ABSTRACT

Title of dissertation: INTERPRETING MACHINE LEARNING MODELS AND APPLICATION OF HOMOTOPY METHODS

Roozbeh Yousefzadeh
Doctor of Philosophy, 2019

Dissertation directed by: Professor Dianne P. O'Leary
Department of Computer Science

Neural networks have been criticized for their lack of easy interpretation, which undermines confidence in their use for important applications. We show that a trained neural network can be interpreted using flip points. A flip point is any point that lies on the boundary between two output classes: e.g. for a neural network with a binary yes/no output, a flip point is any input that generates equal scores for "yes" and "no". The flip point closest to a given input is of particular importance, and this point is the solution to a well-posed optimization problem. We show that computing closest flip points allows us, for example, to systematically investigate the decision boundaries of trained networks, to interpret and audit them with respect to individual inputs and entire datasets, and to find vulnerability against adversarial attacks. We demonstrate that flip points can help identify mistakes made by a model, improve its accuracy, and reveal the most influential features for classifications. We also show that some common assumptions about the decision boundaries of neural networks can be unreliable. Additionally, we present methods for designing the structure of feed-forward networks using matrix conditioning. At the end, we investigate an unsupervised learning method, the Gaussian graphical model, and provide mathematical tools for interpretation.

Interpreting Machine Learning Models and Application of Homotopy Methods

by

Roozbeh Yousefzadeh

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2019

Advisory Committee:
Professor Dianne P. O'Leary, Chair/Advisor
Professor Howard Elman
Professor Furong Huang
Professor Thomas Goldstein
Professor Joseph JaJa

To my loving and wonderful parents!

Acknowledgments

My deepest gratitude goes to my advisor, Dianne O'Leary, for being a great mentor for me. I also thank her for giving me this great idea of studying neural networks as functions and educating me with her deep understanding of scientific computing. She also taught me how to write scientific papers, how to present my research ideas better, and how to write mathematical formulations for complex problems. Her continuous feedbacks greatly helped me improve my writing and thinking. She always was a committed professor and an outstanding mentor, and always gave me the most candid and helpful advice.

Also, Professor O'Leary's lecture notes for her Scientific Computing courses, and her carefully written Survival Guides were really helpful for me, early in my studies, before she became my advisor. I vividly remember the first time I met her in her office. I still have the piece of paper that she gave me with lots of helpful information and pointers at the end of that meeting. Since then, she has been greatly impactful in my life, as someone who is wiser, broad-minded, and honest.

I also thank the committee members of my dissertation. I appreciate their careful reading of my dissertation and their helpful questions, comments, and suggestions. It has been great to be in touch with each of them and to benefit from their insights and suggestions.

I thank Professor Elman for being on my preliminary exam and the dissertation committees, and for his helpful comments. It has been nice to be in touch with him these years and to benefit from his advice. Being his TA was also a great experience for me and I was honored when he let me teach a couple of his CMSC460 lectures.

I thank Professor Huang, who was on both my preliminary exam and dissertation

committees, for her helpful and detailed comments, and also for her advice.

I thank Professor Goldstein for being on my dissertation committee despite being on sabbatical, and also for his helpful suggestions. Taking his course was educational, too.

Professor JaJa's comments were thoughtful and his suggestions, useful. I sincerely thank him for being on my dissertation committee.

The work presented in Chapter 7 was partially supported by the Applied Machine Learning Fellowship at Los Alamos National Laboratory. It was a great experience for me to work with Diane Oyen, and I thank her for that.

Walking along the bridge from Structural Engineering to Computer Science was a gradual process and I remember most of my steps along that bridge, and how different people helped me or inspired me.

Professor O'Leary has been a true inspiration and help during these years.

I thank Professor Daniel Robinson for teaching nonlinear optimization in its best way. Professor Robinson's class was one of the most educational and enjoyable classes I have taken.

I learned a lot from all the courses I took as a graduate student in the Computer Science Department. That knowledge has proven to be very useful. The first CS course I took was with Professor Don Perlis which I found fascinating, especially because he kept emphasizing on the history of AI and how ideas have evolved to current practices. I thank Professor Jeff Foster for his helpful and clear advice when I talked to him about the Computer Science program, and when I stopped by his office a few times after that. Professor David Mount's lecture notes are amazingly educational. I learned a lot from his

CMSC451 notes and also from his CMSC 754 class. I hope one day I use Computational Geometry to optimize the form of a space structure.

Professor Elman's Scientific Computing class was one of the other pieces that drew me to computer science, when I attended several weeks of his lectures. I had learned Finite Element Method as an engineer, by the engineer's standards. But, when I saw how he teaches FEM, it was much more fascinating and profound. Taking Professor Goldstein's class was inspiring, too. He is a great teacher and the contents of his class were practical and tangible.

Finally, I thank Poorti, whose friendship has been amazing, these years that I have been working on this dissertation.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

In this dissertation, we study several open research problems in the field of machine learning. We study neural networks with respect to their behavior as a function, their interpretation and debugging, and their structural design, in Chapters 2 through 6. We also study interpretation of Gaussian graphical models as an unsupervised learning method in Chapter 7.

All the problems that we study in this thesis can be described as optimization problems. Most of these problems, especially the ones related to neural networks, have been considered intractable, in papers as recent as 2019. In order to solve these problems, we consider application of homotopy methods as well as off-the-shelf optimization algorithms.

Homotopy methods have been studied by many researchers, for example, Watson [1986], Dunlavy and O'Leary [2005], and Mobahi and Fisher III [2015], and proven to be effective in solving many optimization problems. From a broader point of view, homotopy is a subfield in mathematical topology, and as a principle, homotopy refers to "continuous transformation" between two functions. In the context of optimization, homotopy methods transform a "hard to solve" problem into a related but "easy" problem with a known solution and some desired properties. The "easy" problem is then transformed back into the original problem, through a series of iterations, in which the intermediary problems are solved at each iteration and the obtained solution is used as the starting point for the next

iteration. Using this procedure we avoid dealing directly with the "hard" optimization problem.

In general, homotopy methods can be considered a relatively young field in mathematical optimization. Despite their general effectiveness, these methods need to be specifically developed and tailored for individual problems, and as we illustrate here, there are still many prominent problems in computer science that can benefit from using these methods.

Our main research focus is on deep learning models. Application of these models have become wide spread among researchers and practitioners. Despite their capabilities in achieving high accuracies, deep learning models have been criticized for their lack of easy interpretation, which undermines confidence in their use for important applications. We study this problem and introduce a novel technique, interpreting a trained neural network by investigating its *flip points*. A flip point is any point that lies on the boundary between two output classes: e.g. for a neural network with a binary yes/no output, a flip point is any input that generates equal scores for "yes" and "no". So far, finding exact points on the decision boundaries of trained deep models has been considered an intractable optimization problem.

To prepare to solve this optimization problem, we formulate our neural network as a function in Chapter 2. We investigate its computational properties with respect to optimization. We study the Lipschitz continuity of the model, derive a bound on its Lipschitz constant, investigate its derivatives and their rank, and use a tunable activation function so that we have control over the derivatives of the neural network function.

In Chapter 3, we propose a practical method to find exact points on the deci-

sion boundaries of these models. The flip point closest to a given input is of particular importance, and this point is the solution to a well-posed optimization problem. This optimization problem incorporates the neural network function, and it is nonlinear, nonconvex, and usually high dimensional. It also has highly nonlinear equality constraints involving the output of the neural network function. The neural network function itself has the issue of vanishing and exploding gradients besides the previous issues. These all make the optimization problem rather hard to solve.

Therefore, in Chapter 3, we also develop a homotopy algorithm to solve this problem more effectively. This algorithms relies on the neural network function formulated in Chapter 2. Our algorithm transforms the network via a homotopy, in order to overcome the issue of vanishing and exploding gradients, and to ensure that we have a feasible starting point. The homotopy transformation is gradually backtracked until we find the closest flip point for the original network. We find this algorithm to be quite reliable.

With regard to interpretation of deep learning models, we provide an overview of the uses of flip points in Chapter 4. Through results on standard datasets, we demonstrate how flip points can be used to provide detailed interpretation of the output produced by a neural network. Moreover, for a given input, flip points enable us to measure confidence in the correctness of outputs much more effectively than softmax score. They also identify influential features of the inputs, identify bias, and find changes in the input that change the output of the model. We show that distance between an input and the closest flip point identifies the most influential points in the training data. Using principal component analysis (PCA) and pivoted QR factorization, the set of directions from each training input to its closest flip point provides explanations of how a trained neural network processes

an entire dataset: what features are most important for classification into a given class, which features are most responsible for particular misclassifications, how an adversary might fool the network, etc. Although we investigate flip points for neural networks, their usefulness is actually model-agnostic for models with continuous output.

Studying the flip points and the decision boundaries of deep learning models have far reaching implications in other areas of research, besides the interpretation, as we study in Chapter 5. For example, training, generalization error, and robustness to adversarial attacks are all areas of research about deep learning models that speculate about the decision boundaries, and sometimes make simplifying assumptions about them. A trained model is defined by its decision boundaries, and therefore, studying the decision boundaries is a natural and direct approach to study the models, despite the computational difficulties.

In Chapter 5, we provide mathematical tools to investigate the surfaces that define the decision boundaries. Through numerical results, we demonstrate these techniques and show them more accurate than previous results that rely on simplifying assumptions such as local linearity. We show that the complexities of decision boundaries can make linear approximation methods quite unreliable for models with nonlinear activation functions. Instead, flip points provide better estimates of distance and direction from data points to decision boundaries. We also study decision boundaries in relation to adversarial robustness, and show that computing flip points can reveal the weakest vulnerabilities of models towards adversarial attacks. We show that computing the closest flip point can be done at a cost similar to that of computing an adversarial point using the loss function, the common approach in the literature. We also study the shape and connectedness of sub-manifolds that define the decision regions of trained networks.

The success of deep learning models is partly attributed to the years of work on designing and hand-crafting specific network structures that can effectively learn from the data and generalize well on unseen data. Although many researchers choose to work on predesigned networks for standard datasets, it is a rather hard task to design a network from scratch to learn an unfamiliar dataset. There are plenty of approaches in the literature that can prune a network that already achieves high accuracy. But designing a network that can achieve high accuracy requires considerable human cost and computational power. Some of these model design methods rely on training many many networks and choosing the best model; therefore, they are not practical for many applications. We propose practical and inexpensive algorithms based on matrix conditioning, in order to effectively design feed-forward networks from scratch. As we show in Chapter 6, our methods lead to very compact networks with high accuracies.

We also study interpretation of an unsupervised learning method, the Gaussian graphical model, in Chapter 7. These models have many applications for real world problems in understanding the underlying relationship among features in the data. The output of this learning method, the graphical model, may vary significantly with changes in the data. However, the relationship between the data itself and the obtained model was not clear. The data, on the other hand, is not usually identically and independently distributed, and existence of noise and corrupt data is relatively common. This undermines the confidence in the correctness of results, and may cause practitioners to be hesitant in using these models, if they do not have insight on how the edges in the graph are related to the data, and how robust the obtained graph is. Here, we study this problem, provide a computational method that can interpret individual edges in Gaussian graphical

models, and show the robustness of the overall graph. We achieve this by formulating these questions as optimization problems, providing a formulation that can compute the derivatives analytically, and solving the optimization problems effectively, using tailored algorithms.

Finally, in Chapter 8 we summarize our results and outline directions for future research.

Chapter 2: Neural Networks as Functions

2.1 Introduction

In this chapter, we formulate a standard neural network model for our experiments, and examine its computational properties as a function.

Computational properties of neural network functions have been investigated in many studies, from different perspectives. Neural networks can be described as composition of many, many functions. It is well known and proved that neural networks are able to approximate any measurable function to any desired degree of accuracy; in other words, they can be seen as universal approximators [Cybenko, 1989, Hornik et al., 1989, Mhaskar et al., 2016, Mhaskar, 1993, Strang, 2019, Zhou, 2019]. There are practical difficulties, however, in creating a neural network that achieves small error. There have been many studies that aim to address these difficulties and to make the composition of networks more efficient. For example, Shaham et al. [2018] used wavelet decomposition of functions to approximate them with bounded error. The size of network in their study depends on the dimension and curvature of the manifold, the complexity of the function, and the ambient dimension. Hanin [2017] studied the required depth of networks to approximate functions, when the networks have bounded width. Blcskei et al. [2019] studied approximability of arbitrary function classes in L_2. Petersen and Voigtlaender [2018] studied approximability of piecewise smooth functions, and Opschoor et al. [2019] studied approximability of deep

networks in relation to high order finite element methods.

In this thesis, we use deep networks not specifically to approximate functions, but in a standard supervised machine learning setting, where the network is first trained on a training set and then used as a function to perform the same task on unseen data. We consider a variety of tasks and datasets, including image classification, medical decisions, and financial risk assessment.

A neural network trained to perform such tasks is a rather complex function that is hard to understand. Despite the unprecedented success of neural networks in the past decade in achieving high accuracy in machine learning problems, they have become too complex to be investigated or audited by writing out the functions they compute. There is not adequate understanding about how their output is related to their input. Therefore, it is difficult to provide any explanation on why a certain output is produced. Consequently, we do not know how confident we can be in the accuracy of an output. This becomes problematic when the output of the network is about vital decisions in human lives, such as medical decisions, problems in criminal justice, or even problems as simple as classifying an image.

In our research, we use trained neural networks as functions, and we define optimization problems that incorporate such functions in their objective function and/or their constraints. By solving those optimization problems, we interpret trained neural networks, audit and debug them, investigate their decision boundaries, etc.

2.2 Neural network functions and the difficulties that arise

The difficulty in using a trained neural network as a function can be explained via its output surface and the derivatives of output with respect to its input. A trained neural

network is a nonlinear and nonconvex function. Moreover, the output of a trained neural network can be constant over vast regions of its domain, and it can be very volatile and/or steep in other regions. This can be explained through the hierarchy of the neural network which can cause the gradients to vanish and/or explode through its layers. Regardless of the explanation, the gradient matrix can be badly scaled and potentially uninformative or misleading. Solving an ill-conditioned optimization problem involving such functions is hard, and in high dimensional space, it can become an intractable task.

This issue with the gradients is commonly encountered in the training process, too, and in the neural network literature it is referred to as the issue of "vanishing and exploding gradients" [Bengio et al., 1994, Hanin, 2018]. It is important to make the distinction that in this research, we are concerned with the gradient of the output of the network with respect to its input, while in the training process one would be concerned with the gradient of the loss function with respect to the training parameters. In both cases, the "vanishing and exploding gradients" phenomenon can be studied by investigating individual matrices in the chain rule formulation of the gradient matrix.

We will examine the mathematics of these difficulties in more detail, in the next chapter. In the meantime, keep in mind that to overcome these issues, it would be desirable to formulate our network such that we have control over the gradients of the output of the network with respect to input.

In the following section, we first formulate a feed-forward neural network with a tunable activation function, continuous and easily differentiable. The tunable activation function gives us control over the curvature of the neural network function. We later use this for homotopy transformation of the network and interpretation of the model.

2.3 Formulating the neural network

2.3.1 Notation

In our notation, vectors and scalars are in lower case and matrices are in upper case, except for Kronecker delta. Bold characters are used for vectors and matrices, and the relevant level on the network is shown as a superscript in parenthesis. Subscripts denote the size of vectors and matrices, and the index for a particular element of a matrix or vector is shown inside brackets.

$\odot$: Hadamard product

erf() : error function, defined by equation (2.2)

softmax() : softmax function, defined by equation (2.1)

Jac($\boldsymbol{a}, \boldsymbol{b}$) : Jacobian matrix of vector $\boldsymbol{a}$ with respect to vector $\boldsymbol{b}$

$\sigma[i]$: tunable parameter for layer i (element i from vector of parameters $\boldsymbol{\sigma}_{1,m-1}$)

$\boldsymbol{\delta}_{m,n}$: Kronecker delta matrix with m rows and n columns

$\mathbb{1}_{m,n}$: matrix of ones with m rows and n columns

$\boldsymbol{b}^{(i)}$: bias vector for neurons on layer i

m : number of layers in the network excluding the input layer

n_i : number of neurons on layer i, $= |y^{(i)}|$

$\boldsymbol{W}^{(i)}$: weights of edges connecting neurons on layer $i-1$ to layer i

$\boldsymbol{x}$: inputs, vector with n_x elements

$\boldsymbol{y}^{(i)}$: output of layer i

$\boldsymbol{z}$: output of network, vector with n_m elements

2.3.2 Our neural network

In this thesis, we consider a feed-forward neural network prototype, $\mathcal{N}$, shown in Figure 2.1, as an example. Our methods can be easily generalized to neural networks with different architectures, such as convolutional and residual networks.

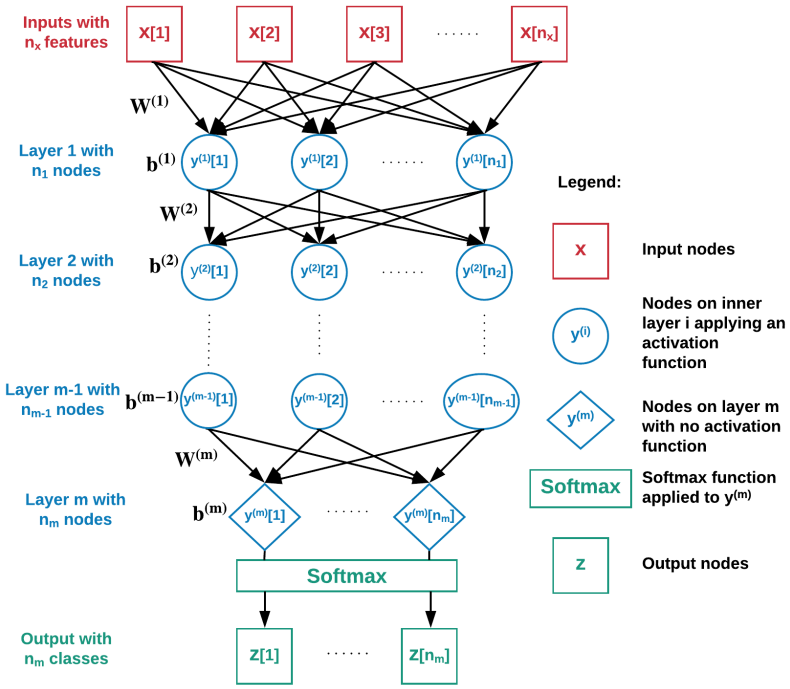

Figure 2.1: Sketch of a prototype feed-forward neural network $\mathcal{N}$ with n_x inputs, m layers, and n_m outputs.

The network in Figure 2.1 operates on an input vector $\boldsymbol{x}$ which has n_x features to produce an output $\mathcal{N}(\boldsymbol{x})$. We refer to the number of neurons on each layer as n_i, where i is the number of the hidden layer. The input to the first layer is $\boldsymbol{x}\,\boldsymbol{W}^{(1)} + \boldsymbol{b}^{(1)}$, where $\boldsymbol{W}^{(1)}$ is the weight matrix and $\boldsymbol{b}^{(1)}$ is the bias vector. Then the activation function is applied to obtain the output $\boldsymbol{y}^{(1)}$ for the first layer of network. This first layer output is a row vector with n_1 elements, corresponding to each neuron on the layer.

This process continues down the layers of the network until we reach the layer m.

For all the layers above layer m, an activation function, discussed in Section 2.3.3, is applied to obtain the output. But for the very last layer m, instead of the activation function, we apply the softmax function, defined by:

$$\text{softmax}(\boldsymbol{y})_{1,n_i} = e^{\boldsymbol{y}_{1,n_i}} / \left(e^{\boldsymbol{y}_{1,n_i}} \ \mathbb{1}_{n_i,1}\right). \tag{2.1}$$

to normalize the output of the network. The output of the softmax function on layer m has n_m elements.

2.3.3 Activation function

There are many activation functions that are commonly used in neural networks, such as the Rectified Linear Unit (ReLU), sigmoid, and hyperbolic tangent, to name a few [Strang, 2019]. The ReLU activation function has achieved great success in practice and its use is wide-spread [Jarrett et al., 2009, Nair and Hinton, 2010]. Nevertheless, there are recent studies that have effectively used other activation functions or combinations of them [Du and Lee, 2018, Gulcehre et al., 2016, Ramachandran et al., 2018].

In this thesis, we use the error function (erf) in a tunable manner, as in

$$activation(c|\sigma) = \text{erf}(\frac{c}{\sigma}) = \frac{1}{\sqrt{\pi}} \int_{-\frac{c}{\sigma}}^{+\frac{c}{\sigma}} e^{-t^2} dt, \tag{2.2}$$

where c is the result of applying the weights and bias to the neuron's inputs. The tuning parameter σ is strictly positive. We choose a single parameter σ_i for layer i and optimize it during the training process. Hence, for the whole network, we have a vector of $m-1$ tuning parameters, $\boldsymbol{\sigma}$, where each element corresponds to one hidden layer in the network. It is possible to allow more parameters in σ.

While erf is not a very common choice for activation function, it has been shown that its performance in terms of accuracy is comparable to other activation functions

[Ramachandran et al., 2018]. Mobahi [2016] has also reported success in using the erf for training recurrent neural networks.

We note that when σ is small, then the activation function resembles a step function, while when σ is large, it resembles a linear function, as shown in Figure 2.2, so erf captures the behavior of popular activation functions while preserving differentiability. For example, for the domain shown in Figure 2.2, when $\sigma = 20$ the activation function is computationally linear, when $\sigma = 1.0$ activation function resembles a sigmoid function, and when $\sigma = 0.1$ our activation function becomes similar to a step function.

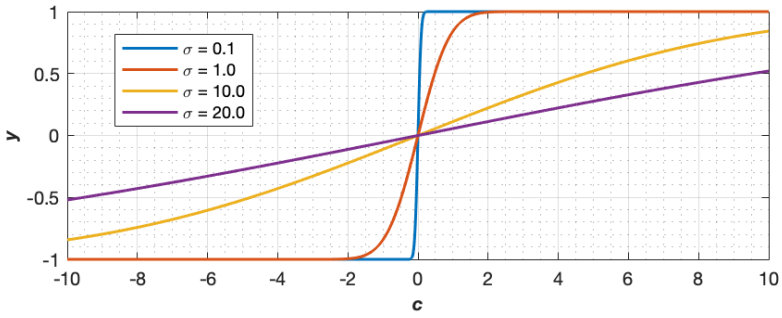

Figure 2.2: Shape of erf function as σ varies.

As we will explain in the next chapter, tuning the activation functions allows us to upper- and lower-bound the gradients of the output of the network with respect to its input. This enables us to solve optimization problems incorporating the original trained network in its objective function and/or its constraints.

2.3.4 Formulae

Using the erf activation function, the output of the first hidden layer for input $\boldsymbol{x}$ is

$$\boldsymbol{y}^{(1)}_{1,n_1} = \text{erf}\left(\frac{\boldsymbol{x}_{1,n_x}\,\boldsymbol{W}^{(1)}_{n_x,n_1} + \boldsymbol{b}^{(1)}_{1,n_1}}{\sigma[1]}\right). \tag{2.3}$$

And recursively, the output of hidden layer i can be written in terms of the output

of layer $i-1$, as

$$\boldsymbol{y}^{(i)}_{1,n_i} = \text{erf}\left(\frac{\boldsymbol{y}^{(i-1)}_{1,n_{i-1}}\, \boldsymbol{W}^{(i)}_{n_{i-1},n_i} + \boldsymbol{b}^{(i)}_{1,n_i}}{\sigma[i]}\right). \tag{2.4}$$

Finally for a neural network that has $m-1$ hidden layers, the output of the last layer is

$$\boldsymbol{y}^{(m)} = \boldsymbol{y}^{(m-1)}_{1,n_{m-1}}\, \boldsymbol{W}^{(m)}_{n_{m-1},n_m} + \boldsymbol{b}^{(m)}_{1,n_m}, \tag{2.5}$$

and the output of the network is

$$\boldsymbol{z}_{1,n_m} = \mathcal{N}(\boldsymbol{x}) = \text{softmax}\left(\boldsymbol{y}^{(m)}\right). \tag{2.6}$$

The output obtained from equation (2.6) is continuous between 0 and 1 for each neuron and the sum of values obtained on the output layer add to 1. The softmax function is commonly used in deep learning models for classification, prediction, and decision making problems. Each neuron on the output layer, for example, may represent a class, and the output for a neuron can be interpreted as the probability for the input to be in that class. As can be noted in the equations (2.5) and (2.6), the softmax function replaces the erf activation function on layer m. This setting is common for deep learning models with softmax on their output layer but can be modified as desired and is not a limitation imposed on our formulation. In the next sections, we calculate the derivatives throughout the network.

2.4 Derivatives of the output with respect to input

We are interested in finding the derivatives of the output with respect to the inputs, in order to solve optimization problems incorporating the neural network function, $\mathcal{N}$. The derivative of the outputs of the network with respect to its inputs is a Jacobian matrix, $\text{Jac}(\boldsymbol{z}, \boldsymbol{x})$, when the network has more than one input feature and more than one

output class. We compute the derivatives analytically, which is generally more efficient and reliable than using finite differences.

Analytic computation of the derivatives is possible for many different kinds of network architectures, including feed-forward, convolutional, and residual networks, assuming that the network does not contain non-differentiable elements such as non-differentiable activation functions or max pooling. Even in the presence of non-differentiable elements in the network, in most cases, we can still rely on analytical computation of sub-gradients, which is more efficient than the finite difference option. This is similar to the approaches used by common software when computing the gradients of the loss function with respect to the trainable parameters.

For the feed-forward networks used in our work, the computation of the derivatives is analogous to the back-propagation approach commonly used to compute the gradients with respect to the training parameters of the networks [Rumelhart et al., 1988]. Hence, we calculate the derivatives, layer by layer, and use the chain rule. We first take the derivative of the output of the first layer with respect to the inputs:

$$\text{Jac}\left(\boldsymbol{y}^{(1)}, \boldsymbol{x}\right)_{n_1,n_x} = \frac{2}{\sigma[1]\sqrt{\pi}}\left(\mathbb{1}_{n_x,1}\; e^{-\left(\frac{\boldsymbol{x}\,\boldsymbol{W}^{(1)}+\boldsymbol{b}^{(1)}}{\sigma[1]}\right)^2_{1,n_1}}\right)^T \odot {\boldsymbol{W}^{(1)}_{n_x,n_1}}^T. \tag{2.7}$$

In general, we can write the Jacobian for the output of any hidden layer in terms of the Jacobian for the layer above it. This recursive relation is expressed by

$$\text{Jac}\left(\boldsymbol{y}^{(i)}, \boldsymbol{x}\right)_{n_i,n_x} = \\ \frac{2}{\sigma[i]\sqrt{\pi}}\left(\mathbb{1}_{n_x,1}\; e^{-\left(\frac{\boldsymbol{y}^{(i-1)}\,\boldsymbol{W}^{(i)}+\boldsymbol{b}^{(i)}}{\sigma[i]}\right)^2_{1,n_i}}\right)^T \odot \left({\boldsymbol{W}^{(i)}_{n_{i-1},n_i}}^T\; \text{Jac}\left(\boldsymbol{y}^{(i-1)}, \boldsymbol{x}\right)_{n_{i-1},n_x}\right). \tag{2.8}$$

We continue this process until we reach the last output layer, which has a softmax function instead of the activation function. We use the chain rule to obtain the Jacobian

as:

$$\text{Jac}(\boldsymbol{z}, \boldsymbol{x})_{n_m, n_x} = \Big(\big(\boldsymbol{z}_{1,n_m}^T \ \mathbb{1}_{1,n_m} \big) \odot \big(\boldsymbol{\delta}_{n_m,n_m} - \mathbb{1}_{n_m,1} \ \boldsymbol{z}_{1,n_m} \big) \Big) \Big({\boldsymbol{W}^{(m)}_{n_{m-1},n_m}}^T \ \text{Jac} \left(\boldsymbol{y}^{(m-1)}, \boldsymbol{x} \right)_{n_{m-1},n_x} \Big). \quad (2.9)$$

Thus, for any given input $\boldsymbol{x}$, we can calculate the derivative (or in other words the sensitivity) of the output with respect to each element of the input.

Although the formulation presented here is for a feed-forward neural network, it is not limited to its architecture and can be easily extended to other architectures such as convolutional and/or residual neural networks. Using this formulation, in Chapter 3, we will write optimization problems that explain a trained neural network.

2.4.1 Rank of the derivatives of output

We know from linear algebra that the rank of product of two matrices is less than or equal to the rank of each individual matrix. We also know that the rank of any matrix of any size is at most equal to its smallest dimension. Now, let's evaluate the rank of the Jacobian we just computed.

Using the two rules above, we can conclude that

$$\text{rank}(\text{Jac}(\boldsymbol{z}, \boldsymbol{x})) \leq \min(n_x, n_1, n_2, \ldots, n_m), \quad (2.10)$$

which means the rank of the derivative of output of a network with respect to input is at most equal to $\min(n_x, n_1, n_2, \ldots, n_m)$.

Among neural network architectures, it is very common that $\min(n_x, n_1, n_2, \ldots, n_m) = n_m$. For auto-encoders, the layer with the smallest number of neurons will correspond to the "code" layer.

2.5 Derivatives of the loss function with respect to trainable parameters

Different functions can be used to evaluate the loss of neural network. Perhaps the most common loss function for classification models is the cross-entropy function as defined in Strang [2019]. The loss function, $\mathcal{L}$, will apply to the output of the neural network function $\mathcal{N}(\boldsymbol{x})$, for a particular input $\boldsymbol{x}$, and produce a scalar representing the loss for that input.

In general, the loss of the neural network over a set of inputs is the sum of losses for the individual inputs. Hence, the derivative of the loss for a set of inputs (e.g., training set) is the sum of the derivatives for the individual inputs.

For a particular input, $\boldsymbol{x}$, the derivative of the loss function with respect to the trainable parameters (e.g., weight matrices) can be computed using the chain rule

$$\nabla(\mathcal{L}(\mathcal{N}(\boldsymbol{x})), \boldsymbol{W}^i) = \nabla(\mathcal{L}(\mathcal{N}(\boldsymbol{x})), \mathcal{N}(\boldsymbol{x}))\ \mathrm{Jac}(\mathcal{N}(\boldsymbol{x}), \boldsymbol{W}^i). \tag{2.11}$$

For computing the $\mathrm{Jac}(\mathcal{N}(\boldsymbol{x}), \boldsymbol{W}^i)$, we use back-propagation as explained previously. To perform this computation, we use the TensorFlow software [Abadi et al., 2015] developed by Google.

