

Using the DPI and Equation 5.7, we can deduce another fundamental relationship. Again, we assume that $H(X) > H(Y)$:

$$H(X) > H(Z_1) > \cdots > H(Z_{k-1}) > H(Z_k) \approx H(Y) \tag{5.9}$$

This follow logically from the assumption, $H(X) > H(Y)$; since the Z_i's approach Y so must their entropies. It also follows from the relationship, $I(Y;X) = H(Y) - H(Y|X)$, and Equations 5.6 & 5.7. Interestingly, Jacobsen et al. [73] showed that while this discarding of irrelevant information does happen, *it is not a necessary condition for learning representations the generalize well.*

Information Filtering

The above three observations from information theory comprise the **information filtering view** on deep learning:

> *The goal of each layer (or representation) is to filter away information in Z_i, that does not carry information about the label, Y.*

Therefore, the amount of information *decreases* with every layer. This might be somewhat counter-intuitive. For example, thinking about the layers as a *feature hierarchy*, may leave one with the impression that information actually builds up through the layers. That is deceptively misleading! **What increases is *not* the information, but our *certainty* about the label:**

> *When the certainty goes up, the uncertainty goes down, and consequently so does the entropy (AKA the information).*

5.4 k-Filtering & k-Redundancy

In line with the observations above, that the classes gradually separate and that the pure entropy decreases (Equation 5.4), we propose a simple method for removing redundant samples before or during training. We use the following **redundancy criterion**:

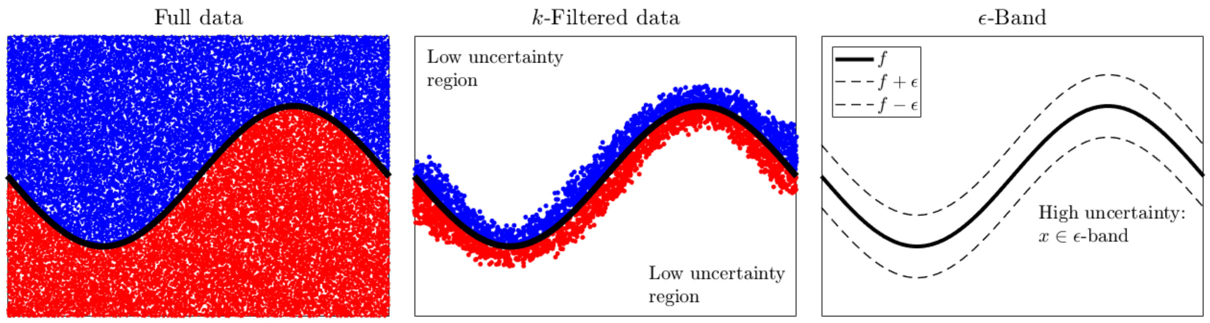


Figure 5.13: Illustrating the effect of k-filtering on toy data. Removing the examples whose k or more nearest nearest neighbors belong to its own class, has an effect similar to that of removing examples whose distance to the decision boundary, f, is higher than some value, ϵ. Conceptually, we can thus think of k-filtering as leaving an ϵ-band of data points around the decision boundary. The points inside this ϵ-band are associated with high uncertainty (AKA information) about the class label.

> *A sample is considered* ***redundant*** *if all of its k nearest neighbors belong to its own class.*

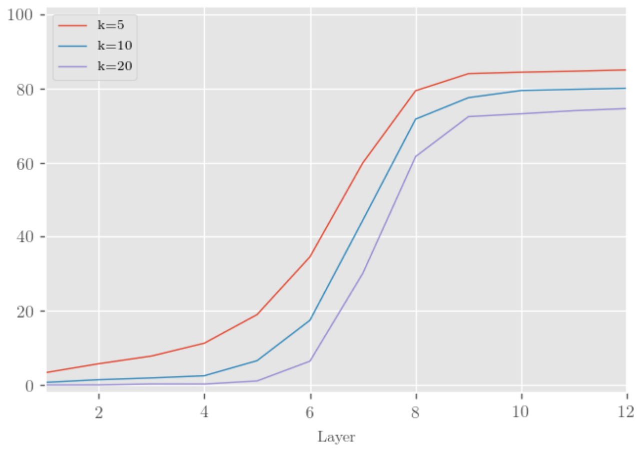
Intuitively, an example positioned far away from the decision boundary, such that all the other examples in its neighborhood belong to one single class, has little to no *uncertainty* attached to the question of what its own label should be. Conversely, an example placed close to the decision boundary, with neighbors belonging to multiple different classes, has *high* uncertainty; i.e. it belongs to a high *entropy* region (or neighborhood). Thus, such examples carry more *information* about the decision boundary. More precisely,

> *The shortest distance from an example to any point on the decision boundary is directly indicative of how much information the example carries about the decision boundary (and the function that must be learned). Specifically, being close to the boundary indicates that a sample belongs to a region in input space where the conditional entropy, $H(Y|X)$, is high. There is thus a notion of* ***locality*** *assigned to the concept of entropy.*

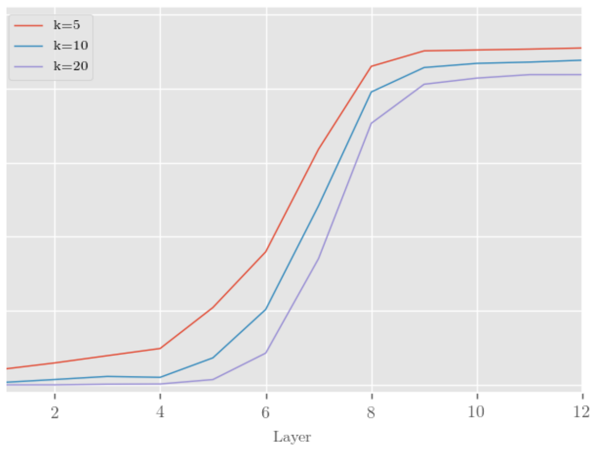
In practice, ***we run KNN on large mini-batches of the data, and remove samples that satisfy the redundancy criterion.*** Conceptually, we can think of k-filtering as leaving an ϵ-band of data points around the decision boundary. Figure 5.13 illustrates the idea.

Definition 8 (k-redundancy)**.** *For a dataset, $\mathcal{D}(X, Y)$, the k-redundancy is the percentage of samples that were removed by the k-filtering algorithm for a given choice of k.*

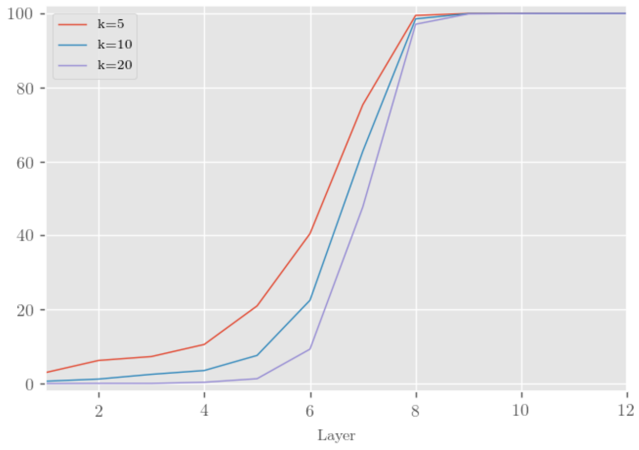
The **k-redundancy** makes for a simple method for quantifying something meaningful about the individual layer representations. Figure 5.14 shows the layerwise redundancies recorded during the training of a VGG13 convnet on the CIFAR10 dataset. As we can see, the recording *confirms* the inequality, Equation 5.4, by saying that the k-redundancy increases monotonically, layer by layer. **Meaning, that the probability of a sample being surrounded by mostly other samples from its own class *increases*. Consequently, the pure entropy *decreases*.**



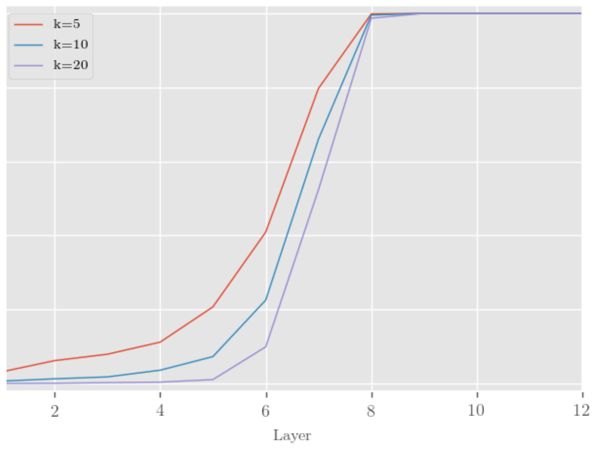
(a) Epoch 40



(b) Epoch 120



(c) Epoch 200



(d) Epoch 280

Figure 5.14: The k-redundancy of the 12 hidden layers of the VGG-13 architecture after 40, 120, 200, and 280 epochs of training on the CIFAR10 dataset, for $k = 5$, 10 and 20. The k-redundancy grows slowly in the lowest layers, but very rapidly in the middle layers. After 200 epochs, the k-redundancy of the top five hidden layers is nearly 100%. Thus, the k-redundancy looks like a sigmoidal function of the layer index.

A related observation was reported in [119]. For the CIFAR-10 dataset, the authors showed that a 1-nearest neighbor classifier applied to the learned representations achieved increasingly higher accuracy, layer by layer.

5.4.1 A Few Experiments

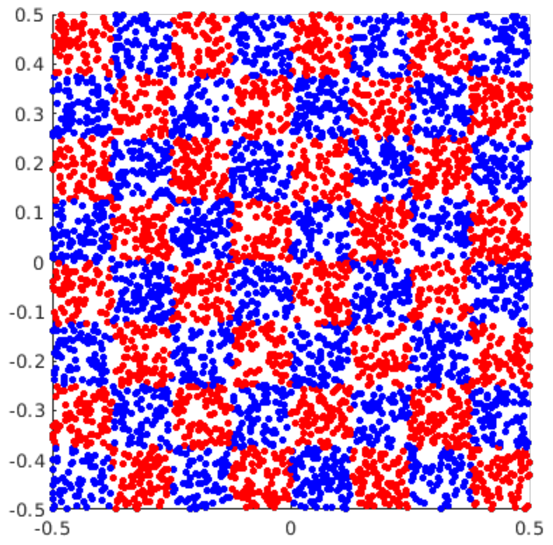
The well-known MNIST dataset, and similar ones like KMNIST, and FashionMNIST [22, 90, 152], are so redundant that 30-60% of the examples can be filtered away before training—without significantly impacting the final test accuracy. Table 5.1 lists the results we got when k-filtering the datasets before the training began. We tested the method on two architectures, a vanilla 4-layer CNN and VGG-13 [135], and for a wide range of choices for k.

Dataset	Baseline	k*	k=5	k=10	k=20
Vanilla CNN-4					
MNIST	99.54(100%)	**k=24: 99.55(57%)**	97.17(18%)	99.43(34%)	99.50(53%)
FashionMNIST	92.02(100%)	**k=42: 92.25(80%)**	90.83(42%)	91.55(56%)	91.89(69%)
KMNIST	97.09(100%)	**k=25: 97.40(69%)**	97.17(24%)	97.25(42%)	97.29(62%)
VGG-13					
MNIST	99.60(100%)	**k=27: 99.61(61%)**	99.20(18%)	99.48(34%)	99.53(53%)
FashionMNIST	93.41(100%)	**k=104: 93.65(92%)**	92.37(42%)	93.15(56%)	93.27(69%)
KMNIST	98.26(100%)	**k=9: 98.76(39%)**	98.63(24%)	98.58(42%)	98.45(62%)

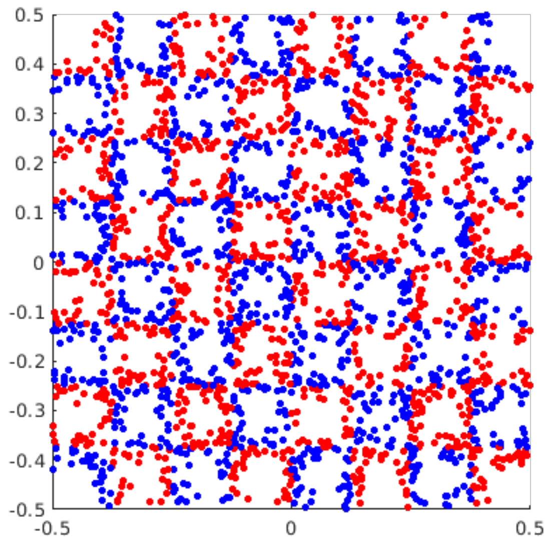
Table 5.1: **k-Filtering Results.** Test accuracies achieved with pre-filtered data, for different choices of k, with two different convnet architectures: a vanilla 4-layer CNN, and VGG-13. k* is the choice of k that produced the highest test accuracy. The percentages in parentheses show the fraction of the dataset that was used for that training run.

Considering the amount of MNIST-models trained over the past almost three decades, it is quite staggering to think about how much time and energy could potentially have been saved (especially when computers were much slower than today). Other datasets, like CIFAR10 & CIFAR100, are not redundant at all. Rather, they are under-sampled, which is why we rely heavily on data augmentation in order to get good results.

5.4.2 The Checkerboard Conundrum



(a) Full Dataset



(b) k-Filtered Data

Figure 5.15: **The checkerboard classification problem:** given an (x, y)-coordinate, what is the color (class) of the square it belongs to? Showing both the full dataset (left) and the k-filtered one (right).

In our work, we rather stumbled upon a bit of a conundrum. Namely, the checkerboard problem. As depicted in Figure 5.15, it is a binary classification problem where, given an (x, y)-coordinate, the task is to label it as belonging to either a red or a blue square. The data consists of 5,000 points sampled randomly from an 8×8 checkerboard. There was no noise added to the data, and we do not worry about generalization, we just want to train an MLP to essentially memorize the points. Clearly, this is a very simple task. What makes it interesting is that, against all intuition, **no good solution can be found by backprop + SGD.** This is very surprising, because this problem is a cakewalk for a decision tree or an SVM. A binary decision tree needs just 64 levels to get zero error on the (training) data; one for each of the 64 squares. So, it seems just wrong to say that an MLP cannot be trained to at least very high precision on the same data? However, that was exactly the case! No matter what we tried, with respect to the hyperparameters, number of layers, activation functions, and choice of optimizer, the learning always got stuck and around 51-53% classification accuracy.

The Problem: Symmetry

Our analysis eventually led us to visualize the error gradients *in the input space*. And that clearly illustrated what was going on. The points *and the classes* were sampled uniformly on the interval $[-0.5, 0.5]$ (along both axes), and **the learning problem is symmetric around both diagonals.** This causes a "face-off" between directly opposing gradients along the two diagonals in input space; as depicted in Figure 5.16. The magnitude of these gradients dominate the optimization problem, which causes the optimizer to get stuck.

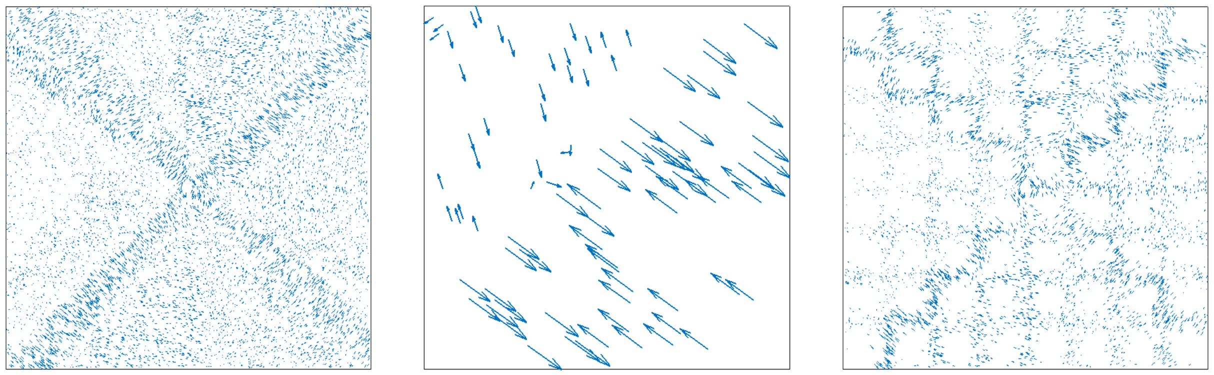


Figure 5.16: Vector field of the error gradients (as they look in input space) of a two-layer MLP, while training with SGD + backprop on the checkerboard problem. **Left:** directly opposing gradients appear along the diagonals when training with the full dataset. **Middle:** zooming in on a diagonal (in the left figure), clearly showing the gradient "face-off". **Right:** filtering the center points of each square away reduces the dominance of the diagonal s.t. the optimizer doesn't get stuck.

However, as shown in Figure 5.16 (right), after k-filtering the data, the dominance of the gradients along the diagonals is reduced. Consequently, the optimization does not get stuck during training, and now an MLP can be fitted to arbitrary precision.

This result begs the question: does there exist a class of similar (symmetric) learning problems, that cannot be learned by neural networks, or gradient descent, *without filtering the data?* It is easy to imagine other strategies for removing samples. In this instance, we could simply remove examples along the diagonals. However, in the general case the gradients would not line up in such an obvious way, and deciding what examples to remove could get complicated. k-filtering offers a simple and relatively safe way of removing redundant (and possibly problematic) samples.

5.5 *How* Neural Networks Learn

In Section 5.3, we presented some alternate views on *what* the layers of a deep neural network learns. Our observations led us to develop a rather useful way of viewing *how* they learn. We call it: *the correlation filter view.*

5.5.1 The Correlation Filter View

Most, if not all, machine learning algorithms find structure in data by utilizing some notion of *similarity* or *locality*. kNN selects examples by counting the labels of their k nearest neighbors. k-means assigns examples to clusters based on their (Euclidean) distance to k centroids. And a decision tree gradually selects examples based on how they fall into smaller and smaller bins (intervals) of individual features. The fundamental operation applied in neural networks for quantifying similarity is that of *correlation*.

5.5.2 Correlation

The type of correlation in question is not one of *causality*, or the correlation between events in *time*. Rather, it measures correlation in *space or direction*. In an MLP, it is expressed as the **dot product** between an input vector and a weight vector. Formally, the term refers to slightly different but related things in different fields. The one most resembling our notion of correlation is cross-correlation.

For two discrete-time signals, f and g, the **cross-correlation** is defined as

$$R_{fg}(n) = \sum_i f[i]g[i-n] \tag{5.10}$$

Note that the (one-dimensional) convolution differs only from the cross-correlation by a reversal of the order of the indices into the second signal, g. Thus, the convolution of f by g is given by $f * g = C_{fg}(n) = \sum_i f[i]g[n-i]$. In other words, cross-correlation is simply a convolution where the second signal is time-reversed: $R_{fg} = f[i] * g[-i]$.

In statistics, if two random variables X and Y have zero mean, then their covariance is their dot product, $Cov(X, Y) = \mathbb{E}[X \cdot Y]$. **Their correlation is the cosine of the angle between them.**

As the cosine of the angle between vectors x and w is given by their dot product divided by the product of their norms, we can think of $x \cdot w$ as the "un-normalized" correlation (or cosine similarity). If x and w are positively correlated, $x \cdot w > 0$, then the angle between them is acute, if the correlation is negative, $x \cdot w < 0$, the angle is obtuse, and if they are uncorrelated the two vectors are orthogonal and $x \cdot w = 0$. *Note that we usually omit the $\cdot$ when writing the dot product, but we have included it in this section for clarity.*

Geometry & Sorting

Figure 5.17 shows the correlation heatmap for a vector, w. For any vector, x, that points into the half-space denoted S_+ the correlation is positive, and vice versa for the half-space denoted S_-. For neural nets, we often think of x as a *point* and of w as a *weight vector*. Thus, S_+ is the set of *points* such that $\forall x \in S_+ . xw > 0$, and conversely for S_- we have $\forall x \in S_- . xw < 0$.

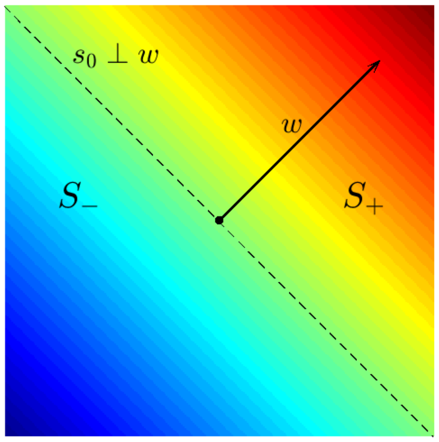


Figure 5.17: **Correlation heatmap.** Showing the correlation (dot product) heatmap for the vector w (red means high/positive, and blue means low/negative correlation). s_0 is the vector (or plane, in higher dimensions), orthogonal to w, of zero correlation. S_+ and S_- denote the half-space of positive ($x \cdot w > 0$) and negative ($x \cdot w < 0$) correlation, respectively. Adding a bias, shifts s_0 along the direction of w, and is thus equivalent to shifting the *activation threshold* (of e.g. a ReLU).

Geometrically, for a set of training examples contained in the rows of a matrix, $X \in \mathbb{R}^{m \times n}$, and a weight vector, $w \in \mathbb{R}^{n \times 1}$, the matrix-vector product, $a = Xw$, denotes a linear *projection* of the examples, $x_i \in X$, onto the real line. The order in which the examples, x_i, fall onto $\mathbb{R}$ is exactly decided by their *correlation* (or angle) with w. Thus, computing $a = Xw$ is akin to **sorting the examples by their correlation** (or "similarity") with the weight vector, w.

5.5.3 Filtering

For a single feature, $z_i = \sigma(Xw_i)$, if σ is the ReLU and the bias is zero, then all $x \in S_-$ are mapped to 0, while all $x \in S_+$ are mapped to $\mathbb{R}^+$. Consequently, for all the points, $x \in S_+$, the activation map, $\sigma : x \mapsto z$, is **bijective**; i.e. it is *invertible* and represents a *one-to-one* relationship. However, for the points, $x \in S_{-}$, it is **injective**, not invertible, and denotes a *many-to-one* relationship. Such *many-to-one* mappings, employed by the large flat regions of the commonly used activation functions, are effectively **filtering the input.**

These regions are characterized by having zero, or nearly zero, first derivatives. Thus, they also act as filters on the gradient signals during backpropagation. That is, all inputs that fall into a flat region of any $\sigma(\cdot)$ will lead to a multiplication by zero (or a very small number) in the gradient-backpropagation phase.

Please note that the filtering, as we refer to it in this context, is not strictly equivalent to losing (or filtering away) *information*. From the point of view of information theory, this not guaranteed to happen from merely applying an activation function with flat regions. In practice, the activation functions will almost certainly play a role when actual information is filtered away in a DNN. But so will any dimensionality reduction applied by narrow layers. For example, by reducing the number of dimensions to below the *intrinsic dimensionality* of the data. Conversely, if there is a massive dimensionality increase in a layer, then even if most samples fall into the flat regions of σ, there is no guarantee that information will indeed be lost (in that layer as a whole).

The filtering, that we are referring to here, is the filtering away of individual positions along a single dimension or feature. A set of samples, or inputs to σ, that are all mapped to the same value by σ, become *indistinguishable* from each other (in that dimension). This allows the following layer to treat them as one (again, in that dimension) such that they can easily be ignored entirely, or may all contribute in the exact same way to an upstream computation.

We should add that, strictly speaking, most of the commonly used activation functions, such as the squashers (the hyperbolic tangent and the logistic sigmoid), are actually fully invertible. However, in practice, that does not matter to our argument. Firstly, *in practice*, they are not fully invertible because their inverses yield complex numbers in the flat regions. Secondly, if you move far enough out in the asymptotically flat regions, the differences in the output of those functions will be so infinitesimal that, *in practice*, it makes no real difference to the computations in a neural net.

5.5.4 The Process: Sort & Filter

According to the correlation filter view, the mantra a neural network lives by is thus: to **"sort & filter"** the input. The fundamental process that enables a network to solve the Rubik's cube (Section 5.3.4), so to speak, is therefore:

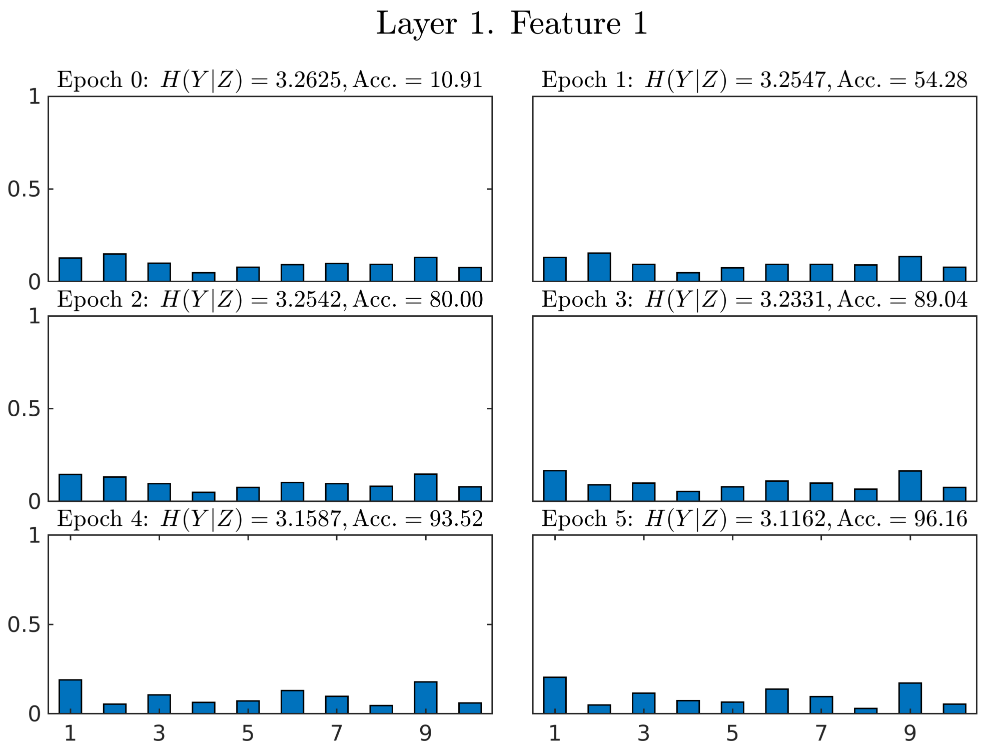
> *For each feature, $z_i = \sigma(Xw_i)$, to sort and filter the input such that same-class examples line up next to each other, as much as possible, along each of the feature-dimensions.*

Selection

It is useful to distinguish between the inputs that get filtered, and those that do not. We will use the terms **"deselected"** and **"selected"**, respectively. The *selection* of a feature, $z = \sigma(Xw)$, shall be denoted $S_{+}(z)$ and refer to the subset of examples (row vectors), $x_i \in X$, whose projections onto $\mathbb{R}$, $a_i = x_i w$, fall *outside* any flat regions in the activation map, $\sigma : \mathbb{R} \mapsto \mathbb{R}$ (i.e. they pass through the activation *unfiltered*). Conversely, the *deselection* of z shall be denoted $S_{-}(z)$ and refer to the subset of examples whose projections onto $\mathbb{R}$ fall *inside* a flat region of σ (i.e. they are *filtered*).

Arguably, we can now view the job of any feature as that of *selection* (through sorting and filtering). That is, to select a subset of all the samples from the training data, $S_{+}(z) \subset X$, such that the entropy over the class distribution is lowered. In other words, we could interpret it to mean that $H(Y|S_{+}(z)) < H(Y|X)$. And similarly, that $S_{+}(z)$ has higher k-redundancy and lower *pure* entropy—in that one feature-dimension (see Sections 5.4 & 5.3.4). Again, relating back to the Rubik's Cube view of Section 5.3.4, and to Tishby et al. with their Information Bottleneck view [133, 143], the word "selection" is but a synonym for "binning".

To support our view, we recorded some feature-wise class distributions from the selections of a 3-layer MLP (784-300-50-10 with ReLUs) over 5 epochs of training on the MNIST dataset. The results are shown in Figures 5.18, 5.19, and 5.20, epoch 0 refers to the state *prior* to training (i.e. the random initialization). In the first layer, for the first feature $H(Y|Z)$ goes from 3.26, at the onset of training, to 3.12 after the fifth epoch. In layer 2, $H(Y|Z)$ drops from 3.26 to 2.55 (feature 1), while in the third layer $H(Y|Z)$ goes from 3.03 to 0.09 (also feature 1).



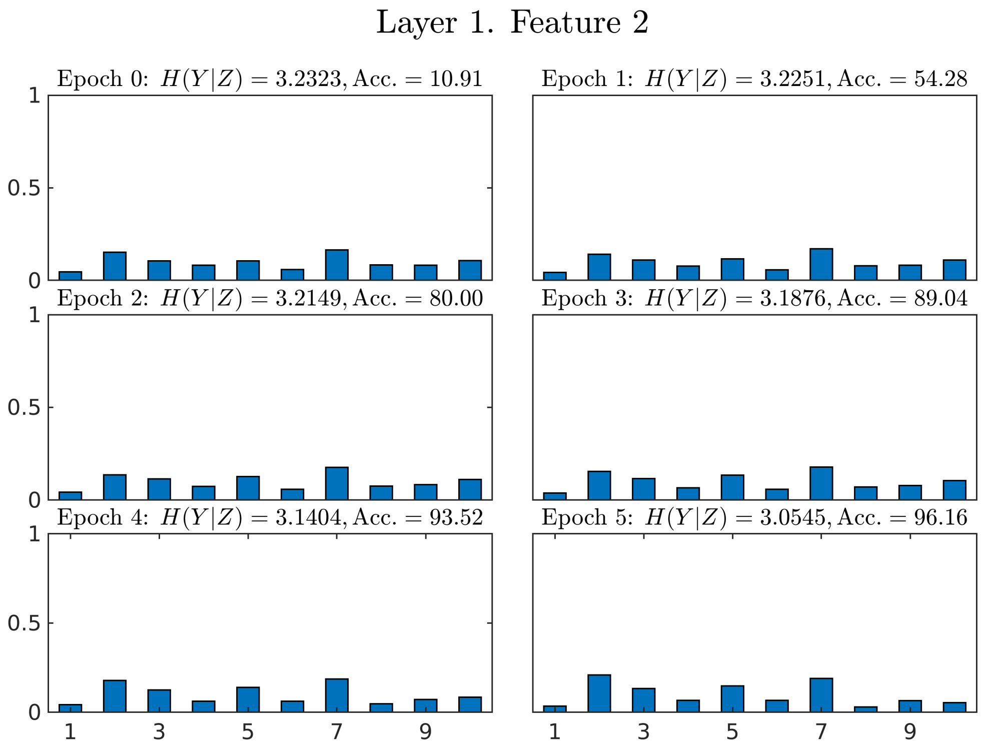


Figure 5.18: **Layer 1 of a 3-layer MLP.** The class distributions of the selections of two features over the first five epochs, the corresponding feature-wise conditional entropies, $H(Y|Z)$, and the total classification accuracy of the model at that epoch. Epoch 0 refers to the state *prior* to training (i.e. the random initialization).

