DISSERTATION

LOCAL LINEARITY OF RELU NEURAL NETWORKS

Submitted by

Ben Sattelberg

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Spring 2026

Committee:

Advisor: Bruce Draper

Ewan Davies
Michael Kirby
Chris Peterson

ABSTRACT

LOCAL LINEARITY OF RELU NEURAL NETWORKS

Despite their impressive practical results and broad usage, ReLU neural networks are still widely considered to be black boxes. Modern networks are complex, high-dimensional, nonlinear functions frequently applied to problems where other methods perform poorly. Building an understanding of their behavior is, therefore, both difficult and necessary. Significant progress has been made towards this, but results remain limited in comparison to empirical success.

This research proposes two primary methodologies to add to current understanding: formal analysis of network behavior on simple problems and investigation of the piecewise linear behavior induced by the ReLU activation function. Network behavior on simple problems, such as approximation of Boolean functions or minimal n-dimensional classification, has reduced complexity that makes answering questions about optimal or minimal network size feasible. The behavior of networks on these restricted domains provides an interpretable method for determining the effects of network size on representational capacity and training success rate in a way that is still applicable to more complex problems of interest.

The piecewise linear nature of the ReLU function also allows for simplified local evaluation of network behavior. ReLU neural networks form convex polytopes in the input space with the network behaving as a simple linear mapping from input to output within each of these polytopes. By exploiting this structure, it is possible to examine locally linear behavior in aggregate to construct metrics for network complexity and similarity. This analysis is able to avoid traditional difficulties stemming from symmetries and architectural differences by largely maintaining the interior of the network as a black box.

ACKNOWLEDGEMENTS

I received a great deal of support and assistance throughout the process of creating this document. I would first like to thank Ross Beveridge for his deep interest and enthusiasm for the insights gained over the course of the many experiments that led to the results here. His perspectives have been incredibly valuable for helping me identify elements I would not have considered and pushing me to further investigate my initial results. His recommendations to work with the members of my committee, and especially his support in developing a working group to identify the mathematical foundations of neural networks, helped expose me to a variety of paths that were crucial to this work.

Bruce Draper and Michael Kirby's support when I came back after my pause in working on my dissertation was invaluable for allowing me to finish. I deeply appreciate their, and the other members of my committee, work to push me to succeed.

I appreciate Ross Beveridge, Renzo Cavalieri, Ewan Davies, Bruce Draper, Michael Kirby, and Chris Peterson for their support in the many meetings we have had to discuss the research in this document. Their feedback has been critical for my progress through this research, and I am grateful for the time they have taken to help me attain better understandings.

I would also like to thank David White for his invaluable support during my time at CSU. Our discussions about the nature of neural networks were deeply important for building my understanding of the mathematical and philosophical elements of their behavior.

I would also like to thank Elliot Forney for the creation of the LaTeX template used for this document.

TABLE OF CONTENTS

Chapter 1

Introduction

Despite their broad usage, the behavior of neural networks is still widely considered to be a black box. Significant and important work has been done to build an understanding of this behavior, but results remain limited in comparison to the practical success of these networks. This treatment of networks as black boxes has led to widespread concerns with their usage — it has become clear that in usage cases such as facial recognition, hiring, and language processing, network behavior reflects the biases and limitations of their creators and datasets [1–3]. Building a deeper understanding of their behavior so that decisions can be examined and these biases identified and reduced is of the utmost importance as neural networks continue to grow in usage.

A simple way to investigate neural network behavior is to build understanding on small problems. Neural networks are typically used on complex, high-dimensional problems where investigation is complicated by the dynamics of high dimensions, sampling error, and difficulty in representing the problem [4]. By restricting the problems and networks we consider to the simplest possible, we can draw meaningful conclusions about network dynamics. For Boolean functions and classifying simple regions in low-dimensional space, we can investigate minimal network sizes and the relationship between network size and training success rates.

An alternative method to investigate neural network behavior is through linearization. Linearization allows us to treat a complex, non-linear function as locally linear by ignoring the effects of higher order terms in a Taylor series approximation. This analysis allows for approximate solutions and understanding of functions that we would otherwise be unable to analyze. This can significantly increase our ability to understand what a function is doing, but such analysis typically has the downside of restricting our investigation to local behavior of well-behaved differentiable functions.

Linearization naturally fits neural networks using piecewise linear functions, such as ReLU, as their nonlinearities are piecewise linear. A network using only piecewise linear functions as

nonlinearities will itself be a piecewise linear function, and so a given linearization with respect to an input is exact within a small polytope surrounding that input. In some ways this restricts our usage of linearization, as it turns a continuous problem into a discrete but combinatorially large one, but it also opens many possible options for analyzing the locally linear behavior.

For example, this local behavior can be considered in aggregate to construct metrics for the complexity of a network or to determine the similarity between two networks. The behavior of these linear coefficients and their associated polytopes can be evaluated as a proxy for the whole network, focusing on input regions of interest and building locally linear models. Previously, such work has focused on enumerating these regions to study network complexity and visualizing these regions to determine how space is partitioned. The goal of this work is to extend the investigation of linear regions to analyze aggregate linearized network behavior.

1.1 Overview

This dissertation starts with a partial overview of historical work relating to theoretical analysis of neural networks, with the focus for modern results primarily on representational capacity, theoretical training behavior, and network complexity. From there, an investigation of network behavior on Boolean functions is discussed. This investigation focuses on the representational capacity of networks on Boolean functions, how closely trained networks are able to match minimal network sizes, and to what extent trained, overparameterized networks can be pruned to minimal networks. Following that, we investigate classification of a circle versus its surrounding annulus as a way to investigate to what extent those results apply to higher dimensional behavior. We then transition to a discussion of the different ways of considering linearizations of ReLU neural networks. Low-dimensional problems are used to visualize certain equivalent representations of neural networks. This leads into visualizations of the piecewise structure of ReLU neural networks as a set of polytopes in high dimensional space. Finally, we demonstrate how linearizations can be used as a set of aggregate data and exploited to compute complexity metrics of a neural network or to compare different networks.

1.1.1 Historical Context

In Chapter 2, we provide an overview of previous work done to understand the behavior of neural networks. A historical perspective is taken to better illustrate the independence of practical success of networks from theoretical results or understandings. We start with an analysis of Perceptrons and the difficulties that arise when attempting to solve problems using linear methods [5, 6]. This leads into the work done in the 1980s, when the addition of hidden layers allowed neural networks to theoretically approximate any reasonably well-behaved function, and the practical algorithms that allowed such approximations to be constructed. We then discuss how in the 2010s, the explosion in size and shift towards ReLU activation functions for many problems has shifted theoretical analysis. Theoretical results in the past decade have broadened in their considerations significantly, so we emphasize results relevant to, and building the foundation for, our own work which focuses on representational capacity and similarity.

Much of the original work dealing with the linear regions of ReLU neural networks has focused on investigating expressivity and complexity. Results show that networks are universal approximators, that is, subject to certain mild constraints, they are able to approximate any well-behaved function to within arbitrary precision as the size of the network increases [7–11]. As foundational as these results are, they are typically not applicable to practical neural networks because they do not say anything about the expressivity of a *given* neural network without extensive analysis of the associated task's structure (and even then they do not include any information about ease of training a network of that size). As a proxy for the expressivity of networks in practice, various groups found and improved upper and lower bounds on the maximum number of linear regions that ReLU neural networks can have [12–14]. The main result of this work is that the maximum number of linear regions a network can have typically grows polynomially in the width (number of nodes per layer) and exponentially in the depth (number of layers) [14]. This partially explains the success of the trend in many modern neural networks to go deeper, such as ResNet [15].

However, empirical investigations of the number of linear regions actually achieved by many neural networks have shown different results. Untrained neural networks after initialization have

a number of linear regions that grows linearly in the number of ReLU functions along any one-dimensional subspace of the input space [16]. Furthermore, they tend to grow polynomially in the number of ReLU nodes in the network and exponentially in the dimension of the inputs to the network [17].

Other styles of analysis of these linear regions have also been undertaken. Novak *et al.* [18] considered these regions to investigate the effect of hyperparameters on input sensitivity and found that overparameterization, surprisingly, can help with generalization. They and Zhang *et al.* [19] investigated how the linear region structure can be used to predict the generalization of a network and what effect various regularization methods have on the geometry of the linear regions.

This piecewise linear structure has also allowed for the study of neural networks using other mathematical fields. Zhang *et al.* [20] showed that under certain mild assumptions, the piecewise linear structure of these neural networks means that the set of ReLU neural networks, the set of piecewise linear functions, and the set of tropical rational functions are equivalent. We do not significantly extend our results to the realm of tropical algebra, but we do take inspiration from the concept of the dual as commonly expressed in tropical algebra.

The goal of this chapter is twofold. First, we would like to demonstrate the complexities of building theoretical understanding of neural networks — even methods considered today to be somewhat "trivial" were, at the time, subjects of significant debate about their theoretical capabilities. This problem of understanding is compounded by the rapid increases in size and broadening in types of architectural methodologies used in modern day networks. Second, we would like to highlight how progress on these problems has been intertwined with the practical success of neural networks, even as the practical success has outpaced the theoretical results.

1.1.2 Boolean Functions

In Chapter 3, we investigate the behavior of neural networks when representing Boolean functions. Boolean functions have been historically relevant to neural networks since their inception, with Minsky and Papert using Boolean parity to demonstrate shortcomings of the Perceptron [6].

Boolean functions have relatively simple behavior, as their domain is only the corners of the d-dimensional hypercube, and they only have two possible output values. This simplicity allows for more sophisticated analysis than would be possible with a less restricted domain, while still providing a broad class of problems with distinct behaviors.

This work starts by building bounds and conjectures for the minimal network sizes capable of representing Boolean functions. We then demonstrate how network size affects training success rate by training multiple sizes of network across all Boolean functions of input size four and comparing their performance. As minimal networks are rarely successful, we then consider the feasibility of pruning overparameterized networks on parity to determine if the bounds are attainable by training an overparameterized network and reducing its size. In addition to this analysis, we conduct a steady-state analysis of a one-dimensional version of two parity using simplified networks to determine how parameters and initialization affect the ability of the network to train to good solutions. Our conclusions mirror results common in the Lottery Hypothesis literature, where minimal networks can find success, but it is far more common that large networks succeed as they are likely to contain a subnetwork that approximates a good minimal network [21]. However, we also find that identifying these minimal networks is difficult, and that modifying training hyperparameters provides a more effective way of training a minimal network with minimal computation.

1.1.3 Circle Versus Annulus Problem

In Chapter 4, we investigate the behavior of neural networks on circle versus annulus and hypersphere versus hypershell problems. Boolean functions, as discussed in Chapter 3 provide a structured problem where it is possible to gain a great deal of information about network behavior. However, they are fundamentally solvable by neural networks operating as if they only had a single input dimension, and so it is necessary to identify a problem where that is not possible. Classifying a closed, bounded region as separate from a surrounding region provides a method to do this. We first investigate what geometric shapes minimal neural networks are able to construct as decision boundaries for each dimension in order to determine what constraints the dimension imposes on

the construction of the hypersphere versus hypershell problem for it to be solvable by a neural network.

Once we've identified the constraints on problem construction, we determine good choices of hyperparameters for training neural networks in low dimensions. The previous two analyses combine to allow us to determine how problem difficulty changes as dimension increases. Our primary result is that unlike Boolean functions, where as the dimension increases the average problem becomes exponentially more complex to represent, as long as the distance between the n-dimensional hypersphere and hyperannulus grows as $\sqrt{n}$, the only increased difficulty in the problem comes from the exponential increase in training samples necessary to maintain sampling density as n increases.

1.1.4 Linearizations

In Chapter 5, we provide background and mathematical foundations for the structure of linearizations and simple examples of how they can be used. Different ways of considering the piecewise linear behavior of ReLU neural networks will be shown — as derivatives, as bit patterns, as piecewise linear functions, and as tropical rational functions. Those understandings will be shown on two simple problems, approximation of the sine and XOR functions.

When considering linearizations of neural networks, we discuss how the linear portion of the Taylor series expansion can be used as an approximation in a small region around a point of interest. In the general case, this approximation is not exact, but for piecewise linear functions (such as neural networks with ReLU activation functions), the approximation perfectly describes the behavior of the function in a polytope around the point of interest. This method of linearization allows for an easily computed way to sample the local behavior of practical networks.

Alternatively, we can focus on the linear regions themselves. For ReLU neural networks, the linear regions are constrained to be polytopes in the input space. Investigating the structure of the polytopes can give insight into how the network is structured and act as a simple measure of how much "complexity" the network is using in a given area.

Piecewise linear functions can also be considered as tropical rational functions — the difference between two maximums of various linear terms. This representation is potentially useful for theoretical results, but it is difficult to meaningfully compute the terms of the tropical polynomials for deep neural networks due to their complexity.

1.1.5 Polytope Structure

In Chapter 6 we visualize polytope structures of both simple and convolutional neural networks and note the equivalent formulation in tropical algebraic geometry. Animations of how the polytope structure evolves throughout training are shown for the problem of separating a circle from an annulus with different levels of network complexity. Similar polytope structures are constructed for the larger image classification datasets, although cross-sections of the space are necessary for visualization.

Visualizing the polytope structures of simple two-dimensional classification problems allows us to better build intuition into how the constraints of the neural network make themselves known in classification problems. Creating animations of how structures arising from the weights evolve during training also lets us determine how those structures are exploited by the network.

Although networks trained for image classification cannot have their polytope structure visualized in its entirety, we can create cross-sections of it and look at the individual "images" that correspond to the linear terms used for classification within a region. This allows us to both get an understanding of how complex the structures are for various network architectures and build intuition for how these structures are utilized for classification.

Sensitivity analysis provides an alternative method for investigating the behavior of networks across linear regions. It gives a primitive method of looking at what is important to the network. By constructing the directions along which network outputs vary most, we can identify features in input space that are potentially meaningful for classification. This idea has been explored in more detail in the existing saliency mapping literature, but it is still useful for gaining insight into network behavior before discussion of more complex metrics in the next chapter [22].

1.1.6 Metrics for Network Analysis

In Chapter 7 we use local linearizations to compare the behavior of networks by calculating tolerance to compression of linear coefficients and the similarity of linear coefficients between networks. These methods take sampled network behavior at points in the testing and training sets, and allow us to compare networks holistic manner over their domains.

Clustering the linear coefficients used by a network into a fixed number of values allows us to investigate how tolerant a network may be to simplification. It also allows us to determine how many linear regions are actually meaningful for classification. Most networks have vastly more linear regions than training elements, and even when clustering to reduce only the regions used for training, significant accuracy can be preserved.

Investigating the similarity between linear coefficients of different networks gives us a metric for comparing networks with different architectures or training sets. By investigating the correlation or ability to transform between the linear coefficients, we can determine to what extent the networks appear to be doing similar things on the inputs considered.

Chapter 2

History of Neural Network Understanding and Related Work

It is commonly believed that there is a dearth of meaningful theoretical results for neural networks. This is not quite true. Although there are no airtight results for how and when neural networks may train to a useful representation of data, there are still meaningful results about their behavior.

Questions in neural network theory can be grouped into five broad classes:

1. What functions can neural networks represent (i.e. can neural networks work at all)?

2. Can training methods for neural networks find close representations of these functions (i.e. do neural networks work)?

3. What training data and architectural design are necessary for a neural network to approximate these functions (i.e. when can one make neural networks work)?

4. What information is actually learned by a neural network (i.e. how do neural networks work)?

5. What properties do neural networks have that account for their exceptional success in practice (i.e. why do neural networks work)?

The third, fourth, and fifth of which are the main topics of current theoretical research.

The first question, "what functions can neural networks represent," is mostly a closed question. Many variations on universal approximation theorems have been shown for a wide variety of network architectures [23–28]. These results typically state that, given sufficiently large width or depth, a network is able to approximate any reasonably well-behaved function. They do not,

however, identify the problems a specific neural network with fixed width and depth is able to solve.

The second question, "can the current backpropagation methods find good solutions," is mostly an open theoretical question. There is extensive empirical evidence that they do, and given large enough networks it can be shown that training methodologies will converge to good solutions, but there is less information about the behavior of smaller networks [29–31].

The first part of the third question, what training data is necessary for success, is also partially an open question. In practice, neural networks are data hungry, but no strong results have been shown for what a good requirement is. There are results that there is a combinatorial explosion in the complexity attainable in neural networks, suggesting that they may have more complexity than the datasets they are approximating [32–34]. The second part, what architectural designs should be used, is also a partially open question. There are a number of results showing depth likely improves expressivity and loss surfaces [23, 29, 35] and a number of theoretical motivations for structures such as convolutional layers that have not necessarily been analyzed as closely [36, 37]. However, beyond those theoretical motivations, there are limited results about what the minimal architectures able to solve a problem are.

The fourth and fifth questions, what do neural networks learn and why do they work so well, are areas of current active research. Empirical (and some theoretical) evidence suggests that networks learn close approximations to a wide class of functions. However, the ways in which the networks represent or learn that approximation are unclear.

This chapter discusses the history of theoretical results related to neural networks, from the beginning of what is typically considered a neural network with the Perceptron to modern results attempting to answer some of these open questions in neural network theory. Where possible, intuitive proofs of theorems discussed will be provided.

2.1 Perceptrons (1950s-1970s)

Perceptrons are the precursor to modern neural networks. They classify a set of inputs into positive or negative classes based on the sign of a single numerical output. When they were created, the popular view of them was somewhat romanticized — there was a strong belief that they could be used to solve any kind of problem [6]. This is technically true, but it requires a representation of the data that effectively is the solution to the problem as Perceptrons are fundamentally linear in their inputs.

2.1.1 Rosenblatt's Perceptrons

Frank Rosenblatt created perceptrons based on the understanding of animal neurons during the 1950s. They were designed to classify inputs on a "retina," which would now commonly be called an input vector, but was then an abstraction of the way the retina in biological systems works. They were constructed to classify input samples as belonging to either a positive or negative class, and had outputs in $\{-1, 1\}$ to do so. Their structure was derived from Warren McCulloch's, Walter Pitts's, and Donald Hebb's work that proposed neurons with binary threshold activation functions could be considered as logical predicates and trained in some way by encouraging neurons that fired together to strengthen their connections [5].

A common modern interpretation of the perceptron is that classification to a positive or negative class on a d-dimensional input $x \in \mathbb{R}^d$ using a weight vector $w \in \mathbb{R}^d$ and bias $b \in \mathbb{R}$ can be done using the function F:

$$F(x) = \begin{cases} 1 & w^T x + b \geq 0 \\ -1 & \text{otherwise.} \end{cases} \tag{2.1}$$

A threshold other than zero could be chosen, but any such choice is equivalent to changing the bias term, b. This definition differs from the original construction of a perceptron where a simple perceptron F on a retina x (which in this case can be any type of input) took the form

$$F(x) = \begin{cases} 1 & w^T\psi(x) \geq 0, \\ -1 & \text{otherwise}, \end{cases} \tag{2.2}$$

where $\psi(x)$ is a vector consisting of N "features" of the original input,

$$\psi(x) = \begin{bmatrix} \psi_1(x) \\ \psi_2(x) \\ \vdots \\ \psi_N(x) \end{bmatrix} \tag{2.3}$$

Each of these $\psi_i, i = 1, ..., N$ feature functions were chosen to map from the set of possible x to $\{0, 1\}$. There were no restriction as to what these features could be as long as they had binary outputs. The bias/threshold is not included in this definition, as one of these feature functions could be the constant value 1. The choice to have these feature functions map to $\{0, 1\}$ was based on neurological understanding from the time of McCulloch and Pitts, where neurons were considered to be either "off" or "on" without concept of magnitude beyond that.

To train these, the "perceptron convergence algorithm" was created, based roughly on Donald Hebb's idea that neurons that fire together are strengthened in their connections and neurons that don't fire together are weakened in their connections. Formally, the rule is

1. Given a set of C data samples $\{(x_i, t_i)\}_{i=1}^{C}$ with inputs x_i and target classes $t_i \in \{-1, 1\}$
2. Calculate $f(x_i)$ using the definition of a simple perceptron
3. Based on the "true" class and the predicted class, update weights according to

$$\begin{aligned} w_j &= w_j + r(t_i - f(x_i))(x_i)_j, \quad j = 1, ..., N \\ b &= b + r(t_i - f(x_i)) \end{aligned} \tag{2.4}$$

where r is a learning rate, typically taken to be between 0 and 1.

4. Repeat 2 and 3 until all data samples have predictions equivalent to the target class or a fixed number of iterations have occurred

This rule is designed to modify weights that result in incorrect classification based on their contribution to that incorrect classification. It is a precursor to the more modern backpropagation algorithms with much of the complexity removed.

The original *Principles of Neurodynamics* book states that this perceptron architecture is enough to show "a solution exists for every possible classification." The argument made to justify this is that every possible configuration of the input functions (which are binary valued) could be taken as an input to the perceptron architecture. Then weight choices could then be made to exactly classify each pattern to the expected output. Another simpler way of showing this is to consider a single input feature, $\Psi(x)$, that is zero when the sample is in the negative class and one when the sample is in the positive class. This universal ability for classification is somewhat vacuous — neither construction is feasible in practice, and they are either fitting exactly the dataset or moving all the learning out of the perceptron.

To consider more meaningful conditions, Rosenblatt considered perceptrons where each input function looks only at a subset of the original locations on the retina and is restricted in computational ability. With this restriction, he derived more reasonable conditions for which a solution can exist. The main condition is a restated version of the constraint that the two classes must be linearly separable in the input space. This means that the class of problems that are solvable using a perceptron is somewhat limited unless features are carefully selected by hand. However, perceptrons were used to get strong performance on a variety of problems of the time [6].

Rosenblatt also showed that the perceptron convergence algorithm results in a solution to a problem in a finite number of iterations through the dataset [1]. Specifically, the number of iterations is bounded by a finite function of the initial error, the learning rate, and the size of the dataset. A

[1]Rosenblatt notes that his is not the original proof of this, and that there are many variations of the proof. He also mentions a proof by Papert, one of the authors of *Perceptrons*, that contains "several logical errors," which is interesting in the context that *Perceptrons* stymied perceptron research for a number of years.

corollary to this theorem is that the set of solution vectors is of nonzero measure. This means that there are an infinite number of solution vectors that are all equally able to separate the classes.

Although the theoretical results shown in *Principles of Neurodynamics* have strong assumptions, careful selection of input features could and did result in successes on a variety of problems. However, towards the end of the 1960s interest in them was starting to wane due to their inability to successfully classify more complex problems [6].

2.1.2 The *Perceptrons* Book

In 1969, seven years after the publication of *Principles of Neurodynamics*, Marvin Minsky and Seymour Papert published *Perceptrons*. It contained a variety of theoretical results relating to (a limited class of) perceptrons. Many of these pertained to situations in which a perceptron would perform extremely poorly (e.g. unbounded weight values or an exponential explosion in the number of input features required), although there were a few theorems relating to successes of perceptrons. Some in the machine learning community consider *Perceptrons* to have "killed" machine learning research for almost two decades, though Minsky and Papert dispute this [6].

One of their main tools for analysis of perceptrons is the Group Invariance Theorem:

Theorem 2.1.1. *Suppose that*

1. *G is a finite group of transformations of a finite space R;*

2. *Φ is a set of predicates on R closed under G;*

3. *F is a perceptron that is invariant under G. Then there is a linear representation of F,*

$$F(x) = \begin{cases} 1 & \sum_{\psi \in \Phi} w_\psi \psi(x) \geq 0, \\ -1 & \textit{otherwise}, \end{cases}$$

for which the coefficients w_ψ depend only on the G-equivalence classes of ψ, that is if ψ and ψ' belong to the same equivalence class, then $w_\psi = w_{\psi'}$.

Effectively what this theorem states is that if the perceptron should have the same output for equivalent patterns under group operations such as translation with wraparound, then the network will not be able to differentiate between some dissimilar patterns of input. For example, when considering binary strings with the group operation translation with wraparound, a perceptron cannot differentiate between two different inputs that have the same number of "on" values. As each pattern is tiled across the binary string, it will activate each input a number of times equal to the number of "on" values. The perceptron is not able to discriminate between shifts because over all the translations each input is activated four times for each pattern, so it cannot learn any discriminatory functionality to determine that the patterns are different.

They used the Group Invariance Theorem to show their main negative result for perceptron ability: the parity problem. In the parity problem, a perceptron must learn, given binary strings as inputs, to classify the number of "on" values as being even or odd. The most common interpretation of their results on this problem is:

Theorem 2.1.2. *A perceptron with two binary inputs cannot compute the XOR of its inputs.*

This can be seen by the fact that XOR is not linearly separable, and a necessary and sufficient criteria for a perceptron to solve a classification problem is linear separability. Specifically, for 00 and 11 to be classified correctly, it is necessary for

$$\begin{aligned} w_1 + w_2 + b &\geq 0, \\ w_1 * 0 + w_2 * 0 + b = b &\geq 0. \end{aligned} \tag{2.5}$$

For 01 and 10 to be classified correctly, it must be the case that

$$\begin{aligned} w_1 + w_2 * 0 + b = w_1 + b &< 0, \\ w_1 * 0 + w_2 + b = w_2 + b &< 0. \end{aligned} \tag{2.6}$$

Combining the equations in Equation ((2.5)) yields

$$w_1 + w_2 + 2b \geq 0 \tag{2.7}$$

and combining the equations in Equation ((2.6)) yields

$$w_1 + w_2 + 2b < 0. \tag{2.8}$$

These cannot both be true, so this linear classification system cannot work.

A more general statement of the parity problem is

Theorem 2.1.3. *To be able to classify a bit string of length R as having even or odd parity using bit masks as inputs to the perceptron requires all 2^R masks [6].*

This theorem means that the problem of computing parity is intractable using this structure of features. This theorem (and specifically the XOR example) were taken to mean that simple perceptrons *could not* compute these simple problems without an exponential increase in the size of the network. However, a more accurate statement of what this implies is that there are problems where hand-selecting features as inputs to a perceptron based on reasonable assumptions can lead to unforeseen difficulties.

Minsky and Papert did not only publish negative theoretical results about perceptrons — they also included an elegant form of the proof that the perceptron convergence algorithm works in finite time. They showed that the discovered weight vectors approach the cone of valid solutions at a fast rate. They also compared perceptrons favorably to other machine learning techniques in use at the time.

However, their negative results outweighed their positive findings. For 17 years after the publication of *Perceptrons*, there was limited research in machine learning in the West. Its publication (and really the public response to it) led to a shift in funding and opinion away from learning algorithms and towards more traditional AI topics. That's not to say that there was no research on neural networks, as a few connectionists (as perceptron enthusiasts were called) continued to

investigate neural networks [38]. However, there were relatively limited theoretical results during that time.

2.2 The New Connectionism (1980s-2000s)

In 1986, 17 years after the publication of *Perceptrons*, David Rumelhart, James McClelland and the Parallel Distributed Processing (PDP) group published *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, a two-volume set of books. These books demonstrated experimentally that by using perceptrons as the feature function inputs to perceptron, they can achieve successful classification on a variety of problems, such as XOR. Furthermore, in the first volume, *Foundations* (hereafter referred to as *Parallel Distributed Processing*), they demonstrated that many criticisms of *Perceptrons* did not apply when using this architecture and provided a method of training that generalizes the perceptron learning algorithm [39].

Shortly thereafter, in 1989, George Cybenko published the paper "Approximation by Superpositions of a Sigmoidal Function" proving what is commonly now known as the "Universal Approximation Theorem." This theorem loosely states that the multilayer perceptrons described in the PDP book can approximate any reasonably well-behaved function given an arbitrary number of intermediate feature functions to train [26]. Although it does not provide a bound on the number of feature functions for a given problem, it still demonstrates that this architecture could work on most problems with the caveat that it may take many feature functions. Many similar papers were published in that and later years extending the original statement of the theorem.

2.2.1 *Parallel Distributed Processing* and Backpropagation

Parallel Distributed Processing and the associated paper "Learning Representations by Back-Propagating Errors" authored by Rumelhart, Hinton, and Williams provided an effective method for constructing the feature functions of a network during the learning process [39, 40]. Rather than using predetermined features as inputs to the perceptron, each feature function was itself a perceptron. This architecture was called a multilayered perceptron or neural network. Since

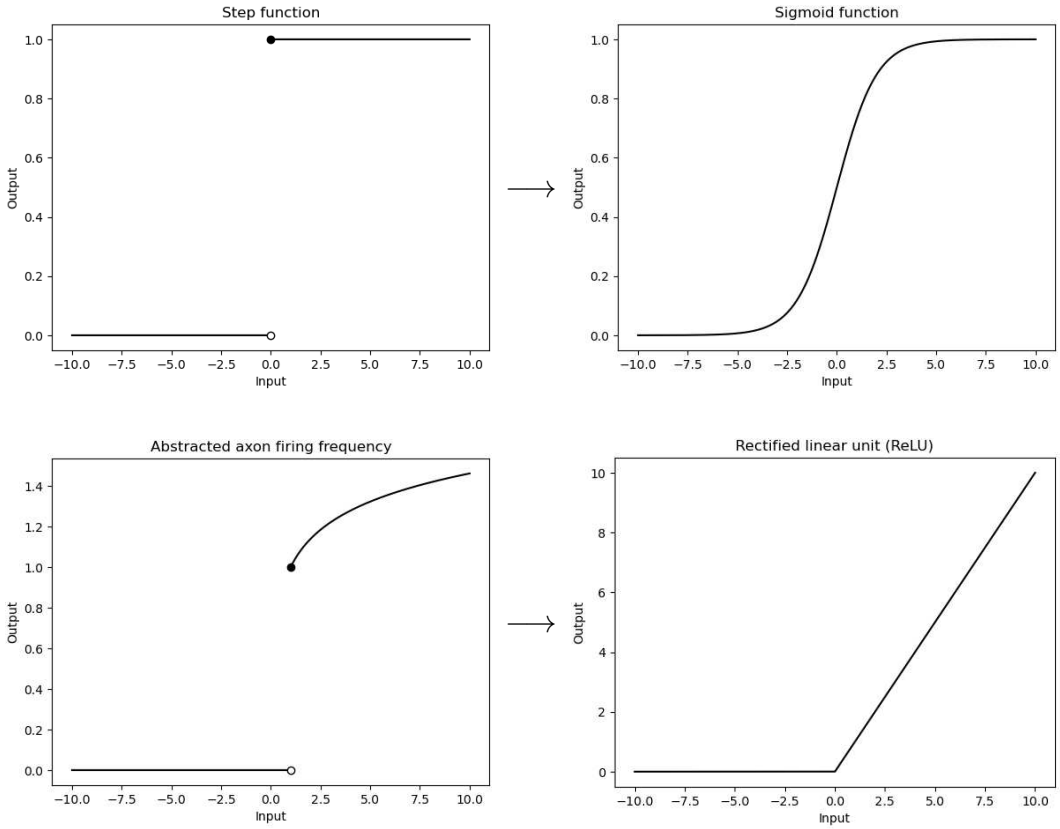


Figure 2.1: The physical meanings sigmoidal and ReLU activation functions grew out of. Top: Sigmoid functions are continuous and differentiable approximations of the step function, which is used to represent a neuron either firing or not. Bottom: ReLU can be considered a continuous and differentiable almost everywhere approximation of the firing rate of a neuron. The left bottom plot is a version of the firing frequency of a neuron based on the injected current as found by the Hodgkin-Huxley model [41].

training these nested perceptrons using the perceptron convergence algorithm would have been impossible, a variation of gradient descent was created to update the weights in the system. This required the activations of the perceptron to be modified from step functions to continuous and differentiable functions[2]. The two main activation functions proposed were the sigmoid function and the threshold linear (ReLU) function shown in Figure 2.1. ReLU can be viewed as a way to match behavior in animal neurons, where the neuron only begins firing after sufficient input and the firing rate continues to increase as it receives more input [41].

[2]As long as the functions are differentiable almost everywhere (in the technical sense of all but a finite set of points) they are acceptable, so ReLU works even with this requirement.

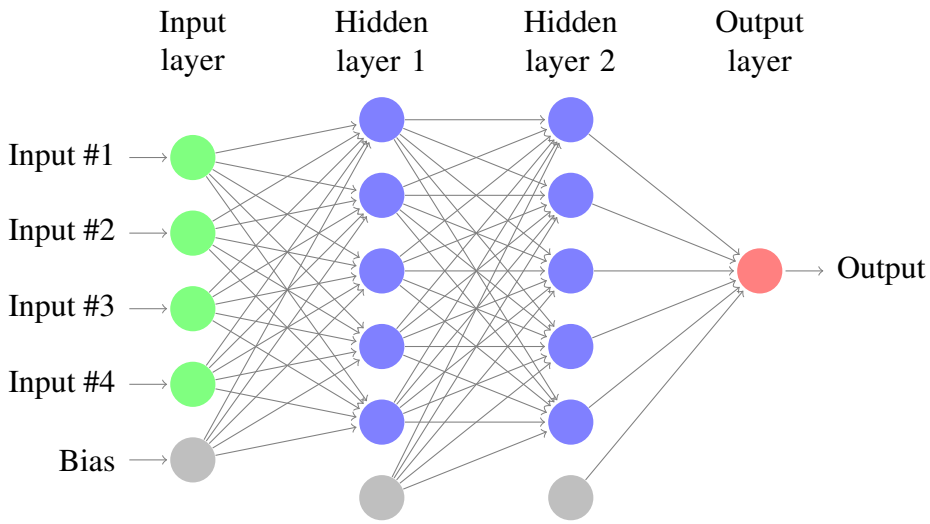


Figure 2.2: The structure of a general multilayer perceptron. Green nodes are inputs, blue nodes are hidden nodes, the red node is an output, and the gray nodes represent constant inputs used as biases. Every line in the diagram represents a weight connecting its source to its destination. The set of weights between a layer and the next is one of the W matrices. After the inputs to each blue and red node are summed, an activation function, σ, is taken of the input before propagating the value forward. Each blue and red node could also be considered as a simple perceptron. More outputs, inputs, layers, or nodes per layers could be used.

To denote these activation functions as being different from the feature functions of the original perceptron, σ will be used to refer to them. Then, a multilayer perceptron F with L layers and N_l hidden nodes in layer $l = 1, ..., L$ (with $N_0 = d$, the original input dimension, and N_L equal to the number of outputs) can be described mathematically as a composition of activation functions, $\sigma^{(l)}$, and affine transformations, $W^{(l)} \in \mathbb{R}^{N_l \times N_{l-1}}$, $l = 1, ..., L$:

$$F(x) = \sigma^{(L)} \circ W^{(L)} \circ \sigma^{(L-1)} \circ W^{(L-1)} \circ \cdots \circ \sigma^{(1)} \circ W^{(1)}(x). \tag{2.9}$$

This style of architecture is shown visually in Figure 2.2. The final activation function, $\sigma^{(L)}$ is typically taken to be linear, sigmoidal, or — in modern times — a softmax function.

It had been previously discussed in *Principles of Neurodynamics* and *Perceptrons* that using a perceptron as a feature function input to another perceptron was possible, but this was posed as more of an intellectual curiosity by Rosenblatt and as something unlikely to be meaningful in practice by Minsky and Papert. The main skeptical ideas mentioned by Minsky and Papert were difficulty in training and the intuition that many of their theorems would generalize to the

multilayer perceptron architecture. *Parallel Distributed Processing* showed that neither of these concerns were relevant.

The learning system described is a specific form of gradient descent using the squared error of the output and target values and computing the gradient for every input sample either one sample at a time or calculated over the whole dataset. It can be thought of as an extension of the perceptron convergence algorithm to multiple layers — the reinforcement that a weight would be given in the original perceptron is propagated backwards according to derivative rules. Originally, the error defined over all C input/output pairs was based on the Euclidean distance between the target output and the network output,

$$E = \frac{1}{2} \sum_{i=1}^{C} ||t_i - F(x_i)||_2^2. \tag{2.10}$$

The fraction $\frac{1}{2}$ is here to cancel with the exponent when the derivative is taken. Since a learning rate will be chosen later, the multiplication by a fixed constant here does not make a meaningful difference. For each output value, $f_j, j = 1, ..., N_L$, the partial derivative of that output can be calculated for a specific input/output pair with index i as,

$$\frac{\partial E}{\partial F_{i,j}} = t_{i,j} - F_{i,j}. \tag{2.11}$$

Although originally the backpropagation rule was designed for this specific error function, we will derive the backpropagation rules for an arbitrary differentiable error function, E, following a similar format as *Parallel Distributed Processing*, *Neural Networks and Deep Learning*, and "Learning Representations by Back Propagating Errors." Additionally, For notational convenience, let $a^{(l)}$ be the vector of pre-activation weighted sums of the outputs of layer $l-1$ and $o^{(l)}$ be the vector of post-activation outputs of layer l, so that

$$o^{(0)} = x, \tag{2.12}$$

$$o^{(L)} = f(x), \tag{2.13}$$

$$o^{(1)} = \sigma(W^{(1)}x), \tag{2.14}$$

$$a^{(l)} = W^{(l)}o^{(l-1)}, \quad l = 1, ..., L, \tag{2.15}$$

$$o^{(l)} = \sigma(a^{(l)}) = \sigma(W^{(l)}o^{(l-1)}), \quad l = 1, ..., L. \tag{2.16}$$

It is also useful to define the partial derivatives of E with respect to $a^{(L)}$. For layer L, a specific element of the vector, $a_j^{(L)}$, and σ' representing the derivative of σ with respect to its input,

$$\frac{\partial E}{\partial a_j^{(L)}} = \frac{\partial E}{\partial o_j^{(L)}} \frac{\partial o_j^{(L)}}{\partial a_j^{(L)}} = \frac{\partial E}{\partial o_j^{(L)}} \left(\frac{\partial}{\partial a_j^{(L)}} \sigma(a_j^{(L)}) \right) = \frac{\partial E}{\partial o_j^{(L)}} \sigma'(a_j^{(L)}). \tag{2.17}$$

This can be vectorized, using $\odot$ as the Hadamard product (elementwise multiplication) and $\nabla_{o^{(L)}}$ as the gradient with respect to $o^{(L)}$, to

$$\nabla_{a^{(L)}} E = \nabla_{o^{(L)}} E \odot \sigma'(a^{(L)}). \tag{2.18}$$

It is useful to also write the gradients of E with respect to these $a^{(l)}$ as $\sigma^{(l)}$ so that for a non-output layer, l, the value $\delta^{(l)}$ represents the error to the input to the l^{th} layer, i.e.

$$\delta^{(l)} = \nabla_{a^{(l)}} E. \tag{2.19}$$

To find a formula for this, we can write this gradient in terms of the gradient of E with respect to $a^{(l+1)}$,

$$\nabla_{a^{(l)}} E = \left(\begin{bmatrix} \frac{\partial a_1^{(l+1)}}{\partial o_1^{(l)}} & \cdots & \frac{\partial a_1^{(l+1)}}{\partial o_{N_l}^{(l)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial a_{N_{l+1}}^{(l+1)}}{\partial o_1^{(l)}} & \cdots & \frac{\partial a_{N_{l+1}}^{(l+1)}}{\partial o_{N_l}^{(l)}} \end{bmatrix}^T \nabla_{a^{(l+1)}} E \right) \odot \nabla_{a^{(l)}} o^{(l)}. \tag{2.20}$$

This first gradient term is $\delta^{(l+1)}$ by definition. The matrix of gradients is exactly $W^{(l+1)}$. The second gradient is exactly $\sigma'(a^{(l)})$, so this can be written as

$$\delta^{(l)} = \nabla_{a^{(l)}} E = \left((W^{(l+1)})^T \delta^{(l+1)}\right) \odot \sigma'(a^{(l)}). \tag{2.21}$$

Then, for a specific weight in layer l mapping from input node k to output node j, we have that the partial derivative of the error with respect to it is given by the input to that element, $o_k^{(l-1)}$ multiplied by its "effect" on the final output, $\delta_j^{(l)}$:

$$\frac{\partial E}{\partial w_{jk}^{(l)}} = o_k^{(l-1)} \delta_j^{(l)}. \tag{2.22}$$

Combining Equations ((2.18)), ((2.21)), and ((2.22)) and explicitly writing a bias derivative rule for a bias b (where the "input" to the bias is a constant one) gives the set of equations

$$\begin{aligned}
\delta^{(L)} &= \nabla_{o^{(L)}} E \odot \sigma'(a^{(L)}), \\
\delta^{(l)} &= \left((W^{(l+1)})^T \delta^{(l+1)}\right) \odot \sigma'(a^{(l)}), \quad l = 1, ...L-1, \\
\frac{\partial E}{\partial w_{jk}^{(l)}} &= o_k^{(l-1)} \delta_j^{(l)}, \\
\frac{\partial E}{\partial b_j^{(l)}} &= \delta_j^{(l)}.
\end{aligned} \tag{2.23}$$

The term backpropagation comes from the fact that the error is propagated "backwards" through these deltas to earlier layers. Weights can then be updated according to

$$\begin{aligned}
w_{jk}^{(l)} &= w_{jk}^{(l)} - r \frac{\partial E}{\partial w_{jk}^{(l)}} \\
b_j^{(l)} &= b_j^{(l)} - r \frac{\partial E}{\partial b_j^{(l)}}
\end{aligned} \tag{2.24}$$

where $0 < r < 1$ is a learning rate.

Two initial variations of this backpropagation were proposed. In the first, which is similar to the perceptron convergence algorithm, each sample was fed through individually and weights were adjusted immediately. In the second, all samples were fed through and the gradient was computed over the entire training set. They noted that the second method is potentially preferred because it has stronger theoretical results for convergence and convergence rate, but that the first method tended to converge faster. This has been a common trend in neural network theory, where theoretical results for weaker algorithms exist (gradient descent on the whole training set) but there are algorithms that drastically improve in speed or performance over those algorithms (such as stochastic gradient descent).

Although this method is similar to the perceptron convergence algorithm, it unfortunately does not have the same theoretical results for convergence time. The increase in depth from a perceptron also changes the problem of optimizing weight values from a convex problem to a non-convex problem, meaning that there may be local minima and saddle points for the discovered weight vectors that are not close to the global optima. Although quality and speed are no longer guaranteed as in the perceptron converge algorithm, it had high practical success. It was also conjectured at the time, based on experimental results, that poor local minima occurred relatively frequently. No significant theoretical evidence existed for this at the time [3].

One other note to make here is that gradient descent methods for similar optimization problems were proposed in the 1970s[4] and applied to multilayer perceptrons in 1974 by Paul Werbos in his Ph.D. dissertation [43]. These results were not as publicized, meaning that the later independent discoveries of backpropagation for multilayer perceptrons are considered to be the "original" description of the idea due to their popularizing effect.

[3] but as will be discussed later, it has been shown that large, deep neural networks potentially have few local minima not near the global minima and the chance of recovering poor solutions approaches zero as network complexity increases.

[4] The gradient descent algorithm was originally published by Cauchy in 1847 [42].

2.2.2 The Universal Approximation Theorem for Wide Networks

A general Universal Approximation Theorem for depth-bounded networks is, based on work by Cybenko and Hornik [26–28],

Theorem 2.2.1. *Let $\sigma : \mathbb{R} \to \mathbb{R}$ be a nonconstant, continuous, and bounded function. Let I_m be the m-dimensional unit hypercube, $[0,1]^m$. Then, for arbitrary $\epsilon > 0$ and any continuous function f defined on I_m, there exist an integer N, constants $v_i, b_i \in \mathbb{R}$, and vectors $w_i \in \mathbb{R}^m$ for $i = 1, ..., N$ that can be used to define*

$$F(x) = \sum_{i=1}^{N} v_i \sigma\left(w_i^T x + b_i\right)$$

as an approximation to the function f, that is

$$|F(x) - f(x)| < \epsilon$$

for all $x \in I_m$. Equivalently, functions of the form of F are dense in the continuous functions on I_m.

We will go through each piece of this individually.

Let $\sigma : \mathbb{R} \to \mathbb{R}$ be a nonconstant, continuous, and bounded function: This is a statement of the constraints on the activation function. It must be nonconstant so that there can be variation as inputs change. Continuity is a requirement so that the neural network output will be a continuous function. The bounded requirement is that there is a maximum value the magnitude of the function is less than. Although this seems to disallow the ReLU activation function, this is on a compact set and so ReLU will achieve a maximum value. These three requirements combine to mean that activation functions are discriminatory (nonconstant), well-behaved (bounded and continuous), and usable in the measure theory identities used in the proof (bounded and continuous).

Let I_m be the m-dimensional unit hypercube, $[0, 1]^m$: This is a requirement that the domain of interest must be in some way bounded — input values cannot vary to infinity. The specific restriction to the hypercube isn't necessary, and any compact (closed and bounded) set can be used. Any finite or bounded infinite set of data necessarily satisfies this assumption.

For arbitrary $\epsilon > 0$ and any continuous function f defined on I_m: Here, f is the function of interest that is being approximated. It can be approximated ϵ closely by the constructed network.

There exist an integer N, constants $v_i, b_i \in \mathbb{R}$, and vectors $w_i \in \mathbb{R}^m$ for $i = 1, ..., N$ that can be used to define $F(x)$: These are the parameters of the network. The width of the network is N (which can be arbitrarily high), w_i are the initial weight vectors on the input, b_i are the initial biases, and v_i are the weights from the hidden layer to the output layer. There is no bias from the hidden to the output layer as a w_j could be taken to be all zeros and the value $\sigma(w_j^T x + b_j) = \sigma(b_j)$ would then be constant.

$|F(x) - f(x)| < \epsilon$ for all $x \in I_m$: This is the statement that the neural network approximates the function of interest arbitrarily well. Any $L^p(I_m)$ norm could be used instead[5] to extend the set of functions that can be approximated [28] : If f is a member of $L^p(I_m)$, i.e. finite under the Lebesgue integral,

$$||f||_p = \left(\int_{I_m} |f(x)|^p dx \right)^{1/p} < \infty$$

so that the p-norm of f is calculable, then the approximation closeness can be calculated as

$$||F - f||_p < \epsilon. \tag{2.25}$$

The proofs of these theorems are somewhat complex, involving topics from measure theory, but the book *Neural Networks and Deep Learning* provides a more intuitive proof of these theorems for the sigmoidal function [44]. This is shown visually in Figure 2.3. The general idea is to take two sigmoidal functions, each of which approximates a step function, and use them to construct a rectangle. Then, many sets of these pairs can be used to make rectangles of different heights across space. More pairs of sigma functions can be added, taking the width of each rectangle smaller, until the error becomes arbitrarily small. A similar construction can be done for ReLU, but it requires four nodes to make each rectangle rather than two.

[5] Any continuous function on a compact set is a member of L^p for $p = 1, 2, ...$, so any L^p norm is usable in the original theorem.

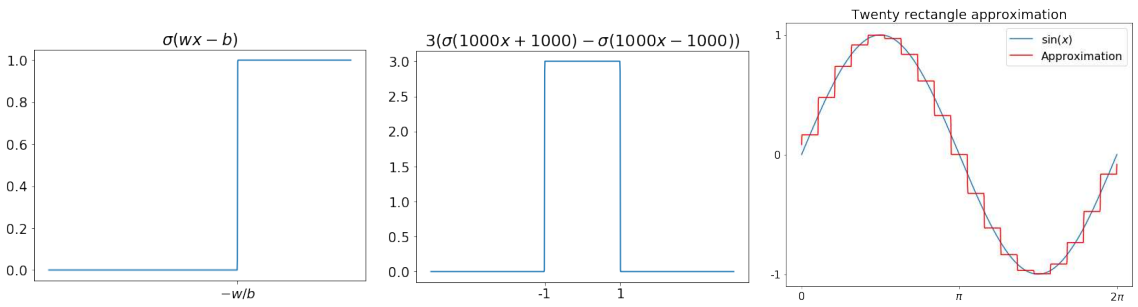


Figure 2.3: A visual representation of the Universal Approximation using sigmoidal activation functions. Left: A sigmoidal function can be made arbitrarily close to the step function by increasing w. Middle: Two sigmoidal functions that have been made to be step functions and added together to create a rectangle. Right: An approximation of a $\sin(x)$ by a set of 20 rectangles which requires 40 sigmoidal units. More pairs of sigmoidal units could be used to increase the number of rectangles and increase their widths.

2.2.3 Architectural Improvements

There have been many architectural improvements for constructing or training neural networks. Many of them have been motivated by theoretical concepts, but often the theory does not fully analyze the advancements. Many of the architectures can also be shown to simplify to a standard feedforward fully connected network. Two of those improvements — convolutional layers with max-pooling and dropout — are discussed as a sample of these advancements.

In 1980, Kunihiko Fukushima proposed the "neocognitron" architecture of a neural network. Neocognitrons provided an unsupervised method for learning visual patterns. The main advancements in this were two-dimensional filter connections, rather than fully-connected nodes, and average pooling to downsample the network [38]. This extended previous ideas from early work with perceptrons and fit with research on biological vision systems. Yann LeCun extended this in 1998 to create neural networks that use "convolutional" layers and max-pooling [36]. These lowered the number of weights necessary for a network to learn and gave local translation invariance.

Convolutional layers have filters with limited receptive fields, so each filter operates on only a subset of the total inputs at a time. Each filter is tiled across the input using the same weights so that fewer parameters are necessary. This is frequently stated as giving translational invariance, but it is really translational equivalence over the filter outputs — which still preserve locality information based on the input regions. That's not to say that they don't help with translational invariance,

though, as it is likely that they provide stronger methods for the network to "learn" representations that are translationally invariant. To assist further with translation invariance, downsampling is used to allow small translations to be ignored. The size of translation that the network will then be invariant over will be roughly the size of the receptive field of the downsampling method.

Geoffrey Hinton and his collaborators proposed a regularization technique called dropout in 2012[6] [37]. Previously, a method for regularization of networks had been to train many independent networks and use some averaging method over their outputs when testing. Unfortunately, this is infeasible when large networks can take weeks or months to train. The dropout technique approximates this ensemble method by sampling from a set of many smaller neural networks while training.

When using dropout during training, each hidden node in the network has a probability p of being ignored by the next layer. This creates a set of "subnetworks" (2 to the number of nodes of them) based on which nodes are active. Networks are sampled from this set during training based on random chance. When training has completed, all the hidden nodes are used and the network can be considered to be an approximation to the ensemble of all the subnetworks. Using a dropout of 50% leads to the highest expectation of sampled models, but other proportions may result in better results.

Although both of these techniques have limited theoretical analysis of *whether* they work, they are motivated by theoretical ideas for *why* they work. This is a common occurrence in advancements, where a theoretical motivation drives a change that results in "better" experimental results on test problems, but theoretical explanations for what effect the method has on the network are underdeveloped. Similar techniques to this include momentum for gradient descent (which does have theoretical results when a convergence rate is already known), recurrent neural networks, batch normalization, etc.

[6]This is after the advent of deep learning. The reason this discussion is in this subsection is because architectural improvements fit better here.

2.3 Deep Learning (2012-Today)

Although there were strong theoretical results, the limitations of computational power in the 1990s and early 2000s led to a decrease in the usage of neural networks for solving problems. However, in 2012 there was a resurgence in neural network research after the success of AlexNet on the ImageNet dataset [45]. With this resurgence, there was a shift towards deep neural networks, where there is more than one (and frequently many more than one) hidden layer. The ReLU activation function also primarily replaced sigmoidal functions, as it is faster to compute. This brought along with it an entirely new aspect of the theory in width-bounded networks, comparisons between wide and deep networks, and combinatorial investigations of ReLU networks.

Additionally, with the proliferation of architectures designed for specific domains, especially image classification and language modeling, theoretical results have broadened in their goals. With the "low-hanging fruit" already found, results also have become more niche. This lead to a spread in what results say, but fewer results that are universally meaningful.

A result from 2017 is that ReLU neural networks with a width of at most $d + 4$ are universal approximators for single output functions as the depth increases (this was later refined to d plus the dimension of the output [24]). In the same paper, it was shown that width d networks cannot approximate many problems [23]. A related result is that deep networks are in many respects exponentially more expressive in terms of functions they can represent than wide networks [23,35].

A few other results combine to show that networks with many parameters are likely to train faster and find better local minima (in terms of generalization) than smaller networks [29]. Specifically, many of the problems encountered when training large neural networks arise from the distribution of saddle points rather than the distribution of local minima.

2.3.1 Necessary Width

The Universal Approximation Theorem shown for width-bounded neural networks is [23–25]

Theorem 2.3.1. *For any Lebesgue-integrable function $f : \mathbb{R}^d \to \mathbb{R}$ and any $\epsilon > 0$, there exists a fully connected ReLU network, F with width $N = d + 1$ such that*

$$\int_{\mathbb{R}^d} |f(x) - F(x)|dx < \epsilon$$

The requirement that f be a Lebesgue-integrable function is so that the Lebesgue integral of f,

$$\int_{\mathbb{R}^d} |f(x)|dx$$

is calculable. This requirement is different than the Universal Approximation Theorems from the '80s and '90s in that it no longer considers a compact set and f is no longer required to be a continuous function.

Similar to the method for showing wide networks can be constructed out of a set of rectangles, the proof of this theorem relies on constructing "blocks" of width $n + 4$ and depth $4n + 1$ that can approximate Lebesgue integrable functions on a compact set with arbitrary accuracy while maintaining (and summing with) the approximations from previous blocks in the network.

They also show networks with maximum width below a bound cannot learn a wide class of functions well [23]:

Theorem 2.3.2. *For any Lebesgue-integrable function $f : \mathbb{R}^d \to \mathbb{R}$ satisfying that $\{x : f(x) \neq 0\}$ is a positive measure set in Lebesgue measure, and any function F represented by a fully-connected ReLU network with width $N \leq d$, the following equation holds:*

$$\int_{\mathbb{R}^d} |f(x) - F(x)|dx = \infty \textit{ or } \int_{\mathbb{R}^d} |f(x)|dx$$

One important assumption here is **$\{x : f(x) \neq 0\}$ is a positive measure set in Lebesgue measure**. The intuitive meaning of this is that the area the function is nonzero on must have volume (in the d-dimensional sense, not in the 3-dimensional sense) in d-dimensional space. If this weren't true, there would be a reduction of the function to $d - 1$ dimensions, where the network can succeed with d width. This is a meaningful distinction, as it has been hypothesized that most (if not all) high-dimensional datasets are really a set of lower-dimensional manifolds embedded in

high-dimensional space. If this is true, it permits a certain amount of reduction in width necessary for a network to successfully approximate many datasets.

Another important note is that functions that are impossible to solve with d dimensional neural networks are not necessarily hard to solve with $d+1$ dimensional networks. As shown in Figure 2.4, this doesn't necessarily mean that problems that can't be solved by a small network are "hard." There are problems that are "easy" or difficult to solve in n dimensions, and there problems that are impossible to solve well in n dimensions using a neural network, but may be "easy" to solve by increasing the width of the network slightly.

2.3.2 Number of Linear Regions

It has been shown that the maximum number of piecewise linear patches that a ReLU neural network constructs is bounded above by, for a constant k smaller than the lowest width layer in the network, i.e. $k \leq N_l, l = 1, ..., L$, [33, 34]

$$\prod_{l=1}^{L} \sum_{i=0}^{k} \binom{N_l}{i} \tag{2.26}$$

If $k \leq N_1, ..., N_L, \leq N$, then the asymptotic behavior of the number of linear regions is $\mathcal{O}(N^{k(L-1)})$.