2.6 Lipschitz continuity of the neural network function

Finally, we investigate the Lipschitz continuity of our neural network. Knowing the Lipschitz constant of a neural network allows us to use a line to approximate the output of a neural network between two points in its domain, with bounded approximation error. In later chapters we will discretize the domain of a trained network and use a line to approximate the output of network between the discretization points. Knowing the Lipschitz constant we can choose the discretization points such that our approximation

error is bounded and small.

To evaluate the Lipschitz constant one can take two approaches. The **first approach**, is to compute an upper bound on the norm of the derivatives of the output of the network. The **second approach**, is to estimate the local Lipschitz constant by searching in the neighborhood of $\boldsymbol{x}$.

For the first approach, we write the chain rule decomposition of equation (2.9) and compute the maximum norm of each matrix in the decomposition. The derivative will be bounded by the product of the norms. Consider equation (2.7) as the first step. We have

$$\begin{aligned} \| \operatorname{Jac}\left(\boldsymbol{y}^{(1)}, \boldsymbol{x}\right) \|_\infty &\leq \frac{2}{\sigma[1]\sqrt{\pi}} \left\| \left(\mathbb{1}_{n_x,1}\, e^{-\left(\frac{\boldsymbol{x}\,\boldsymbol{W}^{(1)}+\boldsymbol{b}^{(1)}}{\sigma[1]}\right)^2_{1,n_1}} \right) \odot \boldsymbol{W}^{(1)} \right\|_\infty \\ &\leq \frac{2}{\sigma[1]\sqrt{\pi}} \|\boldsymbol{W}^{(1)}\|_\infty. \end{aligned} \tag{2.12}$$

because $0 \leq \mathbb{1}_{n_x,1}\, e^{-\left(\frac{\boldsymbol{x}\,\boldsymbol{W}^{(1)}+\boldsymbol{b}^{(1)}}{\sigma[1]}\right)^2_{1,n_1}} \leq 1$.

Continuing this process down the layers of the network, using equations (2.8) and (2.9), we can easily bound the derivative of output. We note that for each hidden layer in the network, $\mathbb{1}_{n_x,1}\, e^{-\left(\frac{\boldsymbol{y}^{(i-1)}\,\boldsymbol{W}^{(i)}+\boldsymbol{b}^{(i)}}{\sigma[i]}\right)^2_{1,n_i}}$ is bounded between 0 and 1, and can be dropped.

We obtain the upper bound on the Lipschitz constant as

$$\| \operatorname{Jac}\left(\boldsymbol{z}, \boldsymbol{x}\right) \|_\infty \leq \frac{2^{m-1}}{\pi^{(m-1)/2} \prod_{i=1}^{m-1} \sigma[i]} \prod_{i=1}^{m} \|\boldsymbol{W}^{(i)}\|_\infty. \tag{2.13}$$

For the second approach, we can investigate the behavior of the Jacobian for a particular output class, or in the neighborhood of a particular $\boldsymbol{x}$. Our maximization problem here is non-convex and we cannot be certain that the maximum derivative we find is actually the "maximum". However, even if we find a local maximizer, as opposed to the global one, it might be insightful. This approach might be more plausible if we are interested in the neighborhood between two nearby inputs, for example the neighborhood between an input and the closest point to it on a decision boundary.

2.7 Summary

In this chapter, we formulated our neural network and investigated some of its computational properties as a function. We used an activation function that gives us control over the derivatives of the neural network function. We provided formulation for computing the derivatives of the output of network with respect to its input, and examined the Lipschitz continuity of the neural network function and the rank of its derivatives.

In the next chapter, we will define optimization problems that incorporate trained neural network functions and propose a systematic method to compute exact points on the decision boundaries of the models. We refer to such points as *flip points*, and we would be interested to find the closest flip point to any given input.

Chapter 3: Finding Closest Flip Points

3.1 Introduction

In this chapter, we introduce the concept of "closest flip point" and provide an algorithm to compute it.[1]. In later chapters, we describe in detail how closest flip points can be used as a tool to investigate and interpret neural network functions. As a brief introduction, for a particular trained model, the closest flip point to a given input $\boldsymbol{x}$ is the closest point on the decision boundary of the trained model. Computing such points is a systematic way to investigate the decision boundaries, especially nonlinear and high-dimensional boundaries. Although we define the closest flip points in the context of a neural network model, the concept is actually model agnostic and can be used for other types of models that have continuous output.

We first consider a neural network that has two output classes and then extend our work to neural networks with an arbitrary number of outputs, and neural networks with quantified outputs.

3.2 Neural networks with two outputs: a binary classification

Consider a neural network with two output nodes. For definiteness, let's refer to the output of the neural network as a prediction of "cancerous" or "noncancerous", but

[1] This work has been published in "Interpreting Neural Networks Using Flip Points" [Yousefzadeh and O'Leary, 2019a].

our methods are equally applicable to other types of output, such as decisions and classifications. As mentioned in the previous chapter, we assume that the network output $\boldsymbol{z} = \mathcal{N}(\boldsymbol{x})$ is normalized using softmax so that the two elements of the output sum to one. Since $z[1] + z[2] = 1$, we can specify the prediction by a single output: $z[1] > \frac{1}{2}$ is a prediction of "cancerous", and $z[1] < \frac{1}{2}$ is a prediction of "noncancerous". If $z[1] = \frac{1}{2}$, then the prediction is undefined.

Now, given a prediction $z[1] \neq \frac{1}{2}$ for a particular input $\boldsymbol{x}$, we want to investigate how changes in $\boldsymbol{x}$ can change the prediction, for example, from "cancerous" to "noncancerous". In particular, it would be very useful to find the *least change* in $\boldsymbol{x}$ that makes the prediction change.

Since the output of the neural network is continuous, $\boldsymbol{x}$ lies in a region of points whose output $z[1]$ is greater than $\frac{1}{2}$, and the boundary of this region is continuous. So what we really seek is a nearby point on that boundary, and we call points on the boundary *flip points*. So given $\boldsymbol{x}$ with $z[1] > \frac{1}{2}$, we seek a nearby point $\hat{\boldsymbol{x}}$ with $\hat{z}[1] = \frac{1}{2}$, where $\hat{\boldsymbol{z}} = \mathcal{N}(\hat{\boldsymbol{x}})$.[2]

The closest flip point $\hat{\mathbf{x}}^c$ is the solution to an optimization problem

$$\min_{\hat{\boldsymbol{x}}} \|\hat{\boldsymbol{x}} - \boldsymbol{x}\|, \tag{3.1}$$

where $\|.\|$ is a norm appropriate to the data. Our only constraint is

$$\hat{z}[1] = 1/2.$$

Specific problems might require additional constraints; e.g., if $\boldsymbol{x}$ is an image, upper and

[2] One technical point: Because z_1 is continuous, there will be a point arbitrarily close to $\hat{\boldsymbol{x}}$ for which z_1 is less than $1/2$ and the prediction becomes "noncancerous" *unless* $\hat{\boldsymbol{x}}$ is a local minimizer of the function z_1. In this extremely unlikely event, we will have the gradient $\bigtriangledown z_1(\hat{\boldsymbol{x}}) = \mathbf{0}$ and the second derivative matrix positive semidefinite, and $\hat{\boldsymbol{x}}$ will not be a boundary point. In practice, this is not likely to occur.

lower bounds might be imposed on $\hat{\boldsymbol{x}}$, and discrete inputs will require binary or integer constraints. It is possible that the solution $\hat{\mathbf{x}}^c$ is not unique, but the minimal distance is always unique.

3.3 Neural networks with multi-class outputs

For neural networks with multi-class outputs, we can use this same approach to define flip points between any pair of classes and to find the closest flip points for a given input. Suppose our neural network has n_z outputs and, for $\boldsymbol{x}$, $z[i]$ is the largest component of $\boldsymbol{z}$. If we want to find a flip point between classes i and j, then the objective function (3.1) remains the same, and the constraints become

$$\hat{z}[i] = \hat{z}[j],$$

and, for $k \neq i, j$,

$$\hat{z}[i] > \hat{z}[k].$$

Thus, for each individual input, we can compute $n_z - 1$ closest flip points $\hat{\mathbf{x}}^{c(i,j)}$ between the class for that input and each of the other classes.

3.4 Neural networks with a quantified output

Neural networks can also be used to specify a quantity. For example, a neural network can be trained to determine the appropriate dosage of a medicine. In such applications, flip points have a different meaning. For example, we can ask for the least change in the input that changes the dose by a given amount. Again, we can formulate and answer these questions as optimization problems.

In this thesis, we focus on neural networks with 2 or more output classes. Decision boundaries of trained networks in high dimensional space can be complex, and it can be

quite hard to investigate them as a whole. However, computing the closest flip point is an approach to investigate them systematically and to gain many insights about the trained networks, as we will explain in the next chapters. The rest of the current chapter will focus on their computation.

3.5 Computing the closest flip point

Now that we have formulated the optimization problems to compute the closest flip point, we can attempt to solve the formulation.

3.5.1 Characteristics of the problem

Our optimization problem is nonconvex, so we cannot be sure that optimization algorithms will find the global minimizer. One important fact that makes the optimization easier is that we have a good starting point, the data point itself.

Besides the nonconvexity, the problem has nonlinear equality constraints on the output elements of the neural network function, $\mathcal{N}$. And the gradients of the neural network can explode or vanish because $\mathcal{N}$ can be very steep or constant over large regions.

3.5.2 General approach

If the activation function is differentiable (e.g., erf), we can make use of its gradient in solving the optimization problems we have introduced. Otherwise, subgradients can be used, but this can make the optimization algorithms more costly.

Using the gradients, we minimize (3.1) subject to the constraints mentioned in previous section, in order to find the closest flip point. In the case of inputs with discrete features, we can add the discrete constraints to the problem or add regularization terms

to the objective function using the techniques described by Nocedal and Wright [2006].

Our optimization problem can be considered a generally solvable problem using off-the-shelf methods available in the literature. However, difficulties sometimes arise in solving nonlinear non-convex optimization problems, and therefore it is beneficial to design an optimization method tailored to our particular problem.

In our numerical results, we have solved our optimization problem using the applicable algorithms in 3 packages, NLopt [Johnson, 2014], IPOPT [Wächter and Biegler, 2006], and Optimization Toolbox of MATLAB, as well as our own custom-designed homotopy algorithm.

For neural networks of small size, with only two output classes, all algorithms almost always converge to the same point; occasionally, the interior point algorithms find closer flip points. For networks of larger size with multi-class outputs and/or discrete features, our homotopy algorithm sometimes finds better solutions. However, for the majority of data points, all algorithms find the same closest flip point. The variety and abundance of global and local optimization algorithms in the above optimization packages give us confidence that we have indeed usually found the closest flip point. In any case, we demonstrate in Chapter 5 that our flip points are closer than those estimated by methods such as linear approximations.

We use a tunable error function as the activation function. This allows us to introduce nonlinearity into the model while having control over the magnitude of the derivatives. Keep in mind that one can compute flip points for trained models and interpret them regardless of the architecture of the model (number of layers, activation function, etc.), the training set, and the training regime (regularization, etc.).

3.5.3 Homotopy algorithm for computing the closest flip points

Here, we explain the framework of our homotopy algorithm for computing the closest flip points in the context of our network. Our method can be easily generalized to neural networks with different architectures, such as convolutional and residual networks. The homotopy algorithm applies an optimization module to a series of networks.

3.5.3.1 Optimization module

We define the numerical process of computing the closest flip point $\hat{\mathbf{x}}^c$ to an input $\boldsymbol{x}$ between classes i and j by the function $\mathcal{F}$:

$$\hat{\mathbf{x}}^c{}_{(i,j)} = \mathcal{F}(\boldsymbol{x}, \mathcal{N}, \mathbf{x_0}, \mathcal{C}, i, j).$$

We assume $\mathcal{F}$ is a standard off-the-shelf optimizer. The inputs to $\mathcal{F}$ include the trained neural network $\mathcal{N}$, the starting point $\mathbf{x_0}$, and the constraints $\mathcal{C}$. As a general practice and based on our numerical experiments, an interior-point algorithm can be considered a good choice, as it is known to be successful in solving constrained, nonlinear, non-convex optimization problems with high dimensional variables [Nocedal and Wright, 2006].

Ideally, $\mathcal{F}$ efficiently finds the closest flip point for our network, possibly using the input $\boldsymbol{x}$ as the starting point. If this fails, then we use a *homotopy method*, starting by applying $\mathcal{F}$ to an easier network and gradually transforming it to the desired network, each time using the previously determined flip point as our starting point for $\mathcal{F}$. We now discuss the family of networks used in the homotopy.

3.5.3.2 Homotopy method

Our homotopy method, defined by Algorithm 1, begins with a neural network for which $\boldsymbol{x}$ is a flip point, and then computes flip points for a series of networks, gradually

transforming to the original network, using the closest flip point found at each iteration as the starting point for the next iteration. This way, the algorithm follows a path of flip points starting from $\boldsymbol{x}$, until it finds the closest flip point to $\boldsymbol{x}$ for the original network.

Algorithm 1 Homotopy algorithm for calculating closest flip point

Inputs: $\mathcal{N}$, $\boldsymbol{x}$, η, τ, $\mathcal{C}$, i, j

Output: Closest flip point to $\boldsymbol{x}$

1: Compute $\boldsymbol{\sigma}^h$ and $\boldsymbol{b}^{h(m)}$ using Algorithm 2 with inputs ($\mathcal{N}$, $\boldsymbol{x}$, τ, i, j)
2: $\hat{\mathbf{x}}^{c,0} = \boldsymbol{x}$
3: **for** $k = 1$ to η **do**
4: $\quad \boldsymbol{\sigma}^k = \boldsymbol{\sigma}^h + k(\frac{\boldsymbol{\sigma}^{\mathcal{N}} - \boldsymbol{\sigma}^h}{\eta})$
5: $\quad \boldsymbol{b}^{k(m)} = \boldsymbol{b}^{h(m)} + k(\frac{\boldsymbol{b}^{\mathcal{N}(m)} - \boldsymbol{b}^{h(m)}}{\eta})$
6: $\quad$ Replace $\boldsymbol{\sigma}^k$ and $\boldsymbol{b}^{k(m)}$ in $\mathcal{N}$, to obtain $\mathcal{N}^k$
7: $\quad \hat{\mathbf{x}}^{c,k} = \mathcal{F}(\boldsymbol{x}, \mathcal{N}^k, \hat{\mathbf{x}}^{c,k-1}, \mathcal{C}, i, j)$
8: **end for**
9: **return** $\hat{\mathbf{x}}^{c,\eta}$ as the closest flip point to $\boldsymbol{x}$

The initial neural network used in the algorithm is the same as the original network except that it has parameters $\boldsymbol{\sigma}^h$ for the erf and $\boldsymbol{b}^{h(m)}$ for the bias on the last layer. These are computed in Algorithm 2, discussed below.

The parameter η defines the number of iterations that Algorithm 1 uses to transform the network back to its original form. A large η means that each neural network is a small change from the previous one, so the starting point is close to the solution. A small η means that only a few optimization problems are solved, but each starting point may be far from the solution. We want to perform enough iterations so that the global minimizer is found, but we also want to keep the computational cost low. We have achieved best results with η ranging between 1 and 10. Choosing $\eta = 1$ is equivalent to not using the homotopy algorithm and directly applying $\mathcal{F}$ to the original network with starting point $\boldsymbol{x}$.

The initial transformation of the network is performed by Algorithm 2, pursuing

Algorithm 2 Algorithm to transform the network for the Homotopy algorithm

Inputs: $\mathcal{N}$, $\boldsymbol{x}$, τ, i, j

Output: $\boldsymbol{\sigma}^h$ and $\boldsymbol{b}^{h(m)}$

1: $\gamma = \sqrt{log(\frac{2}{\tau\sqrt{\pi}})}$
2: $\boldsymbol{y}^{(0)} = \boldsymbol{x}$
3: **for** $k = 1$ to $m - 1$ **do**
4: $\quad \sigma_k^h = \max(\frac{2}{\sqrt{\pi}}, \frac{1}{\gamma}\,\|\boldsymbol{y}^{(k-1)}\boldsymbol{W}^{(k)} + \boldsymbol{b}^{(k)}\|_\infty)$
5: $\quad$ **if** $\sigma_k^h > \frac{2}{\tau\sqrt{\pi}}$ **then**
6: $\quad\quad \boldsymbol{c} = \boldsymbol{y}^{(k-1)}\boldsymbol{W}^{(k)} + \boldsymbol{b}^{(k)}$
7: $\quad\quad$ **for** $t = 1$ to n_k **do**
8: $\quad\quad\quad \sigma_{k,t}^h = \max(\frac{2}{\sqrt{\pi}}, \frac{1}{\gamma}\,c_t)$
9: $\quad\quad$ **end for**
10: $\quad$ **end if**
11: $\quad \boldsymbol{y}^{(k)} = \mathrm{erf}(\frac{\boldsymbol{y}^{(k-1)}\boldsymbol{W}^{(k)} + \boldsymbol{b}^{(k)}}{\sigma_k^h})$
12: **end for**
13: $\min\limits_{\boldsymbol{b}^{h(m)}} \|\boldsymbol{b}^{h(m)} - \boldsymbol{b}^{\mathcal{N}(m)}\|_2$, subject to:
$\quad$ (1) $\boldsymbol{y}^{(m)} = \boldsymbol{y}^{(m-1)}\boldsymbol{W}^{(\boldsymbol{m})} + \boldsymbol{b}^{h(m)}$,
$\quad$ (2) $y_i^{(m)} = y_j^{(m)}$,
$\quad$ (3) $\forall\ l\ \neq\ i, j \mid y_i^{(m)} > y_l^{(m)}$
14: **return** $\boldsymbol{\sigma}^h, \boldsymbol{b}^{h(m)}$

two goals: first, bounding the flow of gradients through the layers of the network by changing the value of tuning parameters (lines 1 through 12), and second, changing the bias parameters in the last layer of the network so that $\boldsymbol{x}$ is a flip point for the transformed network (line 13).

The tuning parameters for the original network are $\boldsymbol{\sigma}^{\mathcal{N}}$, and $\boldsymbol{\sigma}^h$ denotes the transformed parameters computed by Algorithm 2. Similarly, $\boldsymbol{b}^{\mathcal{N}(m)}$ and $\boldsymbol{b}^{h(m)}$ denote the original and transformed bias in the last layer of the network.

By changing $\boldsymbol{\sigma}^{\mathcal{N}}$ to $\boldsymbol{\sigma}^h$, we try to control the magnitudes of the gradients of output with respect to inputs. The hierarchy of neural networks can cause the gradients to vanish

and/or explode through its layers, which could lead to a badly scaled gradient matrix and eventually an ill-conditioned optimization problem, and we would like to avoid this.

To compute the $\boldsymbol{\sigma}^h$, we trace the $\boldsymbol{x}$ as it flows through the layers of the network. As the input reaches each hidden layer, before applying the activation function, we tune the corresponding element of $\boldsymbol{\sigma}^h$, so that the absolute values of the gradients of the output of each neuron, with respect to neuron's input, is greater than or equal to τ, and less than or equal to 1. In our numerical experiments, we have used different values of τ ranging between 10^{-5} and 10^{-9}.

In Algorithm 2, line 1 computes a scalar γ such that the derivative of the erf is equal to τ. Lines 3 through 12, tune the σ, layer by layer, starting from the first layer and ending at the last hidden layer. Line 4 bounds the individual gradient between τ and 1. Choosing the $\sigma_k^h > \frac{2}{\sqrt{\pi}}$ ensures the gradients of neurons are upper bounded by 1. This relationship can be easily derived by setting the maximum derivative of erf equal to τ.

Choosing $\sigma_k^h \geq \frac{1}{\gamma} \|\boldsymbol{y}^{(k-1)}\boldsymbol{W}^{(k)} + \boldsymbol{b}^{(k)}\|_\infty$ can potentially make the gradients of all the neurons in layer k lower bounded by τ. Sometimes, this might not be possible to achieve for all the neurons in a layer, if we obtain $\sigma_k^h > \frac{2}{\tau\sqrt{\pi}}$. In such situations, we calculate the σ_k^h separately for each neuron on that layer (lines 5 through 10), and use a non-uniform σ_k^h in the homotopy algorithm. Line 11 computes the output of each layer after the σ is tuned for that layer.

Since our activation function is erf, we can effectively control the gradients and make them bounded. The maximum gradient of erf is at point zero, and by moving away from zero, its gradient decreases monotonically, until it asymptotically reaches zero. This boundedness and the monotonicity of both the erf and its gradient are helpful features that we leverage in our homotopy method. When using activation functions other than

erf, we have to avoid exploding and vanishing gradients, depending on the properties of the activation function in use.

By changing $\boldsymbol{b}^{\mathcal{N}(m)}$ to $\boldsymbol{b}^{h(m)}$, computed at line 13 of Algorithm 2, the input $\boldsymbol{x}$ actually becomes a flip point for the transformed network. Having a starting point that is feasible with respect to flip point constraints considerably facilitates the optimization process. The optimization problem on line 13 of the algorithm is a convex quadratic programming problem and can be solved by standard algorithms.

3.5.4 Performance of homotopy algorithms in finding closest flip points

Here, we investigate the performance of our homotopy algorithm on a neural network trained on the Adult Income dataset which has a combination of continuous and discrete features. Details about this dataset is provided in Section 4.4.2.

To compare the quality of solution, let's define the distance ratio as the ratio of the closest flip point distance found by the other solvers, to the distance found by our homotopy algorithm. Figure 3.1 shows the distribution of distance ratios for 1,000 data points randomly chosen from the testing set for this dataset.

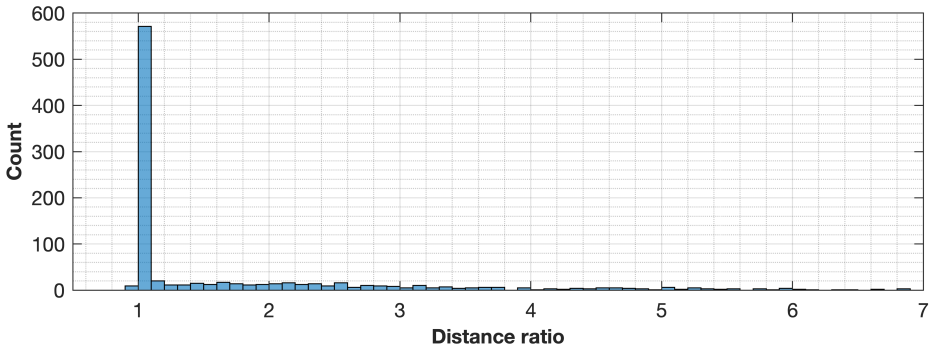

Figure 3.1: Homotopy algorithm finds closer flip points for the Adult Income dataset which has a combination of continuous and discrete features. The distance ratio is the ratio of distance found by the other algorithms to the distance found by the homotopy algorithm, which is shown for 1,000 data points randomly chosen.

As we can see, for nearly 60% of the data points, the distance ratio is very close to 1. In very rare occasions the distance ratio is smaller than 1, but it does not go below 0.95. The rest of the distance ratios are spread out beyond 1 and 7. For 1.4% of the data points, other algorithms do not yield a feasible flip point, while the homotopy algorithm always finds a feasible flip point. This clearly demonstrates the effectiveness of homotopy in finding closest flip points.

The average time spent to compute the closest flip points on a Macbook 2017 is 0.33 seconds for the closest distance found by other algorithms, while it is 8.98 seconds for the homotopy algorithm. Although the homotopy algorithm has taken longer, the quality of its solutions is much better, and we can conclude that spending the extra time is worth the reward of finding closer flip points.

For networks with continuous input space and binary outputs, for example the model used in Chapter 5, our homotopy algorithm finds flip points similar to the flip points found by the NLopt, IPOPT, and Matlab Optimization Toolbox.

Clearly, the performance of our homotopy algorithm may vary compared to other algorithms, for each dataset, in terms of the quality of solutions and the computation time. Therefore, the user should choose the best algorithm, case by case.

3.5.5 A note on cost

Computing the closest flip point requires computation of derivatives of the output of network with respect to inputs, at each iteration. For any given model, the cost of computing such derivatives is slightly less than computing the derivatives of loss function of neural network for *a single input*, with respect to the weight parameters of the first layer. The slight decrease in cost is because the loss function has one more layer of computation

after computing the output of a network. Note that in the training of a network we have to compute the derivatives of loss function, not only for the weight parameters of the first layer, but also for all the other layers; which involves one separate matrix multiplication per layer, and requires more memory, too. Overall, the derivative computation for finding the closest flip point is considerably less expensive than the derivative computation for training.

Additionally, for training a network, the number of training points is usually in the range of several thousands, sometimes millions. Hence, at each epoch of training, one has to compute such derivatives, thousands or even millions of times more, compared to each iteration of computing the closest flip point for an input. In fact, the cost of computing the derivatives for one iteration of training, with a mini-batch that contains a given number of points, dominates the cost of computing the derivatives for equivalent number of inputs in one iteration of flip point computation.

The number of iterations it takes to find the closest flip point would vary, depending mainly on the location of input and its distance from the closest decision boundary, the output surface of the network in that vicinity, and the optimization method. However, all those factors are present in the training of the network, as well, in a much more complex fashion, especially because the number of training parameters of a network are usually orders of magnitude larger than the number of input features.

This gives a clear idea how inexpensive computation of the closest flip point is, compared to the training process of the same network.

3.6 Summary

In this chapter, we introduced the concept of closest flip point as the point on the decision boundary of a trained model, closest to a given input. We formulated optimization problems to compute such points, and developed a homotopy algorithm that can compute them effectively.

In the next chapter, we will show how closest flip points can be used to interpret trained models, improve their training, and debug them.

Chapter 4: Interpreting and Debugging Neural Networks Using Flip Points

4.1 Introduction

In this chapter, we study the interpretation of neural networks and its implications for training and debugging.[1]

In real-world applications, neural networks are usually trained for a specific task and then used, for example to make decisions or to make predictions. Despite their unprecedented success in performing machine learning tasks accurately and fast, these trained models are often described as black-boxes because they are so complex that one cannot interpret their output in terms of their inputs.

When a trained network is used as a black-box, users cannot be sure how confident they can be in the correctness of each individual output. Furthermore, when an output is produced, it would be desirable to know the answer to questions such as, what changes in the input could have made the output different? A black-box cannot provide answers to such questions. This inexplainability becomes problematic in many ways, especially when the network is utilized in tasks consequential to human lives, such as in criminal justice, medicine, and business. Because of this, there have been calls for avoiding neural net-

[1] Part of this work has been published as "Interpreting Neural Networks Using Flip Points" [Yousefzadeh and O'Leary, 2019a] and part of it has been presented as "Debugging Trained Machine Learning Models Using Flip Points" at International Conference on Learning Representations (2019), Debugging Machine Learning Models Workshop [Yousefzadeh and O'Leary, 2019b].

works in high-stakes decision making [Rudin, 2018]. Alternatives include Markov decision processes [Lakkaraju and Rudin, 2017], scoring systems [Chen et al., 2018b, Rudin and Ustun, 2018], binary decision trees [Bertsimas and Dunn, 2017], and Bayesian rule sets [Wang et al., 2016].

There have been several approaches for interpreting neural networks and general black-box models. We mention here some of the papers representative of the field.

Some recent studies have tried to find the least changes in the input that can change the decision of the model. Spangher et al. [2018] have (independently) defined *a flip set* as the set of changes in the input that can flip the prediction of a classifier. Their algorithm applies to linear classifiers only. They use flip sets to explain the least changes in individual inputs but do not go further to interpret the overall behavior of the model or to debug it. Wachter et al. [2018] defined *counterfactuals* as the possible changes in the input that can produce a different output label and use them to explain the decision of a model. For a continuous model, the closest counterfactual is ill-defined, since there are points arbitrarily close to the decision boundaries, and the proposed algorithm uses enumeration, applicable only to a small number of features. Russell [2019] later suggested integer programming to solve such optimization problems, but the models used as examples are linear with small dimensionality.

Some studies have taken a model-agnostic approach to interpreting black box models such as neural networks. For example, the approach taken by Ribeiro et al. [2016] builds an explanation for an output via a linear model in the vicinity of a specific input. Similarly, Ribeiro et al. [2018] derive if-then rule explanations about the local behavior of black box models.

Methods based on perturbing each input feature individually have severe computa-

tional limitations. First, they can be prohibitively expensive when dealing with a complex high-dimensional nonlinear function such as that represented by a neural network. Second, the output of a neural network can be constant over vast areas of its domain, while it might be very volatile in other regions. Therefore, it can be hard to find a suitable vicinity that gives sensible results when perturbing high-dimensional inputs. Third, the features may have incompatible scalings, so determining meaningful perturbations is difficult. Finally, the features of the inputs can be highly correlated; therefore, perturbing the inputs one by one will be inefficient and possibly misleading. Koh and Liang [2017] have used influence functions to guide the perturbation and interpret black-box models with emphasis on finding the importance of individual points in the training data.

Pursuing the interpretation of neural networks from an adversarial point of view, Ghorbani et al. [2017] generate adversarial perturbations that produce perceptively indistinguishable inputs that are assigned the same label, yet have very different interpretations. They further show that interpretations based on exemplars (e.g. influence functions) are similarly susceptible to adversarial attack.

Another line of research focuses on performing insightful pre-processing to make the inputs to the neural network more interpretable. One promising approach uses prototypes to represent each output class [Chen et al., 2018a, Li et al., 2018, Snell et al., 2017]. Individual inputs are compared to the prototypes (e.g., by measuring the 2-norm distance between each input and all the prototypes), and that information is the input to the neural network. In the context of text analysis, Lei et al. [2016] has introduced a model that first specifies distributions over text fragments as candidate rationales and then uses the rationales to make predictions.

Taking a different approach, Lakkaraju et al. [2017] have used decision rules to

emulate a neural network in a subdomain of the inputs. Although the emulated model in their numerical example is interpretable, its outputs are different than the outputs of neural network for about 15% of the data.

Many alternative models such as decision trees and rule lists have been in competition and co-existence with neural networks for decades, but in many applications have not been very appealing with respect to accuracy, scalability, and complexity, particularly with high-dimensional data. Our goal is to improve the interpretability of neural networks and other black-box models so that in cases where they have computational or accuracy advantages over alternative models, they can be used without hesitation. Through the use of *flip points* we are able to make neural networks interpretable, improve their training, and indicate the reliability of the output classification.