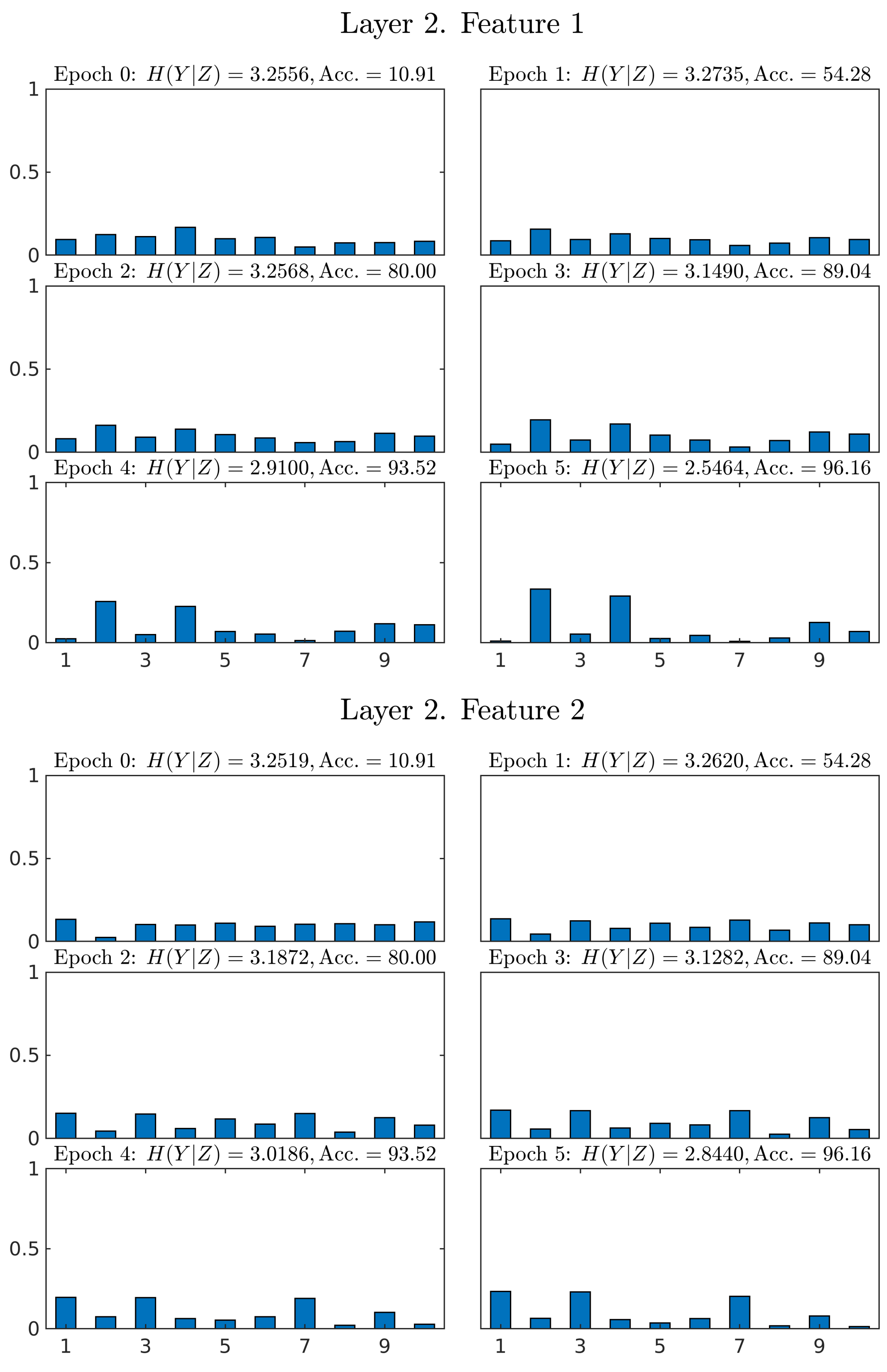


Figure 5.19: **Layer 2 of a 3-layer MLP.** The class distributions of the selections of two features over the first five epochs, the corresponding feature-wise conditional entropies, $H(Y|Z)$, and the total classification accuracy of the model at that epoch. Epoch 0 refers to the state *prior* to training (i.e. the random initialization).

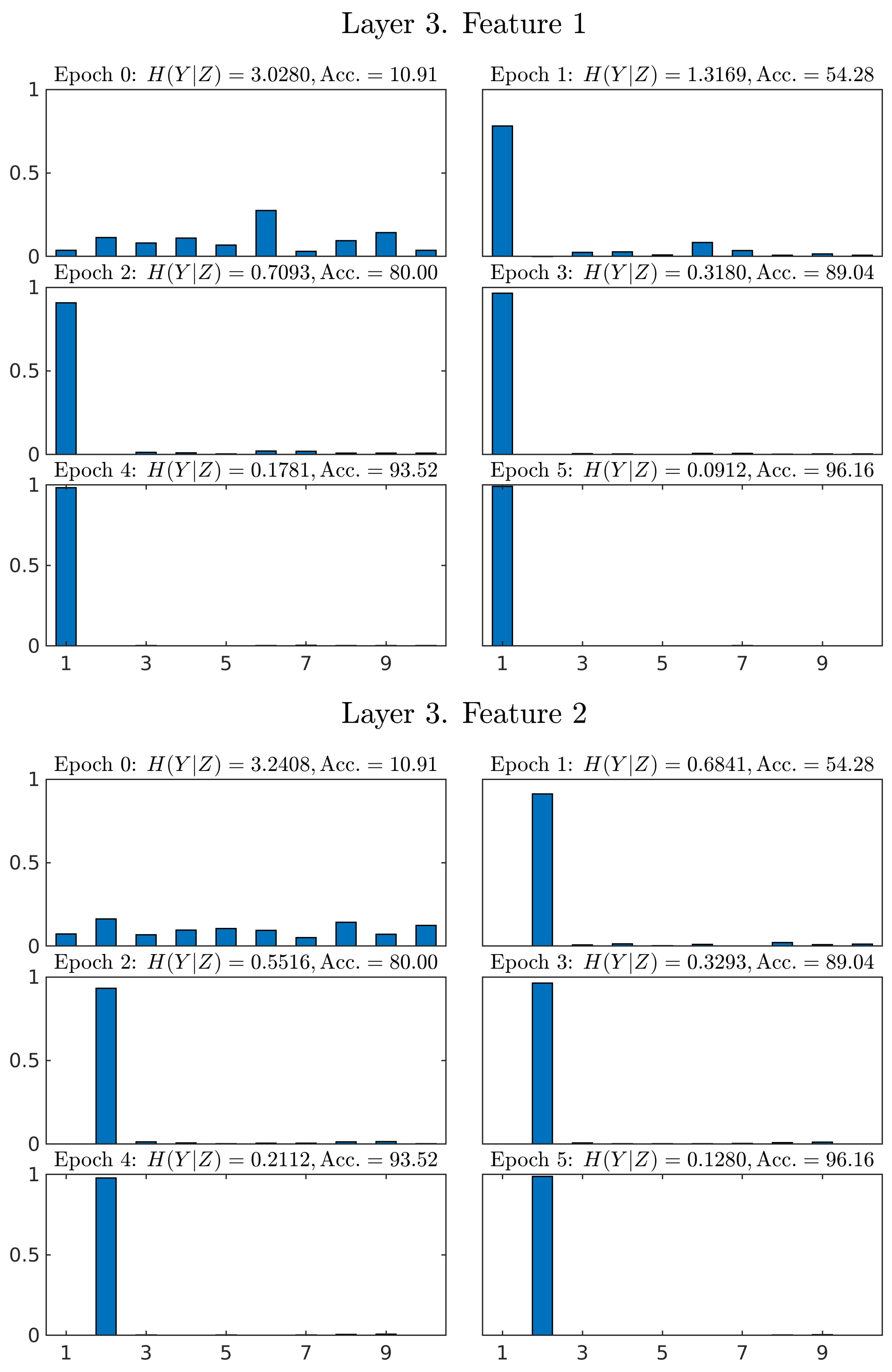


Figure 5.20: **Layer 3 of a 3-layer MLP.** The class distributions of the selections of two features over the first five epochs, the corresponding feature-wise conditional entropies, $H(Y|Z)$, and the total classification accuracy of the model at that epoch. Epoch 0 refers to the state *prior* to training (i.e. the random initialization).

In addition, we trained 4-layer MLP (784-392-196-50-10 with ReLUs) for 50 epochs on the MNIST dataset while recording $H(Y|Z)$ for each of the hidden layers. The result is shown in Figure 5.21. For the i'th layer, $H(Y|Z_i)$ was computed as the average entropy over the class distributions of the selections of all the individual features in the layer. In order to produce the plot, we deliberately used a very low learning of 3e-06.

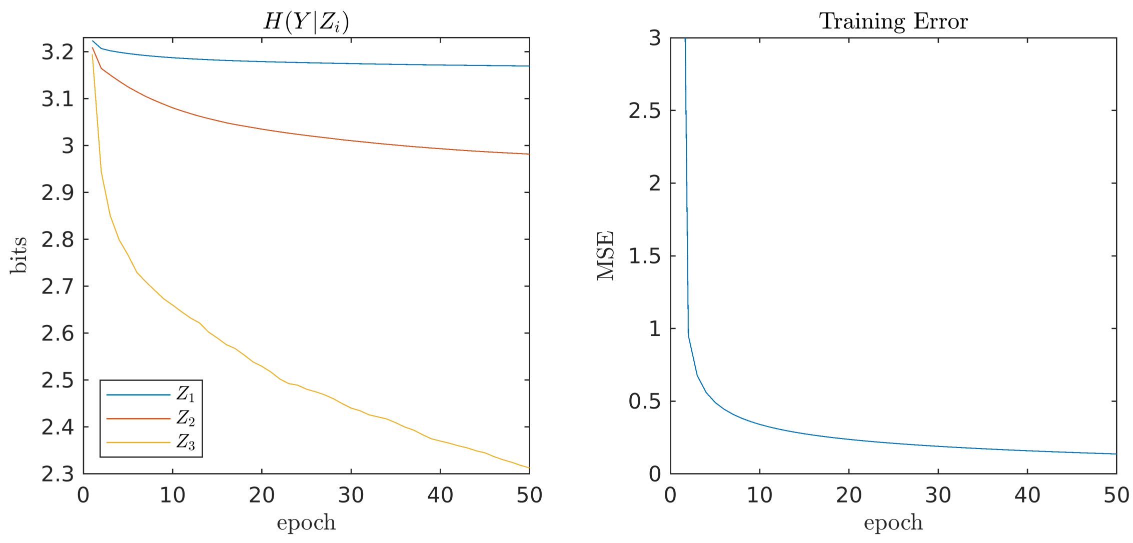


Figure 5.21: **Layer-wise Conditional Entropies. Left:** $H(Y|Z_i)$ recorded for each hidden layer over 50 epochs of training. **Right:** The mean squared error (MSE). $H(Y|Z_i)$ is computed as the average entropy over the class distributions of all the features in the i'th layer (i.e. their selections, $S_+(z_j)$). The 4-layer MLP, with ReLU activations, was trained on the MNIST dataset. **Notice that the layer-wise entropies, $H(Y|Z_i)$, closely resemble the situation depicted for the least squares distance (LSD) in Figure 5.6.**

While these results alone are by no means sufficient to make any strong claim about the validity of our view, they clearly favor it. Also, they are consistent with our results on the k-redundancy and k-filtering (Section 5.4), as they are essentially expressing the same thing—just at the feature level instead.

Discriminative Intervals & Locality of Entropy

While we suspect $H(Y|S_+(z)) < H(Y|X)$ to hold pretty generally, one can also imagine situations where it does not—at least not for *all* of $S_+(z)$. For example, it could hold only for a subset of the selection that lie on the same interval. If σ is the ReLU (i.e. all positive inputs are selected), then as we move away from zero along an interval $(0, b]$ in the image of σ, we may encounter mostly examples from a single class, such that the conditional entropy *over that interval* is low. Moving beyond the point b, we may then observe the opposite: examples from many different classes lined up next to each other. Meaning that, over the interval, $(b, \infty]$, the corresponding class distribution is highly uniform; making the conditional entropy high. **We may say, that $(0, b]$ is a *discriminative interval*,** while $(b, \infty]$ is not.

Figure 5.22 illustrates the idea of such discriminative intervals. In the depicted example, applying the ReLU would select the most discriminative interval, $(0, b]$. Similarly to Figure 5.13 of Section 5.4, this illustrates that there is a notion of **locality** assigned to the concept of conditional entropy.

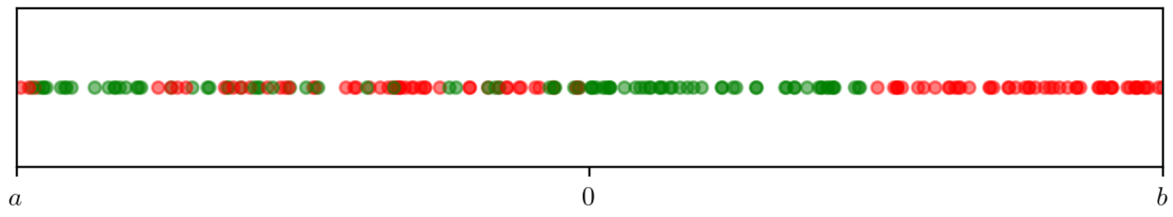


Figure 5.22: **Discriminative interval.** Illustrating how input examples may fall in the interval $[a, b]$, as they are projected onto the real line via a weight vector, w. Along the interval, $[a, 0]$, the label (color) of the samples changes frequently. We associate this interval with *high* conditional entropy, $H(Y|X)$. Conversely, the interval $(0, b]$ is associated with *low* conditional entropy; i.e. *locally* in the interval the average uncertainty about what the label should be is low. In other words, $(0, b]$ is more **discriminative** than $[a, 0]$.

5.5.5 The Learning: Probe Vectors

There is an interesting analogy with our view and the Fourier transform.

The Fourier Transform & The Probe Phasor

For a *continuous* signal, or function, the Fourier transform is defined as:

$$X(f) = \int_{-\infty}^{\infty} x(t)e^{-i2\pi ft}\mathrm{d}t \qquad (5.11)$$

$e^{-i2\pi ft}$ is sometimes referred to as the *kernel* function of the Fourier transform, but generally in physics and complex number theory it is called a *phasor*. It relies on *Euler's formula*, $e^{i\theta} = \cos\theta + i\sin\theta$, and can be thought of as a **probe phasor**; a pulse with which the signal, $x(t)$, is probed for its energy at frequency f. The probing is done by computing *the correlation* between the signal and sinusoids of different frequencies (and phases).

Probe Vectors

The process of sorting and filtering, that each single weight vector applies to its input, lends us a new view on weight vectors, namely as **probe vectors**. Just as the probe phasor of the Fourier transform probes a signal for the presence of frequencies:

> *Weight vectors* ***probe*** *their input space for* ***discriminative directions.*** *That is, for directions that yield discriminative selections.*

A discriminative direction in the i'th layer is thus any choice of w_j such that $Z_{i-1}w_j \in \mathbb{R}$ contains a discriminative interval that can be selected by the activation function of choice, σ. Preferably, the cardinality of the discriminative selection, $|S_+(\sigma(Z_{i-1}w_j)|$, is large.

To elaborate further on the Fourier transform as an analogy, we can take a look at Figure 5.23. It shows an *imaginary labelling function*, ℓ, assigning labels to two classes over an interval on the real line. $\ell(x)$ is a smooth function that oscillates around 0, and its sign determines the class labels, $y_i = \text{sign}(\ell(x_i))$. In consequence, the frequency components contained in the signal, $\ell(x)$, represent the **frequencies of variation in the data (i.e. their labels).**

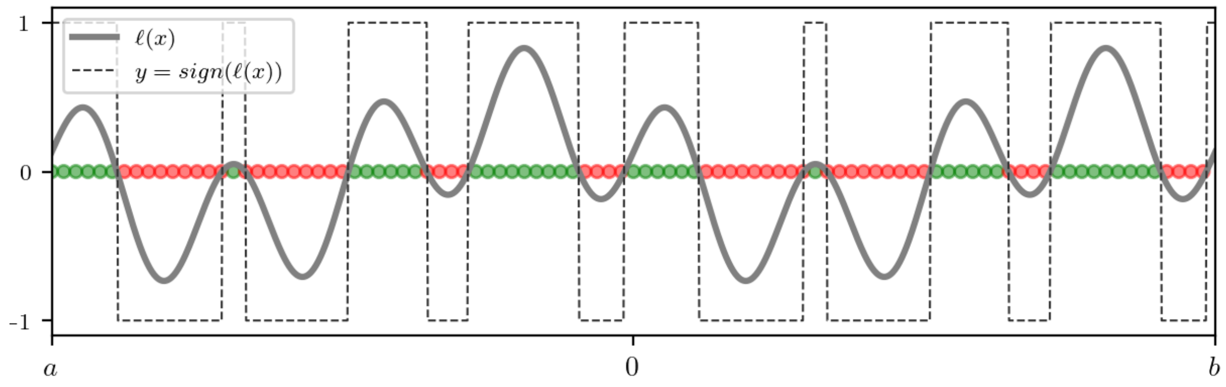


Figure 5.23: **Labelling Function & Frequency of Variation.** An imaginary labelling function, ℓ, assigning labels over an interval, $[a, b]$, for a binary classification problem s.t. $y_i = \text{sign}(\ell(x_i))$. The oscillations in ℓ thus determine the *frequency of variation* in the data/labels.

We can now think of learning as the process of identifying intervals where ℓ is dominated by low-frequency components. For example, by using the short-time Fourier transform (STFT) to compute a spectrogram of ℓ.

Learning With Probe Vectors

The process of fitting a neural network to some training data, $\mathcal{D}(X, Y)$, can now be conceptually outlined as follows:

1. Initialize the weight vectors at random s.t. that they visually would resemble a bunch of pins stuck in a ball (centered at the origin); approximately covering the surface of the ball uniformly.
2. For each vector (or "pin"), drag it around in different directions until a discriminative interval is identified.
3. Repeat the process for all feature vectors in the layer, while making sure that all the samples in X are contained in at least one feature's selection.

In step 2, we might consider adding that one should attempt to find the *most* discriminative interval possible; making it a greedy algorithm. Deciding what interval is the "most discriminative" would probably entail making some kind of trade-off between the cardinality of the corresponding selection and the conditional entropy. For instance, selecting a single example would, in the view discussed here, be equivalent to minimizing $H(Y|S_+(z_j))$. That is hardly ideal in the general case.

This process could be applied in a bottom-up layer-wise manner, but it is not required. Perhaps, picking individual weights vectors at random, from all the layers, would have some kind of regularizing effect. Regardless, our intention is not actually to train neural nets in this exact way; rather, as stated, it is a *view* on the learning process.

Finally, we note that this line of thinking bears some resemblance with that of our department's own, Scott Fahlmann (and collaborators), in his work on *the cascade-correlation learning architecture* [32, 33]. In this method, hidden units are added to an MLP and trained, one by one, in a greedy manner.

Chapter 6

Deep Layer-wise Learning

6.1 Introduction

The past decade or more has seen a consistent increase in the number of layers and parameters used in deep neural networks. This has spawned a lot of research on how to deal with the resulting increase in the complexity of the optimization problem, and how to build systems that can efficiently handle the massive data-processing involved. While the dominant trend in deep learning literature is training very large models on hundreds or thousands of GPUs, there is an increasingly important use-case for mobile and embedded systems. Such systems are typically significantly resource limited, both in compute and memory capacity. To address the latter, we propose a simple method for deep layer-wise learning (DLL). In brief, we simply extract an individual layer (or e.g. a residual block) from an architecture, attach a decision-layer (or block) on top, and train that subnetwork in the usual way with backpropagation and SGD. In the simplest case, we train each layer individually, from the bottom up, for n epochs.

DLL alleviates the vanishing gradient problem [57] by significantly reducing the number of layers that gradients must be backpropagated through. We trained a 1000-layer MLP on the MNIST dataset using greedy layer-wise learning—this is impossible with regular *full* backprop. With this, we can at most train MLPs of 15-25 layers, and besides that: the entire model would not fit in memory. DLL is also quite flexible. Arbitrary size blocks of layers can be trained at a time, such that one can maximize resource utilization by matching the memory footprint of the algorithm to the available memory. Furthermore, in cases where the full model *does* fit in memory, but only for very small batch sizes, deep partition-wise learning (DPL) may offer faster execution (whereas DLL is usually significantly *slower* than full backprop). By splitting a very deep model into a few large *partitions* (e.g. 10-40 layers or residual blocks), we allow for significantly larger batch sizes to be used during training. On a GPU, there can easily be a 10-fold speedup when training with a large batch size, compared to a very small one. Thus, there is often plenty of room for any overhead incurred by fitting the model partition by partition. Moreover, assuming that the number of partitions is *much* smaller than the number of layers in the architecture, near-optimal model-parallelization may be achieved with DPL (see Section 6.8).

Lastly, sequential bottom-up DLL is a constructive training scheme allows for the incremental addition of layers, such that the final architecture need not be known in advance. Potentially, this may lead to new methods for online architecture search, where layers (or blocks) are added and trained, as needed, until a desired precision has been achieved for the model.

6.1.1 Gradual Class Separability

Historically, neural nets have largely been considered a black box with respect to what each individual layer learns. This is due to the highly non-linear nature of the internal processing of the input data. Our method is based on the hypothesis that the classes separate gradually, layer by layer, in the hidden representations; as discussed in detail in Chapter 5. **Disregarding any implicit penalties for regularization, this would imply that the layer-local training objectives are largely the same.** We have empirically confirmed the hypothesis on a number of trained DNNs. An example of the gradual separation of classes is depicted in Figure 5.9. Please refer to Chapter 5 for more information on our motivation and theoretical justification for our proposed method.

6.2 Deep Layer-wise Learning

DLL operates on the *architecture* of a neural network. That is, the set of all properties required to sufficiently describe a particular network, such as the number of neurons and the activation functions for each layer (see Chapter 2, Definition 3).

6.2.1 The Fundamental Algorithm

Deep layer-wise learning is similar to the training process for stacked autoencoders (SAEs)[52]. Only, instead of using the identity mapping as the target in each layer—we simply use the actual target, Y. Thus, we can think of DLL as the process of stacking *target-encoders*. The canonical use-case is to train an individual layer, ψ_i, by adding a single decision-layer on top. For this, we take a dataset, $\mathcal{D}(X, Y)$, and a k-layer architecture, Ψ, and decompose the learning problem into $k-1$ two-layer sub-architectures, $\Psi^{(i)} = (\psi_i, \psi_k)$, which we train sequentially, one by one, **from the bottom up**. The hidden representation, Z_i, learned at the i'th sub-architecture, becomes the input to the next sub-architecture. Hence, the subnet specified by $\Psi^{(i+1)} = (\psi_{i+1}, \psi_k)$, is trained on the dataset $\mathcal{D}^{(i)}(Z_i, Y)$. Note, that ψ_k denotes the decision-layer, and at each subnet its input-dimensionality is adapted to match the output-dimensionality of the layer being trained. Figure 6.1 illustrates the process for a four-layer architecture.

The pseudocode for the training process is listed in Algorithm 3. This is the algorithm in its simplest form, and it expects all the layers to have parameters. However, we may choose to train more than one hidden layer at a time. For example, as in [60], we could train the equivalent of a ResNet-block at a time. And some layers could have no trainable parameters (e.g. pooling layers), in which case it makes sense to include them in the training of the last preceding parameter-layer(s). Moreover, in most cases, it is more efficient to train *segments of layers* at a time (or segments of residual-blocks). We will discuss that in Section 6.5.

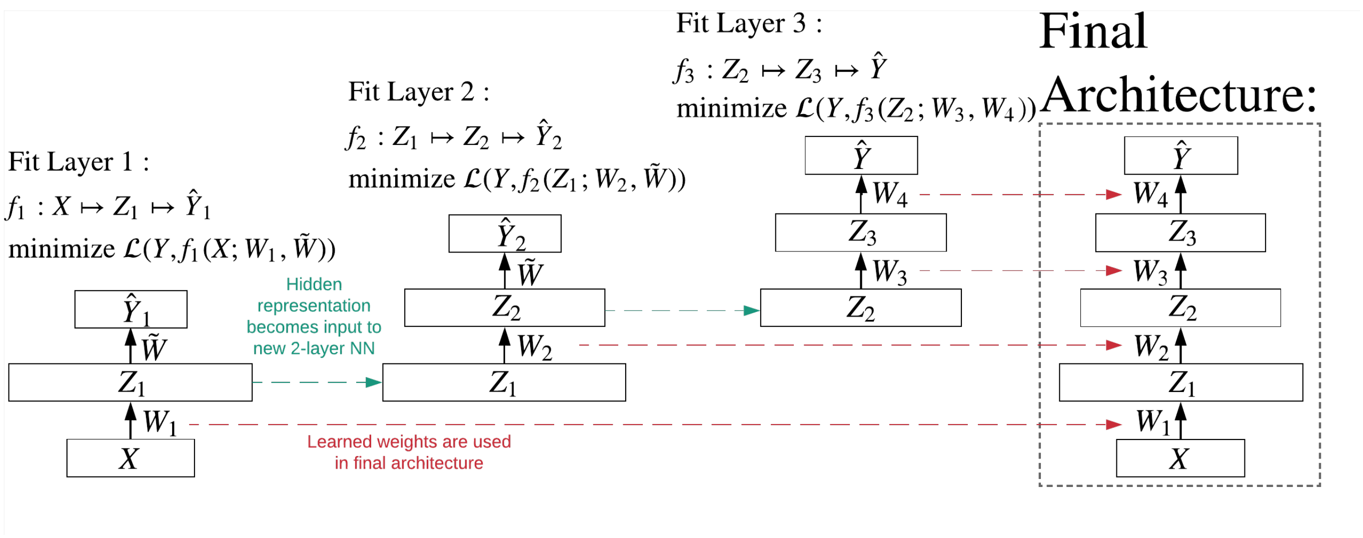

Fit Layer 1 :
$f_1 : X \mapsto Z_1 \mapsto \hat{Y}_1$
minimize $\mathcal{L}(Y, f_1(X; W_1, \tilde{W}))$
Hidden representation becomes input to new 2-layer NN
Fit Layer 2 :
$f_2 : Z_1 \mapsto Z_2 \mapsto \hat{Y}_2$
minimize $\mathcal{L}(Y, f_2(Z_1; W_2, \tilde{W}))$
Fit Layer 3 :
$f_3 : Z_2 \mapsto Z_3 \mapsto \hat{Y}$
minimize $\mathcal{L}(Y, f_3(Z_2; W_3, W_4))$
Final Architecture:
Learned weights are used in final architecture

Figure 6.1: **Layer-wise Training of a 4-layer NN:** First, fit the 2-layer NN, f_1, then discard $\tilde{W}$ and keep W_1 for the final architecture. Get the hidden representation Z_1 (by forwarding X through the first layer of f_1), and repeat the process for f_2 using Z_1 as the input. Do the same for f_3, but this time keep both set of weights, W_3 and W_4 for the final architecture.

Algorithm 3 Deep Layer-wise Learning Algorithm

```
1:  Input: Dataset $\mathcal{D}(X, Y)$, $k$-layer architecture $\Psi = (\psi_1, \ldots, \psi_k)$, and loss function $\mathcal{L}$
2:  for layer $i = 1..k-1$ do
3:      Let $\psi'_k$ be a decision-layer with input dimensionality matching the output of $\psi_i$
4:      Sub-architecture $\Psi^{(i)} := (\psi_i, \psi'_k)$
5:      Two-layer neural net $f_{\Psi^{(i)}} : X \mapsto Z \mapsto \hat{Y}$
6:      Weights $W_i, \tilde{W} := \underset{W_i, \tilde{W}}{\operatorname{argmin}} \mathcal{L}(Y, f_{\Psi^{(i)}}(X; W_i, \tilde{W}))$
7:      if $i < k-1$ then
8:          Input $X :=$ Forward$(X, \psi_i, W_i)$
9:      else
10:         Weights $W_k := \tilde{W}$
11:     end if
12: end for
13: Output: $\{W_1, \ldots, W_k\}$
14:
15: function FORWARD$(X, \psi, W)$
16:     Let $f_\psi(\cdot; W)$ denote the function specified by $\psi$, as computed using weights $W$
17:     Representation $Z := f_\psi(X; W)$
18:     Return $Z$
19: end function
```

6.2.2 Practical Concerns

Decision-block vs decision-layer

Very often the number of features in a hidden layer is so large, that using a single (fully-connected) decision-layer becomes needlessly expensive, due to a quadratic blow-up in the number of parameters. We can reduce the number of features by adding a few extra decision-layers. For example, **down-conv** or **down-pooling** layers; i.e. convolutional or pooling layers that reduce the size of the feature map, and / or the number of channels. In many cases, this uses less memory, runs faster, and gets better results. The latter is due to implicit interlayer regularization; see Section 6.6. Whenever we use more than a single decision-layer, we refer to it as a *decision-block*.

Feeding data to the training layer (or block)

For all but the first layer, we need to feed the training data through previous layers, in order to get the input representation to the layer currently being trained. In Algorithm 3, we simply assume that the entire data fits in memory (or on disk), and that no data augmentation is used. In practice, it is rarely so! Unless dealing with small data and narrow layers, it is mostly not even practical to write the intermediate representations to disk; it easily blows up to terabytes of data. Moreover, when heavy data augmentation is required, you get a unique dataset for each epoch of training. The augmentation is done directly on the input images (or whatever your data is), and can generally not be replicated in latent space. Hence, we use a **feednet** to provide data to the training process. That is, mini-batches are drawn from the raw data, $\mathcal{D}(X, Y)$, augmented as usual, and then fed through the feednet. The output of the feednet now becomes the input to the actual layer that is under training. When fitting the i'th layer, the corresponding feednet, $f_{feed}^{(i)}$, has the architecture, $\Psi_{feed}^{(i)} = (\psi_1, \ldots, \psi_{i-1})$. Note, that even while using a deep feednet, we still see very significant reductions in the memory consumption during training; easily 40-70%. Of course, it depends on the architecture and the batch size being used, but the overhead of training (vs. inference) is significant. For the feednet, we do not need to store layer activations (or their gradients) for backpropagation, or any optimizer-related parameters such as momentum.

Feature checkpointing

When training a very deep network, the feednets can also become very deep. If necessary, checkpointing the latent features at a few *narrow* layers, can be useful in order to reduce the execution time or the device memory consumption. Suppose we pick layer j to be checkpointed. During the *training* of that layer, we store *all* the mini-batches of output activations, Z_j, for that layer in RAM or on disk. Whenever the q'th layer, $q > j$, is training, we then draw the exact mini-batches, Z_j, in *the same order* as they were recorded, and feed them through the cropped feednet with architecture, $\Psi_{feed}^{(q)} = (\psi_{j+1}, \ldots, \psi_{q-i})$.

6.3 Backprop & Related Work

The problem that backpropagation addresses, in the training of multi-layer neural networks, is that of *credit assignment*. Specifically, to each of the hidden units, it assigns a measure of responsibility for the error obtained at the output end of the NN. Backprop represents an "everything is connected" type of view on NN training. In this view, all layers (and weights) must be trained jointly, and absent any information about the upper layers, layer-wise training is not possible; see Section 5.2.3.

Already in 1986, LeCun presented a different idea, called *target propagation* [87, 89]. Simply speaking, the idea is to estimate good *targets* for the hidden layers, and use those for learning the weights. More recently, this idea was picked up by Bengio (2014), who proposed to use autoencoders for credit assignment via target propagation [12]. Bengio argues that this is "biologically more plausible" with respect to how the brain works. The idea was later implemented by Lee et al. (2015) in the form of *difference target propagation* [92]. Although the method proposed by Bengio is not a greedy layer-wise training algorithm, it is conceptually related to our work.

In the 1990s, multiple alternatives to backprop were explored in the literature. One such alternative is to train (shallow) neural nets using decision trees; first fit a decision tree to the given data, and then map the learned tree to a neural network, see e.g. [78, 120, 131]. This method obviously limits the space of functions that can be reached. Yet, it is of some interest to us, because it is also a *locally greedy algorithm for training NNs*.