Similarly, a lower bound on the maximum number of piecewise linear patches can be shown to be, for a network with N_0 input units and $N_i \geq N_0, i = 1, ..., L$ [32]

$$\left(\prod_{i=1}^{L-1} \left\lfloor \frac{N_i}{N_0} \right\rfloor^{N_0} \right) \sum_{j=0}^{N_0} \binom{N_L}{j} \tag{2.27}$$

so the asymptotic behavior is, for width N in all layers,

$$\Omega \left(\left(\frac{N}{N_0} \right)^{(L-1)N_0} N^{N_0} \right) \tag{2.28}$$

These upper and lower bounds are asymptotically tight. This suggests that the "complexity" of a neural network grows polynomially in width and exponentially in depth. However, the number

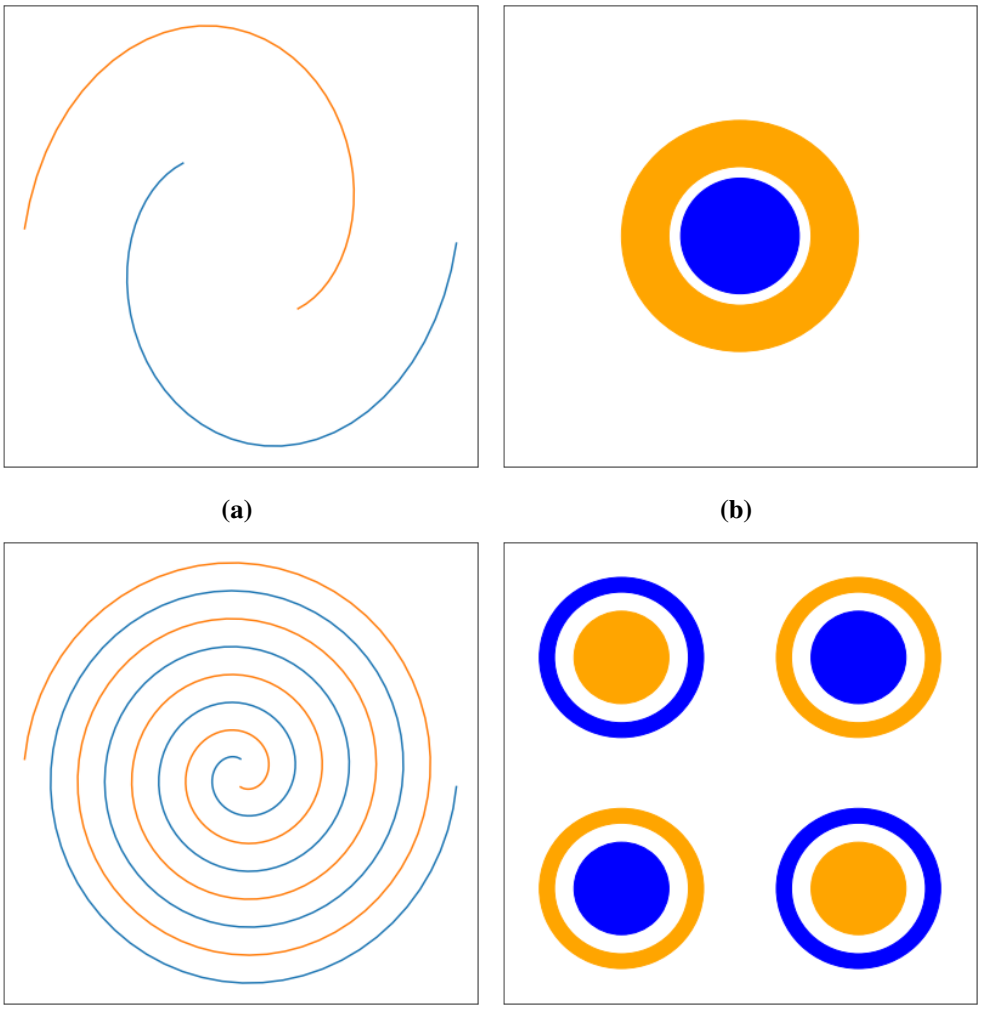


Figure 2.4: A set of classification problems illustrating easy, difficult, and impossible problems for certain networks. Blue and orange points have different classes. **(a)** A simple problem that can be solved by a width-2 bounded ReLU network. **(b)** A simple problem that cannot be solved with any width-2 bounded ReLU network. **(c)** A complex problem that can be solved by a width-2 bounded ReLU network with sufficient depth. Increasing width may assist in classification. **(d)** A complex problem that cannot be solved by a width-2 bounded ReLU network.

of linear regions isn't necessarily a strong indicator of the complexity representable. There are constraints on the values of the piecewise linear regions, and this analysis does not include that.

Additionally, this is the possible expressivity of a neural network, not the actual expressivity. A simple illustration is that a neural network with a single hidden layer where every node computes the exact same combination of the inputs will only have two linear regions, as it can be reduced to a neural network with a single hidden node. However, there is typically a combinatorially large number of used linear regions in this style of network.

2.3.3 Width Versus Depth

Although knowing the number of linear regions does not give true insight into how complex a neural network is, there are results related to that. Two such theorems are

Theorem 2.3.3. *For every natural number k, there exists a function representable by a ReLU deep neural network with k^2 hidden layers and total size k^3, such that any ReLU deep neural network with at most k hidden layers will require at least $\frac{1}{2}k^{k+1} - 1$ total nodes. [35]*

Theorem 2.3.4. *For integer k, there exist a class of width-$\mathcal{O}(k^2)$ and depth[7]-2 ReLU networks that cannot be approximated by any width-$\mathcal{O}(k^{1.5})$ and depth-k networks. [23]*

What these theorems combine to give is that any wide network can be transformed into a deep network with only up to a polynomial size increase, but certain deep networks cannot be transformed into a wide network without an exponential increase in width. Although increasing the depth of a network will increase the total number of weights more than increasing the width of a shallow network, this increase is only polynomial so the result remains a polynomial increase. Intuitively what this means is that increasing depth (after having *enough* width — at least $d + 1$) will increase the class of representable functions by significantly more than increasing the width.

The Vapnik-Chervonenkis (VC) dimension of neural networks with various activation functions gives a different way of determining representational capacity. For networks with ReLU acti-

[7] input to hidden and hidden to output

vation functions, the VC dimension can be shown to be $\mathcal{O}(W*d*\log(W))$ and $\Omega(W*d*\log(W/d))$ where W is the number of weights and d is the depth. For U ReLU activations, it is $\Theta(WU)$ [46]

Investigation of the outputs of various layers of the network has also taken place. Methods to do so include visualization [47–49], semantic interpretation [50, 51], and similarity metrics.

2.3.4 "The Loss Surfaces of Multilayer Networks"

In 2015, Choromanska *et al.*published "The Loss Surfaces of Multilayer Networks" [29]. In this paper, they showed that under certain assumptions, neural network learning could be analyzed using random matrix theory and viewed as the physical problem of spin glasses [29]. They were motivated by previous work that showed that the cost surfaces of many non-convex optimization problems had significant numbers of saddle points, but potentially a low number of poor local minima.

One of their preliminary results is that common loss functions (L2 error and cross-entropy) can be interpreted as polynomials in the weights with degree equal to the number of layers and number of monomials equal to the number of paths from inputs to outputs (which is a large number). As the weights and inputs vary, some of these monomials may be deactivated (since the activation functions are ReLU), resulting in a piecewise continuous polynomial with monomial switches on the boundaries of the piecewise regions.

Their main result is that under a few mild assumptions and two unreasonable assumptions a neural network with ReLU activations can have its error function transformed into the Hamiltonian of a spherical spin-glass model. The two unreasonable assumptions are that paths from input nodes to output nodes are independent of the input and that paths have input data that is independent. These effectively would mean that activation patterns are not being learned for similar inputs and that hidden nodes in the network would only see one input. However, by making these assumptions they are able to use theorems proved for the Hamiltonian of the spherical spin-glass model. Specifically, as the "mass" of the neural network, which is the total size in number of parameters, increases, the local minima of the system become situated in a narrow bound around

the global minima with high probability. Additionally, outside of that band there are likely many saddle points. They conjecture that this means much of the difficulty in training networks comes from these saddle points and not from finding poor local minima.

One corollary to this work is that increasing the size of the network can increase success, both on the set considered and on more general problems, counter to traditional effects of overfitting. By increasing the size, poor local minima are transformed into saddle points or removed, and the "good" local minima near the global minima typically generalize well.

2.3.5 Lottery Hypothesis and Network Compression

Even with the knowledge that increasing network size can result in better performance, much of that size may become irrelevant after initialization. Much work has been done to investigate how networks can be compressed — determining how network architectures can be constructed to be smaller [52], use of weight sharing [53], or reduction of the bits necessary to represent a weight, sometimes to a single bit [54]. One promising area is the idea of the "lottery hypothesis." Here, certain paths through the network win the "lottery" of initialization, and the training problem boosts their significance above other paths. The less significant paths can then be removed from the network without significantly decreasing its performance (and in certain cases removal can increase the performance) [21]. It can be shown that for every bounded distribution and target network with bounded weights, a sufficiently overparameterized network with random weights contains a subnetwork with roughly the same accuracy as the target network without any training [30].

Various methods have been used to do this removal. One of the most common is simply attenuating the bottom quantiles of the weight sizes, with removal up to 99% preserving original network performance [55]. Alternatively, rather than constructing a sparse network, pruning the structure of the network can also be done to more directly increase the speed of the network [56]. In addition to using the new, attenuated network, some methods train the result, and some also reinitialize the network to the starting weights after removing the paths and train from the start. The widespread use of pruning methods has led to a plethora of pruning papers, many of which present their results

in incompatible ways [55, 57]. There are also concerns that many pruning methods do not actually achieve significant successes compared to simply training a smaller architecture effectively [58].

Identifying a holistic way of comparing pruned networks, both with each other and against the original unpruned network, is necessary to fully understand the potential benefits obtained by using distinct methods of pruning. It is an open question to what extent do pruned, unpruned, and independently trained smaller architecture find similar solutions on a task.

2.3.6 Convergence Guarantees

The previous two sections discuss how overparameterized networks are likely to have many local optima that are effective for solving the task of interest. However, they do not necessarily directly answer the question of how well a network will train to good optima.

Goodfellow *et al.* [59] found that although overparameterized networks are solving large, non-convex problems, they are unlikely to encounter difficulties in training arising from the non-convexity. Specifically, on the straight path in weight space from a network initialization to the solution found, they found that for a variety of models that were state-of-the-art in 2014, local minima and saddle points did not appear to slow the trajectory of stochastic gradient descent. Their results are experimental, and do apply only to large-production style networks, but they suggest that converge is in many ways "easy" for those networks.

Linear neural networks (networks with linear activation functions) have been studied as a proxy for more complex behavior when nonlinearities are added. Despite the fact that the network behaves as a linear function, there are nonlinear dynamics of training from the multilinear structure of the combination of weights that result in the existence of optima that are not global minima. Various methods of studying this have been used, including work by Eftekhari [60], Arora *et al.* [61] and Nguegnang *et al.* [62]. Primary findings are that convergence to a global at a linear rate is guaranteed when the widths of the hidden layers are at least the minimum of the input and output dimension; weight matrices are approximately balanced at initialization, that is

$||W_{j+1}^T W_{j+1} - W_j W_j^T||_F \leq \delta$ for some small δ and all the W_j weight matrices; and the initial loss is smaller than that of any rank-deficient solution.

Allen *et al.* [31] demonstrated that for sufficiently overparameterized networks with ReLU activation functions and a smooth loss function, stochastic gradient descent finds global minima in polynomial time. This result requires that the network width be polynomial in the number of layers and number of samples. They determine that for networks of at least this size, the training landscape near random initializations is almost convex and smooth. However, the requirement that the network widths be polynomial in the number of layers and number of samples is significant, as the number of training samples for many modern problems of interest is large enough that networks of sufficient size are not feasible to construct [4, 63, 64].

2.3.7 Representational Similarity

Various methods have also been undertaken to understand how similar the models generated by using different architectures or training data are. Being able to compare models more meaningfully than using a simple accuracy measure on a test set is vitally important for being able to understand what meaningful improvements various architectural structures have, and when networks are likely to contain similar issues for classification.

Li *et al.* [65] aligned individual hidden units of ImageNet-trained deep convolutional neural networks (CNN) using a matching algorithm to investigate how similar representations were across different networks. They conclude "some features are learned repeatedly in multiple networks, but other rare features are not always learned." This doesn't necessarily mean that those rare features are not encoded, or that semantic information is reduced — it's argued that the entire feature space encodes semantic information, and individual nodes do not necessarily [66]. Wang *et al.* [67] adds to this work by identifying minimal subsets of neurons which correspond linearly. They find virtually no similarity between different networks, in contrast to many other results, but their experiments involve comparisons of convolutional layer activations (which can be spatially sensitive) and consider subsets rather than full representations or individual nodes.

In addition to matching neuron behaviors, correlation methods have been used for studying representational similarity as well. Various methods with slightly different properties are used for this, such as canonical correlation analysis (CCA), singular vector CCA (SVCCA), projection weighted CCA, and centered kernel alignment (CKA) [68–70]. There are a number of results that have been discovered through this work — earlier layers tend to be learned faster than later layers [68], networks of similar sizes tend to be more similar [69], and networks trained on different datasets can be similar [70]. Depending on the correlation technique used, such methods may be invariant to some combination of transformations such as orthogonal transforms, isotropic scaling, or linear transformations. This variation in what the methods "ignore" means that the selection of a good correlation method requires meaningful evaluation. Kornblith *et al.* [70] approached this problem by assessing how well similarity indexes performed when attempting to identify if two CNNs were architecturally equivalent but trained from different random initializations.

Lenc and Vedaldi [71] used affine mappings between layers of AlexNet [72], VGG-16 [73], and ResNet-v1-50 [15] on the ImageNet Large Scale Visual Recognition Challenge 2012 (ILSVRC2012) dataset to investigate similarity. Their experiments focused on mappings between convolutional layers (which can be spatially sensitive), and they state that "there is a correlation between the layers' resolution and their compatibility."

McNeely-White *et al.* [74] identified linear similarity in the representations of Inception and ResNet networks trained on ILSVRC2012-trained despite differences in architecture. This result has been expanded by McNeely-White *et al.*to cover multiple different models trained on ILSVRC2012 and various facial recognition networks trained on disparate datasets [75, 76]. However, as the authors note, for certain networks, such linear similarity is expected.

All of these efforts focus on identifying a correspondence between the outputs of various layers of neural networks. In contrast, our work here focuses on identifying correspondence between the underlying linear models used for classification, which may provide a more holistic image of correspondence between networks.

2.3.8 Linear Region Analysis

As discussed in Section 2.3.2, upper and lower bounds have been calculated on the number of linear regions a network can have. However, empirical investigations of the number of linear regions actually achieved by many neural networks have shown different results. Untrained neural networks after initialization have a number of linear regions that tends to grow linearly in the number of ReLU functions along any one-dimensional subspace of the input space [16]. Furthermore, they tend to grow polynomially in the number of ReLU nodes in the network and exponentially in the dimension of the inputs to the network [17]. This analysis of practical networks is necessary for truly understanding how networks behave — knowing that neural networks can be vastly complex is less useful if they are only *very* complex.

More recently, analysis of the linear regions of piecewise linear neural networks have been studied. These linear regions have been used empirically to measure the sensitivity of neural networks. As will be discussed in Section 5.1, the Jacobian of a neural network at a point, together with the value of the neural network at the point, describes exactly the linear function that agrees with the network in a polytope around that point. Novak *et al.* [18] utilized this fact to investigate the effect of hyperparameters on input sensitivity and found that overparameterization, surprisingly, can help in generalization. Additionally, they and Zhang *et al.* [19] investigated how the linear region structure can be used to predict the generalization of a network and what effect various regularization methods have on linear region geometry.

Jamil *et al.* [77,78] and Liu *et al.* [79] demonstrated that in-domain and out-of-domain samples have different bitstrings corresponding to the activations of the ReLUs within the networks — i.e. in-domain and out-of-domain samples belong to different classes of the polyhedra in the input space. Certain saliency mapping algorithms use gradients of the network with respect to the input to determine what portions of the image are relevant to classification [80].

This analysis of the localized behavior of networks, and the structures that they form, is useful for understanding how they behave. Our work here is most similar to these, as we use these structures to investigate various metrics for comparing across networks.

2.3.9 Tropical Algebraic Geometry

Analysis of ReLU neural networks has also been extended to investigation using tropical[8] algebraic geometry. One of the main results of Zhang *et al.* [20] is that the set of ReLU feedforward neural networks with integer-valued weights[9], the set of piecewise linear functions with integer-valued coefficients, and the set of tropical rational functions are equivalent (effectively the difference between two maximums of multiple linear terms — more detail on their structure is in Section 5.4).

This is not necessarily immediately useful, beyond giving a different formulation of something like a universal approximation theorem, but it does provide an alternative toolset to build theoretical foundations for neural networks. Zhang *et al.* [20] used this formulation to reprove the upper bound on the number of linear regions that a ReLU neural network can have, as originally proved by Montufar *et al.* [12].

This property has also been used for further analysis. Alfarra *et al.* [81] used this reformulation to identify evidence for the lottery hypothesis and construct adversarial attacks. Trimmel *et al.* [82] used tropical algebraic geometry insights to construct the TropEx algorithm, which simplifies a network to use only the linear regions training samples lie in.

2.4 Open Questions

Although there is a substantial amount of theory, there are still far more unknowns than knowns. Very little is known about the structure of datasets and under what conditions neural networks will perform well given an architecture. Hyperparameter selection for neural networks is often still done using variations of random search [83, 84]. Many of the criticisms levied by Minsky and Papert in the addendum to *Perceptrons* in 1989 are still valid complaints about the machine learn-

[8]The term tropical comes from an early adopter of the field working in Brazil rather than from any underlying mathematical meaning to the term.

[9]This may appear to be a significant constraint, but real values can be approximated by a rational value and then the denominator of all of the weights can be factored out of the network to yield a scaled network with integer valued weights.

ing community — there are still far more practical advancements than theoretical understanding. However, those concerns have mainly shifted from "what problems can perceptrons solve or not solve" to "what problems can neural networks solve or not solve *well*." Current theoretical results also tend to be of less value to practical users of machine learning techniques as they frequently require assumptions that are not feasible.

A few of the large open questions in the field are:

- Is the manifold hypothesis true? — most high-dimensional data is conjectured to be a set of lower-dimensional manifolds embedded in a high-dimensional space.

- Do deep neural networks exhibit useful implicit regularization? — there is work that shows large neural networks undergo a specialized form of regularization while training using small enough batch sizes in stochastic gradient descent [85].

- When can neural networks be trusted? — it's difficult for users to understand what neural networks have learned and when they will generalize their results well. If neural networks are to be trusted in success-critical situations such as self-driving cars, there needs to be a way for practitioners to understand what the network has learned.

- What is the optimal size of neural network for a given problem? — overparameterization increases the likelihood of good solutions on the training set, and may increase generalization [29–31]. However, these aren't guarantees and increasing sizes of networks have led to energy usage problems [86]. This question is relevant both for training a network and using it afterwards. Identifying ways of decreasing network size for training and potentially reducing network size before inference are increasingly important as network sizes increase.

Although the practical success of neural networks has outpaced theoretical development, there are a number of significant results in the theory. Much is known about what neural networks *can* do, but less is known about what neural networks *actually* do. Additionally, many of these theoretical results are not useful for engineering better neural networks or understanding how they work.

Much of the difficulty in analyzing neural networks comes from the difficulty of analyzing highly nonlinear functions on non-convex problems. Simplifications to make the associated theoretical problems tractable typically require impractical assumptions to be made as to the structure of datasets or the neural network. However, there is empirical evidence suggesting neural networks behave similarly to what is expected based on these impractical assumptions [29].

Chapter 3

Boolean Functions

Boolean functions have been used to investigate the properties of neural networks since their inception. Minsky and Papert used parity to demonstrate that simple perceptrons are not able to solve certain problems without exponentially large numbers of inputs [5, 6]. The theoretical behavior of deep networks using threshold activation functions has been studied in detail to determine upper and lower bounds on the size of networks necessary to represent Boolean functions [87]. Since the inception of deep learning with rectified linear units (ReLU) as activation functions, parity functions have been used as examples of problems that are hard to train. Explanations for this difficulty primarily focus on the problem's periodic behavior and poor gradient information [88, 89].

Despite this historical relevance, practical results are not emphasized in the literature, given the limited applications of neural networks that solve Boolean functions. However, the relative simplicity of the domain allows for a more in-depth investigation of behavior across all possible functions, since all possible Boolean functions of n input bits can be enumerated. Although the combinatorial complexity of this enumeration is high, with 2^{2^n} functions of n inputs, it allows us to identify specific problems neural network perform poorly on and analyze why. That simplicity also provides insight into optimal network sizes due to the reduced problem complexity, and for small numbers of inputs we can exhaustively investigate how network size affects training success rates. By looking at network behavior on Boolean problems, we can identify or confirm behavior that may be exploitable in some manner for problems of broader practical interest.

In Section 3.3 we construct a function for determining optimal one-dimensional mappings of Boolean functions and present an upper bound on the number of nodes necessary for a single-hidden layer Leaky-ReLU to solve a Boolean function. We show that width two or higher deep Leaky-ReLU networks can solve any Boolean function, and that on certain well-behaved functions, such as parity, for a fixed width the necessary depth grows only logarithmically in the number of inputs. We also investigate the local optima associated with training on a minimal parity problem

in one-dimension and discuss difficulties arising from symmetries and crossing of nonlinearities in the network.

In Section 3.4 we analyze how well minimal and near-minimal networks are able to train to solve Boolean functions. We start by providing an empirical study of the basins of attraction on the minimal parity problem in one-dimension. We compare ReLU and Leaky ReLU on functions of four inputs, and determine that ReLU's removal of information makes it ineffective for training minimal networks. We investigate how well various metrics on Boolean functions predict network performance, and determine that (1) the degree of influence of each input on the output is high for difficult functions and (2) function entropy grows in relevance to difficulty as the number of inputs increases.

3.1 Related Works

The relationship between neural networks and Boolean functions has been heavily studied. Minsky and Papert demonstrated that the simplest form of Rosenblatt's perceptron was not able to solve parity due to its linearity [5, 6]. It was later demonstrated for various network architectures, domains, and approximation methods that neural networks are universal approximators — for any function under certain constraints, a neural network can be constructed that approximates it arbitrarily closely [9, 24–26]. These theorems originally focused on single-hidden-layer networks, but more recent results have demonstrated that for deep ReLU networks, as long as at least some layers have width greater than or equal to the input dimension plus the output dimension, they too are universal approximators [24].

Further work investigated how the width or depth of a rectangular fully-connected neural network must increase as the depth or width, respectively, decreases. Specifically, there are width $O(k^2)$ shallow networks that cannot be approximated by any width $O(k^{1.5})$ and depth-k network [10] and there are depth k^2 networks of total size k^3 such that any network with at most k hidden layers will require at least $\frac{1}{2}k^{k+1} - 1$ total nodes [35]. From these, decreasing the width of a network requires polynomial increase in depth and decreasing the depth requires an exponen-

tial increase in width to maintain the set of piecewise linear functions that can be represented by a network.

The theoretical behavior of networks using various activation functions, most commonly threshold activation functions, has been studied in detail, with results focusing on networks with threshold activation functions [88]. For multilayer threshold neural networks, at least $\Omega(\sqrt{2^n/n})$ threshold units are required to correctly classify arbitrary Boolean functions of n inputs. For a single-hidden layer network, at least $\Omega(2^n/n)$ threshold units are required [87].

Similar results have been examined for networks consisting of Boolean gates. Shannon demonstrated that for $n > 2$, there are Boolean functions of n variables that require at least $2^n/n$ Boolean gates, and for large n, most functions require at least that many gates [90]. The minimal circuit size using various libraries of functions has been examined for small input dimensions, although the general form of this problem is NP-hard [91, 92]. It has been demonstrated that for Boolean functions of four inputs, ReLU neural network learning rate and the minimal formula length of a Boolean function using a specific library of Boolean operators are correlated [93].

Behavior of the coefficients of the Fourier polynomials of Boolean functions have relationships to the study of graph properties, learning theory, percolation theory, and information theory [94]. The Fourier-Entropy-Influence conjecture, specifically, has implications about efficient learning of decision trees in the agnostic model and the relationship between the number of terms of the disjunctive normal form of a function and the number of nonzero coefficients of the Fourier polynomial [95–97]. ReLU neural networks can represent the disjunctive normal form of a function with a single hidden layer containing a number of nodes equal to the number of conjunctions [98] and polynomials can be efficiently learned by neural networks given sufficient size [99].

Parity functions have been used as examples of problems that neural networks perform poorly on. Training using a subset of the elements of the domain typically results in poor performance due to networks favoring lower complexity solutions, and training using the entire dataset results in poor gradient information as the input dimension increases [89].

Finally, Mingard *et al.*show that when initializing a single-layer perceptron with n neurons, the likelihood that it will classify t points in $\{0,1\}^n$ as 1 is $P(t) = 2^{-n}$ for $0 \leq t \leq 2^n$, and there is a general bias towards lower entropy solutions which becomes stronger as more layers are added. Additionally, it can be shown that neural networks with input dimension n, l hidden layers each with width $n + 2^{n-1-\log_2 l} + 1$ can solve all Boolean functions of n variables [98].

3.2 Notation

We consider binary classification neural networks, $v : \mathbb{R}^n \to \mathbb{R}$ that are rectangular with l hidden layers each having width w; that is, each layer is of the form

$$f_1(x, A_1, b_1) = \text{LReLU}(A_1 x + b_1; \alpha) \tag{3.1}$$

$$f_i(x; A_i, b_i) = \text{LReLU}(A_i f_{i-1}(x) + b_i; \alpha), i = 2, ..., l \tag{3.2}$$

$$f_{l+1}(x; A_{l+1}, b_{l+1}) = A_{l+1}^T f_l(x) + b_{l+1} \tag{3.3}$$

with leaky ReLU, $\text{LReLU}(x) = \max(x, \alpha x), 0 \leq \alpha < 1$, $A_1 \in \mathbb{R}^{w \times n}$, $A_i \in \mathbb{R}^{w \times w}, i = 2, ..., l$, $b_i \in \mathbb{R}^w, i = 1, .., l$, $A_{l+1} \in \mathbb{R}^w$, and $b_{l+1} \in \mathbb{R}$. The network, v, is the composition of these layers,

$$v(x) = (f_{l+1} \circ f_l \circ f_{l-1} \circ ... \circ f_2 \circ f_1)(x). \tag{3.4}$$

For classification, the value of v is thresholded into positive or negative class.

We consider Boolean functions with input strings in $\{-1, 1\}^n$ mapping to $\{-1, 1\}$. We use the term Boolean function to refer to these functions with binary inputs and outputs, rather than using the term binary function, as binary functions are frequently used to refer to either functions of two inputs or neural network classification problems with two classes.

The Fourier-Walsh polynomial of a Boolean function has useful properties for analyzing the function's behavior. They can be constructed, for a function $f : \{-1, 1\}^n \to \{-1, 1\}$, as

$$f(x) = \sum_{S \subseteq [n]} \hat{f}(S)\chi_S(x), \qquad \chi_S(x) = \prod_{i \in S} x_i, \qquad \hat{f}(S) = \sum_{x \in \{-1,1\}^n} f(x)\chi_S, \tag{3.5}$$

with the values $\hat{f}(S)$ corresponding to the coefficients of the Fourier polynomial.

The influence of the kth coordinate, $I_k(f)$, is given by the probability that $f(x) \neq f(\mu_k(x))$ where x is uniformly distributed on $\{-1,1\}^n$ and $\mu_k(x) : \{-1,1\}^n \to \{-1,1\}^n$ is defined as flipping the kth coordinate of its input. The total influence is the sum of all the I_k, $I(f) = \sum_k I_k(f)$. The influence can also be calculated using the Fourier polynomial as

$$I(f) = \sum_{S \subseteq [n]} |S| \hat{f}(S)^2. \tag{3.6}$$

The entropy of the Fourier spectrum, which corresponds to Shannon entropy, is defined as

$$E(f) = \sum_{S \subseteq [n]} \hat{f}(s)^2 \log_2 \frac{1}{\hat{f}(s)^2}. \tag{3.7}$$

3.3 Theoretical Analysis

It is known from representational capacity theory that for broad classes of problems, there exist neural networks that approximate them arbitrarily well. However, there is less information about what the smallest possible neural network for a problem is. In general, this is a difficult question to answer — neural networks are complex, datasets are difficult to analyze, and since overparameterization increases training performance, experimentally trained networks are typically significantly overparameterized.

By narrowing our focus to only consider Boolean functions, we are able to use and extend existing representational capacity theory to make claims about the smallest network sizes that are able to successfully represent the function. We also are able to investigate small networks on low-dimensional Boolean functions to examine local behavior of the optima of the networks.

3.3.1 One-Dimensional Analysis and the Swaps Function

Boolean functions, despite having n inputs, can be linearly reduced to a single input. A simple method is to translate between the binary and decimal representations. This can be naturally exploited by a neural network, as this transformation is linear, and so neural networks are able to approximate Boolean functions using single-dimensional input and output. From the work of Hanin and Sellke, a problem with input dimension d_{in} and output dimension d_{out} can be solved by a neural network of width $d_{in} + d_{out}$ given sufficient depth [24]. This means that for any Boolean function there is a network with layers of at most width two (but potentially large depth) that is able to solve it.

This one-dimensional mapping is further convenient, as existing representational capacity work often uses one-dimensional analysis [10, 35]. The one-dimensional mapping allows for transfer of the insights from that work to Boolean classification. Specifically, when studying classification problems with one-dimensional input and two classes, the number of changes between the two classes along the number line determines the complexity of neural network necessary to solve the problem. We refer to this as the swaps value of a problem. For Boolean functions, this can be calculated by determining the linear mapping which minimizes the number of changes in class.

The Swaps Value for Boolean Functions

For the set of binary strings, $\{-1, 1\}^n$ and Boolean function $f : \{-1, 1\}^n \to \{-1, 1\}$, we can define the function swaps(f) which calculates the minimum number of changes in class when the inputs to f are mapped to a single dimension. To construct the swaps function, we can create the set of possible linear mappings to one-dimension such that they map the elements of $\{-1, 1\}^n$ to $\mathbb{R}$ with no overlap of points with different classes:

$$W = \{w \in \mathbb{R}^n | \forall c_1, c_2 \in \{-1, 1\}^n, c_1^\intercal w = c_2^\intercal w \implies f(c_1) = f(c_2)\}. \tag{3.8}$$

For $w \in W$ this yields the ordering

$$x_1^\intercal w \leq x_2^\intercal w \leq ... \leq x_{2^n}^\intercal w, \tag{3.9}$$

for $x_i \in \{-1, 1\}^n, i = 1, ..., 2^n$, The swaps value of f is then the minimal number of changes in class across the elements of W,

$$\text{swaps}(f) = \min_{w \in W} \sum_{i=1}^{2^n - 1} \frac{|f(x_{i+1}) - f(x_i)|}{2}. \tag{3.10}$$

This swaps function is invariant to negation-permutation-negation (NPN) operations — negation of an input digit (swapping the ith digit between -1 and 1 for all elements of $\{-1, 1\}^n$), permutation of the input digits, and negation of the output value.

Theorem 3.3.1. *For the set of binary strings, $\{-1, 1\}^n$ and Boolean function $f : \{-1, 1\}^n \rightarrow \{-1, 1\}$ with swaps value s, any function which belongs to the NPN class of f will have swaps value s.*

To prove this, we demonstrate that the swaps function is invariant to invertible affine transforms of the input and output, which includes NPN transformations. This follows from the fact that geometrically, the mapping that w performs is a projection to one-dimension. This also includes changing the true and false values of the function from -1 and 1 to other values. A full proof is available in Appendix A.1.

These NPN operations are relevant in determining when two Boolean functions can be represented equivalently [100]. As negation and permutation are linear operators, any network architecture that can represent one element of an NPN class can represent all of them through a linear transform of the first or last layer. Just as NPN operations do not change swaps values, for any network that can represent a Boolean function, for each of the elements of its NPN equivalency there is a network with equivalent architecture but different weight values that represents it.

For small numbers of inputs, it is possible to determine the swaps value by brute forcing all possible orderings resulting from linear mappings to one-dimension. However, a naive brute-force algorithm takes $\mathcal{O}((2^n n!)^2)$ operations and as such is not effective for higher numbers of inputs.

Since Boolean functions with swaps value s can be classified using a single-hidden-layer threshold network with s thresholds, and single-hidden-layer threshold networks require at least $\Omega(2^n/n)$ thresholds, the swaps function must grow at least as $\Omega(2^n/n)$ [87]. We obtain results for all four input functions, as discussed in Subsection 3.4.2 when contrasting the swaps value with ease of training.

An Algorithm for Calculating the Swaps Value for Boolean Functions

To be able to use the swaps value of a Boolean function to analyze it, we must be able to calculate it. One way of doing is by brute-forcing all possible orderings of the problem, and determining the ordering that yields a minimal value. This is computationally complex, but is relatively simple to construct and implement.

To calculate the swaps value of a function computationally we rescale the inputs so that we are investigating the set of Boolean input values $\{0,1\}^n$ rather than $\{-1,1\}^n$ as in the previous section. As discussed for Theorem 3.3.1, an affine transformation of the inputs of a Boolean function does not change the swaps value. The use of zero as the value for false is convenient, as then NPN operations of a given function will have at least one element of the NPN class with specific properties. Across all negations of the input, we need only consider elements of $w \in W$ that have positive values. Additionally, each element of the set of inputs is at some point the zero vector, and so will map to zero. Permuting the elements means that we need only consider $w \in W$ that are ordered — across all possible permutations of the input, there will exist at least one where the elements of its corresponding w are ordered:

$$0 \leq w_1 \leq w_2 \leq ... \leq w_n. \tag{3.11}$$

The elements of w can then be built iteratively —

- For w_1, cases are $w_1 = 0$ or $0 < w_1$
- For w_2, cases are $w_2 = w_1$ or $w_2 > w_1$

- For w_3, cases are $w_3 = w_2$ or $w_2 < w_3 < w_1 + w_2$ or $w_3 = w_1 + w_2$ or $w_3 > w_1 + w_2$

In general, for w_i there are $2i - 2$ cases for each previous choice of $w_1, ..., w_{i-1}$:

1. $w_i = w_{i-1}$
2. $w_{i-1} < w_i < w_{i-1} + w_1, i > 2$
3. $w_i = w_{i-1} + \sum_{j=1}^{k} w_j, k = 1, ..., i - 2$
4. $w_{i-1} + \sum_{j=1}^{k-1} w_j < w_i < w_{i-1} + \sum_{j=1}^{k} w_j, k = 2, ..., i - 2$
5. $w_i > w_{i-1} + \sum_{j=1}^{i-2} w_j$

This is $\mathcal{O}(2^n n!)$ orderings. Since calculating all NPN equivalencies takes $\mathcal{O}(2^n n!)$ operations, this gives that calculating the swaps value of a function using this method takes $\mathcal{O}((2^n n!)^2)$ operations. There are $\mathcal{O}\left(\frac{2^{2^n}}{2^n n!}\right)$ NPN classes, and so calculating swaps values for all functions of n inputs can be done in $\mathcal{O}(2^{2^n} 2^n n!)$ operations using this algorithm [101, Sequence A000370].

For small values of n, we can use this algorithm to identify the the behavior of the swaps values on boolean functions, as shown in Table 3.1. There are only two NPN classes for $n = 1$, which correspond to zero swaps for the constant case and one for the linear case. For $n = 2$ there are four NPN classes, with one constant, two linear (a split of one and three different outputs or two and two adjacent outputs) and the XOR function (which is also two-parity) corresponding to the two-swaps function. Examples of this are shown in Figure 3.1. For $n = 3$ there are a variety of ways of having zero, one, or two swaps. Unlike $n = 2$ only having one way to attain a swaps value of two, there are now multiple NPN classes that have swaps value three.

Table 3.1: The highest swaps values achieved across all $\{0,1\}^n \to \{0,1\}$ functions. Exact values are calculable for low values of n. The formula for n comes from [88] and is lower bound on the number of threshold functions needed to represent a Boolean function.

n	1	2	3	4	...	n
$\max_{g:\{0,1\}^n \to \{0,1\}} \text{swaps}(g)$	1	2	3	5	...	$\Omega(2^n/n)$

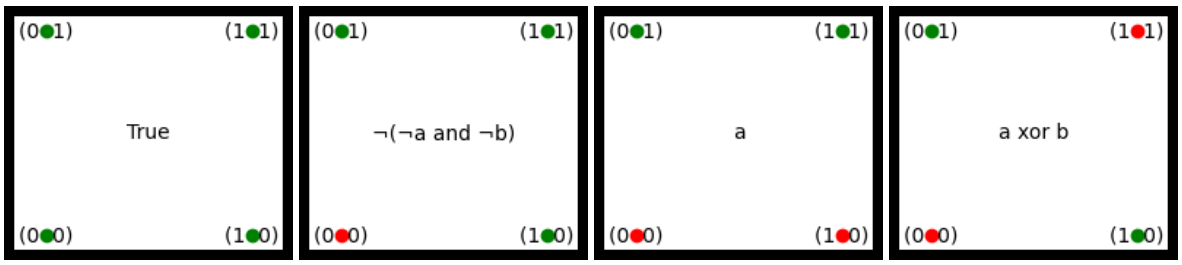


Figure 3.1: The four NPN classes of Boolean functions with two inputs.

The change from $n = 3$ having a maximum of swaps value three to $n = 4$ having a maximum of swaps value five is interesting and is the start of the exponential growth of the swaps value. An example function from each NPN class with swaps value five is shown in Figure 3.2. Certain functions have swaps values that are reduced from this exponential bound. For example, n-parity will always have swaps value n — the mapping vector of all ones maps points with k ones to the value k.

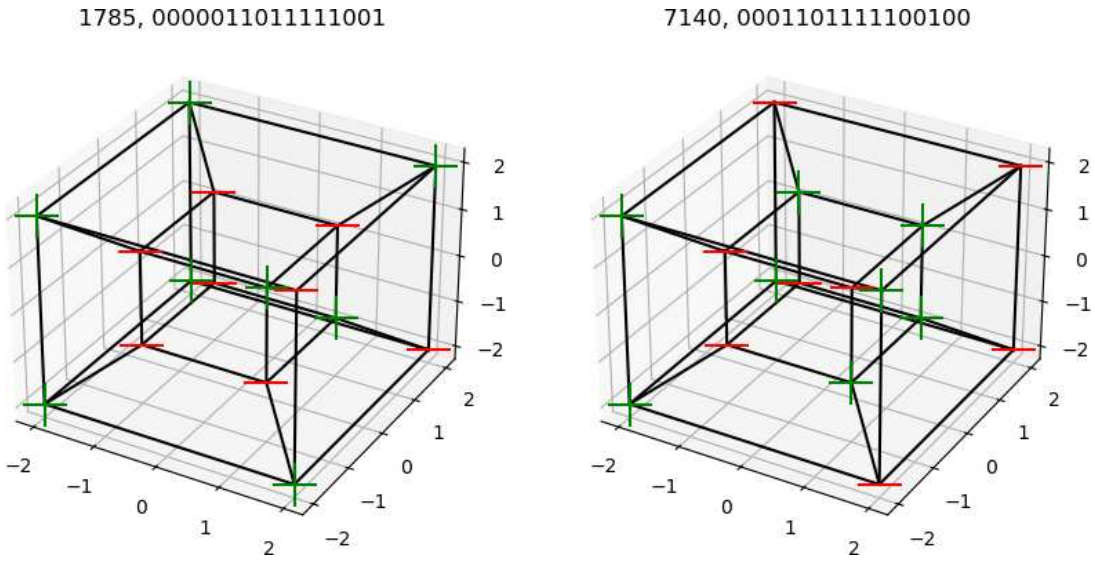


Figure 3.2: Tesseract representations of the Boolean functions 1,785 and 7,140. The NPN classes these functions belong to have swaps values five.

This analysis is sufficient for the applications in this work, but there are still still open questions:

1. Can the algorithm for computing swaps values be made more efficient?

2. What classes of problems have maximal swaps values?

3. How does the distribution of swaps values across all functions of n inputs change as n increases?

3.3.2 Representational Capacity

The swaps function allows us to construct upper bounds on the necessary size for a network. By transforming the problem to one dimension, determining the number of changes in class a network can attain along the number line allows us to determine an upper bound for the minimum network size to solve a specific problem. This will only be an upper bound, as it is a restriction to solving the problem in one-dimension when it is possible that a higher-dimensional solution may be achievable with a smaller network.

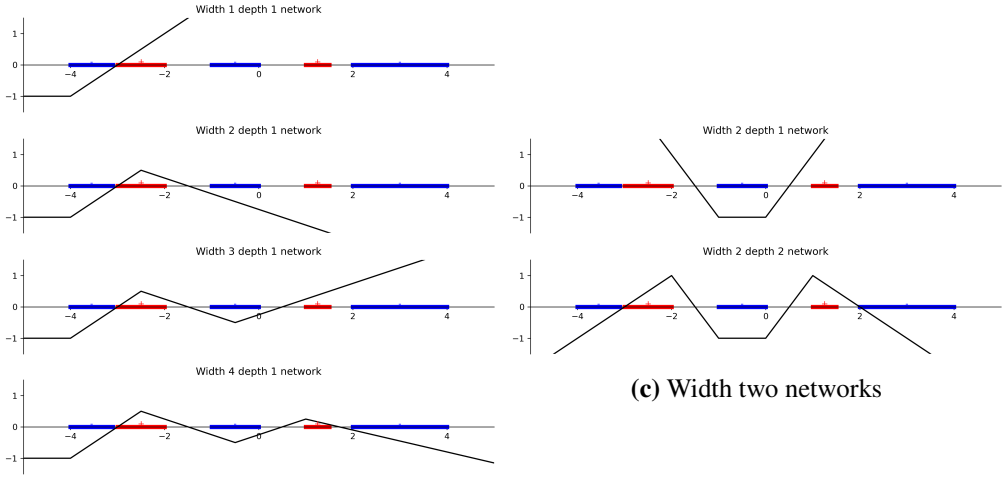
For wide leaky ReLU networks, the maximum number of changes in class is equal to one plus the number of nodes — each node can create an inflection point in the piecewise linear function. The position of these zeros is arbitrary, so any problem with swaps value s can be solved with $s-1$ nodes. This is an extension of existing representational capacity theorems for shallow networks on regression problems to Leaky ReLU and aligns with previous results for ReLU [10, 35].

Two-class classification in one-dimension is the problem of assigning positive values to some number of regions along that dimension. Specifically, for s regions along $\mathbb{R}$ each with starting point $x_{i,1}$ and ending points $x_{i,2} \geq x_{i,1}, i = 1, ..., s$ that do not overlap, so $x_{i-1,2} \leq x_{i,1}, i = 2, ..., s$ with alternating positive and negative classes, a classifier must have positive value in each region with a positive class and negative value in each region with a negative class to correctly classify each region. An example of how the geometry of this is constructed is in Figure 3.3b.

Theorem 3.3.2 (s-region classification by wide networks)**.** *Given s regions in $\mathbb{R}$, $(x_{i,1}, x_{i,2}), i = 1, ...s$ with $x_{i-1,2} \leq x_{i,1}$ with alternating positive and negative classes, there exists a Leaky ReLU neural network, $v : \mathbb{R} \to \mathbb{R}$ with a single hidden layer containing $s-1$ nodes that correctly classifies each region.*

This is demonstrated first by showing that regression on s distinct points requires a network with $s-1$ nodes, which can be done by having each node of the network control the slope going

(a) A one-dimensional problem



(b) Depth one networks

(c) Width two networks

Figure 3.3: (a) shows a simple one-dimensional two-class classification problem, with four changes in class. (b) demonstrates how increasing width of a shallow network can allow for classification. (c) demonstrates how increasing depth of a narrow network can allow for classification.

from point to point and the bias controlling the height at the first point. Classification can be turned into a regression problem, with points at the center of each classification region having sign corresponding to the class. The magnitude of the height of those points can then be set such that the neural network passes through zero between adjacent classes. This means that a Boolean function with swaps value s (which corresponds to $s + 1$ regions) can be classified with a width s single-hidden-layer network. A full version of this proof is available in Appendix A.2.

Deep networks are more complicated, as the requirement of one-dimensional input has a stronger effect when applied to all layers, so the associated theorem is weaker. An example of how the geometry of this is constructed is in Figure 3.3c.

Theorem 3.3.3 (Deep network representation of general classification)**.** *There exists a Leaky ReLU neural network, $v : \mathbb{R} \to \mathbb{R}$ with l hidden layers, each of width w, that partitions a subset of $\mathbb{R}$ into w^l alternating positive and negative regions.*

The full proof for this is available in Appendix A.3. This does not directly say that any set of s regions with alternating classes can be classified by a network of width w and depth l with $w^l > k$. As discussed in [10], there is exponential expressivity, but the number of parameters is only polynomial. There are restrictions on the structure of the network, and so not every classification problem can be represented. We hypothesize that Boolean functions are structured such that this does become a bound for deep networks, but proving such is beyond the scope of this paper. There are certain problems, such as parity, where there is a known structure to the one-dimensional map that allows for more bounds to be determined.

Theorem 3.3.4 (Deep networks representation of parity)**.** *Let v be a neural network with depth l and width w. Then, v can solve s-parity if $n \leq w^l$.*

The full proof for this is available in Appendix A.4. From these theorems, Boolean functions with swaps value s can be solved by networks with a single hidden layer containing $\mathcal{O}(s)$ nodes. Deep networks with width at least two can represent every Boolean function given sufficient depth. We hypothesize that such functions can be solved by deep networks of width w and $\mathcal{O}(\frac{\log(s)}{\log(w)})$ layers. For parity, this bound can be verified.

This bound is not tight, as the one-dimensional restriction for this analysis is a significant constraint. It is always possible to create a solution using one-dimensional input, but there may be more efficient constructions in higher dimensions. The lack of tightness here corresponds to previous work showing that threshold neural networks can solve n-parity with $\mathcal{O}(\sqrt{n})$ nodes [88]. The proof given here for single-hidden-layer networks could equally be applied to threshold activation functions, and so there is likely similarity in behavior between ReLU and threshold networks. In Section 3.4.2, we also show that there are Boolean functions with swaps value s that require width less than s to solve.

3.3.3 Analysis of Local Optima

In addition to determining whether a network *can* represent a given problem, it is of interest to determine if the network *will* train to a solution that successfully represents the problem. In

general, this is difficult — even a simple one-input network with a single hidden layer containing two nodes has seven parameters. That means that determining the trajectories of network weights requires investigating a complex, high-dimensional, nonlinear space.

However, for the simplest problems, this analysis is possible. By looking at the four NPN classes of Boolean functions with two inputs and the minimal network architecture that can successfully classify any of them, we can investigate the number and type of optima using symbolic solvers, such as SymPy [102]. Representative problems for these four classes are shown in Figure 3.4. This does not directly give the probability of training to a given minima, but it does provide information about behavior across different difficulties of problems.

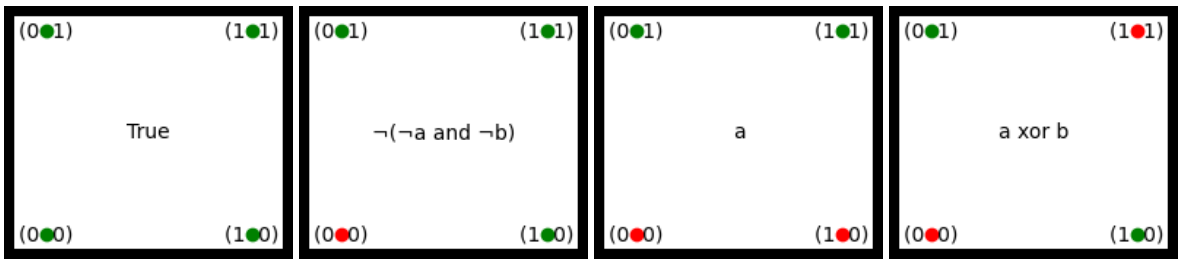


Figure 3.4: The four NPN classes of Boolean functions with two inputs.

As an example of how solutions can be found, consider the simple neural network with one input, one input, and two parameters,

$$v(x; a, b) = \max(ax, 0) + b, \tag{3.12}$$

on the problem which maps -1 to -1 and 1 to 1, so that the sum of squared error loss is, therefore,

$$L(a, b) = (-1 - v(-1; a, b))^2 + (1 - v(1; a, b))^2. \tag{3.13}$$

Identifying solutions to this is done by using the gradients of the loss with respect to the parameters,

$$\frac{\partial L}{\partial a} = 2\left(-b - \max\left(0, -a\right) - 1\right)\theta\left(-a\right) - 2\left(-b - \max\left(0, a\right) + 1\right)\theta\left(a\right) \tag{3.14}$$

and

$$\frac{\partial L}{\partial b} = 4b + 2\max\left(0, -a\right) + 2\max\left(0, a\right) \tag{3.15}$$

where θ is the step function defined as

$$\theta(x) = \begin{cases} 1, x > 0 \\ \\ 0, x \leq 0 \end{cases} \tag{3.16}$$

to align with PyTorch's implementation of ReLU's derivative.

These can equivalently be written using the cases $a \leq 0$ and $a > 0$ as

$$\frac{\partial L}{\partial a} = \begin{cases} 2a + 2b - 2, a > 0 \\ \\ 2a - 2b - 2, a \leq 0 \end{cases} \tag{3.17}$$

and

$$\frac{\partial L}{\partial b} = \begin{cases} 2a + 4b, a > 0 \\ \\ -2a + 4b, a \leq 0. \end{cases} \tag{3.18}$$

The network updates its weights as, for learning rate λ,

$$a_{t+\lambda} = a_t - \lambda\frac{\partial L}{\partial a} \tag{3.19}$$

$$b_{t+\lambda} = b_t - \lambda\frac{\partial L}{\partial a}. \tag{3.20}$$

Converting this difference equation to the continuous differential equation form gives

$$\frac{da}{dt} = -\frac{\partial L}{\partial a} \tag{3.21}$$

$$\frac{db}{dt} = -\frac{\partial L}{\partial b} \tag{3.22}$$

$$\tag{3.23}$$

Optima of the neural network occur when these differential equations are both equivalent to zero — $\frac{da}{dt} = 0$ and $\frac{db}{dt} = 0$. When $a > 0$, this can occur at $(a, b) = (2, -1)$. When $a \leq 0$, this can occur at $(a, b) = (-4, 2)$.

This can be extended to investigating XOR. XOR can be mapped to one dimension by taking its inputs at $(z_1, z_2) \in \{(-1, -1), (-1, 1), (1, -1)(1, 1)\}$ and projecting them using the projection $x = \frac{z_1+z_2}{2}$ to one dimension. The associated function then maps $x = -1$ and $x = 1$ to -1 and $x = 0$ to 1. Using this one-dimensional mapping with network

$$v(x; \Omega) = o_1 \max(w_1 x + b_1, 0) + o_2 \max(w_2 x + b_2, 0) + c, \tag{3.24}$$

the sum of squared error loss is, therefore,

$$L(\Omega) = (-1 - v(-1; \Omega))^2 + 2(1 - v(0; \Omega))^2 + (-1 - v(1; \Omega))^2. \tag{3.25}$$

Note that the coefficient of 2 on the center element of the loss corresponds to the fact that the two-dimensional elements $(-1, 1)$ and $(1, -1)$ both map to zero. Both are maintained in the loss, as balance between classes is frequently beneficial for training neural networks.

Taking the derivatives of the loss with respect to the weights and transforming it to a system of differential equations yields

$$
\begin{aligned}
\frac{dw_1}{dt} = \quad & 2o_1\left(-c - o_1\max\left(0, b_1 - w_1\right) - o_2\max\left(0, b_2 - w_2\right) - 1\right)\theta\left(b_1 - w_1\right) \\
& -2o_1\left(-c - o_1\max\left(0, b_1 + w_1\right) - o_2\max\left(0, b_2 + w_2\right) - 1\right)\theta\left(b_1 + w_1\right) \\
\frac{dw_2}{dt} = \quad & 2o_2\left(-c - o_1\max\left(0, b_1 - w_1\right) - o_2\max\left(0, b_2 - w_2\right) - 1\right)\theta\left(b_2 - w_2\right) \\
& -2o_2\left(-c - o_1\max\left(0, b_1 + w_1\right) - o_2\max\left(0, b_2 + w_2\right) - 1\right)\theta\left(b_2 + w_2\right) \\
\frac{do_1}{dt} = \quad & -4\left(-c - o_1\max\left(0, b_1\right) - o_2\max\left(0, b_2\right) + 1\right)\max\left(0, b_1\right) \\
& -2\left(-c - o_1\max\left(0, b_1 - w_1\right) - o_2\max\left(0, b_2 - w_2\right) - 1\right)\max\left(0, b_1 - w_1\right) \\
& -2\left(-c - o_1\max\left(0, b_1 + w_1\right) - o_2\max\left(0, b_2 + w_2\right) - 1\right)\max\left(0, b_1 + w_1\right) \\
\frac{do_2}{dt} = \quad & -4\left(-c - o_1\max\left(0, b_1\right) - o_2\max\left(0, b_2\right) + 1\right)\max\left(0, b_2\right) \\
& -2\left(-c - o_1\max\left(0, b_1 - w_1\right) - o_2\max\left(0, b_2 - w_2\right) - 1\right)\max\left(0, b_2 - w_2\right) \\
& -2\left(-c - o_1\max\left(0, b_1 + w_1\right) - o_2\max\left(0, b_2 + w_2\right) - 1\right)\max\left(0, b_2 + w_2\right) \\
\frac{db_1}{dt} = \quad & -4o_1\left(-c - o_1\max\left(0, b_1\right) - o_2\max\left(0, b_2\right) + 1\right)\theta\left(b_1\right) \\
& -2o_1\left(-c - o_1\max\left(0, b_1 - w_1\right) - o_2\max\left(0, b_2 - w_2\right) - 1\right)\theta\left(b_1 - w_1\right) \\
& -2o_1\left(-c - o_1\max\left(0, b_1 + w_1\right) - o_2\max\left(0, b_2 + w_2\right) - 1\right)\theta\left(b_1 + w_1\right) \\
\frac{db_2}{dt} = \quad & -4o_2\left(-c - o_1\max\left(0, b_1\right) - o_2\max\left(0, b_2\right) + 1\right)\theta\left(b_2\right) \\
& -2o_2\left(-c - o_1\max\left(0, b_1 - w_1\right) - o_2\max\left(0, b_2 - w_2\right) - 1\right)\theta\left(b_2 - w_2\right) \\
& -2o_2\left(-c - o_1\max\left(0, b_1 + w_1\right) - o_2\max\left(0, b_2 + w_2\right) - 1\right)\theta\left(b_2 + w_2\right) \\
\frac{dc}{dt} = \quad & 8c + 4o_1\max\left(0, b_1\right) + 2o_1\max\left(0, b_1 - w_1\right) + 2o_1\max\left(0, b_1 + w_1\right) \\
& +4o_2\max\left(0, b_2\right) + 2o_2\max\left(0, b_2 - w_2\right) + 2o_2\max\left(0, b_2 + w_2\right)
\end{aligned}
$$

This has 64 cases, corresponding to the signs of b_1, $b_1 + w_1$, $b_1 - w_1$, b_2, $b_2 + w_2$, and $b_2 - w_2$, although only 36 of those cases are possible, as for $b \leq 0$ both $b + w > 0$ and $b - w > 0$ cannot both be true, and for $b > 0$, both $b + w \leq 0$ and $b - w \leq 0$ cannot both be true. Within one of the regions defined by those cases, these equations are multiquadratic (no monomial has any individual variable with degree higher than two), and as such solving for where the system is zero

only requires solving a multiquadratic function. For example, the region where b_1, b_1+w_1, b_1-w_1, b_2, b_2+w_2, and b_2-w_2 are all positive simplifies the differential equation into

$$\begin{aligned}
\frac{dw_1}{dt} &= 4o_1\left(o_1w_1+o_2w_2\right)\\
\frac{dw_2}{dt} &= 4o_2\left(o_1w_1+o_2w_2\right)\\
\frac{do_1}{dt} &= 8b_1^2o_1+8b_1b_2o_2+8b_1c+4o_1w_1^2+4o_2w_1w_2\\
\frac{do_2}{dt} &= 8b_1b_2o_1+8b_2^2o_2+8b_2c+4o_1w_1w_2+4o_2w_2^2\\
\frac{db_1}{dt} &= 8o_1\left(b_1o_1+b_2o_2+c\right)\\
\frac{db_2}{dt} &= 8o_2\left(b_1o_1+b_2o_2+c\right)\\
\frac{dc}{dt} &= 8b_1o_1+8b_2o_2+8c.
\end{aligned}$$

In this region, there are three steady state regions. The first requires $c=-b_1o_1-b_2o_2$ and $w_1=-\frac{o_2w_2}{o_1}$, the second requires $c=0$, $o_1=0$, and $o_2=0$, and the third requires $c=-b_2o_2$, $o_1=0$, and $w_2=0$. Each of these regions has loss four, corresponding to the network having a value of zero at each of the points in the dataset.