4.2 How flip points provide valuable information to the user

4.2.1 Determine the least change in $\boldsymbol{x}$ that alters the prediction of the model

The vector $\hat{\mathbf{x}}^c - \boldsymbol{x}$ is an accurate and clear explanation of the minimum change in the input that can make the outcome different. This is insightful information that can be provided along with the output. For example, in a bond court, a judge could be told what changes in the features of a particular arrestee could produce a "detain" recommendation instead of a "release" recommendation.

4.2.2 Assess the trustworthiness of the classification for $\boldsymbol{x}$

In our numerical examples we show that the numerical value of the output of a neural network, when the last layer is defined by the softmax function, does not indicate how sure we should be of the correctness of the output. In fact, many mis-predictions

correspond to very high softmax values. This has been previously observed by Nguyen et al. [2015] and Guo et al. [2017]. Gal and Ghahramani [2016] propose using information from training using dropout to assess the uncertainty of predictions. Their method is restricted to this particular training method, does not provide the likely correct prediction, and is more expensive than the method we propose. Another approach, proposed by Guo et al. [2017] constructs a calibration model, trained separately on a validation set, and appends it as a post-processing component to the network. Also, Lakshminarayanan et al. [2017] used ensembles of neural networks, trained adversarially with pre-calculated scoring rules, in order to estimate the uncertainty in predictions. Using flip points to assess the trustworthiness of predictions is a novel idea that has certain advantages compared to other approaches in the literature, as we explain.

The distances of incorrectly classified points to their flip points tend to be very small compared to the distances for correct predictions, implying that closeness to a flip point is indicative of how sure we can be of the correctness of a prediction. Small distance to the closest flip point means that small perturbations in the input can change the prediction of the model, while large distance to the flip point means that a larger change is necessary. It is important, of course, that distance be measured in a meaningful way, with input features normalized and weighted in a way that emphasizes their importance. Furthermore, in multi-class predictions, our numerical results indicate that when the model makes a mistake, the class with the closest flip point is actually the correct class.

Using flip points can be viewed as a direct method to assess the trustworthiness of predictions, even when models are calibrated or trained adversarially. Therefore, flip point assessment is not necessarily in competition with other methods in the literature; rather it is a simple and straightforward method that can be used for any model. Flip

points also provide clear explanations for their assessment in terms of input features and can point out to the possible correct prediction when there is low confidence.

4.2.3 Identify uncertainty in the classification of $\boldsymbol{x}$

Often, some of the inputs to a neural network are measured quantities which have associated uncertainties. When the difference between $\boldsymbol{x}$ and its closest flip point is less than the uncertainty in the measurements, then the prediction made by the model is quite possibly incorrect, and this information should be communicated to the user.

4.2.4 Use PCA analysis of the flip points to gain insight about the dataset

Earlier, we discussed using the direction from a single data point to the closest flip point to provide sensitivity information. Using PCA analysis, we can extend this insight to an entire dataset or to subsets within a dataset,

We form a matrix with one row $\hat{\mathbf{x}}^c - \boldsymbol{x}$ for each data point. PCA analysis of this matrix identifies the most influential directions for flipping the outputs in the dataset and thus the most influential features, This procedure provides clear and accurate interpretations of the neural network model. One can use nonlinear PCA or auto-encoders to enhance this approach. Alternatively, for a given data point, PCA analysis of the directions from the data point to a collection of boundary points can give insight about the shape of the decision boundary.

4.3 How flip points can improve the training and security of the model

4.3.1 Identify the most and least influential points in the training data in order to reduce training time

Points that are correctly classified and far from their flip points have little influence on setting the decision boundaries for a neural network. Points that are close to their flip points are much more influential in defining the boundaries between the output classes. Therefore, in online learning and real-time applications, where we have to retrain a neural network using streaming data, we can retrain the network more quickly using only the influential data points, those with small distance from their flip points. As mentioned earlier, Koh and Liang [2017] use influence functions to relate individual predictions of a trained model to training points that are most influential for that prediction. They are not able to draw conclusions about the decision boundaries of the model because they use small perturbations of training data and local gradient information for the loss function, which can be misleading for nonlinear non-convex functions in high dimensional space. Our approach does not just rely on local information but it seeks the closest point that flips the decision of the network. Therefore, the insight we provide goes well beyond their method without adding prohibitive expense.

4.3.2 Identify out-of-distribution points in the data and investigate overfitting

Out-of-distribution points in the training set appear as incorrectly classified points with large distance to the closest flip point. Finding such points can identify errors in the input or subgroups in the data that do not have adequate representation in the training set (e.g., faces of people from a certain race in a facial recognition dataset [Buolamwini

and Gebru, 2018]). Additionally, after we compute the closest flip points for all the points in the training set, we can further cluster the flip points and study each cluster in relation to its nearby data points. This will potentially enable us to investigate whether the model has overfitted to the data points or not. We have not investigated these two opportunities in our numerical results, but believe that they are promising directions for study.

4.3.3 Generate synthetic data to improve accuracy and to shape the decision boundaries

We can add flip points to the training set as *synthetic data* to move the output boundaries of a neural network insightfully and effectively. Suppose that our trained neural network correctly classifies a training point $\boldsymbol{x}$ but that there is a nearby flip point $\hat{\mathbf{x}}^c$. We generate a synthetic data point by adding $\hat{\mathbf{x}}^c$ to the training set, using the same classification as that for $\boldsymbol{x}$. Retraining the network will then tend to push the classification boundary further away from $\boldsymbol{x}$. Similarly, if our trained neural network makes a mistake on a given training point $\boldsymbol{x}$, then we can add the flip point $\hat{\mathbf{x}}^c$ to the training set, giving it the same classification as $\boldsymbol{x}$. This reinforces the importance of the mistake and tends to correct it.

Using flip points to alter the decision boundaries can be performed not just to improve the accuracy of a model but also to change certain traits adopted by the trained network. For example, if a model is biased for or against certain features of the inputs, we could alter that bias using synthetic data. We will demonstrate this later in our numerical results on the Adult Income dataset. There are studies in the literature that have used synthetic data (but not flip points) to improve the accuracy, e.g., Jaderberg et al. [2014]. There is also a line of research that has used perturbations of the inputs in order to

make the trained models robust, e.g., Tsipras et al. [2019]. However, using *flip points* as *synthetic data* is novel and would benefit the studies on robustness of networks, too.

We can also alter the decision boundaries of a trained model by adding flip points with, for example, different gender or race, not labeled as a specific class, but labeled as a flip point (output 1/2) between two classes. This can reduce biases in the model.

4.3.4 Understand adversarial influence

Flip points also provide insight for anyone with adversarial intentions. First, these points can be used to understand and exploit possible flaws in a trained model. Second, adding flip points with incorrect labels to the training data will effectively distort the class boundaries in the trained model and can diminish its accuracy or bias its results. Our methods could be helpful in studying adversarial attacks such as the problems studied by Schmidt et al. [2018], Sinha et al. [2018], Madry et al. [2017], and Katz et al. [2017]. In our numerical results for the FICO dataset (Section 4.4.3), we show that redundant features in the inputs can make the model more vulnerable to adversarial inputs. Also, in Chapter 5, we show that closest flip points can reveal where the models are *most* vulnerable to adversarial inputs.

4.4 Numerical results

In our numerical results, we use feed-forward neural networks with 12 layers and softmax on the output layer. We use a tunable error function as the activation function and use Tensorflow for training the networks, with Adam optimizer and learning rate of 0.001. Keep in mind that one can compute the flip points for trained models and interpret them, regardless of the architecture of the model (number of layers, activation function,

etc.), the training set, and the training regime (regularization, etc.). When calculating flip points, we measure the distance in equation (3.1) using the 2-norm. Calculating the closest flip points is quite fast, under 1 second for the MNIST, CIFAR-10, FICO, and Wisconsin Breast Cancer datasets, using a 2017 MacBook. Calculating the closest flip point for the Adult Income dataset takes about 9 seconds, because it has both discrete and continuous variables. Characteristics of all models are presented in Appendix B.

4.4.1 Image classification

4.4.1.1 MNIST

The MNIST dataset has 10 output classes, the digits 0 through 9. We could use pixel data as input to the networks, but, for efficiency, we choose to represent each data point using the Haar wavelet basis. The 100 most significant wavelets are chosen by pivoted QR decomposition [Golub and Van Loan, 2012] (explained in Appendix C) of the matrix formed from the wavelet coefficients of all images in the training set. The wavelet transformation applies convolutions of various widths to the input data and the reduction applied by using pivoted QR decomposition leads to significant compression of the input data, from 784 features to 100, allowing us to use smaller networks. This idea, independent of flip points, is valuable whenever working with image data, as we explain further in Appendix A. Using pixel input instead of wavelet coefficients would yield interpretation traits similar to those that we present here.

We train two networks, NET1 and NET2, using half of the training data (30,000 images) for each. Table 4.1 shows the accuracy of each network in the 2-fold cross validation. Accuracy could be improved using techniques such as skip architecture, but these networks are adequate for our purposes.

Table 4.1: Classification accuracies for NET1 and NET2 trained on MNIST.

Trained network	Accuracy on 1st half of training set	Accuracy on 2nd half of training set	Accuracy on testing set
NET1	100%	97.62%	97.98%
NET2	97.56%	100%	97.64%

For each of the images in the training set, we calculate the flip points between the class predicted by the trained neural networks and each of the other 9 classes.

Flip points identify alternate classifications. Some images are misclassified and close to at least one flip point. For all of these points, the correct label is identified by the closest of the 9 flip points (or one of those tied for closest after rounding to 4 decimal digits). For example, the image shown in Figure 4.1, from the second half of the MNIST training set, is an "8" mistakenly classified as "3" by NET1 with softmax score of 98%. Its distances to the closest flip points are shown in Table 4.2. Assuming that we do not know the correct label for this image, we would report the label as "3", with the additional explanation that there is low confidence in this prediction (because of closeness to the flip point), and the correct label might be "8".

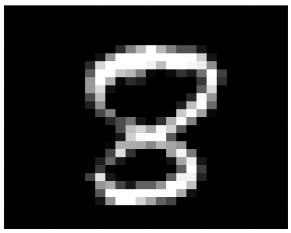

Figure 4.1: MNIST image mistakenly classified as "8" by NET1.

Flip points provide better measure of confidence than softmax. Many practitioners use the softmax output as a measure of confidence in the correctness of the output. As illustrated in Figure 4.2, the softmax scores range between 31% and 100% for

Table 4.2: Distance to closest flip points between class "8" and other classes, for image in Figure 4.1.

Class	0	1	2	4	5	6	7	8	9
Distance	1.27	1.32	0.58	2.16	0.56	1.45	1.51	**0.16**	0.90

the mistakes by NET1 and NET2, and range between 37% and 100% for correct classifications, providing no separation between the groups. If softmax were a good proxy for distance, then the data would lie close to a straight line. Instead, most of the mistakes have small distance but large softmax score: more than 73% of the mistakes have 0.8 or more softmax score. Hence, softmax cannot identify mistakes. Fortunately, the figure shows that the distance to the closest flip point is a much more reliable indicator of mistakes: mistakes almost always correspond to small distances. This is further demonstrated in Figure 4.3 which shows the distinct difference between the distribution of distances for the mistakes and the distribution of distances for the correct classifications.

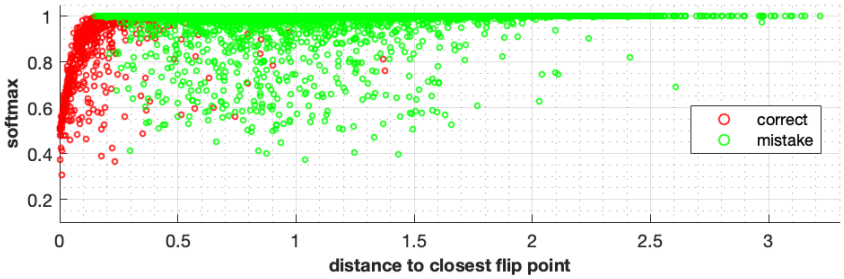

Figure 4.2: For the MNIST data, a large softmax score says nothing about the reliability of the classification. In contrast, distance to the closest flip point is a much more reliable indicator.

Flip points identify influential training points. Images that are correctly classified but are relatively close to a flip point are the most influential ones in the training process. To verify this, consider the first half of the MNIST training set, and order the

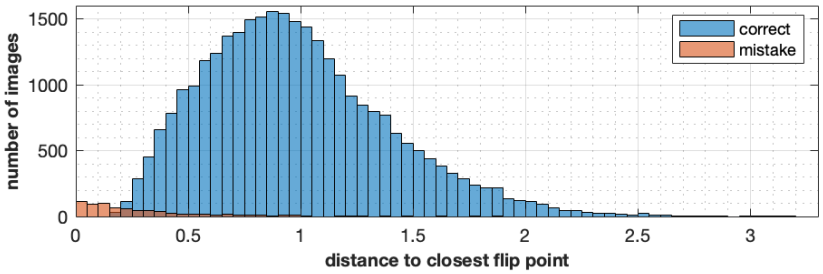

Figure 4.3: Distribution of distance to closest flip point among the images in the MNIST training set for mistakes (orange) and correctly classified points (blue).

images by their distances to their nearest $\hat{\mathbf{x}}^c$ for NET1. We then consider using neural networks trained using a subset of this data.

Data points at most 0.75 from a flip point form a subset of $9,463$ images, about 15% of the training set. A model trained on this subset achieves 97.9% accuracy on the testing set. Training with a subset of $9,463$ images randomly chosen from the training set on average (50 trials) achieves 96.2% accuracy on the testing set. A subset of same size from the images farthest from their flip points achieves only 90.6% accuracy on the testing set. These trends hold for all distance thresholds (Figure 4.4). This confirms that distance to the flip point is in fact related to influence in the training process.

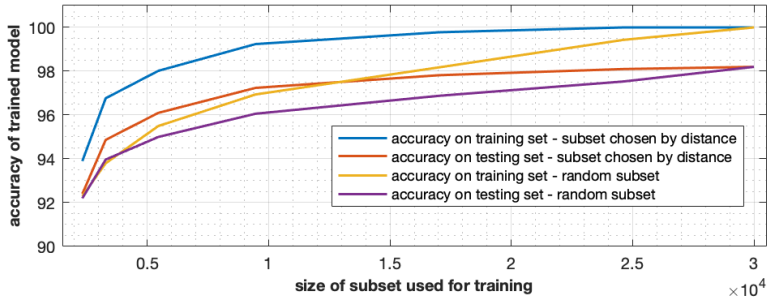

Figure 4.4: Accuracy of models trained on MNIST subsets.

We note that the model learns the entire training set with 100% accuracy when trained on about 16,000 images chosen by the distance measure. In contrast, it only achieves 98.80% accuracy when trained on a randomly chosen subset of the same size.

Also note that flip points are computed by solving a non-convex optimization problem, so we cannot guarantee that we have indeed found the *closest* flip point. Nevertheless, the computation seems to provide very *useful* flip points, validated by the small distances achieved by some flip points and by the results shown in Figures 4.2 – 4.4.

Flip points improve the training of the network. We append to the entire training set a flip point for each mistake in the training set, labeled with the correct label for the mistake. The resulting neural network achieves 100% accuracy on the appended training set and 98.6% on the testing set, an improvement over the 98.2% accuracy of the original network. This technique of appending synthetic images to the training set may be even more helpful for datasets with limited training data.

Comparison with alternative methods to interpret image classification models. Investigating the decision boundaries of a trained neural network using flip points, and especially, efficiently computing and interpreting the closest flip point to an input, are new contributions of this work. Some previous studies investigated decision boundaries of image classification models, but those methods relied on random perturbations of the inputs, which makes them prohibitively expensive and unlikely to find the closest boundary point. For example, Ribeiro et al. [2016] generates random perturbations of the image that would produce labels on both sides of the decision boundary. Ribeiro et al. [2018] reports difficulties with this approach; for example, finding a sensible amount of perturbation is challenging.

A recent line of research has investigated the decision boundaries of a trained network with respect to the generalization error. Elsayed et al. [2018] used a penalty term in the training process to indirectly increase the distance to the decision boundaries for all the training data, in order to improve the generalization error. Also, Jiang et al. [2019]

drew a regression between the distance to the decision boundaries and the generalization error of the model. However, both studies report the optimization problem to find the closest point on decision boundary intractable and instead use the first order derivatives to approximate the distance. Moreover, their focus is limited on the generalization error and do not attempt to interpret the model. Other studies focused on adversarial attacks, such as Ilyas et al. [2019], do not consider the decision boundaries, instead they aim to find inputs with softmax score close to 1, for the opposite label. Studying the decision boundaries of a trained neural network is a new contribution of this work, which has many implications worth further study, not just for interpretation of models, but also for studying generalization error and adversarial attacks.

4.4.1.2 CIFAR-10

We now consider two classes of airplanes and ships in the CIFAR-10 data set. This time we perform 3D wavelet decomposition on images using the Haar wavelet basis and use all of the wavelet coefficients to train a neural network, achieving 100% and 84.2% accuracy on the training and testing sets. We then calculate the flip points for all the images in both sets. Observations that we reported for MNIST apply here, too. So we focus our discussion on the directions to flip points and PCA analysis of them.

Figure 4.5 shows an image in the testing set that is mistakenly classified as an airplane, along with its closest flip point. We have computed the closest flip point in the wavelet space. It is interesting that the 1-norm distance between the image and its closest flip point in the pixel space is 210, and the differences are hard to detect by eye.

The matrix of directions between the misclassified images and their closest flip points is highly rank deficient. While we have 2,304 features for each image, the rank of directions

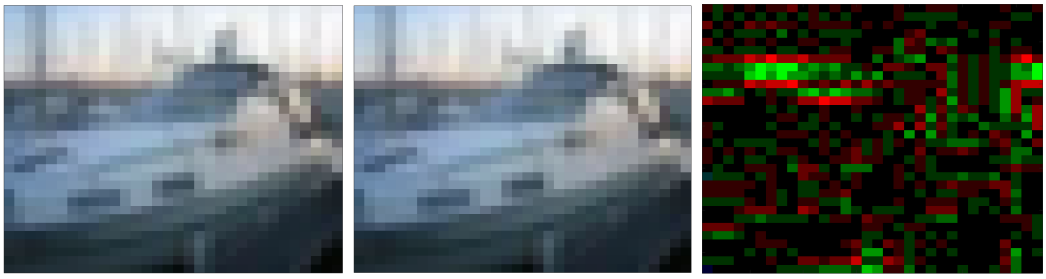

Figure 4.5: A ship image misclassified as airplane (left), its flip point (middle), and their 50X-magnified difference (right).

for flipping an airplane to a ship is 162, and it is 170 for flipping a ship to an airplane. Therefore, we can investigate the mistakes by looking at a very small subset of wavelet features out of the 2304 features.

Moreover, the matrix of directions that flip a misclassified ship to its correct class has 53% sparsity. The first principal component of the directions has the pattern shown in Figure 4.6.

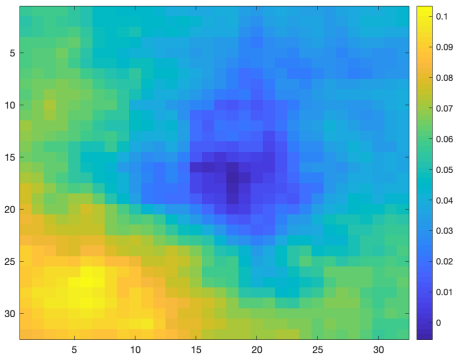

Figure 4.6: First principal component of directions that flip a misclassified ship to its correct class.

We threshold the principal coefficients in Figure 4.6, retaining pixels with coefficient greater than 0.05. Then we plot the corresponding pixels of the misclassified images of ships. Some of those images are plotted in Figure 4.7. One can see that for many of the mistakes, those pixels actually contain the prow of the ship in the image. This points to one vulnerability of our trained neural network, which we could then investigate further.

Moreover, this analysis reveals a bias in the training set, in terms of the orientation of ships pointing towards the bottom left corner. One could overcome such bias by augmenting the training set with the horizontally flipped version of images. Thus, information given by this PCA analysis gives valuable diagnostics to the designer of a neural network in order to identify possible biases adopted by the network.

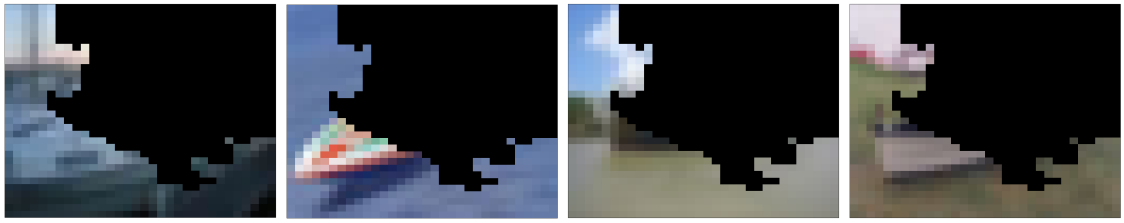

Figure 4.7: Pixels with large principal coefficients for misclassified ships.

When we repeat this for the misclassified airplanes, we observe that the center of image is most significant as in Figure 4.8, which is sensible since most of the misclassified planes are located at the center of image, with either vertical or horizontal orientation.

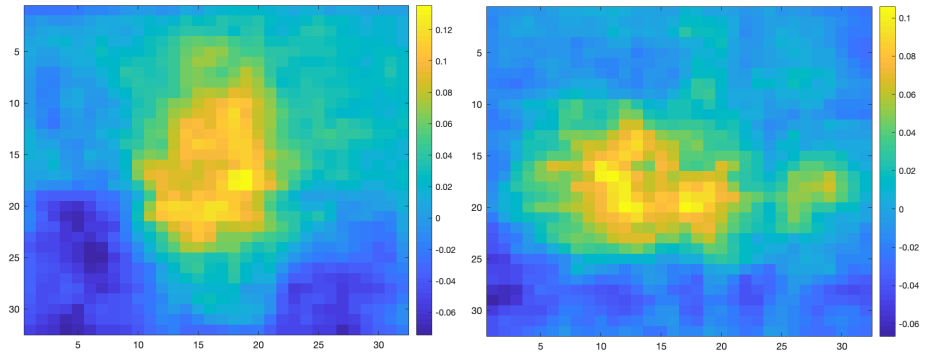

Figure 4.8: First (left) and second (right) principal component of directions that flip a misclassified airplane to its correct class.

Finally, we note that great similarity exists between the directions for the correct classifications in the training and testing sets. Investigating other principal components can provide additional insights.

4.4.2 Adult Income dataset

The Adult dataset from the UCI Machine Learning Repository [Dua and Graff, 2017] has a combination of discrete and continuous variables. Each of the 32,561 data points in the training set and 16,281 in the testing set are labeled, indicating whether the individual's income is greater than \$50K annually. We normalize each of the continuous variables (age, fnlwgt, education-num, capital-gain, capital-loss and hours-per-week) to the range 0 – 100 using upper bounds of 100, 2e6, 20, 2e5, 1e4 and 120, respectively, and we also use these ranges to constrain the search for flip points. We represent each of the category types for the variables workclass, education level, marital status, occupation, relationship, race, sex, and native country) as one binary feature. The categories that are active for a data point have binary value of 1 in their corresponding features, while the other features are set to zero. When searching for a flip point, we allow exactly one binary feature be equal to 1 for each of the categorical variables. Our trained neural network achieves accuracy of 87.3% and 86.1% on the training and testing sets. Our aim here is to show how a trained neural network can be interpreted, not to draw conclusions about the dataset itself. Clearly, alternate pre-processing of the data or an alternate distance measure would change the interpretation. Our preprocessing and scaling choices are suboptimal but illustrative; clearly, application scientists should always be involved in setting the distance metric in order to ensure meaningful results.

Flip points provide interpretations and can expose bias. As an example, consider the 53rd training data point, corresponding to a person with income greater than \$50K. From Table 4.4.2, we can see that the race of this individual is influential in the decision of our particular model, as are other features such as "working hours" and

"work class". These two latter features seem to have an obvious causal relationship with the income, but influence of race should be questioned. We can also constrain selected features when computing flip points. For example, we can ask for the closest flip point corresponding to a person with the same gender or race, or with a different gender or race. This enables us to investigate gender/racial bias in the output of the neural network.

Table 4.3: Difference in features for Adult dataset training point #53 and its closest flip point

Data	Input #53 in training set	Closest flip point
Capital-gain	0	625
Capital-loss	1,902	1,862
Hours-per-week	60	59.8
Race	White	Asian-Pac-Islander
Work class	Private	State-gov
Marital Status	Married-civ-spouse	Married-AF-spouse

Flip points reveal patterns in how the trained model treats the data. As an example, we consider the effect of gender (Male, Female) in connection with the family relationship (Wife, Own-child, Husband, Not-in-family, Other-relative, Unmarried) for individuals that have income "$\leq$ \$50K". For this model, 89% of data points in that income category have the same gender as their closest flip points, while 11% have switched from Female to Male, and 0.2% have switched from Male to Female. This shows that being Male is moderately helpful in being labeled "> \$50K" by the model. But, as we will see later, education is the most influential feature for flipping to the high income category.

For the same income category, we also observe that for 2.5% of the flip points, the family role switches from Husband to Wife, while a third of those have simultaneously switched from Female to Male. This reveals that the trained model considers both the family role of Wife and the gender of Male helpful for having high income. The switch

from family role of Wife to Husband is absolutely rare among the flip points.

PCA on the flip point directions identifies influential features. Consider the subset of directions that flip a "$\leq$ \$50K" income to "$>$ \$50K". The first principal component reveals that, for this neural network, the most prominent features with positive impact are having a master's degree, having capital-gains, and working in the private sector, while the features with most negative impact are having highest education of Preschool, working without-pay, and having capital-loss. Looking more deeply at the data, pivoted QR decomposition of the matrix of directions reveals that some features, such as having a Prof-school degree, have no impact on this flip.

PCA on the directions between the mistakes in the training set and their closest flip points shows that native country of United States has the largest coefficient in the first principal component, followed by being a wife and having capital-gain. The most significant features with negative coefficient are being a husband and native countries of Cambodia and Ireland. These features can be considered the most influential in confusing and de-confusing the neural network. PCA on the direction vectors explains how our neural network is influenced by various features. It thus enables us to calculate inputs that are mistakenly classified, for adversarial purposes.

Flip points can deal with flaws and can reshape the model. Here as an example, we try to change the behavior of the trained model towards the individuals with country of origin "Mexico". We consider all the data points with that country of origin that have a flip point with a different country. 82% of those points have income "$\leq$ \$50K". We generate closest flip points for all those inputs while constraining the country of origin to remain "Mexico". We then add each generated flip point to the training set, using the same label as the data point, and train a new model using the appended set. After

performing PCA analysis on the directions to the new flip points, we observe that Mexico does not appear in any of the first 10 principal components, whereas it had a large value in the first principal component obtained for the original model. The accuracy of the trained model has remained almost the same (slightly increased by 0.05%), confirming that we have achieved our goal. Using this kind of analysis, we can reshape the behavior of the model as needed.

Comparison with other interpretation approaches. Our use of flip points for interpretation and debugging is more comprehensive than existing methods in the literature. For example, Spangher et al. [2018] computes flip sets only for linear classifiers and does not use them to explain the overall behavior of the model, identify influential features, or debug. LIME [Ribeiro et al., 2016] and Anchors [Ribeiro et al., 2018] rely on sampling around an input in order to investigate decision boundaries, inefficient and less accurate than our approach, and the authors do not propose using their results as we do. The extent and accuracy of interpretation we provide for neural networks are comparable and in some aspects surpass the interpretation provided in the literature for simple models. For example, the model suggested by Chen et al. [2018b] for the FICO Explainable ML Challenge, reports the most influential features in decision making of their model, similar to our findings in the next section; and investigates the overall behavior of the model, similar to our results for the Adult dataset. But we are able to find the least changes that can flip the decision of the model for individual inputs and study the decision boundaries to identify and reduce vulnerabilities.

4.4.3 FICO explainable machine learning challenge

This dataset FICO [2018], provided by the Fair Isaac Corporation (FICO), has 10,459 observations with 23 features, and each data point is labeled as "Good" or "Bad" risk. We randomly pick 20% of the data as the testing set and keep the rest as the training set. We regard all features as continuous, since even "months" can be measured that way.

Eliminating redundant features. The condition number of the matrix formed from the training set is 653. Pivoted QR factorization [Golub and Van Loan, 2012], finds that features "MSinceMostRecentTradeOpen", "NumTrades90Ever2DerogPubRec", and "NumInqLast6Mexcl7days" are the most dependent columns; discarding them leads to a training set with condition number 59. Using the data with 20 features, we train a neural network with 5 layers, achieving 72.90% accuracy on the testing set. A similar network trained with all 23 features achieved 70.79% accuracy, confirming the effectiveness of our decision to discard three features. The 72.90% accuracy is a considerable improvement over the accuracy of 69.69% reported by Chen et al. [2018b] for neural networks trained on this dataset.

Interpreting individual outputs. As an example, consider the first datapoint, corresponding to a person with "Bad" risk performance. Table 4.4 shows the change between the data point and its closest flip point, for 5 features. The change in other features is close to zero.

Identifying influential features. Using pivoted QR on the matrix of directions between datapoints labeled "Bad" and their flip points, the three most influential features are "AverageMInFile", "NumInqLast6M", and "NumBank2NatlTradesWHighUtilization". Similarly for the directions that flip a "Good" to a "Bad", the three most influential fea-

Table 4.4: Difference in features for FICO dataset point #1 and its closest flip point

Data	Input #1	Closest flip point (relaxed)	Closest flip point (integer)
AverageMInFile	84	105.6	111.2
NumSatisfactoryTrades	20	24.1	24
MSinceMostRecentDelq	2	0.6	0
NumTradesOpeninLast12M	1	1.7	2
NetFractionRevolvingBurden	33	19.4	8.5

tures are "AverageMInFile", "NumInqLast6M", and "NetFractionRevolvingBurden". In both cases, "ExternalRiskEstimate" has no influence.

We perform PCA analysis on the subset of directions that flip a "Bad" to "Good" risk performance. The first principal component reveals that, for this neural network, the most prominent features with positive impact are "PercentTradesNeverDelq" and "PercentTradesWBalance", while the features with most negative impact are "MaxDelqEver" and "MSinceMostRecentDelq". These conclusions are similar to the influential features reported by Chen et al. [2018b], however, our method provides more detailed insights.