This line of thinking is exactly captured in the *direct feedback alignment* approach proposed by Lillicrap et al. (2016) [94]. Motivated by neuroscience and "biological plausibility" the authors propose a simple way of assigning errors to the hidden layers. Instead of backpropagating exact error gradients through each layer and its non-linearity, random (linear) projections of the output error delta is used. The random feedback weights, i.e. the projections, are fixed during training and the NN learns to adapt to the incoming error signal. "In essence, the network learns to learn", the authors argue. With this approach, they achieve error rates similar to those of standard backprop on a number of tasks.

Deep networks with stochastic depth [61] is another related technique. Conceptually, it could be doing random layer-wise training if all but one layer is omitted in every gradient update. It significantly differs from DLL in that, for each mini-batch during training, you randomly create networks with *expected smaller depth*. This is achieved by adding stochastic skip-connections between the layers (or blocks). Unlike with DLL, you always keep the entire network in memory, and the full architecture is always used for testing. Additionally, with this approach you are effectively training all the layers in concert, just stochastically, and thus you get more coordination between the layers than with bottom-up DLL. This method is obviously substantially differently from ours.

One approach to decoupling the training of the layers is to use synthetic gradients, as proposed by Jaderberg et al. (2017) [75]. That is, at each individual layer an auxiliary model is attached which learns to predict the gradients coming from the layers above. These models also need to be trained, but they can wait for the true gradients to arrive, while the actual layer being trained can proceed (without *locking*). This approach thus consumes more memory and compute than DLL (and even than full-model backprop). To be clear, this is not layer-wise learning but the method uses some layer-local error signals.

An honorable mention must go to Geoff Hinton's proposed a layer-wise training scheme, namely the "Forward-Forward Algorithm" [53]. This method replaces the backward pass of the backpropagation algorithm with a second forward pass. Similar to our own method, it uses a layer-local training objective and eliminates the need to save intermediate activations for all the layers in the network during training; making it more memory efficient. Furthermore, Hinton argues that this layer-wise scheme is also more "biologically plausible". This approach differs significantly from our own, and to our knowledge does not match the state of the art for end-to-end backprop + SGD.

6.3.1 "Local Learning" Literature

Our work falls into the category of *local learning*. The deep learning "era" was indeed, in part, set off by a related method, namely greedy layer-wise *unsupervised* pre-training [13, 54, 122] (followed by a full-model *supervised* stage). Using layer-local error signals has been proposed by multiple authors. In 2015, two papers independently proposed to insert intermediate losses at hidden layers of deep architectures, which are then combined with the global loss. In the 22-layer GoogLeNet, Szegedy et al. [139] added two *auxiliary classifiers* to aid the vanishing error signal backpropagated from the global loss during training. The "deeply supervised nets" of Lee et al. [92] adds intermediate classifiers at every hidden layer—thus, partially resembling a greedy layer-wise training scheme.

To our knowledge, the first work that strictly uses local error signals is that of Nøkland et al. (2019) [117]. While their motivation is both biological plausibility *and* layer-wise training—it is **not actually layer-wise training;** they just use *layer-local losses*. That is, all the layers are still trained in concert, but each hidden layer has **two decision models attached**, one for the cross-entropy loss and another for the *similarity matching loss*. This means, that the layers are still able to adapt to each other, from the bottom up, as the incoming representations change during training. In Section 6.6 we argue that this has a regularizing effect on the training; we refer to it as *forward regularization*. **This is a distinct advantage over our proposed method of strictly bottom-up, and completely decoupled, layer-wise learning. Their method also consumes significantly more memory than ours.** Note, that the authors do not include any results on modern architectures or for heavily augmented data.

Related to this method is the *local unsupervised contrastive learning* proposed by Xiong et al. (2020) [153]. These authors also perform the forward pass in the usual end-to-end way, but then use layer-local contrastive losses for unsupervised training. Thus, this method too benefits from *forward regularization*. Notably, equivalently to our own discovery that a sliding window approach significantly improves the results (see Section 6.5), the authors propose to train two residual blocks at a time (with one overlapping block between each stage). As this is an unsupervised method it is not directly comparable to ours. The authors do not include results on modern architectures.

To our understanding, and as also pointed out in [150], the current consensus in the community is that layer-wise learning suffers from certain challenges, and *generally does not match the performance of full-network backprop!* This is largely based on two papers by Belilovsky et al. (2019 & 2020) [10, 11]. Somewhat surprisingly, despite referring to their work as layer-wise learning, they only report one single *actual* layer-wise result; for AlexNet [83]. **Indeed, all the competitive results reported are for training three layers at a time.** Furthermore, like Nøkland et al. [117], *the authors avoid modern architectures and heavy data augmentation.* Unlike our own work, the authors do not address the issue of overfitting and the lack of implicit interlayer regularization. In [11] they specifically focus on the potential model parallelization and propose a scheme similar to what we present in Section 6.8; i.e. partition-wise learning.

The work by Wang et al. (2021) [150] attempts to overcome the inherent challenge in layer-wise learning. To this end, they hypothesize that too much task-relevant information is discarded in the lower layers; causing problems upstream. They introduce the *Information Propagation (InfoPro) loss* to specifically address the issue. Interestingly, the proposed loss formulation looks very similar (if not equivalent) to the well-known **information bottleneck method** (Tishby (2000)[144]), that was famously applied to deep nets in [133, 143]. *While the authors do cite this work, they do not point out the obvious similarity with their own method.* Similarly to Nøkland et al. (2019) [117], in the actual implementation, they **add two auxiliary models to each layer.** One model for the cross-entropy and another for a *reconstruction loss on a decoder*. The latter is a proxy for the mutual information, $I(Z_i; X)$, and is intended to prevent the aforementioned information loss. **To the best of our knowledge, this work represents the state-of-the-art of published results on local learning.** Again, these authors *do not report a single actual layer-wise learning result.* Indeed, they report unfavorable results on training two layers at a time with ResNet-32, and their approach only matches the results of full-model backprop when they split a network into **two partitions** (what we refer to as *partition-wise* training; see Section 6.8). To be clear, that means that they train 55 layers at a time for ResNet-110, and 16 at a time for ResNet-32. According to our own observations, this is easy to do and does not require any extra auxiliary loss (such a the proposed decoder reconstruction loss); see Section 6.8. Nonetheless, in their head-to-head comparisons they do appear top be doing better than the previous methods for local learning. Furthermore, similar to all authors: **they do not include results on modern architectures or for heavily augmented data.**

The only *actual* single-layer bottom-up layer-wise learning result, that we are aware of, is that of Huang et al. (2017) [60]. The authors show that you *can* in fact train the blocks of a ResNet sequentially, one at a time. To do this, they take a boosting [129] approach to block-wise training, and prove that the training error decays exponentially with the depth of the network. A decision-layer is added to each ResNet-block while training, and all but the last one is discarded afterwards—a process nearly identical to Algorithm 3. It is thus a ResNet-block-wise greedy algorithm, that treats each block like a *weak classifier*, as known from regular boosting theory. The blocks must then satisfy the weak learning condition, i.e. that they must do better than random guessing with respect to the classification task at hand; see e.g. [129, 130]. Unfortunately, and as also pointed out in [150], while Huang et al. provide mathematical proof for their method, they provide only very limited empirical evidence. *We contend, that this method will overfit signifi-*

cantly absent any additional regularization (such as what we propose in Section 6.6.3). To this point, we note that both Wang et al. [150] and Belilovsky et al. [10], reported quite poor results for this method.

State of the Art

In our own experiments, we have observed that matching the results of full-model backprop with DLL is significantly harder when:

1. Using *heavily* augmented data.
2. The layers are *narrow*.
3. Actually training *a single layer at a time*; i.e. it becomes progressively easier the more layers you train at a time, as discussed in Section 6.5.
4. Using modern *block-based architectures* such as EfficientNet or vision transformers (see Sections 6.6.2 & 6.6.3).

Therefore, we believe that it is not a coincidence that, prior to the present work, no authors have reported results on modern architectures. They all strictly report on older architectures like VGG and Wide ResNets—and do not use heavily augmented data. Most likely, they observed the same issues that we did for modern architectures and heavy data augmentation; i.e. when no forward and backward dropout is applied (see Section 6.6.3).

We would like to point out that we are the first to:

1. Match the performance of full-model (end-to-end) backprop with *actual* single-layer decoupled bottom-up training.
2. Report results on heavily augmented data.
3. Report results on modern architectures.

Thus, to the best of our knowledge: *the method presented in this chapter significantly outperforms all previously published results for local learning.*

6.4 First Results

We first tested DLL on two architectures: a vanilla 4-layer CNN, and the well-known VGG-13 architecture [135], using the MNIST, Kuzushiji-MNIST (KMNIST), and FashionMNIST datasets [22, 88, 152]. No data augmentation was used. The VGG architecture has three affine (decision) layers with dropout at the top, so it seemed natural to use those as a decision-block. For the vanilla CNN, we used a single affine decision-layer. The results are shown in Table 6.1.

Architecture	**Dataset**	**Baseline**	**DLL**
Vanilla CNN	MNIST	99.54	**99.58**
	KMNIST	**97.09**	96.72
	FashionMNIST	92.02	**92.20**
VGG-13	MNIST	99.60	**99.65**
	KMNIST	**98.26**	97.94
	FashionMNIST	93.41	**93.45**

Table 6.1: **First DLL Results.** No data augmentation. Batch size 128. Baseline trained 100 epochs. Learning rate and weight-decay were the same for both baseline and DLL; 0.05 and 8e-5, respectively. The learning rate was fixed for all epochs.

Observations and Takeaways

- **Early stopping matters:** for these relatively easy datasets, and when no data augmentation is used. The lower layers will overfit if trained for too long. We trained layers 1-3 for 30, 40, and 50 epochs, respectively. All subsequent layers trained 60 epochs.
- **Not all layers are necessarily needed:** For VGG-13, the peak test accuracies were achieved already in layer 4-7, leaving the remaining layers redundant. Thus, DLL acts as a kind of *online pruning*, insofar that it reduces the number of redundant parameters (i.e. layers) in the network—at least when the architecture has significantly more capacity than needed for a particular task.

The good news is, that it was simple and easy to implement and run the layer-wise training, and it worked well on the MNIST-like datasets. The bad news is, that on harder benchmarks, such as CIFAR-10 [82], this single-layer bottom-up training scheme induces overfitting. This is caused by a reduction in the amount of *implicit interlayer regularization* (see Section 6.6). One way to address this, is by training *segments of layers* at a time.

6.5 Deep Segment-wise Learning

While our main focus is on layer-wise learning, deep segment-wise learning (DSL) deserves to be mentioned. For this, we use a sliding window approach. That is, instead of a single layer, we train segments of k consecutive layers at a time. Segments may or may not overlap, depending on the window step size. Figure 6.2 shows the effect of increasing the segment (or *subnet*) size: *higher and higher test accuracies are achieved.* Thus, this figure illustrates the effect of *implicit interlayer regularization*; namely, that **depth regularizes.**

This behavior has been consistently observed in our work; over a wide range of architectures and datasets. For practical purposes, the take-home message is clear: *the more layers you train at a time, the easier it gets, so maximize the subnet size to match the available memory.* This observation is consistent with the result reported by other authors; please see the related work section.

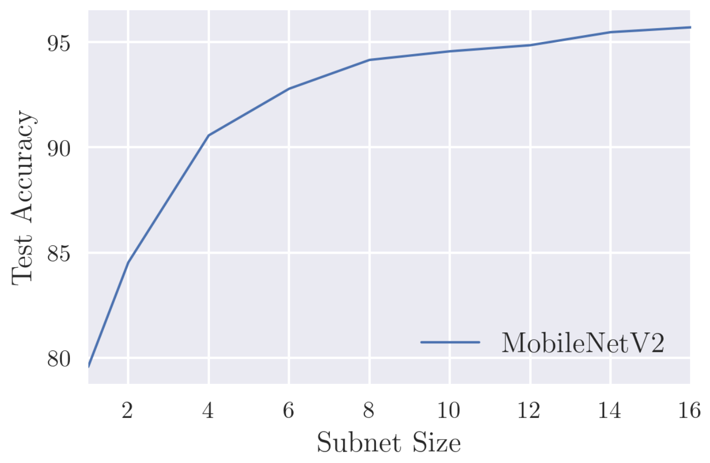


Figure 6.2: **Effect of Increasing Subnet Size.** Training the MobileNetV2 architecture on the CIFAR10 dataset for 50 epochs per subnet (transfer learning); without any dropout or other additional regularization. For a subnet of size k, the overlap with previous layers is $k - 1$. Bigger-sized subnets have more implicit interlayer regularization and thus achieve much better test accuracy.

6.6 Interlayer Regularization

Implicit regularization (AKA implicit *bias*) has been widely studied in the literature. It is an inherent property of gradient descent [98], and stochastic gradient descent (SGD) [18, 136]. Several applications have been studied such as matrix factorization [40], SGD with deep nets [4, 8, 29, 84, 102, 115, 116, 142], dropout [108, 156], and data augmentation [95]. Importantly, several authors have discussed how *depth* regularizes by **inducing low-rank solutions** [4, 37, 46, 69, 74, 142].

6.6.1 How Layers Regularize

Interlayer regularization, or the implicit bias of depth, is a rich topic that cannot be fully covered in the present work. We will briefly present two conjectures that motivate our method. A few observations from the literature are relevant, namely that **lower frequencies of variation in the data are learned first**, and high frequencies are learned later in the training process [9, 17, 121, 154], and that DNNs **converge bottom-up;** i.e. lower layers converge first, and upper layers last [20]. A logical inference from these two observations is that **lower layers are more likely to capture lower frequencies of variation in the data**, while upper layers capture higher frequencies.

In our opinion, two mechanisms, that have not previously been explicitly mentioned in the literature, contribute to the regularizing effect of depth: *forward* and *backward* regularization. The underlying assumption is, that *inconsistency* (or *noise*) regularizes. This should not be controversial, as it is well established in the literature. Notably, the motivation behind *dropout* is to prevent “the co-adaptation of feature detectors” by adding noise [56]. We prefer the term *inconsistency* over noise, in this context, because it refers to the consistency of weight updates. If they are inconsistent (e.g. the signs are constantly changing), then the weights are less likely to change a lot over time; they will mostly wobble back and forth.

Conjecture 1 (Forward Regularization)**.** *The input distribution to all but the first layer is **inconsistent** (constantly changing) as long as the layers below it are still learning. This has a regularizing effect, in that only the most **consistent**, least varying, parts of the input can be learned. This effect is exaggerated with increasing depth; i.e. the upmost layer receives the most inconsistent input distribution over time.*

Conjecture 2 (Backward Regularization)**.** *The gradient distribution backpropagated from the top layer becomes increasingly **inconsistent** with each layer. Thus, the first/lowest layer receives the most noisy, most varying, gradient signal during training. This has a regularizing effect, in that the most **consistent**, least varying, parts of the gradient signal will affect the weights the most over time. This effect is exaggerated with increasing depth.*

It is our contention, that these conjectures are consistent with and, at least in part, account for the aforementioned observations, and that lower layers capture lower frequencies. The frequencies of variation stem from the data, where regions of *low complexity*, that e.g. are homogenous with respect to the class label, are associated with low frequencies (and vice versa). Thus, lower layers mostly capture the least complex regions in the data, whereas upper layers capture the more complex (heterogeneous) regions.

6.6.2 One Solution: Random Robin

With DLL training, we can simulate forward regularization by adapting a *random robin* training scheme. That is, instead of training the layers strictly sequentially from the bottom up, we split the training into multiple *rounds*. For each round, we randomize the layer ordering and train only a few epochs at a time. **This way, upper layers still get the regularizing benefit of inconsistent and changing input distributions!** This scheme requires us to keep track of the state of each of the training processes (one for each of the layers), and thus takes up extra memory for optimizers etc. The pseudocode is listed in Algorithm 4 (assume $IsRandom = True$).

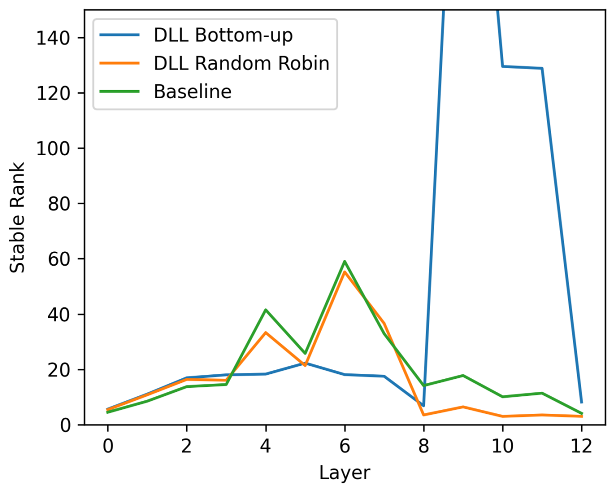


Figure 6.3: **Rank & Random Robin.** Stable rank of the weights in each layer of VGG-13 models trained in three different ways: bottom-up (DLL), random robin (DLL), and full-model backprop (baseline). Random robin is clearly much closer to the baseline.

Algorithm 4 Round Robin Training

```
1:  Let $S$ be a list of training states (one per layer)
2:  Let *IsRandom* be a boolean variable
3:  Let $N_{epochs}$ be the no. of epochs to train per round
4:  Let $N_{rounds}$ be the no. of robin rounds
5:  Let $I = [1, 2, \ldots, k]$ be the layer ordering
6:  **for** $round = 1 \ldots N_{rounds}$ **do**
7:      **if** *IsRandom* **then**
8:          **RandomlyPermute**($I$)
9:      **end if**
10:     **for** $i$ in $I$ **do**
11:         **for** $epoch = 1 \ldots N_{epochs}$ **do**
12:             **TrainOneEpoch**($S[i]$)
13:         **end for**
14:     **end for**
15: **end for**
```

We tested this scheme on the VGG architecture [135], and saw a very clear regularizing effect. In fact, we were unable to match the results of full backprop with strict bottom-up training, but with the random robin scheme we could. Another positive indication in favor of the method is depicted in Figure 6.3: *the stable rank of the weights matches the baseline markedly better than with bottom-up training.* However, on more modern architectures, such as EfficientNet v1 & v2 [140, 141], random robin did not suffice.

6.6.3 A Better Solution: Forward & Backward Dropout

A typical modern DNN architecture starts with a single stem convolutional layer, followed by a number of repeated building blocks, and a classifier/decision head. Some common building blocks are: residual, inverted residual, and transformer blocks. These blocks are carefully designed, and have a regularizing effect that we cannot match with forward regularization (random robin) alone. Therefore, we take another approach to regularizing DLL training.

For training a single block (or layer) from the original architecture, we construct a temporary subnet by adding a *decision-block* atop the block in question. We now simulate forward regularization by applying dropout to the input. To capture how the input distributions will stabilize during training as the (lower) layers converge, the dropout probability decays exponentially in the epoch number. Additionally, we also simulate backward regularization, by applying dropout only to the *gradients* being backpropagated from the decision-block to the layer/block being trained. This dropout probability is fixed for all epochs. This is closely related to the idea of adding Gaussian noise to the gradients, as in [113]. By only applying dropout to the gradients, we assure that the layer does not overfit to high frequencies, while allowing the decision-block to do exactly that. Overfitting in the decision-block is perfectly fine, as we discard it after training. Also, we want the decision-block to be able to absorb a good amount of the loss, so to speak. If we over-regularize it, the learning will get stuck too early. Figure 6.4 shows a DLL subnet with forward and backward regularizers.

6.7 DLL Experiments

*The point of these experiments is **not** to claim that DLL achieves higher test accuracies.* Rather, we merely wish to show that layer-wise training is a reasonable option for memory-limited systems. As can be seen in Table 6.2, for all but one single experiment (out of 44), DLL matches the result achieved by the standard full-model backpropagation to within 0.3 percentage point. The DLL training was strictly bottom-up (each layer was fully trained before proceeding to the next layer). We used several well-known convolutional and transformer architectures [99, 106, 128, 135, 140, 141], and four standard benchmarks for image classification [23, 82, 114].

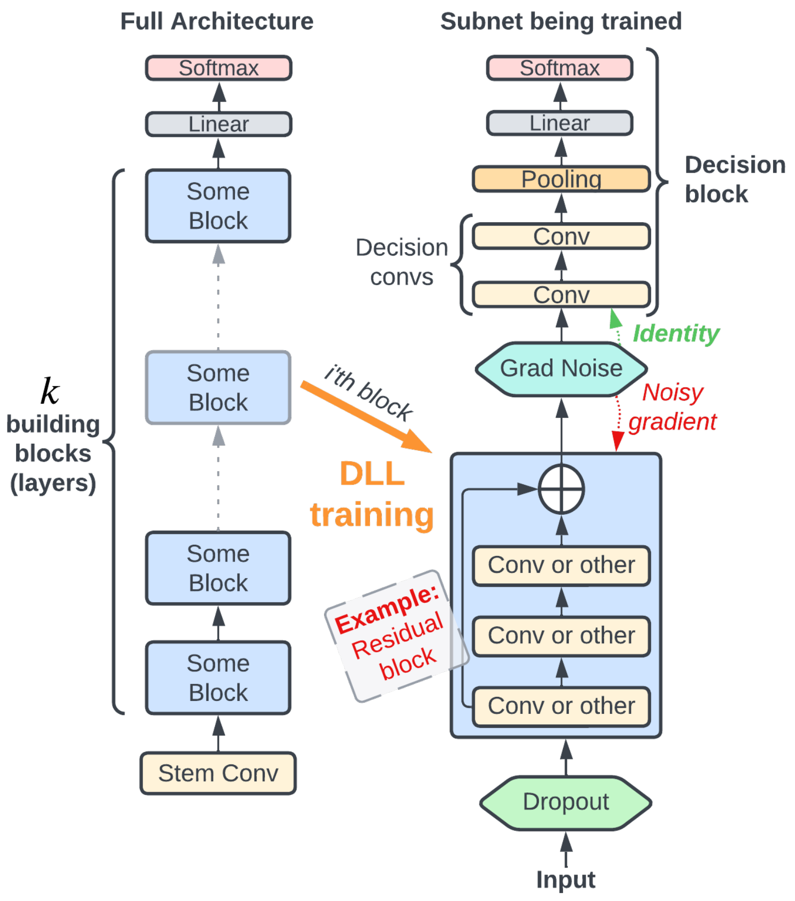


Figure 6.4: **DLL Training. Left:** a typical modern DNN architecture starts with a single stem convolutional layer, followed by a number of repeated building blocks, and a classifier/decision head. Some common building blocks are: residual, inverted residual, and transformer blocks. **Right:** when training a single block (or layer) from the original architecture, we construct *a temporary* ***subnet*** by adding a *decision-block* atop the block in question. To simulate interlayer regularization, dropout is applied to the input, as well as to the *gradients* being backpropagated from the decision-block to the layer/block being trained. Note, that the no. of decision convs, and the number of channels that they have, is adapted to match the available memory (and other constraints).

Architecture	CIFAR10 Basel.	CIFAR10 DLL	CIFAR100 Basel.	CIFAR100 DLL	SVHN Basel.	SVHN DLL	STL10 Basel.	STL10 DLL
VGG-13	**94.40**	94.38 (30%)	72.95	**73.85** (35%)	97.47	**97.62** (30%)	85.33	**86.06** (40%)
ShuffleNetV2×0.5	91.70	**91.71** (23%)	**69.78**	66.25 (30%)	**97.29**	97.07 (30%)	78.58	**83.66** (30%)
ShuffleNetV2×1.0	**94.20**	94.01 (30%)	**73.65**	73.55 (35%)	**97.68**	97.48 (30%)	85.51	**86.30** (30%)
ShuffleNetV2×2.0	**95.49**	95.46 (30%)	**77.07**	76.90 (30%)	**97.89**	97.60 (30%)	80.92	**88.65** (22%)
MobileNetV2	**95.56**	95.18 (30%)	**76.22**	75.99 (35%)	97.30	**97.56** (30%)	80.51	**80.76** (30%)
EfficientNet-B0	93.99	**94.30** (30%)	67.20	**73.25** (30%)	97.40	**97.73** (30%)	78.35	**83.59** (30%)
EfficientNet-B1	**94.76**	94.67 (30%)	63.46	**72.65** (30%)	97.48	**97.66** (30%)	76.08	**84.08** (30%)
EfficientNet-B2	93.85	**94.60** (18%)	68.18	**73.16** (30%)	97.61	**97.63** (30%)	74.29	**83.12** (30%)
EfficientNet-B3	95.26	**95.32** (30%)	64.94	**74.12** (30%)	**97.68**	97.63 (30%)	78.22	**84.54** (30%)
EfficientNetV2-S	**95.96**	95.69 (30%)	68.90	**71.68** (30%)	**97.85**	97.54 (30%)	72.65	**81.08** (35%)
MobileViT-v2	93.88	**94.63** (30%)	72.01	**72.53** (30%)	**97.31**	97.07 (30%)	77.72	**85.28** (30%)

Table 6.2: **DLL Results.** DLL matches full-model backpropagation to within 0.3 percentage-point (in all but one case); with much lower memory consumption (shown in parentheses). Further details of the DLL training configurations are listed in Table 6.3).

For each subnet trained, we adjusted the number of channels in, and the number of, conv-layers in the decision-block to match the **memory restriction of $30 \pm 0.5\%$** (relative to full-model backprop). This restriction held for all but 6 of 44 experiments, where we allowed 35-40% relative memory use, while in two instances approximately 20% sufficed. **For most experiments, the test accuracy peaked after training roughly half of the layers; leaving the following layers redundant.** We generally achieved better results when *the learning rate decreased exponentially in the layer index, and conversely for the weight decay.* To match the baseline results, each model was trained with the same batch size (128), and the same number of epochs; 300 **per layer.** *If the test accuracy was not within 0.5 percentage point of the baseline result, we increased the number of epochs.* This was required in 10 of 44 runs. In four cases, we had to increase the subnet size to 2; effectively making it DSL with a segment size of 2 and an overlap of 1 layer.

All models were trained using a **batch size of 128**, and the **ADAM-W** optimizer [97], as well as with **Cutout + AutoAugment** data augmentation. The baselines were trained for **300 epochs** with learning rates and weight decay found via hyperparameter search (using Optuna [3] with default settings). We used a multi-step learning rate schedule, multiplying the learning rate by 0.1 after 50% and 75% of the epochs were completed. For VGG-13, a single layer is an actual conv-layer, for all other architectures it is a residual block (or similar building block). See Table 6.3 for more details.

Architecture	Dataset	Subnet size	Epochs	Learn. rate	Weight dec.	Grad noise	Dropout	Max. DC	Peak layer	Relative mem. use %
VGG-13	CIFAR10	1	300	3e-03	1e-06	0.6	0.8	4	5 of 10	30.0
VGG-13	CIFAR100	1	300	3e-03	1e-06	0.6	0.8	4	5 of 10	35.1
VGG-13	SVHN	1	300	3e-03	1e-06	0.6	0.8	4	5 of 10	30.2
VGG-13	STL10	1	300	3e-03	1e-06	0.6	0.8	4	4 of 10	40.3
ShuffleNetV2×0.5	CIFAR10	1	450	1e-02	8e-06	0.3	0.8	3	8 of 16	23.3
ShuffleNetV2×0.5	CIFAR100	1	600	3e-03	1e-06	0.6	0.8	4	6 of 16	30.3
ShuffleNetV2×0.5	SVHN	1	300	3e-03	1e-06	0.6	0.8	4	10 of 16	30.2
ShuffleNetV2×0.5	STL10	1	300	9e-04	3e-06	0.6	0.8	4	8 of 16	30.0
ShuffleNetV2×1.0	CIFAR10	1	300	8e-03	8e-06	0.3	0.8	4	6 of 16	30.1
ShuffleNetV2×1.0	CIFAR100	2	300	3e-03	1e-06	0.6	0.8	4	10 of 16	35.1
ShuffleNetV2×1.0	SVHN	1	600	4e-03	1e-06	0.6	0.8	4	7 of 16	30.1
ShuffleNetV2×1.0	STL10	1	300	4e-03	1e-06	0.6	0.8	4	7 of 16	30.0
ShuffleNetV2×2.0	CIFAR10	2	300	3e-03	1e-06	0.6	0.8	4	6 of 16	30.0
ShuffleNetV2×2.0	CIFAR100	2	300	3e-03	1e-06	0.6	0.8	4	11 of 16	30.1
ShuffleNetV2×2.0	SVHN	1	450	3e-03	1e-06	0.6	0.8	4	9 of 16	30.0
ShuffleNetV2×2.0	STL10	1	300	2e-03	2e-06	0.6	0.8	4	6 of 16	21.9
MobileNetV2	CIFAR10	1	450	3e-03	1e-06	0.6	0.8	4	5 of 17	30.0
MobileNetV2	CIFAR100	2	300	3e-03	1e-06	0.6	0.8	4	13 of 17	35.1
MobileNetV2	SVHN	1	300	3e-03	1e-06	0.6	0.8	4	9 of 17	30.0
MobileNetV2	STL10	1	300	2e-03	2e-06	0.6	0.8	4	14 of 17	29.9
EfficientNet-B0	CIFAR10	1	450	6e-03	1e-06	0.3	0.5	4	5 of 18	30.0
EfficientNet-B0	CIFAR100	1	300	2e-03	1e-06	0.3	0.8	4	4 of 18	30.0
EfficientNet-B0	SVHN	1	300	3e-03	1e-06	0.6	0.8	4	5 of 18	30.0
EfficientNet-B0	STL10	1	300	3e-03	1e-06	0.6	0.8	4	3 of 18	30.0
EfficientNet-B1	CIFAR10	1	400	5e-03	2e-06	0.3	0.8	4	7 of 25	30.0
EfficientNet-B1	CIFAR100	1	300	1e-03	1e-06	0.3	0.8	4	4 of 25	30.0
EfficientNet-B1	SVHN	1	300	3e-03	1e-06	0.6	0.8	4	7 of 25	30.5
EfficientNet-B1	STL10	1	300	3e-03	1e-06	0.6	0.8	4	5 of 25	30.4
EfficientNet-B2	CIFAR10	1	450	8e-03	8e-06	0.3	0.8	3	7 of 25	18.4
EfficientNet-B2	CIFAR100	1	300	3e-03	1e-06	0.6	0.8	4	4 of 25	30.4
EfficientNet-B2	SVHN	1	300	3e-03	1e-06	0.6	0.8	4	6 of 25	30.0
EfficientNet-B2	STL10	1	300	8e-04	2e-06	0.6	0.8	4	5 of 25	30.1
EfficientNet-B3	CIFAR10	1	400	5e-03	8e-06	0.3	0.8	4	4 of 28	30.0
EfficientNet-B3	CIFAR100	1	300	3e-03	1e-06	0.6	0.8	4	4 of 28	30.3
EfficientNet-B3	SVHN	1	300	3e-03	1e-06	0.6	0.8	4	6 of 28	30.0
EfficientNet-B3	STL10	1	300	3e-03	1e-06	0.6	0.8	4	5 of 28	30.2
EfficientNetV2-S	CIFAR10	1	600	4e-03	8e-06	0.3	0.8	4	5 of 33	30.1
EfficientNetV2-S	CIFAR100	1	300	5e-03	1e-06	0.6	0.8	4	16 of 33	30.0
EfficientNetV2-S	SVHN	1	300	5e-03	1e-06	0.6	0.8	4	4 of 33	30.0
EfficientNetV2-S	STL10	1	300	5e-03	1e-06	0.6	0.8	4	2 of 33	35.0
MobileViT-V2	CIFAR10	1	300	3e-03	1e-05	0.5	0.8	3	5 of 10	30.0
MobileViT-V2	CIFAR100	1	300	3e-03	1e-05	0.5	0.8	3	6 of 10	30.0
MobileViT-V2	SVHN	1	150	3e-03	1e-05	0.5	0.8	3	5 of 10	30.0
MobileViT-V2	STL10	1	300	3e-03	1e-05	0.5	0.8	3	4 of 10	30.0

Table 6.3: **DLL Training Details.** The **subnet size** is the number of layers trained at a time. The learning rate and weight decay are the *base* values used at the first layer ($i = 0$). At the i'th layer the **learning rate** is $base \cdot 0.8^i$, and the **weight decay** is $base \cdot 1.15^i$. **Grad noise** is the dropout probability on the *gradients* backpropagated from the decision-block to the layer/block being trained. **Dropout** refers to the *input* dropout rate at the first epoch (of training any individual layer); decays by 0.95 after each epoch. **Max. DC** is the maximum no. of *decision conv-layers* allowed when searching for a suitable decision-block configuration (that meets the memory restriction of $30 \pm 0.5\%$). **Peak layer** is the layer index (counting from 1), where the highest *test accuracy* was achieved. **Relative memory use** is the maximal percentage of memory consumed (including the feednet) during the training of any layer—relative to training the whole architecture with the same batch size (i.e. the baseline).