There is a significant degree of symmetry in optima of this system. The order of nodes in the network does not matter, and so swapping the parameters o_1, w_1, and b_1 with o_2, w_2, and b_2 does not result in meaningfully different solutions. The dataset is symmetric about $x=0$, and so changing the sign of w_1 or w_2 does not result in meaningfully different solutions. This means that it is common for optima to have a class of eight "equivalent" optima that have the same behavior. The scale invariance of transferring magnitude between the inner parameters of a ReLU and the exterior parameter is also well known, although that will always create regions of optima rather than potentially creating distinct optima that are the same under symmetries.

Additionally, optima may exist that exist across multiple of the 36 piecewise linear regions. Their symbolic form may change as different samples have different activation patterns in each of the regions, and so determining if two solutions are of the same form is further nontrivial.

These symmetries also complicate the comparison of trained solutions to problems — for a given solution there are only up to seven more solutions in its equivalence class, but that number combinatorially increases as more layers or nodes are added. Any similarity metric must therefore account at least for permutations of nodes within a layer.

This analysis also provides limited information about the basin of attraction of local minima. Knowing that solutions exist does not give information about how likely the network is to train to any given solution. To account for this, Section 3.4.1 provides results for training 1,000 networks on this problem and determining which piecewise region the resulting solution lies on and which minima the network has trained to.

3.4 Application

In the previous section, we determined expected theoretical behavior for minimal network sizes. However, it is not clear if training algorithms will reliably converge to correct minimally-sized solutions. We investigate the training behavior of small networks for four input Boolean functions in Subsection 3.4.2. We also look at the influence, Fourier degree, and Fourier entropy of Boolean functions to determine their relation to problem difficulty.

In Subsection 3.4.3 we look at higher-dimensional problems. Four bit Boolean functions are very small, so we also experiment with ten bit Boolean networks, which although still small, are big enough to exhibit phase changes in behavior relative to four bit functions.

We use PyTorch to run all experiments [103]. In all cases we use networks with constant width hidden layers and activation function leaky ReLU. We use leak factors of 0.1 everywhere except in Subsection 3.4.2 where we compare ReLU to leaky ReLU. We use the Adam optimizer with learning rate 0.025 with up to 4,000 epochs and logit-based binary cross entropy as the loss function. Weights are initialized from the He-Kaiming normal distribution [104].

3.4.1 Experimental Analysis of Local Optima

The analysis of local optima in Section 3.3.3 is a good start for understanding the behavior of neural networks, but it has difficulty addressing the likelihood of any set of initial starting weights training to a given solution. Instead of continuing the theoretical analysis, instead investigating experimentally which initial conditions train to which solutions provides a way of understanding the basins of attraction of each minima.

This analysis is complicated by the fact that neural networks do not necessarily train to an exact theoretical minima. Many additions to training algorithms are specifically designed *not* to follow the path of the differential equation exactly, as doing so can be slow. Adding stochasticity to gradient descent through batches transforms the loss function for each batch, momentum assists in avoiding poor local behavior, and optimizing only subsets of the parameters in each step can avoid saddle points. We focus specifically on gradient descent without momentum, as it provides an analysis that best matches the theoretical results.

Even in the case that we use gradient descent, it is not necessarily true that a network will converge to an exact minima. The standard gradient descent algorithm has, for learning rate λ, $\mathcal{O}(\lambda)$ error. This means that solutions may not exactly follow trajectories to minima that would be suggested by theory (although, as discussed above, this is less relevant to the field of neural networks). More relevantly, the training algorithm is unlikely to reach a minima and stop, instead moving about a small region around the local minima, as the training steps overestimate the distance to the minima. This means that matching theoretical solutions must be done to within a certain tolerance. This holds both for identifying a specific solution, and for determining which parameter regime the network is in — for solutions near the boundary $b_i = 0$, a trained network may move between positive and negative values for b_i, even if only one of those regions has a minima nearby.

We train networks of the forms discussed in the previous section, $v(x; w, o, b, c) = o_1 \max(w_1 x + b_1) + o_2 \max(w_2 x + b_2) + c$. Networks are trained using gradient descent on the four training points, with a learning rate of $\lambda = 0.1$, and for 500 epochs.

There are three classes of minima for the single input networks — those with loss 0, $\frac{8}{3}$, or 4. There are saddle points with equivalent loss to the poor local minima. Examples of the minima are given in Table 3.2. Across 1,000 networks, this shows the 15 most common solutions in symbolic form, their associated loss, and the number of times it was found by trained networks. There were a total of 53 minima found by the 1,000 networks.

Solutions are not points — they are typically high-dimensional, with multiple free variables corresponding to freedom in scale. Scaling w_i and b_i by a and w_o by $\frac{1}{a}$ does not change the output of the network, so there are, at minimum, two degrees of freedom corresponding to this scale invariance.

The analysis and experimentation here is an initial exploration of what is possible. However, there are a number of complications that make this line of research unappealing. Even for the simple problem investigated here, it is computationally expensive to determine the symbolic optima. Identifying optima that belong to the same "class" is difficult due to the number of symmetries that arise from the structure of the network and problem and the fact that solutions of the same class in adjacent regions may not have the same symbolic form. Investigation of basins of attraction still requires training networks and determining which solution they identify.

3.4.2 Training Comparison Across Four-Dimensional NPN Classes

To investigate the training difficulty of small networks, we use four-digit Boolean functions. NPN operations can all be represented by linear transforms, so with careful network initialization it is only necessary to consider one element from each class. To ensure that training is invariant to the NPN operations, we use inputs drawn from $\{-1, 1\}^n$ and weights initialized using a normal distribution with mean zero. By using weight initialization distributions that are symmetric about zero, negating one dimension of the input will maintain the distribution of the product of the input and the initial weight matrix after the transform. This means that expected network performance will be invariant to negation of the input. Similarly, negation of the outputs only changes the class labels. By using loss functions that are not biased towards one class, such as binary cross entropy

Table 3.2: Solutions for networks of the form $v(x; w, o, b, c) = o_1 \max(w_1 x + b_1, 0) + o_2 \max(w_2 x + b_2, 0) + c$ when attempting to classify the points $0, 0, 1$ and -1 to match the function $f(0) = 1, f(-1) = -1, f(1) = -1$. Zero is included twice to balance the classes. All steady states of the network are found, then 1,000 networks of this form are trained and the count of how many find each solution is provided. Only the 15 most common solutions are provided out of the 53 found

loss	solution	times found in 1,000 trials
0	$b_1 < 0, b_1 + w_1 < 0, b_1 - w_1 \geq 0, b_2 < 0, b_2 + w_2 \geq 0, b_2 - w_2 < 0,$ $b_1 = w_1 - \frac{2}{o_1}, b_2 = -w_2 - \frac{2}{o_2}, c = 1$	23
0	$b_1 < 0, b_1 + w_1 < 0, b_1 - w_1 \geq 0, b_2 < 0, b_2 + w_2 \geq 0, b_2 - w_2 < 0,$ $b_1 = w_1 - \frac{2}{o_1}, b_2 = -w_2 - \frac{2}{o_2}, c = 1$	25
0	$b_1 < 0, b_1 + w_1 < 0, b_1 - w_1 \geq 0, b_2 \geq 0, b_2 + w_2 \geq 0, b_2 - w_2 < 0,$ $b_1 = \frac{-c+o_1 w_1 - 1}{o_1}, b_2 = \frac{1-c}{o_2}, w_2 = -\frac{2}{o_2}$	26
4	$b_1 \geq 0, b_1 + w_1 \geq 0, b_1 - w_1 \geq 0, b_2 < 0, b_2 + w_2 < 0, b_2 - w_2 < 0,$ $c = -b_1 o_1, w_1 = 0$	26
0	$b_1 < 0, b_1 + w_1 \geq 0, b_1 - w_1 < 0, b_2 \geq 0, b_2 + w_2 \geq 0, b_2 - w_2 < 0,$ $b_1 = \frac{-o_1 w_1 - o_2 w_2 - 2}{o_1}, b_2 = \frac{2}{o_2}, c = -1$	26
4	$b_1 < 0, b_1 + w_1 < 0, b_1 - w_1 < 0, b_2 \geq 0, b_2 + w_2 \geq 0, b_2 - w_2 \geq 0,$ $c = -b_2 o_2, w_2 = 0$	26
0	$b_1 < 0, b_1 + w_1 \geq 0, b_1 - w_1 < 0, b_2 < 0, b_2 + w_2 < 0, b_2 - w_2 \geq 0,$ $b_1 = -w_1 - \frac{2}{o_1}, b_2 = w_2 - \frac{2}{o_2}, c = 1$	31
4	$b_1 \geq 0, b_1 + w_1 \geq 0, b_1 - w_1 \geq 0, b_2 \geq 0, b_2 + w_2 \geq 0, b_2 - w_2 \geq 0,$ $c = -b_1 o_1 - b_2 o_2, w_1 = -\frac{o_2 w_2}{o_1}$	33
8/3	$b_1 < 0, b_1 + w_1 < 0, b_1 - w_1 < 0, b_2 < 0, b_2 + w_2 < 0, b_2 - w_2 \geq 0,$ $b_2 = w_2 - \frac{4}{3 o_2}, c = \frac{1}{3}$	58
8/3	$b_1 < 0, b_1 + w_1 < 0, b_1 - w_1 \geq 0, b_2 < 0, b_2 + w_2 < 0, b_2 - w_2 < 0,$ $b_1 = w_1 - \frac{4}{3 o_1}, c = \frac{1}{3}$	61
8/3	$b_1 < 0, b_1 + w_1 \geq 0, b_1 - w_1 < 0, b_2 < 0, b_2 + w_2 \geq 0, b_2 - w_2 < 0,$ $b_1 = \frac{-b_2 o_2 - o_1 w_1 - o_2 w_2 - \frac{4}{3}}{o_1}, c = \frac{1}{3}$	62
4	$b_1 < 0, b_1 + w_1 < 0, b_1 - w_1 < 0, b_2 < 0, b_2 + w_2 < 0, b_2 - w_2 < 0,$ $c = 0$	64
8/3	$b_1 < 0, b_1 + w_1 < 0, b_1 - w_1 \geq 0, b_2 < 0, b_2 + w_2 < 0, b_2 - w_2 \geq 0,$ $b_1 = \frac{-b_2 o_2 + o_1 w_1 + o_2 w_2 - \frac{4}{3}}{o_1}, c = \frac{1}{3}$	66
8/3	$b_1 < 0, b_1 + w_1 < 0, b_1 - w_1 < 0, b_2 < 0, b_2 + w_2 \geq 0, b_2 - w_2 < 0,$ $b_2 = -w_2 - \frac{4}{3 o_2}, c = \frac{1}{3}$	67
8/3	$b_1 < 0, b_1 + w_1 \geq 0, b_1 - w_1 < 0, b_2 < 0, b_2 + w_2 < 0, b_2 - w_2 < 0,$ $b_1 = -w_1 - \frac{4}{3 o_1}, c = \frac{1}{3}$	75

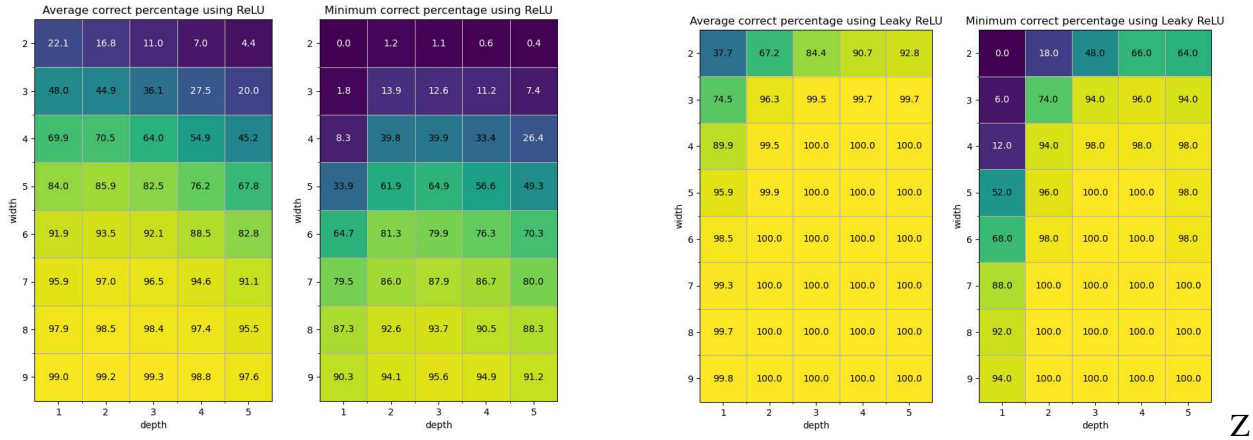


Figure 3.5: Networks trained for each combination of widths between two and ten and one to five hidden layers on all 4-input NPN classes. The left two images show results using ReLU and the right two images show results using Leaky ReLU. In both sets, the left image shows the average number of successes over 100 trials across all 222 problems for each size of network. The right image shows the number of successes for problem with the lowest success rate for each size of network.

with logits, and weight initializations that are symmetric about zero for the final layer, expected network performance will be invariant to negations of the output. Permutation of the inputs is already accounted for in the invariance of the network to permutations of inputs. By structuring the networks this way, each Boolean function within an NPN class will have equivalent expected behavior.

We choose four-input problems to maintain computational feasibility, as there are 222 NPN classes for four inputs and 616,126 for five inputs. Additionally, the swaps algorithm detailed in Section 3.3.1 is computationally infeasible for detailing the behavior of all five-input functions.

We train networks with multiple shapes on instances of each of the 222 problem classes. We consider rectangular networks with width w and depth l for all values of w from two to ten and l from one to five. For each network shape, we train 100 trials of each network architecture. The training is stochastic with regard to initialization, so this provides an overview of possible behavior. We consider leak factors of $\alpha = 0$ (ReLU) and $\alpha = 0.1$. Results of the average and worst-case success rates of classification rate across all 222 problems over ten trials for ReLU and leaky ReLU are shown in Figure 3.5.

For ReLU, the minimal networks perform poorly, with networks with width less than four rarely finding success. Single-hidden-layer networks only succeed at least once on every problem if they have widths of five or more. No width two network solves every problem at least once, although width three depth two networks can. Single hidden layer networks perform worse than networks with two hidden layers, and success rates on the hardest problems are significantly increased by adding a second hidden layer.

Success rates are higher for leaky ReLU. The minimal width three depth one and width two depth two networks perform poorly, but have nonzero success rates on all problems. Slight increases in size beyond those reach high success rates. Additionally, we see better performance of wide networks than predicted by the swaps analysis — width three depth one networks are able to solve every problem despite the existence of Boolean functions with swaps value five. The fact that overparameterized networks succeed significantly more frequently than minimal networks matches existing theoretical work, but the fact that only adding a single node per layer or moving to two hidden layers increases success rates so significantly is surprising [29–31].

Table 3.3 shows the ten hardest problems across all sizes of network. ReLU has a number of functions that show up multiple times, but Leaky ReLU has a significant dropoff after parity (27030). ReLU's poor performance likely comes from the fact that it removes information. Nodes of the network can initialize such that they have negative arguments for every input to the network, and so do not receive a training signal. When networks are sufficiently wide, this effect is mitigated each layer has sufficient nodes to ensure information is passed through, but when a minimal network has a node removed it will no longer be able to succeed on the problem. Many of the hard problems for ReLU do not show up for leaky ReLU and are unbalanced, suggesting that the loss of information resulting from poor initializations can cause issues when given unbalanced input. Including a leak factor significantly improves performance, as it entirely avoids this problem.

We investigate the relationship between network performance, swaps value, Fourier degree, Fourier entropy, and influence. Fourier degree, Fourier entropy, and influence are invariant to NPN operations due to the behavior of NPN operations as reparameterization of the original function,

Table 3.3: The ten hardest problems across all network sizes for ReLU and Leaky ReLU networks. The binary column corresponds to the function values when the inputs are in lexicographical order, with the decimal column corresponding to the decimal representation of the binary value. For each size of network for widths between two and ten and depths between one and five, all networks with success rates within one percentage point of the lowest performance problem are counted as the hardest for that size, and then the count over all network sizes is presented as the count in the tables. Results are reported for both ReLU and Leaky ReLU. Problem 27030 corresponds to parity.

ReLU		
decimal	binary	count
7140	00011011 11100100	4
1716	00000110 10110100	4
5804	00010110 10101100	4
6042	00010111 10011010	4
1641	00000110 01101001	5
5805	00010110 10101101	5
406	00000001 10010110	6
5737	00010110 01101001	12
278	00000001 00010110	14
27030	01101001 10010110	19

Leaky ReLU		
decimal	binary	count
2022	00000111 11100110	2
1713	00000110 10110001	2
5786	00010110 10011010	2
1717	00000110 10110101	2
5774	00010110 10001110	2
7905	00011110 11100001	2
5737	00010110 01101001	2
6038	00010111 10010110	2
7128	00011011 11011000	5
27030	01101001 10010110	12

as discussed in more detail in Appendix A.5. The relationship between swaps value and network performance is of interest to determine to what extent it truly corresponds both to minimal network size and difficulty of training. Fourier degree and influence are metrics that relate to the degree to which the inputs influence the function, and Fourier entropy relates to the complexity of the function. Their relationship to network performance is of interest in determining which classes of function are difficult and why.

Figure 3.6 shows these metrics, width two depth two network performance, width three depth one network performance, and average network performance across all sizes for the 222 four-input NPN classes. The three performance measures are similar, but there is variance between them, especially with the harder problems on the width two depth two network. Hard problems have high Fourier degree and swaps value, but easy problems can have high values as well. Entropy has a weak relationship with performance. Influence is inversely correlated with difficulty. This suggests that Fourier degree and a high swaps value are minimal requirements for the hardest problems, but influence is more explanatory in general.

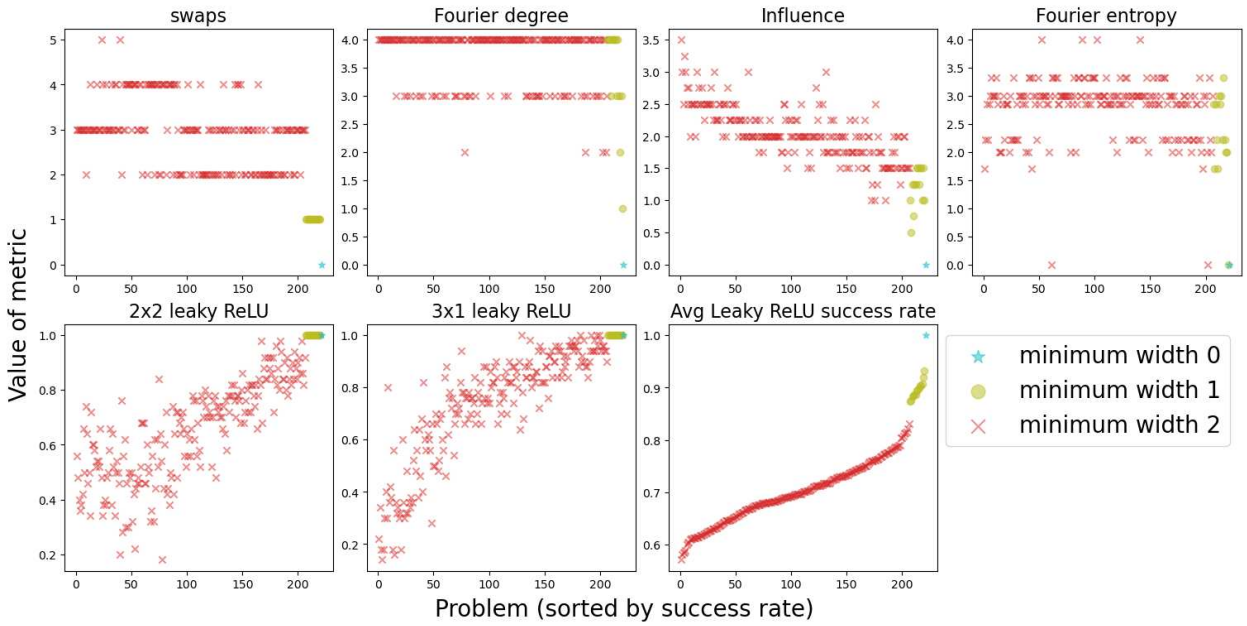


Figure 3.6: The swaps value, Fourier degree, influence, Fourier entropy, 2x2 network performance, 3x1 network performance, and average network performance across all sizes for the 222 four-input NPN classes. Results are sorted by the average network performance, and colors correspond to the minimum width of single-hidden-layer networks that achieves success on that problem.

3.4.3 Higher Dimensions

Four-input problems are useful for examining all possible functions, but four is a very small number for Boolean functions. We look at ten input functions in detail, and aspects of five through nine input functions to determine if the results for four-input functions are representative of broader Boolean function behavior. Due to the difficulty of identifying meaningful functions, we consider three classes of function: random functions, n-parity, and four-input functions XORed with n-parity. This provides insight into average behavior (random functions), a problem known to be difficult for neural networks (parity), and structured problems likely to be similar to parity (XORed problems).

Results comparing 10-parity to random 10-input functions are in Figure 3.7. 10-parity has performance very close to what is predicted by the swaps analysis, with minimal networks performing poorly but success rates increasing as size increases beyond that. 10-input random functions, however, have significantly lower performance. Parity is a highly structured problem, and despite its

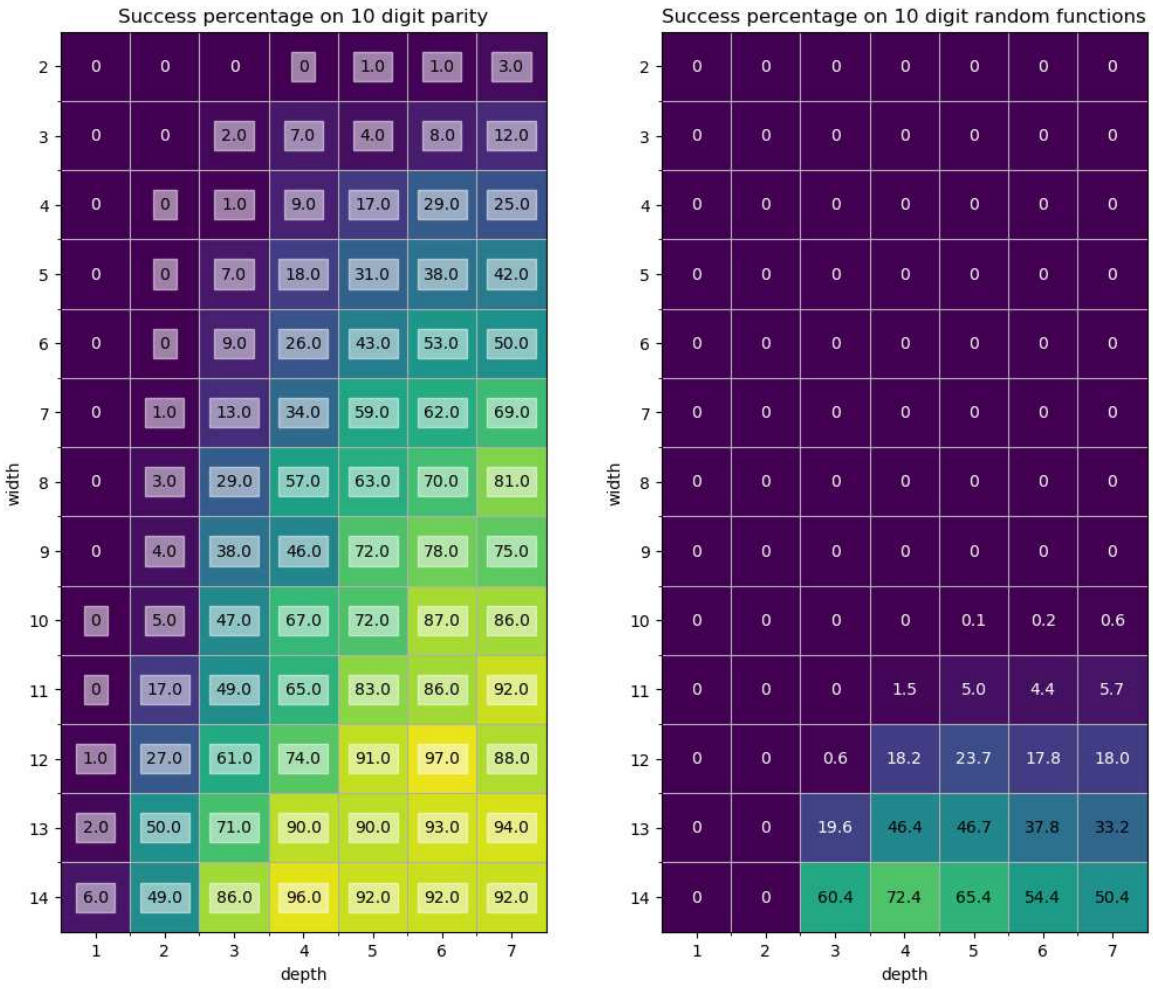


Figure 3.7: Comparison of success rates for 10-parity and random 10-input functions. For 10-parity, numbers with white outlines are those predicted to be able to solve the problem by the swaps analysis. Results are over 100 trials and ten random functions have their performance averaged.

performance as the hardest function in lower dimensions, higher-dimensional problems appear to have sufficient complexity to allow for it to be dethroned.

Comparison of network performances with Fourier degree, Fourier entropy and influence are in Figure 3.8. Each class of problem has significantly different performance for influence — parity and the XORed problems have high influence (the XOR problems are closely related to parity, which has maximal influence), and random functions have middling influence. However, the random functions are significantly harder than the other classes of function. Both random functions and the XORed problems have high Fourier degree, with some easier XORed problems having

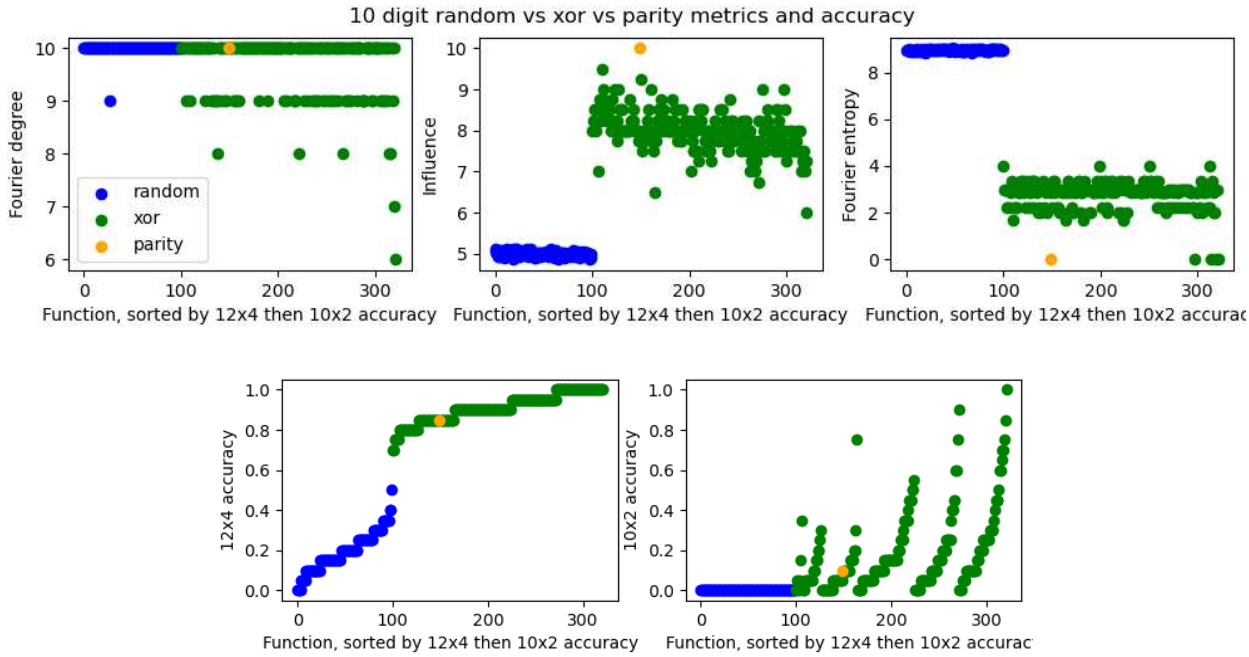


Figure 3.8: A comparison of network performance and Fourier degree, Fourier entropy and influence for 10-input functions. Network performance here is not measured in success rate, but instead by number of points successfully classified by the network out of 2^{10}. Results are over 20 trials for each of 100 random, 222 XOR, and one parity functions.

lower degrees. Random functions have high entropy while the XORed problems have reduced entropy. This again suggests that influence and degree are minimal requirements for a problem to be difficult, with entropy better explaining difficulty given sufficient influence and degree.

Comparisons of average network performances with average Fourier degree, Fourier entropy and influence for networks with 5-10 inputs are in Figure 3.9. These metrics are scaled by their maximum for each digit, which for all three is n. This is achieved for Fourier degree on a number of functions, Fourier entropy is maximized on OR functions, and Fourier influence is maximized on n-parity.

Random functions and parity have constant behavior for degree and influence, with the XORed functions increasing as dimension increases. As dimension increases, the degree to which parity overwhelms the four-input functions will increase, so the change in those is expected. Random functions and parity will have similar expected degree and influence. However, the entropy of random functions increases as the dimension increases, and the difficulty of random functions

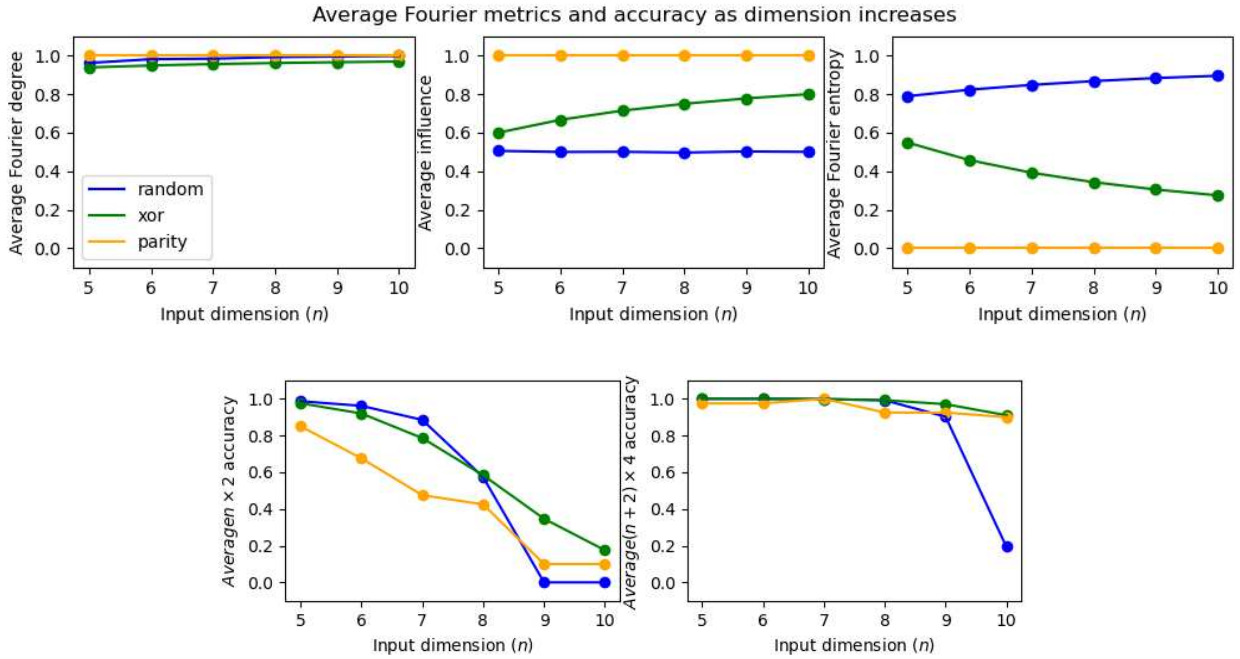


Figure 3.9: Comparison of network success rates and Fourier metrics for 5-10 dimensional inputs. Influence, degree, and entropy are normalized by the input dimension. Results are over 40 trials for each of 100 random, 222 XOR, and one parity functions for each input dimension.

increases significantly faster than the other functions. Entropy is not sufficient for a problem to be difficult — the NPN class of OR functions has maximum entropy, and those are solvable by a linear classifier.

3.4.4 Pruning Overparameterized Networks

Training with minimal networks using ReLU does not lead to success. The previous sections focused on the useage of leaky ReLU to avoid this issue, but there are alternatives. Specifically, we can investigate minimal networks able to represent a problem by training an overparameterized network and then pruning structurally down until the network no longer represents the problem. By determining nodes in the network that can be removed without changing the behavior of the network, we can identify subnetworks that are smaller than the original while maintaining 100% classification accuracy.

We investigate two methods for doing this — an exhaustive method for wide networks and a heuristic-based method for deep networks. For wide networks,

1. Train a width w network on $n \leq w$ dimensional parity

2. Then for each $i = 1, ..., w$

    - Construct all width i subnetworks of w
    - Fine-tune each subnetwork by training a fixed number of epochs using gradient descent
    - If there are subnetworks that succeed, count how many and exit
    - Otherwise increase i and repeat

This method allows us to determine exhaustively determine the smallest subnetwork that still represents n-dimensional parity and count how many such subnetworks exists. This method is combinatorially difficult — there are $\binom{w}{k}$ subnetworks of size k in a width w network, which increases significantly as input dimension and network width increase. Results are shown in Figure 3.4. In all cases the pruned size is close to the upper bound on minimal network size found, although it is not exact to that bound and the gap increases for higher dimensional parity. In many cases there are multiple subnetworks that maintain strong performance.

For pruning deep networks,

1. For a given input dimension, n,

2. Train an overparameterized network

3. For each node in the network

    - Create the subnetwork that results after removing that node
    - Fine-tune the subnetwork by training a fixed number of epochs using gradient descent
    - If the subnetwork achieves 100% classification accuracy, assign this node a score

4. If there are nodes that achieved 100% classification accuracy when removed, choose one using the scores and remove it, otherwise exit

5. Loop to 3 and continue from the resulting subnetwork

Table 3.4: Pruning results for wide, shallow networks. Rows correspond to the initial width of the trained network, and columns denote the number of input digits the pruning takes place on. Results are formatted as $p, r/\binom{w}{p}$ where p is the pruned width, r is the number of ways that pruning the original network results in a width p network, w is the original width, and $\binom{w}{p}$ is the number of possible ways of creating width p networks from the original network. Dashes correspond to elements where no network of width w was found over four trials that represents n-dimensional parity.

		Digits					
		1	2	3	4	5	6
Original Width	2	1, 1/2	-	-	-	-	-
	3	1, 2/3	2, 3/3	3, 1/1	3, 1/1	-	-
	4	1, 2/4	2, 3/6	3, 3/4	4, 1/1	-	-
	5	1, 3/5	2, 8/10	3, 8/10	4, 3/5	-	-
	6	1, 3/6	2, 4/15	3, 18/20	3, 1/20	5, 1/6	6, 1/1
	7	1, 4/7	2, 8/21	3, 26/35	3, 3/35	5, 3/21	6, 1/7
	8	1, 5/8	2, 5/28	3, 41/56	3, 5/56	5, 4/56	6, 2/28
	9	1, 7/9	2, 8/36	3, 66/84	3, 3/84	5, 4/126	5, 1/128
	10	1, 5/10	2, 8/45	3, 95/120	3, 4/120	4, 1/210	5, 2/252
	11	1, 6/11	2, 13/55	3, 143/165	3, 4/165	4, 1/330	5, 1/462
	12	1, 5/12	2, 24/66	3, 166/220	3, 9/220	4, 2/495	5, 2/792
	13	1, 3/13	2, 12/78	3, 245/286	3, 5/286	4, 5/715	5, 4/1287
	14	1, 5/14	2, 25/91	3, 278/364	3, 10/364	5, 22/2002	5, 3/2002

Here, we assign each successful node equal to the inverse of its loss value after fine-tuning, and choose a node by assigning each node a probability proportional to its score and selecting randomly. Due to the random processes in this algorithm, we train four overparameterized networks, and for each overparameterized network we attempt pruning ten times. Results of the best pruning values are in Figure 3.5. For all the cases shown, the network is able to be pruned down to the conjectured behavior from Section 3.3.2. Finding subnetworks with only a single hidden layer only occurs for two and three digit parity.

For both wide and deep networks, pruning gets to or near the optimal bound for parity. However, these algorithms are computationally intensive — for the pruning of wide networks, a width six network trains to success on four-parity approximately 65% of the time. That width six network contains at least one subnetwork of width three that trains to success on four-parity approximately 15% of the time. This gives an approximately 10% chance of being able to train a width six network and prune it to width three. Given that the width six network contains a successful width

Table 3.5: Results of pruning deep networks with width $5n$ and depth five. Results reported are the best over four trainings of original network and ten prunings for each original network.

Digits in parity	Depth pruning $5n \times 5$ network
6	$6 \to 2 \to 2 \to 2 \to 1$ $6 \to 2 \to 3 \to 1$
5	$5 \to 2 \to 2 \to 2 \to 1$ $5 \to 2 \to 3 \to 1$
4	$4 \to 2 \to 2 \to 1$
3	$3 \to 2 \to 2 \to 1$ $3 \to 3 \to 1$
2	$2 \to 2 \to 1$

three subnetwork, only an average of approximately 1.5 width three subnetworks succeed out of 20 possible subnetworks in 4,000 epochs. This means that multiple subnetworks need to be considered. As shown in Figure 3.10, these subnetworks do not necessarily train quickly. The only successful width three subnetwork of the example width six network starts with maximum loss, reduces to a saddle point where it stays for a long period, reduces to a different saddle point where it stays for a long period, and then finally reduces to a local minima that solves the problem. This means that identifying good subnetworks is likely to be difficult.

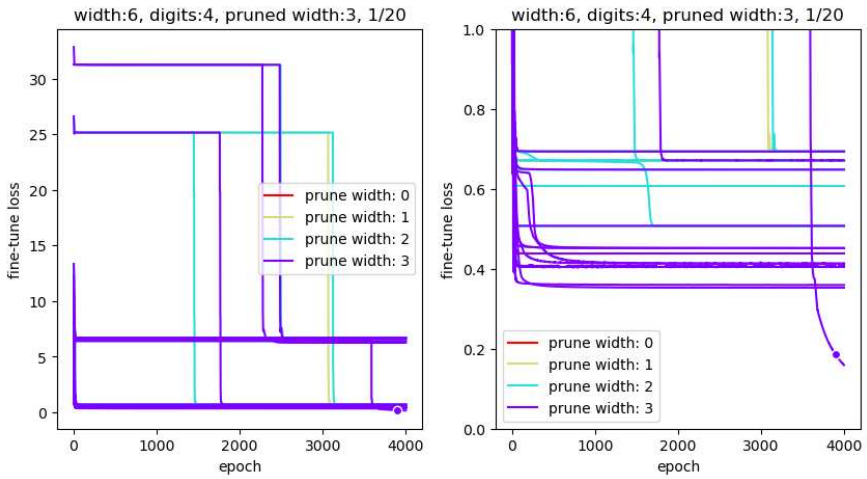


Figure 3.10: The loss at each epoch of the width one, two, and three subnetworks of a width six single-hidden-layer network on four-parity. The right image shows all loss values, and the right zooms in to only show loss values from zero to one.

Width three networks train to success on four-parity approximately 2% of the time whereas the width six networks train and the prune to width three approximately 10% of the time. This means that fewer trials are needed to find a width three network by pruning, but each trial is significantly more costly.

Similarly, a width two depth two network trains successfully approximately 1% of the time. Width 15 depth four networks train successfully almost 100% of the time, but only contain a subnetwork of width two and depth two approximately 20% of the time. Reducing to that size iteratively requires training at least every network on the trajectory from width 15 depth four to width two depth two, which is a significant cost.

Better pruning algorithms likely to exist, but compared to the significant improvement in success rate that leaky ReLU offers, the pruning results are not as appealing. However, in domains where hyperparameter optimization may be more costly so that training near-minimal networks from scratch is difficult, this method may be appealing.

3.4.5 Extension of Pruning to Image Classification Networks

The pruning of Boolean functions to near-minimal networks, as presented in Section 3.4.4, presents an enticing methodology to use for investigating larger. Many of the common metrics used to evaluate pruned networks emphasize network performance, with limited investigation of what effect the pruning has on what the network is fundamentally doing. Even when considering performance, many pruning algorithms do not publish their results in a method that is easy to compare [55]. Our work in similarity presented in Chapter 7 offers a way to determine how similar original networks are to their pruned subnetworks and determine how pruning affects the similarity of different networks.

In contrast to Boolean functions, where meaningful if imprecise bounds on minimal network size can be found, many practical problems have complications that make this impossible. Unknown underlying functions, sampling error, and reduced interpretability of the problem complicate understanding of how large networks need to be to find success. In practice, networks must be

overparameterized. This overparameterization leads to good solutions, but there is evidence that the level of overparameterization used is significant [55]. Many networks are able to memorize the class labels of their inputs, even when those labels are assigned randomly rather than according to the behavior of the underlying function [105]. This presents a domain where knowing the size of a minimal network is hard, but reducing the size of a network is of significant interest. By applying the iterative method discussed in Section 3.4.4 to image-classification networks, we can contrast its performance with existing magnitude-of-weights-based methodologies. Our methodology is similar to some existing methods, typically called channel pruning [55, 56].

We look at networks on the MNIST handwritten digit and CIFAR tiny image recognition datasets, discussed in more detail in Section 6.1. We look at networks consisting of 32 channels of 3x3 two-dimensional convolution followed by a max pool, followed by a convolution layer, followed by a max pool, followed by a convolutional layer, followed by a max pool, followed by a linear layer that maps the results to the ten outputs corresponding to the classes. Networks are trained for 30 epochs using stochastic gradient descent with learning rate 0.001, batch size 40, momentum 0.9, and weight decay 0.0001.

For pruning, we compare structural pruning with magnitude-of-weights pruning. For magnitude-of-weights pruning, the smallest $m\%$ of weights are removed from the network, then the network is fine-tuned by training for a single epoch. For structural pruning, we test the removal of each convolutional channel from the network and determine the resulting loss, then remove the channel with lowest loss after removal. The resulting pruned network is fine-tuned by training for a single epoch, then the process repeats until the network accuracy falls below a threshold. Comparing across these methodologies also will illuminate whether the behavior of different pruning methodologies result in distinct networks. Magnitude-of-weight pruning is able to achieve a high degree of sparsity, with a reduction to 1/100 of the original network size being possible for certain networks, but structural pruning allows for a reduction of the dimensions of the problem, both of which speed up network inference in meaningful ways. Results of pruning on simple MNIST and

CIFAR convolutional networks are shown in Figures 3.11 and 3.12, which demonstrate that a high degree of performance can be preserved as convolutional channels or weights are removed.

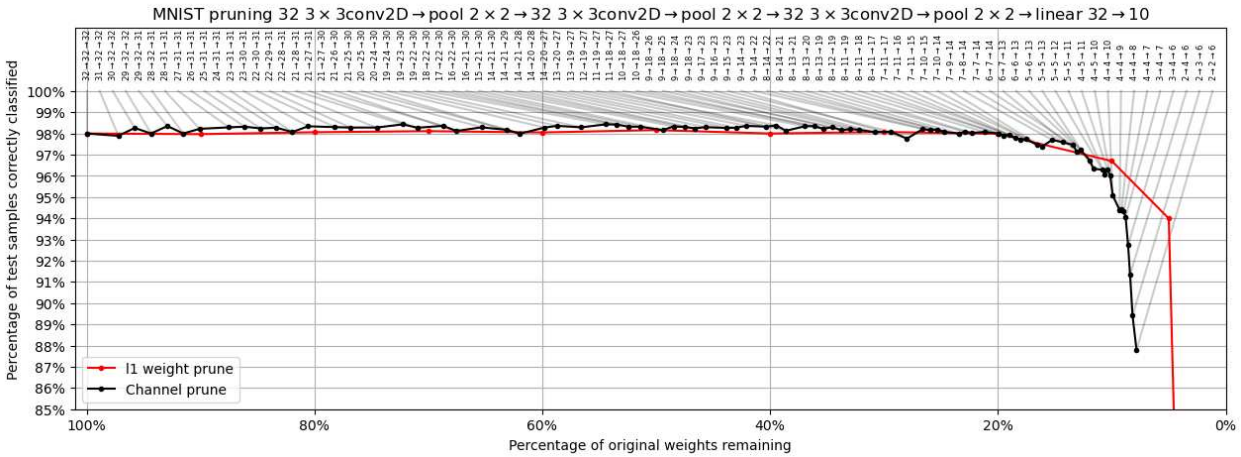


Figure 3.11: The results of pruning a convolutional network trained on MNIST by removing channels from each convolutional layer iteratively. Each point is annotated with the remaining number of channels in each layer. Results are compared to those attained by pruning the network to remove the smallest percentage of weights.

In both cases, the magnitude-of-weight based pruning and structural pruning maintain accuracy near the original accuracy of the network even when a significant portion of the network is removed. The MNIST network loses significant accuracy only when dropping below 20% of the original weights in the network. The CIFAR network retains its original accuracy with 60% of the weights for structural pruning or 40% with magnitude-of-weight based pruning. Structural pruning in both cases has a faster drop in accuracy, but allows for inference to be done more quickly. Magnitude-of-weight pruning results in sparse calculations, but structural pruning results in fewer calculations. The results presented here are not as promising as others in the field, suggesting that the methodology here is too naive [55].

3.5 Conclusions

By using one-dimensional analysis through construction of the swaps function, we can determine that width two Leaky ReLU neural networks can solve any Boolean function and construct

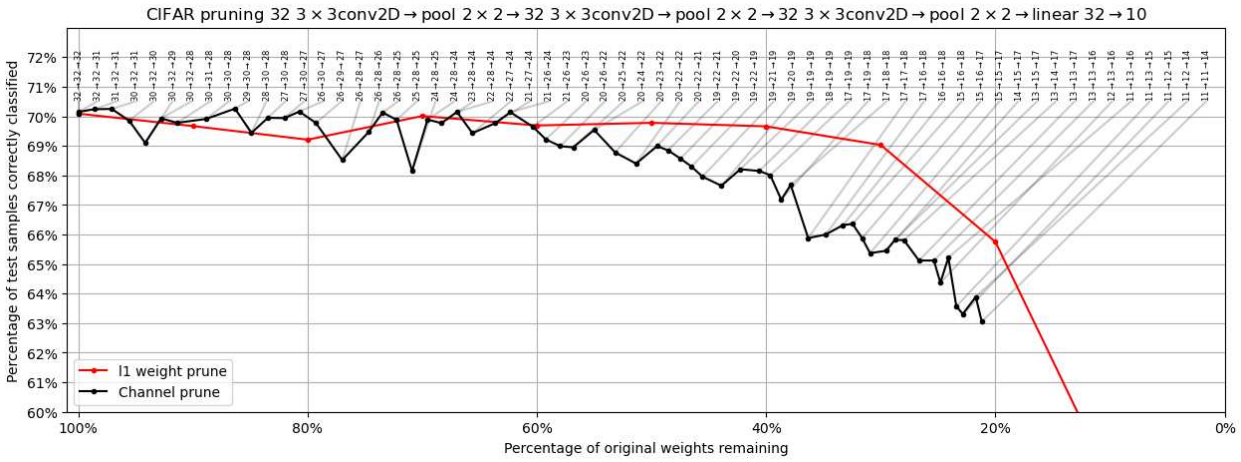


Figure 3.12: The results of pruning a convolutional network trained on CIFAR by removing channels from each convolutional layer iteratively. Each point is annotated with the remaining number of channels in each layer. Results are compared to those attained by pruning the network to remove the smallest percentage of weights.

an upper bound on minimal single-hidden layer network size. On certain problems, like parity, we can construct bounds that are logarithmic in the input dimension.

When training networks in practice, ReLU networks perform poorly at near-minimal sizes, but allowing a leak factor significantly improves success rates. Minimal sized networks still have low success rates, but increasing network size just slightly achieves good success rates. Pruning overparameterized networks can result in minimal networks, but is computationally complex. High influence and Fourier degree are minimal requirements for a problem to be difficult, but easy problems can still have high influence and degree. Entropy appears to be correlated with difficulty as the input dimension increases, but easy problems, such as OR, will still have high entropy.

Open questions remain. What classes of problems have maximal swaps values? What classes of problems have bounded swaps values? Can a network with width w and depth l solve every Boolean function with swaps value $s < w^l$? Is there are relationship between the swaps function and existing metrics for Boolean functions?

Chapter 4

Circle Versus Annulus Classification

Boolean functions provide a way to investigate how complexity of function impacts network performance. However, they are fundamentally linearly mappable to a single dimension. This means that high-dimensional behavior is not guaranteed to appear in analysis. Finding a simple problem that is guaranteed to exhibit high-dimensional behavior is necessary to determine how dimensionality impacts network performance. Classifying a closed, bounded region as distinct from a region surrounding it provides a way of doing this.

The only operations a fully-connected ReLU neural network can perform are affine transformations (the affine mappings) and compressions of space (the ReLUs). Using only these operations, a closed, bounded region can never be separated from its surrounding region as long as the operations the network performs are restricted to the problem dimensionality or lower. However, as demonstrated in Figure 4.1, by adding a single additional layer, the compression can utilize the additional dimension to successfully separate the region and its surroundings. More generally, a fully-connected neural network with n inputs and o outputs can solve every $n \rightarrow o$ problem as long as it has at least one layer with width $n + o$ [7, 10, 25]. For the purposes of classification, this means that classifying an n-dimensional hypersphere and its surrounding n-dimensional hypershell requires a network with at least width $n + 1$ regardless of the number of hidden layers.

Additionally, a single hidden layer with width $n + 1$ is sufficient to create a decision boundary that forms a closed, bounded region. It is of interest to determine the geometry of what regions are possible with that minimal network, which is discussed in Section 4.1, and how difficult the problem is for a network to solve given that the geometry of the problem can be matched by the network, which is discussed in Section 4.2

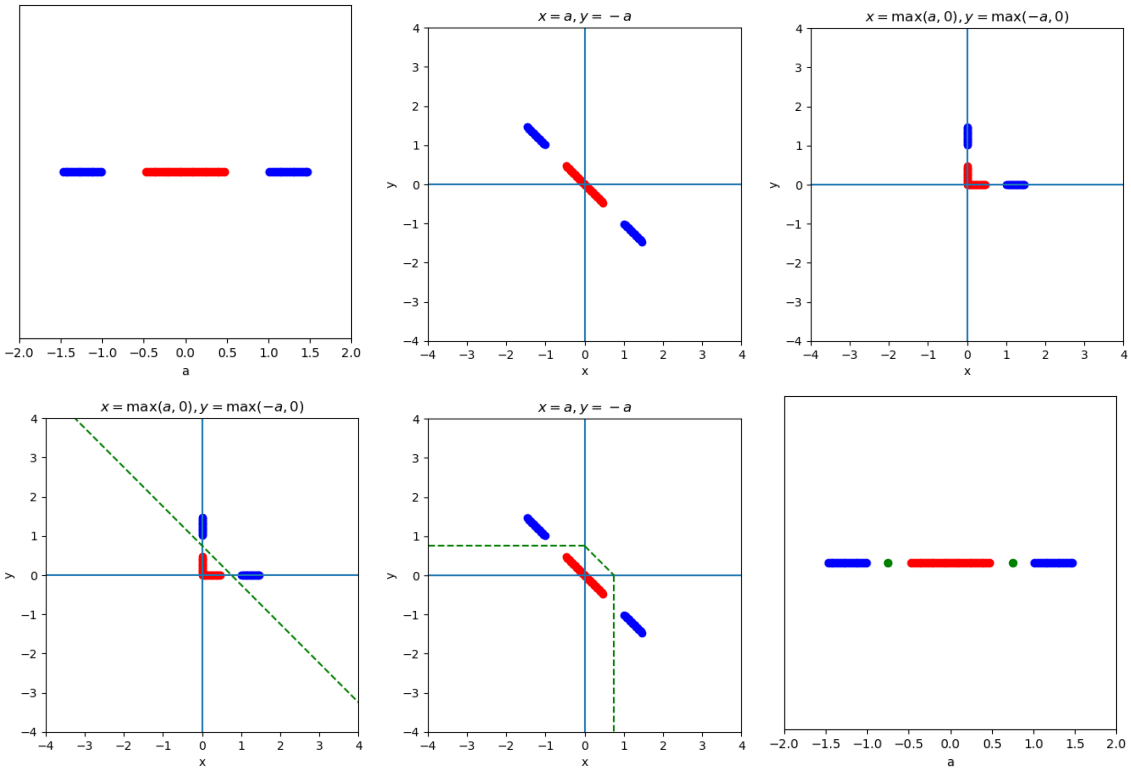


Figure 4.1: The process a width two, single-hidden-layer network may take to classify red and blue points along one-dimension. The top left image shows the original problem. The top center shows the problem after an affine transformation that projects it into two dimensions. The top right shows the outputs of the network's hidden layer, after the ReLU activation. The bottom left shows the decision boundary in the projected, compressed space. The bottom middle shows the decision boundary in the projected space. The bottom right shows the original problem with the decision boundary, which in this case is simply two points.

4.1 Geometric Analysis

A width $n + 1$ single-hidden-layer classification ReLU neural network is able to construct a closed-bounded region in n-dimensional space with some geometry. The geometry of the resulting decision boundary is of interest. Specifically, under what conditions can the network classify a hypersphere with radius r_1 versus its surrounding hypershell with inner radius r_2 and outer radius r_3? We present answers for the two-dimensional case in Section 4.1.1, the three-dimensional case in Section 4.1.2, and extend results to n-dimensions in Section 4.1.3.