Studying redundant variables and their effect on behavior of model and generalization error. Interestingly, for the model trained on all 23 features, the most significant features in flipping its decisions are "MSinceMostRecentTradeOpen", "NumTrades90Ever2DerogPubRec" and "NumInqLast6Mexcl7days", exactly the three dependent features that we discarded. This reveals an important vulnerability of machine learning models regarding their training sets. When dependent features are included in the training set, it might not affect the accuracy on the training set, but it adversely affects the generalization error. Additionally, the decision of the trained model is more susceptible

to changes in the dependent features, compared to changes in the independent features. One can argue that the dependent features are confusing the trained model.

Revealing patterns in directions to the closest flip points. Figure 4.9 shows the directions to the closest flip points for features "NumInqLast6M" and "NetFractionRevolvingBurden". Directions are distinctly clustered for flipping a "Bad" label to "Good" and vice versa.

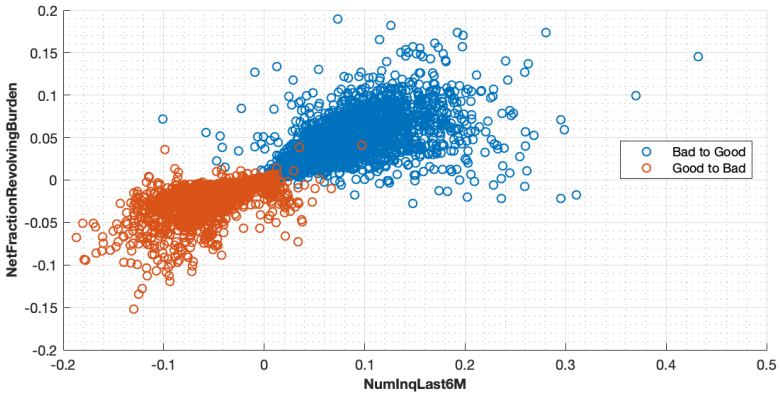

Figure 4.9: Directions between the inputs and their closest flip point for two influential features.

Furthermore, Figure 4.10 shows the directions in coordinates of the first two principal components. We can see that the directions are clearly clustered into two convex cones, exactly in opposite directions. Also, we see misclassified inputs are relatively close to their inputs while correct predictions can be close or far. Overall, misclassified inputs have similar patterns compared to correct classifications, which explains why the model cannot distinguish them from each other.

4.4.4 Default of credit card clients

This dataset from the UCI Machine Learning Repository [Dua and Graff, 2017] has 30,000 observations, 24 features, and a binary label indicating whether the person will

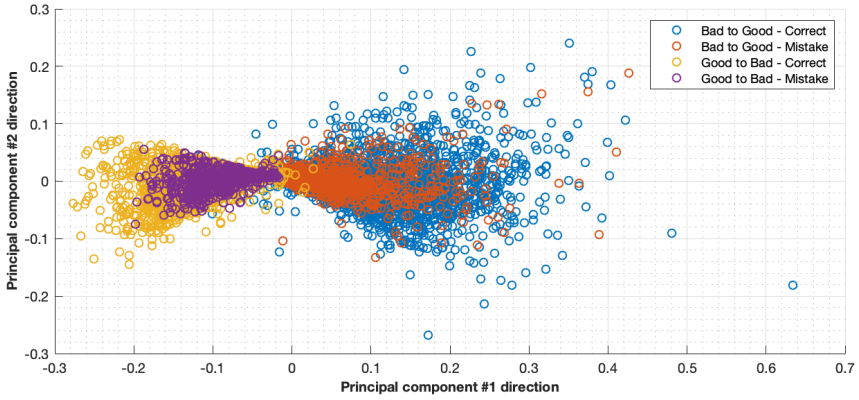

Figure 4.10: Change between the inputs and their flip points in the first two principal components

default on the next payment or not.

We binarize the categorical variables "Gender", "Education", and "Marital status". The condition number of the training set is 129 which implies linear independence of features. Using a 10-fold cross validation on the data, we train a neural network with 5 layers, to achieve accuracy of 81.8% on the testing set. When calculating the closest flip points, we require the categorical variables to remain discrete.

Identifying influence of features. We perform pivoted QR decomposition on the directions to the flip points. The results show that "BILL-AMT3" and "BILL-AMT5" are the most influential features, and "Age" has the least influence in the predictions. In fact, there is no significant change between the age of all the inputs and their closest flip points.

Revealing patterns in how the trained model treats the data. We briefly make some observations about the overall behavior of the trained model. The influence of gender is not significant in the decisions of the model, as only about 0.5% of inputs have a different gender than their flip points. However, we observe that those changes are not gender neutral. We see that for flipping a "no default" to "default", changing the gender

from "Female" to "Male" has occurred 5 times more often than the opposite. Similarly, for flipping a "default" to "no default", gender has changed from "Female" to "Male", 5 times more often than the opposite. We also observe that changing the marital status from "Married" to "Single" is helpful in flipping "no default" predictions to "default". This kind of analysis can be performed for all the features, in more detail.

Flip points can deal with flaws and can reshape the model. Similar to the study by Spangher et al. [2018], we train a model with a subset of the training set, where young individuals are under-sampled. In both our training and testing sets, about 52% of individuals have age less than 35. We keep the testing set as before, but remove 70% of the young individuals from the training set. After training a new model, we obtain 80.83% accuracy on the original testing set. We also observe that the "Age" is the 3^{rd} most influential feature in flipping its decisions. Moreover, PCA analysis shows that having less Age has a negative impact on the "no default" prediction and vice versa.

We consider all the data points in the training set labelled as "default" that have closest flip point with older age, and all the points labelled "no default" that have closest flip point with younger age. We add all those flip points to the training set, with the same label as their corresponding data point, and train a new model using the appended training set. Investigating the behavior of the new model reveals that Age has become the 11^{th} influential feature and it is no longer significant in the first principal component of directions to flip points; hence, the bias against Age has been reduced.

Adding synthetic data to the training set has great potential to change the behavior of model, but we cannot rule out unintended consequences. Therefore, it is important to interpret the overall behavior of the reshaped model with respect to all features, and ensure that it behaves as we expect. By investigating the influential features and PCA

analysis, we see that the model has been altered only with respect to the Age feature, and the overall behavior of model has not changed.

4.4.5 Wisconsin breast cancer dataset

Neural networks have shown promising results in identifying cancer [Agrawal and Agrawal, 2015]. As a simple example, we use the Wisconsin breast cancer database from the UCI repository which has 30 features extracted from digitized images of fine needle aspirate of 569 breast masses. We divide standard error features by their corresponding mean feature, and then normalize the mean and worst features between 0 and 1. The label is binary: "malignant" or "benign".

We randomly divide the dataset into a training set and testing set, consisting of 80% and 20% of data respectively. We achieve 100% and 94.7% accuracy on the training and testing sets, respectively. The average distance to the closest flip point is 0.022 for the mistakes in the testing set and 0.103 for the correct classifications in the testing set. The average distance is 0.106 for correct classifications in the training set, very similar to the average distance in testing set. All of the mistakes have softmax score of at least 97.4%. In fact, the average softmax for all the correct and wrong classifications are both more than 99%. Again, the distance to the closest flip point is a reliable measure to identify classifications that are possibly wrong, while softmax score is not.

Flip points can be used to improve the model. What features in the input are most important? As an example, consider the first data point which is classified correctly as "malignant" by the trained neural network. Its closest flip point differs mostly in features "standard error of texture" and "standard error of fractal dimension".

We perform PCA on the matrix of directions between each "benign" input and its

closest flip point, and look at the first principal component. The most prominent features that can flip the decision of the network to "malignant" are "standard error of radius" and "standard error of texture". Similarly, the most prominent features to flip a "malignant" decision to "benign" are "standard error of texture" and "worst area".

A clinician can use this information to validate the trained neural network as a computational tool. The information also enables the designer of the neural network to work with a clinician to rescale the data to emphasize features believed to be over- or under-emphasized by the current model and to provide better classifications.

4.5 Summary

1. We studied the problem of model interpretation, using flip points to investigate the boundaries between output classes. We defined and solved optimization problems to find the closest flip point to a given input, providing accurate explanations about changes in the input that can flip the output from one class to another.

2. The distance of an input to the closest flip point proved to be a very effective measure of the confidence we should have in the correctness of the output, much more reliable than softmax score, and should be interpreted using a practitioner's knowledge of uncertainty in the data. The distance also enabled us to identify most/least influential points in the training data.

3. PCA analysis identified the most influential features in the inputs. Also, for each output class, PCA identified the directions and magnitudes of change in each of the features that can change the output.

4. Adding flip points as synthetic data boosted the accuracy of a neural network, but

we demonstrated that it can also be used adversarially.

5. The distance and direction to the nearest flip point, coupled with a practitioner's knowledge of the measurement uncertainty in each of the features, can provide insight into whether the classification is unique or ambiguous.

Flip points exist for any model, not just neural networks and can provide insight, debugging, and interpretability.

In the next chapter, we will focus more specifically on the decision boundaries of trained networks, question common simplifying assumptions about them in the literature, compare our methods with approximation methods, and provide mathematical tools to study the decision boundaries, systematically.

Chapter 5: Shape of the Decision Boundaries of Neural Networks

5.1 Introduction

In this chapter, we study the decision boundaries of trained neural networks and investigate some of the common simplifying assumptions about them in the literature.

Interpreting the behavior of trained neural networks, their generalization error, and robustness to adversarial attacks are open research problems that all deal, directly or indirectly with the decision boundaries of these models. The decision boundaries of neural networks have typically been investigated through simplifying assumptions or approximation methods. As we will show in our numerical results, many of these simplifications may lead to unreliable results. We also show that some of the speculations about the decision boundaries are accurate and some of the computational methods can be improved. We advocate for verification of simplifying assumptions and approximation methods, wherever they are used. Finally, we demonstrate that the computational practices used for finding adversarial examples can be improved, and computing the closest point on the decision boundary reveals the weakest vulnerability of a model against adversarial attack.

Some previous work has highlighted the importance of boundary points. Studies such as Lippmann [1987] investigate the decision boundaries of single-layer perceptrons, while describing the difficulties that arise regarding the complexity of decision boundaries for multi-layer networks.

As mentioned in Chapter 4, some studies on interpretation of deep models have made simplifying assumptions about the decision boundaries. For example, Ribeiro et al. [2016] assumes that the decision boundaries are locally linear. Their approach tries to sample points on two sides of a decision boundary, then perform a linear regression to approximate the decision boundary and explain the behavior of the model. However, as we show in our numerical results, decision boundaries of neural networks can be highly nonlinear, even locally, and a linear regression can lead to unreliable explanations.

Regarding the generalization error of trained models, Elsayed et al. [2018] and Jiang et al. [2019] have shown there is a relationship between the closeness of training points to the decision boundaries and the generalization error of a model. However, they regard computing the distance to the decision boundary as an intractable problem and instead, use the derivatives of the output to derive an approximation to the closest distance. In our numerical results, we compare their approximation to our results, and show the advantages of computing the distance directly. Other studies such as Neyshabur et al. [2017], approximate the closest distance to the decision boundary by the closest distance to an input with another label, which is an overestimate. They use the modified margin between softmax outputs as a measure of distance, which we see later can be misleading.

Regarding adversarial attacks, there are many studies that seek small perturbations in an input that can change the classification of the model. For example, Fawzi et al. [2017], Jetley et al. [2018], Moosavi-Dezfooli et al. [2016] apply small perturbations to the input until its classification changes, but, since they do not attempt to find the closest point on the decision boundaries of the model, they do not reveal its weakest vulnerabilities. Most recent studies on adversarial examples, such as Tsipras et al. [2019] and Ilyas et al. [2019] minimize the loss function of the neural network for the adversarial label, subject to a

distance constraint. They impose the distance constraint in order to find an adversarial example similar to the original image. Although this method is an important tool, this form of seeking adversarial examples has certain limitations, regarding the ability to make the models robust, and regarding the measurement of robustness of models, as we explain through numerical examples. We show that finding the closest point on the decision boundary accurately represents the least perturbation needed for adversarial classification, and, therefore, studies on adversarial examples can benefit from direct investigation of decision boundaries.

5.2 Numerical experiment setup

To illustrate our ideas, we use a 12-layer feed-forward neural network trained on 2 classes of the CIFAR-10 dataset, ships and planes. To train the network, we have used Tensorflow, with Adam optimizer, learning rate of 0.001, and Dropout with rate 50%. Further information about the model is provided in Appendix B.

Inputs to our network are not the pixels, but 200 of the 3D Daubechies-1 wavelet coefficients. We choose the 200 coefficients according to the pivoted QR factorization of the wavelet coefficients for the training set. Using the most significant wavelet coefficients removes redundancies in the features of the image as we explain in Appendix A.

The accuracy we obtain on the testing set is 84.05%. This can be improved to near 95% using the calculated flip points as new training points in order to move decision boundaries, or by using more wavelet coefficients.

In our computations, we verify that each computed flip point is a legitimate image, satisfying appropriate upper and lower bounds for each pixel. Also, we measure the distance using the ℓ_2 norm.

5.3 Investigating the neural network function and the closest flip points

Here, viewing the trained neural network as a function, we investigate the paths between inputs and flip points.

5.3.1 Lipschitz continuity of the output of trained model

The output of our neural network is a smooth mathematical function. Because it is the composition of a finite set of Lipschitz continuous functions, the output is also Lipschitz continuous. The Lipschitz constant is bounded as we showed in Chapter 2.

Why does this matter? As we walk along a line connecting one data point to another, the Lipschitz constant can tell us how fine we should discretize that path in order to accurately depict the output of network and identify the locations of decision boundaries. This means that we choose the distance between the discretization points small enough such that the output of network can be considered to change linearly between any consecutive points, with negligible error.

5.3.2 What flip points reveal about decision boundaries

Here, we draw lines between images, discretize those lines, and plot the output of the network along them. Consider two images, $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$, separated by distance $d = \|\boldsymbol{x}_1 - \boldsymbol{x}_2\|_2$. The points on the line connecting them may be defined by $(1-\alpha)\boldsymbol{x}_1 + \alpha\boldsymbol{x}_2$ where α is a scalar between 0 and 1. This line can be extended beyond $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ on either side by choosing $\alpha < 0$ or $\alpha > 1$, respectively.

In Figure 5.1, zero on the horizontal axis corresponds to image $\boldsymbol{x}_1$ and the right boundary corresponds to $\boldsymbol{x}_2$, an image chosen from the same or other class. The lines connecting most pairs of images in the data set resemble the top left plot in Figure 5.1 in

their simplicity; both images are far from the decision boundary, and the line between them crosses the decision boundary once. The other five plots in this figure are hand picked to demonstrate atypical cases. Having multiple boundary crossings is more frequent among the images in the testing set, compared to images in the training set.

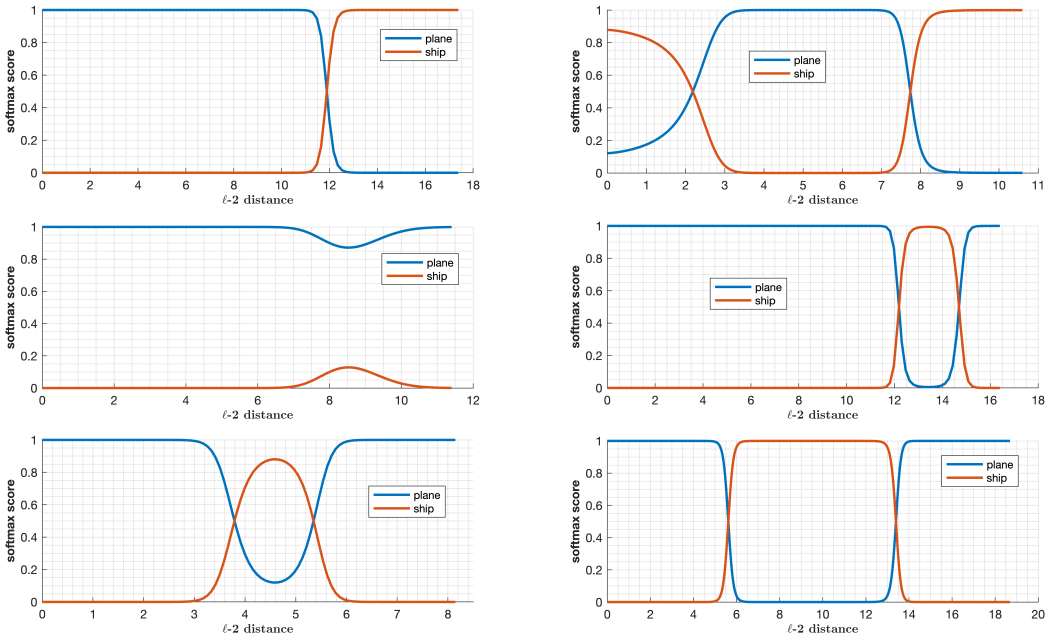

Figure 5.1: Model output along the line connecting two images.

Figure 5.2 shows the output of the model for some lines connecting images to their closest flip points. Notice that the two bottom plots have a much smaller distance scale, and the behavior of the softmax score for correctly and incorrectly classified points is quite similar. These plots clearly show that the decision boundaries in our model are far from linear and a hyperplane would not be able to approximate such boundary surfaces. Fawzi et al. [2018] also have the view that the decision boundaries of deep models are highly curved, but they had not computed exact points on the decision boundaries. Our results confirm their view.

Figure 5.3 considers lines connecting various pairs of flip points. If decision bound-

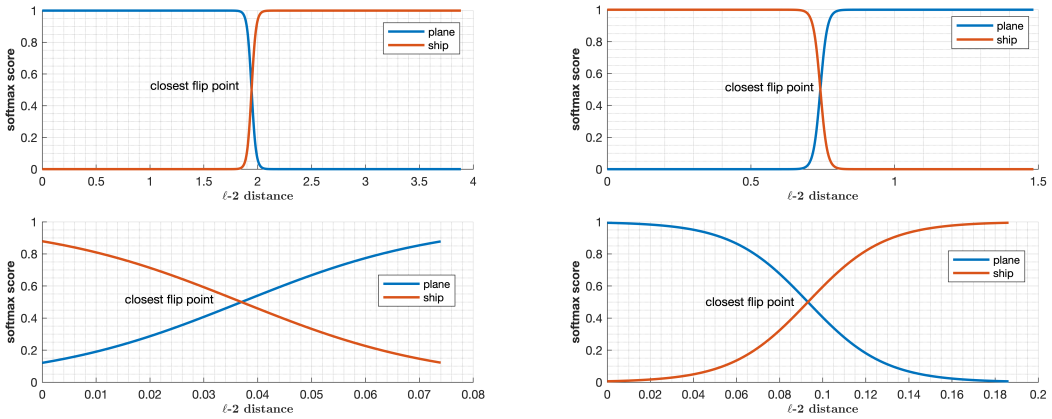

Figure 5.2: Model output along the line connecting an image with its closest flip point. Images for the top row are correctly classified, while images for the bottom row are misclassified.

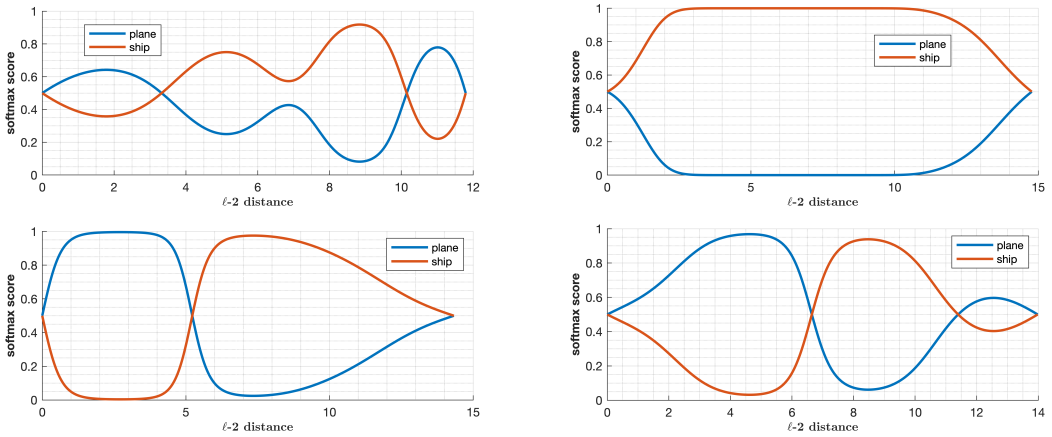

Figure 5.3: Model output along the line connecting two flip points.

aries were linear, we would expect the red and blue curves to have softmax scores of 0.5 all along these lines, and that is certainly not what the plots show. If decision boundaries were convex/concave, then we would expect behavior such as that in the upper right plot, but the other three plots show that the true behavior of the decision boundaries is much more complicated.

5.4 Comparing with approximation methods

Here, we compare our calculated minimum distance to the decision boundaries with approximation methods in the literature. We also compare the direction to the closest point on decision boundary with that predicted by first order derivatives. In both comparisons we observe that relying on approximation methods may be misleading.

Regarding the minimum distance to the decision boundaries, Elsayed et al. [2018] suggested estimating the distance using a approximation method based on first order Taylor expansion, building on other suggestions for linear approximation of the distance, e.g., Matyasko and Chau [2017] and Hein and Andriushchenko [2017]. The approximation method of Elsayed et al. [2018] has also been used by Jiang et al. [2019] to study the generalization error of models. Figure 5.4 shows the distances computed using their approximation method versus the actual distances we have computed using flip points. For distances less than 0.01, the Taylor approximation underestimates the distance by about a factor of 2. For larger distances, the Taylor approximation underestimates by as much as a factor of 20 or more, as shown in Figure 5.5. We note that their approximation method estimates distance to decision boundaries without finding actual points on the decision boundaries. We find that the estimated distances are generally underestimates of both the true distance in that direction and the true distance to a flip point.

Figure 5.6 illustrates the distance to the decision boundary along the direction defined by the Taylor series approximation, compared to the distance to the closest flip point. Finding the Taylor direction and then finding the intersection with the decision boundary (a one-dimensional optimization problem) are both very inexpensive operations, and our results indicate that this approach usually gives a good approximation to the

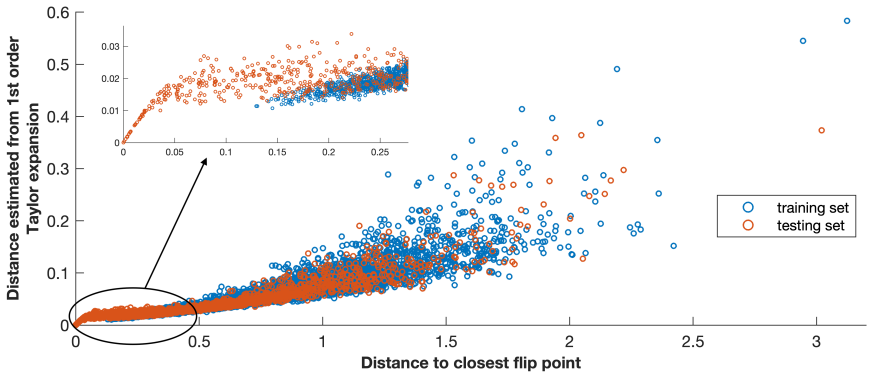

Figure 5.4: Using the first-order Taylor expansion for estimating the minimum distance to decision boundaries siginificantly underestimates the distance, except when points are very close to the decision boundaries (closer than 0.01).

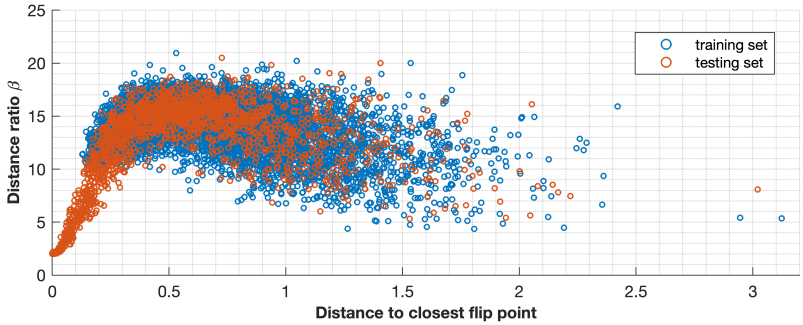

Figure 5.5: β is the distance to the closest flip point divided by the distance predicted by the Taylor approximation. Since these ratios are far from 1, the approximation is not a reliable measure.

closest flip point distance (average of 1.06 times the true distance), but for 7% of the data, a step in that direction goes outside the feasible set of images before passing through a decision boundary. This indicates that it might not be wise to limit the search to the direction of first-order derivatives. And if we limit the search to the direction of first-order derivatives (or any other direction), it would be most reliable to search along that direction for a flip point rather than just estimating the distance. Based on the results in Figures 5.5 and 5.6, we can conclude that the distance obtained by searching the direction of first-order derivatives can be considered a much better approximation method compared

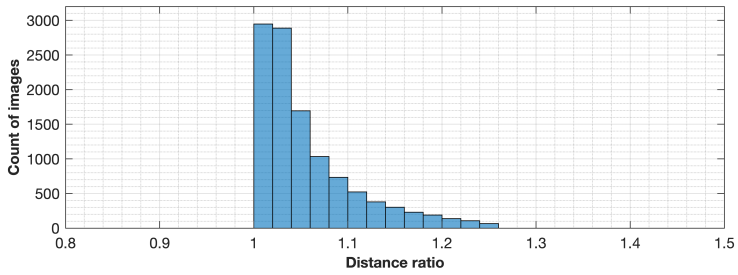

Figure 5.6: Finding the flip point along the direction indicated by the first-order Taylor expansion often gives an accurate estimate of distance to the decision boundary. The horizontal axis is the ratio of the distance to the decision boundary along the Taylor direction to the distance to the closest flip point.

to the distance obtained by the first-order Taylor series approximation method.

Unfortunately, the Taylor direction itself is not so reliable. We look at the angle between the direction defined by the Taylor approximation and that defined by the calculated closest flip point. Large angle between the two directions means the derivative does not point near the closest point on decision boundary. Figure 5.7 shows the distribution of the angles (in degrees) vs the distance to the closest flip points. This clearly shows that the farther an image is from the decision boundary, the larger the angle tends to be. In Figure 5.7, we observe that the lower bound for the angles linearly increases with the distance.

All these observations show that the simplifying assumptions used by Elsayed et al. [2018] and Jiang et al. [2019] can be unreliable, and signify the importance of verification for such simplifying assumptions, whenever used for models with nonlinear activation functions.

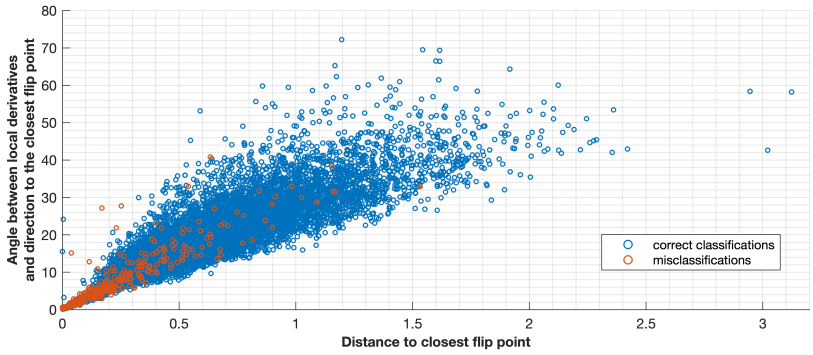

Figure 5.7: Angle between direction of first-order Taylor approximation and direction to closest flip points. These angles are far from 0, indicating that Taylor approximation gives misleading results.

5.5 Shape and connectedness of decision regions

We consider all images of correctly-classified ships in the testing set, and investigate the lines (in image space) connecting each pair of images. 89% of those lines stay within the "ship" class for the model, while 11% do not. The least-connected ship is connected to 220 other ships by lines that do not exit the "ship" region, and there are paths (some using multiple lines) that connect every pair of ships without exiting the "ship" region. This indicates that the "ship" region is star-shaped, providing another reason why linear approximations to decision boundaries are inadequate. These observations also hold for images in the training set. Therefore, the trained network has formed a connected sub-region (in the domain) that defines the "ship" region. This result aligns with the observations reported by Fawzi et al. [2018] that classification regions created by a deep neural network can be connected and a single large region may contain all points of the same label. Fawzi et al. [2018], however, did not investigate the output of network along direct paths between images of the same class.

We performed our analysis by building the adjacency matrix of directly connected

images. Performing spectral clustering [Von Luxburg, 2007] on the graph and the Laplacian derived from the adjacency matrix that incorporates the distance between images, may provide additional insights.

5.6 Adversarial examples and decision boundaries

In most recent studies about adversarial examples, the inputs with adversarial label are obtained by minimizing the loss function of the neural network for that label, subject to a distance constraint [Ilyas et al., 2019, Tsipras et al., 2019]. The distance constraint keeps the adversarial image close to the original image. There are limitations to this approach, as we explain via examples.

Consider the image on the left in Figure 5.8. Figure 5.8 (right) shows the value of the softmax score on the line from this image to its closest flip point and beyond. We compare this with the result of minimizing the loss function of the model for the adversarial label "plane", subject to ℓ_2 distance constraint of 0.5, as suggested by Ilyas et al. [2019]. The adversarial image obtained by this method is much further away, a distance of 0.494 instead of a distance of 0.178 for the closest flip point that we found. So their calculation underestimates the vulnerability of the model. It is also interesting that the line between the image and the adversarial image found by their method crosses a flip point at a distance much less than 0.494, as shown in Figure 5.9, yielding a much better assessment of the vulnerability of the model.

There is another difficulty associated with minimizing the loss for an adversarial label subject to a distance constraint. As an example, consider the image in Figure 5.10, which is at a distance of 2.14 from its closest flip point. Seeking an adversarial image for this image with distance constraint 0.5 will be unsuccessful, as the optimization problem

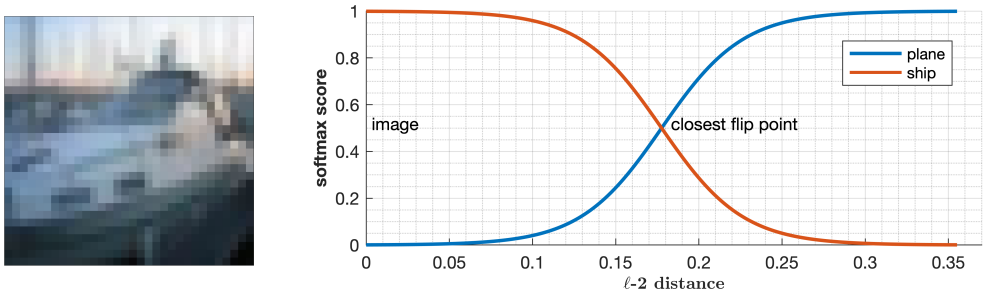

Figure 5.8: Finding the closest flip point reveals the least changes that would lead to an adversarial label for the image.