Architecture	**Dataset**	**Epochs**	**Learn. rate**	**Weight dec.**	**Grad noise**	**Dropout**
EfficientNetV2-M	CIFAR10	$5 \cdot 1.08^i$	$3e{-}5 \cdot 1.08^i$	$5e{-}4 \cdot 0.99^i$	0.0	0.0
EfficientNetV2-M	CIFAR100	$2 \cdot 1.08^i$	$5e{-}6 \cdot 1.08^i$	$5e{-}4 \cdot 0.99^i$	0.0	0.0
EfficientNetV2-M	STL10	$5 \cdot 1.08^i$	$3e{-}5 \cdot 1.08^i$	$5e{-}4 \cdot 0.99^i$	0.0	0.0
EfficientNetV2-M	Flowers	$5 \cdot 1.08^i$	$3e{-}5 \cdot 1.08^i$	$5e{-}4 \cdot 0.99^i$	0.0	0.0
EfficientNetV2-M	Pets	$5 \cdot 1.08^i$	$1e{-}5 \cdot 1.08^i$	$5e{-}4 \cdot 0.99^i$	0.0	0.0
MobileViT-V2	CIFAR10	$15 \cdot 1.20^i$	$8e{-}4 \cdot 1.2^i$	$5e{-}4$	0.3	0.3
MobileViT-V2	CIFAR100	$15 \cdot 1.20^i$	$2e{-}4 \cdot 1.2^i$	$5e{-}4$	0.3	0.3
MobileViT-V2	STL10	$18 \cdot 1.20^i$	$2e{-}4 \cdot 1.2^i$	$5e{-}4$	0.3	0.3
MobileViT-V2	Flowers	$18 \cdot 1.20^i$	$8e{-}4 \cdot 1.2^i$	$5e{-}4$	0.3	0.3
MobileViT-V2	Pets	$18 \cdot 1.20^i$	$2e{-}4 \cdot 1.2^i$	$5e{-}4$	0.3	0.3

Table 6.5: **DLL Transfer Learning Details.** All models were trained using a **batch size of 128**, and the **ADAM-W** optimizer [97]. The baselines were trained for **300 epochs** with learning rates and weight decay found via hyperparameter search (using Optuna [3] with default settings). The layer index is i (counting from zero). To prevent overfitting, we run just 2 to 18 epochs in the first layer, and the learning rates are low and were fixed for all epochs (no schedule applied). *All models were trained with Cutout + AutoAugment data augmentation.*

6.7.1 Transfer Learning

While transfer learning also works well with DLL, it is somewhat more brittle. The obvious reason is that the intricate structure learned between the layers (the feature hierarchy) during pretraining is much easier to maintain and utilize when fine-tuning all the layers in concert. With layer-wise training, it is easy to overfit and break something—especially in the lower layers. Therefore, we use low learning rates and run only very few epochs in the lowest layers (2 to 18), and progressively increase the number of epochs for fine-tuning as we move up through the network. We do the same with the learning rate, i.e. it *grows* exponentially in the layer index (contrary to when training from scratch, where it *decays*). In all experiments, the test accuracy peaked in one of the two top layers. Thus, with transfer learning, we can now take advantage of the full depth of the models. See more details in Table 6.5).

	EfficientNetV2-M		**MobileViTV2-100**	
Dataset	Basel.	DLL	Basel.	DLL
Pets	**89.56**	89.40	90.52	**90.71**
Flowers	**97.79**	97.07	**95.79**	95.51
STL10	**92.85**	92.15	94.05	**94.30**
CIFAR10	**97.34**	95.65	**96.00**	95.87
CIFAR100	**84.23**	82.06	**82.53**	81.09

Table 6.4: **DLL Transfer Learning Results.** The details of the DLL training configurations are listed in Table 6.5.

6.8 Deep Partition-Wise Learning

Our results and observations indicate that training large partitions of a deep model separately should be both possible and fairly straightforward. To confirm this, we tested deep partition-wise learning (DPL) on the EfficientNetV2 architecture [141].

6.8.1 CINIC10+

The CINIC10 dataset [26] has train, test, and validation sets of 90,000 images each. We combine the train and validation sets, such that there are 180,000 images for training and 90,000 for testing. We will refer to it as CINIC10+, and use it for DPL with EfficientNetV2-S. We split the model into four partitions of 10 layers each, and trained on a single GPU. No dropout, forward or backward (AKA gradient noise), was used. The baseline trained for 300 epochs with a batch size of 512, and the learning rate and weight decay were found by hyperparameter tuning (using Optuna with default settings). For DPL we used *exactly the same settings*. However, at the i'th partition (counting from zero), the learning rate and weight decay were multiplied by 0.5^i. The baseline took **5h31m** to train and got a test accuracy of **90.97%**. Bottom-up training with DPL (using a feednet) took **8h50m** and the test accuracy was **90.70%**.

6.8.2 Data Filtering

However, we can speed up DPL by utilizing the fact that the data becomes increasingly redundant in the middle to upper layers with training (see Section 5.4). Thus, by applying online k-filtering and switching to a round-robin training scheme (see Algorithm 4), 10 epochs per round, with feature-checkpointing in device memory (see Section 6.2.2), the training completes in just **4h30m** and achieves **90.93%** test accuracy. *It appears, that the data filtering has a slight regularizing effect.* We paused the k-filtering for the first 30 epochs, and then applied it to batch sizes of 6148 with $k = 10$ (capping the filtering at 60% of the data). Figure 6.5 shows the significant effect of k-filtering on the execution time. Note that part of the overall drop in execution time is due to feature-checkpointing in device memory (avoiding the need for a feednet).

6.8.3 Optimal Model Parallelism

Model parallelism is highly attractive, but hard to achieve in practice. As shown in Figure 6.6, the naive approach leads to severe under-utilization due to the sequential dependencies of the forward and backward passes. This can be somewhat alleviated by using micro-batches [64]. At least in theory, DPL should allow for near-optimal model parallelism, as it only requires pushing the output from each forward pass to the next device/model-partition; via a shared disk or interconnect. In practice, it would also require that the partitioning is done such that a training pass takes approximately the same amount of time on all devices. As illustrated, in Figure 6.7, this guarantees that no device will ever be idle while waiting for data (except for at the very beginning).

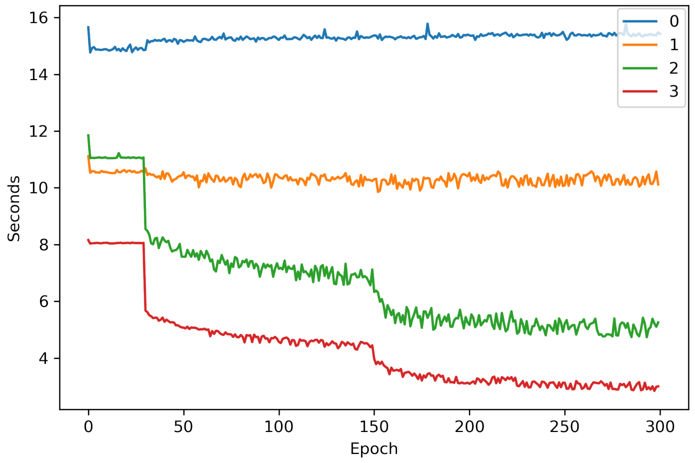


Figure 6.5: **Effect of k-Filtering.** The partition-wise training time per epoch. After 30 epochs, the filtering kicks in, causing a significant drop for partitions 2 & 3, while partition 0 sees a small increase due to the added overhead of online k-filtering. After 50% of the epochs, the learning rate is dropped, causing an increase in both accuracy and k-redundancy; from there, partitions 2 & 3 see only $\sim$50% of the data.

6.8.4 ImageNet Results

We tested DPL on the EfficientNetV2 (small, medium, and large) architecture [141] with the ImageNet 2012 dataset [28] and various standard crop sizes used in the literature (224 to 512). We adapted the training script from PyTorch Image Models [151] to accommodate DPL; our code is available here [INSERT ANONYMOUS REPOS!!!].

Execution Time

Our DPL implementation was the simplest possible, as we used an NVME drive as the interconnect. Thus, we start separate processes for each partition; each of them waiting for data to be written by the preceding partition, while writing their own features to disk for the next. For the baselines, we used PyTorch's Distributed Data Parallel (DDP) backend, which also was used when a DPL partition was allocated more than a single GPU. Table 6.6 shows some examples of the speedup achieved with DPL over full backprop with DDP. The biggest and deepest model (EfficientNetV2-Large), benefits the most (up to 279%), while the smallest one gains the least (only 4-8%). These results are inversely correlated with the batch sizes (i.e. the memory consumption of the models). We chose the batch sizes, and the DPL partitioning, to maximize the image throughput during training. The partitions thus varied greatly in size (i.e. the number of layers / building blocks), as the lowest layers consumed significantly more memory and compute.

Our experiments were run on highly heterogeneous hardware (varying CPU, RAM, PCIe, and NVME speeds), with 4-8 consumer-grade GPUs (NVIDIA RTX 3090 & 4090) with 24 GB of device memory. Note, that with our current implementation, sharing data via an NVME drive, we would *not* expect to see significant speedups when running on enterprise-grade GPUs (e.g. NVIDIA A100 or H100). This would require an implementation that utilizes fast interconnects

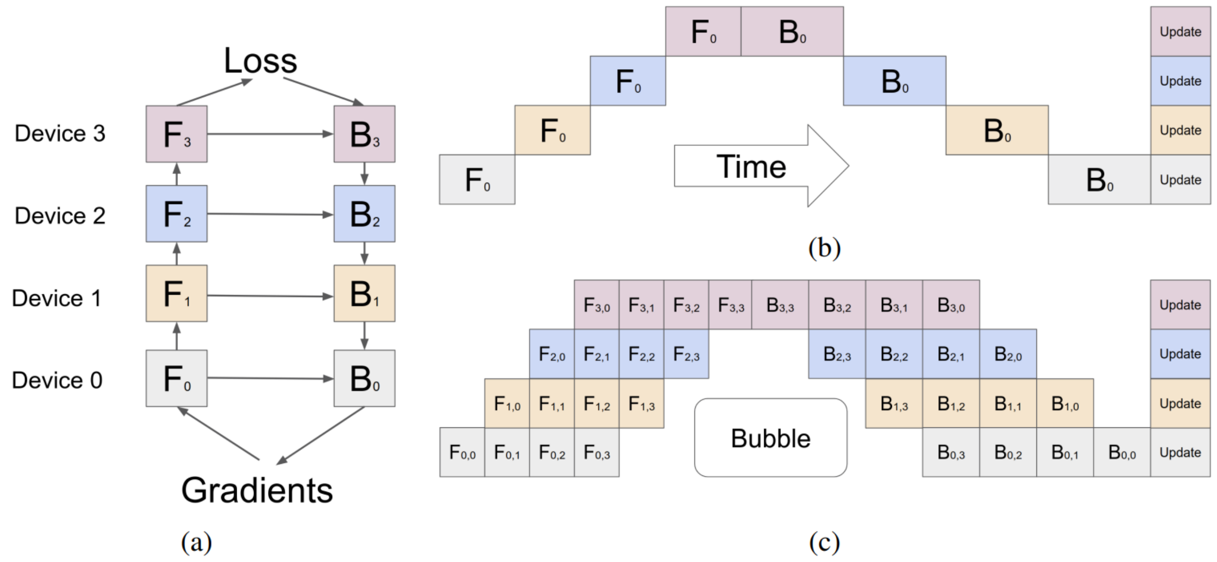

Loss
Device 3
Device 2
Device 1
Device 0
F_3
B_3
F_2
B_2
F_1
B_1
F_0
B_0
Gradients
(a)
Time
Update
(b)
$F_{0,0}$ $F_{0,1}$ $F_{0,2}$ $F_{0,3}$
$F_{3,0}$ $F_{3,1}$ $F_{3,2}$ $F_{3,3}$ $B_{3,3}$ $B_{3,2}$ $B_{3,1}$ $B_{3,0}$
$B_{0,3}$ $B_{0,2}$ $B_{0,1}$ $B_{0,0}$
Bubble
(c)

Figure 6.6: **Naive vs. Pipeline Parallelism. (a)** An example neural network with sequential layers is partitioned across four accelerators. F_k is the composite forward computation function of the k-th cell. B_k is the back-propagation function, which depends on both $B_k + 1$ from the upper layer and F_k. **(b)** The naive model parallelism strategy leads to severe under-utilization due to the sequential dependency of the network. **(c)** Pipeline parallelism divides the input mini-batch into smaller micro-batches, enabling different accelerators to work on different micro-batches simultaneously. Gradients are applied synchronously at the end. *Note: this figure and caption borrowed from the GPipe papers by Huang et al. [64]*

(NVLink), and probably larger models, like GPT-4 [118]. *Therefore, we strongly encourage other researchers (with access to large compute) to test DPL on very large models.*

Test Accuracy

As a sanity check, we completed the training of a few ImageNet models. We used similar settings as in [141], except for a smaller crop size (224 instead of 380 pixels), and lower batch size (160 for the baseline) in order to match the available memory. The learning rate was lowered accordingly (to 0.016). **The baseline had a test accuracy of 81.78%, while our two model-parallel DPL models (two partitions) achieved accuracies of 81.81% and 81.59%.** For the latter, we used **data filtering**, and filtered away 25% of the data fed to the second partition (starting from epoch 31). Instead of k-filtering, which is expensive in high dimensions (due to the need for computing a distance matrix), we used online *loss filtering*. That is, we simply used the loss attached to an example in the first partition as a proxy for its *redundancy*, and omitted those with lowest loss. This is similar to Selective-Backprop [77]; only, we filter away examples in the forward pass.

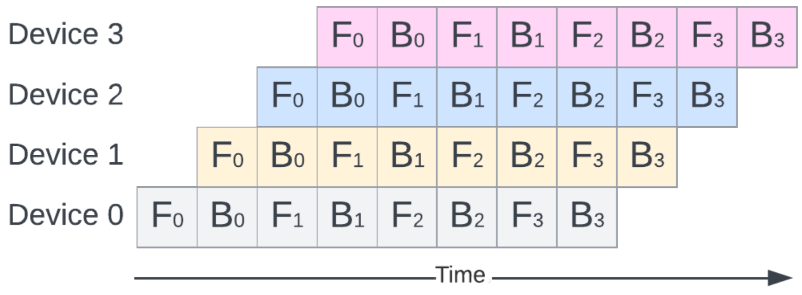


Figure 6.7: **Optimal Model Parallelism.** At least in theory, DPL should allow for near-optimal model parallelism, as it only requires pushing the output from each forward pass to the next device/model-partition; via a shared disk or interconnect. In practice, it would also require that the partitioning is done s.t. a training pass takes approximately the same amount of time on all devices. As illustrated, this guarantees that no device will ever be idle while waiting for data (except for at the very beginning).

6.9 Conclusion & Future Work

The memory-limited training scenario becomes increasingly important as AI & ML become more ubiquitous. We have shown, that layer-wise learning is a viable option for exactly that situation. This, and partition-wise learning, potentially opens up more new ways of distributing deep learning on small and low-cost devices. We hope, that this could be a step toward democratizing the technology, which is currently dominated by a few large and wealthy corporations.

While we have tested our method solely on image classification benchmarks, we have no reason to believe that it would behave substantially differently in other domains. However, highly specialized model architectures and methods may require some additional research and adaptation.

Moreover, other methods exist for reducing the memory footprint of deep net training. However, our method does not compete with those. Rather it stands to gain equally from said methods, such as low-precision training or gradient checkpointing [19]. Thus, we have not included any comparisons with such other methods in this work.

In the short term, we think that DPL could have the most significant impact on the field. For example, large language models are notoriously expensive to train. If DPL can be successfully applied to that domain, it could literally reduce that cost by millions of dollars (and save a lot of energy and emissions). This would be a significant result. **Thus, we humbly repeat our appeal for capable researchers to investigate this.**

Lastly, having successfully applied our two conjectures on interlayer regularization to layer-wise training, we are optimistic that they can be converted to stronger and more rigorous statements.

			DDP		DPL				
Arch.	**Crop**	**GPUs**	**Batch**	**im/s**	**Part.**	**Alloc.**	**Batch**	**im/s**	**Speedup**
Small	224	4x3090	288	2270	2	2-2	2350	2323	4%
	512	4x3090	48	560	2	3-1	80	499	8%
Medium	224	8x4090	160	2952	4	3-2-2-1	512	3752	27%
	224	4x4090	160	1165	2	2-2	320	1624	39%
	256	4x4090	128	900	2	2-2	256	1280	42%
Large	224	4x3090	64	477	4	1-1-1-1	384	741	55%
	384	4x4090	24	87	4	1-1-1-1	128	330	279%
	512	4x3090	16	113	4	1-1-1-1	64	150	33%

Table 6.6: **DPL Performance Examples.** The columns are: **architecture** (EfficientNetV2 variant), **image crop size** in pixels, no. of **GPUs** used (RTX 3090s or 4090s), **DDP:** batch size and image throughput, **DPL:** no. of partitions, GPU allocation per partition, batch size, and image throughput. And finally, the **speedup** achieved with DPL over standard DDP (full backprop).

Chapter 7

Beyond Gradient Descent: Platonic Projections

7.1 Introduction

Since its publication in *Nature* in 1986 [127] the backpropagation algorithm—combined with stochastic gradient descent (SGD)—has been the standard method for training neural networks. Using the chain rule of calculus, it solves the problem of assigning errors to the hidden layers, and thus far no viable alternatives have been found. It is fair to say that without backprop we would not have seen the many recent successes in deep learning. However, while it has been an extremely successful algorithm, it also suffers from severe drawbacks. It typically takes hundreds of passes over the data for the algorithm to converge, which can be very resource-demanding with respect to time, memory, hardware, and energy consumption. Also, backpropagation + SGD relies heavily on hyperparameter tuning and, consequently, expert knowledge. Furthermore, it has proven to be very challenging to build a solid theoretical framework around it, and thus neural networks have long been considered a black box by many (albeit this is changing somewhat). For these reasons, much of the literature is concerned with treating the symptoms rather than the cause, so to speak. Popular methods such as dropout, batch normalization, momentum, ResNets, and LSTMs are thus designed to deal with specific challenges inherent in the backprop + SGD paradigm. Therefore, a quest for an alternative to backprop is warranted. This work is an attempt at finding such an alternative.

Another possibly important reason for pursuing alternatives to backprop, is the much ballyhooed potential of **quantum of computing**. If the prophecies come true, quantum computing will fundamentally change how deep nets are trained and used. Unfortunately, error-backpropagation might not be a suitable algorithm for quantum computation. Recent work by Abbas et al. (NeurIPS 2023) [1], shows how the reliance on reusing intermediate results (layer activations and gradients) makes backprop hard, if not impossible, to use efficiently on quantum computers. The authors state the following:

> *If the difficulty to achieve an efficient scaling is due to inherently quantum properties, perhaps backpropagation is not the correct method for optimization of quantum models, which seems to be a growing belief for classical models too...*

The method we are introducing, in this chapter, does not have the same reliance on intermediate results. It is a strictly bottom-up approach, without any iterations over the data. Additionally, it relies on solving a sequence of linear systems, and there already exists an efficient quantum algorithm for computing matrix inverses: the Harrow–Hassidim–Lloyd (HHL) algorithm [47]. *Potentially, this makes our method a substantially better candidate for training quantum models than backpropagation.*

With our improved understanding of the inner workings of neural networks, gained in the previous chapters, we take a look beyond the confines of gradient descent and error-backpropagation. Thus, we develop a quite elegant, and surprisingly simple, way of training multilayer perceptrons (MLPs). In a nutshell, we will show that MLPs can be fitted to training data by solving a sequence of least squares problems. By following the principle of gradual class separation, we show that computing reasonable targets for the hidden layers is easy. *Specifically, a good target is a representation in which the classes are already separated.*

To be very clear: **we are not making any strong claims about the method presented in this chapter!** *This is very much work in progress.* We present it here for its novelty, its obvious relevance to the topic of this thesis, and because we believe that it does show some promise with respect to finding a viable alternative to the backprop + SGD paradigm.

7.2 Plato & the Allegory of the Data

The proposed method owes its name to the ancient Greek philosopher, Plato (approx. 423–348 BC). Specifically, to the famous **Allegory of the Cave** from his seminal work *Republic* (approx. 375 BC); see e.g. [123]. The idea came by way of a thought experiment, in which we used Plato's allegory as an analogy for what happens inside the layers of a neural network during a supervised learning process.

7.2.1 Allegory of the Cave

In Plato's allegory, some prisoners have spent their entire lives, in chains, sitting with their backs to a wall. Directly in front of them is another wall on which shadows appear. Over the years, they start to recognize the different shapes of the shadows, and assign names to them. Having never experienced anything else, the prisoners believe that what they see on the wall in front of them *is* the actual real world.

Unbeknownst to them, behind the wall that they are chained to, a large flame is burning. Regularly, prison guards walk in front of the flame while holding up different objects. Thus, the shadows that the prisoners regard as their reality, are actually cast by those objects. Their whole world-view is thus controlled and decided by the prisoners in an elaborate scheme to manipulate them. Figure 7.1 shows a typical depiction of the scenario.

Eventually, a prisoner escapes from the cave, and gets to see the actual real world. He quickly realizes, that the world of shadows in which he has lived his entire life, was not real. In a word, it was but a poor *projection* of reality. That prisoner, Plato says, is *a philosopher*.

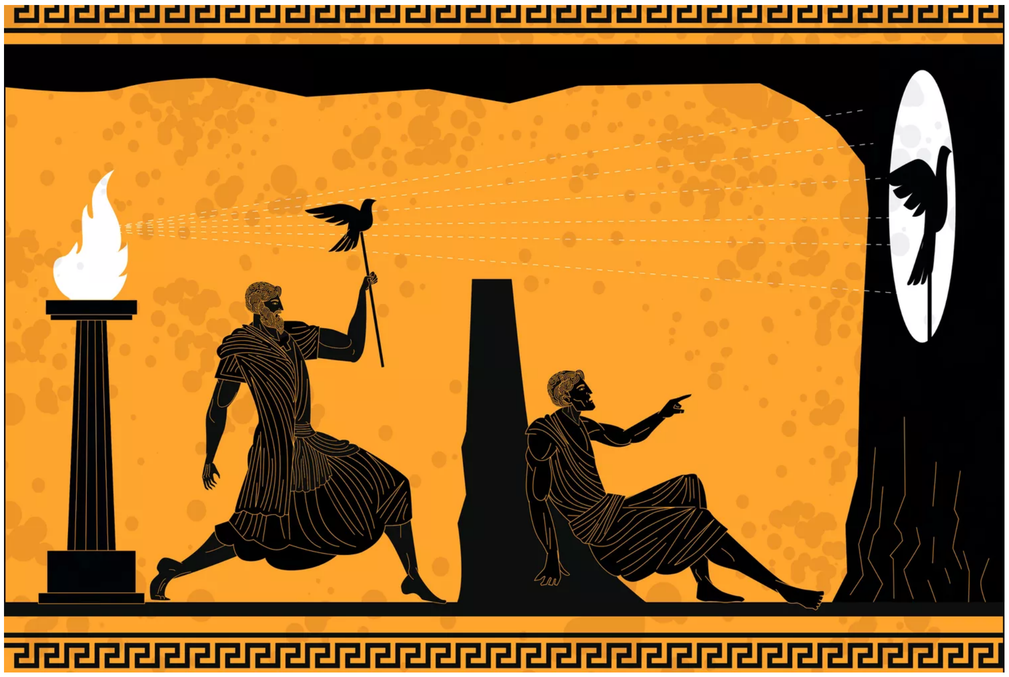


Figure 7.1: **Plato's Cave.** A typical depiction of Plato's famous Allegory of the Cave. The prisoner only ever sees the shadows, not the real objects themselves. He believes that the two-dimensional colorless projections are the actual real world. *Image credit: MatiasEnElMundo / Getty Images.*

7.2.2 Plato's Theory of Forms

The Allegory of the Cave is part of Plato's theory of Forms; often referred to as the theory of Ideas. Plato believed that the physical observed world is, in a sense, not the real world. All the things, objects, patterns, relations, and emotions, that we perceive through our senses are but imperfect images or copies of their true eternal forms. Thus, every single thing, and every single concept, has a real, true, and perfect form that only exists in the eternal and unchanging world of forms (or ideas). Plato refers to this as Eidos, and it is not a place that exists in time or space. **Rather, we can think of Eidos, as the eternal plane in which all universal truth about a concept exists.** As humans, we are destined to perceive only various changing phenomena, that merely *mimic* the true forms; just like the prisoners observing the shadows in the cave.

7.2.3 Allegory of the Data (A Thought Experiment)

As a thought experiment, let us use Plato's cave as an analogy to training deep nets. We, the humans conducting the training, then become the prison guards, and the layers of the DNN become the prisoners. In the supervised learning setting, we know the (ground) truth about all inputs to the network (i.e. their labels)—but we only show the layers random (non-linear) projections of the input examples; namely, $Z = \sigma(XW)$. Like the prisoners in Plato's cave, the layers must now attempt to make sense of what they "see".

If the guards, holding up the objects in front of the flame, represent Eidos, then we must ask:

what exactly does Eidos represent in our machine learning scenario? What is the ML equivalent of the plane in which all truth about a concept exists? Well, all the "truth" that we know, and have access to, is the data itself. So, that plane must somehow contain both the inputs, $X \in \mathbb{R}^{m\times n}$, and the target outputs $Y \in \mathbb{R}^{m\times k}$. However, it cannot just be any random mixture of the two. It must be, that in such an ideal plane, or representation, the concepts captured by the data are clearly encoded. As we have seen, a neural net takes something linear and gradually linearizes it. That is, it gradually separates the classes. Therefore, it makes sense to say that for a neural network, an **ideal representation** of the data is one in which the classes are *linearly separated.* From a philosophical perspective, one could argue that indeed *linearity* is the mathematical equivalent of *understanding.* After all, linear equations, and linear relationships, are the easy ones: it is all the non-linear stuff that usually makes tings hard.

The absolute simplest way of obtaining such an ideal representation is by the concatenation of X and Y. In other words, it is exactly the **augmented matrix** that we all know from Gaussian elimination: $A = \left[\begin{array}{c|c} X & Y \end{array}\right]$. *Any (Platonic) projection of this matrix is essentially the equivalent of the shadows in Plato's cave.* And that is the moral of the story, so to speak. In the following sections, we will show how this simple observation, can be applied to the fitting of MLPs to data. Figure 7.2 shows some examples of what such augmented (Platonic) representations may look like for toy data. As we can readily see, for binary classification problems with $y \in \{-1, +1\}$, the two classes are perfectly separated by the a_1a_2-plane at $a_3 = 0$. Hence, we can exactly recover the targets from the augmented matrix A, by the multiplication $y = A\begin{bmatrix}0 & 0 & 1\end{bmatrix}^T$.

7.3 The Basic Idea

7.3.1 Platonic Projection

As described above, a Platonic (or ideal) representation is one in which the classes are separated. We may simply construct one, in the form of the augmented matrix, $A = \left[\begin{array}{c|c} X & Y \end{array}\right]$. A Platonic *projection* is thus any projection of this augmented matrix—**in which the classes remain separated.** In other words, such that there exists a linear transformation that (sufficiently accurately) recovers Y from said projection:

Definition 9 (Platonic Projection)**.** *For a dataset $\mathcal{D}(X, Y), X \in \mathbb{R}^{m\times n}, Y \in \mathbb{R}^{m\times k}$, and some $P \in \mathbb{R}^{n+k\times d}$, a Platonic Projection is given by $T = \left[\begin{array}{c|c} X & Y \end{array}\right] P$ such that there exists a matrix, $Q \in \mathbb{R}^{d\times k}$, where $\|Y - TQ\|_F < \epsilon$ for some sufficiently small $\epsilon > 0$.*

Note, that in practice, ϵ probably does not need to be very small for many of the layers. Particularly in the early layers of a very deep network, attempting enforce a high degree of class separation is probably unnecessary. When the projection *does not reduce the number of dimensions,* such that $d \geq n + k$, then any dense random matrix, P, will likely guarantee that Q exists. For example, if $d > k + d$, then multiplication by P will take the $(k + d)$-dimensional hyperplane, on which $\left[\begin{array}{c|c} X & Y \end{array}\right]$ lies, and scale, skew, and rotate it in a d-dimensional space. Thus, the classes will remain separated. Only by knocking out one or more dimensions in $\left[\begin{array}{c|c} X & Y \end{array}\right]$,

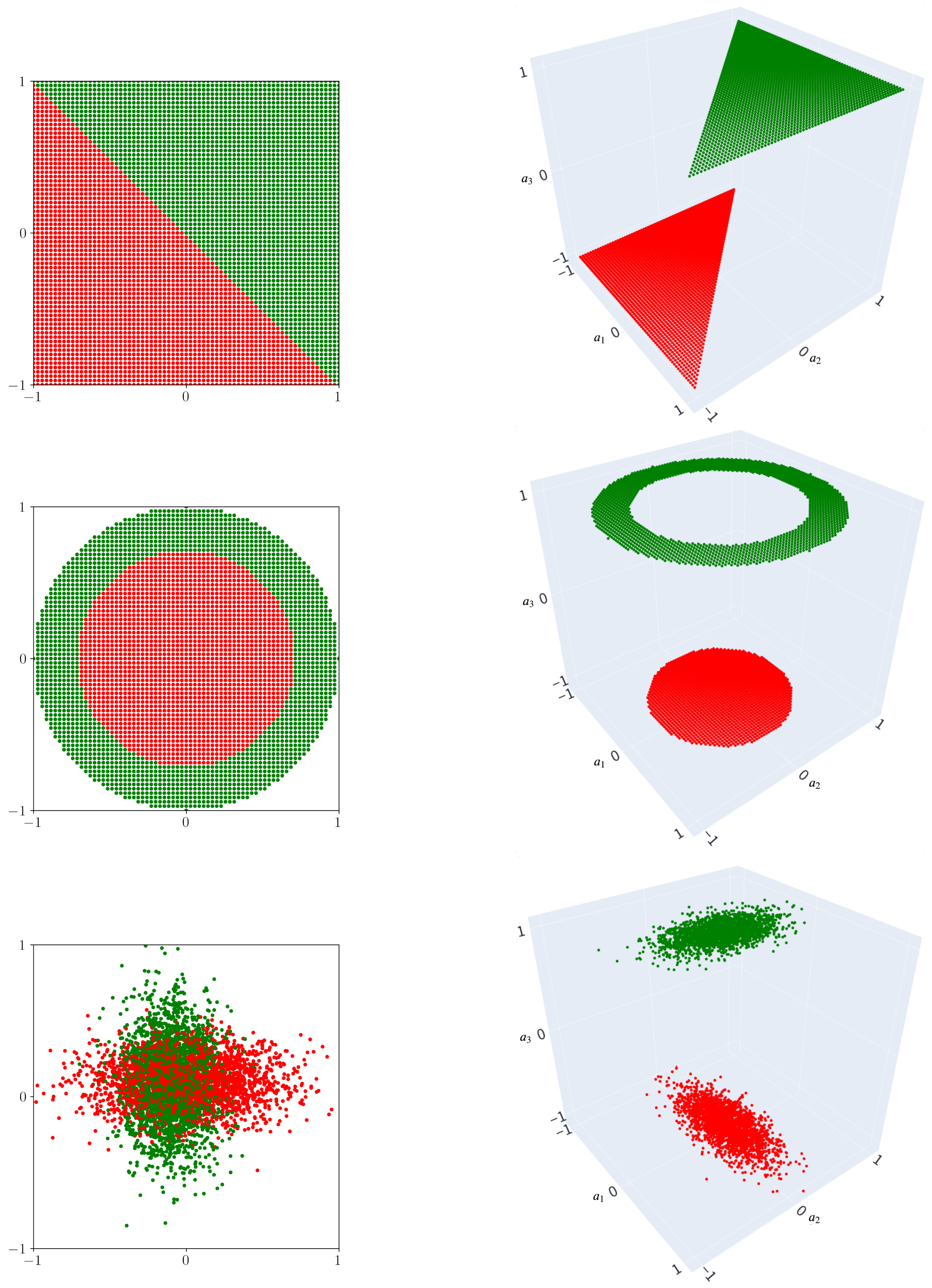


Figure 7.2: **Augmented/Platonic Representations.** Three toy classification problems with labels, $y \in \{-1, +1\}$. **Left:** Input data, $X \in \mathbb{R}^{m \times 2}$. **Right:** Their corresponding augmented (Platonic) representations, $A = \left[\begin{array}{c|c} X & Y \end{array}\right] \in \mathbb{R}^{m \times 3}$. The two classes are perfectly separated by the a_1a_2-plane at $a_3 = 0$.

may we possibly lose information about Y—and this will most likely not happen with a dense random (full-rank) matrix.