4.1.1 Two-Dimensional Solutions

In two dimensions, the problem becomes classifying a circle versus its surrounding annulus, as shown in Figure 4.2. The minimal neural network that can construct a closed bounded region has width three in its single hidden layer.

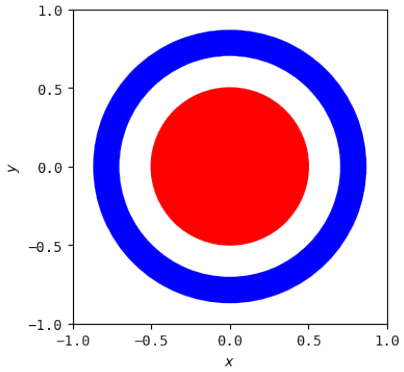


Figure 4.2: The circle versus annulus classification problem. Red and blue regions have different classes.

This minimal neural network can construct closed, bounded regions of two forms. Its decision boundary can form regions in the form of a convex quadrilateral or a hexagon, as shown in Figure 4.3. This decision boundary is formed with its edges connecting the nonlinearities of the hidden layer and its vertices on the nonlinearities. This comes from the fact that the neural network is a piecewise linear function, reducing to a linear function in each convex region bounded by the nonlinearities. Within each of those regions, any change from one class to the next will correspond

to a crossing of zero in the linear region in that region, which will occur along a line. Therefore, the decision boundary is linear in each convex region. The decision boundary is only nonlinear when it intersects one of the nonlinearities of the hidden layers, where it can change direction according to the difference geometries of the linear mappings between the two regions.

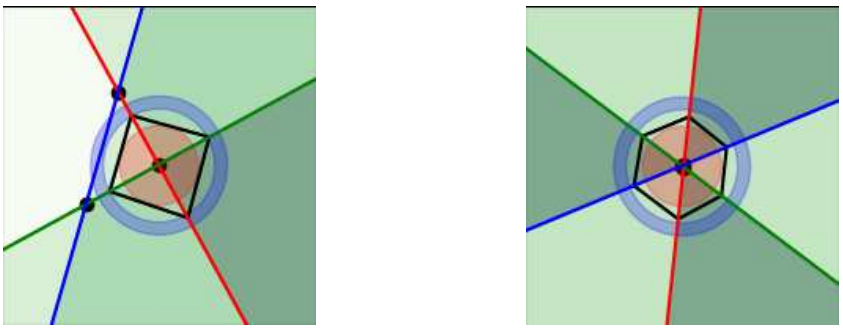


Figure 4.3: The square and hexagon solutions a minimal neural network can form to solve the circle versus annulus problem. Red, blue, and green lines correspond to zeros of the interior three ReLUs. The darkness of green in various regions correspond to how many ReLUs are activated within that region. Black lines form the decision boundary for the full network.

This means that the vertices of the polygon solutions will lie on the nonlinearities. For the convex quadrilateral solution, the third nonlinearity does not appear relevant, but it is still necessary to ensure that in the region where both other nonlinearities are not activated that there is a nonconstant linear function in that region. If the region was allowed to remain constant, the network's output would not shift between positive and negative values and create a decision boundary.

We are interested in when these solutions are able to solve the circle versus annulus problem, which is when their shape is able to fit between the circle and the annulus. To determine when this is the case, we look at the largest circle that can fit within the polygon without overlapping and the smallest circle that surrounds the polygon without overlapping. The inner circle's radius we denote as ρ and the outer circle's radius we denote as R. ρ is related to the inradius of the polygon, which is the radius of the inscribing circle. However, the inradius and ρ are not directly equal, as the inscribed circle is defined as the circle that is tangent to each of the polygon's edges, rather than the largest circle that fits within the polygon [106]. Similarly, R and the circumradius are related, but the circumradius is defined as the circle which passes through each of the vertices

of the polygon and inscribes the polygon [106]. When the polygon is regular (all side lengths are the same), they are equal, but irregular polygons do not have a formal inradius and solutions to higher-dimensional versions of the problem are not always regular.

The outer radius is minimized and the inner radius is maximized for both the hexagon and the square when they are regular. For a square of side length $2k$, the inner circle touches the square at the midpoint of each edge, and so has radius k. The circumscribing circle touches the square at its vertices, and so has radius $\sqrt{k^2 + k^2} = k\sqrt{2}$. Therefore, the square's ratio of inner radius to outer radius is

$$R_{square} = \rho_{square}\sqrt{2} \approx 1.41\rho_{square}. \tag{4.1}$$

Similarly, for a hexagon of side length k, the circumscribing circle touches at the vertices, and so has radius k. The inner circle touches at the midpoint of the faces, and so has radius $\frac{\sqrt{3}}{2}$. Therefore, the hexagon's ratio of inner radius to outer radius is

$$R_{hexagon} = \rho_{hexagon}\frac{2\sqrt{3}}{3} \approx 1.15\rho_{hexagon}. \tag{4.2}$$

This means that for the two-dimensional circle versus annulus problem to be solvable by a neural network, $r_2 \geq 1.15r_1$ for the hexagonal solution and $r_2 \geq 1.41r_1$ for the square solution. The hexagon performs better than the square, as it has more faces and so smooths over the corners.

An alternative construction of the hexagon solution comes from the expansion of the triangle [107]. Given three intersecting lines in the plane that do not all intersect at the same point, connecting their intersection points forms a triangle. Expanding that triangle by separating its edges, moving them radially outward, and adding new edges connecting them results in a hexagon as shown in Figure 4.4. If the triangle is equilateral, opposite sides of the hexagon will be parallel. As the side length of the equilateral triangle shrinks, the side lengths of the long and short edges of the expanded hexagon approach the same value. When the three lines intersect at a single point and have angles of 120° from each other, the expanded hexagon is regular.

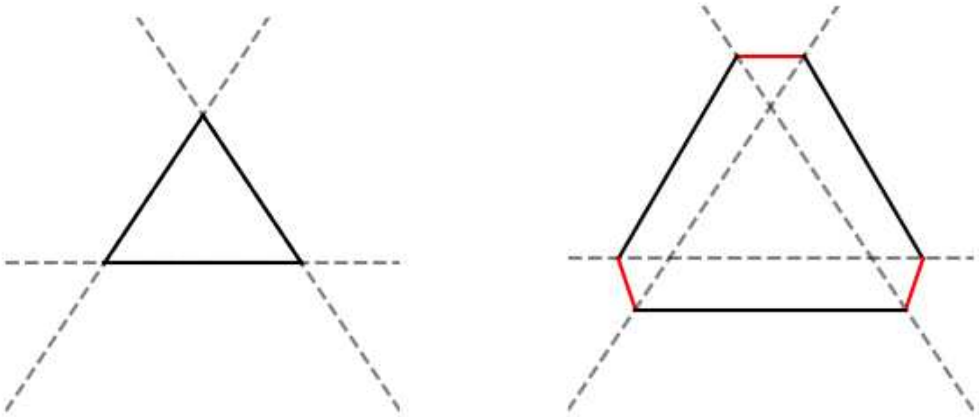


Figure 4.4: An example of how the triangle formed by three lines in the plane can be expanded geometrically to form a hexagon. Left: the triangle formed by the nonlinearities. Right: a possible expansion of that triangle to form a hexagon.

This expansion construction is not particularly meaningful for the two-dimensional case, except as a curiosity. However, it generalizes for higher dimensions — in n-dimensions, the solution that corresponds to the hexagon is the expansion of the n-simplex [108–110].

4.1.2 Three-Dimensional Solutions

Three dimensional solutions generalize from the two-dimensional solutions. The square becomes the octahedron, and the hexagon becomes the cuboctahedron, as shown in Figure 4.5. Much like in two dimensions, the octahedron has minimal ratio of inner radius to outer radius when it is regular, with all faces being equilateral. The cuboctahedron has both triangular and quadrilateral faces, so it cannot be regular, but it has minimal ratio of inner radius to outer radius when it is uniform — that is, all of its faces are regular. The cuboctahedron has radial equilateral symmetry, which means that the distance to each vertex is the same as the edge length.

For an regular octahedron with side length one, its vertices lie a distance of $\frac{\sqrt{2}}{2}$ away from its midpoint, so its circumradius is $\frac{\sqrt{2}}{2}$. Its inner radius will come from the distance to the center of its triangular faces, which is $\frac{\sqrt{6}}{6}$. Therefore, the regular octahedron's ratio of inner radius to outer radius is

$$R_{square} = \rho_{square}\sqrt{3}. \tag{4.3}$$

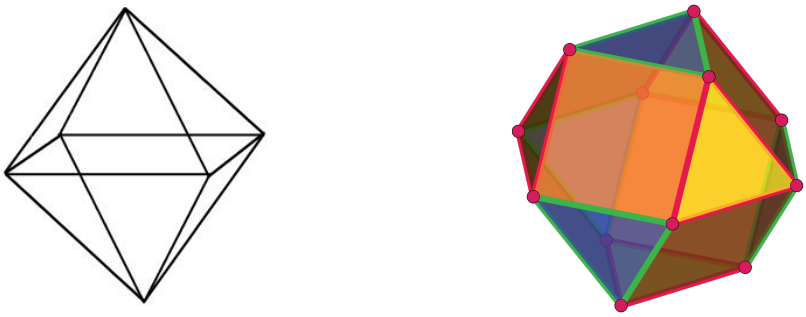


Figure 4.5: Example geometries of the sphere versus shell problem — an octahedron [111] and a cuboctahedron [112].

For the uniform cuboctahedron with side length one, it will have vertices a distance of one away from its center. It has two kinds of faces, with the triangular faces having their closest point to the center occur at a distance of $\frac{\sqrt{6}}{3}$ and the square faces having their closest point to the center occur at a distance of of $\frac{\sqrt{2}}{2}$. The square centers are closer to the center, and so the uniform cuboctahedron has ratio of inner radius to outer radius

$$R_{cuboctahedron} = \sqrt{2}\rho_{cuboctahedron}. \tag{4.4}$$

Much like with the two-dimensional case, the cuboctahedron is the expansion of the tetrahedron [108–110].

4.1.3 Higher dimensions

Higher dimensional solutions extend from the two and three-dimensional solutions. The square and octahedron become the hyperoctahedron, and the hexagon and cuboctahedron become the expanded n-simplex, denoted eS_n in Conway polyhedron notation [113]. The hyperoctahedron follows from the octahedron as having n orthogonal hyperplanes that intersect at the hyperoctahedron center, and a final hyperplane to give the region with the ReLUs that correspond to the orthogonal hyperplanes not activated a linear function, rather than a constant one. The expanded n-simplex comes from the expansion of the n simplex, as with the expansion of the hexagon from the triangle and the cuboctahedron from the tetrahedron. The hyperoctahedron has minimal ratio of inner

radius to outer radius when it is regular, and the expanded n-simplex has minimal ratio of inner radius to outer radius when it is uniform.

The hyperoctahedron has ratio of circumradius to inradius

$$R_{ho} = \rho_{ho}\sqrt{n}. \tag{4.5}$$

The expanded n-simplex is more complicated. The inner radii for the expanded n-simplex of circumradius one are shown in Table 4.1 for dimensions two through ten. Every odd dimension, the expanded n simplex gains an additional type of facet whose center is closest to the origin [108–110, 114]. For $k \in \mathbb{N}, k \geq 1$, and $n \geq 2$ with $n \geq 2k-1$, the facet that appears in dimension $2k-1$ has shortest distance from its face to the center of the expanded n-simplex

$$d(i,n) = \sqrt{\frac{n+1}{2(kn-k(k-1))}}. \tag{4.6}$$

We care about the inner radius, so we need the k such that the distance is minimized for a given n. Since $n \geq 2k-1 > k-1$, $kn-k(k-1)$ grows with respect to k. The facet with shortest distance is then the one added either at n for odd dimensions or $n-1$ for even dimensions. Therefore, either $k = \frac{n}{2}$ when n is even or $k = \frac{n+1}{2}$ when n is odd.

Table 4.1: For an expanded n-simplex with circumradius one, this table shows the corresponding inner radius [114].

Dimension	2	3	4	5	6	7	8	9	10
Inner radius	$\sqrt{\frac{3}{4}}$	$\sqrt{\frac{4}{8}}$	$\sqrt{\frac{5}{12}}$	$\sqrt{\frac{6}{18}}$	$\sqrt{\frac{7}{24}}$	$\sqrt{\frac{8}{32}}$	$\sqrt{\frac{9}{40}}$	$\sqrt{\frac{10}{50}}$	$\sqrt{\frac{11}{60}}$
Approximate value	0.87	0.71	0.64	0.58	0.54	0.5	0.47	0.45	0.43

The inner radius is then

$$\rho_{eS_n} = \begin{cases} \sqrt{\frac{n+1}{2(\frac{n^2}{2} - \frac{n}{2}(\frac{n}{2}-1))}}, & n \text{ even} \\ \sqrt{\frac{n+1}{2(\frac{n(n+1)}{2} - \frac{(n+1)}{2}(\frac{n+1}{2}-1)}}, & n \text{ odd} \end{cases} \tag{4.7}$$

the denominator when n is even is

$$2\left(\frac{n^2}{2} - \frac{n}{2}\left(\frac{n}{2} - 1\right)\right) = n^2 - n\left(\frac{n}{2} - 1\right) = n^2 - \frac{n^2}{2} + n = \frac{n^2}{2} + n = \frac{n(n+2)}{2} \tag{4.8}$$

and when n is odd it is

$$\begin{aligned} 2\left(\frac{n(n+1)}{2} - \frac{(n+1)}{2}\left(\frac{n+1}{2} - 1\right)\right) &= n(n+1) - \frac{(n+1)(n-1)}{2} \\ &= n^2 + n - (n+1)\left(\frac{n}{2} - \frac{1}{2}\right) \\ &= n^2 + n - (\frac{n^2}{2} - \frac{1}{2}) \\ &= \frac{n^2}{2} + n + \frac{1}{2} \\ &= \frac{n^2 + 2n + 1}{2} \\ &= \frac{n(n+2)+1}{2} \end{aligned} \tag{4.9}$$

and so

$$\rho(n) = \begin{cases} \sqrt{\frac{2(n+1)}{n(n+2)}}, & n \text{ even} \\ \sqrt{\frac{2(n+1)}{n(n+2)+1}}, & n \text{ odd} \end{cases} \tag{4.10}$$

which, removing the cases, is

$$\rho(n) = \sqrt{\frac{2(n+1)}{n(n+2) + 0.5(1 - (-1)^n)}} \tag{4.11}$$

This means that the expanded n-simplex has ratio of outer radius to inner radius

$$R_{eS_n} = \rho_{eS_n}\sqrt{\frac{n(n+2) + 0.5(1 - (-1)^n)}{2(n+1)}} \tag{4.12}$$

Therefore, the asymptotic behavior is the same for both the expanded n-simplex and the hyperoctahedron — $\Theta(\sqrt{n})$. However, in the limit, for a fixed outer radius R for both the expanded n-simplex and the hyperoctahedron,

$$\sqrt{2}\rho_{eS_n} \approx \rho_{ho} \tag{4.13}$$

The similarity of the ratios for the hyperoctahedron and expanded n-simplex isn't as surprising when viewed from the perspective of how many hyperplanes are used to construct the polytope — the hyperoctahedron uses n and the expanded n-simplex uses $n+1$. When n is small, such as the two-dimensional case, adding an additional hyperplane has a significant effect. However, when n is large, the impact of the additional hyperplane is relatively smaller.

Another way to consider the change is size is the volume rather than the radius — specifically, the volume of the hypershell between the inner hypersphere and the outer hypersphere. The volume of a radius r n-hypersphere is

$$V(n,r) = \frac{\pi^{n/2}}{\Gamma(\frac{n}{2}+1)} r^n. \tag{4.14}$$

For the hyperoctahedron, the volume of the hypershell is then

$$\begin{aligned} V(n,R_{ho}) - V(n,\rho_{ho}) &= V(n,\sqrt{n}\rho_{ho}) - V(n,\rho_{ho}) \\ &= \frac{\pi^{n/2}}{\Gamma(\frac{n}{2}+1)}(n^{n/2}-1)\rho_{ho}^n \end{aligned} \tag{4.15}$$

and for the expanded n-simplex it is

$$\begin{aligned} V(n,R_{eSn}) - V(n,\rho_{eSn}) &= V\left(n, \sqrt{\frac{n(n+2)+0.5(1-(-1)^n)}{2(n+1)}}\rho_{eSn}\right) - V(n,\rho_{eSn}) \\ &= \frac{\pi^{n/2}}{\Gamma(\frac{n}{2}+1)}\left(\sqrt{\frac{n(n+2)+0.5(1-(-1)^n)}{2(n+1)}}^n - 1\right)\rho_{eSn}^n \end{aligned} \tag{4.16}$$

and so for a fixed $1 = \rho_{ho} = \rho_{eSn}$, the ratio of the volume of the hypershell for the hyperoctahedron versus the expanded n-simplex approaches

$$\lim_{n\to\infty} \frac{V(n, R_{ho}) - V(n, 1)}{V(n, R_{eSn}) - V(n, 1)} = \lim_{n\to\infty} \frac{(n^{n/2} - 1)}{\left(\sqrt{\frac{n(n+2)+0.5(1-(-1)^n)}{2(n+1)}}^n - 1\right)} = 2^{n/2} \tag{4.17}$$

From this perspective, the space that the expanded n-simplex takes up is significantly smaller than the space that the hyperoctahedron fills.

For the hypersphere versus hyperannulus problem to be solvable by the octahedron solution, we require $r_2 \geq \sqrt{n} r_1$. The ratio of $\sqrt{n}$ between r_1 and r_2 means that the ratio of the volume of the region between the hypersphere and hypershell to the hypersphere grows as n increases. The volume of a n-hypersphere with radius r is given in Equation 4.14. The hypersphere then has volume

$$V(n, r_1) = \frac{\pi^{n/2}}{\Gamma(\frac{n}{2} + 1)} r_1^n, \tag{4.18}$$

and the volume between the hypersphere and hypershell is

$$V(n, r_1\sqrt{n}) - V(n, r_1) = \frac{\pi^{n/2} n^{n/2}}{\Gamma(\frac{n}{2} + 1)} r_1^n - \frac{\pi^{n/2}}{\Gamma(\frac{n}{2} + 1)} r_1^n = \frac{\pi^{n/2}(n^{n/2} - 1)}{\Gamma(\frac{n}{2} + 1)} r_1^n. \tag{4.19}$$

The ratio of the hypersphere's volume to the volume between the hypersphere and hyperannulus is then

$$\frac{V(n, r_1\sqrt{n}) - V(n, r_1)}{V(n, r_1)} = n^{n/2} - 1 \tag{4.20}$$

and so the ratio of volume grows as $\Theta(n^{n/2})$. This means that classifying a closed, bounded, region requires an increasingly large space as the dimension increases.

4.2 Experimental results

From the previous section, a width $n + 1$ network is able to classify a hypersphere versus a hypershell as long as they are separated by a distance that grows in $\Omega(\sqrt{n})$. However, it is not clear how frequently the network is able to identify such a solution. To address this, we investigate hyperparameter selection for training on the two-dimensional circle versus annulus problem in Subsection 4.2.1 and performance in higher dimensions in Subsection 4.2.2.

4.2.1 Two-Dimensional Hyperparameter Analysis

We are interested in determining the effect of certain hyperparameters on how well networks are able to train on the circle versus annulus problem. Specifically, the difference between mean squared error and binary cross-entropy, the effect of scaling the inputs, leak values for Leaky ReLU, and batch size. We use Adam with learning rate 0.01 for up to 1,000 epochs with 1,000 samples, 500 of which are sampled uniformly from the annulus and 500 of which are sampled uniformly from the circle. We choose radius of the circle, r_1, inner radius of the annulus $r_2 = r_1\sqrt{2}$ and outer radius of the annulus $r_3 = r_2\sqrt{2}$ so that the circle, annulus, and space between the circle and annulus have the same area. When not normalizing, r_1 is set to 0.5 and when normalizing the samples are scaled numerically such that they have standard deviation one. We investigate leak values of 0 (ReLU), 0.01, and 0.1 and batch sizes of 10, 100, and 1,000.

Rather than looking for 100% classification rate, we count networks that classify 95% of the 1,000 samples correctly as being successful. Solutions that fail are unable to achieve that success rate. For successful networks, because any given batch does not include every point in the circle and the annulus, the network will converge to a region *around* the solution, rather than the exact value. This leads to error, as the optimal quadrilateral solution has outer radius equal to $\sqrt{2}$ times its inner radius, which is the same gap between the circle and annulus. If the solution is not an exact square, and instead a skewed convex quadrilateral, the decision boundary will cut through part of the circle or annulus. An example of this is shown in Figure 4.6

Results for each combination of hyperparameter are in Table 4.2. Average performance for each hyperparameter is in Table 4.3. Using leaky ReLU over ReLU significantly improves network performance, use of binary cross-entropy over mean squared error slightly improves network performance on average, normalizing the inputs improves network performance, and smaller batch sizes have improved performance. All individual hyperparameter choices have high performance in at least one full hyperparameter selection except for usage of a leak value of zero and batch sizes of 1,000.

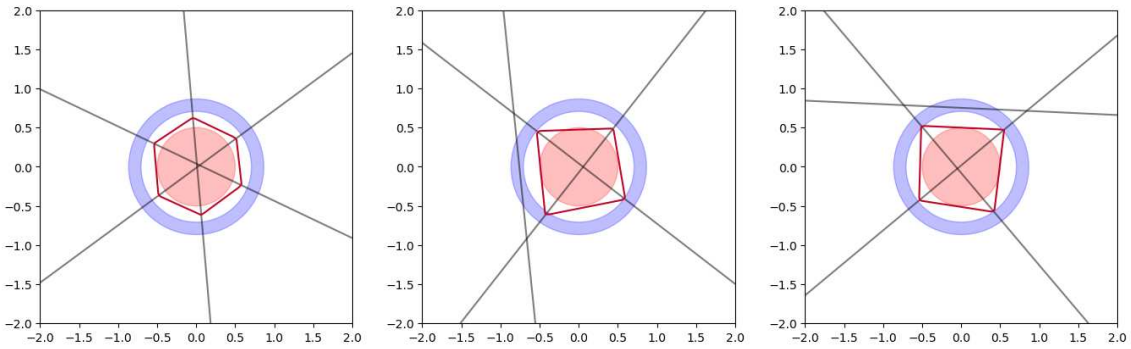


Figure 4.6: Solutions to the circle versus annulus problem in two dimensions. The convex quadrilateral decision boundaries (in red) overlap with the circle and annulus due to the fact that the network does not train on the entirety of the problem in each step.

Table 4.2: Number of networks which classify 95% or more of the 1,000 points on the circle versus annulus problem for various combinations of hyperparameters. BCE is binary cross-entropy and MSE is mean squared error. "leak" refers to the leak parameter in the Leaky ReLU activation function. "batch" refers to the batch size. "normalize" refers to whether inputs are transformed to have standard deviation one. "successes" is the number of networks out of 100 that succeed on that problem.

loss	leak	batch	normalize	successes
MSE	0	1000	False	29
MSE	0	1000	True	37
MSE	0.1	1000	True	37
MSE	0	100	False	38
BCE	0	1000	False	42
MSE	0.01	1000	True	42
MSE	0	10	False	45
MSE	0.1	1000	False	47
BCE	0	10	False	47
BCE	0	100	False	48
MSE	0.01	1000	False	50
MSE	0	100	True	52
BCE	0.01	1000	False	53
BCE	0	1000	True	56
MSE	0.1	100	False	57
MSE	0.1	100	True	58
BCE	0.1	1000	False	59
BCE	0.01	1000	True	61

loss	leak	batch	normalize	successes
BCE	0	100	True	61
MSE	0	10	True	62
BCE	0.1	1000	True	63
MSE	0.01	100	True	64
MSE	0.01	100	False	64
BCE	0	10	True	69
BCE	0.1	100	True	73
BCE	0.01	100	False	74
BCE	0.1	100	False	74
BCE	0.01	100	True	82
BCE	0.01	10	False	83
MSE	0.1	10	False	84
BCE	0.1	10	False	84
MSE	0.01	10	False	85
MSE	0.01	10	True	88
BCE	0.1	10	True	90
BCE	0.01	10	True	91
MSE	0.1	10	True	94

Table 4.3: The average number of successful networks out of 100 across all other hyperparameter selections for each hyperparameter. BCE is binary cross-entropy and MSE is mean squared error. "leak" refers to the leak parameter in the Leaky ReLU activation function. "batch" refers to the batch size. "normalize" refers to whether inputs are transformed to have standard deviation one. "successes" is the number of networks out of 100 that succeed on that problem.

loss	rate
MSE	57
BCE	67

norm	rate
True	66
False	59

leak	rate
0	49
0.01	70
0.1	68

batch	rate
10	77
100	62
1000	48

4.2.2 Performance in Higher Dimensions

For higher dimensions, we consider binary cross-entropy loss, a batch size of 100 out of 10,000 training samples, normalized inputs, and leak values of 0 or 0.01. 10,000 independent testing samples are used to determine accuracy. These parameters come from the experiments in Subsection 4.2.1, where they were determined to have good performance on the two-dimensional case. Results for dimensions two through eight are shown in Figure 4.8. We look at four threshold for what proportion of points must be correctly classified — 0.95, 0.98, 0.99, and 0.9999. There are two conflicting issues that lead to us considering multiple thresholds. As dimension increases, the number of samples to maintain the same sampling density increases exponentially. We are using a fixed number of samples regardless of dimension (although it is high enough for the dimension eight problem to have sufficient density) so we would expect the performance of the network to decrease as it has less data to work with. The other influence is that as the dimension increases, most of the volume of the hyperoctahedron and expanded n-simplex will trend towards their corners. If the hyperoctahedron is not regular or the expanded n-simplex is not uniform, the overlap with the hypersphere and hyperannulus will increase as n increases. The fixed batch size compounds this, as the sampling density within a batch will decrease and solutions will lie in a larger region around the regular or uniform solution.

Additionally, as dimension increases, any fixed number of lines will have increasing probability of being orthogonal. This does not strictly hold for an increasing number of lines, but as shown in Figure 4.7, for n vectors in n dimensional space, they increase in their average orthogonality, and

their angle to the direction they are least orthogonal to decreases as n decreases. This suggests that as the dimension increases, the likelihood of being near the octahedron solution at initialization starts as long as weights are distributed normally.

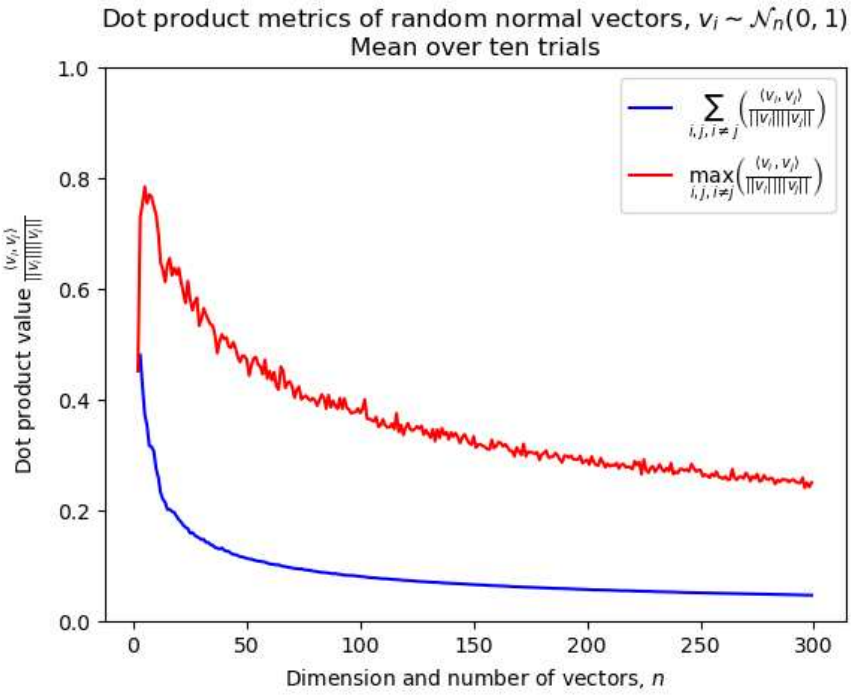


Figure 4.7: The average and maximum dot products of n vectors in n dimensions. Maximum and average dot product are the average of maximums and averages over 100 trials

Results are shown in Figure 4.8. As the dimension increases, the number of networks with near-zero error decreases. However, the number of networks with a small amount of error remains consistent. This suggests that as the dimension increases, the problem does not significantly increase in difficulty except in the way that high-dimensional problems are fundamentally difficult. Networks are able to obtain solutions that approximate the optimal solution, but identifying the optimal solution with a small batch size is unlikely. The likelihood of a network having an irregular or nonuniform decision boundary increases, but the likelihood that the network constructs the hyperoctahedron or expanded n-simplex remains consistent.

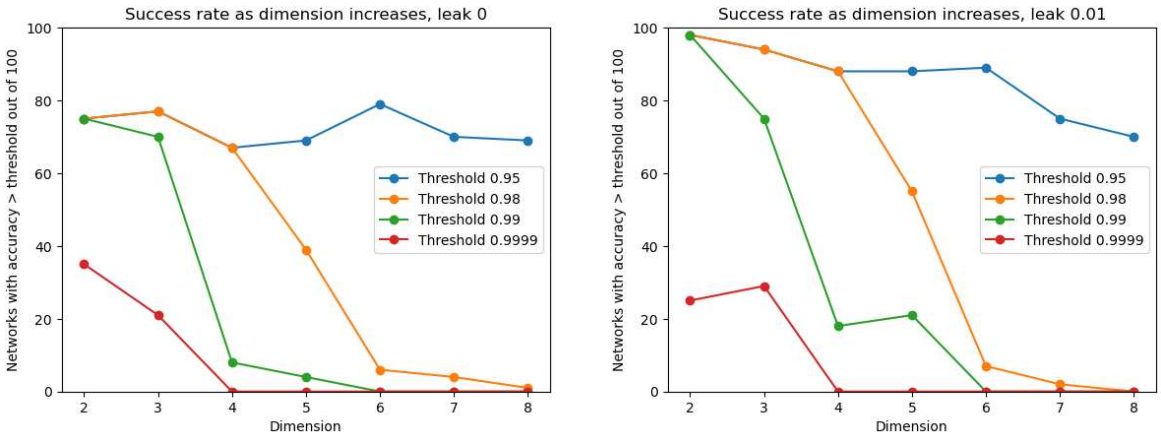


Figure 4.8: Number of networks out of 100 achieving a proportion of points out of 10,000 correctly classified for various input dimensions. Left: Results with the ReLU activation function. Right: Results with leaky ReLU with a leak factor of 0.01.

4.3 Conclusions

Neural networks are able to construct closed, bounded regions with their decision boundary in n-dimensional space as long as they have at least one layer of width $n + 1$. If they only have one layer with width $n + 1$, and they are trying to separate a hypersphere from a hyperannulus, there are two possible optimal solutions. The regular hyperoctahedron, which has a bounding hypersphere with radius equal to $\sqrt{n}$ the radius of the interior sphere and the uniform expanded n-simplex, which has a bounding hypersphere with radius equal to $\sqrt{\frac{n(n+2)+0.5(1-(-1)^n)}{2(n+1)}}$ the radius of the interior sphere. Both of these grow as $\Theta(\sqrt{n})$, although the hyperoctahedron outer radius approaches $\sqrt{2}$ that of the expanded n-simplex. As a corollary to these, the ratio of volumes of the shell between the bounding and interior spheres for the hyperoctahedron and the expanded n-simplex grows as $2^{n/2}$, and so the expanded n-simplex has exponentially less volume.

Experimentally in two dimensions, using leaky ReLU over ReLU significantly improves network performance and use of binary cross-entropy over mean squared error, normalizing the inputs, and smaller batch sizes improves network performance slightly. On higher-dimensional problems, networks tend to identify solutions near the regular hyperoctahedron or the uniform expanded n-simplex, but they do not converge to exactly the regular or uniform versions of those shapes. This is likely due to a small batch size leading convergence to be in a region around the minima rather

then directly at the minima. This results in classification error as the skewed versions of the shapes overlap the hypersphere and hypershell.

Chapter 5

Background and Introduction to Linearization

Traditionally when discussing gradients in the context of neural network, those gradients are of the loss function with respect to the weights in the network, as a means to determine how to update those weights during training. In contrast, this work focuses on gradients of the network *outputs* with respect to the *inputs*. Calculating such gradients allows for us to approximate the behavior of the network linearly in a region around any given input. These linearizations can be used to calculate various metrics related to network behavior or for comparing multiple networks [18].

Networks use nonlinear activation functions, which means that these linearizations are not exact representations of the behavior of the network. However, many common modern activation functions, such as ReLU and max pooling, are piecewise linear. If a network uses only piecewise linear activation functions, the network itself will be piecewise linear. This means that the linearization of the network will be exact in a region defined by the structure of the activation functions. Furthermore, if ReLU is the nonlinearity used, these regions will be convex, forming a polytope structure in the input space. The structure of these polytopes has been exploited to investigate topics such as network complexity, generalization, and defense against adversarial examples [18, 19, 77–79].

Traditionally such linearizations are considered to arise from the Taylor series, where for a function, $f : \mathbb{R} \to \mathbb{R}$, that is infinitely differentiable at some point, x_0,

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2}(x - x_0)^2 + ... = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x - a)^n. \tag{5.1}$$

For all x sufficiently near x_0, this can be used to construct a linear approximation to f as

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \mathcal{O}(x^2) \qquad \text{as } x \to x_0. \tag{5.2}$$

Note that this is a measure as $x - x_0$ shrinks, rather than as x grows (as might be typical in algorithmic complexity analyses) — such approximations are only accurate in a small region around the point of linearization dependent on how quickly the function changes around that point [115].

This method can also be extended to multiple dimensions, where for a function, $f : \mathbb{R}^d \to \mathbb{R}$, linearized about a point, $x_0 \in \mathbb{R}^d$,

$$f(x) \approx f(x_0) + \nabla f(x_0) \cdot (x - x_0) \qquad \text{for } x \text{ near } x_0, \tag{5.3}$$

vectorizing Equation (5.2). The higher order terms in the Taylor series become more complex, as they include relationships between the variables, but by considering only linearizations we avoid this additional complexity.

Unsurprisingly, as this linearization uses the value of the derivative, this method of analysis is useful in determining the local direction and magnitude of growth of the function. It additionally provides a way of investigating the behavior of a single function at points of interest or directly comparing two functions on the same domain. For example, linearizations across the domain of a function can be used to determine directions of maximal linear change across the function, as discussed in Section 6.4 [116]. Similarly, given linearizations of multiple functions about certain shared points of interest, the constant and derivative terms of those linearizations can be compared to determine to what extent those functions agree locally, as discussed in Chapter 7.

Neural networks include nonlinear activation functions, which means that these linearizations are not exact representations of the behavior of the network. By limiting the Taylor series expansion to the first two terms, there is a potential error in the approximation based on deviations in the higher-order terms. However, modern neural networks across a wide variety of domains use the ReLU activation function, as popularized by the success of AlexNet [45]. This activation function, in addition to other common layer structures such as max pooling, are piecewise linear. If a network uses only piecewise linear activation functions, the network itself will be piecewise linear. This means that the linearization of the network will be exact in a region defined by the structure of the activation functions. Furthermore, the structure of ReLU causes these regions to be convex,

forming a structure of polytopes in the initial input space wherein each polytope corresponds to a single linear function that the neural network takes on in that region.

Sections 5.1 and 5.2 discuss the calculation of these linear regions, especially for ReLU networks, in more detail. Section 5.3 gives an example of the calculation of these linear regions in a one-dimensional example to build intuition for neural network linearizations.

5.1 Gradients with Respect to the Input

As previously mentioned, when discussing gradients in relation to neural networks, typically those gradients are of a loss function used as a proxy for determining the "quality" of the network with respect to the weights. This gradient is then used to update those weights to shift the network towards a "better" solution. Specifically, for a network $v : \mathbb{R}^d \to \mathbb{R}^n$ parameterized by a collection of many weights, Ω, and for an input $x \in \mathbb{R}^d$, we typically concern ourselves with

$$\nabla_\Omega \text{loss}(v(x; \Omega)). \tag{5.4}$$

The input is treated as fixed, and the gradient provides a metric for determining what direction in weight space will increase the "quality" of the network according to the loss function. This gradient is then used to update the weights to reduce the total loss as a proxy for achieving some task.

In contrast, when we discuss gradients and linearizations here, we are concerned with the gradient of a network's outputs with respect to an input — that is (for a network with a single output, $n = 1$) and a d dimensional input, $x \in \mathbb{R}^d$,

$$\nabla_x v(x; \Omega). \tag{5.5}$$

In practice, networks have many outputs ($n > 1$) and so we must calculate the gradient of each output with respect to the input. The matrix containing those gradients is known as the Jacobian of v, $\mathbf{J}_v$, which is of size $d \times n$ and can be written as

$$\mathbf{J}_v(x) = \begin{bmatrix} \frac{\partial v}{\partial x_1} & \cdots & \frac{\partial v}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \nabla_x^T v_1 \\ \vdots \\ \nabla_x^T v_d \end{bmatrix}. \tag{5.6}$$

Each element of $\mathbf{J}_v$ is given by $\mathbf{J}_{v,ij} = \frac{\partial v_i}{\partial x_j}$ — the partial derivative of the i^{th} element of the output of v with respect to the j^{th} element of the input.

In general, for two networks $v : \mathbb{R}^d \to \mathbb{R}^n$ and $w : \mathbb{R}^d \to \mathbb{R}^n$, we could directly compare $J_v(x)$ and $J_w(x)$ as functions. However, the Jacobians are typically at least as complex as the function they are of (for simple ReLU neural networks, the derivative is a piecewise *constant* function with the same polytope structure as the original network), so such comparisons are unlikely to be meaningful. Instead, we can calculate the Jacobian for each function at a number of corresponding points to construct linearizations about those points. The Jacobian allows us to write linearizations of v near $x_0 \in \mathbb{R}^d$ as

$$v(x) \approx v(x_0) + \mathbf{J}_v(x_0)x \qquad \text{for } x \text{ near } x_0. \tag{5.7}$$

For m points, $x_0, x_1, ..., x_m \in \mathbb{R}^d$, the collections of linearizations $\{J_v(x_1), J_v(x_1), ..., J_v(x_m)\}$ and $\{J_w(x_1), J_w(x_1), ..., J_w(x_m)\}$ can then be compared to determine how similar v and w are across a selection of corresponding points in their domains.

5.2 Special Properties of ReLU

The rectified linear unit (ReLU) function,

$$\text{ReLU}(x) = \max\{x, 0\}, \tag{5.8}$$

is one of the most commonly used activation functions for neural networks since its adoption for AlexNet [45]. It is typically vectorized elementwise as

$$\text{ReLU}(x) = \text{ReLU}\left(\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix}\right) = \begin{bmatrix} \text{ReLU}(x_1) \\ \text{ReLU}(x_2) \\ \vdots \\ \text{ReLU}(x_d) \end{bmatrix} \tag{5.9}$$

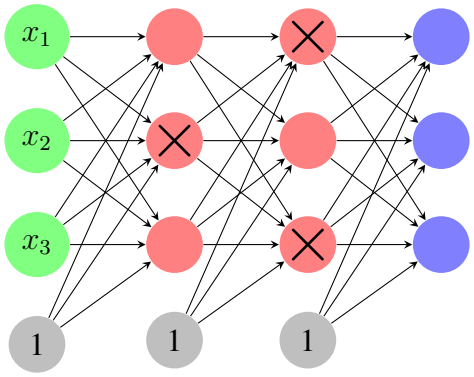
for notational convenience. It has achieved significant practical success despite the analytic issue of not being everywhere differentiable. This presents a problem during training, as the derivative at zero is not defined, and so it is not necessarily clear how updates should be taken. Typically, a value of 0, 1, or 0.5 is chosen to represent its derivative at zero. Similarly, this presents a problem when using Taylor-series linearizations. This becoming an issue is rare in practice, as values tend not to exactly equal zero, and any reasonable choice of value will not affect behavior significantly. Any linearization at the boundary of a ReLU will run into this issue.

However, there is another way to think of ReLU neural networks — they are piecewise linear, and so fundamentally they are a collection of locally linear functions. This means that for neural networks with ReLU activation functions, any given input will either activate or fail to activate each ReLU function as it propagates through the network. The pattern of activations forms a bit string that encodes which paths through the network are utilized. Additionally, for a given input, each ReLU can be replaced either by the identity function or zero depending on if that input activates that ReLU. Once that is done, the network is nothing more than a series of linear transformations as shown in Figure5.1. Hence, the network forms a piecewise linear function in the input space with breaks occurring when a given ReLU function shifts from activated to deactivated or vice-versa.

For an example of how this structure arises, consider a network solving XOR problem given in Table 5.2a:

$$v(x) = w^T \text{ReLU}\left(\mathbf{A}x + b\right) + c, \tag{5.10}$$

with weights



Network:

$$f(x_1,x_2,x_3) = \begin{bmatrix}7&9&6\\4&0&9\\7&3&7\end{bmatrix} \text{ReLU}\left(\begin{bmatrix}2&9&5\\1&3&3\\8&7&0\end{bmatrix} \text{ReLU}\left(\begin{bmatrix}3&0&4\\5&9&1\\6&1&2\end{bmatrix}\begin{bmatrix}x_1\\x_2\\x_3\end{bmatrix} + \begin{bmatrix}7\\2\\9\end{bmatrix}\right) + \begin{bmatrix}7\\7\\8\end{bmatrix}\right) + \begin{bmatrix}5\\5\\2\end{bmatrix}$$

Post ReLU:

$$\begin{aligned} f(x_1,x_2,x_3) &= \begin{bmatrix}7&9&6\\4&0&9\\7&3&7\end{bmatrix}\left(\begin{bmatrix}\cancel{2}&\cancel{9}&\cancel{5}\\1&3&3\\\cancel{8}&\cancel{7}&\cancel{0}\end{bmatrix}\left(\begin{bmatrix}3&0&4\\\cancel{5}&\cancel{9}&\cancel{1}\\6&1&2\end{bmatrix}\begin{bmatrix}x_1\\x_2\\x_3\end{bmatrix} + \begin{bmatrix}7\\\cancel{2}\\9\end{bmatrix}\right) + \begin{bmatrix}\cancel{7}\\7\\\cancel{8}\end{bmatrix}\right) + \begin{bmatrix}5\\5\\2\end{bmatrix} \\ &= \begin{bmatrix}7&9&6\\4&0&9\\7&3&7\end{bmatrix}\left(\begin{bmatrix}0&0&0\\1&3&3\\0&0&0\end{bmatrix}\left(\begin{bmatrix}3&0&4\\0&0&0\\6&1&2\end{bmatrix}\begin{bmatrix}x_1\\x_2\\x_3\end{bmatrix} + \begin{bmatrix}7\\0\\9\end{bmatrix}\right) + \begin{bmatrix}0\\7\\0\end{bmatrix}\right) + \begin{bmatrix}5\\5\\2\end{bmatrix} \\ &= \begin{bmatrix}42&0&120\\20&0&27\\336&21&0\end{bmatrix}\begin{bmatrix}x_1\\x_2\\x_3\end{bmatrix} + \begin{bmatrix}695\\376\\562\end{bmatrix} \end{aligned}$$

Region of validity:

$$\begin{bmatrix}3&0&4\\6&1&2\end{bmatrix}\begin{bmatrix}x_1\\x_2\\x_3\end{bmatrix} + \begin{bmatrix}7\\9\end{bmatrix} \geq \begin{bmatrix}0\\0\end{bmatrix} \qquad \begin{bmatrix}1&3&3\end{bmatrix}\left(\begin{bmatrix}3&0&4\\5&9&1\\6&1&2\end{bmatrix}\begin{bmatrix}x_1\\x_2\\x_3\end{bmatrix} + \begin{bmatrix}7\\2\\9\end{bmatrix}\right) + 7 \geq 0$$

$$\begin{bmatrix}5&9&1\end{bmatrix}\begin{bmatrix}x_1\\x_2\\x_3\end{bmatrix} + 2 < 0 \qquad \begin{bmatrix}2&9&5\\8&7&0\end{bmatrix}\left(\begin{bmatrix}3&0&4\\5&9&1\\6&1&2\end{bmatrix}\begin{bmatrix}x_1\\x_2\\x_3\end{bmatrix} + \begin{bmatrix}7\\2\\9\end{bmatrix}\right) + \begin{bmatrix}7\\8\end{bmatrix} < \begin{bmatrix}0\\0\end{bmatrix}$$

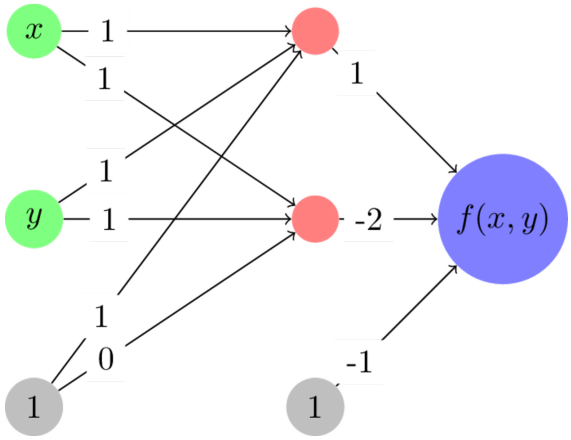
Figure 5.1: An illustration of how the ReLU activation pattern for an input determines the linear mapping used for that input. The region of validity refers to the possible x values for which this ReLU activation pattern exists. All the equations must be satisfied. All numbers shown here have been chosen arbitrarily.

$$w = \begin{bmatrix} 1 \\ -2 \end{bmatrix} \quad c = -1$$
$$\mathbf{A} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \quad b = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \tag{5.11}$$

as shown in Figure 5.2.

x	y	$f(x,y)$
-1	-1	-1
1	-1	1
-1	1	1
1	1	-1

(a) The input and output values of the modified XOR problem.



(b) A network that solves this modified XOR problem.

Figure 5.2: The modified XOR problem. (a) the input and output values — inputs and outputs are rescaled to be from -1 to 1 rather than from 0 to 1. (b) A network architecture and its associated weights that solves this problem. Nodes in red have ReLU applied after calculating their associated input values.

When given the input

$$x = \begin{bmatrix} -1 \\ 0.5 \end{bmatrix}, \tag{5.12}$$

the interior of the ReLU becomes

$$\mathbf{A}x + b = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} -1 \\ 0.5 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.5 \\ -0.5 \end{bmatrix}, \tag{5.13}$$

so the first element of this vector is "activates" its associated ReLU, as it is greater than zero, and the second element does not, as it is less than zero. We can represent this as the bit string 10 since

the "first" (upper) ReLU is activated and the "second" (lower) ReLU is not. Such bit strings can be useful for comparing activation patterns of a given neural network across inputs using metrics such as Hamming distance [77, 78]. The ordering of which bit in the string corresponds to which ReLU in the network is not necessarily meaningful, beyond a necessity of consistency when evaluating a network, due to the arbitrariness of the ordering of elements within a layer of the network.

The activation pattern can be computed for all possible inputs to the network by determining for which values of x_1 and x_2 each ReLU will be activated, and what the resulting weight terms will be. By doing that for this network we see that the piecewise linear formulation of this network is

$$v(x) = \begin{cases} \text{linear term} & \text{polytope} & \text{bit pattern} \\ -x_1 - x_2 + 1, & 0 \leq x_1 + x_2, & 11 \\ x_1 + x_2 + 1, & -1 \leq x_1 + x_2 \leq 0, & 10 \\ -4x_1 - 4x_2 + 3, & 0 \leq x_1 + x_2 \leq -1, & 01 \\ -1, & x_1 + x_2 \leq -1, & 00 \end{cases} \tag{5.14}$$

The way that this structure gives rise to a piecewise linear function and the associated decision boundary is illustrated in Figure 5.3. Keep in mind that the decision boundary does not have to align to the boundaries of the polytopes, as in this case it is the zeroes of $v(x)$.

The region corresponding to the bit pattern 01 is empty — it requires $x_1 + x_2$ to be greater than 0 and less than -1. Each ReLU ostensibly splits its input space in half, so it may appear that for k ReLU functions, there will be 2^k regions. However, each ReLU's activation corresponds to a line in space and k lines do not always yield 2^k partitions of a space. In this specific instance, this comes because the two activation boundaries are parallel. More generally, for single hidden layer networks, k lines (or hyperplanes) in n dimensional space can partition space into at most $\sum_{i=0}^{n} \binom{k}{i}$ regions [117]. When $k <= n$, this simplifies to 2^k, but if we have three lines in two-dimensional space, we can have at most

$$\binom{3}{0} + \binom{3}{1} + \binom{3}{2} = 1 + 3 + 3 = 7 \tag{5.15}$$

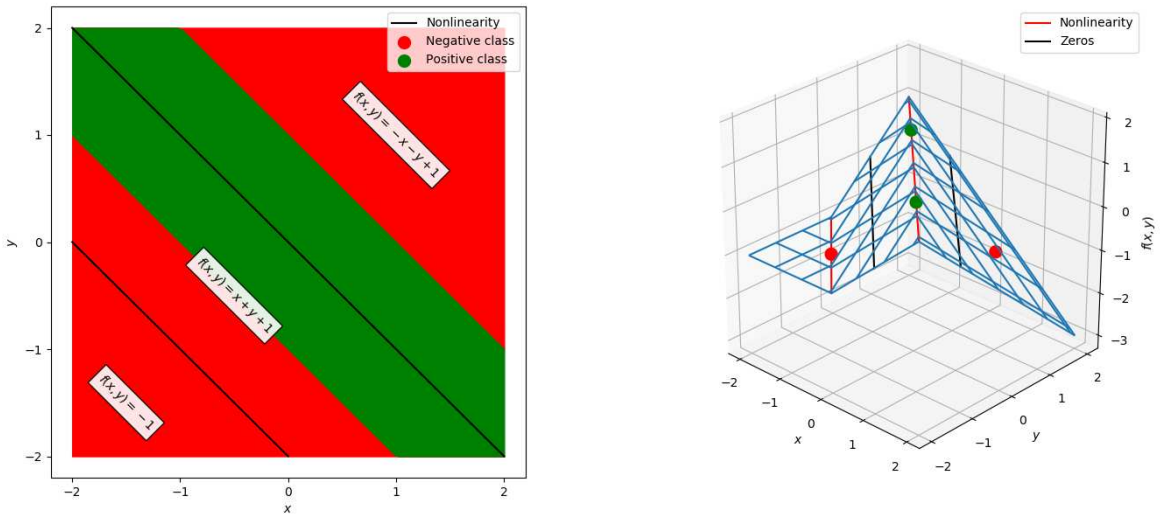


Figure 5.3: The polytopes and associated linear coefficients for a simple network to solve the XOR problem. Left: the cross-section of the network outputs in the plane. Green corresponds to points that would be labeled in the positive class (neural network output greater than zero) and red corresponds to points that would be labeled in the negative class (neural network output less than zero). The black lines correspond to the points at which one of the two ReLU units "activates" or "deactivates" and switches the linear region used for classification. The three polytopes form bands in the plane. Right: the surface of the neural network. The points used for training are shown as green and red dots, the nonlinearities are shown as red lines, and the decision boundary (zeros of the network) are shown as black lines.

regions, which is less than $2^3 = 8$. For deep networks, the upper bound on the number of regions is a product of these sums, with complexity bounded by a term that is polynomial in the width of the network and exponential in the depth [14] (although in practice, fewer terms are used within the domain of meaningful data [17]).

An illustration of this for a more complex network is shown in Figure 5.1. In that image, an architecturally larger neural network with arbitrarily chosen weights is constructed, and it's shown how a given bit pattern defines a region and linear term.

5.3 One-Dimensional Example — Sine

For an example of this with one input dimension and one output dimension, we can consider the problem of approximating sine in the region $[-\pi, \pi]$ using fixed numbers of lines. As shown in Figure 5.4, using 1, 3, or 9 lines interpolating between the values of sine on uniformly spaced points in $[-\pi, \pi]$ can result in increasingly reasonable approximations. Additionally, by looking at the slopes and biases of each of those lines, we can visualize the "density" of sampling that the multiple resolutions are using to approximate the sine curve.

Although general linear approximations are quite interesting, we are more interested in observing neural network behavior. We can apply the same technique for visualization, where the linear approximation and associated weights and biases are shown. In addition to overlaying the approximate function over the sine function, visualizing the coefficients allows us to see how close each region is to the "perfect" linear representation and see in a different way how the network's attention corresponds to different areas in the $[-\pi, \pi]$ region.

To form a baseline for error, we consider the zero function as an approximation. The mean squared error in that case will be

$$\text{MSE}(\sin(x) - 0) = \frac{1}{\pi - (-\pi)} \int_{-\pi}^{\pi} \sin^2(x) dx = 0.5. \tag{5.16}$$

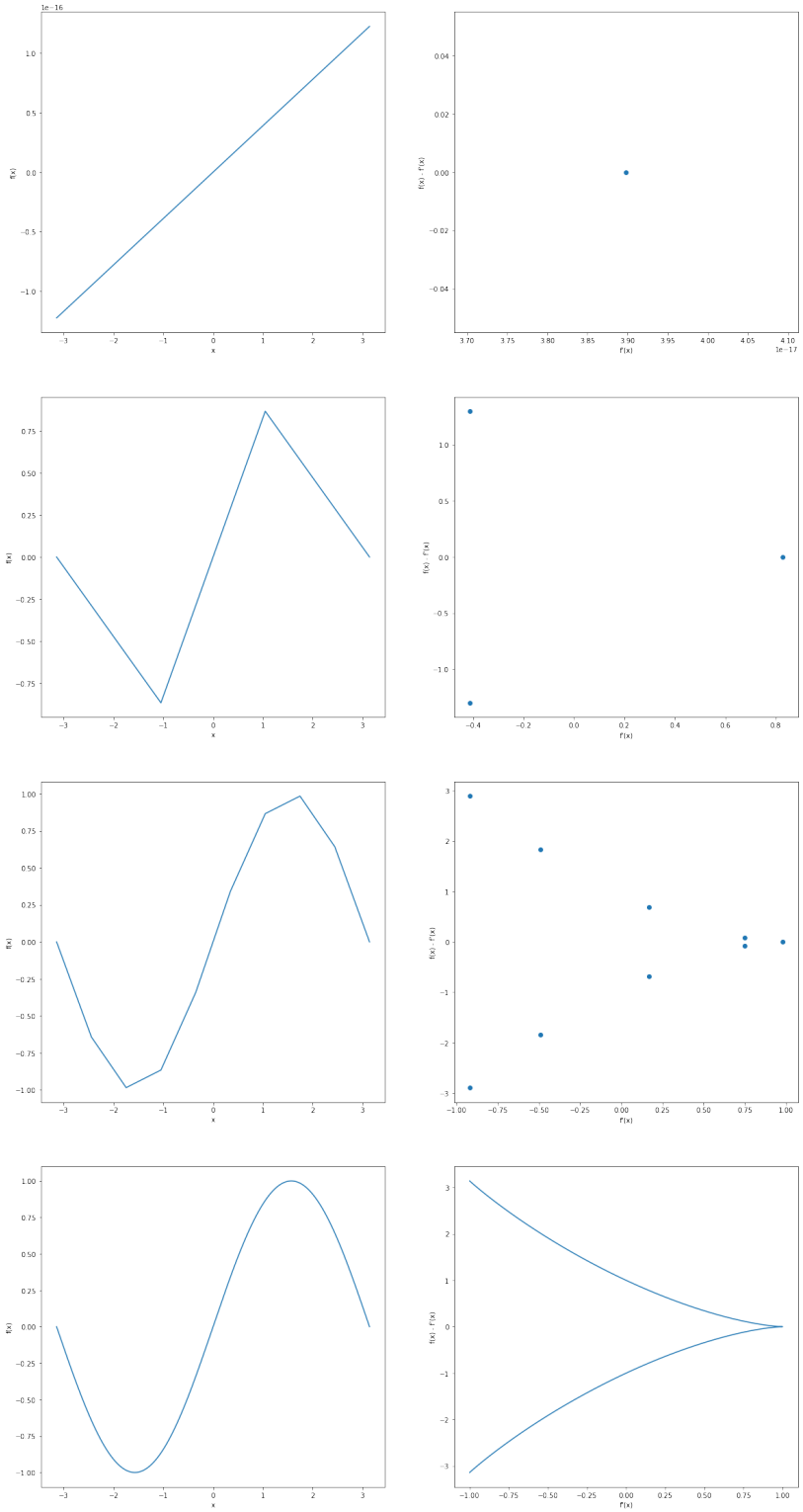


Figure 5.4: An example of multiple resolutions of linear approximations to sine (left) and the corresponding distributions of the weights and biases of their linear coefficients (right). Approximations use one (top), three (second), or nine (third) piecewise linear regions splitting $[-\pi, \pi]$ into uniform regions. The bottom image shows sine and the slope and bias of the tangent line to sine at each point along $[-\pi, \pi]$.