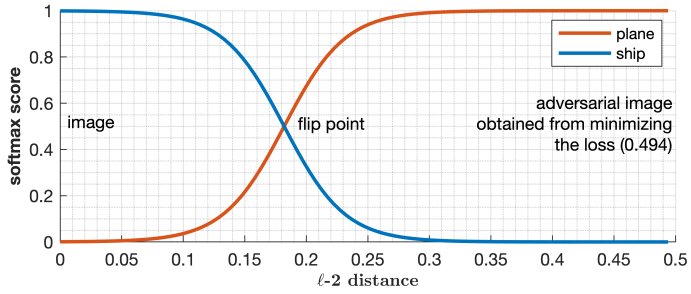

Figure 5.9: Minimizing the loss function subject to a distance constraint may find adversarial examples far from the original image.

has no feasible solution. Finding the closest flip point yields much better information about robustness.

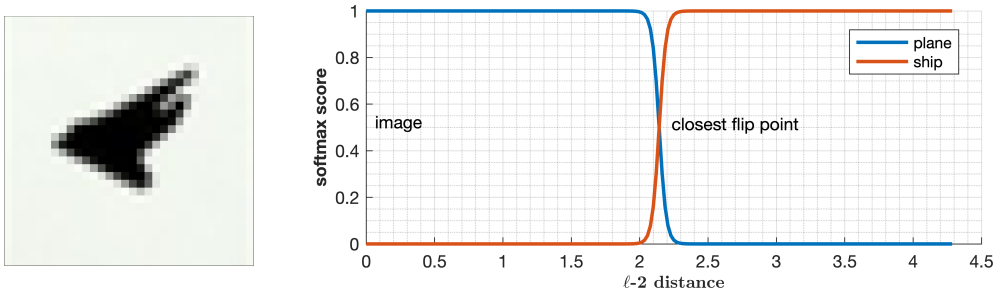

Figure 5.10: Minimizing the loss function subject to a tight distance constraint may not have a feasible solution and would not reveal how robust the model actually is.

We also measure the angle between the direction to the closest flip point, and the direction to the adversarial example found by minimizing the loss function. For the image in Figure 5.9, the angle is 12.7 degrees.

The cost of finding a flip point is comparable to the cost of minimizing the loss function, and it provides much better information.

The distance constraint used by Ilyas et al. [2019] can be viewed as a ball around the input. We showed that choosing the size of that ball can be challenging. If the size of ball is too small, their optimization problem becomes infeasible. If the size of ball is large, they do not find the adversarial example closest to the input. Their problem is non-convex, like ours. The examples above demonstrated that our approach finds a closer adversarial example compared to their approach. The computation times for our method and theirs are quite similar.

Moreover, Figure 5.11 shows that the closest distance to the decision boundary can have a large variation among the images in a dataset. Therefore, tuning the distance constraint for one image may not be insightful for most of the other images in a dataset.

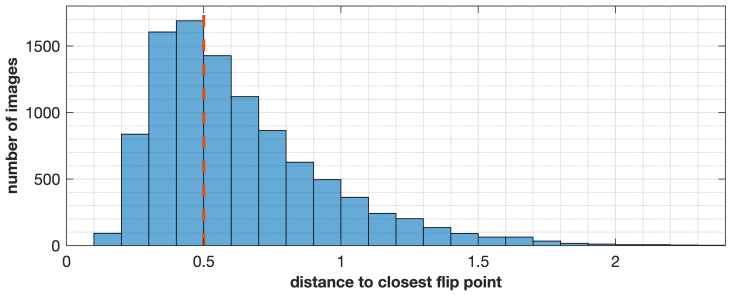

Figure 5.11: Distance to the closest flip point has large variation among images in the training set, which shows that a single distance constraint would not be able to reveal the vulnerabilities of a model for all images. For example, a distance constraint of 0.5 cannot yield an adversarial example for the large fraction of images that are farther than 0.5 from the decision boundaries. It also would not reveal the weakest vulnerabilities for images which are much closer than 0.5 to the decision boundaries.

These observations would still hold for networks trained on the pixels rather than wavelet coefficients.

Regarding the reason for excessive vulnerability of trained neural networks towards

adversarial examples [Goodfellow et al., 2014], there are studies that speculate about the influence of decision boundaries. For example, Tanay and Griffin [2016] argue that "adversarial examples exist when the classification boundary lies close to the submanifold of sampled data", but their analysis is limited to linear classifiers. Shamir et al. [2019] also explain the adversarial examples via geometric structure of partitions defined by decision boundaries; however, they do not consider the actual distance to the decision boundaries, nor the feasibility of changes in the input space, and their analysis is focused on linear decision boundaries.

The analysis provided in this work shows that studies focused on adversarial examples can benefit from using the closest flip points and from direct investigation of decision boundaries, for measuring and understanding the vulnerabilities, and for making the models more robust.

5.7 Summary

We showed the complexities of decision regions of a model can make linear approximation methods quite unreliable, when nonlinear activation functions are used for the neurons. Instead, we used flip points to provide improved estimates of distance and direction of data points to decision boundaries. These estimates can provide measures of confidence in classifications, explain the smallest change in features that change the decision, and generate adversarial examples. Closest flip points are computed by solving a non-convex optimization problem, but the cost of this is comparable to methods used to compute an adversarial point that may be much further away. The closest flip point along a particular direction can be easily computed by a bisection algorithm. Our example involved only two classes and continuous input data, but we have also implemented our

method for problems with multiple classes and discrete features.

In the next chapter, we provide methods to refine the structure of neural networks using matrix conditioning.

Chapter 6: Refining the Structure of Neural Networks Using Matrix Conditioning

6.1 Introduction

In this chapter, we provide computational tools to refine the structure of networks during training, and also to refine networks that are already trained.

Deep learning models have proven to be exceptionally useful in performing many machine learning tasks. However, for each new dataset, choosing an effective size and structure of the model can be a time-consuming process of trial and error. While a small network with few neurons might not be able to capture the intricacies of a given task, having too many neurons can lead to overfitting and poor generalization. Here, we provide a complete set of low-cost computational tools to design the layers of a feed-forward neural network from scratch for any dataset, guided by matrix conditioning and partial training. Results on sample image and non-image datasets demonstrate that our method results in small networks with high accuracies. Finally, using pivoted QR factorization, we provide a method to effectively squeeze models that are already trained. Our techniques reduce the human and computational cost of designing deep learning models and therefore, reduce the expense of using neural networks for applications.

6.2 Literature review and problem statement

We first review the literature from different perspectives and relate our approach to previous methods.

6.2.1 Model design and its difficulties

Among the most important decisions to be made in model design is determining an appropriate size for the network.

The trade-off between the size and accuracy of networks has been studied extensively for benchmark datasets in machine learning, for example by Nowlan and Hinton [1992], Srivastava et al. [2014], Zhang et al. [2016], and Neyshabur et al. [2019]. Through trial and error, standard models have been developed that can achieve the best accuracies on some of those datasets. These achievements are impressive, but they do not give us much guidance about how to approach an unfamiliar dataset.

Furthermore, the standard models are often massive and require specialized hardware, which makes them unapproachable for modest real-world tasks. A few studies focus on developing compact models that can achieve acceptable accuracies on standard datasets, e.g., [Iandola et al., 2016, Zhou et al., 2016]. Still, there is a great need for systematic and affordable procedures to decide an appropriate number of neurons on each layer of a network for an unfamiliar dataset.

Obtaining a compact model might sometimes come at the cost of losing some accuracy. Nevertheless, that compromise might be justifiable or even necessary in certain applications. The huge computational cost or power consumption for some of the best models is prohibitive for certain computers and applications [Canziani et al., 2016], and

hence there has been a focus on developing more economical models that maintain acceptable accuracies [Denton et al., 2014, Han et al., 2015, Howard et al., 2017]. With that in mind, our focus is not to improve the benchmark accuracies, rather to achieve a modest accuracy with a compact model.

One of the reported advantages of deep learning models is sometimes considered to be the automatic detection of important features from the raw data, saving the time required for preprocessing and feature selection. That view is not completely correct as we showed in Section 4.4.3. However, even if these models save time on data preprocessing, the structural design of deep models can be very time-consuming. This can become an obstacle in deploying neural networks in mainstream applications, for example problems related to education [Jiang et al., 2018].

Alvarez and Salzmann [2016] have given a review of earlier approaches to adjusting the size of a neural network. Their method of reducing the size of a neural network is based on adding a penalty term to the loss function in order to detect and remove redundant neurons, while ours is based on partial matrix decompositions layer by layer which can be used to expand or contract a network. Like their method, we do not need to fully train a network before adjusting its size.

Starting with a large network and adding a regularization term to the loss function of the neural network during training is another common approach to reducing its size. For example, Zhou et al. [2016] imposed sparsity constraints on the dense layers of the standard CNNs and demonstrated that most of the neurons in those models can be eliminated without any degradation of the "top-1" classification accuracy. Regularization has also been used in other studies, e.g., Murray and Chiang [2015] for language models.

Although adding regularization terms in the training process is effective in reducing

the over-fitting for over-sized networks, and in identifying redundancies in the standard models, this cannot be considered a direct method to design a neural network from scratch for an unknown dataset. Unlike our method, these methods require an over-sized network with high-accuracy to begin with, and their performance depends on specific optimization methods for the training and careful tuning of additional hyperparameters for each dataset. The method we provide for pruning over-sized networks does not need to retrain a network from scratch; rather it relies on straightforward and relatively inexpensive row and column elimination from the weight matrices and applying the original training method to retrain the squeezed network.

6.2.2 Model architecture search methods

There are methods proposed specifically to design the structure of neural networks. These methods mostly consider a pool of candidate models, and try to choose the best model using different approaches, or they define the networks with a set of parameters, then search the space of those parameters to find their optimal configuration. These methods have been able to effectively automate the process of design, using a significant amount of resources to explore large pools of candidates or exploring a large space of design parameters. Some of the earlier methods use statistical methods such as hypothesis testing to find the best models [Anders and Korn, 1999] or genetic algorithms to search the parameter space [Stanley and Miikkulainen, 2002]. More recently, Zoph and Le [2016] and Baker et al. [2016] used reinforcement learning to search the design space, Liu et al. [2018] developed a sequential model-based optimization (SMBO) strategy and a surrogate model to guide the search through structure space, Zoph et al. [2018] used a combination of transfer learning and reinforcement learning, Pham et al. [2018] used a method that

allowed parameter sharing between the candidate models in order to make the search more efficient, Bender et al. [2018] analyzed a class of efficient architecture search methods based on weight sharing, and Hu et al. [2019] used a linear regression feature selection algorithm and was successful in finding competitive models using a few GPU days.

The methods that try to be more efficient risk the possibility of prematurely discarding good candidates that might not appear good in the first stages of training. Cashman et al. [2019] advocates for recycling the training information for the models that are discarded at the initial steps of model search and provides a visual tool to verify assumptions used in the search in order to make the process interactive.

These approaches can be highly effective in finding a good structure for a neural network. However, they can be generally viewed as an automated version of training many networks and finding the best one. Therefore, they are highly resource expensive, as some of them are reported to take GPU months or years to find the best neural network architecture for a given task [Wistuba, 2017]. This prohibits their use for modest applications with limited computational resources.

Our approach clearly does not fit into this category in terms of resource demand, and in terms of the extent of search. For example, the time it takes to train our network for the MNIST dataset on a 2017 Macbook is about two hours. Nevertheless, our goals are similar in the sense that we aim to find the best architecture for a feed-forward neural network. Hence, our methods can be viewed as a low-cost but efficient way to design the structure of networks for mainstream applications in the real-world.

6.2.3 Our approach and its relation to other approaches based on decomposition of weight matrices

Here, we consider feed-forward neural networks as a general-purpose machine learning model, and develop a training method that can achieve high accuracy by optimizing the number of neurons on each layer of the network, systematically and efficiently.

To achieve our goal, we use the singular values or rank-revealing QR factorization of the stacked weight/bias matrices to determine the redundancies in the network and to identify layers that have an excessive number of neurons. The singular values are non-negative real numbers that provide comprehensive and reliable information about the independence of information in a matrix [Golub and Van Loan, 2012].

One of the early applications of Singular Value Decomposition (SVD) to prune the structure of feed-forward neural networks is by Psichogios and Ungar [1994]. Their method uses a two-stage process for training, in which the weights of a single layer network are optimized in one stage, and the biases are optimized in the other stage, iteratively. In the second stage, where the weights are fixed, optimizing the biases becomes a linear least squares problem which they solve using SVD. Small singular values would then indicate redundant neurons which can be eliminated. This method only applies to single layer networks and does not take into account whether the redundancies in the weight matrix are also present in the bias vector.

Teoh et al. [2006] studied single hidden layer neural networks and related the rank of the weight matrix to the complexity of the decision boundaries of a trained network. They study the singular values of the weight matrix. When the weight matrix is rank deficient, they conclude that there are excessive neurons in the network, use the number of small

singular values to determine how many neurons to remove, and train a new network from scratch. When there is no distinct gap between the singular values of the weight matrix, they conclude that number of neurons might not be sufficient, and they add one neuron to the network and train the new network from scratch. This approach is insightful, but has practical limitations as it is designed for single layer networks. Moreover, as we will show later, when pruning a network, it is more efficient to identify and discard specific columns of the weight matrix (i.e. neurons) that are redundant, instead of discarding the entire weight matrix and training a new network from scratch. Adding neurons one by one also will not be a practical approach for real-world applications; rather, a systematic way for growing the network would be essential.

SVD is used by Xue et al. [2013] to restructure deep network acoustic models. Their approach discards small singular components of the weight matrices and replaces each layer in the network with two new layers, one purely linear, each with fewer nodes than the original single layer. This results in a smaller number of parameters if there are many redundant nodes in the network, but a larger number of parameters if there is little redundancy. They then use additional training if necessary. Chung and Shin [2016] took a similar approach using SVD of weight matrices and replacing each layer in the network with two new layers, but instead of discarding small singular values they sparsify the weight matrices. These approaches do not address the problem of setting the initial structure of the network. As we will show in Section 6.5, pivoted QR factorization can reveal candidates for neurons that can be removed from a trained model in a faster and more effective manner that does not require adding new layers to the network.

Alvarez and Salzmann [2017] have proposed an SVD-based regularization term that promotes weight matrices to become low-rank during the training, with the ultimate goal

of compressing such low-rank weight matrices after the training. SVD has also been used in methods that reconstruct a compact version of a trained neural network, for example, Denton et al. [2014], Goetschalckx et al. [2018], and Xu et al. [2018]. These methods require an oversized but accurate trained model to begin with. As their authors explain, these are methods for reproducing a compact version of a trained network, and not methods for designing the structure of networks.

SVD has also been used in convolutional neural networks with a fixed structure to control the behavior of the Jacobian matrix of the function computed by the neural network, enabling better behavior of the optimization algorithms used for training [Sedghi et al., 2019].

6.2.4 A note on cost

Our goal in this work is to reduce the human and computational cost of designing deep learning models which is sometimes prohibitive of their use in real-world applications. Our work is summarized in three algorithms. The first two algorithms are for designing a neural network from scratch. The first eliminates possibly redundant neurons in order to determine a proper proportion of neurons layer by layer, using partial training of network. The second scales a neural network up or down, again with partial training, preserving the proportions determined by the first algorithm, and chooses a size with low validation and generalization errors. The third is applied to a fully-trained network to remove redundant neurons.

The main tool in our approach is matrix decomposition, in particular, pivoted QR decomposition, rank-revealing QR decomposition, or SVD of the weight-bias matrices for the neural network. It is important to note that *the effort needed for any of these*

decompositions for dense matrices is negligible compared to the overall training process. At each step of training for each mini-batch, the derivative of the loss function is computed for individual training points, with respect to each and every element in the weight matrices, which involves multiplication of weight matrices. The complexity of computing singular values or QR decomposition is of the same order (if exact algorithms are used) or less (if approximate or early-termination algorithms are used).

6.3 Framework

We explain our method for the neural network prototype $\mathcal{N}$ shown in Figure 2.1, as an example. Our method can be easily generalized to neural networks with different architectures, such as convolutional and residual networks.

As mentioned in Chapter 2, we specify a neural network $\mathcal{N}$ by weight matrices $\boldsymbol{W}^{(i)}$ and bias vectors $\boldsymbol{b}^{(i)}$ for each layer $i = 1, \ldots, m$. Any of the typical activation functions can be used: sigmoid, relu, erf, etc. Inputs are denoted by the vector $\boldsymbol{x}$.

We use a training function $\mathcal{T}$, specified by

$$[\hat{\mathcal{N}}, \epsilon^{tr}, \epsilon^{v}] = \mathcal{T}(\mathcal{N}, \mathcal{D}^{tr}, \mathcal{D}^{v}, \eta)$$

to train an existing neural network $\mathcal{N}$ using η epochs. Here, $\mathcal{D}^{tr}$ is the training set and $\mathcal{D}^{v}$ is the validation set. $\mathcal{T}$ returns the trained network $\hat{\mathcal{N}}$, and also the accuracies ϵ^{tr} on $\mathcal{D}^{tr}$ and ϵ^{v} on $\mathcal{D}^{v}$. Networks that are partially or fully trained are distinguished with $\hat{\mathcal{N}}$ while untrained networks are shown as $\mathcal{N}$.

6.4 Designing the structure of a neural network by adaptive restructuring

We would like to design our network so that it learns a training set and generalizes well, i.e., performs well on a validation or a testing set. Given the framework described

above, the main goal is to find the number of neurons needed on each layer of the network, in order for the network to generalize well. Training many possible network structures and choosing the best model can be a very expensive approach as mentioned earlier. Here, we take a more insightful approach based on matrix conditioning of trainable parameters.

Let $\hat{\boldsymbol{W}}^{(i)}$ denote the "stacked" matrix formed by

$$\hat{\boldsymbol{W}}^{(i)} = \begin{bmatrix} \boldsymbol{W}^{(i)} \\ \boldsymbol{b}^{(i)} \end{bmatrix}.$$

We use κ_i to denote the 2-norm condition number of $\hat{\boldsymbol{W}}^{(i)}$. We build our design method based on two insights:

- If a parameter matrix $\hat{\boldsymbol{W}}^{(i)}$ has high condition number compared to other layers, or it is close to rank deficient, this can indicate that the number of neurons on layer i is over-proportioned, compared to other layers. In such cases, we make layer i smaller, so that its share of the overall number of neurons becomes proportionate. We repeat this process until all the matrices have roughly small and similar condition numbers, implying that all layers have the right proportion of neurons.

- Once we have found the distribution of neurons among the layers of a network, the network might still be over-sized or under-sized, so we scale the size of network up or down, maintaining the same proportion of neurons for the layers. By partial training of such networks, we find the network that performs best on a validation/testing set.

These two insights lead to Algorithm 3 and Algorithm 4, which we now present.

6.4.1 Finding a distribution of neurons that leads to small condition numbers among the layers of network

To make use of our first insight, we need to compute the numerical rank of $\hat{\boldsymbol{W}}^{(i)}$, i.e., the number of sufficiently large singular values. This can be computed using the SVD or estimated using approximation algorithms, rank-revealing QR decomposition, or pivoted QR decomposition. The two QR algorithms compute an orthogonal matrix $\boldsymbol{Q}$, an upper-triangular matrix $\boldsymbol{R}$, and a permutation matrix $\boldsymbol{P}$ so that

$$\hat{\boldsymbol{W}}\boldsymbol{P} = \boldsymbol{Q}\boldsymbol{R}.$$

Multiplying $\hat{\boldsymbol{W}}$ by $\boldsymbol{P}$ pulls the columns of $\hat{\boldsymbol{W}}$ deemed most linearly independent (non-redundant) to the left. The magnitudes of the main diagonal elements of $\boldsymbol{R}$ are non-increasing, so we can stop the decomposition when a main diagonal element becomes too small relative to the first.

In Algorithm 3, we reduce the number of neurons on each layer of the network until all of the matrices have condition number less than τ. Although written in terms of the SVD, a rank-revealing QR (explained in Appendix C) could be used instead. If we want the network to have a round number of neurons, we can enforce this condition as we remove neurons. After reducing the number of neurons, we train the new network with η epochs. In our numerical experiments, $\eta \leq 3$ epochs were sufficient to identify redundant neurons. Note that at this stage, we are not concerned about the accuracy of models and our focus is on the values and variations of condition numbers among the layers. Algorithm 3 works based on partial training, does not compute the accuracies, and the steps it takes at each iteration does not necessarily improve the accuracy, especially when it is working on an undersized network. It merely adjusts the number of neurons among the layers of the

network such that the number of neurons for each layer is well proportioned compared to the others. In the next stage, we take the accuracies into account.

Algorithm 3 Algorithm for determining distribution of neurons among layers of a feed-forward neural network

Inputs: Initial neural network $\mathcal{N}$, η , τ , $\mathcal{D}^{tr}$

Output: Neural network with well-conditioned parameter matrices

1: $\hat{\mathcal{N}} = \mathcal{T}(\mathcal{N}, \mathcal{D}^{tr}, [-], \eta)$
2: **while** any weight matrix $\hat{\boldsymbol{W}}^{(i)}$ of $\hat{\mathcal{N}}$ has condition number $> \tau$ **do**
3: **for** all such weight matrices **do**
4: If $\hat{\boldsymbol{W}}^{(i)}$ of $\hat{\mathcal{N}}$ has p singular values less than $1/\tau$ times the largest one, then remove p neurons from layer i in $\mathcal{N}$.
5: **end for**
6: $\hat{\mathcal{N}} = \mathcal{T}(\mathcal{N}, \mathcal{D}^{tr}, [-], \eta)$
7: **end while**
8: **return** $\mathcal{N}$

6.4.2 Scaling the size of a neural network

After we have the right proportion of neurons on each layer, we can expand or contract the neural network, maintaining these proportions. The goal here is to find the overall number of neurons needed to achieve the highest accuracy possible for the model. We need to estimate the generalization error as we modify the number of neurons. Therefore, we reserve part of the training set as a validation set, if a separate validation set is not available.

In Algorithm 4, we begin with a base model $\mathcal{N}^0$, possibly obtained from Algorithm 3, with a good proportion of neurons on each layer. Given a set of positive scalars $\{\beta_1, \ldots, \beta_p\}$, we construct p new models, where $\mathcal{N}^j$, increases or decreases the number of neurons in all layers of the base network $\mathcal{N}^0$ by a factor β_j. This way we obtain $p + 1$ models of different size, with the same distribution of neurons on their layers. Each model

is trained q separate times, from scratch, using η epochs, and the errors on the training and validation sets are averaged to obtain $\hat{\epsilon}^{j,tr}$ and $\hat{\epsilon}^{j,v}$, where $j \in \{1, \ldots, p\}$. We use $q = 5$ in our computations, since no significant change was observed when using larger values.

Algorithm 4 Algorithm for optimizing the overall number of neurons in a neural network, while maintaining the proportion of neurons among the layers

Inputs: Base model $\mathcal{N}^0$ (obtained from Algorithm 3), $\{\beta_1, \ldots, \beta_p\}$, $\mathcal{D}^{tr}$, $\mathcal{D}^v$, η , q

Outputs: Refined trained $\hat{\mathcal{N}}$

for $j = 1$ to p **do**
  Change the number of neurons in all hidden layers of $\mathcal{N}^0$, by a factor of β_j, to obtain $\mathcal{N}^j$.
  for $l = 1$ to q **do**
    $[\mathcal{N}^j, \epsilon_l^{j,tr}, \epsilon_l^{j,v}] = \mathcal{T}(\mathcal{N}^j, \mathcal{D}^{tr}, \mathcal{D}^v, \eta)$
  end for
  $\hat{\epsilon}^{j,tr} = \frac{1}{q} \sum_{l=1}^{q} \epsilon_l^{j,tr}$
  $\hat{\epsilon}^{j,v} = \frac{1}{q} \sum_{l=1}^{q} \epsilon_l^{j,v}$
end for
Choose the model that has the least $2\hat{\epsilon}^{j,v} - \hat{\epsilon}^{j,tr}$ as $\hat{\mathcal{N}}$.
Fully train $\hat{\mathcal{N}}$.
return $\hat{\mathcal{N}}$

Among these $p+1$ models, we want to choose the model that has the least validation error, $\hat{\epsilon}^{j,v}$, and the least generalization error, $\hat{\epsilon}^{j,v} - \hat{\epsilon}^{j,tr}$. Hence, we choose the one that has the least sum of validation and generalization errors, $2\hat{\epsilon}^{j,v} - \hat{\epsilon}^{j,tr}$. This procedure is formalized as Algorithm 4.

By finding the model that minimizes $2\hat{\epsilon}^{j,v} - \hat{\epsilon}^{j,tr}$, we avoid over-fitting and under-fitting in the model. If the smallest or largest model happens to have the smallest $2\hat{\epsilon}^{j,v} - \hat{\epsilon}^{j,tr}$, we could extend our investigation beyond the p models, by adding more β's in the direction of smaller or larger models.

6.5 Squeezing trained networks

The two algorithms in the previous section design and train a network from scratch, given the desired number of layers. We now introduce a method to squeeze networks that are already trained but have excess neurons. The method we propose does not necessarily retain the accuracy of the trained model, but it preserves the main essence of it. In our numerical results, we demonstrate that squeezed networks either closely retain the accuracy, or they can be retrained to the best accuracy very quickly.

For squeezing trained networks, we use Algorithm 5. As in Algorithm 3, we reduce the number of neurons whenever we encounter a weight matrix with large condition number, but now we need a way to identify the most useful neurons. To do this, we use the rank-revealing QR decomposition.

Note that we do not need to compute the full QR decomposition; we can stop when a diagonal element of $\boldsymbol{R}$ becomes too small.

The parameter τ defines the threshold for excessive neurons. If it is large, the output of Algorithm 5 can be the same network as the input, and the user might then choose to reduce τ.

Our squeezing method is straightforward and simple to use. Unlike methods that rebuild a trained model using specialized training methods, we keep the trained network intact except for redundancies. After squeezing, one can retrain the obtained network with a few epochs, which sometimes leads to even better accuracy. For retraining in our approach, one can use the original method of training, and there would be no necessity for specific loss functions and optimization methods.

It is important to remember, as mentioned in section 6.2.4, that the effort needed

Algorithm 5 Algorithm for squeezing a trained feed-forward neural network

Inputs: Trained neural network $\hat{\mathcal{N}}$, τ , $\mathcal{D}^{tr}$

Outputs: Squeezed neural network with same or smaller number of neurons

for $i = 1$ to m **do**
  if $\tau < \kappa_i$ **then**
    $[\boldsymbol{Q}, \boldsymbol{R}, \boldsymbol{P}] = \mathrm{QR}\big(\hat{\boldsymbol{W}}^{(i)}\big)$
    Define p so that $|r_{p+1,p+1}| < \tau |r_{11}|$ and $|r_{pp}| \geq \tau |r_{11}|$
    Remove columns $p+1 : n_i$ of $\boldsymbol{P}$ from $\hat{\boldsymbol{W}}^{(i)}$
    Compute condition number of shrunk $\hat{\boldsymbol{W}}^{(i)}$ as κ_i
    while $\tau < \kappa_i$ **do**
      $p = p - 1$
      Remove column $p+1$ of $\boldsymbol{P}$ from $\hat{\boldsymbol{W}}^{(i)}$
      Compute condition number of shrunk $\hat{\boldsymbol{W}}^{(i)}$ as κ_i
    end while
    Remove neurons $p+1 : n_i$ of $\boldsymbol{P}$ from the network, by removing corresponding columns of $\boldsymbol{W}^{(i)}$, rows of $\boldsymbol{W}^{(i+1)}$, and elements of $\boldsymbol{b}^{(i)}$
  end if
end for
Improve $\hat{\mathcal{N}}$ by retraining, if desired.
return squeezed $\hat{\mathcal{N}}$

for computation of pivoted QR decomposition of the weight matrices of a network is negligible compared to the overall training process. We recommend our Algorithm 5 as a computationally inexpensive and approachable method to squeeze trained networks and to gain insight about their compressibility. Other sophisticated methods that rebuild the networks from scratch may have certain advantages in particular applications, but their computational cost may be much higher.

In Algorithm 5, use of pivoted QR or RR-QR is necessary because we need to know which specific neurons are redundant, the information we obtain from the permutation matrix of decomposition. The while loop in our algorithm makes sure the condition

number of resulting matrices are below the τ, after elimination of redundant neurons. This is because line 4 of our algorithm may possibly overestimate the rank of matrix leading to elimination fewer than necessary neurons. This while loop usually takes zero or very few iterations, and it is not an essential part of the algorithm. In cases where a network is squeezed, retrained, and squeezed again, using the while loop may reduce the overall cost, because it could cause line 2 of the algorithm not to be invoked in the subsequent squeeze. In other cases, where squeezing is applied once, the while loop can be dropped.

6.6 Numerical results

In our numerical results, we use TensorFlow to train the networks, with Adam optimizer and learning rate of 0.001. We also use a tunable error function as the activation function as explained in Chapter 2, but keep in mind that our training method does not depend on the choice of activation function. We start with MNIST which can be considered an unfamiliar dataset, because we use the wavelet coefficients of images, instead of the pixel data.

6.6.1 MNIST

The MNIST dataset has 10 output classes, as explained in Section 4.4.1.1. We represent each data point as a vector of length 200, using the Haar wavelet basis. The 200 most significant wavelets are chosen by rank-revealing QR decomposition of the matrix formed from the wavelet coefficients of all images in the training set. Using this small number of wavelet coefficients and a simple feed-forward network will lead to accuracy of about 98.7%. Accuracy could be improved using more wavelet coefficients, and using

regularization techniques in the literature, but this accuracy is adequate to demonstrate the effectiveness of our method.