Conversely, if the projection is *dimensionality-reducing,* then we need to take extra care when picking P. For example, in the binary classification examples shown in Figure 7.2, we want to maintain the information contained in the third dimension, a_3, of the augmented representations. This requires, that at least one of the column vectors in P is not exactly orthogonal to the a_3-axis. More on the topic of picking P later.

7.3.2 An Algorithm

Earlier, in Chapter 5, we saw how the *least squares distance* (LSD) decreases, while the k-redundancy (Definition 8) increases, in each layer during the training. More importantly, the LSD also decreases in the layer index (i.e. from one layer to the next); and again, conversely for the k-redundancy. This is an expression of the gradual linearization and class separability in the hidden representations. Furthermore, in Chapter 6 we saw how layer-wise training of deep networks is able to match the results of full-model backpropagation with SGD. All in all, these observations strongly indicate that the layer-local training objective is largely the same for all layers. It is exactly these observations that motivate our method.

Notwithstanding that, there is a lot to be said about the implicit regularization stemming from gradient descent, depth, and other factors (see Section 6.6), the layer-local training objective is, of course, *class separation (AKA linearization)*. Thus, for a neural network, $Z_i = \sigma(Z_{i-1}W_i), Z_0 = X$, we argue that:

1. A good layer-local *target* representation is one in which the classes are separated.
2. For the right choice of P_i, a Platonic projection $T_i = \left[\begin{array}{c|c} Z_{i-1} & Y \end{array}\right] P_i$ is therefore a good target.
3. Assuming that σ is monotonic, T_i is a good target in the *preimage* of σ.
4. An MLP, $Z_i = \sigma(Z_{i-1}W_i)$, can be solved in a sequential least squares manner, such that

$$W_i = Z_{i-1}^{+} T_i$$

Please note, that we use the term "solved" loosely, and make no claims about optimality or the generalization properties of the solution. In the sections below, we will study the behavior and performance of the method by testing it on a handful of smaller datasets. We will start by fitting a simple sinusoid.

7.4 Related Work

The literature we referenced for deep layer-wise learning (Chapter 6) also somewhat applies here. Moreover, conceptually, our method also bears some resemblance with the well-known *kernel trick*, insofar that we are purposefully creating a high-dimensional space in which the classes a separated. The kernel trick traces all the way back to Mercer's famous theorem (1909) [107], which was first applied to pattern recognition in 1964 [2]. However, to our knowledge, there is not much directly related work out there. We found only two methods that are closely related: the Extreme Learning Machine introduced by Huang et al. in 2004 [62, 63], and the Pseudoinverse Learning Algorithm for MLPs introduced by Gui & Lyu (also in 2004) [41].

The latter resembles our method a lot, insofar that it is a greedy layer-wise bottom-up algorithm that uses the pseudoinverse. Unlike our method, the authors do *not* introduce a method for estimating targets for the hidden layers—**they do not use any targets at all.** Instead, they simply use the layer-local pseudoinverse as the weight matrix: $W_i = Z_{i-1}^+$. That is an interesting idea, but it seems erroneous to not use the target, Y, in the hidden layers at all. Effectively, this method assumes that for a given set of inputs, there is a single configuration of the hidden-layer weights which is ideal *for any arbitrary labeling*. More importantly, for a dataset with m examples, this method strictly fixes the width of *all hidden layers* to m. Thus, all but the first and last layer will have m^2 weights. Even for the MNIST dataset [90], that would mean 60,000 units in every single layer, and even a shallow 3-layer MLP would have 3.6B parameters. That is not just intractable, it is also completely unnecessary. For this reason, we will not be using this method as a reference in our experiments.

The Extreme Learning Machine of Huang et al. is considerably more tractable. Their basic observation is that *random features* work well—if you have a lot of them. Hence, they fit **single hidden-layer MLPs** by simply using **random weights** in the first layer (to which they fit a linear decision layer). Obviously, this approach does not scale to multiple hidden layers, as the final result would depend almost entirely on the narrowest hidden layer; i.e. **adding more layers does not help**. In Section 7.9.2 we compare this method to our own.

7.5 Fitting a Sinusoid

In Figure 7.3, we listed the code we used to fit a 3-layer MLP (with hidden layers of width 16) to a sinusoid, $y = sin(x)$; i.e. two periods, normalized to the domain $(-1, 1)$, 5000 samples. It clearly illustrates the simplicity of the fundamental idea:

- concatenate the layer-local representation of the data, Z_i, with the target, Y,
- multiply with a random matrix, P_i, to get the target, T_i,
- and solve a linear system of equations (using the pseudoinverse of Z_i).

And that is it. It could hardly be much simpler than that. Figure 7.4 shows the fitted function plotted on top of the training data. They are almost indistinguishable, and the mean squared error is 2e$-$5.

7.5.1 Multiple Tries For P

As long as we are picking the projection matrices at random (from e.g. a normal distribution, as above), an obvious improvement to our method is to pick the best P from a set of random matrices. We do this by running the algorithm N_{tries} times (per layer), each with a different choice of P_i, and pick the one that yields the lowest LSD, $\frac{1}{m}\|Y - Z_i Z_i^+ Y\|_F^2$ (see Equation 5.3), on its corresponding representation, Z_i. **Note, that the pseudoinverse only needs to be**

```python
import torch

# CONFIG:
nhid = 16
afun = torch.nn.CELU()
x, y = get_dataset("sine")

# FIT LAYER 1:
b = torch.ones((x.shape[0], 1))
Z_0 = torch.cat([b, Z_0], dim=1)
P_1 = torch.randn(1 + Z_0.shape[1], nhid)
T_1 = torch.cat([Z_0, y], dim=1) @ P_1
W_1 = torch.linalg.pinv(Z_0) @ T_1

# FIT LAYER 2:
Z_1 = afun(Z_0 @ W_1)
Z_1 = torch.cat([b, Z_1], dim=1)
P_2 = torch.randn(1 + Z_1.shape[1], nhid)
T_2 = torch.cat([Z_1, y], dim=1) @ P_2
W_2 = torch.linalg.pinv(Z_1) @ T_2

# FIT OUTPUT LAYER:
Z_2 = afun(Z_1 @ W_2)
Z_2 = torch.cat([b, Z_2], dim=1)
W_3 = torch.linalg.pinv(Z_2) @ y
yhat = Z_2 @ W_3
```

Figure 7.3: **Fitting a 3-layer MLP with Platonic Projections.** All we need is a few lines of code, using random matrices, column-wise concatenation, pseudoinverses, and matrix multiplication.

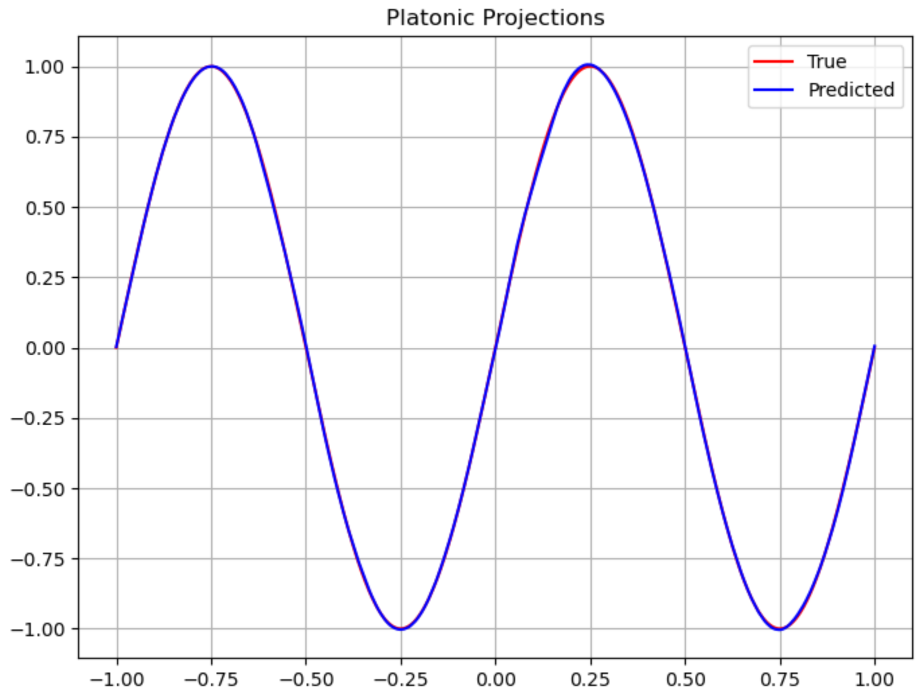


Figure 7.4: Fitting a 3-layer, 1-16-16-1 MLP to a sinusoid.

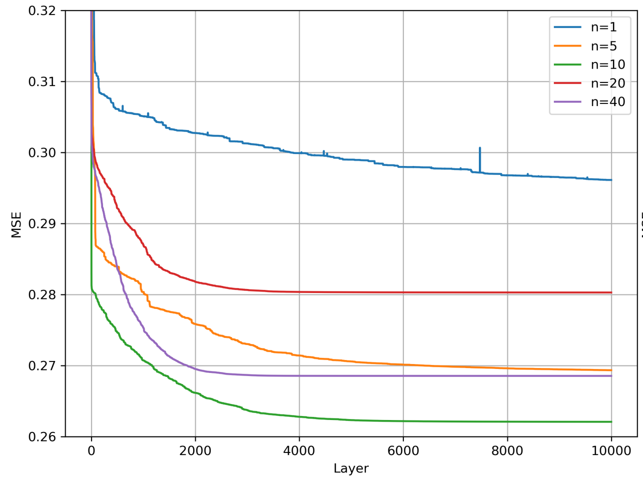
computed once per layer! We only need to recompute the target, T_i, and the solution $W_i = Z_{i-1}^{+}T_i$ (reusing the pseudoinverse).

Figure 7.5 illustrates the effect of increasing N_{tries}. We fitted 10,000-layer networks to the sine problem, using four different layer widths for the hidden layers. There is a clear improvement going from one to just five tries per layer, and error curves become more stable (less noisy). **Clearly, the choice P_i matters.**

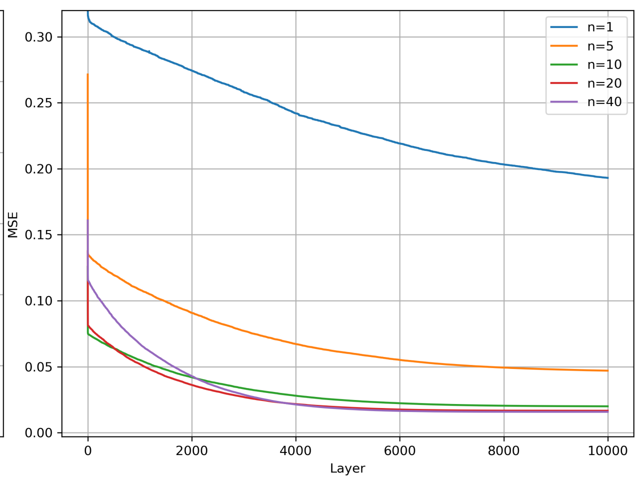
In Figure 7.6, we see the lowest mean squared errors (AKA LSD), achieved at any layer of the 10,000-layer networks, plotted as a function of N_{tries}. Overall, going from one to five tries has the most significant effect.

7.5.2 Scaling Factors α & β

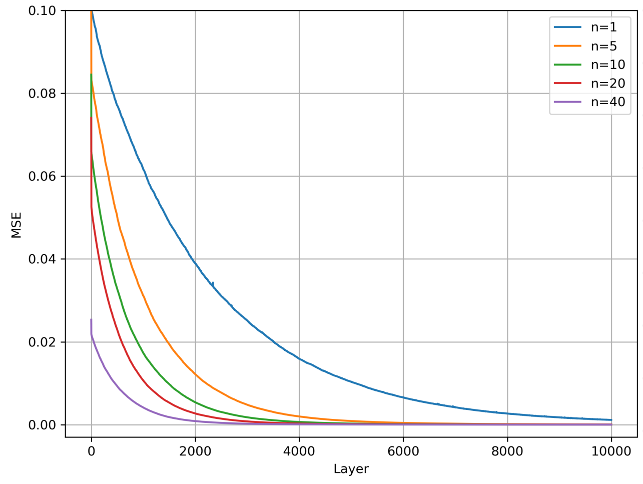
Scaling the target matrices also has a clear effect on the results. In practice, we thus compute them as such: $T_i = \alpha \left[\begin{array}{c|c} Z_{i-1} & \beta Y \end{array} \right] P_i$. We can think of T_i as a linear mixture of Z_{i-1} and Y, where β controls the weight of Y, and α scales the magnitudes of the components in T_i. We observe that it is generally a good idea to give higher weight to Y than to Z_{i-1}. The right choice of α may depend on numerical aspects of the underlying algorithm used to compute the pseudoinverses (typically obtained via the SVD). Figure 7.7 shows the impact of the scaling factors on the results.



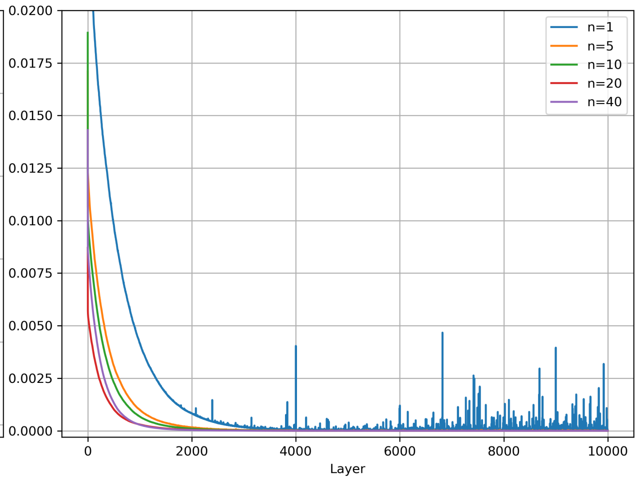
(a) Num. of hidden units = 2



(b) Num. of hidden units = 4



(c) Num. of hidden units = 8



(d) Num. of hidden units = 16

Figure 7.5: Illustrating the effect of increasing the number of tries, N_{tries} (denoted as "n" in the figure legends). 10,000-layer networks were fitted to the sine problem for four different layer widths (i.e. number of units in each of the hidden layers): 2, 4, 8, and 16. There is a clear improvement going from one to just five tries per layer; especially for the two narrowest networks (2 and 4 hidden units). For the widest network (16 hidden units), the error curve of the single-try network becomes very noisy after around 2000 layers—while the others remain smooth.

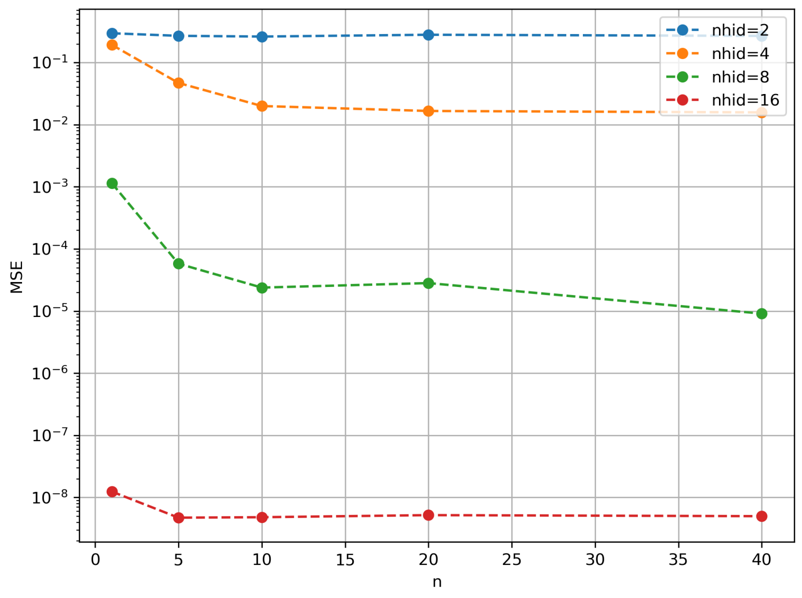


Figure 7.6: The lowest errors achieved (in any layer) as a function of the number of tries, n. “nhid” denotes the number of units in each of the hidden layers (AKA layer widths). The y-axis is logarithmic. Going from 1 to 5 tries has the most significant effect overall.

7.6 The Two Spirals Problem

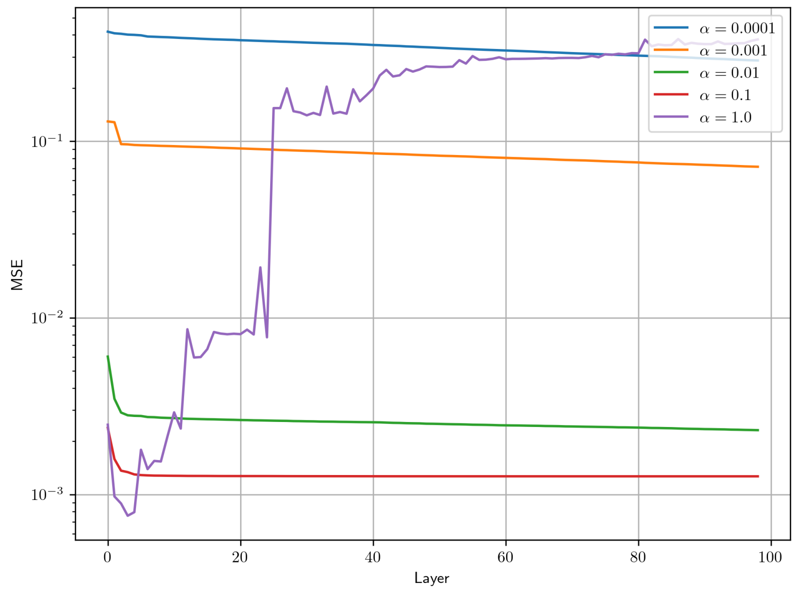
The two spirals problem, depicted in Figure 7.8, is somewhat of a classic challenge for neural nets. According to [86], it as was first proposed in a post on the connectionist mailing list by Alexis P. Wieland of the Mitre Corporation (probably around 1987-1988).

7.6.1 Baseline Result

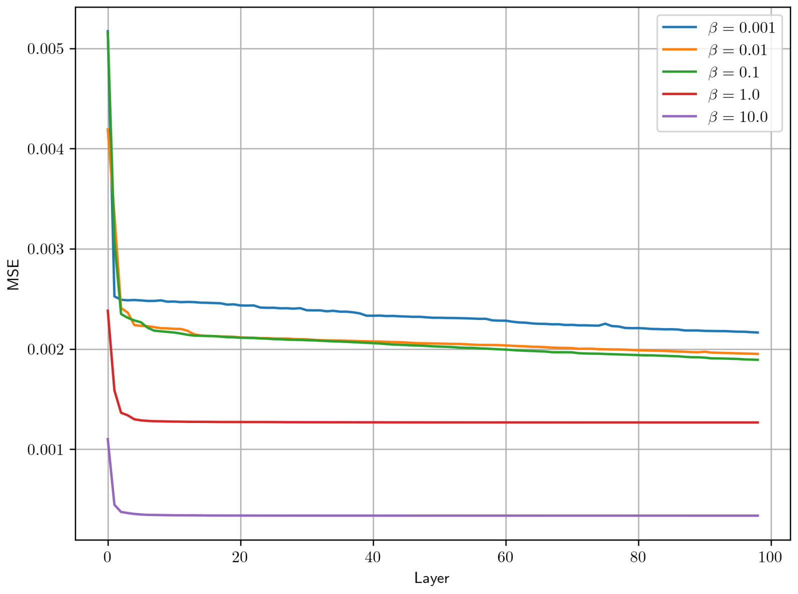
By today’s standards, this is an easy binary classification problem for a neural network. Indeed, we can fit the training data arbitrarily well to a 5-layer MLP with *tanh* activations, using SGD + backprop. We used a 2-32-32-32-32-1 network and trained it for 75 epochs (in single precision) with a learning rate of 0.01, no weight decay, a momentum of 0.9, and a batch size of 32. The learning rate was lowered after 56 epochs (divided by ten). The resulting cross-entropy loss curve is shown in Figure 7.9. **The final cross-entropy loss was 0.334, and the classification accuracy was 99.17%.** The training ran on an NVIDIA RTX 4080 GPU, and took **52 seconds**.

7.6.2 Platonic Projections

Fitting the same 5-layer MLP to the two spirals data with platonic projections, we set our scaling factors to $\alpha = 0.1$ and $\beta = 10$. With $N_{tries} = 1$, the classification accuracy is just 89.73%, and the execution time is 0.018 second, while with $N_{tries} = 5$ it is 92.69% (0.039 second), with



(a) The effect of scaling factor, α. The highest value yields the lowest layer-local error but also causes the process to diverge in the subsequent layers.



(b) The effect of scaling factor, β. Higher is better, but going to e.g. 100 makes the results worse.

Figure 7.7: Illustrating the effect of changing the scaling factors, α and β, for 100-layer networks with 8 units in each hidden layer.

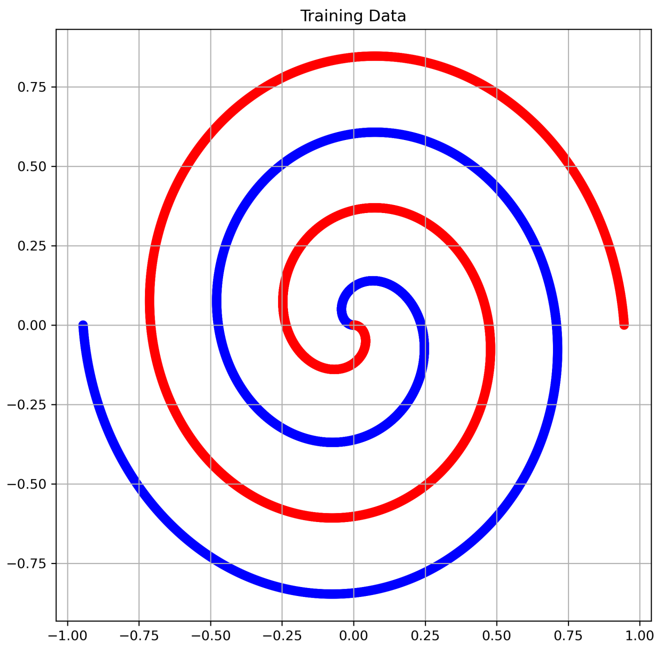


Figure 7.8: The Two Spirals Learning Problem. In our experiments, we used 5,000 samples.

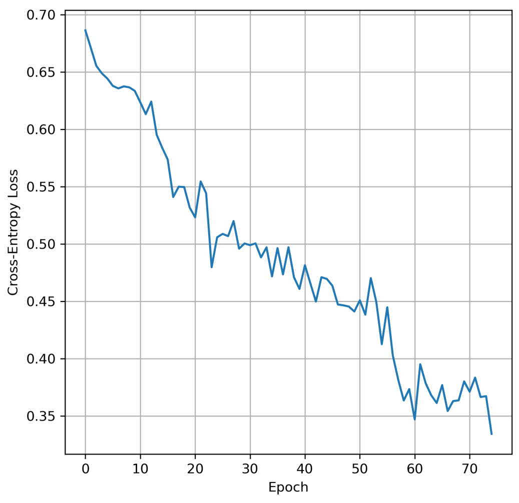


Figure 7.9: Loss curve for the two spirals problem. After 75 epochs, the cross-entropy loss was 0.334 and the classification accuracy was 99.17%.

$N_{tries} = 50$ it is 94.51% (0.249 second), and with $N_{tries} = 500$ the accuracy is 94.37% (2.03 seconds).

The most significant factor contributing to this bad result is that we deliberately seeded the random number generator with a particularly "unlucky" number. However, as it turns out, it also stems from numerical issues related to the underlying algorithm used to compute pseudoinverses in PyTorch. More on that in Section 7.6.2 below.

Faster Execution With Least Squares

When we are not using many tries per layer, and thus do not need to reuse Z^+_{i-1}, we may take advantage of PyTorch's least squares implementation, *torch.linalg.lstsq*. Instead of first computing the pseudoinverse, and then the matrix product, $W_i = Z^+_{i-1}T_i$, we can do it all in one single call. This is usually faster. For example, fitting a 5,000-layer narrow MLP (12 hidden units per layer) takes **3.4 seconds** with least squares, and **6.3 seconds** when using pseudoinverses (on our CUDA device).

To add $\mathbf{L_2}$ regularization, we can use the well-known closed-form solution to ridge regression: $W = (X^TX + \lambda I)^{-1}X^TY$, where the "ridge parameter" $\lambda \geq 0$ controls the strength of the regularization. Adding positive numbers to the diagonal (with λI), also improves the *numerical stability* as it lowers the condition number of X^TX. Moreover, as X^TX is square, we can now also use *torch.linalg.inv*. Even better, we can use *torch.linalg.solve* which is both faster (like with *lstsq*) and more numerically stable than *torch.linalg.inv*.

Numerical Stability

The pseudoinverse implementation relies on the SVD, which is computed using the *gesvd* and *gesvdj* algorithms of NVIDIA's cuSOLVER library [1]. So far, we have been using the standard single-precision (32-bit) floating point numbers on CUDA-enabled GPUs. If we switch to double precision, we immediately get 99.99% accuracy (using the same "bad seed", and $N_{tries} = 1$). Of course this takes longer, 0.1 second, and consumes twice as much memory. **Luckily, we *can* improve the single-precision accuracy, by simply setting the (absolute) tolerance of *torch.linalg.pinv* to 1e-5.**. This yields a slightly better result of 95.60% accuracy. The default tolerance used by PyTorch is set relative to the matrix dimensions and the data type.

We studied the numerical stability of platonic projections over 1,000 runs on both the CPU and a CUDA-enabled GPU. The results are listed in Table 7.1. Comparing the mean and standard deviation of the classification accuracy over 1,000 tests, the best results are achieved in double precision (64 bits) on both the CPU and the GPU. Manually setting the tolerance of *torch.linalg.pinv* significantly improves the single-precision results. Note, that the table includes results for two different settings for the scaling factors, as well as for *torch.linalg.lstsq* and *torch.linalg.solve*. In 64 bits, all three methods (*pinv*, *lstsq*, and *solve*) are fairly robust to the change in scaling factors. In 32 bits, the CPU implementation of *solve* fails with an error message saying that the input matrix is singular. In this case, we could fix the problem by increasing the amount of L_2 regularization (thus lowering the condition number of the matrix). We used a low number for the L_2 penalty in order to illustrate the issue of stability.

[1]See docs.nvidia.com/cuda/cusolver/index.html

	CPU			**CUDA**		
	32 bits tol=default	**32 bits** tol=1e-5	**64 bits** tol=default	**32 bits** tol=default	**32 bits** tol=1e-5	**64 bits** tol=default
	$\alpha = 0.1, \beta = 10.0$			$\alpha = 0.1, \beta = 10.0$		
pinv	80.25 ± 8.11	87.16 ± 2.51	99.17 ± 1.63	80.71 ± 7.73	86.94 ± 2.84	99.15 ± 1.66
lstsq	61.92 ± 8.75	87.75 ± 11.33	98.50 ± 2.57	87.13 ± 0.93	87.14 ± 0.95	99.92 ± 0.21
solve	Failed	Failed	94.87 ± 2.43	84.42 ± 5.72	84.42 ± 5.64	94.87 ± 2.49
	$\alpha = 1.0, \beta = 2.0$			$\alpha = 1.0, \beta = 2.0$		
pinv	97.46 ± 1.80	99.83 ± 0.36	99.93 ± 0.19	97.48 ± 1.74	99.92 ± 0.16	99.93 ± 0.19
lstsq	95.95 ± 3.04	95.70 ± 10.28	99.93 ± 0.17	99.90 ± 0.20	99.88 ± 0.23	99.92 ± 0.18
solve	Failed	Failed	99.95 ± 0.15	92.31 ± 6.47	92.78 ± 5.84	99.94 ± 0.15

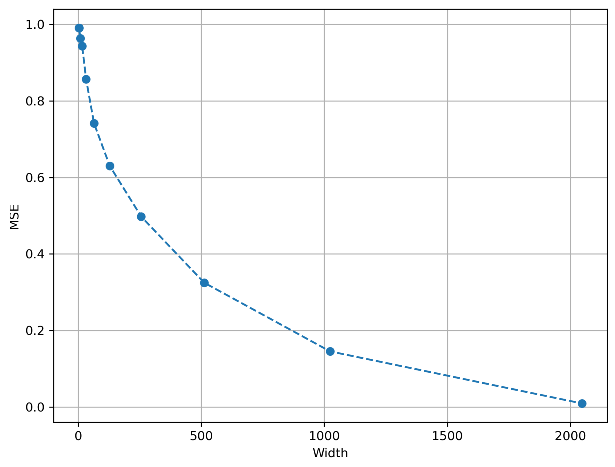
Table 7.1: **Numerical stability of platonic projections.** Showing the mean and standard deviation of the classification accuracy over 1,000 tests. Comparing two different settings for the scaling factors (α and β), and two different PyTorch methods: *torch.linalg.pinv* (**pinv**) and *torch.linalg.lstsq* (**lstsq**). In 64 bits, both methods are fairly robust to the change in scaling factors. Manually setting the tolerance (tol) significantly improves the stability of the results for **pinv** overall, and in one case for **lstsq**.