If a network performs equal to or worse than this baseline, it has not made significant progress towards representing sine. To ensure we get "good" initialization of the networks that lead to good accuracy, we train fully-connected networks with varying widths and depths over 200 epochs using stochastic gradient descent with a mean squared error loss function, a learning rate of 0.1, and 100 uniformly spaced points in $[-\pi, \pi]$ as the training data. If the network yields an MSE of more than 0.45, we reinitialize it and try training it again, up to 1,000 times. If the MSE is less than 0.45, the network is trained for an additional 1,000 epochs to ensure that it converges. The number of network initializations used for each width and depth combination is available in Table 5.1.

Table 5.1: Number of network initializations taken to attain a network with a mean square error below 0.5 when trained to approximate sine on $[-\pi, \pi]$. Networks are rectangular with a number of hidden layers equal to depth and each layer having width nodes. Values of $-$ correspond to more than 1,000 trials.

		Depth									
		1	2	3	4	5	6	7	8	9	10
Width	1	1	1	3	4	1	18	51	333	-	-
	2	1	1	1	2	1	3	14	54	39	392
	3	1	1	1	1	1	5	2	13	7	502
	4	1	1	1	1	1	2	2	33	76	-
	5	1	1	1	1	1	1	2	6	50	599
	6	1	1	1	1	1	1	2	6	70	112
	7	1	1	1	1	1	1	3	9	55	444
	8	1	1	1	1	1	1	1	3	2	428
	9	1	1	1	1	1	1	1	1	30	-
	10	1	1	1	1	1	1	1	5	91	171

This is a less structured version of the experiment in Section 3.4.2 that demonstrates how different sizes of network have different likelihoods of finding "good" solutions. For a full experiment, more care would be taken in hyperparameter selection, especially with number of epochs, and a more careful averaging would be undertaken to determine how many networks "succeed." However, this partial experiment does highlight a few interesting pieces — deeper networks tend to fail more frequently, likely stemming from the fact that deep networks have a chance for all ReLUs in a layer have inputs below zero based on poor initialization. This results in the network not having any signal with which to train, so the weights are not able to be changed from their initial configu-

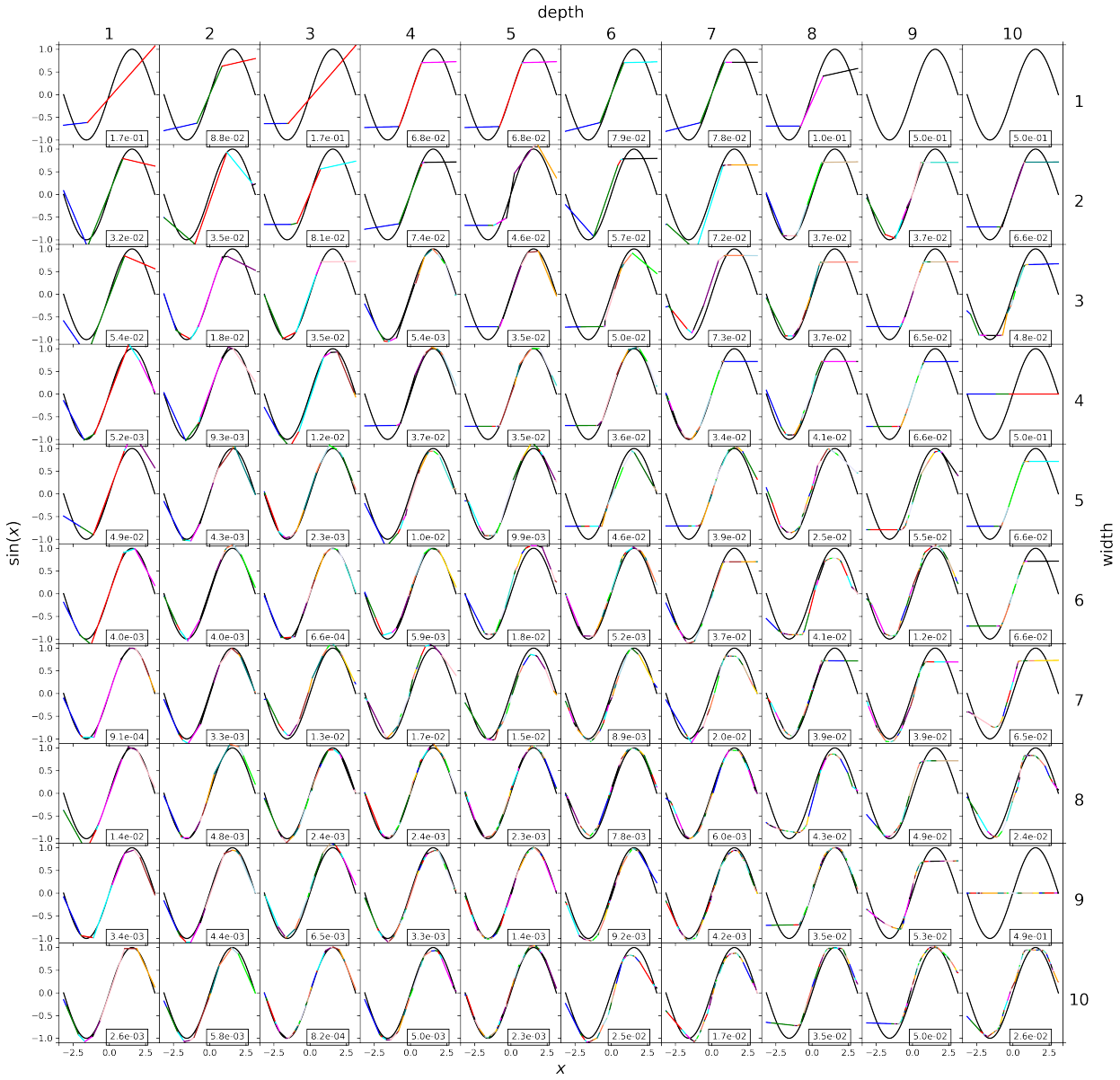
rations and the network stays as a constant function. As the width increases, this effect decreases due to the increased number of paths. This is one, relatively minor, reason why overparameterization can be useful when training neural networks, contrary to traditional understandings around overfitting. This detail is meaningful for these relatively small networks, but for larger networks there are more dominant elements as discussed by LeCun *et al.* [29].

Figure 5.5 shows how rectangular fully-connected neural networks approximate sine. Each row corresponds to a different number of nodes in each hidden layer, and each column corresponds to a different number of hidden layers. Both the actual piecewise lines and the distribution of the weights of those lines are shown to demonstrate how the number of linear regions corresponds to (and frequently does not necessarily correlate with) the accuracy of the approximation of the sine curve. Because the network is attempting to represent a differentiable function, the structures shown in Figure 5.5b are an alternative method for viewing accuracy. Networks that work well have linear coefficients that correspond to the exact linearized weights and biases of the sine function.

The patterns of linear regions, especially as shown in Figure 5.5 demonstrates how the number of regions increases both in width and depth, in this case with multiplicative returns. It also shows how regions can cluster. Especially in the set of networks with width 10, as the depth increases, the networks tend to approximate the negative portion of sine less well. This appears as fewer coefficients in the lower half of the plot. However, these visualizations are relatively simple — in the one-dimensional case, the linearizations and the functions themselves are closely related and the geometry of the "regions" created is simple. Increasing to two or more dimensions allows for more complex behavior, as will be discussed in Chapter 6.

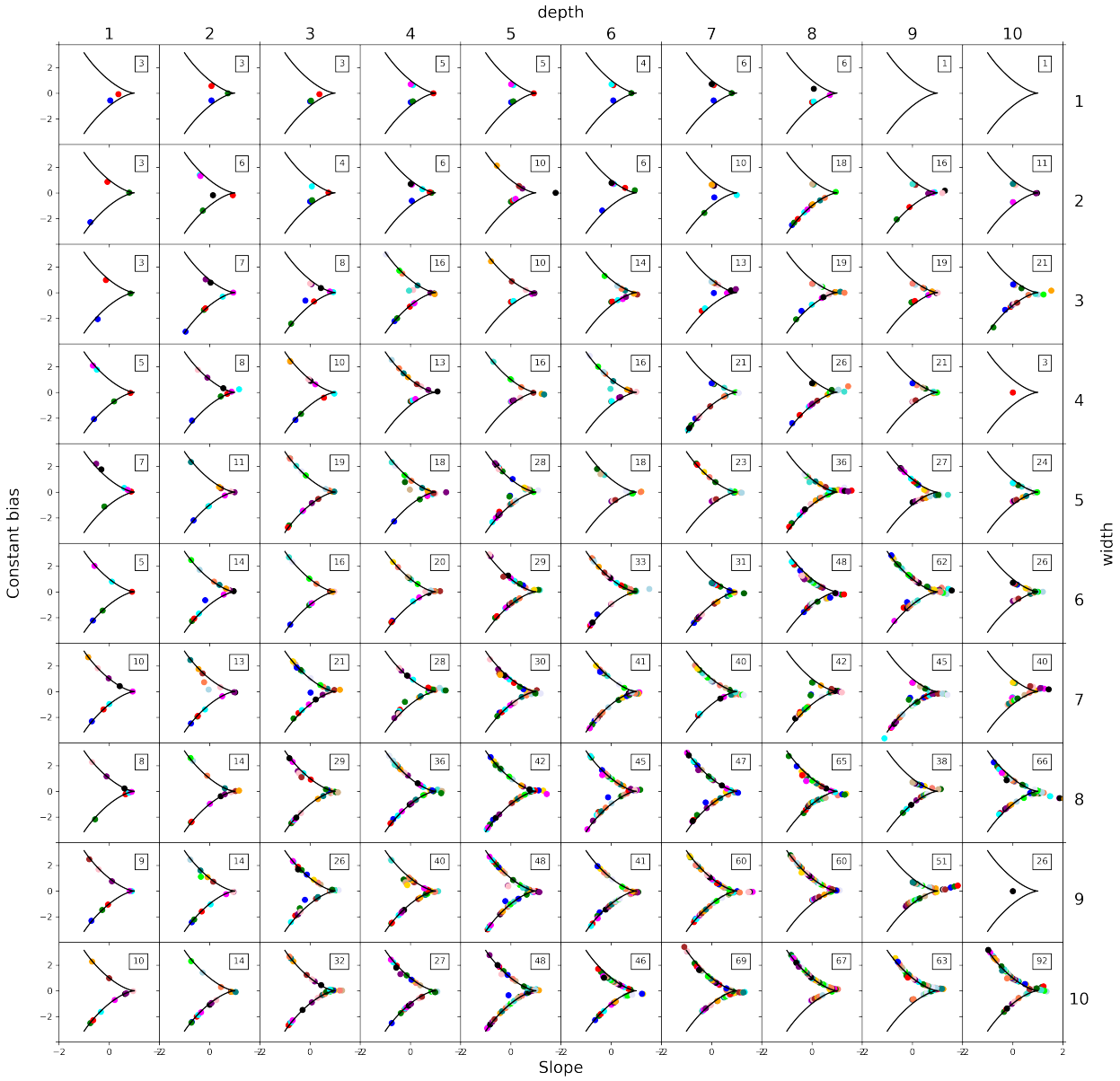
5.4 Tropical Formulation

In addition to the piecewise linear formulation of neural networks, neural networks can be represented through the lens of tropical algebraic geometry. This has been previously investigated, as discussed in Section 2.3.9, by Zhang *et al.*to replicate work demonstrating the number of bounds on linear regions in a novel way [20]. Tropical algebraic geometry is based on the tropical semiring,



(a) Approximations

Figure 5.5: An example of linear regions and their associated distributions of linear coefficients. Various fully-connected networks with different widths (nodes per layer) and depths (number of layers) were trained to approximate sine on the domain $[-\pi, \pi]$. Each row corresponds to a different width, going from a width of one at the top to a width of ten at the bottom. Similarly, each column corresponds to a different depth, going from a depth of one at the left to a depth of ten at the right. 5.5a shows the resulting network approximation versus the actual sine curve. Boxed numbers are the mean squared error across the domain for the given network. 5.5b shows the coefficients corresponding to the resulting linear regions. The horizontal axis of each plot represents the slope (b in $f(x) = a + bx$) associated with the linear region — the first derivative of the function at that point. The vertical axis represents the constant term (a in $f(x) = a + bx$). Boxed numbers are the number of linear regions used within the domain for the given network.(continued on next page)



(b) Weights

Figure 5.5: An example of linear regions and their associated distributions of linear coefficients. Various fully-connected networks with different widths (nodes per layer) and depths (number of layers) were trained to approximate sine on the domain $[-\pi, \pi]$. Each row corresponds to a different width, going from a width of one at the top to a width of ten at the bottom. Similarly, each column corresponds to a different depth, going from a depth of one at the left to a depth of ten at the right. 5.5a shows the resulting network approximation versus the actual sine curve. Boxed numbers are the mean squared error across the domain for the given network. 5.5b shows the coefficients corresponding to the resulting linear regions. The horizontal axis of each plot represents the slope (b in $f(x) = a + bx$) associated with the linear region — the first derivative of the function at that point. The vertical axis represents the constant term (a in $f(x) = a + bx$). Boxed numbers are the number of linear regions used within the domain for the given network. (Cont.)

$\{\mathbb{R} \cup \{+\infty\}, \oplus, \otimes\}$ where tropical addition, $\oplus$, and tropical multiplication, $\otimes$, are defined by

$$\begin{aligned} x \oplus y &= \max\{x, y\} \\ x \otimes y &= x + y. \end{aligned} \tag{5.17}$$

For the purposes of neural networks, it is useful to restrict ourselves to the rational numbers so that we can define exponentiation,

$$x^{\odot y} = x * y \tag{5.18}$$

where $*$ is standard multiplication. This allows us to define tropical monomials,

$$f(x) = c \otimes x^{\odot a} = a * x + c, \tag{5.19}$$

tropical polynomials,

$$F(x) = c_1 \otimes x^{\odot a_1} \oplus ... \oplus c_n \otimes x^{\odot a_n} = \max\{a_1 * x + c_1, ..., a_n * x + c_n\}, \tag{5.20}$$

and tropical rational functions as the tropical division (i.e. standard subtraction) of two tropical polynomials. These definitions can be extended to vector values — vector tropical exponentiation becomes the dot product, vector tropical multiplication is vector addition, and vector tropical addition is the max function applied elementwise.

With these definitions, it has been shown that the following are equivalent [20]:

- the set of ReLU neural networks with rational valued weights,
- the set of tropical rational functions with rational coefficients,
- the set of piecewise linear functions with rational coefficients.

This means that such neural networks, in addition to being transformable into piecewise linear functions, can be transformed into tropical rational functions. For example, the network

$$f(x) = \begin{bmatrix} 1 & -2 \end{bmatrix} \max \left\{ \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} x + \begin{bmatrix} 1 \\ 0 \end{bmatrix}, 0 \right\} - 1, \tag{5.21}$$

which is designed to solve a modified version of the XOR problem, can be transformed into

$$\begin{aligned} f(x) &= \max\{-1, -1 + x_1 + x_2\} - \max\{0, 2x_1 + 2x_2\} \\ &= (-1 \oplus -1 \otimes x_1 \otimes x_2) \oslash (0 \oplus -2 \otimes x_1^{\odot 2} \otimes x_2^{\odot 2} +). \end{aligned} \tag{5.22}$$

Unfortunately, calculating such transformations is *costly*. Networks have exponentially many terms, much as they do in the piecewise linear functions. Calculating the tropical polynomials requires calculating each possible term, then computing the (high-dimensional) convex hull to determine which monomial terms are attainable as the maximum. Trimmel *et al.*constructed an algorithm for sampling from monomial terms from these rational functions [118], but we focus instead on the piecewise linear interpretation due to these difficulties.

5.5 Equivalencies

Neural networks can be represented in many distinct ways. The standard formulation is

$$v(x) = A_L \max\{A_{L-1} \max\{... \max\{A_1 x + b_1, 0\}\} + b_{L-1}, 0\} + b_L \tag{5.23}$$

for weight matrices $A_1, ..., A_L$ and bias vectors $b_1, ..., b_L$. This representation is convenient for using the network. Calculating the value of the network at a given x is simple, and determining how the weights should be modified to improve performance is doable through backpropagation, as discussed in Section 2.2.1.

Alternatively, the network can be represented as a tropical rational function:

$$v(x) = \max\{F_1 x + m_1, F_2 x + m_2, ..., F_f x + m_f\} - \max\{G_1 x + n_1, G_2 x + n_2, ..., G_g x + n_g\} \tag{5.24}$$

for matrices $F_1, ...F_f, G_1, ..., G_g$ and vectors $m_1, ..., m_f, n_1, ..., n_g$. This representation is impractical for experimental use, as it has exponentially many terms and calculating F, G, m, and n values is difficult. However, it can be used to draw meaningful theoretical results, as discussed in Section 5.4.

Networks can also be represented as piecewise linear functions:

$$v(x) = \begin{cases} \mathbf{J}_1 x + y_1, & Q_1 x \leq p_1 \\ \mathbf{J}_2 x + y_2, & Q_2 x \leq p_2 \\ \vdots & \\ \mathbf{J}_r x + y_r, & Q_r x \leq p_r \end{cases} \tag{5.25}$$

where the $Q_i, i = 1, ..., r$ are matrices and p_i are vectors representing the boundaries of the the polytopes within which the network is linear. The $\mathbf{J}_i$ are the constant Jacobian matrices of v in the region $Q_i x \leq p_i$, and the constant y_i vectors can be calculated as $y_i = v(\tilde{x}) - \mathbf{J}_i \tilde{x}$ for any $\tilde{x}$ such that $Q_i \tilde{x} \leq p_i$. We refer to the $\mathbf{J}_i$ and y_i as linear coefficients, which are exact linear approximations to a network within a linear region defined by Q_i and p_i. The number of linear regions, r, has the potential to be large — when the widths of all layers are at least the input dimension, d, it grows as $\mathcal{O}(w^{dl})$, where w is the maximum width of a layer in the network and l is the number of hidden layers [12, 14, 20]. This formulation is not useful for evaluating the output of the network or training, but the $\mathbf{J}_i, y_i, Q_i$, and p_i values can be sampled. This is the primary formulation we use throughout the rest of the paper, as the $\mathbf{J}_i$ and y_i values can be used to investigate various network behavior.

5.6 Conclusions

ReLU neural networks naturally form piecewise linear functions, and the structure of ReLU enforces structure on the geometry of the pieces. There are many equivalent representations arising

from this, with the neural network forming a way to effectively compress the piecewise linear structure into a small number of weights relative to the number of pieces.

Unfortunately, analyzing the piecewise structure directly is difficult for practical networks. The number of regions attainable by a network architecture grows exponentially in the depth and polynomially in the width. Gaining information across the full domain, then, becomes infeasible for large networks. Instead, as will be discussed in the next chapters, it is useful to investigate simple problems to build intuition and then use statistics on samples of the linear regions.

Chapter 6

Polytope Structure

When considering linearizations of ReLU neural networks, we have two primary elements of interest — what is the region in which it is valid, and what is the actual linearization? This chapter focuses on the first question, investigating what the piecewise linear structure of ReLU neural networks looks like.

We first state the datasets and networks we investigate the polytope structure of. Then, we build intuition by investigating simple two-dimensional problems and determining how the piecewise linear structure is formed, what behavior it has, and what restrictions are put in place by the iterated structure of the network and ReLU activation functions. We then consider image-processing datasets, visualizing cross-sections of their polytope boundaries to get a sense for how behaviors translate to higher-dimensional problems. The results here are an extension of previous work by the author [119].

6.1 Networks and Datasets Considered

To build intuition for the behavior of these linearization schemes, we consider two-dimensional toy problems. Although the sine function was discussed in Section 5.3, one-dimensional problems have significantly reduced complexity of polytopes. Any polytope in one-dimension will be a line segment, which does not have sufficient geometric complexity to be of interest.

We consider two main two-dimensional problems — that of classifying a circle versus a surrounding annulus and a combination of that problem with XOR, as shown in Figure 6.1. Because these problems have two-dimensional inputs, visualization of the polytope structures and distribution of the local linear coefficients are possible. Increasing to three dimensions means that the linear coefficients with the bias would be four-dimensional and that the polytope structure would be three-dimensional. Restricting ourselves to two-dimensions allows us to build intuition for what kinds of behavior these structures may exhibit. This restricts our analysis, as these are simple

enough problems that they do not necessarily accurately describe the behavior of more complex networks, but they still demonstrate meaningful behavior that can be applied to higher-dimensional problems.

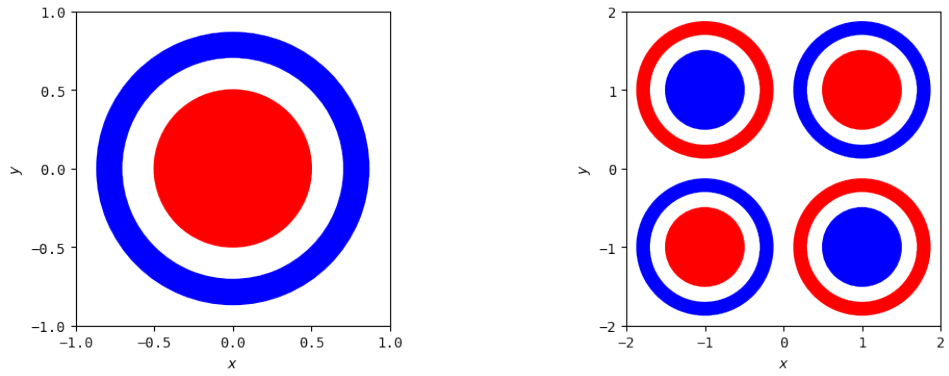


Figure 6.1: Two two-class classification problems used to illustrate linearization behavior in a visualizable way. In both cases the goal is to classify blue and red points/regions differently. Left: A circle versus an annulus. Right: a combination of the center problem with the XOR problem to demonstrate more sophisticated network behavior.

When constructing neural networks to solve these problems, we use a variety of architectures. Networks used are rectangular, fully-connected, and trained using stochastic gradient descent with binary cross entropy loss.

In order to extend our investigations to more complex problems that more closely match applications, we also investigate the MNIST and CIFAR-10 image classification datasets [63, 64]. Example images from these datasets are shown in Figures 6.2a and 6.2b. We investigate image classification problems as they are an area of significant practical interest with high-dimensional, complex behavior. The linear coefficients within a polytope are also in the form of an image, and they can be visualized to determine gradient information. This fact has been exploited in various ways to investigate saliency, although we do not focus on this topic [80]. The networks used for solving image classification problems also typically have sophisticated architectural elements, such as convolutions, that have the potential to change the qualitative behavior of the polytope structure.

MNIST is a dataset consisting of 28x28 grayscale images, each corresponding to a handwritten digit between 0 and 9 [63]. It has 60,000 images used for training and and 10,000 images used

(a) 28x28 MNIST images

(b) 32x32 CIFAR-10 images

Figure 6.2: Example images from the MNIST handwritten digits dataset and CIFAR-10 tiny images dataset. MNIST classes are the numbers zero through nine. CIFAR-10 classes are plane, car, bird, cat, deer, dog, frog, horse, ship, and truck.

for testing. MNIST is a relatively simple dataset, as linear classification can achieve upwards of 90% classification accuracy, but it is a good benchmark for determining if methods are plausible for more complex problems. However, it it allows us to use complex architectures that include convolution and pooling and provides a baseline for investigating how our intuition may map to higher dimensions. Images have their numeric representations normalized and centered at zero by channel before being used in networks.

CIFAR-10 consists of 32x32 RGB images taken from the Tiny Images dataset[10] corresponding to the classes airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck [64, 121]. CIFAR-10 serves as our "true" benchmark, although it is still considered a simple dataset relative to many modern image classification datasets which consist of high-resolution images with thousands of classes. Images have their numeric representations normalized and centered at zero by channel before being used in networks.

[10]Which has now been withdrawn due to inappropriate content [120], although CIFAR-10 has not been withdrawn as its selection of images and labels does not contain any of the inappropriate content.

We consider seven network architectures trained MNIST and CIFAR problems, with variations trained using both PyTorch's SGD with learning rate 0.01, momentum 0.5, and weight decay 0.01 and Adamw with default parameters [122]. Batch sizes of 32 are used, and 15 epochs are used for training. The networks are:

- linear — a linear classifier
- dense — a fully-connected feedforward network with a single hidden layer consisting of 128 ReLU nodes. Two identical variants of this network architecture are trained.
- simple — a convolutional network with two convolutional layers with 32 3x3 filters in each layer. Each convolutional layer is followed by a max pooling layer. ReLUs occur after max pooling. A single fully-connected layer occurs after the second max pool. Two identical variants of this network architecture are trained.
- complex — a convolutional network with three convolutional layers with 32 3x3 filters in each layer. Each convolutional layer is followed by a max pooling layer. ReLUs occur before max pooling. A linear combination of the outputs of the last max pool are used for classification. Two identical variants of this network architecture are trained.
- A network with the ResNet-18 architecture as implemented in Torchvision's models package using the pretrained parameters available in that package [11, 123].Images are upsampled, using bilinear interpolation, to the expected size of 224×224 pixels. The final linear classification layer is the only layer that is trained.
- A network with the ResNet-152 architecture as implemented in Torchvision's models package using the pretrained parameters available in that package [11, 123].Images are upsampled, using bilinear interpolation, to the expected size of 224×224 pixels. The final linear classification layer is the only layer that is trained.
- A network with the Inception-v3 architecture as implemented in Torchvision's models package using the pretrained parameters available in that package [123, 124]. Images are upsam-

pled, using bilinear interpolation, to the expected size of 229×229 pixels. The final linear classification layer is the only layer that is trained.

All non-pretrained networks are structured to accept three channel 32×32 images. For MNIST, images are scaled to 32×32 using bilinear interpolation, then repeated three times to match the number of channels expected. Accuracies are reported in Table 6.1. The linear network and pre-trained networks are included partially as controls — their behavior should be dissimilar from networks that can represent the problems and are trained on the problems.

Table 6.1: Accuracies of the networks trained using SGD and Adamw on MNIST and CIFAR

	SGD MNIST	Adamw MNIST	SGD CIFAR	Adamw CIFAR
linear	88.89%	86.83%	25.43%	31.66%
dense 1	96.76%	97.43%	51.01%	47.55%
dense 2	96.91%	97.24%	50.24%	48.41%
simple 1	97.80%	98.86%	68.14%	67.03%
simple 2	97.89%	98.90%	66.34%	67.58%
complex 1	97.73%	98.95%	61.72%	70.55%
complex 2	97.86%	98.81%	62.50%	71.33%
resnet18	96.64%	96.43%	80.52%	80.15%
resnet152	89.67%	91.12%	83.94%	82.57%
inceptionv3	91.54%	91.34%	77.05%	77.05%

The main reason we do not extend our investigations to more complex datasets, is the time and space complexity. Although datasets such as ILSVRC2012 are commonly used in benchmarking image recognition networks due to their increased complexity and resemblance to real-world applications, we are interested in calculating Jacobians for images coming from these datasets. The ILSVRC2012 challenge has 1,000,000 images spread over 1,000 classes and its size is 155.84 gigabytes. Calculating Jacobians for images in the dataset is therefore computationally costly, and storage of those gradients to avoid recomputation across various experiments is infeasible.

In addition to visualizing cross sections of the polytope structure, we can also visualize coefficients for each linear region as images. Previously, these weights were visualized as points in $d+1$ dimensional space, with the extra one coming from the constant term. For higher-dimensional

problems this becomes impossible to do naively. Instead, we can take the weights for a given input, each of which corresponds to a pixel in the input image, and visualize the magnitude and sign of that weight as an image. MNIST and CIFAR-10 both have ten outputs — meaning that each set of linear coefficients is now a Jacobian mapping from the input image to the ten outputs. Each row of the Jacobian can be visualized separately, showing what transformation of the input is done for each output. The polytope structure is the same for each output — the only information used to construct the output weights that varies is the final linear layer that maps from the final hidden layer to the outputs themselves, so the same ReLU activations and max pooling choices will be used for each output.

This visualization has image-related properties, but is not itself a true image. Continuing the example with MNIST, the linear weights associated with a given image lie in $\mathbb{R}^{28\times 28}$, as there is naturally one for each input dimension. This means they have much of the structure of a 28×28 grayscale image, and can be visualized by using the magnitude of the weight for each pixel to determine its color like one. However, the pixels for an MNIST image may only take one of 256 values — in raw form, the integers 0 through 255. This allows for a significant number of images, in this case $256^{(28*28)} \approx 10^{1888}$, but the weights do are not restricted to have integer, bounded, or positive values. In fact, the usage of unbounded positive and negative values of the weights is important for classification. These weights can still be mapped to a 28×28 grayscale image by transforming and scaling based on the minimum and maximum values (or more sophisticated methods to account for different scales across the linear weights associated with different images) then rounding, which allows for effective visualization of how the weights are applied to the input image. However, it is important to remember that these visualizations do not mean that the weights themselves form images in the input space.

Such weights for MNIST corresponding to the input zero shown as the fourth image in the first row of Figure 6.2a are shown in Figure 6.3. The visualization of these linear regions is similar to the idea of saliency mappings, although many modern forms of saliency mapping are more sophisticated than simply visualizing the gradients at an input image, as this is doing [73].

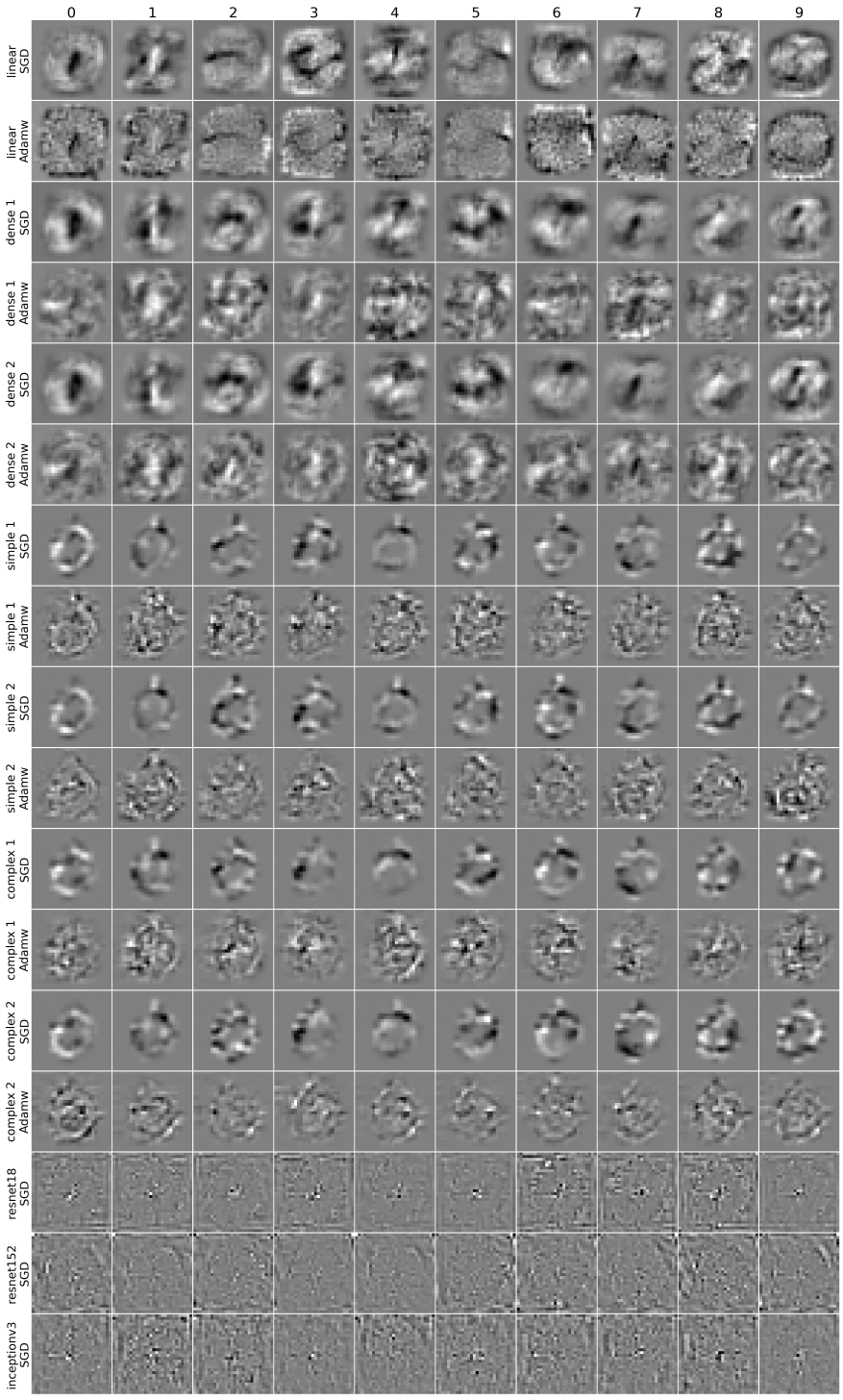


Figure 6.3: The linear coefficients for each output of the networks for the input image with value zero shown as the fourth image in the first row of Figure 6.2a. These visualizations are similar to simple forms of saliency mapping [73]. Gray corresponds to values near zero, white to positive values, and black to negative values.

The weights of individual regions, presented as "images" in the original input space (noting that the meaning of color has changed — black values are negative, white are positive, and the middling gray is zero) represent what information networks use for classifying individual samples. The linear networks have linear functions for each output that appear loosely as the number corresponding to the output. The dense networks using SGD have similar behavior, although the regions are now more "blurry." The convolutional networks have linear functions with non-zero values only corresponding to locations where the original image is nonzero. The ResNet and Inception networks, originally trained on ImageNet, have much less interpretable results with values appearing across the entire image. Edges of the zero shape are slightly visible. The linear functions coming from networks trained with Adamw are more jagged, with higher variation from pixel to pixel and are less understandable as specific shapes.

Based on the appearance of the images, the dense networks appear to have relatively little complexity, so they are classifying based on their "ideal" shape for each output, corresponding to what a linear classifier may do. This suggests a possibility for a reduction of number of linear regions used, discussed in Section 7.2. The other networks have more complexity, tend to focus more sharply on the relevant information being passed in, and classify based on that input. Networks trained with Adamw match less well to naively human-interpretable classification, suggesting a distinction in how they are training compared to the networks trained with SGD. The Adamw networks using the same architecture are also less similar than those with the same architecture trained using SGD. As will be discussed in Section 7.3, the Adamw networks have decreased similarity to themselves and SGD networks compared to the increased similarity of SGD networks to other SGD networks.

6.2 Small Network Polytope Structure

In addition to the tropical definition, we can also investigate the piecewise linear structure of such networks. In this section we look at small, two-dimensional input networks, and in Section 6.3, we look at large image-processing networks.

Polytope structures for networks trained on the toy circles problems discussed previously are in Figures 6.4, 6.5, and 6.6. Animations for how these polytope structures change through the training process are accessible in the YouTube links in the figure captions, and a selection of points in the training process for the simplest network on the single circle versus annulus process are shown in 6.4.

Investigation of how linear regions change through training has been studied [7, 16, 19]. However, these studies focus on MNIST and the usage of summary statistics to analyze behavior beyond the visuals in high-dimensional space. We focus on the visualization for two-dimensional input problems here, so that we can fully visualize the polytope structure and identify patterns of behavior across the entirety of the input.

These images and animations demonstrate some of the structural constraints on these polytopes. A single hidden layer can only construct lines in its input space, corresponding to when the linear transformation of the input is equal to zero, and use those to partition space into regions. However, when the input to a hidden layer is the output of another hidden layer, its input space has been compressed according to the previous layers transformations. This means that the "lines" it creates to partition space will angle at any of the "lines" from previous layers, as shown in Figure 6.7 and on the circle versus annulus problem in the bottom row of images in 6.5. This allows for successive layers to drastically increase the complexity of structures they can create.

This leads to one simple explanation for why a single hidden layer with three nodes is the simplest possible network for solving the circle versus annulus problem. To successfully solve this problem, a closed region needs to be created in space. Having three nodes in the hidden layer creates three lines in the input space leading to the triangular regions shown in the fully trained network in Figure 6.4. Each of the six unbounded triangular regions is associated with a nonzero linear function. The linear classifier that takes the output of those hidden nodes is then able to curve at each of the lines, leading to the hexagonal shape that forms the decision boundary. If there were only two nodes in the hidden layer, it would be able to form four square regions. However, one of those regions is necessarily constant, and so the curve traced by the linear classifier cannot

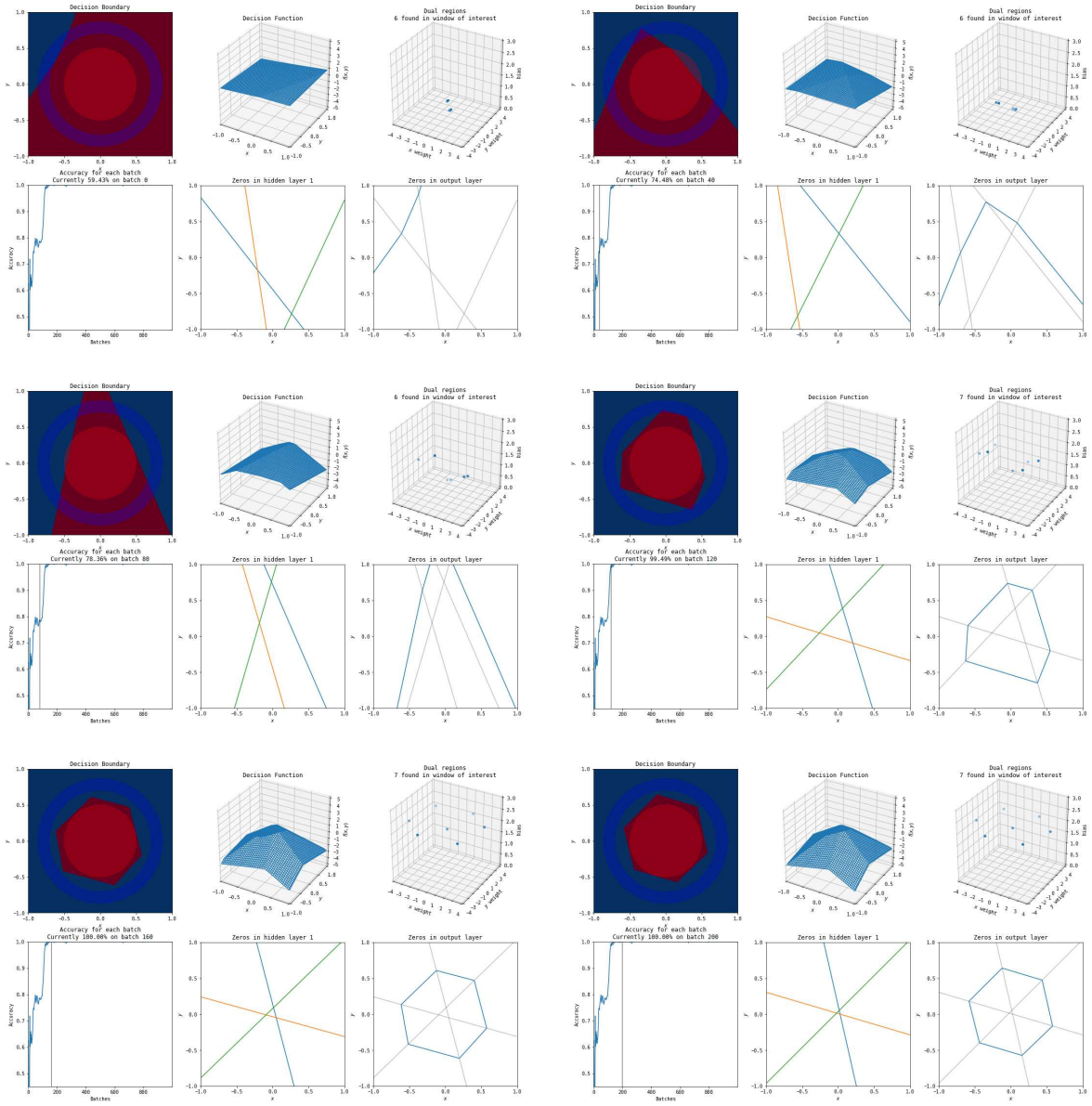


Figure 6.4: The training process of the simplest possible network (three hidden nodes in a single layer) on the circle versus annulus problem problem. An animation of this process is available at https://www.youtube.com/watch?v=lpXQI-UJIZM.

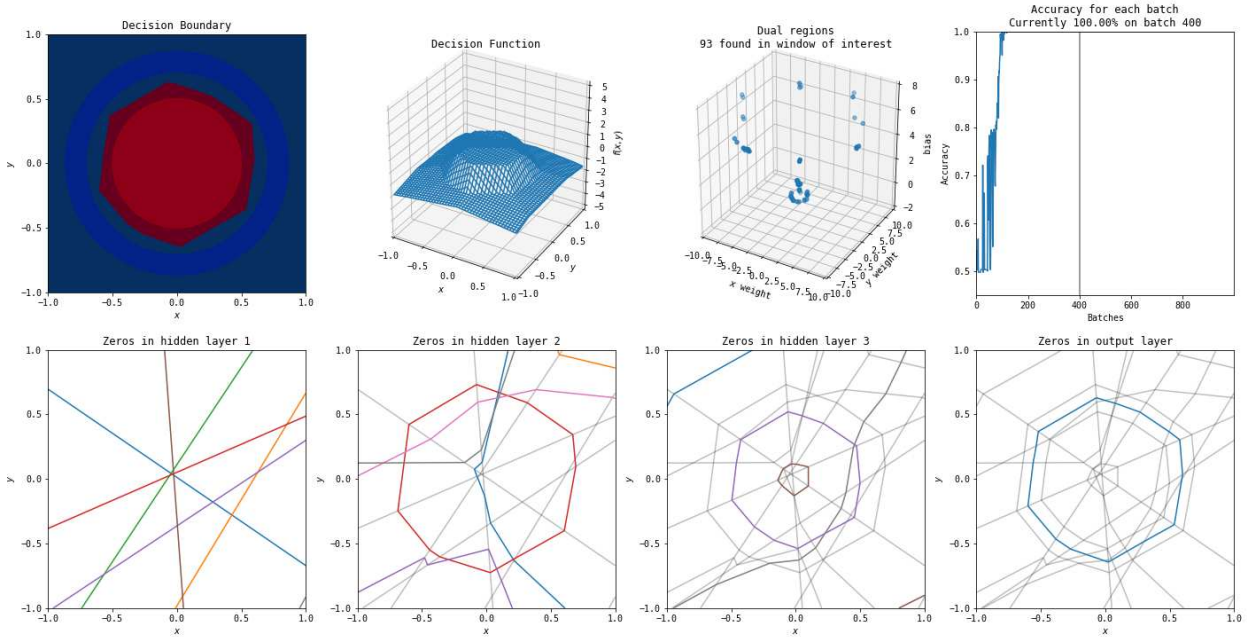


Figure 6.5: A solution to the circle versus annulus problem found by a more complex network (three hidden layers, each with eight nodes). An animation of the training process of this network is available at https://www.youtube.com/watch?v=rANyD9t-X-c.

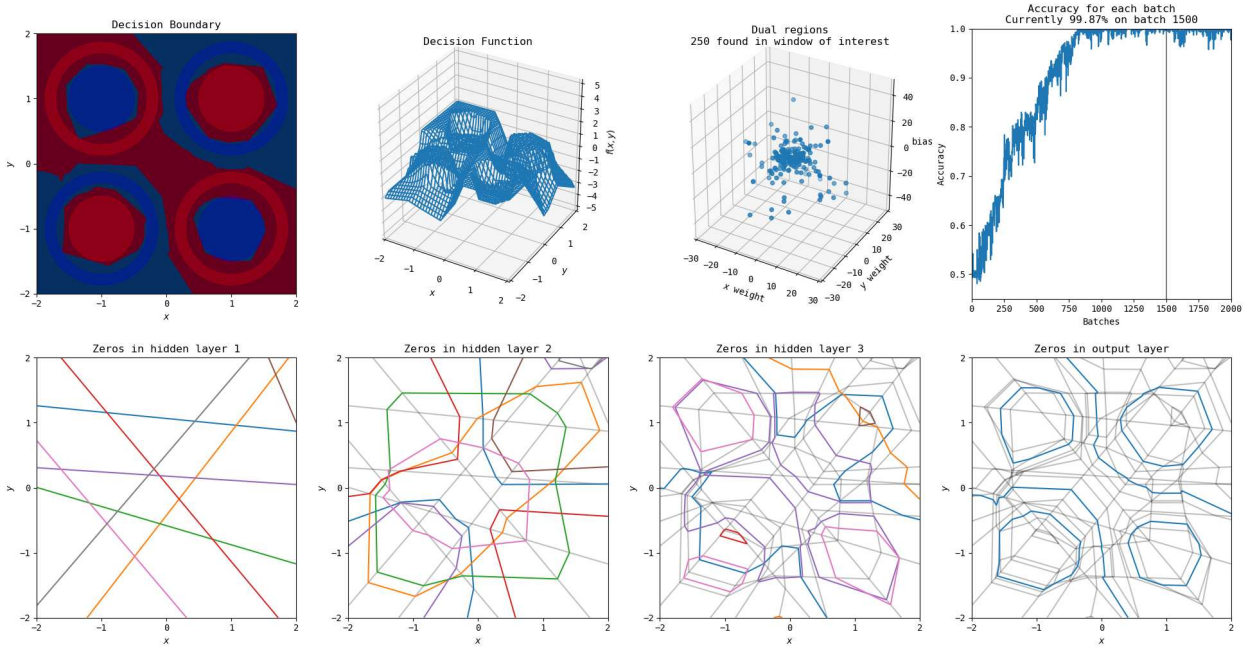


Figure 6.6: A solution to the XOR circle versus annulus problem found by a more complex network (three hidden layers, each with eight nodes). An animation of the training process of this network is available at https://www.youtube.com/watch?v=T_uoGBUOgUY.

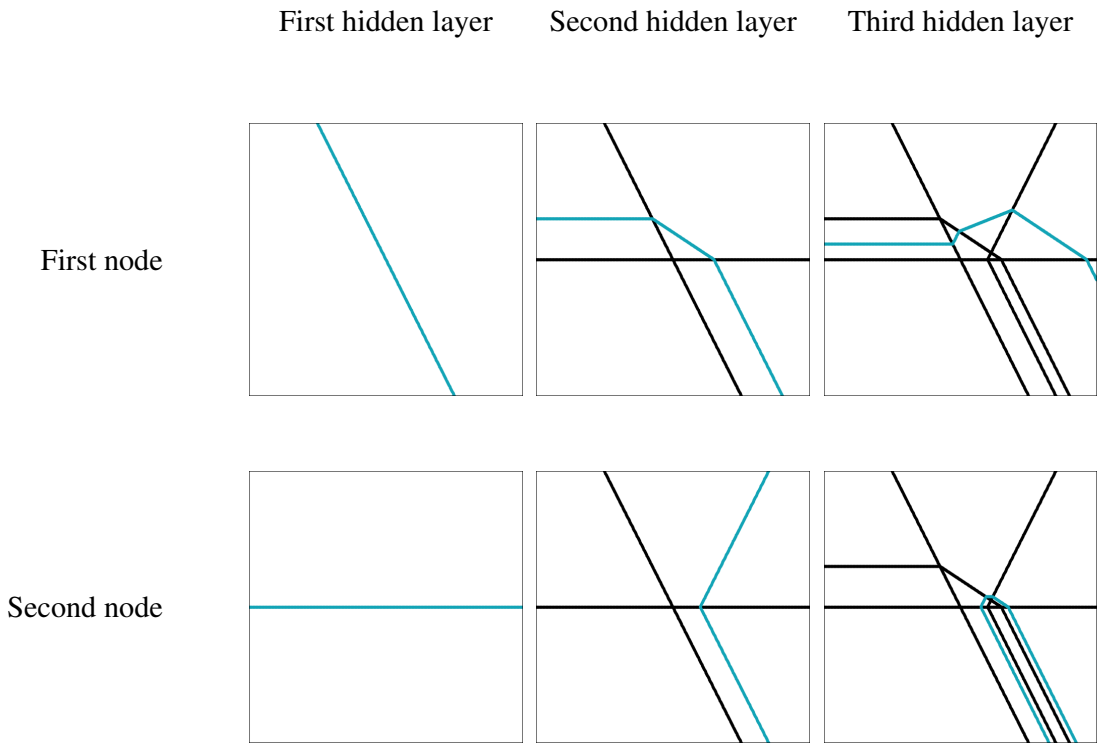


Figure 6.7: The polytope structure induced by increasing depth in a ReLU neural network. Each column constructs two nonlinearities based on the activations of the ReLUs (in blue) for each of the two nodes in that layer. Those nonlinearities "bend" the the line corresponding to the zeros of the ReLUs in layers further to the right, as the input space to those layers has been compressed.

interact meaningfully in that region. Other more analytical formulations of this have been created as elements of universal approximations theorems [7, 10], but this structure demonstrates some of the behaviors geometrically.

The visualization of linear coefficients in Figures 6.4 and 6.5 demonstrates that multiple networks can converge to similar clusters or structures of weights — in this case a central point, and then points arranged in a hexagon around it. This similarity in clustering suggests a potential structure that could be exploited to simplify or compare networks. This is investigated in more detail in Chapter 7.

The animations for the circle versus annulus problem also reveal that a complex network can find a solution similar to that of a simpler network. At roughly 0:25 in the animation for the complex network, the brown, red, and blue lines in the first hidden layer form the triangular structure that is evident in the simple network's solution. The shapes in subsequent layers form similar shapes to the hexagon in the simple network, but with additional complexity from other nodes. As the complex network converges further, a fourth line joins the three in the first layer, allowing for a polygon with more sides in the classification layer. Both networks, after they converge, also expand and contract due to the use of batches in training, but the additional complexity found by the larger network allows it to be more tolerant to the stochasticity.

In the XOR circle versus annulus problem, as training begins, the later layers shift significantly while the first hidden layer shifts slowly. This potentially corresponds to the result that networks tend to learn bottom-up — the earlier layers in networks are trained first, and then the later layers converge later [68]. The slower shifts in the early layers cause significant shifts in the later layers, meaning that the later layers learned representations are rendered obsolete. Once the first layer has converged further, the later layers are able to identify patterns that are successful based on the foundation that the earlier layers build.

6.3 Image-Processing Linear Region Structure

Although visualizations of the polytope structure and linear coefficients are relatively easy to construct for two-dimensional problems, shifting to image processing problems complicates matters. Even a simple dataset like MNIST has $28 \times 28 = 784$ dimensional inputs, and visualizing such a high-dimensional space fully is impossible. To gain insight despite this, we are able to visualize the polytope structures in a given cross section of the full space.

This is not a new idea — many pieces of existing work deal with generating these cross sections or using related algorithms to get information about network behavior [18, 19, 79]. However, those methods typically determine the linear bounds rather than the linear coefficients. For the piecewise linear form of a ReLU neural network, as defined in Section 5.5,

$$v(x) = \begin{cases} \mathbf{J}_1 x + y_1, & Q_1 x \leq p_1 \\ \mathbf{J}_2 x + y_2, & Q_2 x \leq p_2 \\ \vdots & \\ \mathbf{J}_r x + y_r, & Q_r x \leq p_r \end{cases} \tag{6.1}$$

many focus on using the Q_i and $p_i, i = 1, ..., r$ analytically to exactly determine polytope bounds. Other methods compare the Boolean strings corresponding to which ReLU functions are activated at points on a grid, with two points on the grid belonging to the same polytope if they share the same activation pattern.

This work instead identifies the $\mathbf{J}_i$ and y_i at points on the grid and generates the visualization by comparing distances between the linear coefficients, as will be discussed later in this section. Determining the Q_i and p_i or Boolean strings is more precise, but such algorithms are reliant on choices of the nonlinearities of the network. An algorithm that works for ReLU neural networks may not work for networks that use alternative piecewise linear functions such as hard tanh or max pooling. Using the linear coefficients also allows for extension to networks that are not piecewise

linear, although in such cases what will be measured is distance from a sample, rather than a (nonexistent) polytope structure.

Although this method is easier to *implement* and generalizes beyond piecewise linear networks, it is more computationally complex than simply using the activation patterns for comparison. Calculating the Jacobians requires taking the derivative of the network, which is costly in comparison to only calculating only how nonlinearities are activated. Comparing two activation patterns, which are Boolean strings for networks with only ReLU activations, is also cheaper than comparing two matrices whose dimensions are the problem's input and output dimensions.

When visualizing the hypersurfaces of high-dimensional data, we cannot visualize the entirety of the input space. Instead, we must look at cross sections, such as the ones constructed in Figure 6.8 for CIFAR-10 or in Figure 6.9 for MNIST using three points in space to generate a two-dimensional plane. This allows us to create cross-sectional polytope structures such as the ones shown in Figures 6.10 and 6.11. This doesn't give a full understanding of the dynamics of the high-dimensional polytopes, but it does give a glimpse into what level of complexity may occur.