Using Algorithm 3. We consider a neural network of 12 hidden layers with 300 nodes on each layer as the input to Algorithm 3. After the initial training of this model, the condition numbers of the stacked weight matrices vary between 2 and $2,652$, as shown in the third column in Table 6.1. We use Algorithm 3 to adjust the proportions of neurons, with $\tau = 25$ and $\eta = 1$. The number of neurons and the condition numbers of the matrices for the output of Algorithm 3 are presented in the last two columns in Table 6.1. At each iteration, we have rounded down the number of neurons obtained at line 4 of the algorithm to a multiple of 5.

Table 6.1: Condition numbers of the stacked matrices and the number of neurons on each of the 12 layers of the network processed by Algorithm 3 to learn 200 wavelet coefficients for MNIST.

Layer (i)	**Initial network**		**Algorithm 3**	
	n_i	κ_i	n_i	κ_i
1	300	10	300	9
2	300	649	205	10
3	300	301	255	15
4	300	2,652	210	20
5	300	275	250	22
6	300	583	210	23
7	300	946	180	24
8	300	268	150	24
9	300	433	120	17
10	300	1,269	95	14
11	300	398	65	11
12	300	673	25	4
13	10	2	10	4

We observe that the final condition numbers are relatively close to each other and less than τ. Additionally, they monotonically increase towards the middle layer and then

monotonically decrease towards the output layer. This monotonicity of condition numbers is not a requirement and might not be achieved for all models.

Using Algorithm 4. The previous step found a promising set of proportions for the sizes of the layers. Using the output of Algorithm 3, given in Table 6.1, as our base model, we scale this network to try to improve the accuracy. We chose eight β's ranging from 1 to 2.4, with increments of 0.2.

For this step, we need a validation set. Hence, we remove 10,000 images from the training set, randomly selecting 1,000 images from each class to use as a validation set $\mathcal{D}^v$. This leaves the training set $\mathcal{D}^{tr}$ with only 50,000 images.

We use a batch size of 50, and set $q = 5$, $\eta = 1$. Algorithm 4 partially trains all eight models to achieve the errors shown in Table 6.2 and Figure 6.1. It then chooses the model with $\beta = 2$ as the best model and trains it using all 60,000 images in the training set, to achieve 100% and 98.68% accuracies on the training and testing sets, respectively. Achieving this accuracy with such a small neural network is remarkable, considering that we only used 200 wavelet coefficients, and we did not use any regularization or any sophisticated architecture for the network.

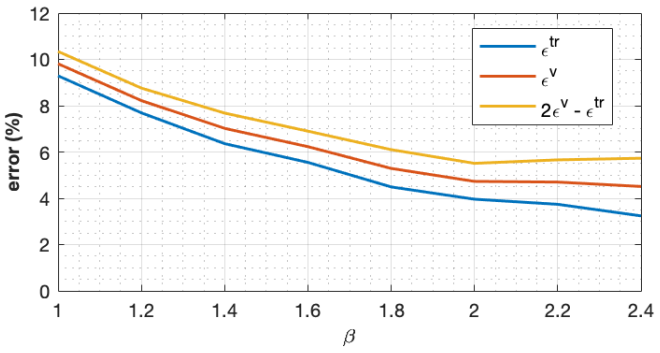

Figure 6.1: Errors of the eight models investigated by Algorithm 4 for MNIST. We have chosen the model with $\beta = 2.0$, because it has the least sum of validation and generalization errors, when models are partially trained with 1 epoch.

Table 6.2: Errors of the eight networks obtained from Algorithm 4, defined by the β's, partially trained on the reduced training set (with 50,000 images) and validated using 10,000 images.

$\boldsymbol{\beta}$	$\hat{\boldsymbol{\epsilon}}^{\boldsymbol{tr}}$	$\hat{\boldsymbol{\epsilon}}^{\boldsymbol{v}}$	$\mathbf{2}\hat{\boldsymbol{\epsilon}}^{\boldsymbol{v}} - \hat{\boldsymbol{\epsilon}}^{\boldsymbol{tr}}$
1.0	9.29	9.81	10.34
1.2	7.69	8.22	8.76
1.4	6.36	7.02	7.68
1.6	5.55	6.23	6.90
1.8	4.49	5.29	6.10
2.0	3.96	4.73	5.51
2.2	3.74	4.70	5.66
2.4	3.24	4.51	5.73

For all of these models, the condition numbers of the stacked matrices are similar to those presented in the 5th column of Table 6.1 and smaller than 25. This indicates that partial training with $\eta = 1$ has adequately captured the conditioning of the matrices. It also indicates that scaling the number of neurons, while maintaining their proportions layer-by-layer, has little effect on the condition numbers for the stacked matrices.

Verifying the results. Here, we investigate whether the model we obtained with $\beta = 2.0$ is in fact the best network we can choose from the pool of networks defined by the eight β's. For this, we fully trained all eight of the networks, obtaining the errors in Table 6.3. Evidently, the best accuracy is achieved by the model chosen by Algorithm 4, confirming the effectiveness of our method.

Table 6.3: Testing error when models are fully trained with all 60,000 images in the MNIST training set

$\boldsymbol{\beta}$	1	1.2	1.4	1.6	1.8	2.0	2.2	2.4
$\hat{\boldsymbol{\epsilon}}^{\boldsymbol{te}}$	1.97	1.76	1.65	1.56	1.43	1.32	1.56	1.61

Clearly, trying to squeeze this trained network using Algorithm 5 with $\tau \geq 25$

will have no effect. Overall, we observed that choosing a value of τ between 20 and 50 in Algorithm 3, and then applying Algorithm 4, leads to similar networks with similar best accuracies. However, choosing τ outside of this range leads to models with slightly inferior accuracies. The key factor in choosing a good value for τ seems to be the variance of condition numbers among the layers. The values of τ that deliver the best results also yield condition numbers with small variance among the layers. In fact, one can choose the initial value of τ as the mean of condition numbers and update it as the condition numbers change, until the variance becomes small. This approach would yield similar results as when we choose the value τ between 30 and 40 in the first place. This range can be viewed as a practical choice for τ.

6.6.2 Adult Income dataset

Next, we consider the Adult Income dataset with the same preprocessing explained in Section 4.4.2.

Using Algorithm 3. We consider a neural network of 12 hidden layers with 50 nodes on each layer as the input to Algorithm 3. Similar to the previous section, the properties of the initial and final network are presented in Table 6.4. For this dataset, we have not rounded the number of neurons, and we have used batch size of 20, $\eta = 3$, and $\tau = 40$. This time, we choose a larger η compared to previous example, because our network is much smaller and training with the larger η still takes just a few seconds. We also choose a larger τ because the condition numbers tend to remain large during the process. We discuss the factors involved in choosing these parameters further in Section 6.6.5.

Using Algorithm 4. Using the proportions found in the previous step, we consider

Table 6.4: Condition numbers of the stacked matrices and the # of neurons on each of the 12 layers of the network processed by Algorithm 3 to learn the Adult Income dataset.

Layer (i)	**Initial network**		**Algorithm 3**	
	n_i	κ_i	n_i	κ_i
1	50	9	44	7
2	50	654	39	37
3	50	658	32	31
4	50	583	22	13
5	50	230	20	30
6	50	224	15	13
7	50	159	12	20
8	50	912	8	20
9	50	136	5	14
10	50	377	4	7
11	50	74	8	16
12	50	110	6	18
13	2	1	2	3

eight β's ranging from 0.6 to 2.0, with increments of 0.2. For the validation set $\mathcal{D}^v$, we randomly remove 10% of the data points from the training set, leaving the training set $\mathcal{D}^{tr}$ with 90% of its data points.

Based on the results of Algorithm 4, shown in Figure 6.2, we choose the neural network with $\beta = 1.4$. After fully training this model we achieve 86.05% accuracy on the testing set, which is comparable to the best accuracies reported in the literature [Friedler et al., 2019, Mothilal et al., 2019].

Verifying the results. To verify the results, we fully train all the models defined by the eight β's to achieve the testing errors in Table 6.5. We observe that the best accuracy is indeed achieved by the model with $\beta = 1.4$.

6.6.3 Using Algorithm 5 to squeeze networks trained on MNIST

Squeezing the model obtained via Algorithms 3 and 4 may lead to even

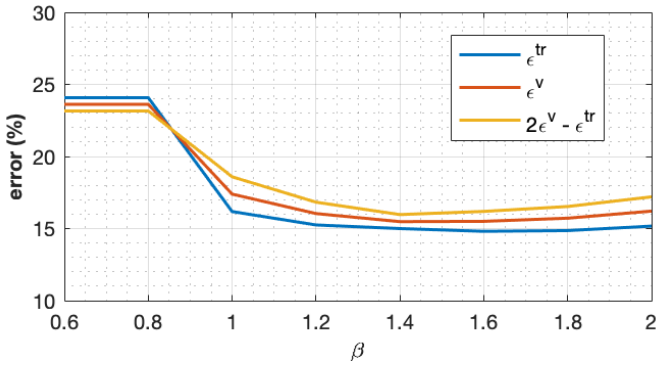

Figure 6.2: Errors of the eight models investigated by Algorithm 4 for the Adult Income dataset. We have chosen the model with $\beta = 1.4$, because it has the least sum of validation and generalization errors when trained partially.

Table 6.5: Testing error when models are fully trained with all data in the Adult Income training set

$\boldsymbol{\beta}$	0.6	0.8	1.0	1.2	1.4	1.6	1.8	2.0
$\hat{\boldsymbol{\epsilon}}^{te}$	14.34	14.19	14.22	14.16	13.95	14.14	14.19	14.32

better accuracy. We first consider the refined network we trained with $\beta = 2$ that achieved 98.68% accuracy on the testing set. For this model, all the condition numbers κ_i happen to be less than 23. Hence, we squeeze the model with τ ranging between 22 and 18. Table 6.6 shows the results. After squeezing, we measure the accuracy of the model on the testing set and then retrain it, stopping when we reach 100% accuracy on the training set.

Table 6.6: Number of neurons and accuracies of the model with $\beta = 2$, trained in Section 6.6.1, squeezed by Algorithm 5 with different values of τ.

$\boldsymbol{\tau}$	**Number of neurons removed**	**Accuracy of squeezed model before retraining**	**Accuracy of squeezed model after retraining**
22	5	98.47	98.68
20	408	90.35	**98.74**
18	502	80.94	98.70

It is notable that retraining the squeezed models may lead to accuracies better than the original model. Table 6.7 shows the size and conditioning of the best model, with accuracy of 98.74%.

Table 6.7: Condition number of the stacked matrices and the number of neurons on each layer of the model with 98.74% accuracy on MNIST.

Layer (i)	1	2	3	4	5	6	7	8	9	10	11	12	13
$\boldsymbol{n}_i$	600	410	510	420	364	319	278	249	212	180	130	50	10
$\boldsymbol{\kappa}_i$	4	10	17	18	20	20	20	20	19	19	10	4	2

Squeezing can accurately detect excess neurons. Here, we consider the model in Table 6.7, add 20 neurons on its 4th and 8th hidden layers, and fully train it to achieve 98.47% accuracy. This decrease in the accuracy can be associated with overfitting, caused by addition of those 40 neurons. Table 6.8 shows the condition numbers of the stacked matrices for this model. We observe that condition numbers have increased not only for layers 4 and 8, but also for the in-between layers 6 and 7.

Table 6.8: Condition number of the stacked matrices for the model that has 20 more neurons on its 4th and 8th layers compared to the model in Table 6.7. The condition numbers of layers 4,6,7 and 8 have noticeably increased above the $\tau = 20$ we had used to squeeze that model.

Layer (i)	1	2	3	4	5	6	7	8	9	10	11	12	13
$\boldsymbol{n}_i$	600	410	510	440	364	319	278	269	212	180	130	50	10
$\boldsymbol{\kappa}_i$	4	10	16	26	19	30	30	75	16	21	12	4	2

Algorithm 5 enables us to extract some extra neurons from this model, but we have to choose the τ wisely. By looking at the condition numbers in Table 6.8, we see that only layer 8 has condition number > 30, hence, we squeeze the model with $\tau = 30$. The algorithm discards 20 neurons from the 8th layer, leaving a model with 98.45% accuracy.

Retraining this model leads to 98.55% accuracy.

Similarly, if we squeeze the model with $\tau = 25$, a total of 37 neurons will be discarded from the network: 3 neurons from the 4th layer, 4 neurons from the 6th layer, 5 neurons from the 7th layer, and 25 neurons from the 8th layer. The accuracy of the squeezed model is 98.39% before retraining, and 98.66% after retraining.

So, Algorithm 5 enabled us to effectively extract extra neurons from the model and obtain better accuracies.

Squeezing an oversized model reduces the overfitting and improves the accuracy. As the last experiment on MNIST, we study an oversized network with 600 neurons on each layer. Training this oversized network leads to accuracy of 98.3% on the testing set, clearly because of overfitting. This model has 3,438 more neurons compared to the model with best accuracy in Table 6.7, leading to an increase of 199% in the number of training parameters.

We squeeze this oversized model using Algorithm 5 with different values of τ. The results of squeezing are presented in Table 6.9. The accuracies of models decrease after squeezing, although after retraining the squeezed models we obtain accuracies as good, or even better than the original oversized model. This improvement demonstrates the effectiveness of Algorithm 5 in reducing the over-fitting and discarding the excess neurons.

After retraining the squeezed models in Table 6.9, the condition numbers of most matrices go above the τ used for squeezing, indicating that models are still highly over-sized and can be squeezed further to achieve better accuracy. However, this process of squeeze/retrain iterations is less effective than using Algorithms 3 and 4. So, as a general practice we do not recommend starting the training process with an oversized model.

In the next section, we will further investigate the squeezing process on the Adult

Table 6.9: Number of neurons removed and the resulting accuracies, after squeezing an oversized model with 600 neurons per hidden layer, using Algorithm 5 with different values of τ.

τ	Number of neurons removed	Accuracy of squeezed model before retraining	Accuracy of squeezed model after retraining
500	27	98.16	98.30
200	87	97.93	98.30
100	178	96.90	98.30
50	371	90.64	98.32
40	499	90.44	98.35
35	573	90.24	98.39
30	660	88.88	98.53
25	787	86.38	98.41
20	998	63.37	98.39

Income dataset, and will also study a highly oversized model.

6.6.4 Using Algorithm 5 to squeeze networks trained on the Adult Income dataset

Squeezing the model obtained via Algorithms 3 and 4 may lead to even better accuracy. Let's consider the best model obtained in Section 6.6.2 with $\beta = 1.4$. Using Algorithm 5, we squeeze that model with different values of τ. Clearly, squeezing with $\tau > 40$ will return the exact same model. Table 6.10 shows the number of neurons removed from the model for three values of τ, along with the accuracy of the squeezed models, before and after retraining.

We see that squeezing the model with $\tau = 35$ has led to a model with even better accuracy: 86.11% on the testing set. Properties of this model are presented in Table 6.11. For retraining, we have only used 5 epochs.

Squeezing an oversized model reduces the overfitting and improves the accuracy. Let's consider a large model with 100 neurons on each layer, which has 110,581

Table 6.10: Number of neurons and accuracies of the model with $\beta = 1.4$ trained in Section 6.6.2, after being squeezed by Algorithm 5 with different values of τ. Accuracies of squeezed models have not dropped drastically, and retraining has led to a better accuracy for $\tau = 35$ and 30.

τ	**Number of neurons removed**	**Accuracy of squeezed model before retraining**	**Accuracy of squeezed model after retraining**
35	20	84.76	86.11
30	46	78.23	86.09
25	77	76.38	86.03

Table 6.11: Condition number of the stacked matrices and the number of neurons for the model with 86.11% accuracy on the Adult Income testing set.

Layer (i)	1	2	3	4	5	6	7	8	9	10	11	12	13
$\boldsymbol{n}_i$	53	48	40	30	26	21	16	11	7	5	11	8	2
$\boldsymbol{\kappa}_i$	35	19	27	24	33	20	30	12	12	14	14	24	2

more trainable parameters, compared to the model in Table 6.11, an increase of more than 10 times in the number of training parameters. After training this network, we obtain 85.28% accuracy on the testing set. When using Dropout [Srivastava et al., 2014] during the training, the common approach to avoid overfitting, we achieve 85.45% accuracy, which is still far less than 86.11% obtained using our methods.

Let's now use Algorithm 5 to squeeze the trained (oversized) model above with 85.28% accuracy, and then retrain it. We perform this squeezing and retraining, with different values of τ, and the corresponding results are presented in Table 6.12. Each squeezed model is retrained with 10 epochs. We observe that squeezing with τ between 30 and 40 has led to best improvements in the accuracy. This improved accuracy (as a result of squeezing and retraining) is smaller than the best accuracy of 86.11%, but, it is better than the accuracy of the model trained using Dropout. We also note that although

squeezing makes the condition number of $\hat{\boldsymbol{W}}$ matrices $\leq \tau$, the condition numbers can increase above τ during retraining. This is to be expected, because the squeezed models are still highly oversized.

Table 6.12: Number of removed neurons and accuracies of an oversized model with 100 neurons per hidden layer, after being squeezed by Algorithm 5 with different values of τ.

τ	**Number of neurons removed**	**Accuracy of squeezed model before retraining**	**Accuracy of squeezed model after retraining**
100	29	82.65	85.25
50	86	82.09	85.54
40	95	80.73	85.63
35	104	80.92	85.69
30	120	79.76	85.62
20	166	76.99	85.45
10	315	76.38	85.57

To provide the last insight, let's look into the model squeezed with $\tau = 35$. After retraining, several of its $\hat{\boldsymbol{W}}$ matrices have condition number greater than 35. We repeat the process, squeezing it with the same τ, and then retraining it with 10 epochs. After 4 squeeze/retrain iterations, we obtain a model that can no longer be squeezed. Table 6.13 shows how the network has evolved through this process.

The squeezed model has 823 more neurons compared to the model that achieved 86.11% accuracy. It is also less accurate because of overfitting. However, we should note that this squeezing process improved the accuracy of oversized model from 85.28% to 85.81%, which is significant for such a computationally inexpensive process. This demonstrates the effectiveness of Algorithm 5 in squeezing networks with excessive neurons.

Table 6.13: Number of neurons for a 12-layer network trained on the Adult Income dataset, squeezed using Algorithm 5 with $\tau = 35$ and retrained with 10 epochs, repeatedly, until it cannot be squeezed further. Reported accuracy is on the testing set, after the retraining. Squeezing is computationally inexpensive and significantly improves the accuracy.

Layer (i)	**Squeeze iteration**				
	0	1	2	3	4
1	100	75	72	68	67
2	100	100	100	100	100
3	100	89	86	86	86
4	100	84	80	80	78
5	100	100	100	100	100
6	100	93	92	92	92
7	100	87	84	84	84
8	100	100	100	100	100
9	100	91	91	90	90
10	100	85	85	85	85
11	100	100	100	100	100
12	100	92	92	92	92
Accuracy (%)	85.3	85.68	85.7	85.73	85.81

6.6.5 Evolution of networks during training and choosing the hyperparameters

Here, we provide more information about the evolution of network parameters during the training and provide guidance to choose the hyperparameters in our algorithms.

Choosing η: This parameter is the number of training epochs before refining the network. In Algorithm 3, if the condition numbers of stacked weight matrices remain mostly similar after a certain number of epochs, then it would be inefficient to choose an η larger than that number of epochs. In our experiments, even half of an epoch captures the condition number closely. Of course, we cannot guarantee that one (or half) epoch will be adequate for all datasets. Hence, the user of our algorithm should perform an initial experiment to see how the condition numbers of weight matrices change relative to

each other, as the number of training epochs increases, and then choose a good value for η accordingly. If computational resources are abundant, choosing a larger η would be a safe approach. Nevertheless, if an insufficient number is chosen for η, some of the condition numbers might go up again and become disproportionate after the final training of the network. This would prompt the user to either squeeze the obtained network or repeat Algorithm 3 with a larger η.

Tables 6.14 and 6.15 show the evolution of condition numbers for two different models trained on the MNIST example, along with the corresponding number of neurons that should be removed using $\tau = 30$. The model in Table 6.14 is an oversized network, and the model in Table 6.15 is undersized, compared to the model we obtained, earlier, with the best accuracy. We can see that condition numbers largely remain the same, as we train the models with more epochs.

Choosing τ and consistency of results: In our experience, different choices of τ within a reasonable range (25 - 40) do not affect the final outcome (See Table 6.6). As mentioned earlier, for any network, the condition numbers of its stacked weight matrices and their variance among the layers of the network can be the best indicator of redundancies present in the network. When some of the layers have condition numbers much larger than others, one could conclude that those layers have excessive neurons compared to others. On the other hand, when condition numbers have small value and small variance, it indicates that the neurons are well-distributed among the layers. Such a network might still need to be scaled up or down, to achieve the best accuracy.

Looking at the mean and variance of the condition numbers guides us to choose a good value for τ for various datasets. If unsure about choosing the τ, the mean of condition numbers among the layers is a reasonable choice. As we progress through our

Table 6.14: Evolution of condition number of stacked weight matrices of a 9-layer neural network with 600 neurons per hidden layer, trained on our MNIST example. Network is oversized and we expect hidden layers 2 through 9 to lose neurons. The high condition number of layers compared to the first layer is aligned with our expectation. Notice that this is noticeable even after training with small number of epochs and the number of neurons removed from each layer, p_i, does not vary much with respect to η.

Layer (i)	n_i	η									
		.5		1		2		5		10	
		κ_i	p_i	κ_i	p_i	κ_i	p_i	κ_i	p_i	κ_i	p_i
1	600	4	0	4	0	4	0	4	0	4	0
2	600	1,050	25	1,041	24	1,207	24	627	24	843	24
3	600	3,609	24	1,551	24	3,398	24	6,878	24	1,265	24
4	600	663	24	642	25	1,083	25	5,657	25	1,317	25
5	600	961	26	1,664	26	2,472	26	661	26	963	26
6	600	2,212	25	1,746	24	1,268	24	751	24	1,485	24
7	600	1,206	26	882	25	904	25	1,961	25	1,667	25
8	600	881	24	899	24	802	24	820	24	768	24
9	600	537	25	576	26	744	26	1,077	26	1,948	26
10	10	1	0	1	0	1	0	1	0	1	0

structure refinement, we can observe how the mean of condition numbers drops to a small range and how the variance among them becomes small.

Table 6.16 shows the evolution of a 9-layer network, when processed by Algorithm 3 with $\tau = 30$. It only takes 7 iterations until the network satisfies $\tau \leq 30$ for all of its layers. Performing these 7 iterations take less than 3 minutes on a 2017 Macbook.

When we apply Algorithm 3 to the same network, choosing τ at each iteration to be the mean of condition numbers, it takes 6 iterations to achieve a very similar network.

Our algorithms are also not very sensitive to the choice of network that we start with. This is mainly because we have uncoupled the question of finding the right proportion of neurons for the layers of the network (Algorithm 3), and the question of finding the overall size of the network (Algorithm 4). Still, being smart in choosing the initial

Table 6.15: Evolution of condition number of stacked weight matrices of a 9-layer neural network with 100 neurons per hidden layer, trained on our MNIST example. Condition numbers indicate that layers 2 through 9 have excessive neurons compared to the first layer, as we expect.

Layer (i)	n_i	η									
		.5		1		2		5		10	
		κ_i	p_i	κ_i	p_i	κ_i	p_i	κ_i	p_i	κ_i	p_i
1	100	5	0	5	0	5	0	5	0	6	0
2	100	124	4	119	4	144	4	249	4	339	4
3	100	218	5	245	5	236	5	332	5	398	5
4	100	101	5	108	5	106	5	109	5	129	5
5	100	196	5	219	5	241	5	249	5	229	5
6	100	439	4	424	4	462	4	449	4	332	5
7	100	184	4	182	4	176	4	166	4	151	5
8	100	365	5	283	5	256	5	222	5	278	5
9	100	153	4	180	5	188	5	213	5	229	5
10	10	2	0	2	0	2	0	2	0	2	0

network structure can significantly reduce the time it takes for the algorithms to refine the structure.

For example, consider the example in Table 6.16. Instead of starting from the network shown for iteration 0 of Table 6.16, we start with a 9-layer network that has 600 neurons on all of its hidden layers. This time, it takes 47 iterations for Algorithm 3 to adjust the distribution of neurons, way more than 7 iterations. However, the output is very similar as shown in Table 6.17, considering the total number of neurons which are 3,214 and 3,298 for the two networks, and the number of trainable parameters which are 1,270,596 and 1,266,728, respectively.

Finally, we note that the final networks can be slightly different depending on the starting network, since there are generally many networks that fit the training data. Overall, in our experience, Algorithm 4 chooses a similarly sized scaled-up network, if we start

Table 6.16: Evolution of number of neurons n_i for a 9-layer neural network, trained on our MNIST example, as it is refined with Algorithm 3.

Layer (i)	n_i for iterations of Algorithm 3							
	0	1	2	3	4	5	6	7
1	600	600	600	600	600	600	600	600
2	540	536	533	532	530	529	529	527
3	480	480	477	474	471	470	468	466
4	420	420	419	418	418	415	413	412
5	360	360	360	360	360	360	359	359
6	300	300	300	300	300	300	300	300
7	240	240	240	240	240	240	240	240
8	180	180	180	180	180	180	180	180
9	120	120	120	120	120	120	120	120
10	10	10	10	10	10	10	10	10
$\max(\kappa_i)$	49.4	42.5	39.2	33.6	31.7	31.4	31.0	29.2
$\Sigma(n_i)$	3,250	3,246	3,239	3,234	3,229	3,224	3,219	3,214

with a smaller network for Algorithm 3. The key point is obtaining networks of similar accuracy and size, not obtaining a particular network. Clearly, using an oversized network as the input to Algorithm 3 and then contacting it with Algorithm 4 will be computationally more expensive than the alternative approach of starting with a modest network for 3 and then expanding it with Algorithm 4. This is because, in the latter case, the partial training of networks by Algorithm 3 will be performed on a smaller network.

6.7 Summary

We have defined a complete set of tools that can be used to design a feed-forward neural network from scratch, given only the number of layers that should be used. Although additional computations are used to refine the number of neurons, these computations are overall much less expensive than the alternative method of training many models and

Table 6.17: Different starting networks lead to similar networks in our experiments. The output of Algorithm 3 for a network with 600 neurons on all its hidden layers, is very similar to the output of Algorithm for a different network in Table 6.16.

Layer (i)	n_i	
	Input to Algorithm 3	Output of Algorithm 3
1	600	600
2	600	518
3	600	448
4	600	389
5	600	344
6	600	300
7	600	267
8	600	238
9	600	184
10	10	10
$\Sigma(n_i)$	5,410	3,298

choosing the one with best accuracy. Results on sample image and non-image datasets demonstrate that our method results in small networks with high accuracies. By choosing the number of neurons wisely, we avoid both over-fitting and under-fitting of the data and therefore, achieve low generalization errors. This enables practitioners to effectively utilize compact neural networks for real-world applications. We also provided a straightforward method for squeezing networks that are already trained. Our method identifies and discards redundancies in the trained networks, leading to compact networks, sometimes with better accuracies.

In the next chapter, we study an unsupervised learning method, Gaussian graphical models, and provide mathematical tools for their interpretation.

Chapter 7: Interpreting Gaussian Graphical Models

7.1 Introduction

In this chapter, we study the problem of interpretation in the context of unsupervised machine learning, specifically for Gaussian graphical models.[1]

Gaussian graphical models (GGM) have been successful at discovering patterns of dependencies among variables in data for application areas such as gene interaction networks, information pathways of the brain, and climate models [Barabasi and Oltvai, 2004, Dobra et al., 2004, Huang et al., 2009, Smith et al., 2011, Zerenner et al., 2014]. Yet in these practical applications of scientific discovery, often a subset of the samples in the data would produce different results. In many cases, this variability is due to the fact that some data samples are produced by a different underlying process than the other samples. For example, in brain imaging data, some subjects may be on medication that affects brain pathways, and in climate models, microclimate regions will exhibit different climate patterns. To facilitate meaningful pattern discovery, the end-user or analyst who is trying to understand the patterns in the data needs to explore how the data affects the learned GGM and vice-versa, how the structure of the GGM relates to the data. We introduce Interpretable Diverse Gaussian Graphical Model learning (iDGGM), which is a method to find data subsets that produce the most variation in the learned GGM structure and

[1]This work has been published as "Learning Diverse Gaussian Graphical Models and Interpreting Edges" [Yousefzadeh et al., 2019] at the SIAM International Conference on Data Mining (SDM19).

interpret that structure with respect to the data.

When using GGM learning for scientific discovery, the end-user has expert knowledge about the data, even before applying any learning algorithms. For machine learning to be used in scientific discovery, transparency in the learning algorithms is critical [Amershi et al., 2011, Kapoor et al., 2010]. iDGGM provides transparency through (1) exploring various models that represent various subsets of data; and (2) explaining which data samples contribute most to edges in the learned GGM.

Other work in learning GGMs from scientific data has focused on reducing the effect of outlier samples [Liu et al., 2012], finding robust edges [Liu et al., 2010b], partitioning data along meta-data features [Liu et al., 2010a], and finding changes in structure between partitions of data [Mohan et al., 2012]. Our approach is novel in identifying data samples that most affect the learned structure. We demonstrate that our approach identifies outlier samples, separates groups of data samples that have been mixed together, evaluates robustness of the learned structure, and provides interpretations of variable dependencies in terms of data samples.

Two major contributions of this chapter are methods to (1) identify a subset of data that if removed, produces the most different GGM; and (2) identify a subset of data that if removed would eliminate a given edge from the GGM. For each method, an optimization problem is defined that can be solved efficiently by gradient-based algorithms. Formulations are then evaluated on data from ingredient networks in online recipes [Ahn et al., 2011] to give insight about relationships among ingredients, as well as to demonstrate the ability to evaluate the robustness of the GGM and individual edges within the GGM. Then insights are demonstrated about the geochemical signatures observed by the Mars rover [Wiens et al., 2012]. By bringing transparency and interpretability, practitioners are

enabled to create and use models that they are confident and insightful about.

7.2 Problem Statement

7.2.1 Preliminaries: Notation

In our formulations, scalars and vectors are in lower case and matrices are in upper case. Bold characters are used for vectors and matrices. Subscripts of vectors and matrices denote their dimensions, and demonstrated when a vector or matrix is first introduced. The index for a particular element of a matrix or vector is shown inside brackets.