7.7 The Checkerboard Problem

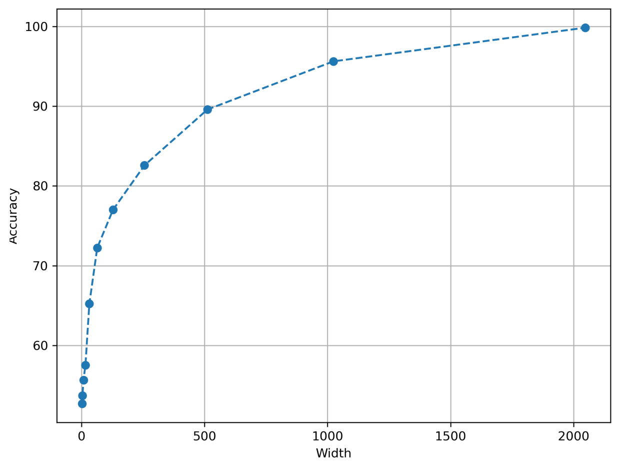
In Section 5.4.2, we saw how the checkerboard problem contains symmetries that prevent it from being learned easily with SGD + backprop. We showed, how filtering away a significant amount of data points could resolve the issue. Not being gradient-based, platonic projections do not suffer from that same problem. We can solve for a solution in closed form.

We fitted two MLPs, a deep and a shallow one, in both single and double precision with $N_{tries} = 1$. The layer-wise least squared distances (MSEs of the layer-local least square solutions) are shown in Figure 7.10. We tried two different settings for the scaling factors. **Interestingly, for this experiment *torch.linalg.pinv* performed much better than *torch.linalg.lstsq*.** The latter had troubles converging, was very slow, and gave poor results. In both precisions, and for both choices of scaling factors, the deep 500-layer MLPs all got 100% classification accuracy. The shallower MLPs got from 99.88% to 100% accuracy.

One thing to note about these results: the networks needed to have a lot of capacity in order to get above 99% accuracy. *This could indicate, that platonic projections are not efficient with respect to network capacity.* Figure 7.11 shows the MSEs and accuracies achieved with a shallow 4-layer MLP for a range of layer widths (number of units in each hidden layer). At 1024 units, the accuracy is 95.62% and at 2048 units it is 99.84%. The latter network thus has $3 \times 2048 + 1 = 6145$ nodes, and more than 8.4 million parameters. That is a lot. In essence, we are just memorizing 5,000 two-dimensional samples. It seems evident, that a much smaller network should be able to capture that information.



(a) MSE vs width



(b) Accuracy vs width

Figure 7.11: Mean squared error (MSE) and classification accuracy as a function of layer width (no. of units per hidden layer). Obtained from shallow (4-layer) MLPs with ReLU activations, $N_{tries} = 5, \alpha = 0.08, \beta = 200$. The widths used are powers of two: $2, 4, 8, 16, \ldots, 2048$.

7.8 Going Deterministic With PCA

Thus far, we have relied on random projection matrices, P_i, drawn from a normal distribution. Obviously, a deterministic solution would be desirable. The big question then is: *how do you deterministically pick a good projection matrix?* Below, we will introduce two complimentary approaches.

7.8.1 Platonic PCA

Consider the representation $Z \in \mathbb{R}^{m \times n}$ and $Y \in \mathbb{R}^{m \times k}$. The purpose of a projection matrix, P, acting on an augmented representation, $A = \alpha \left[\begin{array}{c|c} Z & \beta Y \end{array} \right]$, is to produce a target representation, T, in which the classes are separated. There is a simple way of utilizing A itself to find a reasonable candidate for a projection matrix. Namely, by using the *principal components* of A. In this augmented space, if β is large enough, the variance in the Y-dimensions (i.e. the k last columns in A) will dominate and thus pull (at least some) principal components in their direction. Those components will be *discriminative* directions in the platonic representation.

Figure 7.12 illustrates the situation for linearly separable data in 2D. As we can see, two of the three principal components are discriminative. That is, they are *not* orthogonal to the third dimension (containing the labels, $y \in \{-1, +1\}$). In this specific case, it means that their matrix-vector products with A will yield target vector representations $t_i \in T$ in which the two classes are separated. In the general case, including datasets with highly non-linear decision boundaries, this might not exactly be the case—unless we pick a very large value for β. At this point, it is unclear to us if that would be desirable. We may not need perfect separation of classes in the general case (for the *target vectors*, that is).

As also depicted in Figure 7.12, an appealing property of the principal components of A is that they actually reveal a solution to $Xw = y$. The blue vectors show the optimal solution, w^*, as found by an SVM. Two of the principal components are parallel with w^* in the two-dimensional plane where the original data, X, lies. Or, said in another way, their orthogonal projections onto the plane spanned by the first two dimensions, land exactly on top of w^*. Thus, in this specific case, we can solve the equation, $Xw = y$, by taking the first two components from the first principal component, c: $w = \begin{bmatrix} c_1 & c_2 \end{bmatrix}^T$, where $c = \begin{bmatrix} c_1 & c_2 & c_3 \end{bmatrix}^T$. This gives us the right direction for w, such that $\text{sign}(Xw) = \text{sign}(y)$. To minimize $\|y - Xw\|_2$, we would also need to correctly scale the solution, such that $w = \lambda \begin{bmatrix} c_1 & c_2 \end{bmatrix}^T$, for some $\lambda > 0$.

In the general case, particularly the non-linear one, we cannot expect the principal components to line up this perfectly with the optimal solution. However, according to our preliminary investigations, they do tend to still be correlated with it. For example, this is illustrated with a sinusoidal decision boundary in Figure 7.13. This begs the question, if we could indeed use the principal components of A as our *weight* vectors? However, in our experiments, this does *not* appear to work nearly as well as using them as *projection* vectors, $p_i \in P$.

7.8.2 Sorted PCA

A limitation of Platonic PCA is that it can produce at most $n + k$ weight vectors. This is unfortunate, in that a common pattern for neural nets is to have a vast increase in the number of dimensions (features) in the very first layer. Therefore, we need a deterministic method that can handle that.

Arguably, the role of the very first layer is not necessarily to be discriminative, but rather to act as a good *communication channel* for the layers upstream. To do this, it should contain weight vectors that reflect the distribution of the data, such that they mainly point in directions where a lot of the data lies. In other words, they should be *correlated* with the data. **The simplest way of achieving this is to use samples from the input data itself as weight vectors.** For example, we could draw them uniformly at random from the dataset. For a large enough sample, this selection of vectors *would* indeed reflect the distribution of the data, as required. However, we can do much better than that.

Once again, we will use PCA. In the following manner:

1. Take the first principal component, c, of either X or $A = \left[\ X \mid Y\ \right]$.
2. Compute $v = Xc$.
3. Obtain the indices, i, that sort the components in v in ascending order.
4. Pick the samples (rows) from X that correspond to evenly spaced indices in i (i.e. pick every k'th indice).

To be clear: *this is* ***not*** *a platonic projection, but a way of selecting samples from the dataset to be used as weight vectors.* Our PyTorch implementation of the algorithm is listed in Figure 7.14. This simple approach works quite well in our experiments. In Section 7.9.2, we compare its performance to other approaches.

7.8.3 Fully Deterministic Platonic Projections

As it turns out, our two PCA-based methods work best in combination. Thus, for fully **deterministic platonic projections**, we use *Sorted PCA* in the first layer only, and *Platonic PCA* for all the subsequent layers. Sorted PCA only works well in the first layer, and even when reducing the dimensionality of the input it outperforms Platonic PCA (except from in the more extreme cases; see Figure 7.15). We will show this in the next section.

7.9 MNIST Results

For the final test of our method, we used the MNIST dataset of handwritten digits [90]. Albeit, a classical benchmark in machine learning, it is a very easy learning problem by today's standards. In other words: *if our method does not do very well on this data, it is not reasonable to believe that it will work well in general.*

7.9.1 Baseline Result

As our baseline result, we trained a three-layer, 784-392-50-10, MLP with ReLU activations for ten epochs. We used SGD with a batch size of 128, a learning rate of 0.02, weight decay of 0.003, a momentum of 0.9, and softmax outputs with the cross-entropy loss. The learning rate was divided by 10 after the eighth epoch. On the training set, the final classification accuracy over the ten classes was **99.92%**. The final test accuracy was **98.12%**.

7.9.2 Single Hidden-Layer Networks

As discussed previously, the first layer plays a special role in a neural net. Thus, we compared the performance of four different methods on MLPs with a single hidden layer and ReLU activations:

1. Purely random weights (for the hidden layer), $W_1 \sim \mathcal{N}(0, 1)$; equivalent to the Extreme Learning Machine [62, 63].
2. Platonic projections with random projection matrices, $P \sim \mathcal{N}(0, 1)$.
3. Platonic PCA (deterministic).
4. Sorted PCA (deterministic).

To be clear, we used the above approaches to compute the weights for the single *hidden* layer, and then solved as linear equation for the output layer. **For all our MNIST experiments, we used a one-hot representation for the target class labels.** We used the closed-form ridge regression formulation with *pytorch.linalg.solve*, and the L_2 penalty set to a high value of 50,000. Empirically, this helps a lot for the linear systems being solved; indicating that they are not well conditioned. For all three platonic methods, we set $\alpha = 1$ and set $\beta = 10,000$ for the deterministic methods, and $\beta = 10$ for the random projections. These values were decided empirically.

Table 7.2 shows the results for five different hidden layer widths: 98, 196, 392, 784, and 1568. We computed the mean and standard deviations on the test accuracy over **100 trials** (for

Method	width = 98	width = 196	width = 392	width = 784	width = 1568
Train Acc.					
Random. W	76.32 ± 0.81	83.17 ± 0.34	87.98 ± 0.22	91.66 ± 0.12	94.51 ± 0.07
Random. P	88.84 ± 0.17	$\mathbf{90.41 \pm 0.11}$	91.98 ± 0.08	93.63 ± 0.06	95.26 ± 0.05
Platonic PCA	$\mathbf{88.85 \pm 0.00}$	89.52 ± 0.00	90.58 ± 0.00	91.53 ± 0.00	*N/A*
Sorted PCA	85.64 ± 0.00	90.31 ± 0.00	$\mathbf{92.89 \pm 0.00}$	$\mathbf{94.87 \pm 0.00}$	$\mathbf{96.38 \pm 0.00}$
Test Acc.					
Random. W	77.28 ± 0.89	83.86 ± 0.40	88.26 ± 0.28	91.52 ± 0.23	93.89 ± 0.15
Random. P	89.19 ± 0.22	90.55 ± 0.14	91.89 ± 0.13	93.34 ± 0.12	94.63 ± 0.11
Platonic PCA	$\mathbf{89.21 \pm 0.00}$	89.76 ± 0.00	90.63 ± 0.00	91.36 ± 0.00	*N/A*
Sorted PCA	86.19 ± 0.00	$\mathbf{90.71 \pm 0.00}$	$\mathbf{93.14 \pm 0.00}$	$\mathbf{94.64 \pm 0.00}$	$\mathbf{95.88 \pm 0.00}$

Table 7.2: **Classification accuracies for a two-layer MLP (**ReLU **activations) fitted to MNIST, for five different hidden layer widths**. For the two randomized methods, the results are means and standard deviations over **100 trials**. For all widths, the three platonic projections methods (Random P & Deterministic) do better than random weights (aka Extreme Learning Machine [62, 63]). The deterministic PCA-based algorithms, consistently achieve the highest test accuracies.

the randomized methods). As shown, platonic projections consistently outperform the networks with purely random weights in the hidden layer (the Extreme Learning Machine). Sorted PCA works best for all but the narrowest hidden-layer width (98 units).

We also plotted the MSE and accuracy over a much larger range of hidden-layer widths. For this, we skipped Platonic PCA and stuck to Sorted PCA for the deterministic approach. For the two randomized methods, we picked the models with the highest test accuracy of 5 trials. The results a shown in Figure 7.15. Again, we see a clear dominance of our method over purely random weights. Interestingly, random projections consistently yield lower MSEs on both the training and the test data, while Sorted PCA clearly wins on the accuracy.

7.9.3 3- & 4-Layer MLPs

Finally, we tested deterministic platonic projections on three- and four-layer networks. We started with out baseline network, 784-392-50-10, and increased the number of parameters from there. In total, we tested nine different architectures (all with ReLU activations).

Re-Weighted Samples For Platonic PCA

For these experiments, we employ another method which improves our results a bit. That is, at each layer, we keep track of the errors on each sample. We keep an exponential moving average (EMA) over the squared errors for each sample, and use them as weights (or importance) *only* when we compute the projection matrices, P_i, using platonic PCA.

At the onset of training, we initialize a vector of ones, $d \in \mathbb{R}^{m\times 1}$. After fitting each hidden layer, we fit a decision layer to the obtained hidden representation, Z_i, and compute an output estimate, $\hat{Y}$. Then, for each sample, we update d as such: $d_j = \lambda_{EMA} \cdot d_j + (1 - \lambda_{EMA}) \cdot e_j$,

				Train	Test	Train	Test
Hid. Layer Widths	**#Param.**	**L$_2$**	**Time**	**MSE**	**MSE**	**Acc.**	**Acc.**
392-50	0.33M	0.50	0.06s	1.5E-02	1.5E-02	95.07	95.07
392-250	0.41M	0.50	0.06s	1.0E-02	1.1E-02	95.54	95.36
588-350	0.67M	0.01	0.07s	8.6E-03	9.0E-03	96.50	96.25
784-392	0.93M	0.03	0.09s	7.6E-03	8.4E-03	97.04	96.55
784-588-588	1.43M	0.01	0.15s	6.3E-03	7.6E-03	97.30	96.62
1568-1200-1000	4.32M	0.01	0.40s	4.0E-03	6.0E-03	98.57	97.32
1960-1568	4.63M	0.01	0.39s	4.2E-03	6.2E-03	98.71	97.51
3136-2000-1500	11.75M	0.05	1.04s	2.3E-03	5.3E-03	99.56	97.58
6500-4500-3000	47.89M	0.03	4.80s	1.6E-03	8.2E-03	99.90	97.83

Table 7.3: **MNIST results for *deterministic* Platonic Projections on ten different 3- to 4-layer MLPs with** ReLU **activations.** The weights for the first layer were found using *Sorted PCA*. Subsequent hidden layer were fitted to *weighted Platonic PCA* targets ($\lambda_{EMA} = 0.02$). We used the closed-form ridge regression formulation with L_2 regularization, and *torch.linalg.solve* in single precision on an RTX 4080 GPU. Both the train and test accuracy is strictly increasing in the number of parameters.

where $e_j = \|y_j - \hat{y}_j\|_2^2$ is the sum-of-squares error on the j'th example. Now, when we compute the platonic PCA, we use the augmented matrix, $A = \alpha \left[\ Z_{i-1} \mid \beta d \odot Y\ \right]$. That is, each of the one-hot row vectors $y_j \in Y$ are now scaled by their corresponding weight (or importance), d_j. This scheme improved our classification accuracies by around 0.5 to 1.5 %-point.

Results

For these experiments, we used *very large* values for the β scaling factor: from $1\mathrm{e}6$ to $8\mathrm{e}7$. These large values markedly improve the classification accuracies. For a 784-2500-1500-500-10 MLP we get just 90.19% training accuracy with $\beta = 10$, while it jumps to 99.14% with $\beta = 1\mathrm{e}7$ Also, the MSE drops by an order of magnitude. We set $\alpha = 1$ for all the models.

Again we used the ridge regression formulation with *torch.linalg.solve*. The L_2 factor varied slightly between the nine different architectures, and we used $\lambda_{EMA} = 0.2$ for updating the sample weights, d. The results are listed in Table 7.3.

As can be seen, the classification accuracy is strictly increasing in the number of parameters. This is a good indicator that our method *can* in fact improve with more capacity (and layers). However, the results are not great compared to our baseline. None of the models go above 98% test accuracy—even with almost 48 million parameters. Also, we see a good amount of overfitting. The amount of overfitting increases in the number of parameters, which is consistent with the traditional views on the *bias-variance trade-off*.

7.10 Discussion

By following the principle of gradual class separation (discussed in Chapter 5), we have shown how a simple method for estimating targets for hidden layers can work quite well. The method is elegant in its simplicity, easy to implement, and of course very efficient (at least with respect to execution time). Unfortunately, as we have seen with our MNIST results, it suffers from overfitting and inefficient use of network capacity. Perhaps, it is not so surprising that a closed-form full-batch method overfits. For instance, it has been known for years that using large batch sizes induce overfitting in DNNs [80], and that training longer can mitigate it [59]. Nonetheless, the shear value of a closed-form solution to neural nets (and any method significantly more efficient than backprop), warrants further exploration of our method. And, after all, we *have* shown some rather promising results. *Moreover, to our knowledge, our closed-form solution is the first of its kind. That is, one that works for multiple layers with arbitrary width.*

7.10.1 What Is Missing From Our Method?

Implicit Regularization

As we discussed in Chapter 6, and as our results for deep layer-wise learning (DLL) strongly indicate, there is *a lot* of **implicit regularization** present in the SGD + backprop approach. This is of course completely absent from platonic projections in its current form. It seems key, to explicitly address this if we are to improve our method. Perhaps, part of the solution to this issue is to convert our method into more of a **hybrid iterative approach**; based on mini-batches. This would also reduce the memory footprint, and make our algorithm more tractable. However, we currently do not know how to achieve this.

Poor Use of Network Capacity

Additionally, it is clear that the estimated hidden-layer targets, $T_i = \alpha \left[\ Z_{i-1} \mid \beta Y\ \right]$, are far from ideal with respect to efficiently utilizing network capacity. It is currently not understood why that is the case. For platonic PCA, one contributing factor may be the enforcing of orthogonality. This could very well not be desirable. In Figure 7.12, we can clearly see how the second principal component is not discriminative; it is orthogonal to third axis (containing the target, y). This is a direct effect of the orthogonality constraint. Other methods, such as *independent component analysis* (ICA) [50], do not have that constraint. Arguably, this would better enable them to adapt to the distribution of the data; and to "pay more attention" to the most dense and complex regions of the input space.

Activation Function Alignment

Until now, we have conveniently avoided the topic of the activation functions and their role. Our motivation for proposing platonic projections is heavily based on the *correlation filter view* described in Section 5.5.1. Here, we explained the **sort and filter** process, where the weight vectors *sort* the input examples, and the activation functions *filter* them. The present form of our method implicitly takes the approach that if we just do a good enough job with the sorting (i.e. finding discriminative directions), we can blindly apply any activation function. Admittedly, this is a rather lackluster approach. Most likely, we could improve our method by better **aligning** our targets, T_i, and the resulting weight vectors with the activation functions. Or, conversely, dynamically adapt the activation functions themselves to capture the best *selections* (see Section 5.5.1) from the (sorted) pre-activations, i.e. $a_j = Zw_j$.

7.10.2 Why Does It Work?

Even though our method is clearly motivated by the principle of gradual class separation, it is also not completely understood why it works as well as it currently does (albeit, it still leaves some to be desired). For example, one could imagine that only projecting the targets, Y, such that our hidden-layer targets are of the form $T_i = YP_i$, would also work well. *But it does not!* Our working hypothesis for this is, that by including the input representation, Z_{i-1}, and letting $T_i = \left[\ Z_{i-1} \mid Y\ \right] P_i$, we are more likely to get targets that can actually be reached from Z_{i-1}. That is, the target vectors, $t_j \in T_i$, will be in (or least not orthogonal to) the range of Z_{i-1} (i.e. in $\text{col}(Z_{i-1})$). Furthermore, when $T_i = YP_i$ and Y is a one-hot representation, all the examples belonging to the same class get assigned exactly the same hidden-layer target. This is inconsistent with what neural nets usually learn, and it makes the least squares problem harder (more ill-conditioned). Arguably, when we assign a unique target to each of the samples with $T_i = \left[\ Z_{i-1} \mid Y\ \right] P_i$, we are also relaxing the linear system (i.e. enlarging the set of acceptable solutions).

Another interesting perspective comes from expanding the solution (omitting the i's)

$$W = Z^{+}T = Z^{+} \left[\ Z \mid Y\ \right] P \tag{7.1}$$

Assume that $P = \left[\ P_Z \mid P_Y\ \right]$, and note that $Z^{+}Z = I$. Then Equation 7.1 expands into

$$W = P_Z + Z^{+}YP_Y \tag{7.2}$$

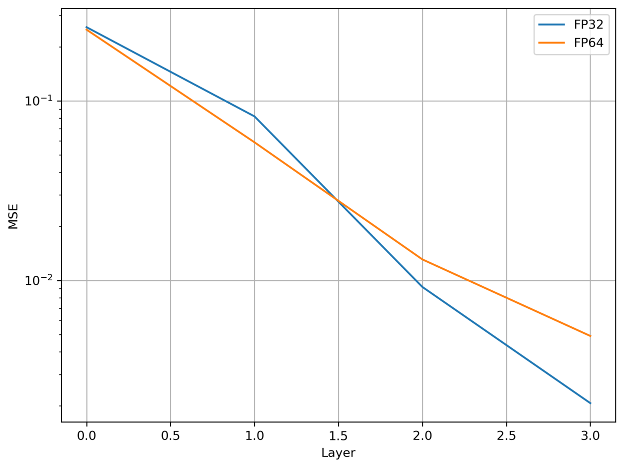
Thus, when P is a random matrix, our solution is little more than a random projection of the least squares solution, $Z^{+}Y$, plus a random matrix, P_Z. However, as mentioned above, $W = Z^{+}YP$ (or $T = YP$) does not work well in practice. This could suggest that adding the noise, P_Z, acts a regularizer in this case. Of course, when P is chosen deterministically, using Platonic PCA as described in Section 7.8.1, Equation 7.2 takes an entirely different meaning. Specifically, P_Z now contains the orthogonal projections of the principal components (of $A = \left[\ Z \mid Y\ \right]$) onto the plane where Z lies. Which is just the first n components in those principal component vectors (if Z is n-dimensional). Some of them will indeed be discriminative directions in the data. So, adding P_Z to $Z^{+}YP_Y$ would "push" the least squares solution (projected by P_Y) in the direction of the principal components of A. And that appears to be important (empirically).

7.10.3 Future Work

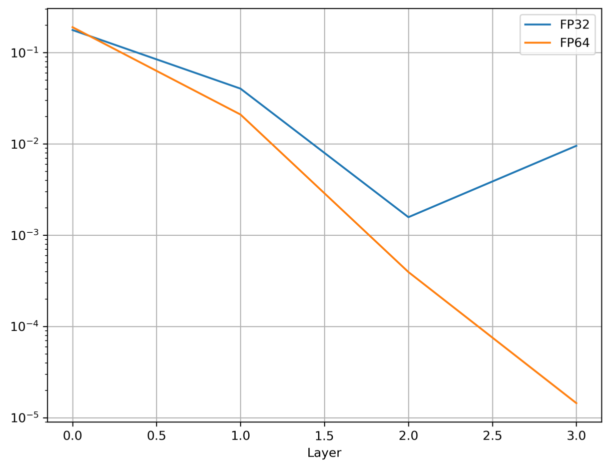
There are several interesting avenues worthy of investigation for the improvement of platonic projections. In no particular order, some of them are:

1. How to robustly improve a found solution. Currently, we are stuck with the first solution we find, and cannot gradually improve it.
2. Enabling the use of mini-batches; effectively turning it into a **hybrid iterative approach.**

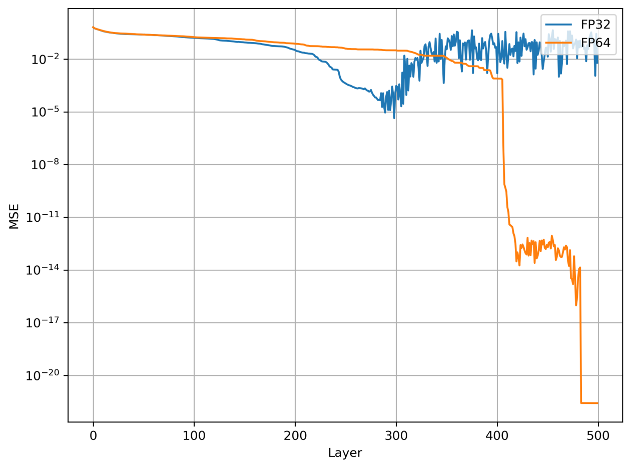
3. Exploiting partitioned, block matrix, and parallelized implementations of the pseudoinverse, such as [5, 25, 70, 138].
4. On a related note: how to use data augmentation techniques without making the least squares problems intractable.
5. Explicitly inducing low rank solutions and match the matrix rank behavior of the traditional approach.
6. Methods for explicitly mimicking implicit regularization. E.g. by using noise, as we did successfully with DLL.
7. Exploring alternatives to PCA for the deterministic methods. For example, ICA.
8. Adapt the method to convolutional and attention layers.
9. Investigate if our method is suitable as a weight initialization scheme.
10. Increasing the utilization of available network capacity. For example, by online pruning and selection of features, weight vectors, and projection vectors.
11. Investigate the suitability of platonic projections for **quantum computing**.



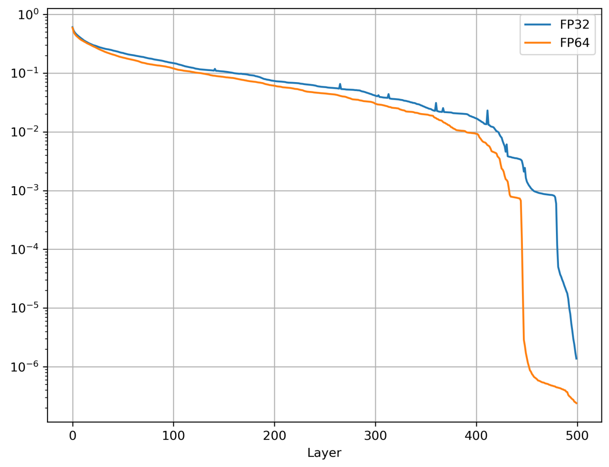
(a) #hidden units = 2500, #hidden layers = 4, $\alpha = 0.06, \beta = 25.0$



(b) #hidden units = 2500, #hidden layers = 4, $\alpha = 0.08, \beta = 200.0$

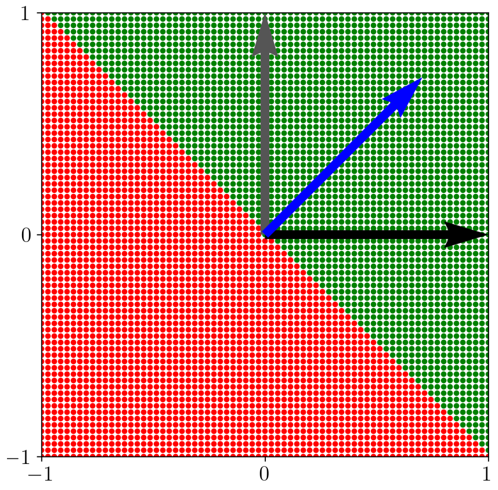


(c) #hidden units = 300, #hidden layers = 500, $\alpha = 0.06, \beta = 25.0$

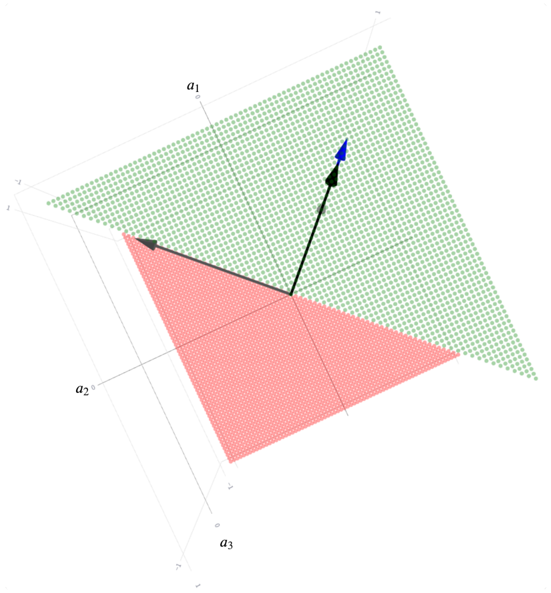


(d) #hidden units = 300, #hidden layers = 500, $\alpha = 0.08, \beta = 200.0$

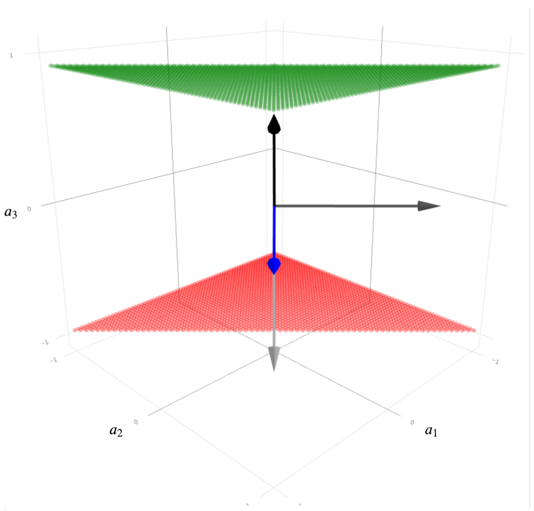
Figure 7.10: Layer-wise mean squared errors (MSE) on the checkerboard problem for two MLPs with $\$ \mathrm{ReLU}$ activations, two choices for α and β, and for both single and double precision (FP32 and FP64). **Top:** 4 hidden layers with 2500 units each. **Bottom:** 500 hidden layers with 300 units each. **Left:** $\alpha = 0.06, \beta = 25.0$. **Right:** $\alpha = 0.08, \beta = 200.0$. **With the deepest double-precision network, we see an extremely low MSE of** $2.6\mathrm{e}{-22}$.



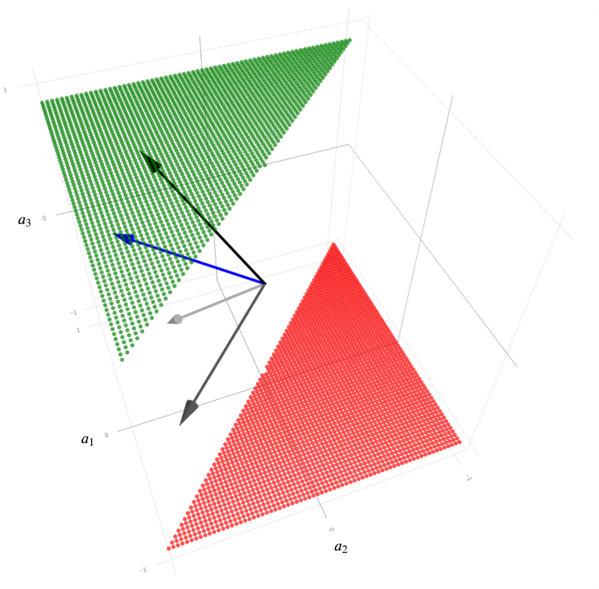
(a) The two-dimensional training data, X.



(b) PCA in the platonic space, $A = \left[\begin{array}{c|c} X & Y \end{array}\right]$. Angle 1.

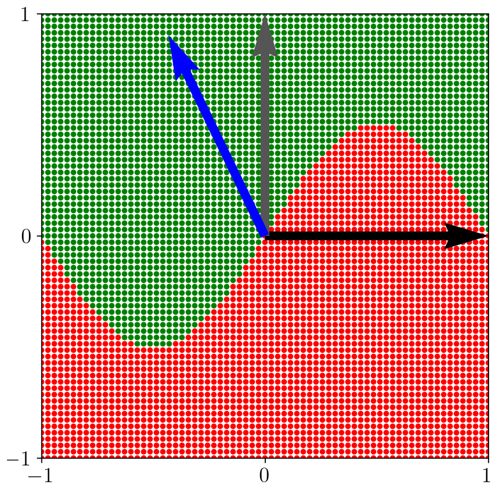


(c) PCA in the platonic space, $A = \left[\begin{array}{c|c} X & Y \end{array}\right]$. Angle 2.