To construct such a cross section, we select three images from the dataset to serve as the anchors. For those three images A, B, and C, each of which can be treated as a point in $\mathbb{R}^{3,072}$ for CIFAR and $\mathbb{R}^{784}$ for MNIST, we can construct a plane that passes through them. To construct this plane, we use the Gram-Schmidt process on the difference of A with B and C to create the axis vectors

$$\begin{aligned} u_1 &= B - A \\ u_2 &= C - A - \frac{(C-A)\cdot(B-A)}{||B-A||_2^2}(B-A) \\ x\text{-direction} &= \frac{u_1}{||u_1||_2} \\ y\text{-direction} &= \frac{u_2}{||u_2||_2}. \end{aligned} \tag{6.2}$$

The resulting plane can be centered at the point A with x and y axes corresponding to movement in the directions defined in Equation (6.2). This zero location does **not** correspond to the zero location in original image space, but instead to an arbitrary selection of where the resulting planar

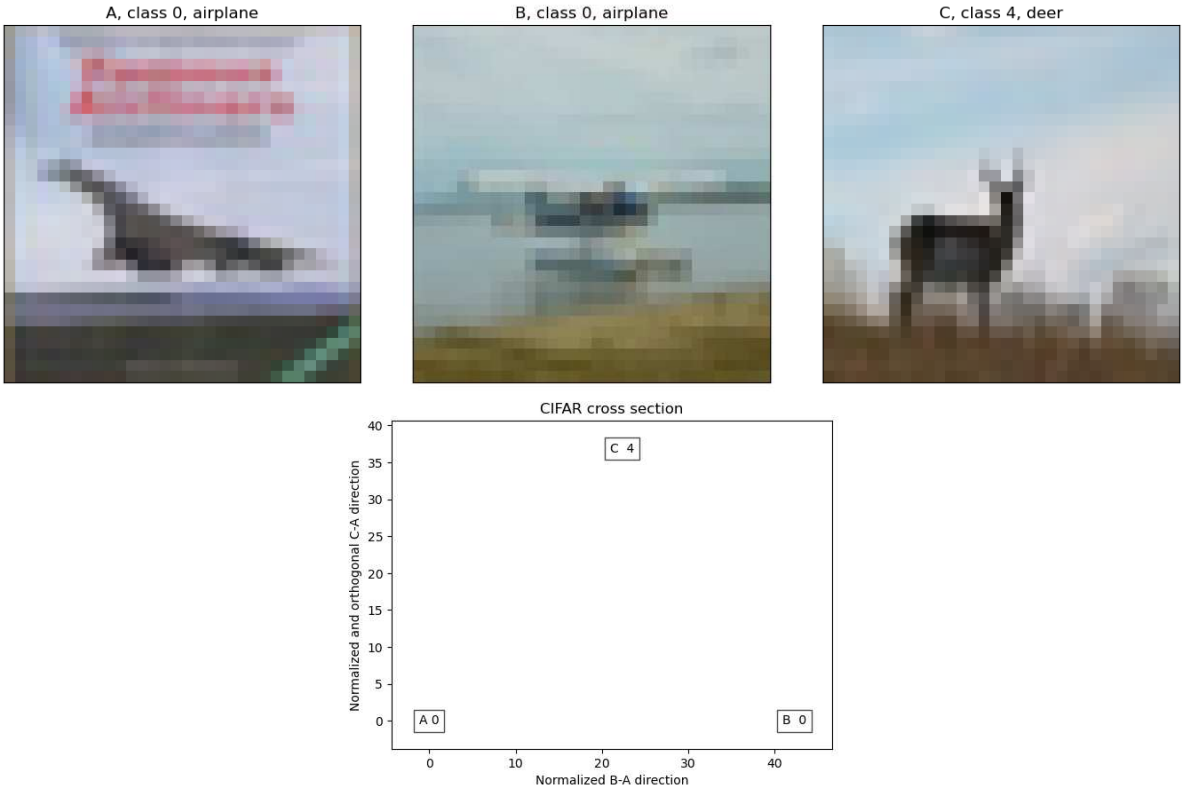


Figure 6.8: The images used to generate a cross-section of the 3,072 dimensional space of CIFAR-10 images. Images were selected to be minimally distant using Euclidean distance. Top: Two images of airplanes (left and center) and a picture of a deer (right) used to generate three points in image space. Bottom: An image showing the arbitrary scaling distance used in the resulting plane.

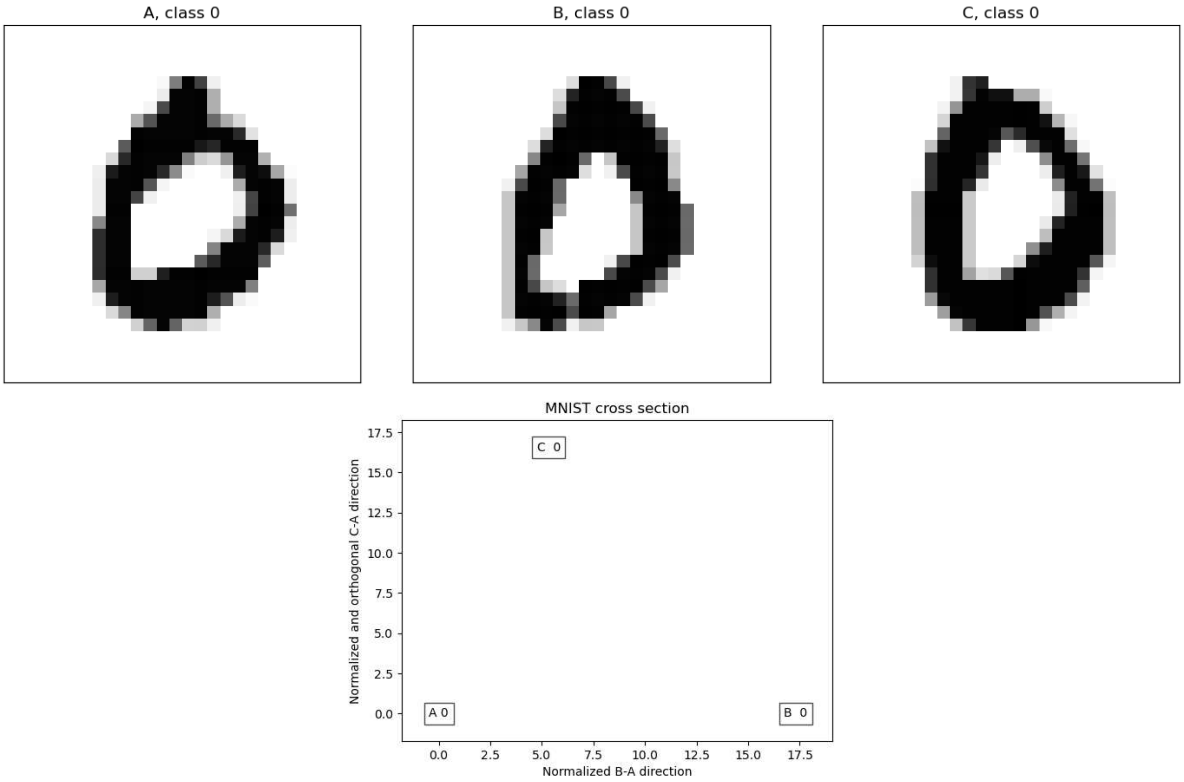


Figure 6.9: The images used to generate a cross-section of the 784 dimensional space of MNIST images. Images were selected to be minimally distant using Euclidean distance. Top: Three images of handwritten zeros used to generate three points in image space. Bottom: An image showing the arbitrary scaling distance used in the resulting plane.

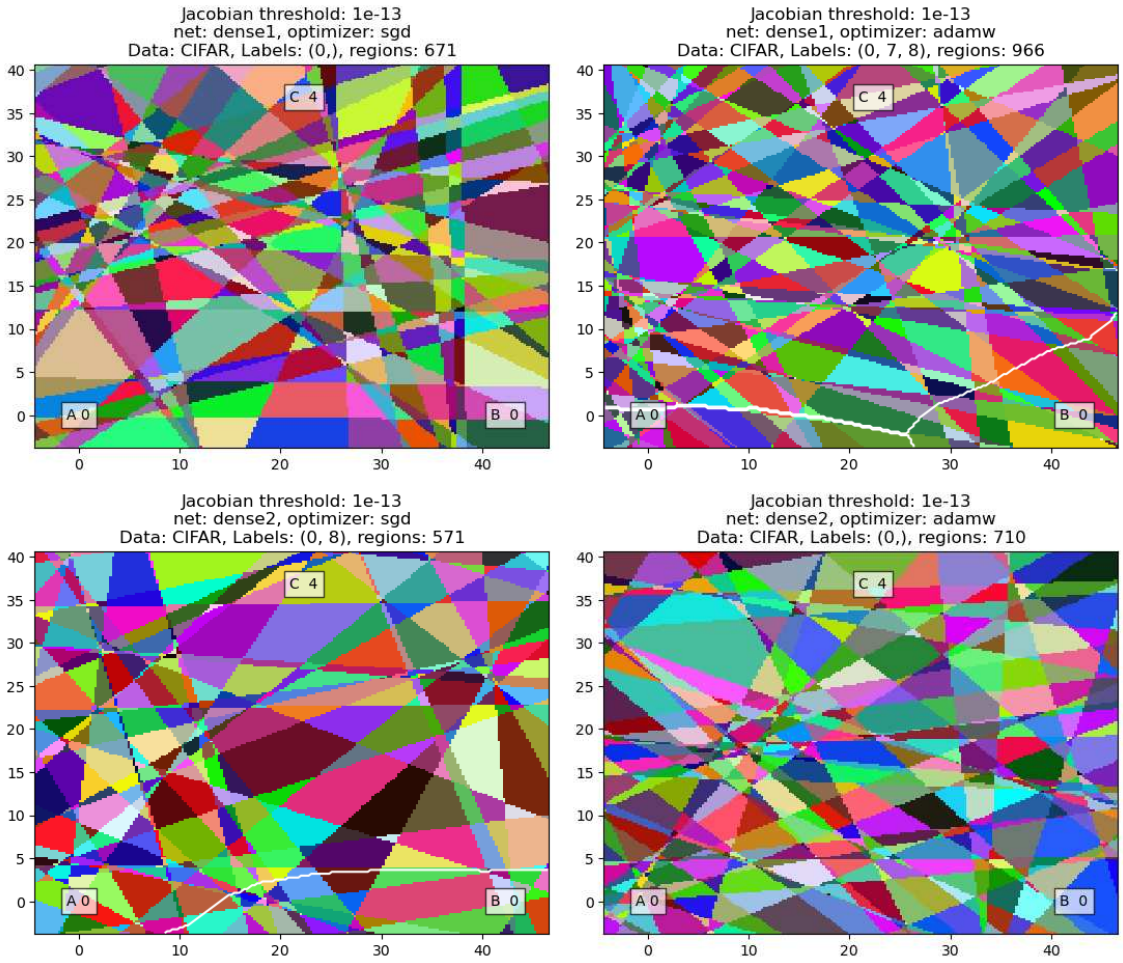


Figure 6.10: The polytope structure for the fully-connected networks on CIFAR-10. The cross-section is defined by the samples in Figure 6.8. Each colored region corresponds to a different linear function for the network. Colors are assigned randomly to assist in showing the variation. White lines correspond to boundaries in classification output.

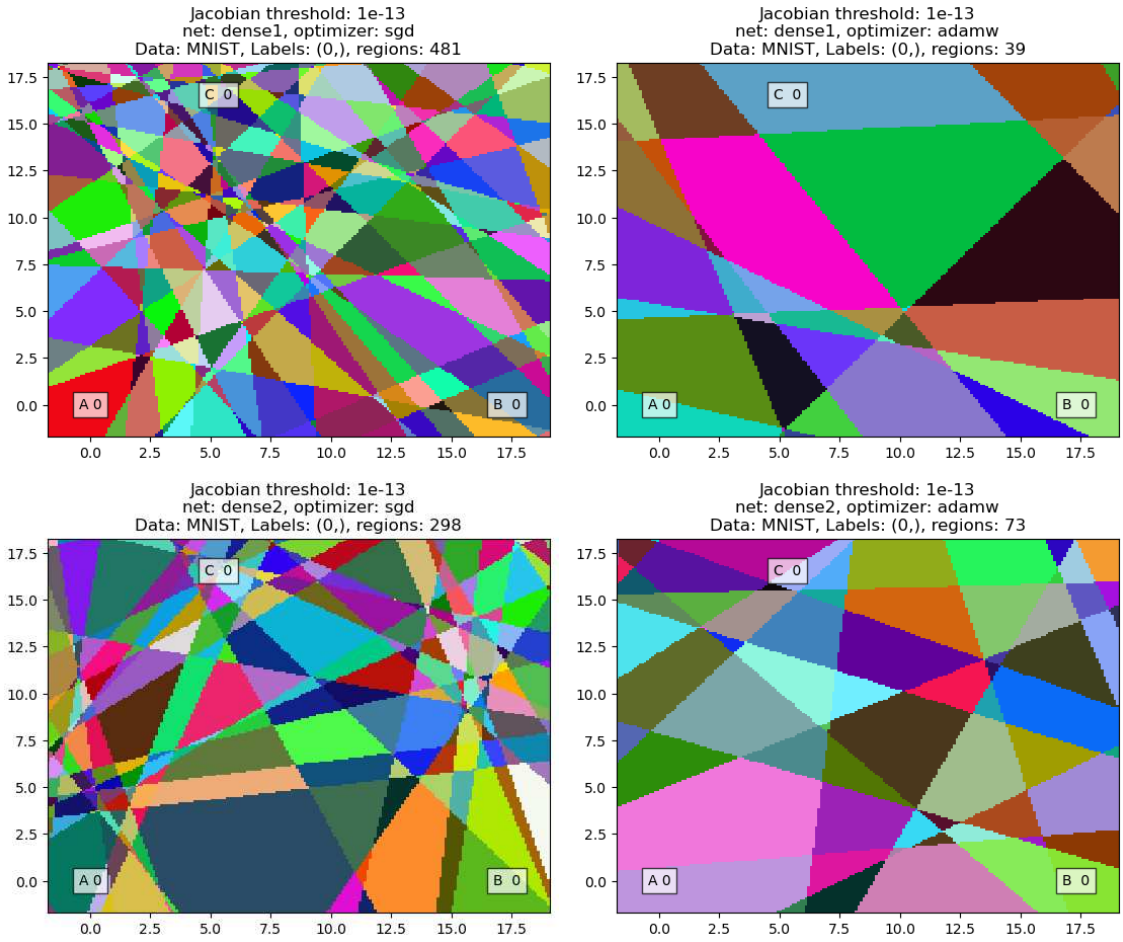


Figure 6.11: The polytope structure for the fully-connected networks on MNIST. The cross-section is defined by the samples in Figure 6.9. Each colored region corresponds to a different linear function for the network. Colors are assigned randomly to assist in showing the variation.

cross section (which may not pass through the zero in image space) is centered. A shift of "one" along either axis also does not necessarily correspond meaningfully to a concept like shifting a single pixel by one in image space — it is movement by a unit length vector in the cross section.

Given this cross-sectional plane, we can then sample points on it and calculate the polytope shapes of the linear regions of various networks. We sample points on the $p \times p$ uniform grid for some positive integer p, and calculate the coefficients of the linear region associated with each point. Adjacent points whose distance between linear coefficients is below a set tolerance are considered to be the same. This tolerance is necessary due to machine error in the calculation of the gradients — two points in the same polytope will have slightly different linear coefficients when calculated computationally with a fixed precision.

To have $\mathcal{O}(p^{3/2})$ runtime complexity instead of $\mathcal{O}(p^2)$ runtime, we consider points in order from left to right, then top to bottom. Each point is only compared to the point directly to its left and to the points in the row above it running from the point directly above the point to its left all the way to its right, as shown in Figure 6.12. This can result in potential problems if a region is thin enough to pass through the gap between points, but this method allows for a much tighter grid due to the significantly decreased computational time which will decrease the likelihood of this occurring. Increasing the number of points considered by a fixed number of rows or columns could also improve this without significantly increasing the time taken.

Additionally, complex networks networks have many small linear regions. Using the "full" cross section that includes all three images used to construct it may result in more linear regions than can be displayed. In those cases, a smaller section of the plane is considered instead.

Results for this process based on the three CIFAR-10 images shown in Figure 6.8 for the two dense networks with each optimizer. Similarly, results for this process based on the three MNIST images shown in Figure 6.9 for the two dense networks with each optimizer. In all cases, the number of regions is large. This is a small portion of a cross-section, which is small enough to only include the three closest images. Despite that, there are hundreds or thousands of regions in this small space. The SGD networks have more regions, which suggests a smoother function.

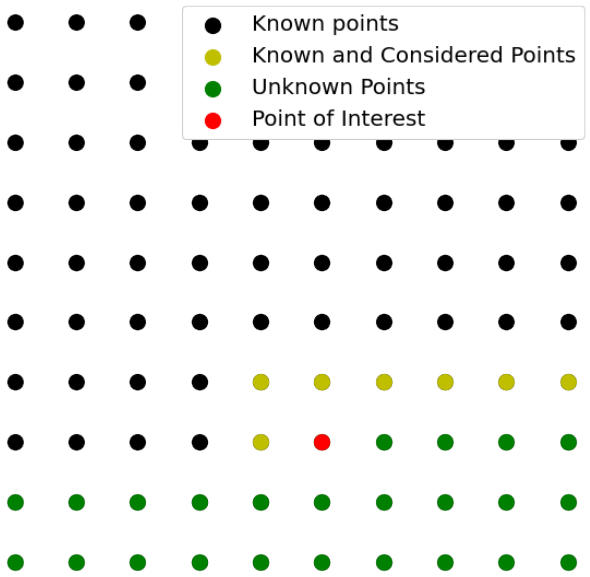


Figure 6.12: An example of which points are considered when calculating which region a point considered. Points in black and yellow have had their regions calculated already. Points in red and green have not. For the point in red, the regions the points in yellow belong to are considered as potential matches. Because the algorithm moves from left to right then top to bottom and regions are convex, if a tight enough grid is used, any region that a point in black belongs to that the point in red belongs to must necessarily also have a point in yellow that belongs to it.

For convolutional networks, the number of linear regions expands even further, so we are unable to view the entirety of even this small portion of the cross-section. We instead look only at a small region around the point corresponding to image A. We additionally compare to the method where activation patterns are compared, as shown in Figure 6.13 for CIFAR-10. As can be seen in these images, the number of regions is still large, but there is the fact that the activation pattern identifies more regions. This comes from the fact that this Jacobian method is inaccurate for a few reasons:

1. Networks are trained with 16-bit floating point values, as the necessary precision for many inference problems is low and higher precision in training does not significantly improve results. Numerical values are transformed to 64-bit floating point values for calculating and comparing the Jacobians, but the original values are still coming from 16-bit floats.

2. Convolutional networks exploit the structured nature of images to utilize the same weights, just shifted spatially across the image. This means that many weight values will be identical, which will cause difficulties in identifying when Jacobians are dissimilar.

3. Multiple trajectories through the network can result in nearly-equivalent values. If this occurs in nearby pixels for max-pooling, such as in the similar sky values in the CIFAR-10 images, which pixel corresponds to the maximum can shift without meaningfully significantly the Jacobian.

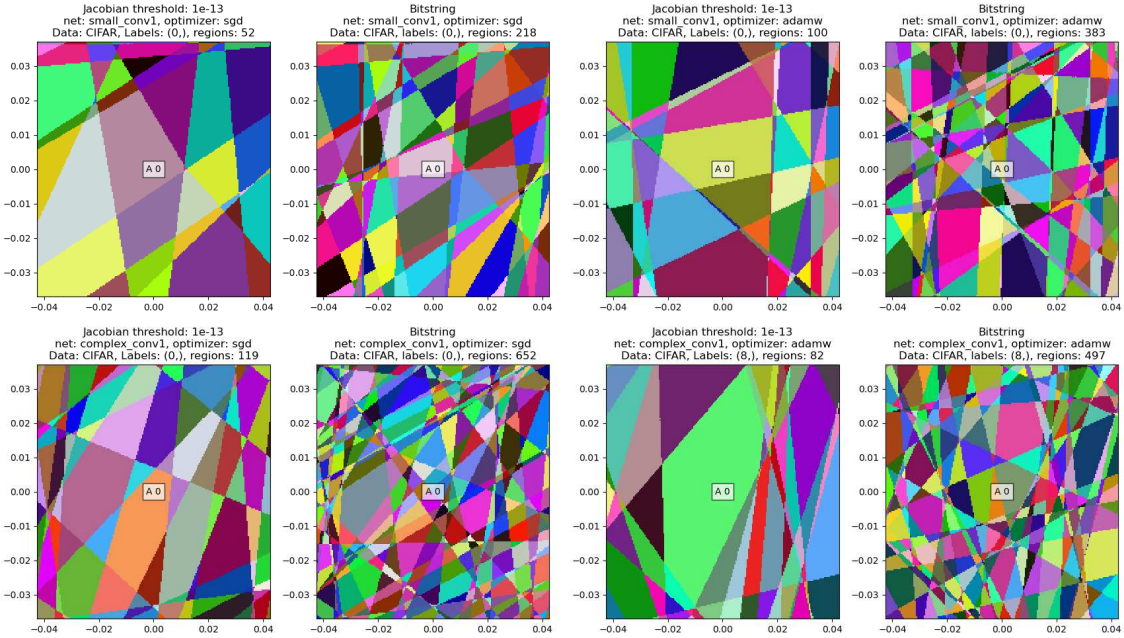


Figure 6.13: The polytope structure for the simple and complex convolutional networks on CIFAR-10. The cross-section is defined by the samples in Figure 6.8. Each colored region corresponds to a different linear function for the network. Colors are assigned randomly to assist in showing the variation. Left polytope structures come from comparing Jacobian values and right polytope structures come from comparing activation patterns.

This is not necessarily a downside, as it means that this method shows areas where the linear functions the network uses are sufficiently different, but it does mean that this methodology is inaccurate for our goals here. When trying to visualize the exact polytope structure of a network, this Jacobian method introduces error and will not include boundaries corresponding to sufficiently

small changes in the Jacobian. It is also more expensive computationally than using activation patterns. Determining boundaries exactly analytically or by using the activation patterns will have significantly improved performance.

6.4 Sensitivity Analysis

Related to the investigation of polytopes, we can use linearizations to determine how sensitive networks are to changes in the input space. One way of doing this utilizes a form of dimensionality reduction called active subspaces, related to PCA [116]. It operates by determining the covariance matrix of the gradients of the output of a function with respect to its inputs. The singular value decomposition of the covariance matrix then yields the linear directions along which the function varies most strongly (the eigenvectors) and how strongly the function varies along them (the singular values).

The gradient covariance matrix can be sampled using gradients at points in the input space. We look specifically at results for image classification networks on their test sets. Results for MNIST are in Figure 6.14, which shows the linear direction of most variance as an image; Figure 6.15, which shows how well that direction explains the outputs of the function; and Figure 6.16 which shows the singular values of the linear directions. In general, SGD directions are "blurry" compared to Adamw — the direction has smoother changes from pixel to pixel. SGD networks have their first directions as significantly more explanatory for the network performance than the Adamw networks, suggesting that the Adamw networks have more nonlinearity in their outputs than the SGD networks

The dense networks have similar shapes to the linear network, which matches the fact that the dense networks have a small number of nonlinearities compared to more complex networks, and so will be similar on average to the linear networks. Shapes corresponding to the output value can be seen for each of the linear and dense networks, but convolutional networks have less obvious shape. The networks trained on ImageNet have values across the entirety of the input space, despite no images in MNIST using the corners. The dataset they are trained on does utilize the entirety

of the image, so this isn't surprising, and is also related to the fact that these linear direction has almost no explanatory power for the network outputs.

From the singular values in Figure 6.16, the dense networks have one important direction, then a steep drop-off for the remaining 127 directions. The fact that they only have 128 nonzero singular values corresponds to the fact that they only have 128 ReLU nonlinearities. The simple convolutional networks have a steeper drop-off, with later singular values being much closer to zero than the complex networks. The ImageNet networks have very little drop-off in singular values, so their behavior is difficult to explain without using the entirety of the information.

Similarly, results for CIFAR are in Figures 6.17, 6.18, and 6.19. Except for the linear network, the first directions are less explanatory than they are for MNIST. MNIST is a nearly linear dataset, so it isn't surprising that a single linear direction explains network performance well. CIFAR-10 is not as linear, and so the linear direction of most change in the output will not be particularly meaningful. Based on this, this method is unlikely to be helpful in interpreting network behavior except on sufficiently linear problems.

This method works well for MNIST, largely because MNIST is a dataset that is close to linearly separable. Most networks appear to be most sensitive to changes along directions similar to those described by a linear classifier, with the main exception of the networks trained on other problems having behavior that is not human interpretable. Networks trained using SGD have "smoother" images, where there is less variation between nearby regions. Networks trained using Adamw are more jagged, with nearby regions being dissimilar.

The results on CIFAR are significantly worse. CIFAR is further from a linearly separable dataset, and so networks are learning functions of the output that are not well-described by linear directions in the input space.

6.5 Conclusions

For the circle versus annulus networks, the larger network's first layer forms a structure similar to the smaller networks. Further layers do not significantly improve the decision boundary,

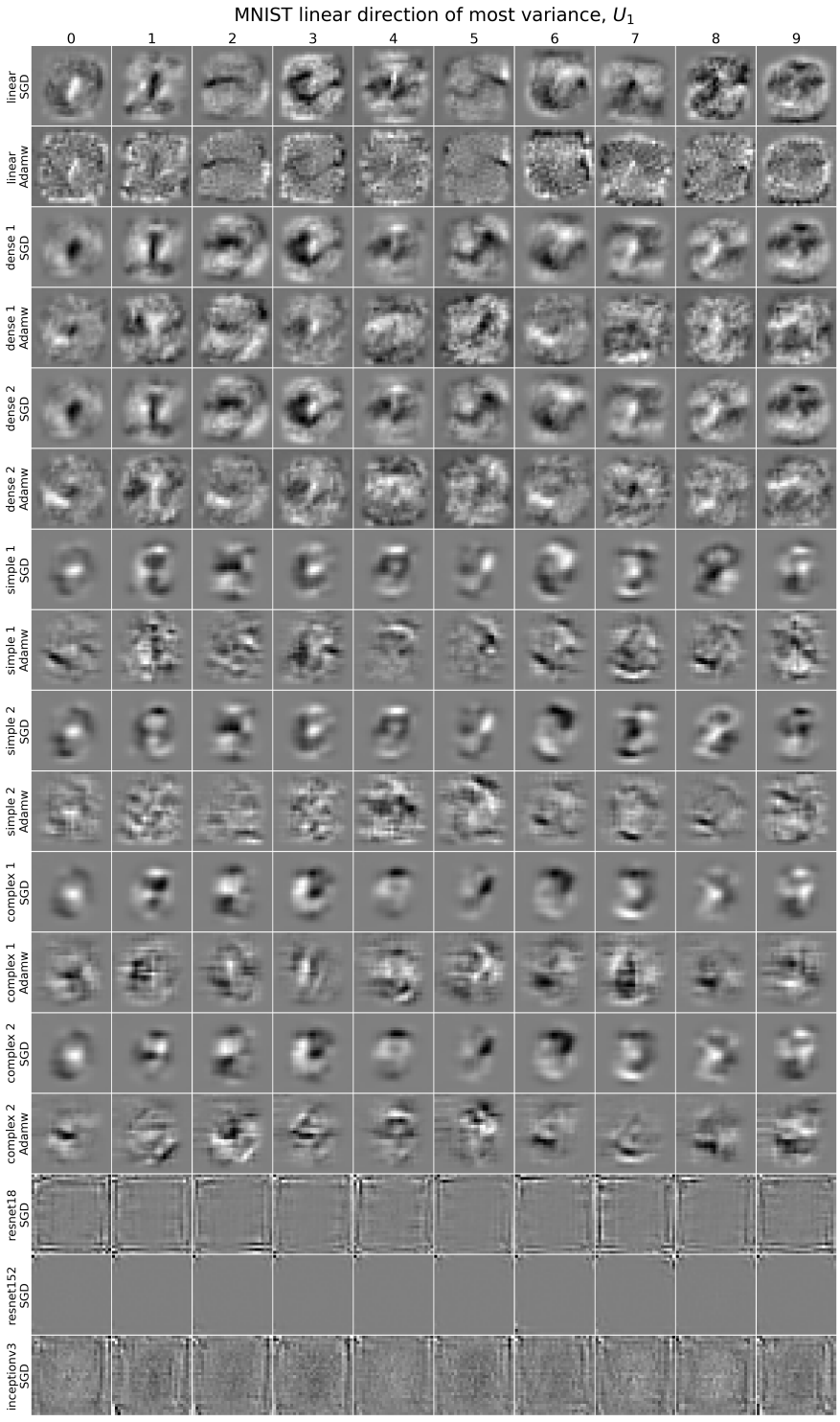


Figure 6.14: Visualizations of the linear directions along which each network's output varies most. Each column corresponds to a given output corresponding to each of the classes. Each row corresponds to a different combination of network and optimizer. The ResNet and Inception networks only have results shown with SGD, as optimizer only affects the final linear layer for those networks.

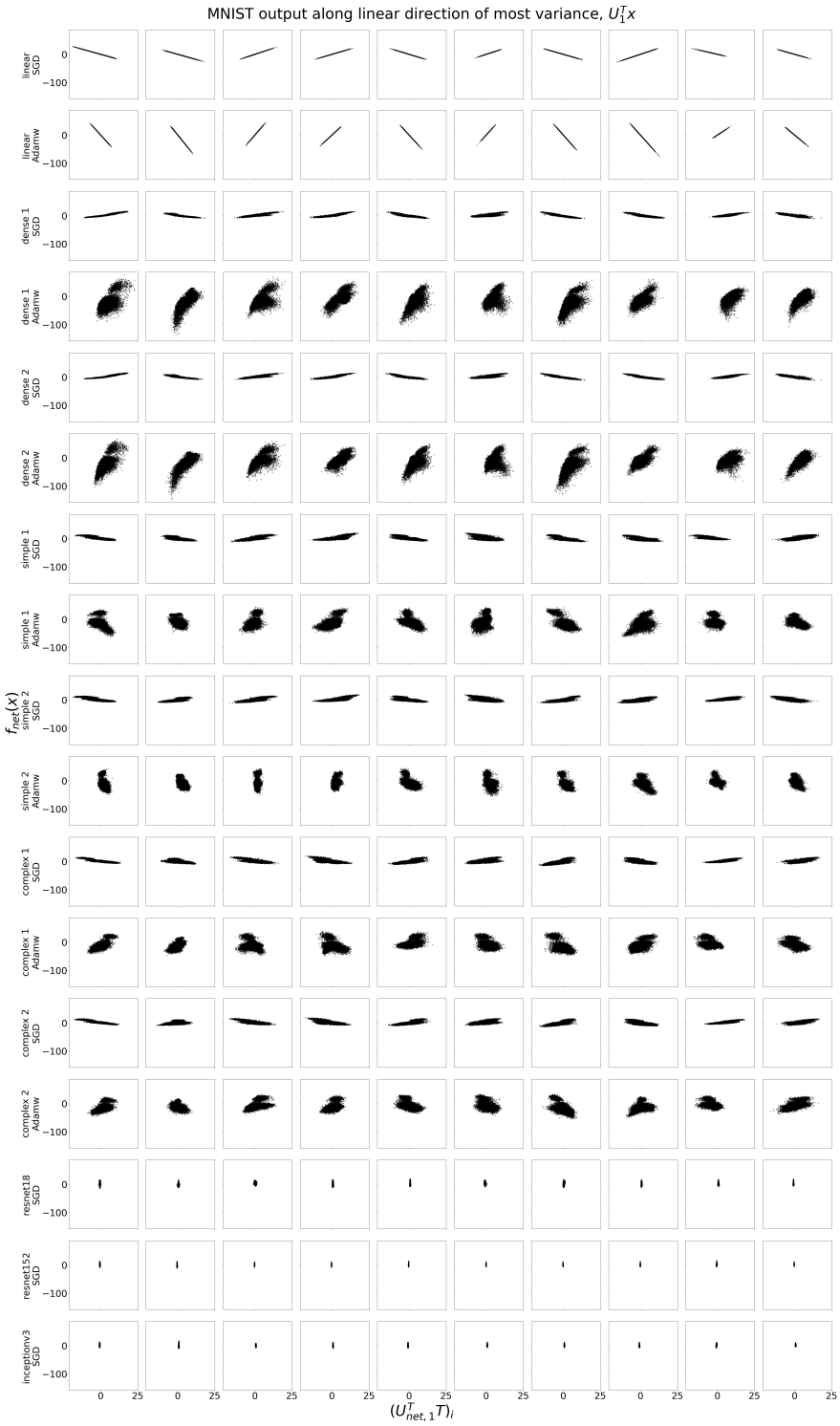


Figure 6.15: Visualizations of the function output along the linear directions along which each network's output varies most. Each column corresponds to a given class. Each row corresponds to a different combination of network and optimizer. The ResNet and Inception networks only have results shown with SGD, as optimizer only affects the final linear layer for those networks.

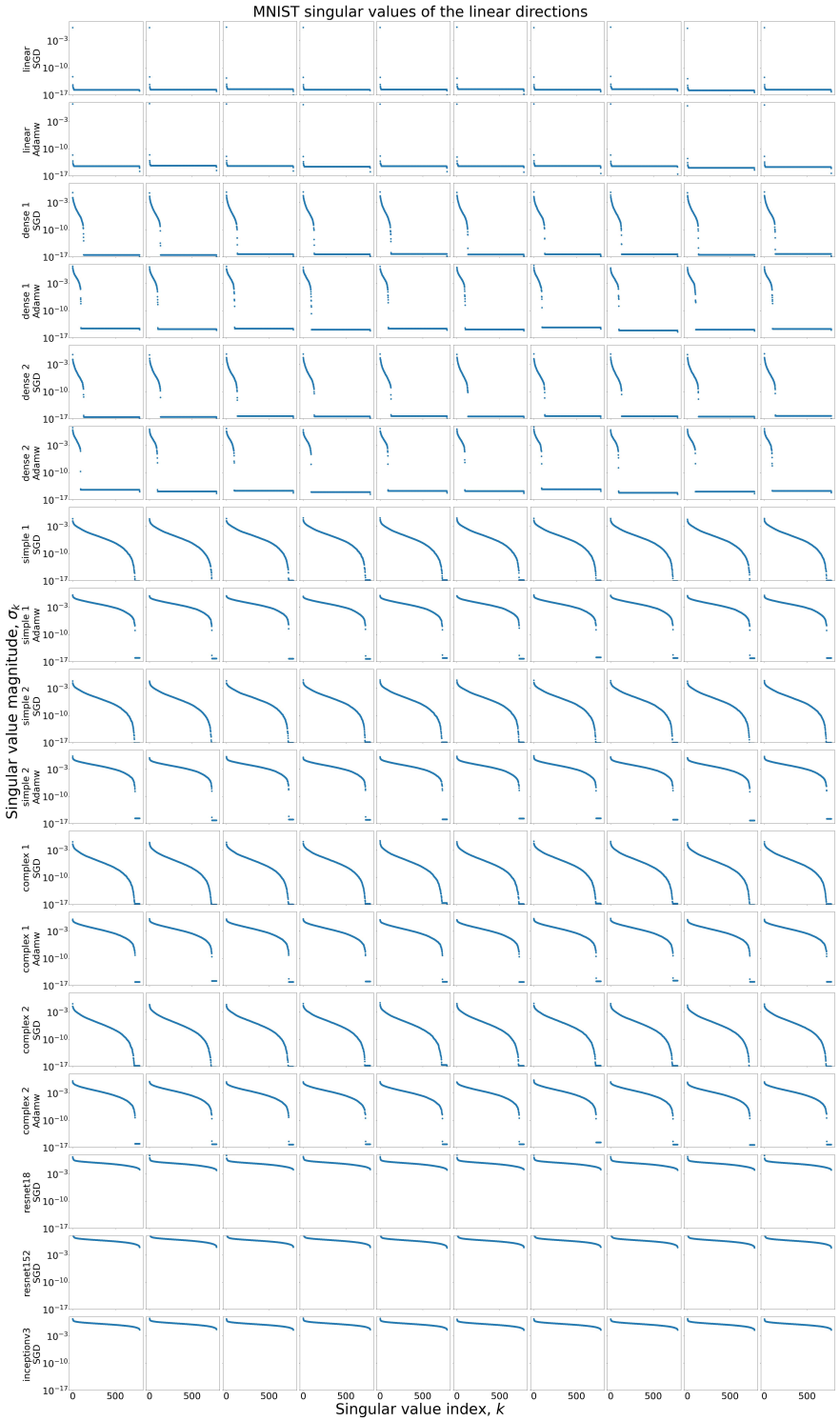


Figure 6.16: The singular values corresponding to the linear directions along which the function's output varies. Each column corresponds to a given output corresponding to each of the classes. Each row corresponds to a different combination of network and optimizer. The ResNet and Inception networks only have results shown with SGD, as optimizer only affects the final linear layer for those networks.

Figure 6.17: Visualizations of the linear directions along which each network's output varies most. Each column corresponds to a given output corresponding to each of the classes. Each row corresponds to a different network. For the red, green, and blue channels, red, green, and blue values are positive and black, purple, and red values are negative

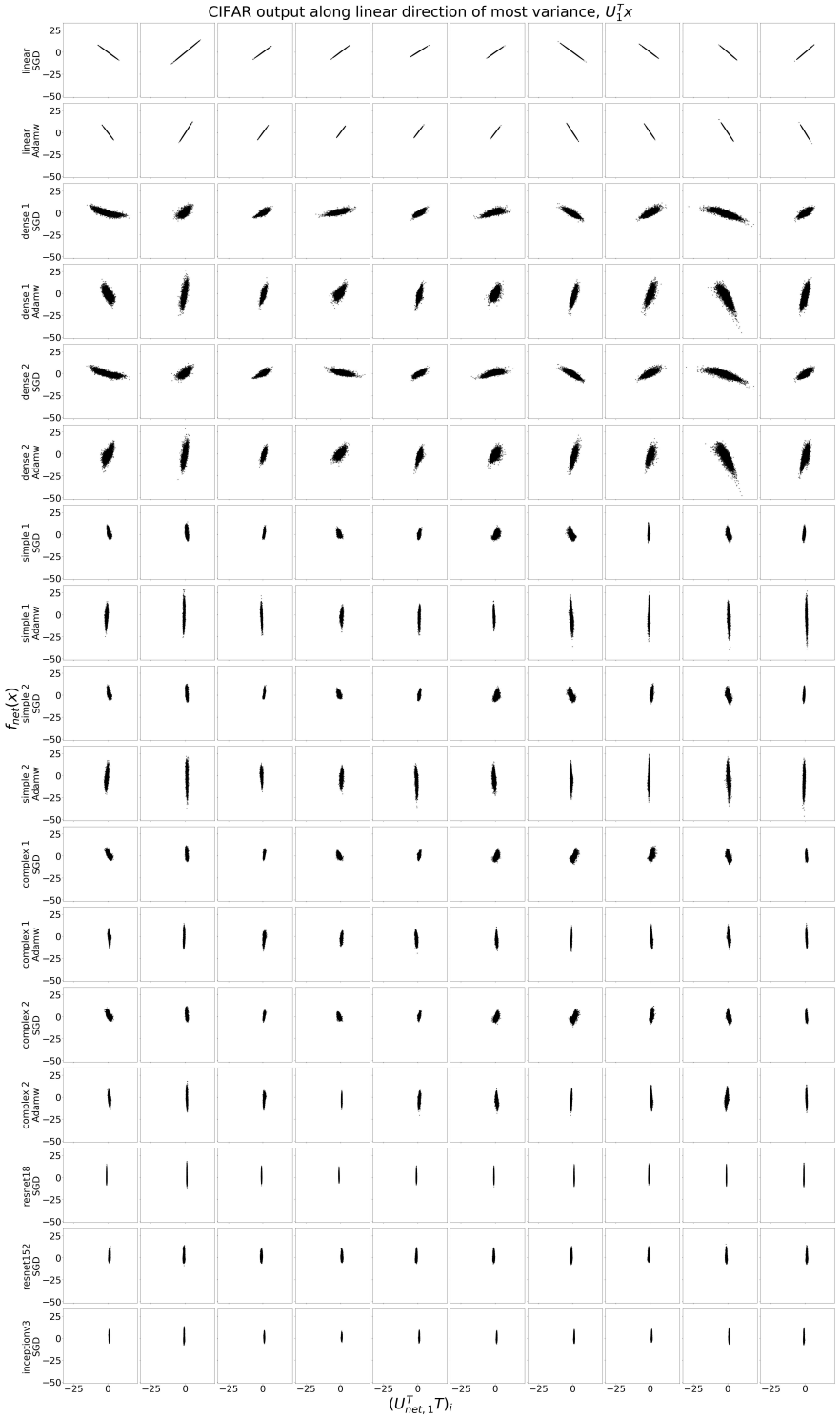


Figure 6.18: The singular values corresponding to the linear directions along which the function's output varies. Each column corresponds to a given output corresponding to each of the classes. Each row corresponds to a different combination of network and optimizer. The ResNet and Inception networks only have results shown with SGD, as optimizer only affects the final linear layer for those networks.

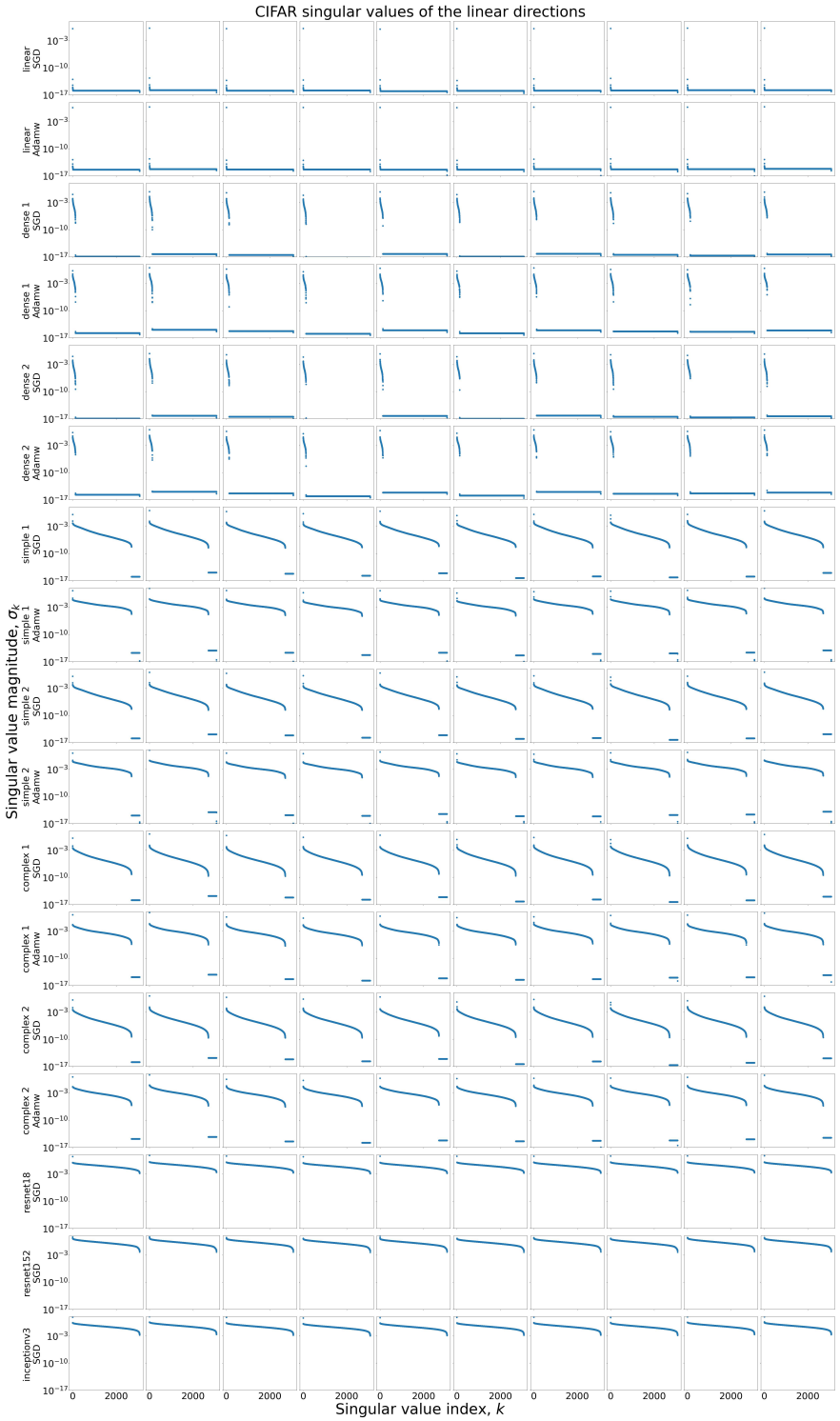


Figure 6.19: Visualizations of the linear directions along which each network's output varies most. Each column corresponds to a given output corresponding to each of the classes. Each row corresponds to a different combination of network and optimizer. The ResNet and Inception networks only have results shown with SGD, as optimizer only affects the final linear layer for those networks.

although they do improve the ability of the network to maintain class labels rather than varying positive and negative values. This redundancy suggests the possibility of compressing the larger network — removing the later layers would change classification performance meaningfully.

The next thing of note is the behavior of the image-classification networks. For those networks, SGD results in smoother weights and performance between networks with the same architecture is more consistent than for networks trained with Adamw. Dense networks have nonzero Jacobian across the entirety of the input, unless regions of the input are constant over the dataset, but convolutions focus on relevant information.

These regions are useful as a way of visualizing behavior, but more structured metrics are still necessary for determining behavior. The saliency analysis can provide insight into how the network performs classification, but modern methods are more sophisticated than simply visualizing the Jacobian [22, 73]. Sampling the Jacobians used for classification and comparing them between networks, as will be discussed in the next chapter, instead provides a way to determine when networks are similar.

Chapter 7

Network Comparisons

In addition to looking at the behavior of individual networks to get a sense for how they behave, it is useful to compare across networks to get a sense for what effect different hyperparameter configurations have. The standard of comparing accuracy across networks to determine which network is "better" has its limitations, and other research into similarity typically focuses on the outputs of various layers, which can be difficult to compare across different styles of networks or holistically in general [65–71]. By using linear coefficients as a similarity metric, we avoid issues with identifying how to relate interior elements of networks by treating the network as a black box with only gradient information being relevant.

In this chapter, we discuss various metrics, both for single networks and for pairs of networks, that may be useful for determining how behavior changes across network structures. We first look at how the number of linear regions can be (naively) reduced used clustering, and what effect that has on network accuracy, as a measure of network complexity. We then look at comparing the sets of linearizations across different networks, both by comparing with linear transformations and by using distance metrics and correlational subspace metrics to determine how similar the networks' behaviors are across the set of inputs. The results here are an extension of previous work by the author [119].

7.1 Networks, Datasets, and Notation Considered

We consider the MNIST and CIFAR datasets as discussed in Section 6.1. We use the XOR problem and circle versus annulus problem, as illustrated in Figure 7.1 as toy problems that demonstrate compressible networks and similar networks.

In addition, we use the same networks as those discussed in Section 6.1. The summary of their training process is repeated here for convenience. We consider seven network architectures trained MNIST and CIFAR problems, with variations trained using both PyTorch's SGD with learning rate

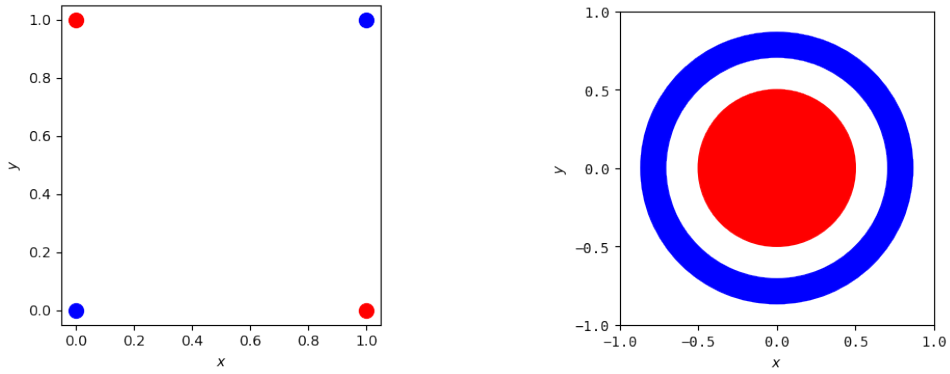


Figure 7.1: Two two-class classification problems used to illustrate network metrics in a visualizable way. Left: XOR. Right: circle versus annulus. Red and blue points have different classes.

0.01, momentum 0.5, and weight decay 0.01 and Adamw with default parameters [122]. Batch sizes of 32 are used, and 15 epochs are used for training. The networks are:

- linear — a linear classifier
- dense — a fully-connected feedforward network with a single hidden layer consisting of 128 ReLU nodes. Two identical variants of this network architecture are trained.
- simple — a convolutional network with two convolutional layers with 32 3x3 filters in each layer. Each convolutional layer is followed by a max pooling layer. ReLUs occur after max pooling. A single fully-connected layer occurs after the second max pool. Two identical variants of this network architecture are trained.
- complex — a convolutional network with three convolutional layers with 32 3x3 filters in each layer. Each convolutional layer is followed by a max pooling layer. ReLUs occur before max pooling. A linear combination of the outputs of the last max pool are used for classification. Two identical variants of this network architecture are trained.
- A network with the ResNet-18 architecture as implemented in Torchvision's models package using the pretrained parameters available in that package [11, 123].Images are upsampled, using bilinear interpolation, to the expected size of 224×224 pixels. The final linear classification layer is the only layer that is trained.

- A network with the ResNet-152 architecture as implemented in Torchvision's models package using the pretrained parameters available in that package [11, 123].Images are upsampled, using bilinear interpolation, to the expected size of 224×224 pixels. The final linear classification layer is the only layer that is trained.

- A network with the Inception-v3 architecture as implemented in Torchvision's models package using the pretrained parameters available in that package [123, 124]. Images are upsampled, using bilinear interpolation, to the expected size of 229×229 pixels. The final linear classification layer is the only layer that is trained.

All non-pretrained networks are structured to accept three channel 32×32 images. For MNIST, images are scaled to 32×32 using bilinear interpolation, then repeated three times to match the number of channels expected. Accuracies are reported in Table 7.1. The linear network and pretrained networks are included partially as controls — their behavior should be dissimilar from networks that can represent the problems and are trained on the problems.

Table 7.1: Accuracies of the networks trained using SGD and Adamw on MNIST and CIFAR

	SGD MNIST	Adamw MNIST	SGD CIFAR	Adamw CIFAR
linear	88.89%	86.83%	25.43%	31.66%
dense 1	96.76%	97.43%	51.01%	47.55%
dense 2	96.91%	97.24%	50.24%	48.41%
simple 1	97.80%	98.86%	68.14%	67.03%
simple 2	97.89%	98.90%	66.34%	67.58%
complex 1	97.73%	98.95%	61.72%	70.55%
complex 2	97.86%	98.81%	62.50%	71.33%
resnet18	96.64%	96.43%	80.52%	80.15%
resnet152	89.67%	91.12%	83.94%	82.57%
inceptionv3	91.54%	91.34%	77.05%	77.05%

For investigating the usage of linear coefficients for building metrics on neural networks, there are two useful steps to take: constructing notation to allow us to refer to the set of affine mappings potentially used for a specific output of a network, and considering only the affine mappings that are used for training or testing to reduce the number to something computationally manageable.

In terms of notation, the $\mathbf{J}_i, i = 1, ..., r$ and associated y_i, which is the Jacobian of the network within the region defined by Q_i and p_i, and described in Section 5.5 can be written as

$$\mathbf{J}_i = \begin{bmatrix} \nabla_x^T v_1 \\ \vdots \\ \nabla_x^T v_d \end{bmatrix} . \text{ and } y_i = \begin{bmatrix} y_{i,1} \\ y_{i,2} \\ \vdots \\ y_{i,o} \end{bmatrix}, \tag{7.1}$$

Where each $(\nabla_x^T v)_{i,j}$ and $y_{i,j}$ correspond to the affine mapping in polytope $1 \leq i \leq r$ for the $1 \leq j^{\text{th}} \leq o$ output of the network. Then, it is possible to construct the matrix containing the set of affine mappings used for a given output, j, as

$$\overline{\mathbf{C}}_j = \begin{bmatrix} (\nabla_x^T v)_{1,j} & y_{1,j} \\ (\nabla_x^T v)_{2,j} & y_{2,j} \\ \vdots & \\ (\nabla_x^T v)_{r,j} & y_{r,j} \end{bmatrix} \in \mathbb{R}^{r \times (d+1)}. \tag{7.2}$$

In practice, it is computationally infeasible to calculate all r linear regions, so for the purpose of empirical studies we choose p points in the input space to sample and construct the matrix

$$\mathbf{C}_j = \begin{bmatrix} (\nabla_x^T v)_{1,j}^T & y_{1,j} \\ (\nabla_x^T v)_{2,j}^T & y_{2,j} \\ \vdots & \\ (\nabla_x^T v)_{p,j}^T & y_{p,j} \end{bmatrix} \in \mathbb{R}^{p \times (d+1)}. \tag{7.3}$$

For simple, two-dimensional input problems, we choose the p points by sampling from a uniform grid. For the MNIST and CIFAR datasets, the p points we sample from are the training or testing input samples from that network. We construct the $\mathbf{C}_j$ matrices using the training samples, and we additionally construct $\tilde{\mathbf{C}}_j$ using the testing samples for evaluation of various metrics.

7.2 Compression

Linear regions have previously been used to investigate network complexity, with results focusing on the number of linear regions attainable by the network or the number of regions within a subset of the domain of interest [14, 16, 17, 19, 20]. In contrast to those results, we can instead look at well the network maintains classification if the number of linear regions is reduced. An example of this on the circle versus annulus problem is shown Figure 7.2. The linear coefficients, as shown in the third graph in the top row, cluster into ten regions despite having 93 total regions in the domain. It's not necessarily clear that clustering is the best method for determining the behavior of the weight space, but it is a simple experiment that we can run to investigate possible behavior.

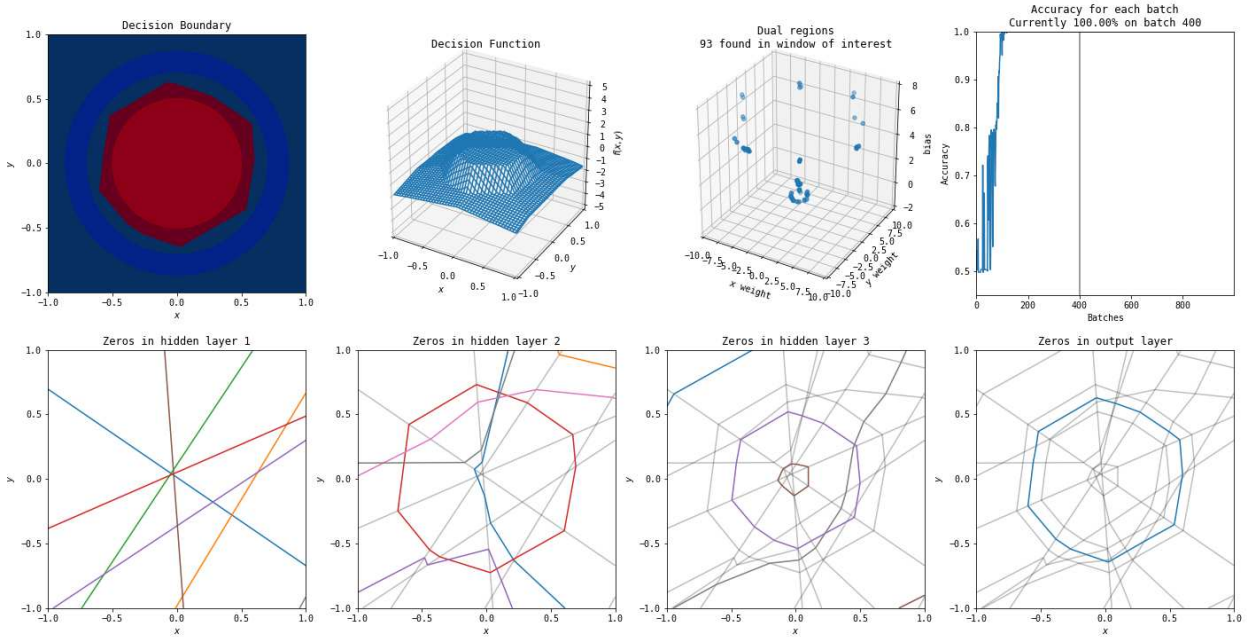


Figure 7.2: A solution to the circle versus annulus problem found by a complex network (three hidden layers, each with eight nodes).