7.2.2 Preliminaries: Gaussian graphical models

A Gaussian graphical model (GGM) [Friedman et al., 2008, Meinshausen and Bühlmann, 2006, Zhao et al., 2012] estimates a sparse set of conditional dependency relationships (partial correlations) among a set of variables. A GGM is estimated from a data matrix $\boldsymbol{X}_{n_x,n_v}$, where columns represent the n_v variables and rows represent the n_x observations or samples. Formally, the GGM represents the precision matrix, denoted by $\boldsymbol{\Theta}_{n_v,n_v}$, which is the inverse of the covariance matrix of data, denoted by $\boldsymbol{\Sigma}_{n_v,n_v}$. A partial correlation of 0 indicates that the pair of variables are conditionally independent given all other variables in the system. A graphical model is a visual representation of the precision matrix in which an edge is drawn connecting two vertices in the graph, if and only if the partial correlation between the corresponding variables is not zero.

Sparsity of a GGM is of significant importance, because it reveals the most important relationships among the variables in the dataset. Ideally the covariance matrix is invertible and the precision matrix is sparse. Invertibility can usually be ensured by pre-processing the data. However, several phenomena can cause the calculated precision matrix to be

non-sparse [Lam and Fan, 2009, Yuan and Lin, 2007]. For example, the number of samples in the dataset might be too few for the model to capture the underlying sparsity in the GGM, or there might be latent variables that are not measured in the dataset, or there might be outliers in the dataset.

One way to obtain a sparse precision matrix is to calculate a sparse *approximation* to the inverse. These numerical methods are diverse and usually depend on the particular cause of non-sparsity. One widely studied approach is to solve the optimization problem

$$\widehat{\mathbf{\Theta}} = \arg\max_{0 \preceq \mathbf{\Theta}} \quad \log\det \mathbf{\Theta} - \mathrm{tr}(\mathbf{\Sigma}\mathbf{\Theta}) - \lambda \|\mathbf{\Theta}\|_1, \tag{7.1}$$

where $\|.\|_1$ is the ℓ_1 norm operator, and $\mathbf{\Theta}$ is restricted to be positive semi-definite. The sparsity of $\widehat{\mathbf{\Theta}}$ increases with the regularization parameter $\lambda > 0$, which can be any non-negative real number. The first two terms in equation (7.1) are the maximum likelihood estimation (MLE) formula, where its first-order optimality condition requires the $\widehat{\mathbf{\Theta}}$ to become equal to the inverse of covariance matrix.

7.2.3 Related work about computation of precision matrix

The difference between the sparse precision matrix obtained via regularization and the inverse of the covariance matrix has been studied as a function of the number of observations, number of variables, and regularization parameter [Banerjee et al., 2008, Raskutti et al., 2009].

Consistency in approximating the sparse precision matrix is also essential. Numerous studies have investigated the GGMs from the sparsity point of view. This has led to the term "sparsistency", which refers to achieving consistent sparsity in GGMs [Lam and Fan, 2009, Rothman et al., 2008].

The issue of latent variables have also been studied, notably in [Chandrasekaran

et al., 2010], and the most common approach is to decompose the precision matrix as a summation of a sparse matrix and a low-rank matrix. Chandrasekaran et al. [2010] have also proved that both matrices are related to the direct inverse of the covariance matrix.

7.2.4 Objectives: Diverse GGMs and interpreting edges in the graph

Our main objective is to explore the diversity of GGMs that can be learned from subsets of a given data set. Formally stated, we need to find a subset of data points such that the GGM obtained from the subset is maximally different from the GGM obtained from the entire dataset, $\widehat{\boldsymbol{\Theta}}$. We are also interested in finding the relationship between each edge in the GGM and the individual data points. This will shed light on how robust the learned GGM is with respect to the individual observations. For example if an edge in the GGM exists because of just a few data points in the dataset, then it is desirable to identify such data points and ensure their validity and incorruptness.

Therefore, we have two goals in this chapter:

1. Finding the subset of rows in $\boldsymbol{X}$ that yields a GGM maximally different from $\widehat{\boldsymbol{\Theta}}$ obtained from the entire dataset. (Section 7.3)

2. Finding the subset of rows in $\boldsymbol{X}$ that is related to each edge in the GGM. (Section 7.4)

7.2.5 Weighted-sample GGM

We first assign binary weights to each of the data points (rows of matrix $\boldsymbol{X}$). Having a zero weight will translate to exclusion of the corresponding observation from the data matrix and vice versa.

The weighted mean and covariance are maximum likelihood estimators of normal distributions with different reliabilities of estimates for each sample. Thus, the weights

are equivalent to judgments of reliability of the measurement. This is appropriate for estimating a graph with some high-variance samples down-weighted (or masked) and for discounting the contribution of samples that are less important while increasing the weight of the contribution for the most important samples.

Given n_x samples, we define the weight vector $\boldsymbol{w}_{n_x,1}$ with one element for each observation in the data

$$\boldsymbol{w}_{n_x,1} \in \{0,1\}^{n_x}. \tag{7.2}$$

We ultimately want the weight vector to be binary as in (7.2), but sometimes we relax the binary requirement and allow

$$0 \leq \boldsymbol{w} \leq 1. \tag{7.3}$$

The weight corresponding to a data point signifies importance of the data point when it is close to 1, and signifies unimportance when it is close to zero.

We denote the weighted dataset by $\boldsymbol{X}^w$ and compute it via

$$\boldsymbol{X}^w_{n_x,n_v} = diag(\boldsymbol{w})\ \boldsymbol{X}, \tag{7.4}$$

where $diag()$ is the diagonal operator. The covariance matrix and the GGM associated with $\boldsymbol{X}^w$ are denoted as $\boldsymbol{\Sigma}^w_{n_v,n_v}$ and $\boldsymbol{\Theta}^w_{n_v,n_v}$, respectively. We can obtain the covariance of the weighted data by

$$\boldsymbol{\Sigma}^w = \frac{1}{n^w_x - 1}\Big[\boldsymbol{X}^T diag(\boldsymbol{w})^2 \boldsymbol{X} - \frac{1}{n^w_x}\boldsymbol{X}^T \boldsymbol{w}\boldsymbol{w}^T X\Big] = \frac{1}{n^w_x - 1}\Big[\boldsymbol{X}^T \widehat{\boldsymbol{W}}\boldsymbol{X}\Big], \tag{7.5}$$

where $n^w_x = \|\boldsymbol{w}\|_1$, and $\widehat{\boldsymbol{W}}_{n_x,n_x} = diag(\boldsymbol{w})^2 - \frac{1}{n^w_x}\boldsymbol{w}\boldsymbol{w}^T$.

In some instances, one might want a lower bound on the sum of elements in $\boldsymbol{w}$. For example, we might want to make sure the number of nonzero elements in $\boldsymbol{w}$ is not less than a certain threshold. We impose this via a constraint

$$\alpha\ n_x \leq \mathbb{1}_{1,n_x}\ \boldsymbol{w}, \tag{7.6}$$

where $0 < \alpha < 1$ and $\alpha = 1$ corresponds to all elements of $\boldsymbol{w}$ being equal to 1, i.e. $\boldsymbol{X}^w = \boldsymbol{X}$. Similarly, $\alpha = 0$ corresponds to all elements of $\boldsymbol{w}$ being equal to zero, i.e. $\boldsymbol{X}^w = 0 * \boldsymbol{X}$. We have examined different values for α in our numerical experiments, and observed that this constraint is not always binding.

7.3 Finding the Maximally Different GGM

For our first goal, we take two different approaches. First we consider the case where inverting the covariance matrix leads to a sparse precision matrix. Then we consider the case where the precision matrix is obtained via numerical optimization and regularization. In our numerical results, we use NLopt which is a free/open-source library for nonlinear optimization [Johnson, 2014].

7.3.1 Precision matrix obtained from direct inverting

We define the objective function as

$$\max_{\boldsymbol{w}} \|(\widehat{\boldsymbol{\Theta}} - \widehat{\boldsymbol{\Theta}}^w) \odot \boldsymbol{F}\|_1, \tag{7.7}$$

where $\widehat{\boldsymbol{\Theta}}$ has been previously obtained from the entire dataset, $\widehat{\boldsymbol{\Theta}}^w$ is a function of $\boldsymbol{w}$, $\odot$ is the Hadamard product and $\boldsymbol{F}_{n_v,n_v}$ is a stencil with ones in its strict upper triangular and zeros elsewhere. The rationale behind the stencil is that the precision matrix is symmetric by nature, hence, we can work with the upper triangular portion. The difference in diagonal entries of precision matrices is not reflected in the GGM and is not of interest for our objective.

We maximize (7.7) subject to constraints (7.3) and (7.6). If we want the weights to be binary, constraint (7.2) shall replace constraint (7.3). We will later show how the binary constraint can be satisfied by via regularization.

Note that we need to calculate the inverse of $\boldsymbol{\Sigma}^w$ obtained by equation (7.5). We assume full column rank for the observed data $\boldsymbol{X}$ which can be ensured via preprocessing of the data. $\widehat{\boldsymbol{W}}$, however, is rank deficient because the vector whose entries are the inverse of entries of $\boldsymbol{w}$ is one of its eigenvectors, corresponding to an eigenvalue of zero.

Defining $\boldsymbol{A} = \boldsymbol{X}^T\ diag(\boldsymbol{w}_{n_x,1})^2\ \boldsymbol{X}$, $\boldsymbol{u} = -\frac{1}{n_x^w}\boldsymbol{X}^T\boldsymbol{w}$ and $\boldsymbol{v} = \boldsymbol{X}^T\boldsymbol{w}$, we obtain

$$\boldsymbol{\Sigma}^w = \frac{1}{n_x^w - 1}\Big[\boldsymbol{A} + \boldsymbol{u}\boldsymbol{v}^T\Big]. \tag{7.8}$$

Then, using the Sherman-Morrison-Woodbury formula [Golub and Van Loan, 2012, Hager, 1989], we can calculate the precision matrix as

$$\widehat{\boldsymbol{\Theta}}^w = (\boldsymbol{\Sigma}^w_{n_v,n_v})^{-1} = (n_x^w - 1)\Big[\boldsymbol{A}^{-1} - \frac{\boldsymbol{A}^{-1}\boldsymbol{u}\ \boldsymbol{v}^T\boldsymbol{A}^{-1}}{1 + \boldsymbol{v}^T\boldsymbol{A}^{-1}\boldsymbol{u}}\Big]. \tag{7.9}$$

$\boldsymbol{A}^{-1}$ exists, since we have assumed $\boldsymbol{X}$ has full column rank

$$\boldsymbol{A}^{-1} = \boldsymbol{X}^{-1}\ diag(\boldsymbol{w})^{-2}\ \boldsymbol{X}^{-T}, \tag{7.10}$$

where $\boldsymbol{X}^{-1} = (\boldsymbol{X}^T\ \boldsymbol{X})^{-1}\ \boldsymbol{X}^T$, which is the left inverse of $\boldsymbol{X}$. $(\boldsymbol{X}^T\ \boldsymbol{X})^{-1}$ is guaranteed to exist when $\boldsymbol{X}$ has full column rank.

It is also important to notice the denominator in equation (7.9) which we need nonzero. If we replace the $\boldsymbol{A}^{-1}$ in the denominator, we will obtain $1 + \boldsymbol{v}^T\ \boldsymbol{A}^{-1}\ \boldsymbol{u} = 1 - \frac{1}{n_x^w}\boldsymbol{w}^T\ \boldsymbol{X}\ \boldsymbol{X}^{-1}\ diag(\boldsymbol{w})^{-2}\ \boldsymbol{X}^{-T}\ \boldsymbol{X}^T\ \boldsymbol{w}$. It is easy to see that the denominator can become zero only if $\boldsymbol{X}\ \boldsymbol{X}^{-1}$ is diagonal, but, this is impossible for a non-square $\boldsymbol{X}$ that has full column rank. However, for the cases where $\boldsymbol{X}$ is square and invertible, we can deal with the problem much easier than the current approach. Therefore, assuming that $\boldsymbol{X}$ is non-square and has full column rank, the denominator can never become zero, and equation (7.9) can be used to obtain the precision matrix.

From the optimization point of view, equation (7.9) considerably facilitates our computations by enabling us to analytically calculate the derivatives of the objective

function with respect to $\boldsymbol{w}$. The corresponding formulae are provided in Section 7.5.1. Relying on the derivatives, we use a gradient-based nonlinear optimization algorithm to find the weights that produce the most diverse GGM.

Equations (7.3) and (7.6) are linear, which is desirable. But, when the binary constraint (7.2) is imposed on the weights, the optimization becomes a mixed-integer nonlinear problem, which requires a difficult class of optimization algorithms. An alternative approach to impose the binary constraint is to add a regularization term to the objective function as in

$$\max_{w} \|(\widehat{\boldsymbol{\Theta}} - \widehat{\boldsymbol{\Theta}}^{w}) \odot \boldsymbol{F}\|_1 - \lambda_2 \|\boldsymbol{w} \odot (\mathbb{1} - \boldsymbol{w})\|_2^2, \tag{7.11}$$

where λ_2 is a positive penalty coefficient, penalizing the objective function when weights are not binary. We find the regularization approach to be more effective in solving the problem, because our optimization problem is nonlinear. At the start of optimization process, we use a small positive value for λ_2 and increase its value, tracing a homotopy path until all the weights are binary. At each iteration as we increase the λ_2, we use the solution obtained from the previous λ_2 as the starting point for the next optimization.

7.3.2 Precision matrix obtained from regularization

In this case, the most obvious approach is to use the optimization problem used to obtain $\widehat{\boldsymbol{\Theta}}$ in the first place, and add a regularization term in the objective function to promote maximal difference between $\widehat{\boldsymbol{\Theta}}^{w}$ and $\widehat{\boldsymbol{\Theta}}$. For example, if equation (7.1) is used to obtain $\widehat{\boldsymbol{\Theta}}$, the objective function we solve to find the optimal weights is

$$\begin{aligned} \widehat{\boldsymbol{\Theta}}^{w} = \max_{\boldsymbol{w}, 0 \preceq \boldsymbol{\Theta}^{w}} \quad & \log\det \boldsymbol{\Theta}^{w} - \operatorname{tr}(\boldsymbol{\Sigma}^{w}\boldsymbol{\Theta}^{w}) \\ & - \lambda \|\boldsymbol{\Theta}^{w}\|_1 + \lambda_1 \|(\widehat{\boldsymbol{\Theta}}^{w} - \widehat{\boldsymbol{\Theta}}) \odot \boldsymbol{F}\|_1 - \lambda_2 \|\boldsymbol{w} \odot (\mathbb{1} - \boldsymbol{w})\|_2^2, \end{aligned} \tag{7.12}$$

subject to constraints (7.3) and (7.6), with $\lambda_1, \lambda_2 \geq 1$.

When optimizing this problem, we can require the weights to be binary, or we can relax that requirement and allow the weights to be continuous between 0 and 1, by eliminating the last term in equation (7.12).

7.4 Interpreting Edges in the GGM

For our second goal, we want to know how each edge in the graph is related to the rows of $\boldsymbol{X}$. In other words, we would like to find a subset of rows such that eliminating them from $\boldsymbol{X}$ causes a specific edge to disappear from the graph, while the rest of graph remains fairly intact.

Here, we have three terms to minimize. The first term is the value of the edge we want to eliminate, represented by element $\widehat{\Theta}^w_{[i,j]}$ in the weighted precision matrix. The second term is the difference between the weighted precision matrix and the original one, excluding the element of interest $\widehat{\Theta}^w_{[i,j]}$. The last term is the penalty term to ensure weights are binary. We obtain

$$\min_{\boldsymbol{w}} \|\widehat{\Theta}^w_{[i,j]}\|_1 + \frac{1}{\lambda_1}\|(\widehat{\boldsymbol{\Theta}} - \widehat{\boldsymbol{\Theta}}^w) \odot \hat{\boldsymbol{F}}\|_1 + \lambda_2 \|\boldsymbol{w} \odot (\mathbb{1} - \boldsymbol{w})\|_2^2, \tag{7.13}$$

where λ_1 and λ_2 are positive penalty parameters that should be greater than or equal to 1. $\hat{\boldsymbol{F}}$ is equal to $\boldsymbol{F}$, except that its $[i, j]$ element is also zero. $\widehat{\boldsymbol{\Theta}}^w$ is obtained from equation (7.9), and $\widehat{\boldsymbol{\Theta}}$ is obtained from inverting the covariance matrix, $\boldsymbol{\Sigma}$, or from a numerical optimization problem such as (7.1).

As discussed before, we start the optimization with small values of λ_1 and λ_2 and increase them gradually, until we find the optimal binary weights. At each iteration, we use the optimal solution as the starting point for the next iteration.

7.5 Solving the optimization problems

Our optimization problems formulated in previous sections are mixed-integer nonlinear and non-convex. Our integer variables are strictly binary, while the rest of the variables are continuous. We can compute the derivatives of all components of the formulation with respect to all variables including the binary ones.

7.5.1 Calculating the derivatives

In order to use a gradient-based optimization method, we need to calculate the derivative of weighted precision matrix, equation (7.9) with respect to the weight vector. First, we start with some preliminaries:

$$\dot{\boldsymbol{v}}_{n_v,n_x} = \frac{d}{d\boldsymbol{w}}(\boldsymbol{v}_{n_v,1}) = \boldsymbol{X}^T, \tag{7.14}$$

and similarly, $\dot{\boldsymbol{u}}_{n_v,n_x}$ can be derived.

Since we have to deal with 3-dimensional matrices, we use tensor notation here. We denote $\times_k$ as the k-mode tensor product, and derive the derivative of $\boldsymbol{A}^{-1}$

$$\dot{\boldsymbol{A}^{-1}}_{n_v,n_v,n_x} = \frac{d}{d\boldsymbol{w}}(\boldsymbol{A}^{-1}_{n_v,n_v}) = \boldsymbol{A}^{-1}_{n_v,n_v} \times_v \dot{\boldsymbol{A}}_{n_v,n_v,n_x} \times_v \boldsymbol{A}^{-1}_{n_v,n_v}, \tag{7.15}$$

in terms of the derivative of $\boldsymbol{A}$, which can be analytically derived and is left for the reader.

Then, using the chain rule and each of the items derived above, we can compute the derivative of the weighted precision matrix in equation (7.9) with respect to the weight vector, which will yield a rank 3 tensor.

7.5.2 An Alternative method for optimizing the binary variables

We explained before that binary variables can be optimized via regularization, as in the last term added to equations (7.11) and (7.13). Here, we describe an alternative

methods for optimizing these variables, using a binary algorithm.

The benefit of using the binary algorithm is that it can be faster than solving the problem with regularization (nonlinear penalties on the binary variables). Moreover, the running time and the quality of the minimizer found by the regularization approach is sensitive to the choice of starting point, while the binary algorithm does not have that sensitivity, and can find a minimizer within a reliable timeframe. However, the quality of solutions found by the binary algorithm is usually inferior to the solution found by the regularization approach.

We recommend using the solution found by the binary algorithm as a fast answer, or as a good starting point for solving the regularization problem. In our experiments, the regularization approach was usually able to improve the solution found by the binary algorithm, but the improvement is sometimes marginal. Both approaches have no guarantee to find the global minimizer.

The third alternative for solving the problem is to use a branch and bound algorithm [Lawler and Wood, 1966]. Using a branch and bound approach may take longer than the other two methods, however, it entails a thorough search of the binary space. In our experiments, we did not observe any necessity for using a branch and bound algorithm. But, it can be considered a viable approach when working with small number of variables.

7.5.2.1 Binary algorithm

Considering the objective function (7.7), subject to constraints (7.2) and (7.6), we can start the optimization with $\boldsymbol{w}_0 = \mathbb{1}_{1,n_x}$ which makes the objective function equal to zero, and satisfies the constraints. Then, we proceed by finding the weights that should be set to zero. We would like this change in $\boldsymbol{w}$ to achieve the maximal change in the

precision matrix, i.e. equation (7.7). We only change the weights from 1 to 0, or the opposite; therefore constraint (7.2) will always be satisfied and the solution at hand will always be feasible. Constraint (7.6) puts a limit on the total number of weights that can be set to zero, so when it becomes binding, the binary algorithm stops.

At each iteration, we will only make a binary change for one of the optimization variables, switching the value of one element in $\boldsymbol{w}$ from 0 to 1 or vice versa. That binary change will be performed on the element in $\boldsymbol{w}$ that leads to maximal change in the objective function. In order to find the element in $\boldsymbol{w}$ that has the highest potential to change the (7.7), we define a new quantity named potential, for each element of $\boldsymbol{w}$. This potential is an estimate of how much the weighted GGM will change compared to the original one, as the result of switching elements of $\boldsymbol{w}$. This estimate is based on the derivatives of $\widehat{\boldsymbol{\Theta}}^{w}$ with respect to $\boldsymbol{w}$, denoted by $\dot{\widehat{\boldsymbol{\Theta}}}^{w}_{n_v,n_v,n_x}$ which is a tensor of rank 3. The potential vector, $\boldsymbol{p}_{1,n_x}$ has the same size as $\boldsymbol{w}$ and can be computed by

$$\begin{aligned}\boldsymbol{p} = \mathbb{1}_{1,n_v} \times_{n_v} \left(|\dot{\widehat{\boldsymbol{\Theta}}}^{w}_{n_v,n_v,n_x}| \odot \left(\mathbb{1}_{1,n_x} \times_1 \left(\boldsymbol{F}_{n_v,n_v,1} \odot (\widehat{\boldsymbol{\Theta}}^{w} == \widehat{\boldsymbol{\Theta}}) \right) \right) \right) \times_{n_v} \mathbb{1}_{1,n_v} \\ + \left(\mathbb{1}_{1,n_v} \times_{n_v} \left(\dot{\widehat{\boldsymbol{\Theta}}}^{w}_{n_v,n_v,n_x} \odot \left(\mathbb{1}_{1,n_x} \times_1 \left(\boldsymbol{F}_{n_v,n_v,1} \odot sign(\widehat{\boldsymbol{\Theta}} - \widehat{\boldsymbol{\Theta}}^{w}) \right) \right) \right) \times_{n_v} \mathbb{1}_{1,n_v} \right) \\ \odot (2 \odot \boldsymbol{w} - \mathbb{1}_{1,n_x}), \quad (7.16)\end{aligned}$$

where == is a binary element-wise equality operator.

Equation (7.16) is a summation of two terms. The first term considers the edges in the weighted GGM that have the same value in the original GGM. The second term considers all other edges.

After calculating the $\boldsymbol{p}$, we find its element with the highest value, $p_{[i]}$. We then switch the binary value of $w_{[i]}$. We continue this process until constraint (7.6) becomes binding, or until there is no switch that can increase the value of the objective function.

It is possible for an element to have a high potential value, while switching its binary value leads to decreasing the objective function. Therefore at each iteration, after identifying the element with the highest potential, we make sure the binary step will actually lead to an increase in (7.7), and if it does not, we do not change the weight for that element but instead proceed to the element with the next highest potential.

The potential vector depends on the derivatives of $\widehat{\boldsymbol{\Theta}}^{w}$, which changes each time we make a change to $\boldsymbol{w}$. Hence, we can update the derivatives and recalculate $\boldsymbol{p}$ at each iteration. But, changing one element of $\boldsymbol{w}_{[i]}$ usually does not significantly affect the derivatives with regard to elements other than $\boldsymbol{w}_{[i]}$, and consequently the $\boldsymbol{p}$ usually does not change significantly in two consecutive iterations. Therefore, one can update the $\boldsymbol{p}$ only every few iterations, in order to speed up the process. Each time a binary variable is changed, it will be removed from the variable space. Therefore, the algorithm will take at most n_x iterations to stop.

7.5.2.2 Notes regarding the binary algorithm

The choice of $\boldsymbol{w}_0 = \mathbb{1}$ is recommended especially when α is relatively large, but other choices for $\boldsymbol{w}_0$ are possible, too. For example, one can relax the binary variables to be continuous between 0 and 1, find the optimal solution to the relaxed problem, and then round them to the closest integer. If rounding violates the constraint (7.6), we set the $(1-\alpha)\, n_x$ of smallest weights in the relaxed solution to zero, and set the rest of weights to 1. This may speed up the process, when α is small.

The potential vector is calculated using the derivatives of the objective function in the continuous space, but the binary algorithm searches for the optimal solution in the binary space. Moreover, the objective function is non-convex, hence, the directions of

derivatives do not necessarily point out to the location of optimal solution. Therefore, $\boldsymbol{p}$, used in the decisions of the binary algorithm, might not always be an accurate estimate of how the objective function would change in the binary space.

Overall, we conclude that the binary algorithm is not as capable as the regularization approach in dealing with the non-linearity and non-convexity aspects of the problem, because it takes binary steps based on a proxy measure, $\boldsymbol{p}$, derived from the continuous space. Nevertheless, we expect it to be effective for at least a subset of variables, and we recommend it as a useful tool to obtain a fast solution in a reliable timeframe.

7.6 Numerical Results

In order to demonstrate the effectiveness of our formulations, we first apply our methods on a dataset about cooking recipes. Later, we will explore a real world application about the Mars rover.

7.6.1 Cooking Recipes

Ahn et al. [2011] gathered 56,498 distinct recipes, categorized under 11 classes as listed in Table 7.1. There are 380 distinct ingredients in the dataset. We organize the data in a matrix with 56,498 rows and 380 columns. The matrix is binary, with ones for each ingredient that is used for a recipe.

7.6.1.1 General trends

We first calculate the covariance and precision matrix for the entire dataset, and also for the recipes in each class. Tables 7.1 and 7.2 show the most strongly positive and negative edges in each of the resulting GGMs, respectively.

Table 7.1: Strongest positive edges in the resulting GGM for each class of recipes

Class	**Ingredient 1**	**Ingredient 2**
African	Juniper berry	Feta cheese
East Asian	Pecan	Clove
Eastern European	Watercress	Salmon
Latin American	Roasted sesame seed	Galanga
Middle Eastern	Tamarind	Squid
North American	Lilac flower oil	Gelatin
Northern European	Blue cheese	Tarragon
South Asian	Chicory	Okra
Southeast Asian	Pecan	Orange flower
Southern European	Papaya	Kidney bean
Western European	Jamaican rum	Oatmeal
All classes	Lilac flower oil	Gelatin

Table 7.2: Most strongly negative edges in the resulting GGM for each class of recipes

Class	**Ingredient 1**	**Ingredient 2**
African	Rum	Bourbon whiskey
East Asian	Cocoa	Chamomile
Eastern European	Champagne wine	Wasabi
Latin American	Kale	Octopus
Middle Eastern	Star anise	Tequila
North American	Condiment	Mandarin
Northern European	Blue cheese	Cheddar cheese
South Asian	Parsnip	Roasted almond
Southeast Asian	Tangerine	Macaroni
Southern European	Seaweed	Sunflower oil
Western European	Papaya	Oatmeal
All classes	Strawberry juice	Strawberry

7.6.1.2 Robustness of the GGM

We consider the GGM obtained from the entire data, and find the subset of data that yields a maximally different GGM. To perform this task, we explore different values for α in constraint (7.6). The results are presented in Table 7.3. It can be observed that decreasing α allows us to find a smaller subset and consequently obtain a GGM more

different than the GGM obtained from the entire data. We should note that there is a trade-off here, between choosing a small subset and finding a diverse GGM. Allowing the subset to become too small may lead to a sample that is unrepresentative of the data; therefore it is important to explore a range of values for α. We can conclude that the GGM obtained from the entire dataset is robust, since it remains considerably intact, until we allow exclusion of more than 40% of the data.

Table 7.3: Diversity of GGMs obtained from the entire recipe data

α	$\|\boldsymbol{w}\|_1$	Eq. (7.7)	% change in $\|\widehat{\boldsymbol{\Theta}} \odot \boldsymbol{F}\|_1$
0.95	0.95	740	1
0.90	0.92	1,910	2
0.80	0.88	2,177	4
0.60	0.81	3,340	8
0.50	0.55	6,289	43
0.40	0.40	13,319	180
0.20	0.21	65,015	1,109
0.10	0.13	91,614	1,537
0.05	0.05	149,100	2,320

7.6.1.3 Working on a mixture of two groups

Next, we mix the recipes from classes "*East Asian (EA)*" and "*North American (NA)*", and calculate the $\text{GGM}^{\text{EA+NA}}$ corresponding to the mixed data. The number of recipes are 2,512 and 41,524 for the "*EA*" and "*NA*" class, respectively.

We then optimize the weights to obtain the weighted GGM maximally different from the $\text{GGM}^{\text{EA+NA}}$, with different values of α. We note that "*EA*" recipes make up 5.7% of the mixture, and "*NA*" recipes make up 94.3% of it. Results obtained with different values of α are presented in Table 7.4.

Interestingly, when α is set to 95% (the percentage of "*NA*" recipes in the mixture),

Table 7.4: Separating the mixture of "*East Asian*" and "*North American*" recipes. $\Delta\boldsymbol{\Theta}^{\text{NA}} = \|(\widehat{\boldsymbol{\Theta}}^{\text{NA}} - \widehat{\boldsymbol{\Theta}}^{w}) \odot \boldsymbol{F}\|_1$

α	$\|\boldsymbol{w}\|_1$	Eq. (7.7)	$\Delta\boldsymbol{\Theta}^{\text{NA}}$
0.98	0.98	404	1,008
0.95	0.955	846	1,492
0.90	0.92	1,121	309
0.75	0.89	1,171	742
0.50	0.53	3,535	3,662
0.25	0.25	7,810	5,274
0.10	0.10	8,728	8,727
0.05	0.05	251,830	243,650

we end up extracting most of the "*EA*" recipes out of the mixture, and obtain a GGM very close to the GGM^{NA}. But, when α is small, extracting the "*NA*" recipes from the mixture does not seem possible, since the formulation can find GGMs more diverse than the GGM^{EA}. This indicates that small values of α allow the formulation to find subsets that are not representative of any specific cluster within the data.

To explain this, we have to look back at our main goal, which is to explore the diversity of GGMs. By defining the α, we allow a certain percentage of the data to be removed in order to produce a maximally different GGM. When α is large, we tend to extract outliers out of the mix and keep the more coherent portion of the data, because removing one outlier from a large dataset changes the resulting GGM more than removing a regular data point from it. However, when α is small, we tend to keep an incoherent portion of the data that produces a very different GGM. This is to be expected, because more incoherent subset of the data can lead to more different GGMs, and such incoherent subsets are not representative of any specific cluster.

7.6.1.4 Interpreting edges in the graph

Here, we solve the optimization problem (7.13) for all the edges in the GGM^{NA} which is the largest class in the dataset. We find that some edges in the graph are very robust and rely on a considerable portion of the dataset, while some edges rely on very few recipes in the dataset. Table 7.5 shows the most robust and least robust edges in the graph.