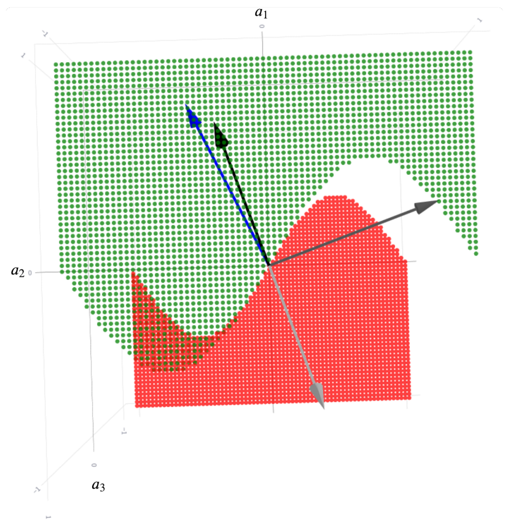


(d) PCA in the platonic space, $A = \left[\begin{array}{c|c} X & Y \end{array}\right]$. Angle 3.

Figure 7.12: **Platonic PCA illustrated.** Showing three different angles of the same system. The blue vectors, w^*, show the optimal solution to $Xw = y$ (found by an SVM). The black, dark gray, and light gray vectors are the first, second, and third principal components, respectively. In the platonic space, two of the principal components are **parallel** with w^* in the a_1a_2-plane where X lies.



(a) The two-dimensional training data, X.



(b) PCA in the platonic space, $A = \left[\begin{array}{c|c} X & Y \end{array}\right]$.

Figure 7.13: **Platonic PCA illustrated for a non-linear problem.** The blue vectors, w^*, show the optimal solution to $Xw = y$ (found by an SVM). The black, dark gray, and light gray vectors are the first, second, and third principal components, respectively. In the platonic space, the first principal component is **positively correlated** (but not parallel) with w^* in the a_1a_2-plane where X lies. The third principal component is negatively correlated with w^*.

```python
def sorted_pca(Z, num_samples):
    """
    Select samples evenly along the first principal component
    """
    # Get the first principal component, pc_1
    pc_1 = pca_torch(Z, num_components=1).pc.T
    # Project the data onto pc_1
    Z = Z @ pc_1
    # Sort the result
    ix = torch.argsort(Z[:, 0])
    n = ix.numel()
    # Pick evenly spaced indices
    i = torch.arange(0, n - 1, n / num_samples).round().long()
    # Return the indices of the samples to be used as weight vectors
    return ix[i]
```

Figure 7.14: Our PyTorch implementation of **Sorted PCA**. Assume that there is a function, *pca_torch*, that returns the principal components of an input matrix.

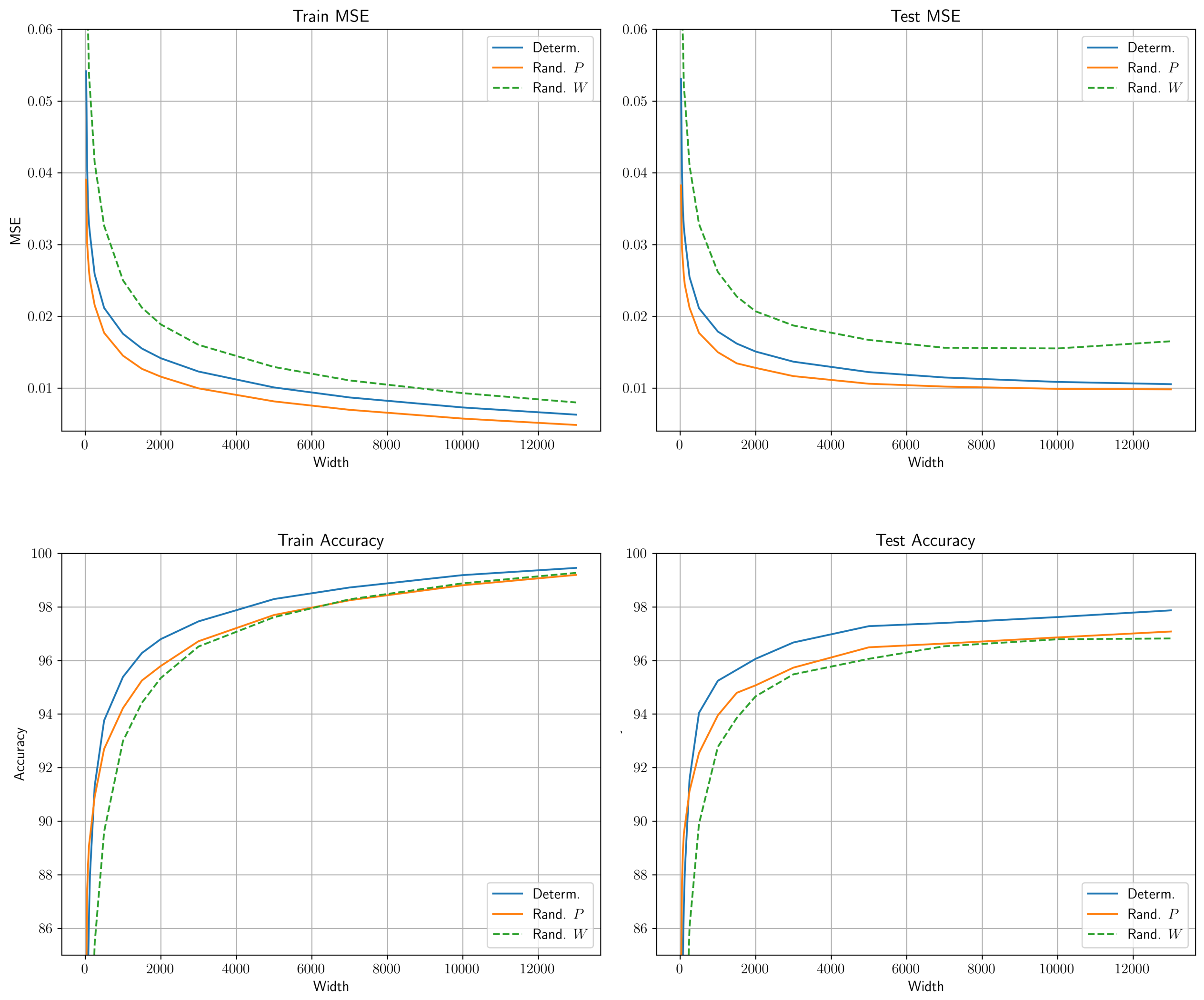


Figure 7.15: Mean squared errors (MSE) and classification accuracies for a two-layer MLP with ReLU activations fitted to the MNIST dataset. Plotted as a function of the width of the single hidden layer. The deterministic method, *Sorted PCA*, compared to platonic projections with random $P \sim \mathcal{N}(0,1)$, and with purely random weights, $W \sim \mathcal{N}(0,1)$. The latter is equivalent to the Extreme Learning Machine proposed in [62].

Chapter 8

Concluding Remarks & Further Thoughts

We have explored various methods and aspects related to the efficiency of deep learning.

8.1 Connecting the Dots: Entropy & Rank

In Chapter 3 we observed how the entropy over the distribution of the individual weights decreases during training, and how it allows us to compress the weights for distribution over a network. This reduction in entropy seems fundamental. Indeed, in Section 5.3.5 on information theory, we argued that well-trained neural nets should satisfy strict inequalities on the layer-wise representational and conditional entropies; at least if we assume that $H(X) > H(Y)$. If we believe that, then making the connection that this must also be reflected in the weights should be possible. From there, it would be valuable to theoretically verify that the entropy over the distribution of the individual weights must also decrease; thus providing a theoretical justification for the method proposed in Chapter 3. Related to this, there is the question of matrix rank, which is known to play an important role with respect to the generalization properties of DNNs. As discussed in Section 6.6, implicit regularization (including interlayer regularization) has been shown to induce low-rank solutions. For both the weights and the representations (activations), it would be interesting if a connection between their entropies and their ranks could be established. However, considering that an identity matrix has full rank and very low entropy, such a connection may be very hard, or impossible, to directly establish.

8.2 Extending Our Work on the Properties of Layers & Representations

Chapter 5 introduced various *useful* properties of the layers and representations of neural nets. *The Rubik's Cube View* 5.3.4 illustrates how same-class examples gradually cluster, or *bin*, together in the layers. And the *pure entropy*, $H^*(X;Y)$, provides a tangible way of quantifying this, while also establishing a natural connection to information theory. As we have seen, these ideas are closely related to the gradual class separability and linearization which has been a recurring theme in this thesis. We would like to extend these views and observations more generally to machine learning. Perhaps, they can inspire related observations on other methods that would be suitable for a different kind of textbook. However, as useful as our observations have been for our present purpose, they are not complete. Something is missing.

In some sense, the Rubik's Cube View, k-redundancy, and pure entropy, are all indicating that a k-nearest-neighbor classifier is all we need. **That is clearly not the case!** It seems obvious that the process of training DNNs with backprop + SGD *must* be doing something that significantly distinguishes the resulting model from a kNN classifier. It is currently not understood precisely what that is. One important aspect has to be the *implicit regularization* reported by so many authors (see Section 6.6), and explicitly addressed by our approach to deep layer-wise learning (DLL) in Chapter 6. It would be very interesting to investigate how the implicit regularization affects the binning of the data; and the pure entropy. Our DLL results clearly indicate, that our greedy layer-wise algorithm needed to be **not so greedy**, after all. In other words: we should not be too quick with the binning of the data.

8.3 How Do Layers *Actually* Regularize Each Other?

In Section 6.6 we briefly presented our two conjectures on forward & backward regularization. As we were able to turn those conjectures into a simple method (forward & backward dropout), that allowed us to match the results achieved with full backprop, it is reasonable to assume that they are valid. But it is also reasonable to say, that they probably do not paint the full picture.

8.3.1 Vanishing Gradients Regularize

One interesting conjecture, that we arrived at during our work on this thesis, is that **vanishing gradients regularize**. *Yes, the (infamous) problem of vanishing gradients [57] could play a key role in the very success of deep learning.* Our reasoning for making this somewhat surprising (possibly controversial) conjecture is comprised by the following statements:

1. The vanishing **magnitudes** of the gradient signals *likely* correspond to vanishing **information**.
2. The key question is: **what information (or magnitude) vanishes first?**
3. **Conjecture:** gradients stemming from the highest frequencies of variation in the data (i.e. in the labels) vanish first, while *the lowest frequencies vanish last.*

We will elaborate on statement 1 in the next section. Our conjecture in statement 3, if correct, would explain *why* lower layers tend to learn to capture the lower frequencies of variation, and conversely for the upper layers; as we suggested in Section 6.6 on interlayer regularization. **This would indeed have a regularizing effect, insofar as we can consider it a type of *smoothing*, or low-pass filtering, on the functions being learned in each of the layers.**

Vanishing Gradient Information

Trivially, scaling any discrete random variable (i.e. multiplying it with a non-zero scalar) does not change its entropy; it merely changes the bin-sizes. Thus, we must be careful when making statements that equate the vanishing magnitudes of the gradients with the loss of information. Nonetheless, we have seen evidence that noise builds up in both the forward and backward passes; i.e. that the representations/signals in the forward pass and the gradients/signals in the backward become increasingly *inconsistent (noisy)*, layer by layer. Somehow, this makes it tempting to conclude that information is lost in the process.

If we take the approach of Tishby et al. [133, 143], and apply the *data processing inequality* (DPI). **Then, a straightforward formulation of the vanishing gradients problem in terms of information follows.** Namely, that the following inequality holds for the gradients being backpropagated using the chain rule:

$$I(\frac{\partial \mathcal{L}}{\partial \hat{Y}}; \frac{\partial \mathcal{L}}{\partial Z_{k-1}}) \geq I(\frac{\partial \mathcal{L}}{\partial \hat{Y}}; \frac{\partial \mathcal{L}}{\partial Z_{k-2}}) \geq \cdots \geq I(\frac{\partial \mathcal{L}}{\partial \hat{Y}}; \frac{\partial \mathcal{L}}{\partial Z_1}) \tag{8.1}$$

The possibly controversial conjecture, that we would like to eventually prove, is that this equation should indeed be a *strict inequality*. Or, at least, if we freeze the weights at a given point in the training and consider the layer-wise gradients random variables then, *with high probability*, a strict inequality would be observed to hold.

One approach to proving this conjecture, would be to argue that the loss of information is caused by the frequent multiplication by zero (due to the flat regions of the activations having zero-gradients); i.e. re-parameterization by non-invertible functions. Over time, during the training process, this is arguably similar to lowering the sample rate. Now, if we apply something like **the Nyquist-Shannon Sampling Theorem**, this implies that the information lost from the gradient signal must be high frequency components; i.e. frequencies above the expected Nyquist rate. It is worth noting, that the Nyquist-Shannon Theorem also holds for non-uniformly sampled signals (see Landau (1967), Mishali et al. (2009)[85, 109]). *While this line of thinking may not end up strictly holding, we do find it appealing.*

8.3.2 What Are the Frequencies of Variation?

Unless when explicitly fitting a sinusoid with multiple frequency components, what exactly "frequencies of variation" means might not be immediately clear. Consider a set of m training examples, $x_i \in \mathbb{R}^2$. Assume that the data is uniformly sampled over an interval such that they form a rectangle centered at the origin. There is a left side of the data, X_L where all samples have negative first coordinates, and a right side, X_R, with positive first coordinates. Now, we assign labels to the samples in X_L such that, in the Rubik's Cube view (Section 5.3.4), they look like a checkerboard with many many squares (exactly as the leftmost representation in Figure 5.10). Conversely, for the right side, X_R, we assign labels such that the checkerboard has very few squares (like the rightmost representation in Figure 5.10). For this dataset, we would say that **locally**, in the left region, the frequency of variation is *high*. Similarly, the frequency of variation is *locally* low in the right side.

8.3.3 How the Frequencies of Variation Are Affected Differently

The Information Argument

Additionally, for our example data, we can conclude that for the left-side samples, the k-redundancy is low, while it is high for the right side. Hence, the k-redundancy is inversely proportional to the frequency of variation. **This relationship between redundancy and frequency is thus similar to the one captured by the Nyquist-Shannon Theorem.** That is, the higher the frequency, the more samples are needed (and the less the signal-redundancy is). Therefore, we would argue that *if* information is lost in the backprop chain, then it is more likely to relate to a high-frequency (high-entropy) region, like X_L, than to a low-frequency region. And, of course, if increasingly more information is lost in the backward pass, then the logical conclusion would then be: that the lower layers are more likely to capture low-frequency components (or regions in the data), because the information about the high-frequency components never reaches them (it has vanished). **In other words:** *low-frequency regions are **more robust to noise** (and information loss), and have a lower **sample complexity** than the high-frequency regions.*

The Magnitude Argument

Besides the information argument, another appealing one can be made in support of our conjecture that vanishing gradients act as a regularizer. It comes from reasoning about **how the magnitude of the gradient contributions of a region in the data is related to its frequency of variation.** Consider again our high-frequency region, X_L. In the loss surface, this region would give rise to a significantly flatter surface than the low-frequency region, X_R. Because, any weight update affecting the samples in X_L can potentially only correct a small number of labels—with a correspondingly small reduction of the loss. Conversely, in the low-frequency region, X_R, a single weight update can contribute a much larger reduction in the loss. **In other words:** *high-frequency (aka high-uncertainty, high-entropy, high-complexity) regions will tend to produce gradients with small magnitudes, while low-frequency regions will produce high-magnitude gradients.* Indeed, it seems there must be an inverse relationship between the frequency of variation and amplitude; i.e. in the loss surface and in the gradients. However, we also need to take into account the number of samples (which would also affect the amplitude), so any formal statement on this relationship must include the sample density of a given region as a factor.

8.4 Robust Layer-wise Training

In Chapter 6 we showed that, contrary to the apparent consensus in the community so far, layer-wise learning (DLL) *can* work well. However, it has to be said that layer-wise learning generally is harder. It is more brittle, probably because of the lack of orchestration and adaptation between the layers that would happen implicitly with full-model backprop. Future work on DLL should definitely include a broader investigation of the robustness of the method—and include data and architectures from more domains (e.g. language models).

It is commonly understood, that starting out with a large learning rate and annealing it during the training process is a good approach. Large learning rates can aid the process in escaping local minima and thus have a regularizing effect (see for example [93]). In our DLL experiments, we exponentially decayed the *initial learning rate* in the layer index, while exponentially increasing the weight decay. The former is in part motivated by the fact that the learning process as a whole is further along whenever we move from one layer to the next. The latter is to compensate for the reduced regularization caused by starting with a lower learning rate. However, we should also consider our brief analysis from above, on how vanishing gradients regularize, and why the layers capture progressively higher frequencies of variation. From this point of view, it makes sense that when you are fitting low-frequency regions in the data, you can afford to take large steps. As we move from the bottom and up through the layers, the frequencies of variation that are captured get higher, and thus the optimizer (i.e. the weights) must take smaller steps.

8.5 Practicality of a Closed-Form Solution

Arguably, the practicality of a closed-form solution to neural nets, such as our Platonic Projections proposed in Chapter 7, is probably somewhat limited. Even if it did match the state of the art on test accuracy and capacity-consumption, it is impractical on very large data and very large models. Therefore, as we stated in the discussion of the chapter, our method will probably evolve into more of a hybrid solution based on mini-batches.

However, there are certain domains in which Platonic Projections, even in its current form, would potentially be very valuable. As with DLL, edge devices could benefit tremendously from a computationally very efficient solution. Also, as mentioned earlier, quantum computing might turn out to be a very important application. Lastly, **every use-case where speed is more important than accuracy** could see huge benefits from our method. One candidate would be quantitative finance. For example, many **algorithmic trading applications** rely on extremely noisy data that comes in real time, and critical trading decisions must be made every second, or every minute. In the field, it is common to use simple linear regression because many many models can be fitted quite fast. Since the data is extremely noisy, high accuracy is not expected. In this scenario, *it is easy to imagine that the ability to fit more powerful models (MLPs) extremely fast (using our method), could have a big impact.*

Bibliography

[1] Amira Abbas, Robbie King, Hsin-Yuan Huang, William J Huggins, Ramis Movassagh, Dar Gilboa, and Jarrod R Mcclean. On quantum backpropagation, information reuse, and cheating measurement collapse. *Advances in Neural Information Processing Systems*, 36: 44792–44819, 12 2023. 7.1

[2] A Aizerman. Theoretical foundations of the potential function method in pattern recognition learning. *Automation and remote control*, 25:821–837, 1964. 7.4

[3] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A Next-generation Hyperparameter Optimization Framework. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 2623–2631, 7 2019. doi: 10.1145/3292500.3330701. URL `https://dl.acm.org/doi/10.1145/3292500.3330701`. (document), 6.7, 6.5

[4] Sanjeev Arora, Nadav Cohen, and Elad Hazan. On the Optimization of Deep Networks: Implicit Acceleration by Overparameterization. In *Proceedings of the 35th International Conference on Machine Learnin*, pages 244–253. PMLR, 7 2018. URL `https://proceedings.mlr.press/v80/arora18a.html`. 6.6

[5] Jerzy K. Baksalary and Oskar Maria Baksalary. Particular formulae for the Moore–Penrose inverse of a columnwise partitioned matrix. *Linear Algebra and its Applications*, 421(1):16–23, 2 2007. ISSN 0024-3795. doi: 10.1016/J.LAA.2006.03.031. 3

[6] Randall Balestriero and Richard G Baraniuk. A Spline Theory of Deep Learning. In *Proceedings of the 35th International Conference on Machine Learning*, pages 374–383. PMLR, 7 2018. URL `https://proceedings.mlr.press/v80/balestriero18b.html`. 5.3

[7] A Bansal, X Chen, B Russell, and AG Ramanan. Pixelnet: Representation of the pixels, by the pixels, and for the pixels. *arXiv preprint arXiv:1702.06506*, 2017. URL `https://arxiv.org/abs/1702.06506`. 4.13

[8] David G.T. Barrett and Benoit Dherin. Implicit Gradient Regularization. *ICLR 2021 - 9th International Conference on Learning Representations*, 9 2020. URL `https://arxiv.org/abs/2009.11162v3`. 6.6

[9] Ronen Basri, Meirav Galun, Amnon Geifman, David Jacobs, Yoni Kasten, and Shira Kritchman. Frequency Bias in Neural Networks for Input of Non-Uniform Density. In *Proceedings of the 37th International Conference on Machine Learning, PMLR 119:685-*

694, pages 685–694. PMLR, 11 2020. URL https://proceedings.mlr.press/v119/basri20a.html. 6.6.1

[10] Eugene Belilovsky, Michael Eickenberg, and Edouard Oyallon. Greedy Layerwise Learning Can Scale To ImageNet. In *Proceedings of the 36th International Conference on Machine Learning*, pages 583–593. PMLR, 5 2019. URL https://proceedings.mlr.press/v97/belilovsky19a.html. 5.3.3, 6.3.1

[11] Eugene Belilovsky, Michael Eickenberg, and Edouard Oyallon. Decoupled Greedy Learning of CNNs. In *Proceedings of the 37th International Conference on Machine Learning*, pages 736–745. PMLR, 11 2020. URL https://proceedings.mlr.press/v119/belilovsky20a.html. 6.3.1

[12] Yoshua Bengio. How Auto-Encoders Could Provide Credit Assignment in Deep Networks via Target Propagation. *arXiv preprint arXiv:1407.7906*, 7 2014. ISSN 0002-7863. doi: 10.1021/ja00062a066. URL http://arxiv.org/abs/1407.7906. 6.3

[13] Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. Greedy Layer-Wise Training of Deep Networks. *Advances in Neural Information Processing Systems*, 19(1):153, 2007. ISSN 01628828. doi: citeulike-article-id:4640046. URL http://papers.nips.cc/paper/3048-greedy-layer-wise-training-of-deep-networks.pdf. 6.3.1

[14] C M Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1 edition, 1995. ISBN 0195667999. 4.2, 4.3

[15] Christopher M Bishop. *Pattern Recognition and Machine Learning*. Springer, New York, NY, 1 edition, 8 2006. ISBN 9780387310732. URL https://link.springer.com/book/9780387310732papers2://publication/uuid/05A8B4CF-0248-4692-8B1D-DCC065B79465. 4.2, 4.3

[16] J.S. Bridle. Probabilistic Interpretation of Feedforward Classification Network Outputs, with Relationships to Statistical Pattern Recognition. In F. F. Soulié and J Hérault, editors, *Neurocomputing: Algorithms, Architectures and Applications*, number C, pages 227–236. Springer, Berlin, Heidelberg, 1990. ISBN 3642761550. doi: 10.1007/978-3-642-76153-9. 4.2

[17] Yuan Cao, Zhiying Fang, Yue Wu, Ding Xuan Zhou, and Quanquan Gu. Towards Understanding the Spectral Bias of Deep Learning. *IJCAI International Joint Conference on Artificial Intelligence*, pages 2205–2211, 12 2019. ISSN 10450823. doi: 10.48550/arxiv.1912.01198. URL https://arxiv.org/abs/1912.01198v3. 6.6.1

[18] Pratik Chaudhari and Stefano Soatto. Stochastic gradient descent performs variational inference, converges to limit cycles for deep networks. *2018 Information Theory and Applications Workshop, ITA 2018*, 10 2018. doi: 10.1109/ITA.2018.8503224. 6.6

[19] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training Deep Nets with Sublinear Memory Cost. 4 2016. URL https://arxiv.org/abs/1604.06174v2. 6.9

[20] Yixiong Chen, Alan Yuille, and Zongwei Zhou. WHICH LAYER IS LEARNING

FASTER? A SYSTEMATIC EXPLORATION OF LAYER-WISE CONVERGENCE RATE FOR DEEP NEURAL NETWORKS. In *The Eleventh International Conference on Learning Representations (ICLR)*, 2023. 6.6.1

[21] Radoslaw Martin Cichy, Aditya Khosla, Dimitrios Pantazis, Antonio Torralba, and Aude Oliva. Comparison of deep neural networks to spatio-temporal cortical dynamics of human visual object recognition reveals hierarchical correspondence. *Scientific Reports*, 6(1):27755, 9 2016. ISSN 2045-2322. doi: 10.1038/srep27755. URL `http://www.nature.com/articles/srep27755`. 5.2.2

[22] Tarin Clanuwat, Mikel Bober-Irizar, Asanobu Kitamoto, Alex Lamb, Kazuaki Yamamoto, and David Ha. Deep Learning for Classical Japanese Literature. *arXiv:1812.01718v1*, 12 2018. doi: 10.20676/00000341. URL `http://arxiv.org/abs/1812.01718http://dx.doi.org/10.20676/00000341`. 5.4.1, 6.4

[23] Adam Coates, Honglak Lee, and Andrew Y Ng. An Analysis of Single-Layer Networks in Unsupervised Feature Learning. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15, pages 215–223. JMLR Workshop and Conference Proceedings, 6 2011. URL `https://proceedings.mlr.press/v15/coates11a.html`. 6.7

[24] Adam Coates, Brody Huval, Tao Wang, David J Wu, Andrew Y Ng, and Bryan Catanzaro. Deep learning with COTS HPC systems. In *International Conference on Machine Learning*, pages 1337–1345. PMLR, 2013. URL `http://proceedings.mlr.press/v28/coates13.pdf`. 3, 3.10

[25] Pierre Courrieu. Fast Computation of Moore-Penrose Inverse Matrices. *Neural Information Processing -Letters and Reviews*, 8(2), 2005. URL `https://arxiv.org/pdf/0804.4809.pdf`. 3

[26] Luke N. Darlow, Elliot J. Crowley, Antreas Antoniou, and Amos J. Storkey. CINIC-10 is not ImageNet or CIFAR-10. *arXiv:1407.7906*, 10 2018. URL `https://arxiv.org/abs/1810.03505v1`. 1.1.1, 6.8.1

[27] Jeffrey Dean, Greg S Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V Le, Mark Z Mao, Marc Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y Ng. Large Scale Distributed Deep Networks. *NIPS 2012: Neural Information Processing Systems*, pages 1–11, 2012. 3

[28] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009. doi: 10.1109/CVPR.2009.5206848. 1.1.1, 6.8.4

[29] Michal Derezinski, Feynman T. Liang, and Michael W. Mahoney. Exact expressions for double descent and implicit regularization via surrogate random design. *Advances in Neural Information Processing Systems*, 33:5152–5164, 2020. 6.6

[30] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An Image is Worth 16x16 Words: Transformers

for Image Recognition at Scale. *ICLR 2021 - 9th International Conference on Learning Representations*, 10 2020. URL `https://arxiv.org/abs/2010.11929v2`. 1.1.1

[31] Mark Everingham, Luc Van Gool, Christopher K. I. Williams, John Winn, and Andrew Zisserman. The Pascal Visual Object Classes (VOC) Challenge. *International Journal of Computer Vision*, 88(2):303–338, 6 2010. ISSN 0920-5691. doi: 10.1007/s11263-009-0275-4. URL `http://link.springer.com/10.1007/s11263-009-0275-4`. 4.9

[32] Scott E Fahlman and Christian Lebiere. The Cascade-Correlation Learning Architecture. *Advances in neural information processing systems*, pages 524–532, 1990. ISSN 10459227. doi: 10.1190/1.1821929. URL `http://papers.nips.cc/paper/207-the-cascade-correlation-learning-architecture.pdf`. 5.5.5

[33] Scott E Fahlman and Christian Lebiere. The Cascade-Correlation Learning Architecture. *Advances in neural information processing systems*, pages 524–532, 1990. ISSN 10459227. doi: 10.1190/1.1821929. URL `http://papers.nips.cc/paper/207-the-cascade-correlation-learning-architecture.pdf`. 5.5.5

[34] Network Field Zhiping Song, Jun Zhang, Yue Fang, Jiasheng Wang, Jeongwoo Lee, Chang Liu, al , Mehmet Demirtas, James Halverson, Anindita Maiti, Matthew D Schwartz, and Keegan Stoner. Neural network field theories: non-Gaussianity, actions, and locality. *Machine Learning: Science and Technology*, 5(1):015002, 1 2024. ISSN 2632-2153. doi: 10.1088/2632-2153/AD17D3. URL `https://iopscience.iop.org/article/10.1088/2632-2153/ad17d3https://iopscience.iop.org/article/10.1088/2632-2153/ad17d3/meta`. 5.3

[35] Kunihiko Fukushima. Cognitron: A self-organizing multilayered neural network. *Biological Cybernetics*, 20(3-4):121–136, 9 1975. ISSN 03401200. doi: 10.1007/BF00342633/METRICS. URL `https://link.springer.com/article/10.1007/BF00342633`. 5.1

[36] Kunihiko Fukushima. Biological Cybernetics Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position. Technical report, 1980. URL `https://www.rctn.org/bruno/public/papers/Fukushima1980.pdf`. 5.1, 5.2.1

[37] Daniel Gissin, Shai Shalev-Shwartz, and Amit Daniely. The Implicit Bias of Depth: How Incremental Learning Drives Generalization. *8th International Conference on Learning Representations, ICLR*, 9 2020. URL `https://arxiv.org/abs/1909.12051v2`. 6.6

[38] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS)*, 9:249–256, 2010. URL `http://machinelearning.wustl.edu/mlpapers/paper_files/AISTATS2010_GlorotB10.pdf`. 4.1

[39] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT Press, 2016.