For this simple problem, it is the case that many points in the uniform grid of the input space share a linear region. However, for image-classification networks, even for potentially large numbers of sampled affine maps it is likely that many samples will have a unique Jacobian matrix, $\mathbf{J}_i$, due to the large number of total linear regions. For example, even simple networks on the MNIST

dataset only have overlap on $< 1\%$ of the training inputs. This isn't necessarily surprising, simply due to the sheer number of possible linear regions the network can construct.

We can cluster these linear weights and determine how well those clusters are able to replicate the behavior of the network as a measure of that redundancy.

1. Calculate the $\mathbf{C}_j$ and $\tilde{\mathbf{C}}_j$ matrices as described in Section 7.1.
2. Train k-means clustering models using the each of the $\mathbf{C}_j$ matrices.
3. For each row of each of the $\tilde{\mathbf{C}}_j$, determine for which cluster center it is closest.
4. Use that cluster center as an affine mapping from input space to $\mathbb{R}$ to determine the value for that output.
5. Classify the input based on which of the newly calculated outputs is highest

By varying k and comparing the resulting accuracy against the original accuracy of the network, we can investigate the degree to which networks can be simplified. If applying this method with $k = 1$ results in near-original accuracy, that suggests the network is behaving holistically as a linear mapping, whereas if it results in near-random accuracy, that suggests the network's behavior can not be well described by a linear transformation from the input space to the output space. Determining at which value of k accuracy approaches the original provides a way to understand how significantly the network can be simplified.

Accuracies shown when doing this for various numbers of clusters are shown in Figures 7.3 and 7.4 for MNIST and CIFAR, respectively. Unsurprisingly, networks perform poorly when using one cluster — this corresponds to a simple linear classifier. The fact that the dense networks do not fail as catastrophically for $k = 1$ corresponds to existing work that dense, wide networks learn along trajectories corresponding to linear classifiers [125]. When increasing k, networks increase their new accuracies. Values close to the original performance are only attained for the dense networks and the non-pretrained networks on MNIST with SGD. It is not surprising that the ResNet and Inceptionv3 networks would have poor performance, as the problem they are trained

on is significantly more complex than CIFAR and so will likely need additional representational capacity.

The significant difference between performance for networks trained using SGD and Adamw is surprising. A possible explanation for this is that Adamw has less weight decay than SGD — 0.1 versus 0.01. Weight decay will tend to decrease the complexity of the regions, by reducing the magnitude of irrelevant trajectories through the network and minimizing the effect of spurious ReLU activations.

The high performance on MNIST suggests that many of the linear regions in a network are not significantly important — a network with many fewer regions would work with similar accuracy. That doesn't mean that this leads to a natural algorithm for compression — using this clustering algorithm still requires every sample to be run forward through the network, then backward to calculate the gradient with respect to the input, then compared with the cluster centers. However, it does suggest that there may be a methodology that reduces the number of extraneous nonlinearities in a network, potentially after training, to result in a highly compressed version.

Additionally, this means that the complexity of many modern neural networks may not be necessary for the results they achieve in terms of representation. As discussed by LeCun *et al.* [29], increasing the complexity of a network can benefit the ability of the network to achieve good local optima, so this supports claims that larger networks train well but can be reduced in complexity after training occurs [21, 30].

7.3 Network Similarity

Determining when two networks are similar is a topic of much interest. Despite this, there are complications to determining such. Similarity as a concept is difficult to define. What property of the networks are similar? Do specific layers align? Do the outputs align? Are the transformations the networks apply to the inputs aligned? Under what metric is it reasonable to determine alignment? On what set is similarity determined? Two networks may have the same performance on

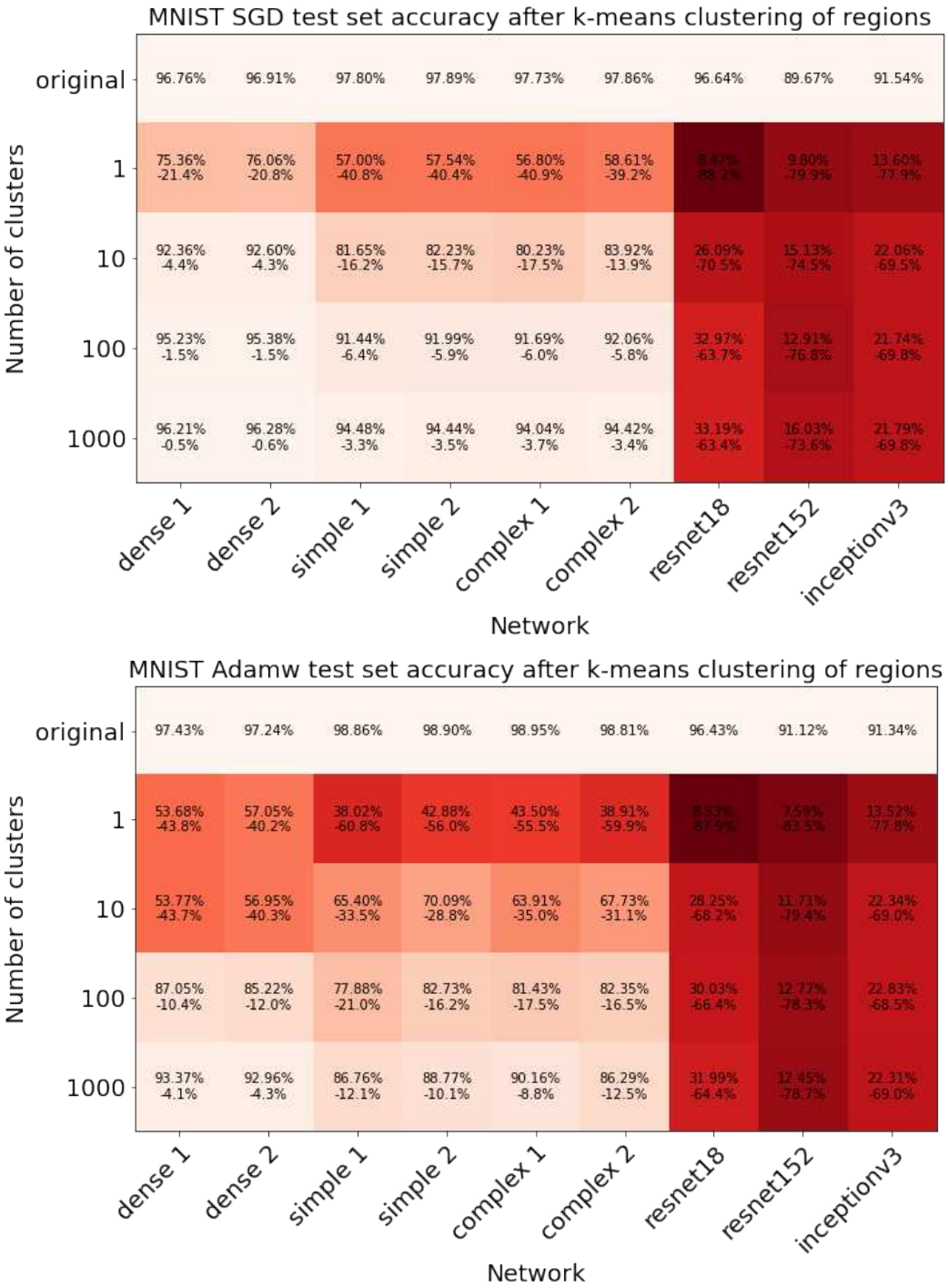


Figure 7.3: The accuracies of networks on the MNIST dataset after applying K-means clustering to their collection of local linear maps. The top row reports the networks' original accuracies before clustering. Values reported are the percentage of correctly labeled test set samples out of 10,000. Negative values under reported percentages is the percentage point drop in accuracy from the original network. Note that the number of clusters for a given network is technically 10 times larger than stated in the table — for a given output there will be that many clusters, but there are 10 outputs for the MNIST networks. Top: networks trained using SGD. Bottom: networks trained using Adamw.

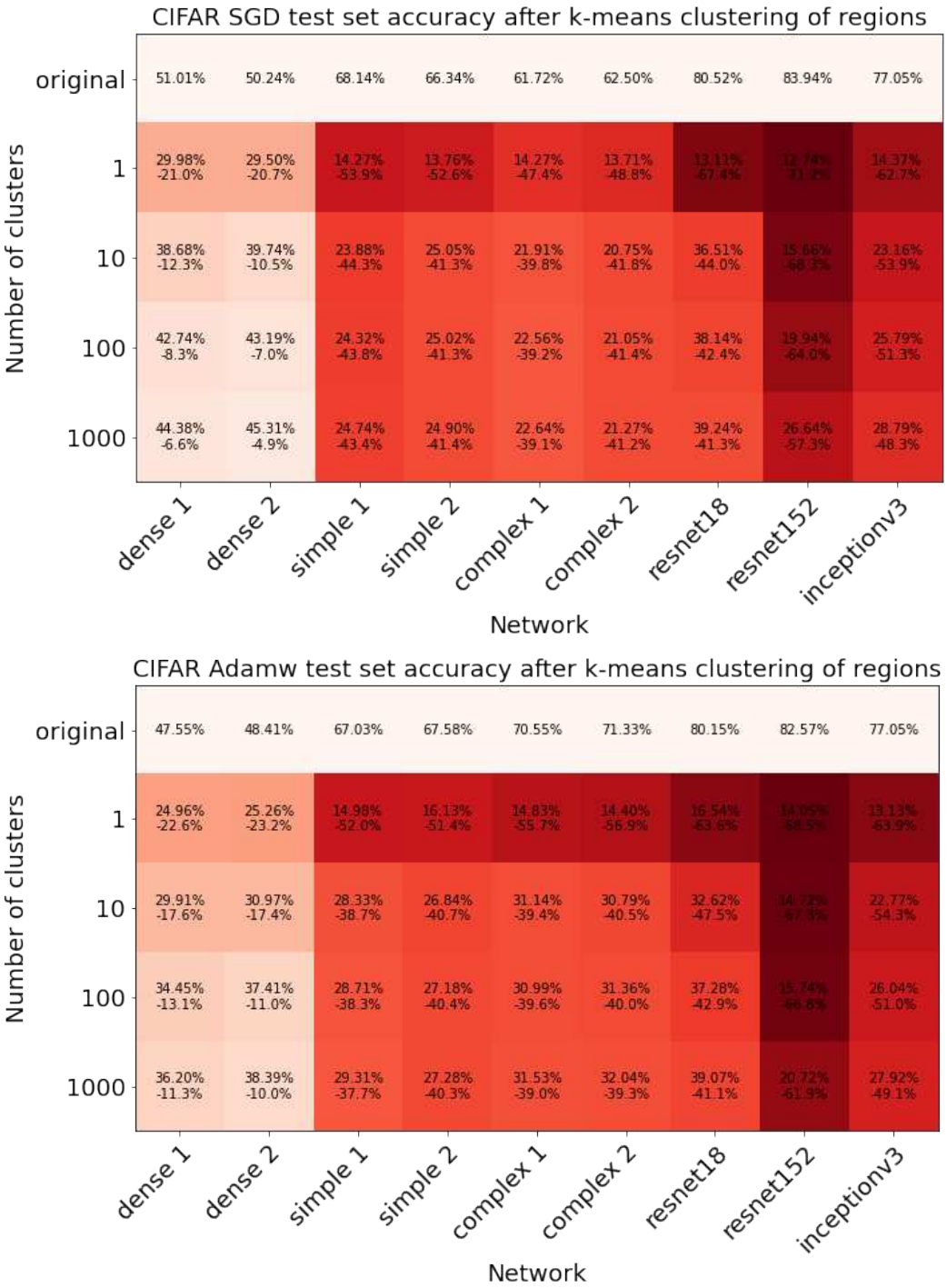


Figure 7.4: The accuracies of networks on the CIFAR dataset after applying K-means clustering to their collection of local linear maps. The top row reports the networks' original accuracies before clustering. Values reported are the percentage of correctly labeled test set samples out of 10,000. Negative values under reported percentages is the percentage point drop in accuracy from the original network. Note that the number of clusters for a given network is technically 10 times larger than stated in the table — for a given output there will be that many clusters, but there are 10 outputs for the CIFAR networks. Top: networks trained using SGD. Bottom: networks trained using Adamw.

the training set, or the testing set, or the manifold that underlies the data, but different performance outside of those regions.

In practice, the standard for comparing networks is to simply compare their performance on a test set. The network that has a higher proportion of samples classified correctly is "better." This has a number of limitations, as it doesn't provide any information about the ways in which the networks construct their outputs, so two networks which have similar performance may utilize distinct properties of the data to solve the problem. This isn't a condemnation of this practice, as it largely satisfies the role of determining which network is more useful on practical problems. However, we are more interested in the question of when two networks are doing similar or dissimilar things.

Other research into similarity typically focuses on the outputs of various layers or on comparing weights between layers, which can be difficult to compare across different styles of networks or holistically in general [65–71].

Since neural networks are piecewise linear functions, we can determine the linear mapping used by the network for classification on a given input sample. Comparing those linear mappings provides a method for determining similarity of the functions the networks are using without relying on the internals of the network. Specifically, we look at how well the linear mappings of one network can represent those of another after an affine transform in Subsection 7.3.1, the mean squared distance between the linear mapping's weights between two networks in Subsection 7.3.2, and the centered kernel alignment in Subsection 7.3.3. We also look at the cross-entropy of the outputs of the networks after a softmax in Subsection 7.3.4 to have a control that aligns with existing methods.

7.3.1 Affine Mapped Representation Performance

It's common when applying neural networks to a problem to use vastly more weights than are strictly necessary to solve the problem. This has a host of practical benefits (not the least of which is that it is difficult to know what a minimal network structure is for a problem), but it does mean that networks may encode redundant or extraneous information. In Figure 7.5, multiple networks

trained on XOR are shown with their decision boundaries and the linear weights associated with their polytopes in the domain shown. As the complexity of the network improves, the accuracy does not increase (since the first network is optimal), but we get tighter resolution approximations of an underlying function (due to underlying dynamics such as loss and regularization). Determining the similarity of networks' linear regions can be useful for determining how they approximate functions and whether they encode novel information.

One way of addressing this is seeing how well the linear coefficients of one network are able to represent the linear coefficients of another network. Given $\mathbf{C}_{j,network1}$ and $\mathbf{C}_{j,network2}$, we can train least-squares regression models for each output to find matrices $\mathbf{M}_j \in \mathbb{R}^{d+1 \times d+1}$ for each $1 \leq j \leq o$ that minimize

$$||\mathbf{C}_{j,network1}\mathbf{M}_j - \mathbf{C}_{j,network2}||_2. \tag{7.4}$$

This method finds a mapping between the linear coefficients, or, equivalently, between the gradients of the outputs with respect to the input. Due to this, as with the k-means clustering method, this method requires running inputs through each original network, calculating the Jacobians, then applying the transformation.

This is similar to the work done by [74] where the authors demonstrated that the outputs of the final layer before the linear classifier of networks trained on ImageNet are affine-equivalent. Unlike their work, our work investigates the connection between the affine mappings of the locally linear functions of networks, rather than the feature vectors of networks.

Results of this process for XOR networks using the $\mathbf{C}$ matrices constructed by sampling points on the 101×101 uniform grid are shown in Figure 7.6, which shows the results of mapping between two networks trained on XOR. Although the two networks have similar behavior, their decision boundaries are somewhat different and their dual representations are close to rotations of each other. The resulting points of $\mathbf{C}_{j,network1}\mathbf{M}_j$ are very similar to $\mathbf{C}_{j,network2}$ and vice versa, meaning that the mapping is successful. The function resulting from this is no longer continuous — because the bias is part of what is being mapped, the result is able to vary based on the position in the plane and regions may no longer join at their boundaries. By applying this method to more

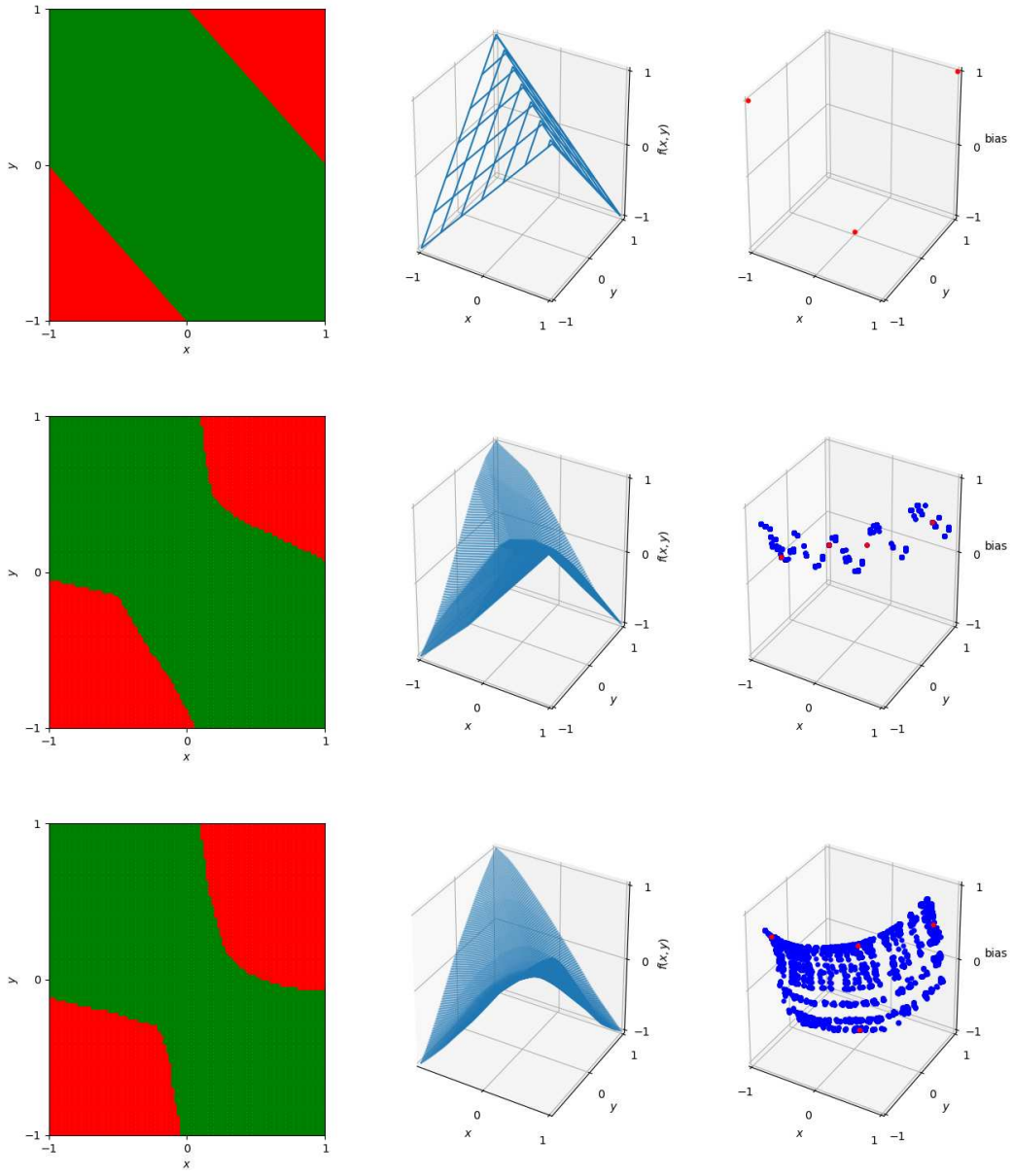


Figure 7.5: The decision boundaries (left), wireframe representations of output (center), and dual representation of the linear coefficients (right) for three networks designed to solve ReLU. The top network is the simple one with a single hidden layer with two nodes described previously. The center and bottom are single hidden layer neural networks with the center having 20 hidden nodes and the bottom having 100 hidden nodes. In the dual, blue dots represent linear coefficients used on the 101×101 uniform grid in $[-1, 1]^2$. The red dots represent the linear coefficients used for the actual classification of the four data points — note that the top image only has three dots corresponding to these, rather than four, as it only has a total of three linear regions.

complex networks where similarity is not as clear, we can determine potential overlap in network behavior.

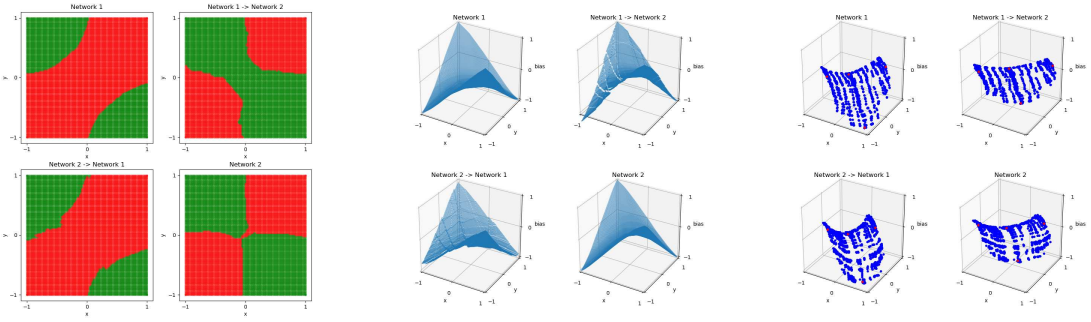


Figure 7.6: The decision boundaries, wireframes, and coefficients in the dual space for two different XOR networks and the result of training an affine mapping from the linear coefficients of one network to the other. In the dual space the four points of XOR are in red and all other points sampled uniformly from the grid in the original input space are in blue.

An example of what this looks like for the linear coefficients of an MNIST network is shown in Figure 7.7. Here, linear coefficients from the Dense network are able to visually match the linear coefficients of a convolutional network well. Results of this process for image processing datasets are shown in Figures 7.8, 7.9 for MNIST and CIFAR with Adam.

For MNIST, mapping from a network to a target network preserves a large amount of the target network's accuracy, except when mapping to the dense networks trained with Adamw or mapping to the networks that are pretrained on ImageNet. Additionally, ResNet18 is able to represent those networks despite those networks not being able to represent ResNet18. When trained with Adamw, the dense networks are difficult for other networks to represent despite the dense networks trained with SGD being easy to represent.

For CIFAR, the dense networks are similar to each other. When mapping from the linear network trained with SGD to the dense networks trained with SGD, the performance increases, which suggests the noncontinuities induced by the mapping of the biases increases performance. SGD networks are more able to represent each other and Adamw networks than Adamw networks are each other or SGD. The simple and complex convolutional networks trained with SGD are

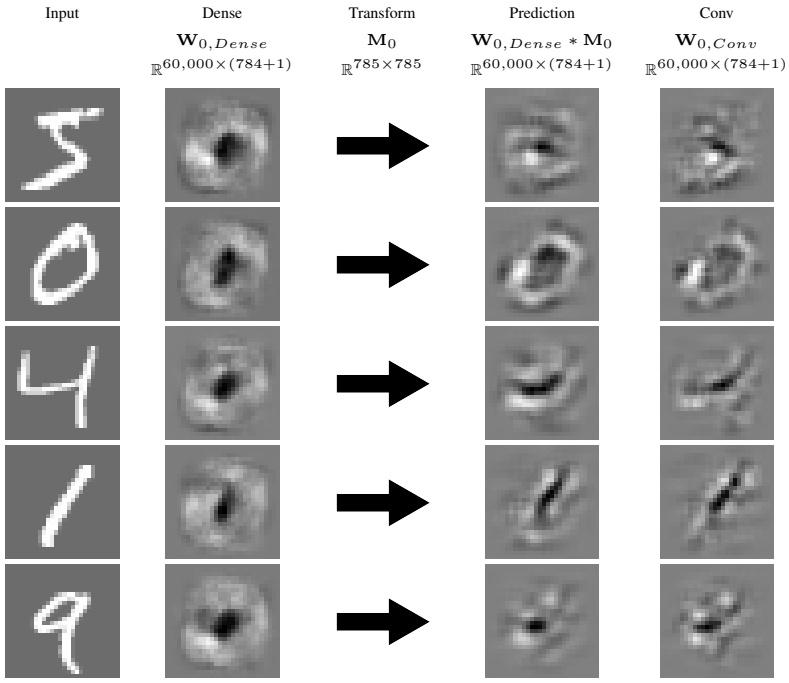


Figure 7.7: An example of the affine mapping from a dense network to a simple convolutional network.

able to represent each other well and transfer to the Adamw simple and complex convolutional networks, but the Adamw convolutional networks appear significantly less able to represent each other.

Portions of these results match what we might expect to see — the ResNets and Inceptionv3 are trained on a different problem, so they are likely to be dissimilar from the other networks. Dense networks and convolutional networks are qualitatively dissimilar in how they approach network architecture, so dissimilarity between the two and similarity within the two is reasonable. However, there are some surprises. The fact that ResNet18 is able to represent the non-pretrained networks on MNIST, but ResNet152 and Inceptionv3 suggests there may be a qualitative difference in complexity between ResNet18 and the larger networks. The dissimilarity of results on SGD and Adamw suggest that hyperparameter selection can play a large role in the behavior of networks that is not obvious merely from looking at the accuracies.

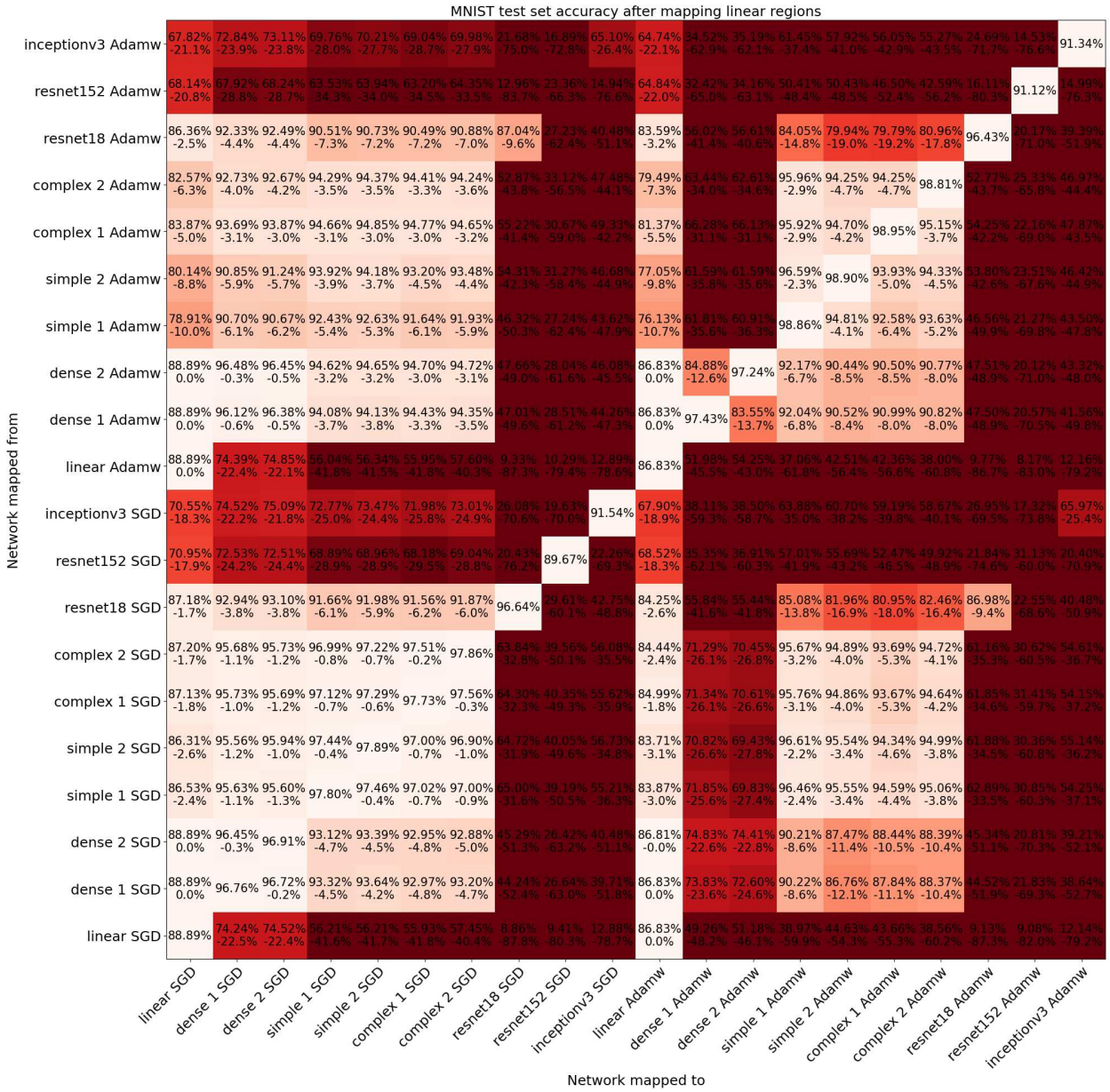


Figure 7.8: The accuracies of networks trained on the MNIST dataset after finding optimal affine maps from the linear coefficients of the networks listed on the left to the linear coefficients of the networks listed on the bottom. The diagonal reports the networks' original accuracies before mapping. Values reported are the percentage of correctly labeled test set samples out of 10,000. Negative values under reported percentages is the percentage point drop in accuracy from the target network's accuracy. Note that the number of affine mappings for a given network is technically 10 times larger than stated in the table — each output of the network has a different mapping trained.

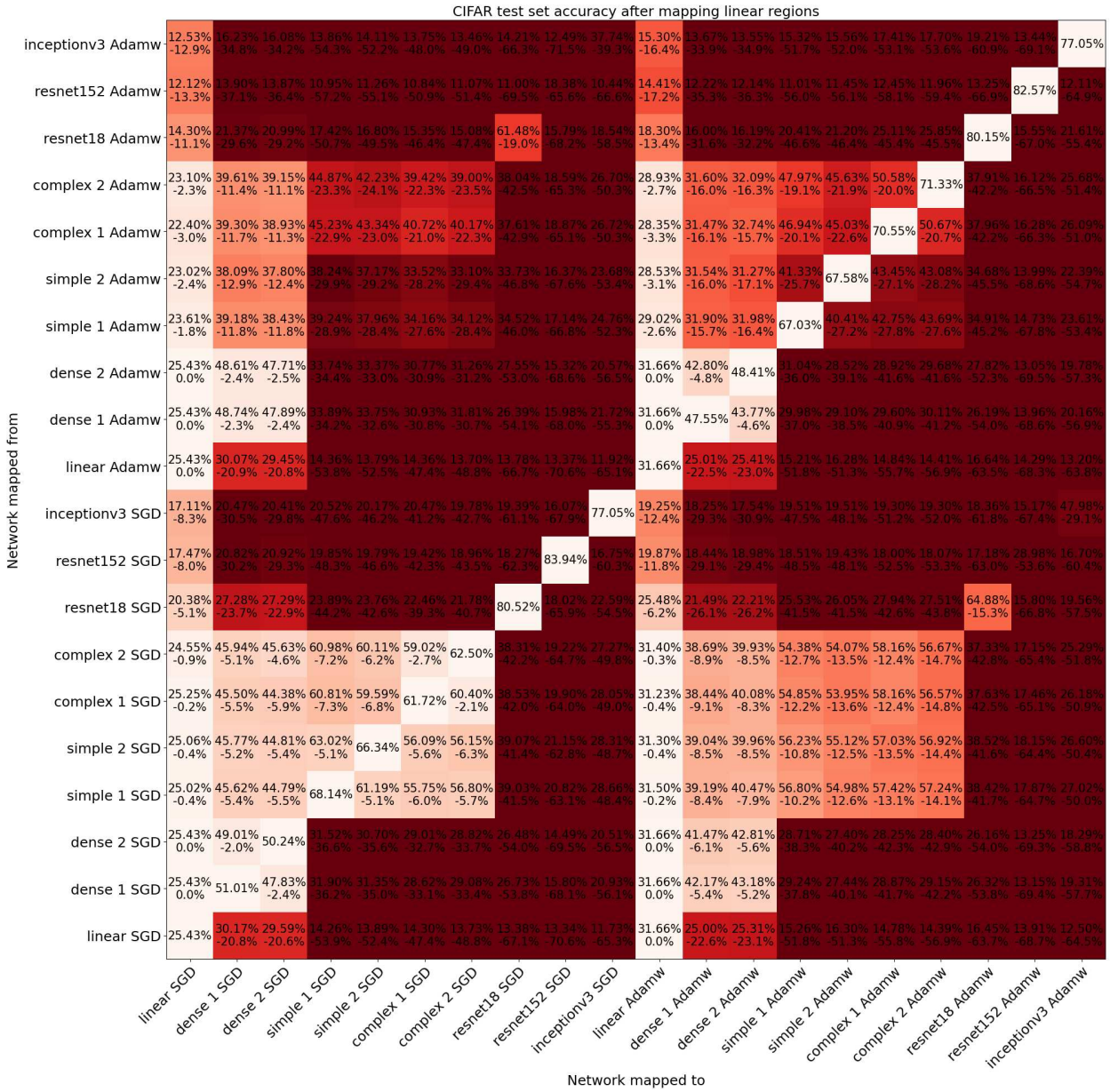


Figure 7.9: The accuracies of networks trained on the CIFAR dataset after finding optimal affine maps from the linear coefficients of the networks listed on the left to the linear coefficients of the networks listed on the bottom. The diagonal reports the networks' original accuracies before mapping. Values reported are the percentage of correctly labeled test set samples out of 10,000. Negative values under reported percentages is the percentage point drop in accuracy from the target network's accuracy. Note that the number of affine mappings for a given network is technically 10 times larger than stated in the table — each output of the network has a different mapping trained.

7.3.2 Mean Squared Euclidean Distance

Using network accuracy as a part of our metric maintains some of the issues that come from using test set accuracy as a measure in general. Behavior is reliant on a good choice of testing data, and may not generalize to data that is distinct from the dataset. Instead, we can look at measures of distance directly between the linear coefficient matrices. Averaged squared Euclidean distance gives an idea of to what extent the linear coefficients match directly between two networks. Since classification occurs by identifying the maximum of the ten outputs, network behavior is invariant to positive scaling of the values. Therefore, to avoid issues of scale when comparing using Euclidean distance, we can define

$$a_{network} = \max_{j\in\{1,\ldots,10\},i\in\{1,\ldots p\},k\in\{1,\ldots,d+1\}} (\tilde{C}_{j,network})_{i,k}, \tag{7.5}$$

the largest value in all of the linear coefficients for a given network. By rescaling the $\tilde{C}$ by the a values, the magnitudes across different networks will be comparable. Then, the distance can be calculated as

$$\frac{1}{10p(d+1)} \sum_{j=1}^{10} || \frac{\tilde{C}_{j,network1}}{a_{network1}} - \frac{\tilde{C}_{j,network2}}{a_{network2}} ||_F^2 \tag{7.6}$$

where $||\cdot||_F$ is the Frobenius norm.

Similarity results under this metric are in Figures 7.10 and 7.11 for MNIST and CIFAR, respectively. In both cases, there is a high degree of similarity among all the non-linear networks. This is problematic for two reasons. First, it means the metric is nondiagnostic — we know the ResNets and Inceptionv3 should appear distinct from the other networks, so if they appear similar than that means this metric may not successfully determine when two networks are dissimilar. Second, it differs from the results in Section 7.3.1. We may expect different measures of similarity to disagree in some respects, but not to the extent that we have significantly qualitatively different results.

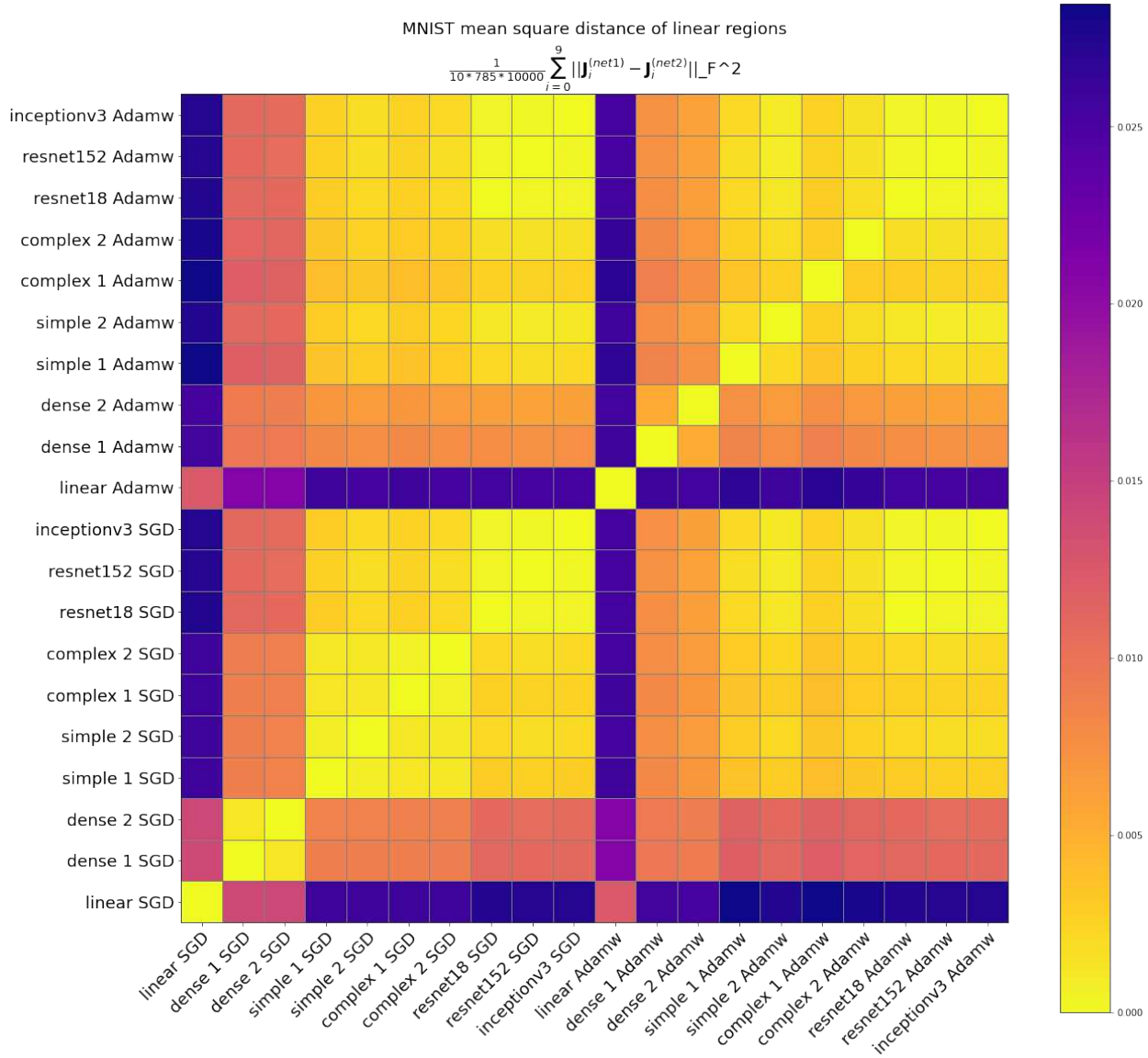


Figure 7.10: Average squared Euclidean distance for the linear coefficients of networks trained on MNIST.

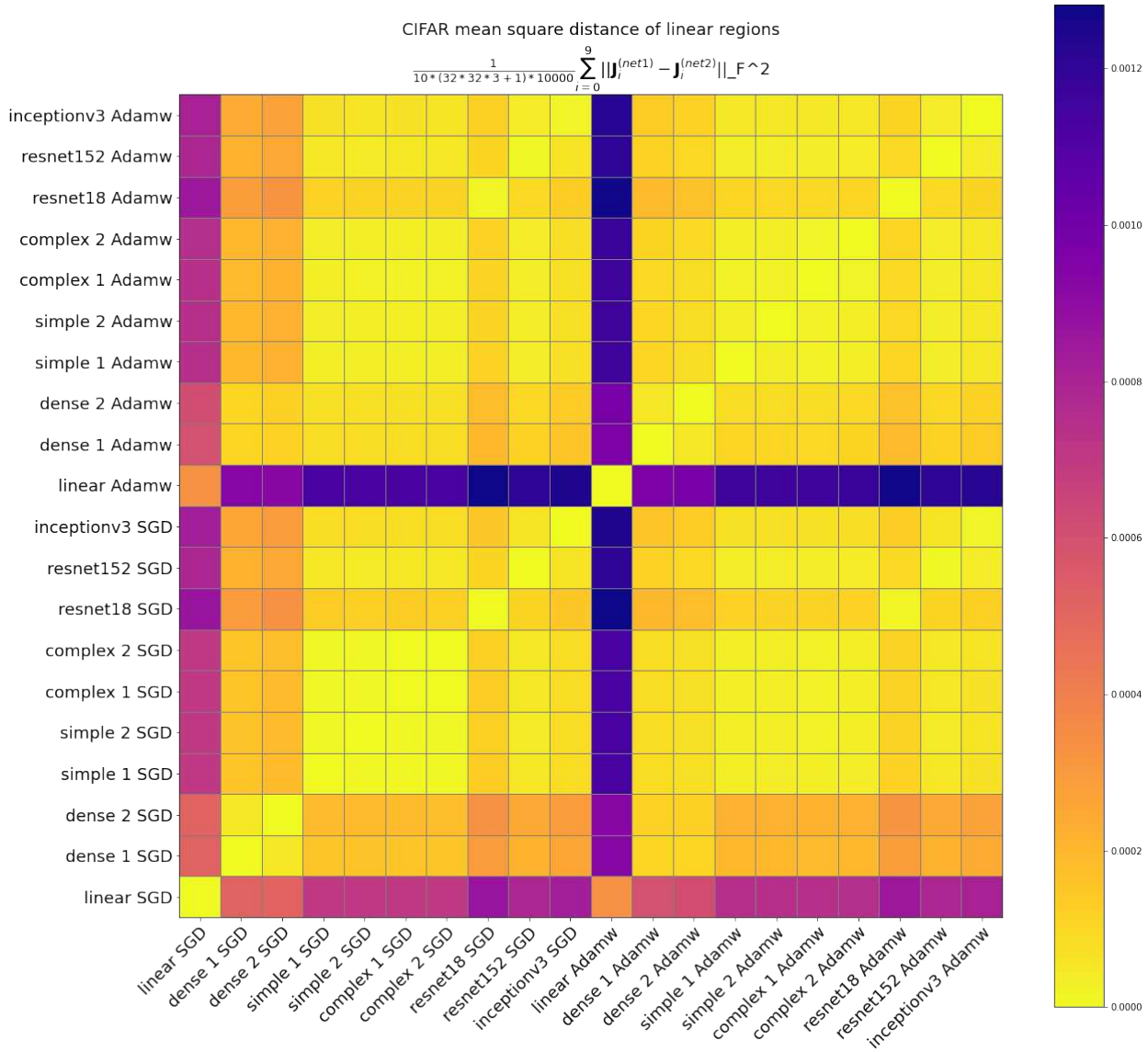


Figure 7.11: Average squared Euclidean distance for the linear coefficients of networks trained on CIFAR.

7.3.3 Centered Kernel Alignment

Since Euclidean distance has potentially poor behavior, we additionally consider angles between the subspaces. We use linear centered kernel alignment (CKA) as described by Kornblith *et al.* [126] and originally constructed by Cristianini *et al.* [127]. This metric is invariant to orthogonal transform and uniform scaling. For two sets of points, X and Y, it is calculated as

$$\text{CKA}(X_j, Y_j) = \frac{||Y_j^T X_j||_F^2}{||X_j^T X_j||_F ||Y_j^T Y_j||_F} \tag{7.7}$$

where $||\cdot||_F$ refers to the Frobenius norm (calculated as the square root of the sum of the squared matrix elements). The X_j and Y_j values we will consider are the $\tilde{C}_{j,network1}$ and $\tilde{C}_{j,network2}$ with mean centered columns. We calculate the mean CKA value across all $j = 1, .., 10$.

Results are shown in Figures 7.12 and 7.13, respectively. Results are similar to, but with some distinctions from, the results in Section 7.3.1. For MNIST, Adamw networks are largely distinct, except for the dense and linear networks. For SGD, the dense and linear networks are again similar, but the non-pretrained convolutional networks form a similar block as well. The ResNets and Inceptionv3 trained on SGD and Adamw are similar to the version trained using the other optimizer. Similarity between SGD and Adam network is otherwise low.

For CIFAR, the dense and linear networks regardless of training method are similar. Again, for SGD the convolutional networks are similar, but now there is additionally some similarity for the complex convolutional networks when trained using Adamw. Cross-optimizer similarity is again low.

These results match both our hypotheses about the behavior of networks and the results in Section 7.3.1. That suggests that this method may be a stronger choice for measuring similarity than the Euclidean distance in Section 7.3.2

7.3.4 Cross-Entropy of Outputs

It is useful to have a control to compare similarity scores to. In this case, we investigate cross-entropy, $H : [0,1]^n \times [0,1]^n \to \mathbb{R}_{\geq 0}$ which is defined between two n-dimensional probability

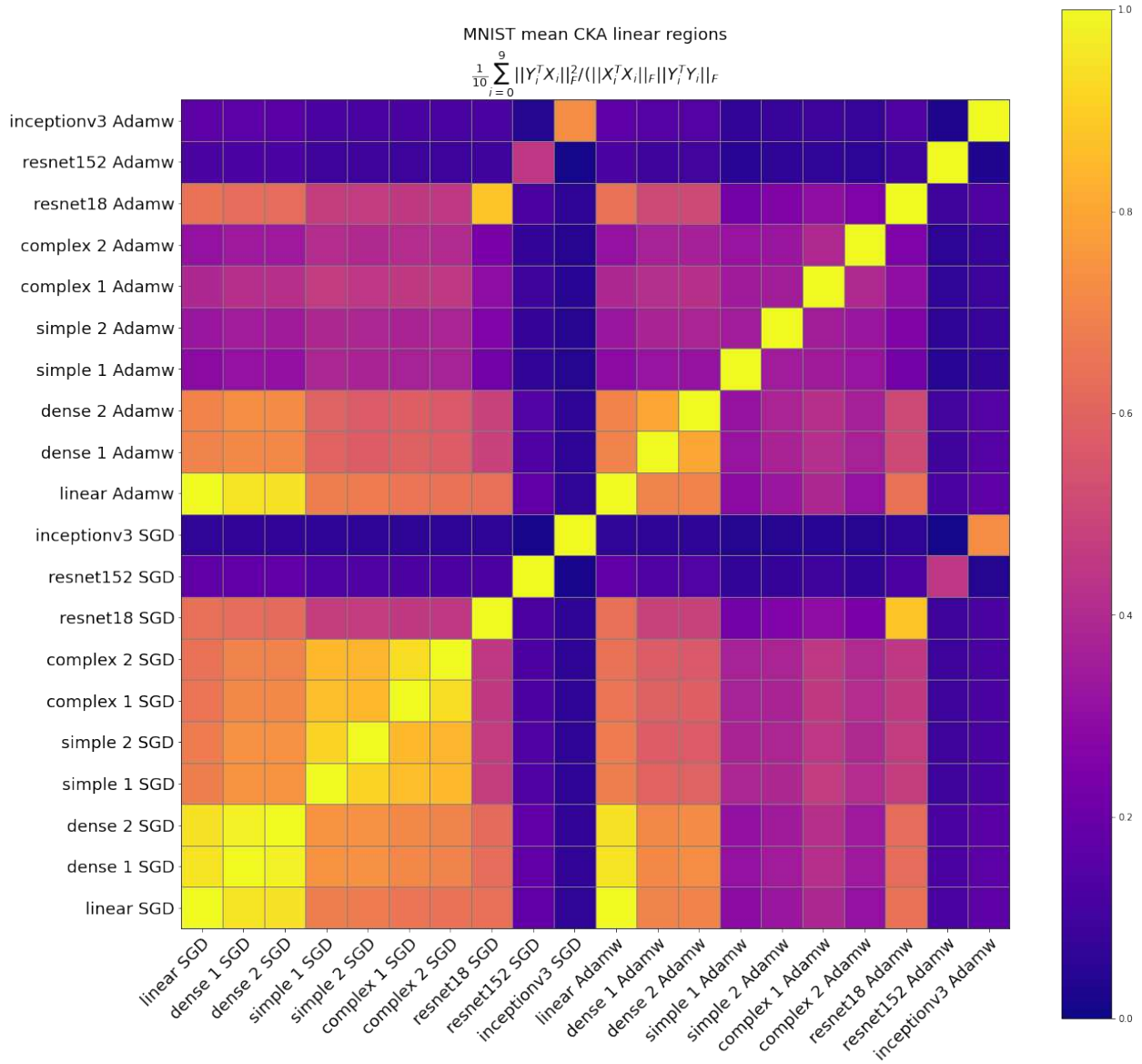


Figure 7.12: Centered kernel alignment similarity scores for linear coefficients of networks trained on MNIST.

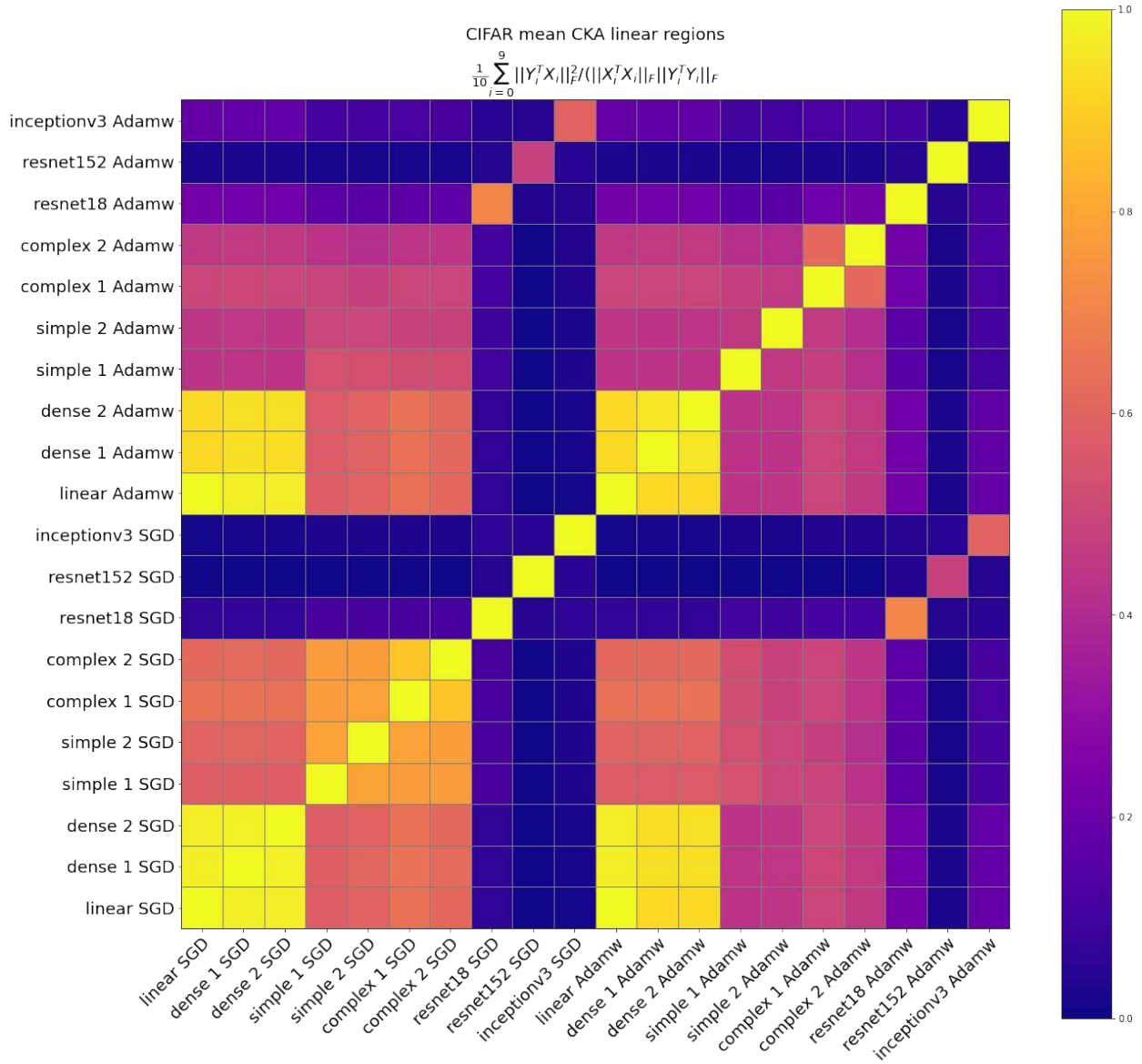


Figure 7.13: Centered kernel alignment similarity scores for linear coefficients of networks trained on CIFAR.

distributions p and q, with the cross-entropy of q relative to p as

$$H(p, q) = -\sum_{i}^{n} p \log q. \tag{7.8}$$

Cross-entropy is applied to the outputs of the networks after a softmax, $\sigma : \mathbb{R}^n \to [0, 1]^n$ which is a generalization of the logistic function to multiple dimensions [128, 129]. For outputs of a network $y \in \mathbb{R}^n$,

$$\sigma(y)_i = \frac{e^{y_i}}{\sum_{j=1}^{n} e^{y_j}}. \tag{7.9}$$

Cross-entropy of the outputs after softmax compares the probability distributions of the networks across input data. As a loss function, this is used in knowledge distillation work where the goal is to train a small network to recover the outputs of a larger network [130]. The fact that this is used practically is appealing, as it ensures that results will have a relation to practical interests.

Results for MNIST are shown in Figure 7.14. ResNet152 trained with SGD is difficult for networks to represent to the point that its dissimilarity overwhelms all other results, so results are shown with a cap on maximum entropy in Figure 7.15. Similarly, raw results for CIFAR are shown in Figure 7.16, with linear network being difficult to represent, so capped results are shown in Figure 7.16. Results are similar to previous similarity metrics, where the dense networks and convolutional networks are similar to themselves for SGD, but there is reduced similarity for Adamw and going between SGD and Adamw.

7.4 Conclusion

We have presented one metric for measuring the degree to which networks may be overly complex in terms of number of linear regions and three novel metrics for investigating similarity of networks. The complexity metric is relevant for determining that dense networks are similar to linear networks and that networks trained on MNIST are likely to find simple solutions. However, it is not as relevant for networks on CIFAR, suggesting it may not be helpful in a broader sense.

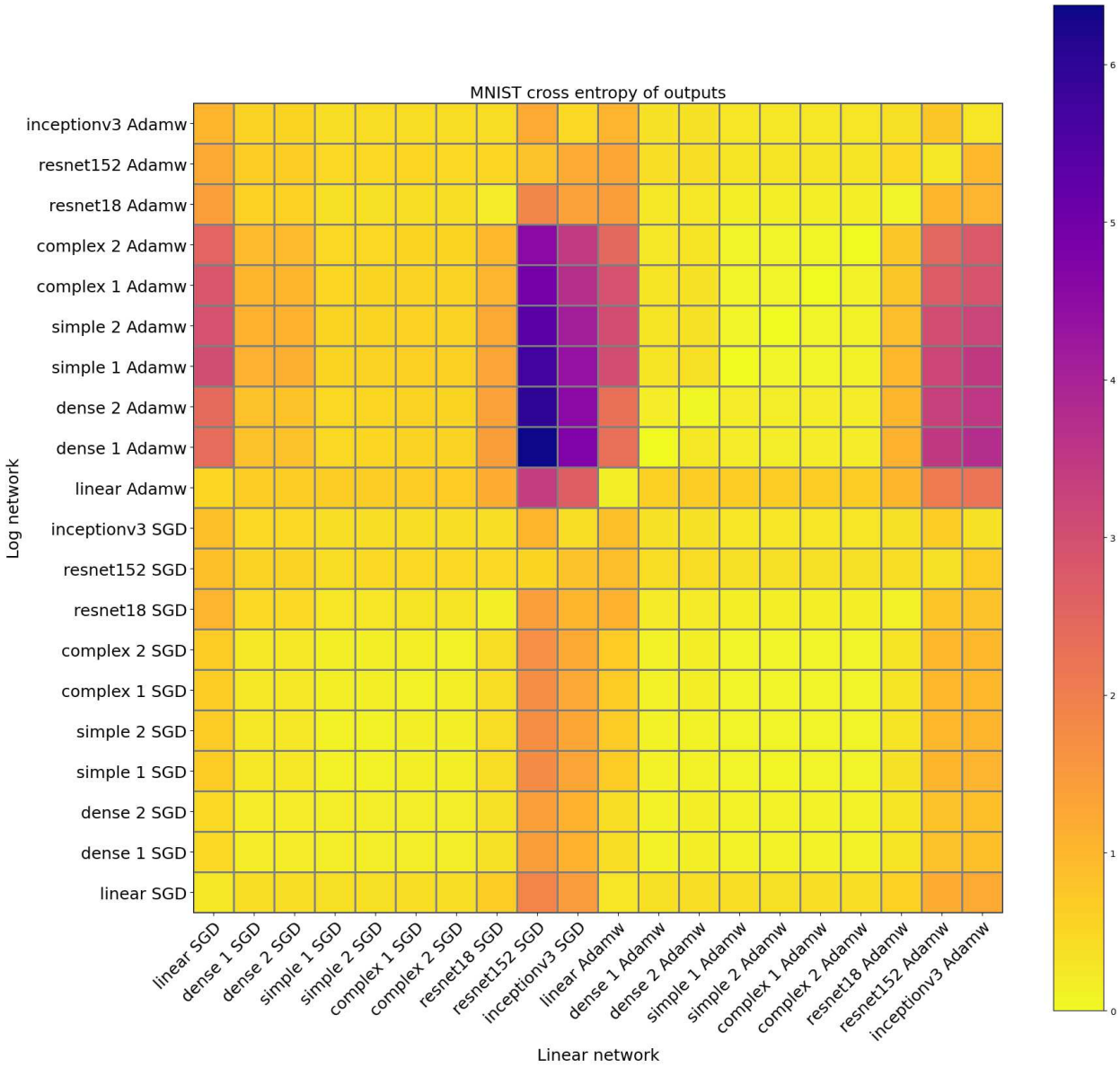


Figure 7.14: Softmax output cross-entropy similarity scores for networks trained on MNIST.

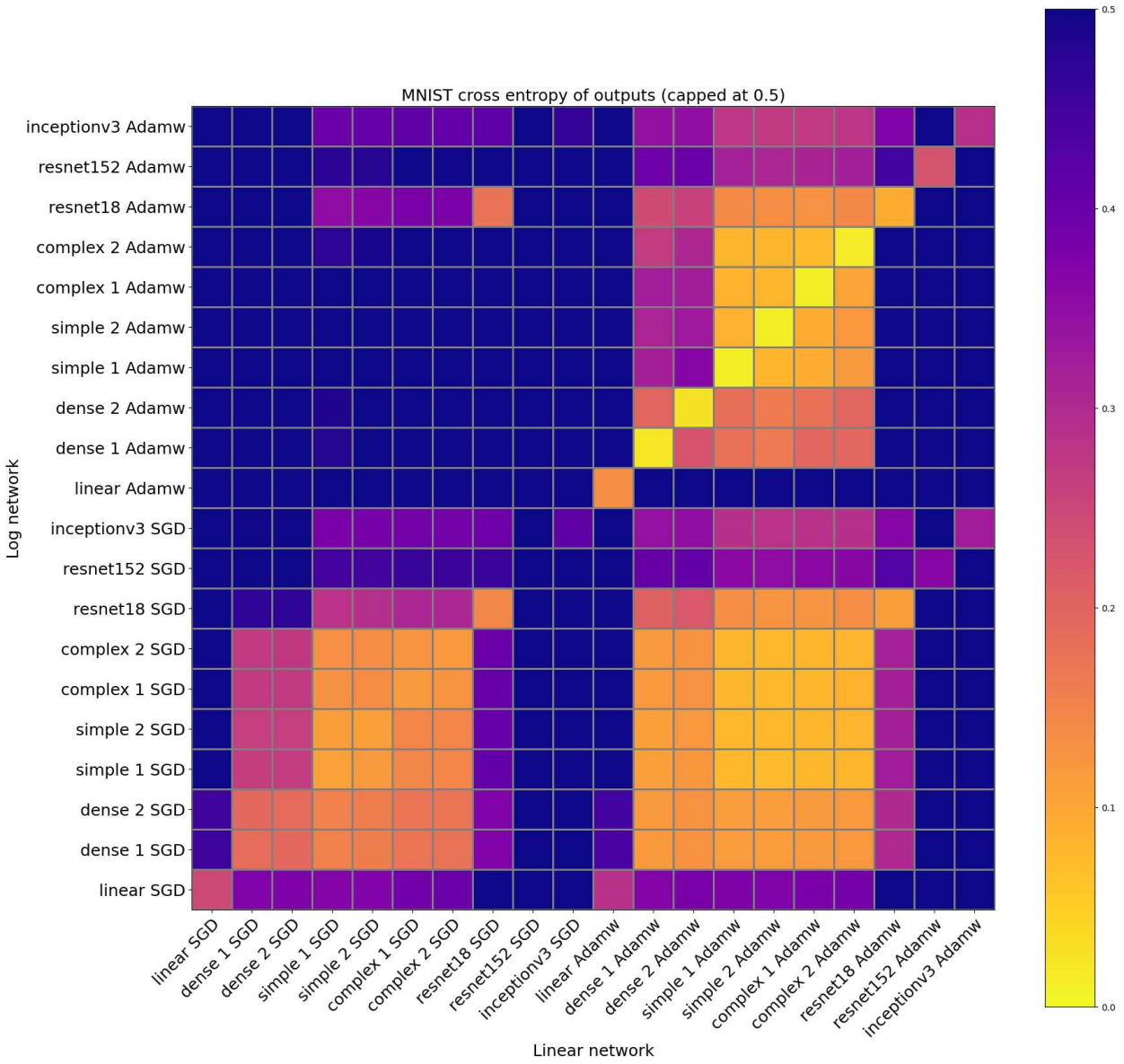


Figure 7.15: Softmax output cross-entropy similarity scores for networks trained on MNIST. Coloration is capped to show more behavior of networks that are similar, rather than being overwhelmed by very dissimilar networks.

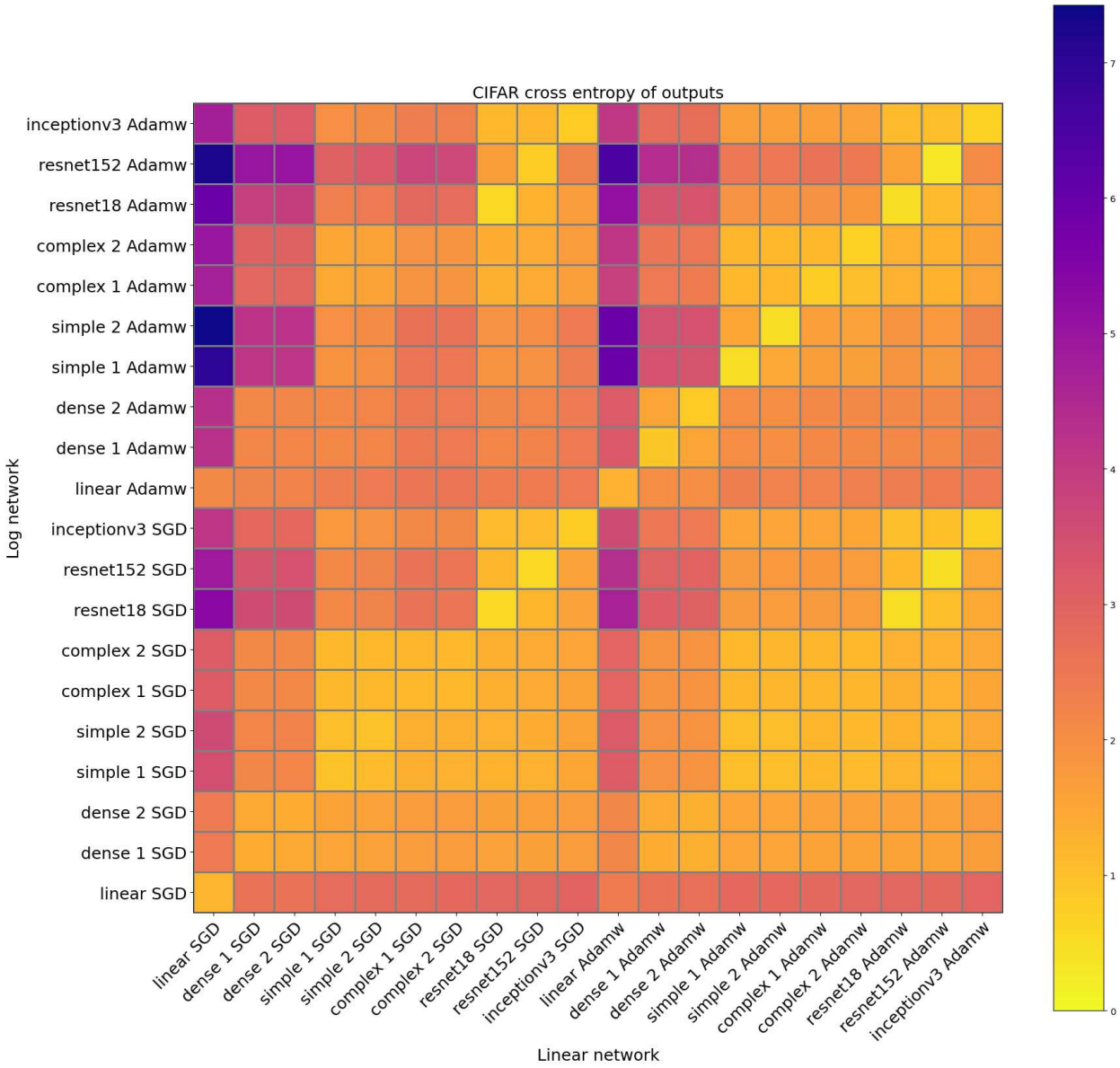


Figure 7.16: Softmax output cross-entropy similarity scores for networks trained on CIFAR.

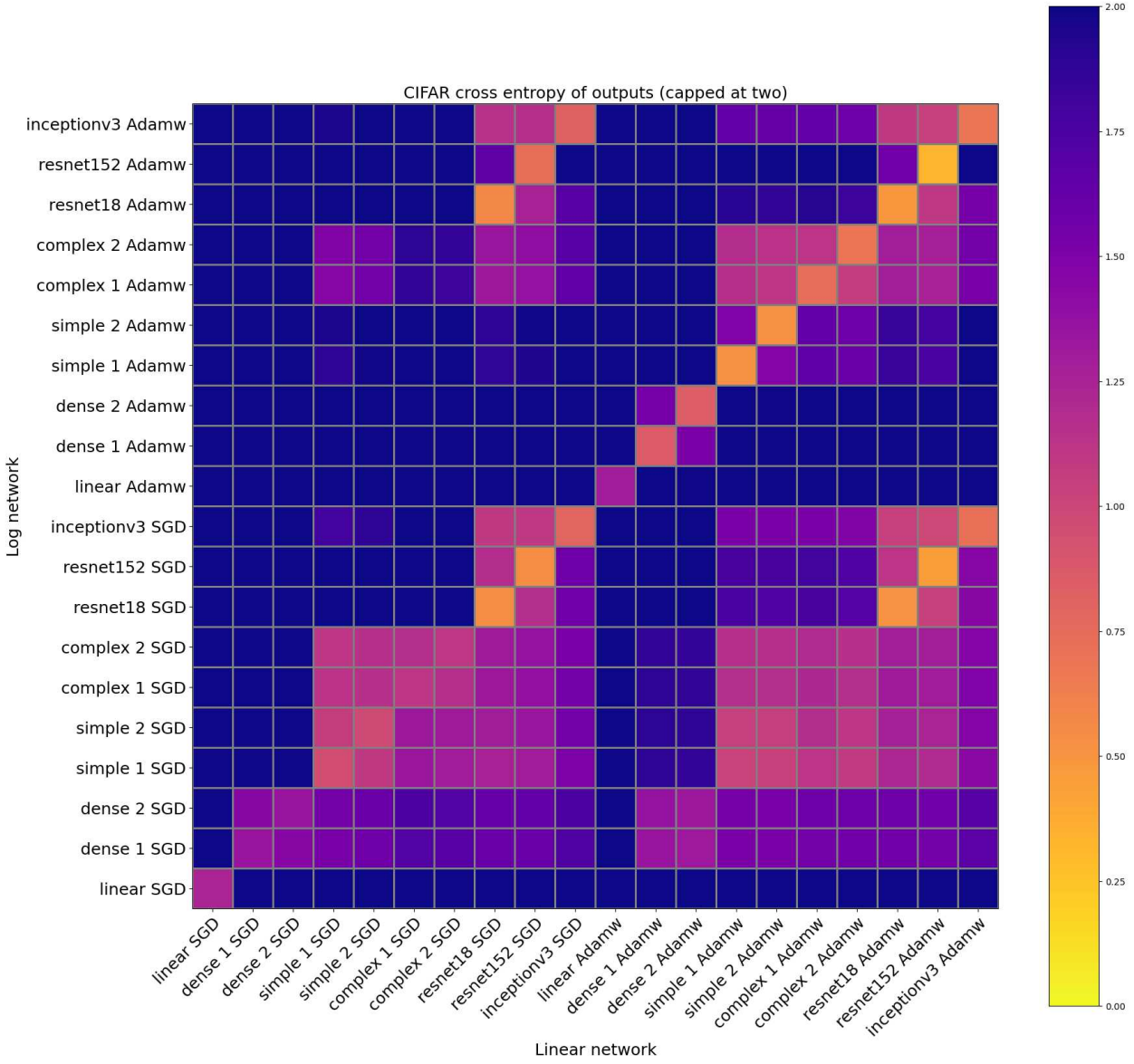


Figure 7.17: Softmax output cross-entropy similarity scores for networks trained on CIFAR. Coloration is capped to show more behavior of networks that are similar, rather than being overwhelmed by very dissimilar networks.

For the similarity metrics, we have affine mapped representation performance, Euclidean distance, and centered kernel alignment on the linear coefficients of the networks with cross-entropy of the post-softmax outputs as a control. Affine mapped representation performance, centered kernel alignment, and cross-entropy have results that have qualitatively similar behavior, but Euclidean distance is distinct. From the results, we can determine that single-hidden layer networks tend to behave similarly, and have overlap with linear classifiers. Convolutional networks have distinct behavior for the fully-connected dense network, and in most cases the convolutional networks are similar to each other. Networks trained on other problems (the ResNets and Inception) have behavior that is distinct, although ResNet18 is potentially able to represent the behavior of simpler networks.

Chapter 8

Conclusion

Using linearizations and simple problems to understand the black box behavior of neural networks gives us significant insight into network behavior. This method also lends itself naturally to visualizations, which is helpful for building human understanding. Creating such understanding is vital for ensuring that neural networks behave well, rather than simply acting as correlation machines that encode any bias in the dataset or their creator. Determining what the network is doing, even locally, allows for that behavior to be examined in a meaningful way.

By analyzing simple problems and exploiting the piecewise linear nature of the ReLU activation function, we find further evidence that networks of minimal size to represent a problem do not perform well when training, but they can be recovered by either using good choices of hyperparameters or by removing unnecessary weights or structures in the network after training an overparameterized network. Overparameterized networks have significantly improved training performance over these minimal networks, and are similar in behavior to smaller networks.

We are also able to identify what structures small neural networks are able to represent. Width two networks are able to represent any Boolean functions, and parity is solvable by a network with size logarithmic in the input dimension, but the required size of single-hidden-layer network increases exponentially in the dimension for the hardest problems in each dimension. Width $n + 1$ single-hidden-layer networks are able to classify an n-hypersphere versus its surrounding hyperannulus, but only if the distance between the hypersphere and hyperannulus grows as $\Omega(\sqrt{n})$.

Linearization allows us to compare or compress networks. Significant effort is given to training novel methodologies that do not necessarily have meaningfully different behavior, merely being slightly more efficient hyperparameter tuning for an over-studied problem [131]. Determining the extent to which such networks have qualitatively different behavior allows for the focus to be on meaningful shifts, rather than slight tweaks that do not generalize. By using linearizations, either through Taylor series approximations or decomposition of the ReLU activation function, we can

investigate local behavior of of neural networks. Specifically in the case of piecewise linear neural networks, this has a natural meaning with the geometry of the neural network, as the network is a piecewise linear function. Each piece forms a convex polytope in the original input space in which a linear map of the inputs uniquely determines the network's behavior.

By visualizing the structure of these polytopes, we can investigate how the neural network uses its hidden nodes to solve a problem of interest. Constructing the visualizations of these polytope structures and the coefficients associated with them as animations throughout the training process for two-dimensional problems creates a natural way of building intuition for the polytope region behavior while still gaining insight into how a network solves a particular problem. We can identify how behavior we see matches to previous results in the field, namely that networks tend to learn from a bottom up perspective [68] and how networks that have additional complexity tend to find solutions faster [29].

Considering the coefficients associated with these linear regions also provides a method for analyzing network behavior. By considering the toy two-dimensional problems, we can identify that networks tend to converge to solutions which have clusters of coefficients — suggesting that many linear regions have functions that are nearly equivalent. This matches with previous work from fields such as pruning and the lottery hypothesis [21], and provides a method for determining how tolerant networks may be to compression by reducing the number of regions that networks can use. We can also identify that networks that do find apparently dissimilar solutions frequently have coefficients that are linear transformations of each other — typically rotation. This also matches with previous work from the representational similarity field, where various features of neural networks tend to be related by linear transformations [75].

We also find that fully-connected networks are similar to each other and linear networks but dissimilar from convolutional networks. Convolutional networks are similar to each other, and are frequently able to represent fully-connected networks. Networks trained on more complex datasets and fine-tuned are dissimilar from other networks. This suggests that CIFAR-10 and MNIST are distinct from ImageNet, which is unsurprising considering the difference in resolution. It also

suggests that networks that use similar architectural blocks will identify similar solutions, but that assumptions encoded in those architectural blocks may result in dissimilar behavior from other kinds of network.

Bibliography

[1] Elham Albaroudi, Taha Mansouri, and Ali Alameer. A comprehensive review of ai techniques for addressing algorithmic bias in job hiring. *ai*, 5(1):383–404, 2024.

[2] Kaiji Lu, Piotr Mardziel, Fangjing Wu, Preetam Amancharla, and Anupam Datta. Gender bias in neural natural language processing. In *Logic, language, and security: essays dedicated to Andre Scedrov on the occasion of his 65th birthday*, pages 189–202. Springer, 2020.

[3] James Zou and Londa Schiebinger. Ai can be sexist and racist—it's time to make it fair. *Nature*, 559(7714):324–326, 2018.

[4] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. ImageNet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[5] F. Rosenblatt. *Principles of neurodynamics: perceptrons and the theory of brain mechanisms*. Report (Cornell Aeronautical Laboratory). Spartan Books, 1962.

[6] Marvin Minsky and Seymour Papert. *Perceptrons - an introduction to computational geometry*. MIT Press, 1987.

[7] Boris Hanin and Mark Sellke. Approximating continuous functions by ReLU nets of minimal width. *arXiv preprint arXiv:1710.11278*, 2017.

[8] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, 1989.

[9] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

[10] Zhou Lu, Hongming Pu, Feicheng Wang, Zhiqiang Hu, and Liwei Wang. The expressive power of neural networks: A view from the width. In *Advances in Neural Information Processing Systems*, pages 6231–6239, 2017.

[11] Hongzhou Lin and Stefanie Jegelka. ResNet with one-neuron hidden layers is a universal approximator. In *Advances in Neural Information Processing Systems*, pages 6169–6178, 2018.

[12] Guido F Montufar, Razvan Pascanu, Kyunghyun Cho, and Yoshua Bengio. On the number of linear regions of deep neural networks. In *Advances in Neural Information Processing Systems*, pages 2924–2932, 2014.

[13] Razvan Pascanu, Guido Montufar, and Yoshua Bengio. On the number of response regions of deep feed forward networks with piece-wise linear activations. *arXiv preprint arXiv:1312.6098*, 2013.

[14] Maithra Raghu, Ben Poole, Jon Kleinberg, Surya Ganguli, and Jascha Sohl-Dickstein. On the expressive power of deep neural networks. In *international Conference on Machine Learning*, pages 2847–2854, 2017.

[15] 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*, pages 770–778, 2016.

[16] Boris Hanin and David Rolnick. Complexity of linear regions in deep networks. *arXiv preprint arXiv:1901.09021*, 2019.

[17] Boris Hanin and David Rolnick. Deep ReLU networks have surprisingly few activation patterns. In *Advances in Neural Information Processing Systems*, pages 361–370, 2019.

[18] Roman Novak, Yasaman Bahri, Daniel A Abolafia, Jeffrey Pennington, and Jascha Sohl-Dickstein. Sensitivity and generalization in neural networks: an empirical study. *arXiv preprint arXiv:1802.08760*, 2018.

[19] Xiao Zhang and Dongrui Wu. Empirical studies on the properties of linear regions in deep neural networks. In *International Conference on Learning Representations*, 2020.

[20] Liwen Zhang, Gregory Naitzat, and Lek-Heng Lim. Tropical geometry of deep neural networks. *arXiv preprint arXiv:1805.07091*, 2018.

[21] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations*, 2018.

[22] Guanbin Li and Yizhou Yu. Visual saliency detection based on multiscale deep cnn features. *IEEE Transactions on Image Processing*, 25(11):5012–5024, 2016.

[23] Zhou Lu, Hongming Pu, Feicheng Wang, Zhiqiang Hu, and Liwei Wang. The expressive power of neural networks: A view from the width. In *Advances in Neural Information Processing Systems*, pages 6231–6239, 2017.

[24] Boris Hanin and Mark Sellke. Approximating continuous functions by relu nets of minimal width. *ArXiv*, 10 2017.

[25] Sho Sonoda and Noboru Murata. Neural network with unbounded activation functions is universal approximator. *Applied and Computational Harmonic Analysis*, 43(2):233 – 268, 2017.

[26] George Cybenko. Approximation by superpositions of a sigmoidal function. *MCSS*, 2:303–314, 1989.

[27] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359 – 366, 1989.

[28] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251 – 257, 1991.

[29] Anna Choromanska, Mikael Henaff, Michael Mathieu, Gérard Ben Arous, and Yann LeCun. The loss surfaces of multilayer networks. In *Artificial Intelligence and Statistics*, pages 192–204, 2015.

[30] Eran Malach, Gilad Yehudai, Shai Shalev-Schwartz, and Ohad Shamir. Proving the lottery ticket hypothesis: Pruning is all you need. In *International Conference on Machine Learning*, pages 6682–6691. PMLR, 2020.

[31] Zeyuan Allen-Zhu, Yuanzhi Li, and Zhao Song. A convergence theory for deep learning via over-parameterization. In *International conference on machine learning*, pages 242–252. PMLR, 2019.

[32] Guido Montúfar, Razvan Pascanu, Kyunghyun Cho, and Yoshua Bengio. On the number of linear regions of deep neural networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, pages 2924–2932, Cambridge, MA, USA, 2014. MIT Press.

[33] Maithra Raghu, Ben Poole, Jon Kleinberg, Surya Ganguli, and Jascha Sohl-Dickstein. On the expressive power of deep neural networks. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 2847–2854, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR.

[34] Liwen Zhang, Gregory Naitzat, and Lek-Heng Lim. Tropical geometry of deep neural networks. In *ICML*, 2018.

[35] Raman Arora, Amitabh Basu, Poorya Mianjy, and Anirbit Mukherjee. Understanding deep neural networks with rectified linear units. *arXiv preprint arXiv:1611.01491*, 2016.

[36] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998.

[37] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[38] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4):193–202, Apr 1980.

[39] David E. Rumelhart, James L. McClelland, and PDP Research Group. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations*. MIT Press, Cambridge, MA, USA, 1986.

[40] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back propagating errors. *Nature*, 323:533–536, 1986.

[41] A. L. Hodgkin and A. F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *Bulletin of Mathematical Biology*, 52(1):25–71, Jan 1990.

[42] Augustin Cauchy et al. Méthode générale pour la résolution des systemes d'équations simultanées. *Comp. Rend. Sci. Paris*, 25(1847):536–538, 1847.

[43] P. J. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University, 1974.

[44] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[45] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[46] Peter L Bartlett, Nick Harvey, Christopher Liaw, and Abbas Mehrabian. Nearly-tight vc-dimension and pseudodimension bounds for piecewise linear neural networks. *Journal of Machine Learning Research*, 20(63):1–17, 2019.

[47] Dumitru Erhan, Yoshua Bengio, Aaron Courville, and Pascal Vincent. Visualizing higher-layer features of a deep network. *University of Montreal*, 1341(3):1, 2009.

[48] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.

[49] Chris Olah, Alexander Mordvintsev, and Ludwig Schubert. Feature visualization. *Distill*, 2017. https://distill.pub/2017/feature-visualization.

[50] Been Kim, Martin Wattenberg, Justin Gilmer, Carrie Cai, James Wexler, Fernanda Viegas, and Rory Sayres. Interpretability beyond feature attribution: Quantitative testing with concept activation vectors (tcav). *arXiv preprint arXiv:1711.11279*, 2017.

[51] Bolei Zhou, Aditya Khosla, Agata Lapedriza, Aude Oliva, and Antonio Torralba. Learning deep features for discriminative localization. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2921–2929, 2016.

[52] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[53] Karen Ullrich, Edward Meeds, and Max Welling. Soft weight-sharing for neural network compression. *arXiv preprint arXiv:1702.04008*, 2017.

[54] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

[55] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. What is the state of neural network pruning? *arXiv preprint arXiv:2003.03033*, 2020.

[56] Zhuangwei Zhuang, Mingkui Tan, Bohan Zhuang, Jing Liu, Yong Guo, Qingyao Wu, Junzhou Huang, and Jinhui Zhu. Discrimination-aware channel pruning for deep neural networks. *Advances in neural information processing systems*, 31, 2018.

[57] Tailin Liang, John Glossner, Lei Wang, Shaobo Shi, and Xiaotong Zhang. Pruning and quantization for deep neural network acceleration: A survey. *Neurocomputing*, 461:370–403, 2021.

[58] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. Rethinking the value of network pruning. In *International Conference on Learning Representations*, 2018.

[59] Ian J Goodfellow, Oriol Vinyals, and Andrew M Saxe. Qualitatively characterizing neural network optimization problems. *arXiv preprint arXiv:1412.6544*, 2014.

[60] Armin Eftekhari. Training linear neural networks: Non-local convergence and complexity results. In *International Conference on Machine Learning*, pages 2836–2847. PMLR, 2020.

[61] Sanjeev Arora, Nadav Cohen, Noah Golowich, and Wei Hu. A convergence analysis of gradient descent for deep linear neural networks. *arXiv preprint arXiv:1810.02281*, 2018.

[62] Gabin Maxime Nguegnang, Holger Rauhut, and Ulrich Terstiege. Convergence of gradient descent for learning linear neural networks. *Advances in Continuous and Discrete Models*, 2024(1):23, 2024.

[63] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 2002.

[64] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[65] Yixuan Li, Jason Yosinski, Jeff Clune, Hod Lipson, and John E Hopcroft. Convergent learning: Do different neural networks learn the same representations? In *FE@ NIPS*, pages 196–212, 2015.

[66] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[67] Liwei Wang, Lunjia Hu, Jiayuan Gu, Yue Wu, Zhiqiang Hu, Kun He, and John Hopcroft. Towards understanding learning representations: To what extent do different neural networks learn the same representation. *arXiv preprint arXiv:1810.11750*, 2018.

[68] Maithra Raghu, Justin Gilmer, Jason Yosinski, and Jascha Sohl-Dickstein. Svcca: Singular vector canonical correlation analysis for deep learning dynamics and interpretability. In *Advances in Neural Information Processing Systems*, pages 6076–6085, 2017.

[69] Ari Morcos, Maithra Raghu, and Samy Bengio. Insights on representational similarity in neural networks with canonical correlation. In *Advances in Neural Information Processing Systems*, pages 5727–5736, 2018.

[70] Simon Kornblith, Mohammad Norouzi, Honglak Lee, and Geoffrey Hinton. Similarity of neural network representations revisited. *arXiv preprint arXiv:1905.00414*, 2019.

[71] Karel Lenc and Andrea Vedaldi. Understanding image representations by measuring their equivariance and equivalence. *International Journal of Computer Vision*, 127(5):456–476, May 2019.

[72] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[73] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv preprint arXiv:1312.6034*, 2013.

[74] David McNeely-White, J. Beveridge, and Bruce Draper. Inception and ResNet features are (almost) equivalent. *Cognitive Systems Research*, 59, 10 2019.

[75] David G McNeely-White. Same data, same features: Modern imagenet-trained convolutional neural networks learn the same thing. Master's thesis, Colorado State University, 2020.

[76] David McNeely-White, Ben Sattelberg, Nathaniel Blanchard, and Ross Beveridge. Exploring the interchangeability of CNN embedding spaces. *arXiv preprint arXiv:2010.02323v4*, 2020.

[77] Huma Jamil, Yajing Liu, Nathaniel Blanchard, Michael Kirby, and Chris Peterson. Leveraging linear mapping for model-agnostic adversarial defense. *Frontiers in Computer Science*, 5:1274832, 2023.

[78] Huma Jamil, Yajing Liu, Christina Cole, Nathaniel Blanchard, Emily J King, Michael Kirby, and Christopher Peterson. Dual graphs of polyhedral decompositions for the detection of adversarial attacks. In *2022 IEEE International Conference on Big Data (Big Data)*, pages 2913–2921. IEEE, 2022.

[79] Yajing Liu, Turgay Caglar, Christopher Peterson, and Michael Kirby. Integrating geometries of relu feedforward neural networks. *Frontiers in big Data*, 6:1274831, 2023.

[80] Marco Ancona, Enea Ceolini, Cengiz Öztireli, and Markus Gross. Towards better understanding of gradient-based attribution methods for deep neural networks. *arXiv preprint arXiv:1711.06104*, 2017.

[81] Motasem Alfarra, Adel Bibi, Hasan Hammoud, Mohamed Gaafar, and Bernard Ghanem. On the decision boundaries of neural networks: A tropical geometry perspective. *arXiv preprint arXiv:2002.08838*, 2020.

[82] Martin Trimmel, Henning Petzka, and Cristian Sminchisescu. Tropex: An algorithm for extracting linear terms in deep neural networks. In *International Conference on Learning Representations*, 2020.

[83] Tong Yu and Hong Zhu. Hyper-parameter optimization: A review of algorithms and applications. *arXiv preprint arXiv:2003.05689*, 2020.

[84] Mohaimenul Azam Khan Raiaan, Sadman Sakib, Nur Mohammad Fahad, Abdullah Al Mamun, Md Anisur Rahman, Swakkhar Shatabda, and Md Saddam Hossain Mukta. A systematic review of hyperparameter optimization techniques in convolutional neural networks. *Decision Analytics Journal*, 11:100470, 2024.

[85] Charles H Martin and Michael W Mahoney. Implicit self-regularization in deep neural networks: Evidence from random matrix theory and implications for learning. *arXiv preprint arXiv:1810.01075*, 2018.

[86] James O'Donnell and Casey Crownhart. Everything you need to know about estimating ai's energy and emissions burden. *MIT Technology Review*, 2025.

[87] Kai-Yeung Siu, Vwani Roychowdhury, and Thomas Kailath. *Discrete neural computation: A theoretical foundation*. Prentice-Hall, Inc., 1995.

[88] Martin Anthony. Boolean functions and artificial neural networks. *Boolean Functions*, 2, 2003.

[89] Shai Shalev-Shwartz, Ohad Shamir, and Shaked Shammah. Failures of gradient-based deep learning. In *International Conference on Machine Learning*, pages 3067–3075. PMLR, 2017.

[90] Claude E Shannon. The synthesis of two-terminal switching circuits. *The Bell System Technical Journal*, 28(1):59–98, 1949.

[91] Elizabeth Ann Ernst. *Optimal combinational multi-level logic synthesis*. University of Michigan, 2009.

[92] Culliney, Young, Nakagawa, and Muroga. Results of the synthesis of optimal networks of and and or gates for four-variable switching functions. *IEEE Transactions on Computers*, C-28(1):76–85, 1979.

[93] Fausto Carcassi and Jakub Szymanik. Neural networks track the logical complexity of boolean concepts. *Open Mind*, 6:132–146, 2022.

[94] Xiao Han. A new bound for the fourier-entropy-influence conjecture. *Combinatorica*, 45(1):4, 2025.

[95] Ehud Friedgut and Gil Kalai. Every monotone graph property has a sharp threshold. *Proceedings of the American mathematical Society*, 124(10):2993–3002, 1996.

[96] Yishay Mansour. An o (n log log n) learning algorithm for dnf under the uniform distribution. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 53–61, 1992.

[97] Parikshit Gopalan, Adam Tauman Kalai, and Adam R Klivans. Agnostically learning decision trees. In *Proceedings of the fortieth annual ACM symposium on Theory of computing*, pages 527–536, 2008.

[98] Chris Mingard, Joar Skalse, Guillermo Valle-Pérez, David Martínez-Rubio, Vladimir Mikulik, and Ard A Louis. Neural networks are a priori biased towards boolean functions with low entropy. *arXiv preprint arXiv:1909.11522*, 2019.

[99] Alexandr Andoni, Rina Panigrahy, Gregory Valiant, and Li Zhang. Learning polynomials with neural networks. In *International conference on machine learning*, pages 1908–1916. PMLR, 2014.

[100] Chao-Kong Chow. On the characterization of threshold functions. In *2nd Annual Symposium on Switching Circuit Theory and Logical Design (SWCT 1961)*, pages 34–38. IEEE, 1961.

[101] OEIS Foundation Inc. The On-Line Encyclopedia of Integer Sequences. Published electronically at http://oeis.org.

[102] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, January 2017.

[103] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 929–947, 2024.

[104] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[105] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. *arXiv preprint arXiv:1611.03530*, 2016.

[106] Roger Arthur Johnson. *Modern geometry: an elementary treatise on the geometry of the triangle and the circle*. Houghton, Mifflin Company, 1929.

[107] Harold Scott Macdonald Coxeter. *Regular polytopes*. Courier Corporation, 1973.

[108] HSM Coxeter. Regular and semi-regular polytopes. i. *Mathematische Zeitschrift*, 46(1):380–407, 1940.

[109] HSM Coxeter. Regular and semi-regular polytopes. ii. *Mathematische Zeitschrift*, 188(4):559–591, 1985.

[110] HSM Coxeter. Regular and semi-regular polytopes. iii. *Mathematische Zeitschrift*, 200(1):3–45, 1988.

[111] Nova Division. Tetrahedron with edges, 2011. This file is made available under the Creative Commons CC0 1.0 Universal Public Domain Dedication.

[112] tomruen. Cuboctahedron-tetrahedral symmetry colored, 2025. This file is made available under the Creative Commons CC0 1.0 Universal Public Domain Dedication.

[113] John H Conway, Heidi Burgiel, and Chaim Goodman-Strauss. *The symmetries of things*. CRC Press, 2016.

[114] Richard Klitzing. Polytopes & their incidence matrices. https://bendwavy.org/klitzing/explain/analog.htm#exp-simplex, 2026. Accessed: 2026-2-27.

[115] Brook Taylor. *Methodus incrementorum directa & inversa*. Inny, 1717.

[116] Paul G. Constantine. *Active Subspaces*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2015.

[117] William Sir Thomson. On the division of space with minimum partitional area. *Acta mathematica*, 11(1):121–134, 1887.

[118] Martin Trimmel, Henning Petzka, and Cristian Sminchisescu. Tropex: An algorithm for extracting linear terms in deep neural networks. *International Conference on Learning Representations*, 2021.

[119] Ben Sattelberg, Renzo Cavalieri, Michael Kirby, Chris Peterson, and Ross Beveridge. Locally linear attributes of relu neural networks. *Frontiers in Artificial Intelligence*, 6:1255192, 2023.

[120] Vinay Uday Prabhu and Abeba Birhane. Large image datasets: A pyrrhic win for computer vision? *arXiv preprint arXiv:2006.16923*, 2020.

[121] Antonio Torralba, Rob Fergus, and William T Freeman. 80 million tiny images: A large data set for nonparametric object and scene recognition. *IEEE transactions on pattern analysis and machine intelligence*, 30(11):1958–1970, 2008.

[122] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[123] Sébastien Marcel and Yann Rodriguez. Torchvision the machine-vision package of torch. In *Proceedings of the 18th ACM international conference on Multimedia*, pages 1485–1488, 2010.

[124] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 2818–2826, 2016.

[125] Jaehoon Lee, Lechao Xiao, Samuel Schoenholz, Yasaman Bahri, Roman Novak, Jascha Sohl-Dickstein, and Jeffrey Pennington. Wide neural networks of any depth evolve as linear models under gradient descent. In *Advances in neural information processing systems*, pages 8572–8583, 2019.

[126] Simon Kornblith, Mohammad Norouzi, Honglak Lee, and Geoffrey Hinton. Similarity of neural network representations revisited. In *International Conference on Machine Learning*, pages 3519–3529. PMLR, 2019.

[127] Nello Cristianini, John Shawe-Taylor, André Elisseeff, and Jaz Kandola. On kernel-target alignment. In T. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems*, volume 14. MIT Press, 2002.

[128] John Bridle. Training stochastic model recognition algorithms as networks can lead to maximum mutual information estimation of parameters. *Advances in neural information processing systems*, 2, 1989.

[129] John S Bridle. Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. In *Neurocomputing: Algorithms, architectures and applications*, pages 227–236. Springer, 1990.

[130] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[131] Kiri L Wagstaff. Machine learning that matters. In *Proceedings of the 29th International Coference on International Conference on Machine Learning*, pages 1851–1856, 2012.

Appendix A

Proofs Relating to Boolean Functions

A.1 Proof of Theorem 3.3.1

Input negation and permutation invariance can be demonstrated by showing that the swaps function is invariant to nonzero affine transforms of the inputs to the function. Negation of the output follows from the fact that $|f(x_i) - f(x_{i+1}| = | - f(x_i) - (-f(x_{i+1}))|$.

Lemma A.1.1 (Swaps orthonormal invariance)**.** *For the set of binary strings, $B = \{-1, 1\}^d$, Boolean function $f : B \to \{-1, 1\}$ with swaps value s, and orthonormal matrix $P \in \mathbb{R}^{n \times n}$ such that $PP^T = I_n$, the mapping $\tilde{f}(Pc) = f(c)$ for $c \in B$ is a function and has swaps value s.*

Proof. Because P is invertible, for any two vectors $c_1, c_2 \in B$, $Pc_1 = Pc_2$ if and only if $c_1 = c_2$. Therefore, since f is a function, $\tilde{f}$ is a function. We then need to demonstrate that the set of possible orderings of B is equivalent to the set of orderings of $\tilde{B} = \{Pc|c \in B\}$ to confirm that $\tilde{f}$ has swaps value s.

First define the set of ordering vectors of $\tilde{B}$,

$$\tilde{W} = \{\tilde{w} \in \mathbb{R}^d | \forall \tilde{c}_1, c_2 \in \tilde{B}, \tilde{c}_1^\intercal \tilde{w} = \tilde{c}_2^\intercal \tilde{w} \implies \tilde{f}(\tilde{c}_1) = f(\tilde{c}_2)\}. \tag{A.1}$$

We wish to demonstrate that for W as defined in Equation (3.8)

$$\tilde{W} = \{Pw | w \in W\} \tag{A.2}$$

For any $w \in W$ as defined in Equation (3.8), there exists a $\tilde{w} = Pw$ such that for any $b \in B$,

$$(Pb)^T \tilde{w} = (Pb)^T Pw = b^T P^T Pw = b^T w. \tag{A.3}$$

Therefore for $\tilde{c}_1, \tilde{c}_2 \in \tilde{B}$, $\tilde{c}_1^T \tilde{W}^T = \tilde{c}_1^T \tilde{W}^T \implies \tilde{f}(\tilde{c}_1) = \tilde{f}(\tilde{c}_2)$. Thus, $\tilde{w} \in \tilde{W}$, and so $\{Pw | w \in W\} \subseteq \tilde{W}$. Similarly we can demonstrate that for any $\tilde{v} \in \tilde{W}$ there is a $v = P\tilde{v} \in W$, and so $\tilde{W} \subseteq \{Pw | w \in W\}$. Therefore Equation (A.2) holds.

The set of possible orderings of B and the set of possible orderings of $\tilde{B}$ are therefore equivalent, and so the swaps values of f and $\tilde{f}$ must be equivalent. □

Lemma A.1.2 (Swaps scaling invariance)**.** *For the set of binary strings, $B = \{a_1, a_2\}^d$, Boolean function $f : B \to \{b_1, b_2\}$ with swaps value s, and diagonal matrix $D \in \mathbb{R}^{d \times d}$ with nonzero diagonal entries, the Boolean function $\tilde{f}(Db) = f(b)$ for $b \in B$ has swaps value s.*

Proof. The proof of this follows exactly as in Theorem A.1.1, except for the replacement of P with D, and for Equation (A.3), we need $\tilde{w} = D^{-1}w$ so that

$$(Db)^T \tilde{w} = (Db)^T D^{-1} w = b^T D^T D^{-1} w = b^T D D^{-1} w = b^T w. \tag{A.4}$$

□

Lemma A.1.3 (Swaps translation invariance)**.** *For the set of binary strings, $B = \{a_1, a_2\}^d$, Boolean function $f : B \to \{b_1, b_2\}$ with swaps value s, and vector $q \in \mathbb{R}^d$, the Boolean function $\tilde{f}(b+q) = f(b)$ for $b \in B$ has swaps value s.*

Proof. The proof of this follows exactly as in Theorem A.1.1, except for the replacement of multiplication by P with addition by q, and for Equation (A.3), we have

$$(b+q)^T \tilde{w} = b^T w + q^T w. \tag{A.5}$$

$q^T w$ is constant for any choice of B, and so the addition of q will not affect the ordering. □

Permutation is an orthonormal transform of the inputs, so by Lemma A.1.1, the swaps function is invariant to permutation of the inputs. Negation of of an input is an affine transform of the inputs, so by Lemmas A.1.1, A.1.2, and A.1.3, the swaps function is invariant to negations of the input.

A.2 Proof of Theorem 3.3.2

Theorem 3.3.2 states that given s regions in $\mathbb{R}$, $(x_{i,1}, x_{i,2}), i = 1, ...k$ with $x_{i-1,2} \leq x_{i,1}$ with alternating positive and negative classes, there exists a Leaky ReLU neural network, $v : \mathbb{R} \to \mathbb{R}$ with a single hidden layer containing $s - 1$ nodes that correctly classifies each region.

An example of the geometry for this construction is in Figure A.1.

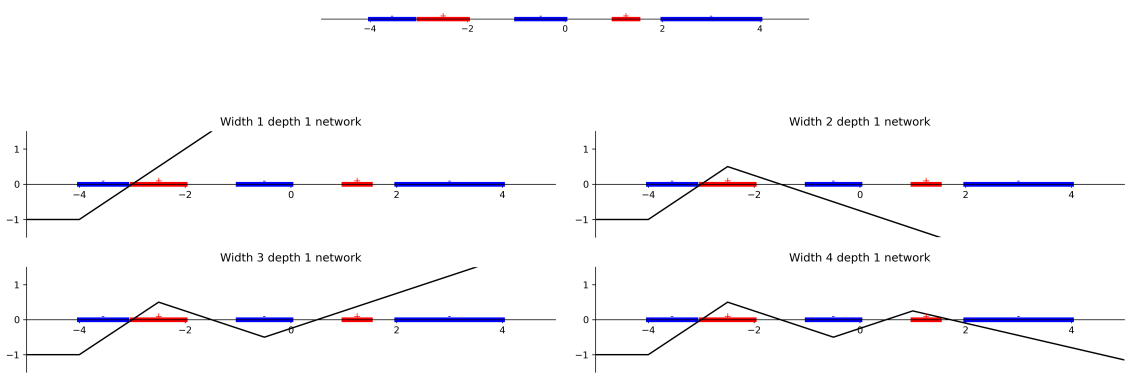


Figure A.1: The top image shows a simple one-dimensional two-class classification problem, with four changes in class. The other images show the effect of adding additional ReLU nodes to a single-hidden-layer network.

We first prove a lemma about regression on s points,

Lemma A.2.1. *Given k points in $\mathbb{R}$, $x_1 < x_2 < ... < x_k$, and a function defined on them, $f : \{x_1, ..., x_k\} \to \mathbb{R}$, there exists a Leaky ReLU neural network, $v : \mathbb{R} \to \mathbb{R}$ with a single hidden layer containing $k - 1$ nodes such that for all $x_i, i = 1, ..., k$, $v(x_i) = f(x_i)$ with the nonlinearities of the network occurring at $x_1, ..., x_{k-1}$.*

Proof. A network with Leaky ReLU nonlinearities with leak factor $0 \leq \alpha < 1$, v, can be written as

$$v(x) = d + \sum_{i=1}^{k-1} v_i \max(\alpha(w_i x + b_i), w_i x + b_i). \tag{A.6}$$

We need only consider ReLUs that are activated on the right, that is $w_i \geq 0$. The network can then be reparameterized for $a_i = v_i w_i$ and $c_i = \frac{b_i}{w_i}$ as

$$v(x) = d + \sum_{i=1}^{k-1} a_i \max(\alpha(x + c_i), x + c_i). \tag{A.7}$$

We wish for the nonlinearities to occur at $x_1, ..., x_{k-1}$, so we can set $c_i = -x_i$:

$$v(x) = d + \sum_{i=1}^{k-1} a_i \max(\alpha(x - x_i), x - x_i). \tag{A.8}$$

When evaluated at a point $x_j < x < x_{j+1}$ this becomes

$$v(x) = d + \sum_{i=1}^{k-1} a_i \max(\alpha(x - x_i), x - x_i). \tag{A.9}$$

$$= d + \sum_{i=1}^{j} a_i(x - x_i) + \sum_{i=j+1}^{k-1} \alpha a_i(x - x_i). \tag{A.10}$$

This has slope

$$v'(x) = \sum_{i=1}^{j} a_i + \sum_{i=j+1}^{k-1} \alpha a_i \tag{A.11}$$

and the slope necessary to go from x_j to x_{j+1} with a linear function is

$$s(x) = \frac{f(x_{j+1}) - f(x_j)}{x_{j+1} - x_j}, x_j < x < x_{j+1}. \tag{A.12}$$

Setting $v'(x) = s(x)$ in each of the $k - 1$ linear regions yields the linear system

$$\begin{bmatrix} \frac{f(x_2)-f(x_1)}{x_2-x_1} \\ \frac{f(x_3)-f(x_2)}{x_3-x_2} \\ \vdots \\ \frac{f(x_k)-f(x_{k-1})}{x_k-x_{k-1}} \end{bmatrix} = M \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_{k-1} \end{bmatrix} \tag{A.13}$$

where

$$M_{ij} = \begin{cases} 1, & i \leq j \\ \alpha, & i > j \end{cases} \tag{A.14}$$

The matrix M is invertible with inverse

$$M_{ij}^{-1} = \begin{cases} \frac{1}{1-\alpha}, & i = j \\ \frac{1}{\alpha-1}, & i = j+1 \\ \frac{\alpha}{\alpha-1}, & i = 0, j = k-1 \\ 0, & \text{else.} \end{cases} \tag{A.15}$$

Finally, setting

$$d = f(x_1) - \sum_{i=1}^{k-1} a_i \max(\alpha(x_1 - x_i), x_1 - x_i) \tag{A.16}$$

yields the neural network v with a single hidden layer containing $k-1$ nodes such that for all $x_i, i = 1, ..., k$, $v(x_i) = f(x_i)$ with the nonlinearities of the network occurring at $x_1, ..., x_{k-1}$.

□

Then the proof for Theorem 3.3.2 follows as

Proof. By setting $y_i = \frac{x_{i,1}+x_{i,2}}{2}$ with associated function $f : \{y_1, ..., y_k\} \to \mathbb{R}$ such that $f(y_1) = 1$ if the positive class is first or -1 if the negative class is first, and

$$a = \frac{x_{i-1,2} + x_{i,1}}{2} \tag{A.17}$$

$$m = \frac{-f(y_{i-1})}{a - y_{i-1}} \tag{A.18}$$

$$b = f(y_{i-1}) - m y_{i-1} \tag{A.19}$$

$$f(y_i) = m y_i + b \tag{A.20}$$

so that changes in class occur between $x_{i-1,2}$ and $x_{i,1}$. From Lemma A.2.1, there exists a Leaky ReLU neural network with a single hidden layer with nonlinearities occurring at the y_i that achieves the values of $f(y_i)$ at each y_i. By constructing the values of the $f(y_i)$, the slope in each region guarantees that the network has a class change between $x_{i-1,2}$ and $x_{i,1}$. Therefore the function

achieves positive value in each positive-class region and negative values in each negative-class region. □

A.3 Proof of Theorem 3.3.3

Theorem 3.3.3 states there exists a Leaky ReLU neural network, $v : \mathbb{R} \to \mathbb{R}$ with l hidden layers, each of width w, that partitions a region of $\mathbb{R}$ into w^l alternating positive and negative regions.

An example of the geometry for this construction is in Figure A.2.

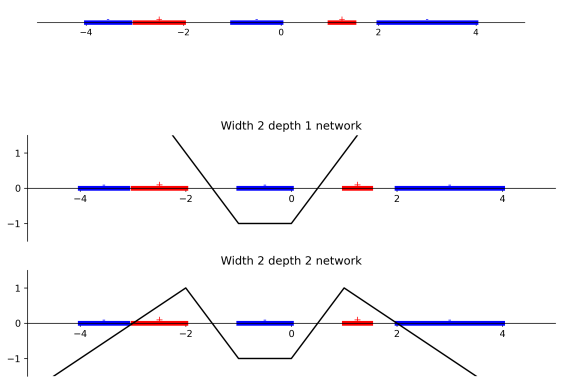


Figure A.2: The top image shows a simple one-dimensional two-class classification problem, with four changes in class. The other images show the effect of adding additional layers to a width two ReLU network, where the second layer negates the slope of the first layer when the single-dimensional output of the first layer exceeds one.

Proof. From Lemma A.2.1, we can construct a single-hidden-layer network of width w that forms a triangle wave with peaks one and troughs zero in the region $[a, b], a < b \in \mathbb{R}$. This network will have w inflection points. A second layer can be added to this network, using the single output of the first layer as the input to each node. We wish for the second layer, to match the piecewise linear function $f : [0, 1] \to \mathbb{R}$,

$$f_2(x) = \begin{cases} wx, & x \leq \frac{1}{w} \\ \vdots & \\ w(-1)^i(x - \frac{i}{w}), & \frac{i}{w} < x \leq \frac{i+1}{w} \\ \vdots & \\ w(-1)^{w-1}(x - \frac{w-1}{w}), & \frac{w-1}{w} < x \end{cases} \tag{A.21}$$

From Theorem A.2.1 there is a network that approximates this, using the points $x_i = \frac{i}{w}, i = 0, ..., w-1$ and regression targets $f(x_i)$. This will take each of the w triangles in the first layer and transform it into w new triangles with peak one and trough zero. This results in w^2 triangles, and can be iterated l times to result in w^l triangles. Setting the final bias of the network to -0.5 will result in w^l changes in sign across the region $[0, 1]$. This network partitions the region B into w^l alternating positive and negative regions, with changes in class occurring at the points $a_i = a + i\frac{b-a}{w^l}$. Choice of the sign of the original network sign can be changed by negating the final weight and bias of the network. □

A.4 Proof of Theorem 3.3.4

Theorem 3.3.4 states let v be a neural network with depth l and width w. Then, v can solve s-parity if $n \leq w^l$.

Proof. n-parity can be mapped to one-dimension using the vector with each element b and the scalar c, resulting in b being the negative class and then alternating positive and negative classes at the values $ib + c, i = 1, ..., n$.

From Theorem 3.3.3, we can construct a network with width w and depth l such that it alternates positive and negative regions, starting negative, in the region $[a, b], a < b \in \mathbb{R}$ at the points $a_i = a + i\frac{b-a}{w^l}$. Setting $c = a + \frac{b-a}{2w^l}$ and $b = \frac{b-a}{w^l}$ will have each of the n points of n-parity occurring in the first n partitions of the network, each of which will have the correct sign for classification. □

A.5 Proof of NPN invariance of Fourier degree, Fourier entropy, and influence

Proof. **Output negation** For the n-input Boolean function $f : \{-1, 1\}^n \to \{-1, 1\}$ with Fourier polynomial

$$f_{\mathbb{R}}(x) = \sum_{S \subseteq [n]} \hat{f}(S)\chi_S(x), \qquad \chi_S(x) = \prod_{i \in S} x_i, \qquad \hat{f}(S) = \sum_{x \in \{-1,1\}^n} f(x)\chi_S, \tag{A.22}$$

the function generated by negation of the output is $g(x) = -f(x)$ which has Fourier coefficients $\hat{g}(S) = -\hat{f}(S)$. Both influence and Fourier entropy vary only in the square of the Fourier coefficients, and so f and g have the same influence and entropy. The Fourier degree varies only in the degree of the highest monomial, which will remain the same.

Permutation of inputs Permutation of inputs is equivalent to a relabeling of the inputs, and so will only affect the ordering of the Fourier coefficients and which variables are in the highest degree monomial.

Negation of inputs For negation of the inputs, a function g with x_k negated has Fourier coefficients

$$\hat{g}(S) = \begin{cases} \hat{f}(S), & x_k \notin S \\ -\hat{f}(S), & x_k \in S. \end{cases} \tag{A.23}$$

Both Fourier influence and Fourier entropy vary only in the square of the Fourier coefficients, and so f and g have the same influence and entropy. The same terms are in the Fourier series, and so the Fourier degree will remain the same. □