ISBN 9780262035613. URL http://www.deeplearningbook.org. 4, 4.10, 5.2.1

[40] Suriya Gunasekar, Blake E. Woodworth, Srinadh Bhojanapalli, Behnam Neyshabur, and Nati Srebro. Implicit Regularization in Matrix Factorization. *Advances in Neural Information Processing Systems*, 30, 2017. 6.6

[41] Ping Guo and Michael R. Lyu. A pseudoinverse learning algorithm for feedforward neural networks with stacked generalization applications to software reliability growth data. *Neurocomputing*, 56(1-4):101–121, 1 2004. ISSN 0925-2312. doi: 10.1016/S0925-2312(03)00385-0. 7.4

[42] James Halverson. Building Quantum Field Theories Out of Neurons. *arXiv:2112.04527*, 12 2021. URL https://arxiv.org/abs/2112.04527v1. 5.3

[43] Dongyoon Han, Jiwhan Kim, and Junmo Kim. Deep pyramidal residual networks. In *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, volume 2017-Janua, pages 6307–6315, 10 2017. ISBN 9781538604571. doi: 10.1109/CVPR.2017.668. URL http://arxiv.org/abs/1610.02915. 4.12

[44] Bharath Hariharan, Pablo Arbelaez, Lubomir Bourdev, Subhransu Maji, and Jitendra Malik. Semantic contours from inverse detectors. In *2011 International Conference on Computer Vision*, pages 991–998. IEEE, 11 2011. ISBN 978-1-4577-1102-2. doi: 10.1109/ICCV.2011.6126343. URL http://ieeexplore.ieee.org/document/6126343/. 4.13

[45] Bharath Hariharan, Pablo Arbelaez, Ross Girshick, and Jitendra Malik. Hypercolumns for Object Segmentation and Fine-Grained Localization. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 447–456, 2015. URL https://www.cv-foundation.org/openaccess/content_cvpr_2015/html/Hariharan_Hypercolumns_for_Object_2015_CVPR_paper.html. 4.13

[46] Kais Hariz, Hachem Kadri, Stephane Ayache, Maher Moakher, and Thierry Artieres. Implicit Regularization with Polynomial Growth in Deep Tensor Factorization. In *Proceedings of the 39th International Conference on Machine Learning*, pages 8484–8501. PMLR, 6 2022. URL https://proceedings.mlr.press/v162/hariz22a.html. 6.6

[47] Aram W. Harrow, Avinatan Hassidim, and Seth Lloyd. Quantum Algorithm for Linear Systems of Equations. *Physical Review Letters*, 103(15):150502, 10 2009. ISSN 00319007. doi: 10.1103/PhysRevLett.103.150502. URL https://journals.aps.org/prl/abstract/10.1103/PhysRevLett.103.150502. 7.1

[48] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016. URL http://image-net.org/challenges/LSVRC/2015/. 5.1

[49] Donald O. Hebb. *The Organization of Behavior: A NEUROPSYCHOLOGICAL THEORY*. John Wiley & Sons, Inc., 1949. ISBN 9780471367277. URL https://pure.mpg.

de/rest/items/item_2346268/component/file_2346267/content. 5.2.1

[50] Jeanny Herault, Christian Jutten, and Bernard Ans. Detection de grandeurs primitives dans un message composite par une architecture de calcul neuromimetique en apprentissage non supervise. In *Proceedings of GRETSI*, pages 1017–1020, 1985. URL https://cir.nii.ac.jp/crid/1572261549592583040. 7.10.1

[51] T M Heskes and B Kappen. On-line learning processes in artificial neural networks. In *Mathematical approaches to neural networks*, page 382. North-Holland, 1993. ISBN 9780080887395. 3.10

[52] G E Hinton and R R Salakhutdinov. Reducing the Dimensionality of Data with Neural Networks. *Science*, 313(5786):504–507, 7 2006. doi: 10.1126/science.1127647. URL http://www.ncbi.nlm.nih.gov/pubmed/16873662http://www.ncbi.nlm.nih.gov/pubmed/16873662. 6.2.1

[53] Geoffrey Hinton and Google Brain. The Forward-Forward Algorithm: Some Preliminary Investigations. *arXiv:2212.13345*, 12 2022. URL https://arxiv.org/abs/2212.13345v1. 6.3

[54] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A Fast Learning Algorithm for Deep Belief Nets. *Neural Computation*, 18(7):1527–1554, 7 2006. ISSN 0899-7667. doi: 10.1162/neco.2006.18.7.1527. URL http://www.mitpressjournals.org/doi/10.1162/neco.2006.18.7.1527. 6.3.1

[55] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 7 2012. URL http://arxiv.org/abs/1207.0580. 3.10, 4.3

[56] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 7 2012. URL https://arxiv.org/abs/1207.0580v1. 5.1, 6.6.1

[57] Sepp Hochreiter. *Untersuchungen zu dynamischen neuronalen Netzen*. PhD thesis, Diploma thesis, Institut für Informatik, lehrstuhl Prof. Brauer, Technische Uiversität München, 1991. 4.10, 6.1, 8.3.1

[58] Sepp Hochreiter. The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 06(02):107–116, 1998. ISSN 0218-4885. doi: 10.1142/S0218488598000094. URL http://www.bioinf.jku.at/publications/older/2304.pdfhttp://www.worldscientific.com/doi/abs/10.1142/S0218488598000094. 4.10

[59] Elad Hoffer, Itay Hubara, and Daniel Soudry. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. *Advances in Neural Information Processing Systems*, 30, 2017. 7.10

[60] Furong Huang, Jordan Ash, John Langford, and Robert Schapire. Learning Deep ResNet Blocks Sequentially using Boosting Theory. In *International Conference on Machine Learning*, pages 2058–2067, 2018. URL `https://arxiv.org/pdf/1706.04964.pdf`. 6.2.1, 6.3.1

[61] Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Q. Weinberger. Deep networks with stochastic depth. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9908 LNCS:646–661, 2016. ISSN 16113349. doi: 10.1007/978-3-319-46493-0{_}39/FIGURES/8. URL `https://link.springer.com/chapter/10.1007/978-3-319-46493-0_39`. 6.3

[62] Guang-Bin Huang, Qin-Yu Zhu, and Chee-Kheong Siew. Extreme Learning Machine: A New Learning Scheme of Feedforward Neural Networks. In *Proceedings of the International Joint Conference on Neural Networks*, volume 2 of *2004 IEEE International Joint Conference on Neural Networks - Proceedings*, pages 985–990, 2004. doi: 10.1109/IJCNN.2004.1380068. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?&amp;arnumber=1380068://www.scopus.com/inward/record.url?eid=2-s2.0-10944272650&partnerID=40&md5=29f3ecc4e8d675517e578425206c17ea`. (document), 7.4, 1, 7.2, 7.15

[63] Guang-Bin Huang, Qin-Yu Zhu, and Chee-Kheong Siew. Extreme learning machine: Theory and applications. *Neurocomputing*, 70(1-3):489–501, 2006. ISSN 09252312. doi: 10.1016/j.neucom.2005.12.126. URL `http://www.sciencedirect.com/science/article/pii/S0925231206000385`. (document), 7.4, 1, 7.2

[64] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, Hyouk Joong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. *Advances in Neural Information Processing Systems*, 32, 11 2019. ISSN 10495258. URL `https://arxiv.org/abs/1811.06965v5`. (document), 6.8.3, 6.6

[65] D. H. Hubel and T. N. Wiesel. Receptive fields of single neurones in the cat's striate cortex. *The Journal of Physiology*, 148(3):574–591, 10 1959. ISSN 00223751. doi: 10.1113/jphysiol.1959.sp006308. URL `http://doi.wiley.com/10.1113/jphysiol.1959.sp006308`. 5.2.1

[66] D. H. Hubel and T. N. Wiesel. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of Physiology*, 160(1):106–154, 1 1962. ISSN 00223751. doi: 10.1113/jphysiol.1962.sp006837. URL `http://doi.wiley.com/10.1113/jphysiol.1962.sp006837`. 5.2.1

[67] D. H. Hubel and T. N. Wiesel. Receptive fields and functional architecture of monkey striate cortex. *The Journal of Physiology*, 195(1):215–243, 3 1968. ISSN 00223751. doi: 10.1113/jphysiol.1968.sp008455. URL `http://doi.wiley.com/10.1113/jphysiol.1968.sp008455`. 5.2.1

[68] D A Huffman. A Method for the Construction of Minimum-Redundancy Codes. In *Pro-*

ceedings of the IRE, pages 1098–1101, 1952. 1.2.1, 3

[69] Minyoung Huh, Hossein Mobahi, Richard Zhang, Brian Cheung, Pulkit Agrawal, and Phillip Isola. The Low-Rank Simplicity Bias in Deep Networks. *arXiv:2103.10427v1*, 2021. 6.6

[70] Ching hsiang Hung and T. L. Markham. The Moore-Penrose inverse of a partitioned matrix M=(ADBC). *Linear Algebra and its Applications*, 11(1):73–86, 1 1975. ISSN 0024-3795. doi: 10.1016/0024-3795(75)90118-4. 3

[71] Yuji Igarashi, Katsumi Itoh, Jan M Pawlowski, James Halverson, Anindita Maiti, and Keegan Stoner. Neural networks and quantum field theory. *Machine Learning: Science and Technology*, 2(3):035002, 4 2021. ISSN 2632-2153. doi: 10.1088/2632-2153/ABECA3. URL `https://iopscience.iop.org/article/10.1088/2632-2153/abeca3https://iopscience.iop.org/article/10.1088/2632-2153/abeca3/meta`. 5.3

[72] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arXiv:1502.03167*, pages 1–11, 2015. URL `http://arxiv.org/abs/1502.03167`. 4.3, 5.1

[73] Jörn Henrik Jacobsen, Arnold Smeulders, and Edouard Oyallon. i-RevNet: Deep Invertible Networks. *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*, 2 2018. URL `https://arxiv.org/abs/1802.07088v1`. 5.3.3, 5.3.5

[74] Arthur Jacot. Implicit Bias of Large Depth Networks: a Notion of Rank for Nonlinear Functions. In *The Eleventh International Conference on Learning Representations, ICLR 2023*, 9 2023. URL `https://arxiv.org/abs/2209.15055v4`. 6.6

[75] Max Jaderberg, Wojciech Marian Czarnecki, Simon Osindero, Oriol Vinyals, Alex Graves, David Silver, and Koray Kavukcuoglu. Decoupled Neural Interfaces using Synthetic Gradients. In *Proceedings of the 34th International Conference on Machine Learning*, pages 1627–1635. PMLR, 7 2017. URL `https://proceedings.mlr.press/v70/jaderberg17a.html`. 6.3

[76] Kevin Jarrett, Koray Kavukcuoglu, Marc' Aurelio Ranzato, and Yann LeCun. What is the best multi-stage architecture for object recognition? In *2009 IEEE 12th International Conference on Computer Vision*, pages 2146–2153. IEEE, 9 2009. ISBN 978-1-4244-4420-5. doi: 10.1109/ICCV.2009.5459469. URL `http://ieeexplore.ieee.org/document/5459469/`. 4.1

[77] Angela H. Jiang, Daniel L. K. Wong, Giulio Zhou, David G. Andersen, Jeffrey Dean, Gregory R. Ganger, Gauri Joshi, Michael Kaminksy, Michael Kozuch, Zachary C. Lipton, and Padmanabhan Pillai. Accelerating Deep Learning by Focusing on the Biggest Losers. *arXiv:1910.00762*, 10 2019. URL `https://arxiv.org/abs/1910.00762v1`. 6.8.4

[78] N Kalouptsidis, A Pouliakis, and N Kalouptsidis. DIVIDE AND CONQUER ALGORITHMS FOR CONSTRUCTING NEURAL NETWORKS ARCHITECTURES. In *9th European Signal Processing Conference (EUSIPCO 1998)*, pages 1–4, 1998. URL

https://ieeexplore.ieee.org/abstract/document/7089934/. 6.3

[79] Christian Keck, Cristina Savin, Jörg Lücke, AJ King, and H Ros. Feedforward Inhibition and Synaptic Scaling – Two Sides of the Same Coin? *PLoS Computational Biology*, 8(3):e1002432, 3 2012. ISSN 1553-7358. doi: 10.1371/journal.pcbi.1002432. URL https://doi.org/10.1371/journal.pcbi.1002432. 4

[80] Nitish Shirish Keskar, Jorge Nocedal, Ping Tak Peter Tang, Dheevatsa Mudigere, and Mikhail Smelyanskiy. On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*, 9 2016. URL https://arxiv.org/abs/1609.04836v2. 7.10

[81] Diederik P. Kingma and Jimmy Lei Ba. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*. International Conference on Learning Representations, ICLR, 12 2014. URL https://arxiv.org/abs/1412.6980v9. 1.1.1

[82] Alex Krizhevsky. Learning Multiple Layers of Features from Tiny Images. Technical report, University of Toronto, Toronto, CA, 2009. URL https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf. 4.5, 4.9, 6.4, 6.7

[83] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances In Neural Information Processing Systems*, pages 1–9, 2012. URL https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdfhttp://papers.nips.cc/paper/4824-imagenet-classification-with-deep-4.1, 6.3.1

[84] Masayoshi Kubo, Ryotaro Banno, Hidetaka Manabe, and Masataka Minoji. Implicit Regularization in Over-parameterized Neural Networks. *arXiv:1903.01997*, 3 2019. URL https://arxiv.org/abs/1903.01997v1. 6.6

[85] H. J. Landau. Necessary density conditions for sampling and interpolation of certain entire functions. *Acta Mathematica*, 117(1):37–52, 7 1967. ISSN 00015962. doi: 10.1007/BF02395039. 8.3.1

[86] Kevin J Lang and Michael J Witbrock. Learning to Tell Two Spirals Apart. In *Proceedings of the 1988 Connectionist Models Summer School*, 1988. 7.6

[87] Y Lecun. *Modèles connexionnistes de l'apprentissage (connectionist learning models)*. PhD thesis, Universitè de Paris VI, 1987. URL https://nyu.pure.elsevier.com/en/publications/phd-thesis-modeles-connexionnistes-de-lapprentissage-connectionis. 6.3

[88] Y LeCun, L Bottou, Y Bengio, and P Haffner. Gradient Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 1998. ISSN 00189219. doi: 10.1109/5.726791. 3.1, 4.3, 6.4

[89] Yann LeCun. Learning Process in an Asymmetric Threshold Network. *Disor-*

dered Systems and Biological Organization, pages 233–240, 1986. doi: 10.1007/978-3-642-82657-3{_}24. URL `https://link.springer.com/chapter/10.1007/978-3-642-82657-3_24`. 6.3

[90] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2323, 1998. ISSN 00189219. doi: 10.1109/5.726791. 1.1.1, 5.4.1, 7.4, 7.9

[91] Yann A. LeCun, Léon Bottou, Genevieve B. Orr, and Klaus Robert Müller. Efficient BackProp. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Alfred Kosba, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, and Gerhard Weikum, editors, *Neural Networks: Tricks of the Trade*, chapter 1, pages 9–48. Springer, Berlin Heidelberg, 2 edition, 2012. ISBN 978-3-642-35289-8. doi: 10.1007/978-3-642-35289-8. 4, 4.1, 4.5

[92] Dong Hyun Lee, Saizheng Zhang, Asja Fischer, and Yoshua Bengio. Difference target propagation. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9284, pages 498–515, 2015. ISBN 9783319235271. doi: 10.1007/978-3-319-23528-8{_}31. 6.3, 6.3.1

[93] Yuanzhi Li, Colin Wei, and Tengyu Ma. Towards Explaining the Regularization Effect of Initial Large Learning Rate in Training Neural Networks. *Advances in Neural Information Processing Systems*, 32, 7 2019. ISSN 10495258. URL `https://arxiv.org/abs/1907.04595v2`. 8.4

[94] Timothy P Lillicrap, Daniel Cownden, Douglas B Tweed, and Colin J Akerman. Random synaptic feedback weights support error backpropagation for deep learning. *Nature communications*, 7:13276, 11 2016. ISSN 2041-1723. doi: 10.1038/ncomms13276. URL `http://www.ncbi.nlm.nih.gov/pubmed/27824044http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC5105169`. 6.3

[95] Chi-Heng Lin, Chiraag Kaushik, Eva L. Dyer, and Vidya Muthukumar. The good, the bad and the ugly sides of data augmentation: An implicit spectral regularization perspective. *Journal of Machine Learning Research xx (xxxx)*, pages 1–86, 10 2022. URL `https://arxiv.org/abs/2210.05021v2`. 6.6

[96] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully Convolutional Networks for Semantic Segmentation. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3431–3440, 2015. URL `https://www.cv-foundation.org/openaccess/content_cvpr_2015/html/Long_Fully_Convolutional_Networks_2015_CVPR_paper.html`. 4.13

[97] Ilya Loshchilov and Frank Hutter. Decoupled Weight Decay Regularization. *7th International Conference on Learning Representations, ICLR 2019*, 11 2017. URL `https://arxiv.org/abs/1711.05101v3`. (document), 6.7, 6.5

[98] Cong Ma, Kaizheng Wang, Yuejie Chi, and Yuxin Chen. Implicit Regularization in Non-

convex Statistical Estimation: Gradient Descent Converges Linearly for Phase Retrieval and Matrix Completion. In *Proceedings of the 35th International Conference on Machine Learning*, pages 3345–3354. PMLR, 7 2018. URL `https://proceedings.mlr.press/v80/ma18c.html`. 6.6

[99] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. ShuffleNet V2: Practical Guidelines for Efficient CNN Architecture Design. In *Proceedings of the European Conference on Computer Vision (ECCV), 2018*, pages 116–131, 2018. 6.7

[100] Stéphane Mallat. Understanding deep convolutional networks. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 374 (2065), 4 2016. ISSN 1364503X. doi: 10.1098/RSTA.2015.0203. URL `https://royalsocietypublishing.org/doi/10.1098/rsta.2015.0203`. 5.3.3

[101] M. Manassi, B. Sayim, and M. H. Herzog. When crowding of crowding leads to uncrowding. *Journal of Vision*, 13(13):10–10, 11 2013. ISSN 1534-7362. doi: 10.1167/13.13.10. URL `http://jov.arvojournals.org/Article.aspx?doi=10.1167/13.13.10`. (document), 5.2.2, 5.1

[102] Charles H. Martin and Michael W. Mahoney. Implicit Self-Regularization in Deep Neural Networks: Evidence from Random Matrix Theory and Implications for Learning. *The Journal of Machine Learning Research*, 22(1):7479–7551, 10 2018. URL `http://arxiv.org/abs/1810.01075`. 6.6

[103] James L. McClelland and David E. Rumelhart. *Parallel Distributed Processing, Volume 2. Explorations in the Microstructure of Cognition: Psychological and Biological Models*. MIT Press, 1986. ISBN 9780262132183. 5.2.1

[104] James L. McClelland and David E. Rumelhart. *Explorations in Parallel Distributed Processing: A Handbook of Models, Programs, and Exercises*. MIT Press, 1989. ISBN 9780262631297. URL `https://mitpress.mit.edu/books/explorations-parallel-distributed-processing-macintosh-version`. 5.2.1

[105] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133, 12 1943. ISSN 0007-4985. doi: 10.1007/BF02478259. URL `http://link.springer.com/10.1007/BF02478259`. 5.2.1

[106] Sachin Mehta, Apple Mohammad, and Rastegari Apple. Separable Self-attention for Mobile Vision Transformers. *arXiv:2206.02680*, 6 2022. URL `https://arxiv.org/abs/2206.02680v1`. 6.7

[107] J Mercer. Functions of Positive and Negative Type, and their Connection with the Theory of Integral Equations. *Philosophical Transactions of the Royal Society of London. Series A, Containing Papers of a Mathematical or Physical Character*, 209:415–446, 1909. ISSN 02643952. URL `http://www.jstor.org/stable/91043`. 7.4

[108] Poorya Mianjy, Raman Arora, and Rene Vidal. On the Implicit Bias of Dropout. In *Proceedings of the 35th International Conference on Machine Learning*, pages 3540–3548. PMLR, 7 2018. URL `https://proceedings.mlr.press/v80/mianjy18b.`

html. 6.6

[109] Moshe Mishali and Yonina C. Eldar. Blind multiband signal reconstruction: Compressed sensing for analog signals. *IEEE Transactions on Signal Processing*, 57(3):993–1009, 2009. ISSN 1053587X. doi: 10.1109/TSP.2009.2012791. 8.3.1

[110] Roozbeh Mottaghi, Mohammad Rastegari, Abhinav Gupta, and Ali Farhadi. “What Happens If...” Learning to Predict the Effect of Forces in Images. In *European Conference on Computer Vision*, pages 269–285. Springer, Cham, 2016. doi: 10.1007/978-3-319-46493-0{_}17. URL http://link.springer.com/10.1007/978-3-319-46493-0_17. 4.13

[111] Tasha Nagamine, Michael L. Seltzer, and Nima Mesgarani. Exploring how deep neural networks form phonemic categories. *Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH*, 2015-January:1912–1916, 2015. ISSN 19909772. doi: 10.21437/INTERSPEECH.2015-422. 5.3.3

[112] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *ICML’10 Proceedings of the 27th International Conference on International Conference on Machine Learning*, 2010. ISBN 9781605589077. URL http://machinelearning.wustl.edu/mlpapers/paper_files/icml2010_NairH10.pdf. 4.1, 5.1

[113] Arvind Neelakantan, Luke Vilnis, Quoc V Le, Ilya Sutskever, Lukasz Kaiser, Karol Kurach, Google Brain, and James Martens. Adding Gradient Noise Improves Learning for Very Deep Networks. 11 2015. URL https://arxiv.org/abs/1511.06807v1. 6.6.3

[114] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. Reading Digits in Natural Images with Unsupervised Feature Learning. In *In NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, 2011. URL http://ufldl.stanford.edu/housenumbers/. 1.1.1, 6.7

[115] Behnam Neyshabur, Nathan Srebro, Yury Makarychev, Ruslan Salakhutdinov, and Gregory Shakhnarovich. Implicit Regularization in Deep Learning. *arXiv:1709.01953*, 9 2017. URL https://arxiv.org/abs/1709.01953v2. 6.6

[116] Behnam Neyshabur, Ryota Tomioka, Ruslan Salakhutdinov, and Nathan Srebro. Geometry of Optimization and Implicit Regularization in Deep Learning. *arXivI:1705.03071*, 5 2017. URL https://arxiv.org/abs/1705.03071v1. 6.6

[117] Arild Nøkland and Lars Hiller Eidnes. Training Neural Networks with Local Error Signals. In *Proceedings of the 36th International Conference on Machine Learning*, pages 4839–4850. PMLR, 5 2019. URL https://proceedings.mlr.press/v97/nokland19a.html. 6.3.1

[118] OpenAI, :, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mo Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-

Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan

Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. GPT-4 Technical Report. *arXiv:2303.08774*, 3 2023. URL `https://arxiv.org/abs/2303.08774v4`. 1.1.1, 6.8.4

[119] Edouard Oyallon. Building a Regular Decision Boundary With Deep Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5106–5114, 2017. URL `https://github.com/edouardoyallon/`. 5.3.3, 5.4

[120] Youngtae Park. A comparison of neural net classifiers and linear tree classifiers: Their similarities and differences. *Pattern Recognition*, 27(11):1493–1503, 11 1994. ISSN 00313203. doi: 10.1016/0031-3203(94)90127-9. URL `https://www.sciencedirect.com/science/article/pii/0031320394901279`. 6.3

[121] Nasim Rahaman, Aristide Baratin, Devansh Arpit, Felix Draxler, Min Lin, Fred A Hamprecht, Yoshua Bengio, and Aaron Courville. On the Spectral Bias of Neural Networks. In *Proceedings of the 36th International Conference on Machine Learning, Long Beach, California, PMLR 97*, 2019. URL `https://github.com/nasimrahaman/SpectralBias`. 6.6.1

[122] Marc'aurelio Ranzato, Christopher Poultney, Sumit Chopra, and Yann LeCun. Efficient Learning of Sparse Representations with an Energy-Based Model. In B Schölkopf, J C Platt, and T Hoffman, editors, *Advances in Neural Information Processing Systems 19*, pages 1137–1144. MIT Press, 2007. URL `http://papers.nips.cc/paper/3112-efficient-learning-of-sparse-representations-with-an-energy-basec` `pdf`. 6.3.1

[123] C D C Reeve. *Republic (Hackett Classics)*. Hackett Publishing Company, Inc., 2 edition, 11 1995. ISBN 0872201368. 7.2

[124] Frank Rosenblatt. The Perceptron: A Perceiving and Recognizing Automaton. Technical report, Cornell Aeronautical Laboratory, Buffalo, N. Y., 1957. URL `https://blogs.umass.edu/brain-wars/files/2016/03/rosenblatt-1957.pdf`. 5.2.1

[125] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386–408, 1958. ISSN 0033-295X. doi: 10.1037/h0042519. 5.2.1

[126] David E. Rumelhart and James L. McClelland. *Parallel Distributed Processing, Voume 1. Explorations in the Microstructure of Cognition*. MIT Press, 1986. ISBN 9780262181204. 5.2.1

[127] Davide E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–538, 1986. URL `http://lia.disi.unibo.it/Courses/SistInt/articoli/nnet1.pdf`. 1.1.1, 5.2.3, 7.1

[128] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4510–4520, 6 2018. 6.7

[129] Robert E. Schapire. The strength of weak learnability. *Machine Learning*, 5(2):197–227, 6 1990. ISSN 0885-6125. doi: 10.1007/BF00116037. URL `http://link.springer.com/10.1007/BF00116037`. 6.3.1

[130] Robert E. Schapire and Yoav. Freund. *Boosting : Foundations and Algorithms*. The MIT Press, 2012. ISBN 9780262017183. 6.3.1

[131] Ishwar K. Sethi. Entropy Nets: From Decision Trees to Neural Networks. *Proceedings of the IEEE*, 78(10):1605–1613, 1990. ISSN 15582256. doi: 10.1109/5.58346. URL `http://ieeexplore.ieee.org/document/58346/`. 6.3

[132] C. E. Shannon. A Mathematical Theory of Communication. *Bell System Technical Journal*, 27(3):379–423, 7 1948. doi: 10.1002/j.1538-7305.1948.tb01338.x. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6773024`. 3

[133] Ravid Shwartz-Ziv and Naftali Tishby. Opening the Black Box of Deep Neural Networks via Information. *arXiv preprint arXiv:1703.00810*, 3 2017. URL `http://arxiv.org/abs/1703.00810`. 5.3.4, 5.3.5, 5.5.4, 6.3.1, 8.3.1

[134] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv preprint arXiv:1409.1556*, 9 2014. URL `http://arxiv.org/abs/1409.1556`. 4.11, 4.13

[135] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *3rd International Conference on Learning Representations, ICLR*. International Conference on Learning Representations, ICLR, 9 2015. URL `https://arxiv.org/abs/1409.1556v6`. 5.4.1, 6.4, 6.6.2, 6.7

[136] Samuel L. Smith, Benoit Dherin, David G.T. Barrett, and Soham De. On the Origin of Implicit Regularization in Stochastic Gradient Descent. *ICLR 2021 - 9th International Conference on Learning Representations*, 1 2021. URL `https://arxiv.org/abs/2101.12176v1`. 6.6

[137] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. STRIVING FOR SIMPLICITY: THE ALL CONVOLUTIONAL NET. *arXiv preprint arXiv:1412.6806*, 2014. 4.10

[138] Vukašin Stanojević, Lev Kazakovtsev, Predrag S. Stanimirović, Natalya Rezova, and Guzel Shkaberina. Calculating the Moore–Penrose Generalized Inverse on Massively Parallel Systems. *Algorithms 2022, Vol. 15, Page 348*, 15(10):348, 9 2022. ISSN 1999-4893. doi: 10.3390/A15100348. URL `https://www.mdpi.com/1999-4893/15/10/348/htmhttps://www.mdpi.com/1999-4893/15/10/348`. 3

[139] C Szegedy, W Liu, Y Jia, and P Sermanet. Going deeper with convolutions. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015. URL `http://www.cv-foundation.org/openaccess/content_cvpr_2015/html/Szegedy_Going_Deeper_With_2015_CVPR_paper.html`. 6.3.1

[140] Mingxing Tan and Quoc V Le. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In *Proceedings of the 36th International Conference on Machine*

Learning, PMLR, pages 6105–6114. PMLR, 5 2019. URL https://proceedings.mlr.press/v97/tan19a.html. 6.6.2, 6.7

[141] Mingxing Tan and Quoc V Le. EfficientNetV2: Smaller Models and Faster Training. In *Proceedings of the 38th International Conference on Machine Learning, PMLR*, pages 10096–10106. PMLR, 7 2021. URL https://proceedings.mlr.press/v139/tan21a.html. 6.6.2, 6.7, 6.8, 6.8.4, 6.8.4

[142] Nadav Timor, Gal Vardi, Ohad Shamir OHADSHAMIR, Shipra Agrawal, and Francesco Orabona. Implicit Regularization Towards Rank Minimization in ReLU Networks. In *Proceedings of The 34th International Conference on Algorithmic Learning Theory, PMLR*, volume 201, pages 1429–1459. PMLR, 2 2023. URL https://proceedings.mlr.press/v201/timor23a.html. 6.6

[143] Naftali Tishby and Noga Zaslavsky. Deep Learning and the Information Bottleneck Principle. *Ieee*, pages 1–5, 2015. doi: 10.1109/ITW.2015.7133169. URL https://arxiv.org/pdf/1503.02406.pdf. 5.3, 5.3.4, 5.3.5, 5.3.5, 5.5.4, 6.3.1, 8.3.1

[144] Naftali Tishby, Fernando C Pereira, and William Bialek. The information bottleneck method. *arXiv preprint arXiv:physics/0004057*, 2000. URL https://arxiv.org/pdf/physics/0004057.pdf. 6.3.1

[145] Gul Varol, Javier Romero, Xavier Martin, Naureen Mahmood, Michael J. Black, Ivan Laptev, and Cordelia Schmid. Learning From Synthetic Humans. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 109–117, 2017. URL http://openaccess.thecvf.com/content_cvpr_2017/html/Varol_Learning_From_Synthetic_CVPR_2017_paper.html. 4.13

[146] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. In *Advances in Neural Information Processing Systems*, volume 2017-December, pages 5999–6009. Neural information processing systems foundation, 6 2017. ISBN 1706.03762v7. URL https://arxiv.org/abs/1706.03762v7. 1.1.1, 5.1

[147] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning - ICML '08*, pages 1096–1103, 2008. ISBN 9781605582054. doi: 10.1145/1390156.1390294. URL http://delivery.acm.org/10.1145/1400000/1390294/p1096-vincent.pdf?ip=128.237.145.228&id=1390294&acc=ACTIVESERVICE&key=A792924B58C015C1.5A12BE0369099858.4D4702B0C3E38B35.4D4702B0C3E38B35&CFID=834184392&CFTOKEN=18564436&__acm__=1511758126_48c99d3cafcf0476f. 3.10

[148] L Wan, M Zeiler, and S Zhang. Regularization of Neural Networks Using DropConnect. In *Proceedings of the 30th International Conference on Machine Learning*, pages 1058–166, Atlanta, Georgia, 2013. PMLR. URL http://machinelearning.wustl.edu/mlpapers/papers/icml2013_wan13. 3.10

[149] Xiaolong Wang, David Fouhey, and Abhinav Gupta. Designing Deep Net-

works for Surface Normal Estimation. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 539–547, 2015. URL http://openaccess.thecvf.com/content_cvpr_2015/html/Wang_Designing_Deep_Networks_2015_CVPR_paper.html. 4.13

[150] Yulin Wang, Zanlin Ni, Shiji Song, Le Yang, and Gao Huang. Revisiting Locally Supervised Learning: an Alternative to End-to-end Training. *ICLR 2021 - 9th International Conference on Learning Representations*, 1 2021. URL https://arxiv.org/abs/2101.10832v1. 5.3.3, 6.3.1

[151] Ross Wightman. PyTorch Image Models. https://github.com/rwightman/pytorch-image-models, 2019. 6.8.4

[152] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. *arXiv:1708.07747*, 8 2017. doi: 10.48550/arxiv.1708.07747. URL https://arxiv.org/abs/1708.07747v2. 5.4.1, 6.4

[153] Yuwen Xiong, Mengye Ren, and Raquel Urtasun. LoCo: Local Contrastive Representation Learning. *Advances in Neural Information Processing Systems*, 33:11142–11153, 2020. 6.3.1

[154] Zhi-Qin John Xu, Yaoyu Zhang, Tao Luo, Yanyang Xiao, and Zheng Ma. Frequency Principle: Fourier Analysis Sheds Light on Deep Neural Networks. *Communications in Computational Physics*, 28(5):1746–1767, 1 2019. doi: 10.4208/cicp.OA-2020-0085. URL http://arxiv.org/abs/1901.06523http://dx.doi.org/10.4208/cicp.OA-2020-0085. 6.6.1

[155] Matthew D. Zeiler and Rob Fergus. Visualizing and Understanding Convolutional Networks. In D Fleet, T Pajdla, B Schiele, and T Tuytelaars, editors, *ECCV 2014. Lecture Notes in Computer Science, vol 8689*, pages 818–833. Springer, Cham, 2014. doi: 10.1007/978-3-319-10590-1{_}53. URL http://link.springer.com/10.1007/978-3-319-10590-1_53. (document), 5.2.2, 5.3, 5.3.1, 5.3.3

[156] Zhongwang Zhang and Zhi-Qin John Xu. Implicit regularization of dropout. *arXiv:2207.05952*, 7 2022. URL https://arxiv.org/abs/2207.05952v2. 6.6

[157] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 9 1978. ISSN 0018-9448. doi: 10.1109/TIT.1978.1055934. URL http://ieeexplore.ieee.org/document/1055934/. 3.1