Table 7.5: Robustness of edges in the GGM^{NA}

Robustness of edge	**% of data related to edge**	**Ingredient 1**	**Ingredient 2**
Most robust	10.3	Mozzarella Cheese	Fenugreek
Least robust	0.002	Bergamot	Angelica

7.6.1.5 Identifying outliers and corrupt data

We now generate 50 randomly built 10-ingredient recipes and add them to the "*NA*" class, as an example of corrupt data. The GGM^{NA} changes drastically, with 853 new edges appearing in the graph, corresponding to a 112% increase compared to the original graph. This makes our GGM less informative and ambiguous, as the sparsity is fading away.

We then start finding the subsets of recipes corresponding to each of the edges whose strength ranks in the lowest 2 percentile. We observe that all those edges are related to at least 76% of the corrupt data. Using this edge interpretation enables us to identify most of the corrupt data with relatively high confidence. We can also provide clear explanations about the reason behind the removal of corrupt data.

7.6.1.6 Comments on optimization

For this recipe dataset, we observed speed up of about 40 times using our analytic derivative formulas compared to the central difference alternative, which is remarkable.

In many of our objective functions, we have regularization terms that are controlled by coefficients denoted with λ. The optimization problem is much easier to solve when those coefficients are small, and as we increase the regularization coefficients, the problem gradually gets transformed to the state where the optimal solution is binary. In numerical optimization, this technique can be categorized as a "homotopy" or "continuation" method which is known to be effective in solving hard optimization problems [Dunlavy et al., 2005, Mobahi and Fisher III, 2015, Nocedal and Wright, 2006]. We found this "continuation" approach to be effective in the quality of solutions obtained.

7.7 Applications

In a novel application of GGM learning, we apply iDGGM to observations of chemical spectra from the ChemCam instrument on the Mars rover Curiosity (ChemCam data available at `http://pds-geosciences.wustl.edu/missions/msl/chemcam.htm`) [Wiens et al., 2012]. The goal is to identify geological signatures indicating dust layers, surface coatings and thin stratigraphic layers [Lanza et al., 2012]. The GGM gives a good visual summary of the general chemical trends of each rock observed on Mars, while iDGGM allows an analyst to interpret which elements contribute to the discovered patterns.

7.7.1 Geochemical trends in ChemCam observations

In preliminary work by Oyen and Lanza [2016], it has been shown that Gaussian graphical models provide a good visual summary of geochemical trends indicative of sur-

face features of rocks on Mars. The ChemCam instrument onboard the Curiosity rover collects observations of the chemical composition of rocks using laser-induced breakdown spectroscopy (LIBS) [Wiens et al., 2012]. The spectra represent the elements present in sample targets. The laser is fired multiple times in a single location, so that a depth sequence of chemical observations is made for each target. Each laser shot ablates the rock surface, and therefore ChemCam produces a sequence of samples at increasing depth, revealing compositional trends such as coatings and weathering rinds (from interaction with water or atmosphere); and thin stratigraphic layers (from sedimentation or volcanic activity) that give clues about the past environmental conditions of Mars [Lanza et al., 2012].

The spectral response is given as a table of intensity values for each wavelength band for each shot. A typical sequence of shots includes 30 - 150 shots on a fixed location. We model the correlations of rock chemistry among these shots, as measured by the sample covariance matrix calculated from the observed spectra.

For each shot, we initially have 6,144 channels over the UV, VIS, and VNIR spectral ranges. As pre-processing, we remove channels with wavelength above 840nm, set all negative values to zero and normalize the values for each of ChemCam's three component spectrometers separately. After the pre-processing, we obtain a spectral observation consisting of 5,810 wavelength bands between 224nm and 840nm for each LIBS shot. To investigate shot-to-shot correlations, shots are the vertices in the graph while the 5,810 wavelength bands are treated as data samples.

The visual summary from a single GGM only provides a starting point for planetary geologists to explore the observational data collected. The geologists also need to know which elements contribute to the geochemical trends that are represented in the learned

GGM. Therefore, our goal is to present to the user a collection of GGMs produced by various subsets of the data that indicate trends produced by various combinations of elements.

7.7.2 GGM diversity

We consider the spectral data obtained from the rock target Bell Island on Mars. This data has 30 shots and 5,810 wavelength bands as discussed earlier. We examine the diversity of GGMs that can be obtained by selecting subsets of the wavelengths.

We first learn a GGM from all the observed spectral data obtained on this target. The obtained GGM, shown in Figure 7.1, has a relatively dense set of edges.

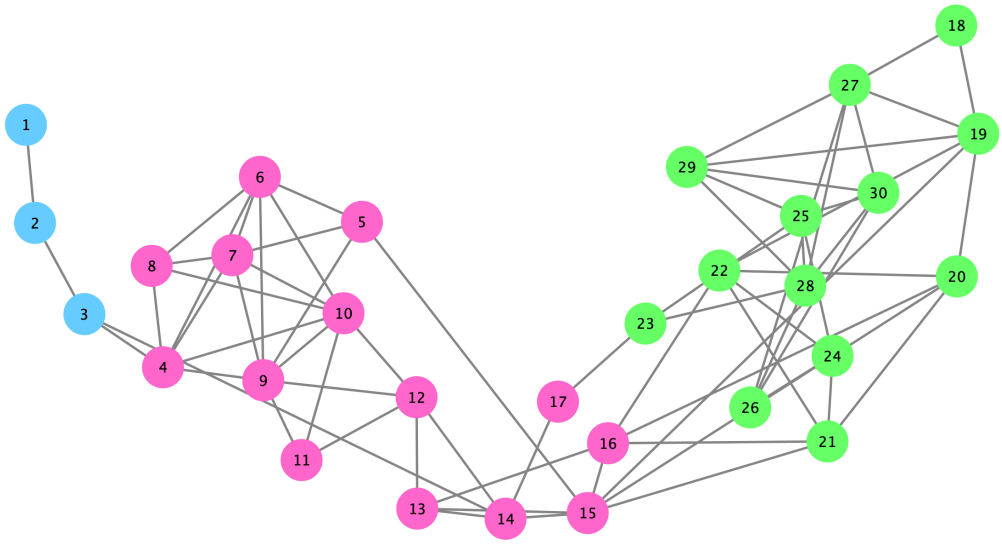

Figure 7.1: GGM obtained from all spectral data gathered by the Curiosity rover at rock target Bell Island on Mars

We then allow 10% of wavelengths to be masked out, by setting $\alpha = 0.9$, in order to find the GGM maximally different from the original GGM in Figure 7.1. The most diverse GGM after masking 10% of wavelengths is shown in Figure 7.2. We can clearly observe that in the diverse GGM, nodes have clustered into groups, revealing the most prominent

relationships among the shots.

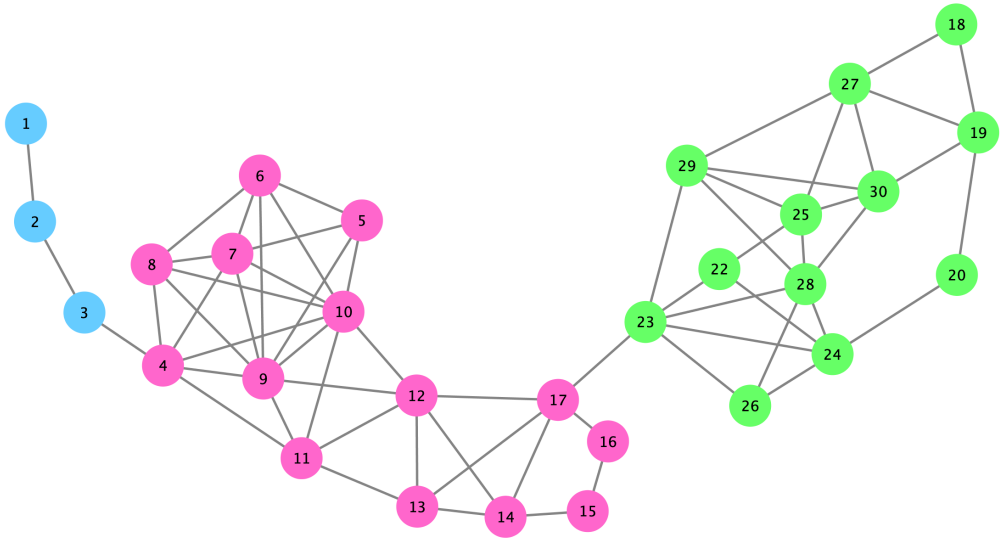

Figure 7.2: GGM obtained from a subset of data, maximally different from the GGM in Figure 7.1

Furthermore, one can identify and study the wavelengths that have been masked out, and possibly increase or decrease the value of α.

This analysis enables the geoscientists to use the model insightfully, and to extract additional information from their data and the model.

7.7.3 Interpreting edges

By solving the optimization problem (7.13) for all edges in the original GGM in Figure 7.1, we find the subset of wavelength bands related to each edge. The most robust edge is between shots 1 and 2, which is related to 26% of wavelengths, while the least robust edge is the one between shots 15 and 20, related to only 0.5% of wavelengths.

Interpretation results allow us to provide explanation about each edge in the graph, and it makes us insightful about details of the model. This can be the subject of more detailed study from the geoscience point of view.

7.8 Summary

1. We defined optimization problems that ultimately make a GGM transparent regarding its underlying dataset. We provided methods to find how diverse a GGM can be with respect to subsets of its dataset. We also showed how each edge in a GGM is related to observations in the dataset.

2. In order to find the importance of each observation in the dataset, we assigned weights to each of the observations which led to a weighted GGM. We treated the weights as optimization variables, and optimized them to obtain certain patterns in the weighted GGM.

3. By imposing binary restrictions on the weights, we were able to find subsets within the data that would lead to a GGM maximally different, or subsets that would make a particular edge appear/disappear in the graph. We have developed a computationally fast and precise formulation to compute the derivatives of a weighted GGM with respect to the weights. This enables us to efficiently solve these otherwise intractable optimization problems.

4. We demonstrated the effectiveness of our formulations by examining a dataset of cooking recipes. We were able to: reveal how the model is related to the dataset; measure the robustness of the GGM as a whole; find the specific observations in the data that cause an edge to appear in the GGM. Through these findings, we could effectively divide mixed data into its original clusters, and identify corrupt/adversarial data mixed into the original dataset. These methods enabled us to provide explanations about different aspects of the GGM and the data itself for the end user of the model.

5. We presented the analysis of ChemCam data from the Mars rover as a real-world application of machine learning. We explained how our methods can facilitate the process in which the gathered data from the red planet is analyzed, and how they enable the end user to interact with the model. Making the GGM transparent and explainable to the end user makes the use of machine learning practical for this application, and allows the user to be insightful and confident about the obtained results.

In the last chapter, we summarize the thesis and our overall findings.

Chapter 8: Conclusions

8.1 What we achieved

In this thesis, we studied a broad range of problems related to machine learning. Our main focus was on deep learning models and their interpretation. We also studied structural design of feed-forward networks and interpretation of Gaussian graphical models.

In Chapter 2, we formulated a neural network and explored some of its computational properties as a function. Since we use the trained networks as functions in our optimization problems, we studied the derivatives of the outputs of the network with respect to inputs, and computed an upper bound on the Lipschitz constant of the network. To have control over the derivatives, we used a tunable error function for activation of neurons. This laid the foundation for us to develop a homotopy algorithm for computing the closest flip points.

In Chapter 3, we proposed the closest flip point as a tool to study the decision boundaries of neural networks. We defined optimization problems to compute exact points on the decision boundaries of trained networks. The flip point closest to a given input is of particular importance, and this point is the solution to a well-posed optimization problem. We developed a homotopy algorithm to compute the closest flip point more effectively compared to off-the-shelf optimization algorithms.

In Chapter 4, we showed how the flip points can be used to systematically interpret and debug trained neural networks, with respect to individual inputs and entire datasets, and to find vulnerability against adversarial attacks. The flip point indicates the least changes in the input required to flip the decision of a trained model, which is a fundamental interpretation question. We demonstrate that flip points can help us to assess the trustworthiness of classifications and identify mistakes made by a model. They can also be used to identify the most and least influential points in the training data in order to reduce training time, identify out-of-distribution points in the data, and investigate overfitting. Using the flip points, we generated synthetic data and were able to improve the accuracy, reshape the decision boundaries and alter certain behaviors adopted by the trained models. PCA analysis of the directions to flip points helped us gain insight about entire datasets or subsets of it. PCA and pivoted QR factorization identified the most and least influential features for classifications. Flip points also help us understand adversarial influence. We demonstrated numerical results for all of these applications.

In Chapter 5, we investigated the decision boundaries of networks in more detail. We developed mathematical tools to systematically investigate the surfaces that define the decision boundaries. We demonstrated these techniques and showed them more useful than previous methods that rely on simplifying assumptions such as local linearity of decision boundaries for models with nonlinear activation functions. We questioned common simplifying assumptions about the decision boundaries and demonstrated that many of them can be misleading. We also showed that flip points reveal the weakest vulnerability of trained models with respect to adversarial attacks.

In Chapter 6, we developed methods to refine the structure of feed-forward networks using matrix conditioning and showed how refining the structure makes the models

compact and reduces the generalization error. Common model selection methods require significant computations amounting to GPU months and years. The methods we developed are straightforward as they use pivoted QR factorization, they are computationally inexpensive compared to the training process, and hence, they are practical for modest applications in the real-world.

Finally in Chapter 7, we studied Gaussian graphical models (GGM), an unsupervised machine learning method. We defined and solved optimization problems that ultimately make a GGM transparent regarding its underlying dataset. We provided methods to find how diverse a GGM can be with respect to subsets of its dataset. We also showed how each edge in a GGM is related to observations in the dataset. In order to find the importance of each observation in the dataset, we assigned weights to each of the observations which led to a weighted GGM. We treated the weights as optimization variables, and optimized them to obtain certain patterns in the weighted GGM. Again, this was not an easy optimization problem. We developed a computationally fast and precise formulation to compute the derivatives of a weighted GGM with respect to the weights. This enables us to efficiently solve these otherwise intractable optimization problems. Making the GGM transparent and explainable to the end user makes the use of machine learning practical for many applications.

Overall in this thesis, we showed that by tailoring proper mathematical tools and optimization methods, we can facilitate solving hard problems that are commonly encountered in machine learning. Whether the problem is interpretation of neural networks and their debugging, structure design of networks, or interpretation of Gaussian graphical models, they all can be viewed as optimization problems. Studying the underlying mathematical problem in each case enabled us to solve the problem more effectively.

In particular, we demonstrated that homotopy algorithms have great potential when they are designed and tailored for specific hard-to-solve optimization problems.

8.2 Future work

There are many directions of research that can be pursued based on this thesis.

Here, we provided several numerical examples in different contexts, using standard datasets on image classification, financial risk, medical decisions, and income prediction. However, it would be very natural to use our methods for models in the real world. Clearly, real datasets come with special complexities and difficulties, which makes the ability to interpret much more essential.

Concerning flip points, we performed PCA and pivoted QR factorization on the directions. It would be interesting to perform more sophisticated analysis, such as nonlinear PCA, auto-encoders, clustering, or other methods, on the flip points and on the directions to them.

We showed that models could learn images from their wavelet coefficients instead of pixels. We believe this is an interesting direction of research to pursue, comparing the convolution with the wavelets with the computations performed via convolutional neural networks.

One of our results in Chapter 4 is that certain points in the training set are more influential in shaping the model. We also showed that not all features are helpful in generalization of a neural network. For image classifications, we have initial indications that wavelet coefficients of images can be used to identify the influential images of training sets via clustering. This would be a significant improvement compared to the method proposed by Birodkar et al. [2019].

We believe that the error function, erf, has great, yet unexplored potential as an activation function for deep neural networks because of its ability to mimic the behavior of functions such as ReLU, sigmoid, and sign. It would be a promising direction of research to study the universal approximability of deep neural network functions that use erf as activation function.

There are several directions to expand our work on decision boundaries. We can study the curvature of decision boundaries via quadratic regression or other methods available for the study of manifolds and multidimensional surfaces. We can also use spectral clustering to study patterns in the flip points in relation to the data.

As demonstrated in the thesis, the study of decision boundaries is directly related with the study of adversarial robustness/attacks. The possibilities of research in this area are numerous including both the practical and theoretical aspects. For example, the weakest vulnerabilities of the models revealed by the flip points can be used to improve the models and make them robust against adversarial attacks. Reduction of the input dimensionality of image classification models from large pixel space to wavelet coefficient space may also make the models less vulnerable.

About our methods for structural design of networks, one could expand the use of matrix conditioning to convolutional networks. By investigating the singular values of the convolutional layers, Sedghi et al. [2019] have shown that there are significant redundancies in those networks. Reducing the structure by discarding such redundancies is a promising direction to refine the networks and to design the networks from scratch, the same way we did for feed-forward networks. Another direction of research is to expand our methods to refine the number of layers in the networks, as well.

Appendix A: Learning Images by wavelet coefficients

We propose learning images by their wavelet coefficients instead of their pixels. The wavelet transformation applies convolutions of various widths to the input data, and the reduction applied by using rank-revealing or pivoted QR decomposition leads to significant compression of the input data. For example, for the MNIST example in Chapter 4, we reduced the input dimension from 784 features to 100, which allowed us to use a much smaller network while achieving near 99% accuracy on the testing set.

Figure A.1 shows the first ship image in the CIFAR-10 training set along with its reconstructions from subsets of wavelet coefficients. With fewer coefficients, the reconstructed image looks less similar to the original image. Nevertheless, the model is able to correctly classify most of the images by learning from those representations. This result may be in agreement with the arguments of Ilyas et al. [2019] that neural networks learn Gaussian representations of images.

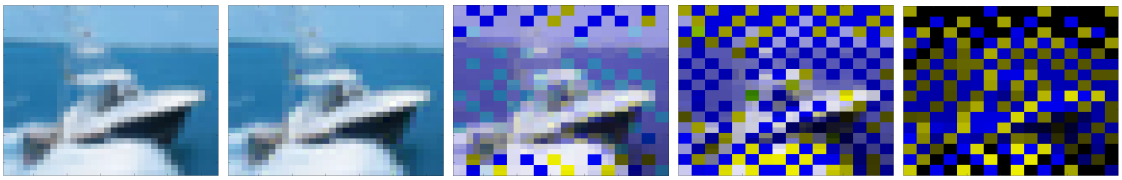

Figure A.1: Reconstruction of an image from a subset of wavelet coefficients leads to different representations. The original image (left), with 4096 wavelet coefficients, is reconstructed using the most significant 2200, 1000, 500, and 200 wavelet coefficients (respectively, from left to right), chosen according to pivoted QR factorization.

This idea of learning from wavelet coefficients is valuable whenever working with

image data and it has implications not only for the training accuracy and generalization error, but for adversarial robustness. The overwhelming weakness of deep learning models against adversarial attacks has often been attributed to the use of all the unnecessary pixels in the training process. As we showed for the FICO Explainable ML Challenge in Chapter 4, when models learn only the important and non-redundant features, their vulnerability against adversarial attacks can be significantly reduced, and their accuracy may increase, as well. A similar argument can be made for learning images with wavelet coefficients. When images are learned with a smaller number of features, it means the adversary has a smaller number of variables to produce adversarial examples. And if all those features have specific meaning to the model, producing adversarial examples may become even harder. Investigating these arguments for image datasets remains for future work.

Appendix B: Information about neural networks used in the numerical results

Here, we provide more information about the models we have trained and used in our numerical results in Section 4.4 and Chapter 5.

For all the models, we have used a tunable error function (defined in Chapter 2) as the activation function of neurons. The tuning parameter σ is constant among the neurons on each layer and is optimized during the training. We have used softmax on the output layer, and cross entropy for the loss function.

B.1 Trained models in Section 4.4

We have used fully connected feed-forward neural networks with 12 hidden layers. The number of neurons for the models used for each dataset is shown in Table B.1.

For the FICO and Credit datasets, we have used networks with 5 hidden layers and number of neurons as described in Table B.2.

B.2 Trained models in Chapter 5

The model used in this Chapter is a fully connected feed-forward neural network with 12 hidden layers. The inputs to the model are 200 wavelet coefficients. The number of neurons for each layer are shown in Table B.3.

Table B.1: Number of nodes in 12-layer neural networks used for interpretation in Section 4.4.

Dataset	MNIST	CIFAR-10	Adult	Cancer (WBCD)
Input layer	100	2304	107	30
Layer 1	500	400	44	40
Layer 2	500	400	39	20
Layer 3	500	400	32	15
Layer 4	400	350	22	10
Layer 5	300	300	20	5
Layer 6	250	250	15	5
Layer 7	250	250	12	5
Layer 8	250	250	8	5
Layer 9	200	200	5	5
Layer 10	150	150	4	5
Layer 11	150	150	8	5
Layer 12	100	100	6	5
Output layer	10	2	2	2

Table B.2: Number of nodes in neural networks trained on financial data sets used in Section 4.4.

Dataset	FICO	Credit default
Input layer	20	28
Layer 1	13	14
Layer 2	9	9
Layer 3	6	8
Layer 4	5	8
Layer 5	4	7
Output layer	2	2

Table B.3: Number of nodes in neural network used for the restricted CIFAR-10 dataset used in Chapter 5.

Data set	Restricted CIFAR-10
Input layer	200
Layer 1	700
Layer 2	600
Layer 3	510
Layer 4	440
Layer 5	375
Layer 6	325
Layer 7	285
Layer 8	250
Layer 9	215
Layer 10	160
Layer 11	100
Layer 12	40
Output layer	2

Appendix C: Pivoted/Rank-revealing QR factorization

Rank-revealing QR factorization (RR-QR), developed by Chan [1987], and also pivoted QR factorization [Golub and Van Loan, 2012] are algorithms that decompose the m by n matrix A, by computing a column permutation and a QR factorization. Pivoted QR, first presented by Businger and Golub [1965], is conceptually similar to RR-QR, but it uses a lower-cost strategy in ordering the permutation matrix. We use this method of matrix factorization frequently in this thesis, so here, we formally explain the RR-QR (which has more details) using two references: Chan [1987] and O'Leary [2009].

The RR-QR decomposition is specified by

$$[\boldsymbol{Q}, \boldsymbol{R}, \boldsymbol{P}] = \text{RR-QR}(\boldsymbol{A}),$$

which computes an orthogonal matrix $\boldsymbol{Q}_{m,n}$, an upper-triangular matrix $\boldsymbol{R}_{n,n}$, and a permutation matrix $\boldsymbol{P}_{n,n}$ so that

$$\hat{\boldsymbol{A}}\boldsymbol{P} = \boldsymbol{Q}\boldsymbol{R}.$$

The RR-QR algorithm computes the $\boldsymbol{R}$ matrix, partitioned as

$$\boldsymbol{R} = \begin{bmatrix} \boldsymbol{R}_{11} & \boldsymbol{R}_{12} \\ 0 & \boldsymbol{R}_{22} \\ 0 & 0 \end{bmatrix},$$

where $\boldsymbol{R}_{11}$ is $p \times p$ and upper triangular and $\boldsymbol{R}_{22}$ is $(n-p) \times (n-p)$ and upper triangular. It computes the factorization such that $\|\boldsymbol{R}_{22}\|$ is small, relative to the main diagonal

elements of $\boldsymbol{R}_{11}$.

We can verify that σ_{p+1}, the $(p+1)$th singular value of $\boldsymbol{A}$, is less than $\|\boldsymbol{R}_{22}\|$. It follows that if $\|\boldsymbol{R}_{22}\|$ is small, then $\boldsymbol{A}$ has at least $(n-p)$ small singular values, indicating its numerical rank deficiency. Hence, the numerical rank of $\boldsymbol{A}$ can be considered p, when $\|\boldsymbol{R}_{22}\|$ is small. It also follows that $\boldsymbol{A}$ is quite close to $\boldsymbol{Q}\hat{\boldsymbol{R}}$, where $\hat{\boldsymbol{R}}$ is obtained by replacing $\boldsymbol{R}_{22}$ by 0.

Rank-revealing QR factorization is guaranteed to reveal the rank of A for matrices with low rank deficiency. It also never underestimates the numerical rank, but it may overestimate it.

The cost of computing RR-QR is only slightly more than the cost of regular QR factorization. However, it is significantly less expensive than computing SVD. Using a modified version of Gram-Schmidt algorithm, computing the RR-QR takes mn^2 operations, whereas the cost of computing SVD is $\mathcal{O}(mn^2)$ with constant factor of usually order 10.

In summary, we can leverage properties of rank-revealing QR factorization in several ways. First, the permutation matrix $\boldsymbol{P}$ sorts the columns of the matrix $\boldsymbol{A}$ based on their importance, valuable information when studying a matrix. Second, the permutation matrix enables us to choose subsets of the columns of $\boldsymbol{A}$ that are most significant. It also allows us to discard insignificant columns of $\boldsymbol{A}$; in contrast, SVD also tells us how many columns to discard in order to obtain a well-conditioned matrix, but gives no insight into which columns are redundant. Third, its computation is faster than SVD, so we can remove the rank deficiencies from matrices, fast. Finally, we can use it to perform a low-cost Principal Component Analysis on matrices by dropping the least significant columns of $\boldsymbol{R}$. This approach is sometimes referred to as the "poor man's" PCA.

Bibliography

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

Shikha Agrawal and Jitendra Agrawal. Neural network techniques for cancer prediction: A survey. *Procedia Computer Science*, 60:769–774, 2015.

Yong-Yeol Ahn, Sebastian E Ahnert, James P Bagrow, and Albert-László Barabási. Flavor network and the principles of food pairing. *Scientific Reports*, 1:196, 2011.

Jose M Alvarez and Mathieu Salzmann. Learning the number of neurons in deep networks. In *Advances in Neural Information Processing Systems (NeurIPS 2016)*, pages 2270–2278, 2016.

Jose M Alvarez and Mathieu Salzmann. Compression-aware training of deep networks. In *Advances in Neural Information Processing Systems (NeurIPS 2017)*, pages 856–867, 2017.

Saleema Amershi, James Fogarty, Ashish Kapoor, and Desney Tan. Effective end-user interaction with machine learning. In *AAAI Conference on Artificial Intelligence*, 2011.

Ulrich Anders and Olaf Korn. Model selection in neural networks. *Neural Networks*, 12 (2):309–323, 1999.

Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. *arXiv preprint arXiv:1611.02167*, 2016.

Onureena Banerjee, Laurent El Ghaoui, and Alexandre d'Aspremont. Model selection through sparse maximum likelihood estimation for multivariate Gaussian or binary data. *Journal of Machine Learning Research*, 9(Mar):485–516, 2008.

Albert-Laszlo Barabasi and Zoltan N Oltvai. Network biology: understanding the cell's functional organization. *Nature Reviews Genetics*, 5(2), 2004.

Gabriel Bender, Pieter-Jan Kindermans, Barret Zoph, Vijay Vasudevan, and Quoc Le. Understanding and simplifying one-shot architecture search. In *International Conference on Machine Learning (ICML 2018)*, pages 549–558, 2018.

Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994.

Dimitris Bertsimas and Jack Dunn. Optimal classification trees. *Machine Learning*, 106 (7):1039–1082, 2017.

Vighnesh Birodkar, Hossein Mobahi, and Samy Bengio. Semantic redundancies in image-classification datasets: The 10% you don't need. *arXiv preprint arXiv:1901.11409*, 2019.

Joy Buolamwini and Timnit Gebru. Gender shades: Intersectional accuracy disparities in commercial gender classification. In *Conference on Fairness, Accountability and Transparency*, pages 77–91, 2018.

Peter Businger and Gene H Golub. Linear least squares solutions by Householder transformations. *Numerische Mathematik*, 7(3):269–276, 1965.

Helmut Blcskei, Philipp Grohs, Gitta Kutyniok, and Philipp Petersen. Optimal approximation with sparsely connected deep neural networks. *SIAM Journal on Mathematics of Data Science*, 1(1):8–45, 2019.

Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. An analysis of deep neural network models for practical applications. *arXiv preprint arXiv:1605.07678*, 2016.

Dylan Cashman, Adam Perer, and Hendrik Strobelt. Mast: A tool for visualizing CNN model architecture searches. In *ICLR 2019 Debugging Machine Learning Models Workshop*, 2019. URL `https://debug-ml-iclr2019.github.io/cameraready/DebugML-19_paper_24.pdf`.

Tony F Chan. Rank revealing QR factorizations. *Linear Algebra and its Applications*, 88: 67–82, 1987.

Venkat Chandrasekaran, Pablo A Parrilo, and Alan S Willsky. Latent variable graphical model selection via convex optimization. In *IEEE Allerton Conference on Communication, Control, and Computing*, 2010.

Chaofan Chen, Oscar Li, Alina Barnett, Jonathan Su, and Cynthia Rudin. This looks like that: Deep learning for interpretable image recognition. *arXiv preprint arXiv:1806.10574*, 2018a.

Chaofan Chen, Kangcheng Lin, Cynthia Rudin, Yaron Shaposhnik, Sijia Wang, and Tong Wang. An interpretable model with globally consistent explanations for credit risk. *arXiv preprint arXiv:1811.12615*, 2018b.

Jaeyong Chung and Taehwan Shin. Simplifying deep neural networks for neuromorphic architectures. In *53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2016.

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

Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in Neural Information Processing Systems (NeurIPS 2014)*, pages 1269–1277, 2014.

Adrian Dobra, Chris Hans, Beatrix Jones, Joseph R Nevins, Guang Yao, and Mike West. Sparse graphical models for exploring gene expression data. *Journal of Multivariate Analysis*, 90(1), 2004.

Simon S Du and Jason D Lee. On the power of over-parametrization in neural networks with quadratic activation. *arXiv preprint arXiv:1803.01206*, 2018.

Dheeru Dua and Casey Graff. UCI machine learning repository, 2017. URL `http://archive.ics.uci.edu/ml`.

Daniel M Dunlavy and Dianne P O'Leary. Homotopy optimization methods for global optimization. Technical report, Sandia National Laboratories, 2005.

Daniel M Dunlavy, Dianne P O'Leary, Dmitri Klimov, and Devarajan Thirumalai. HOPE: A homotopy optimization method for protein structure prediction. *Journal of Computational Biology*, 12(10):1275–1288, 2005.

Gamaleldin Elsayed, Dilip Krishnan, Hossein Mobahi, Kevin Regan, and Samy Bengio. Large margin deep networks for classification. In *Advances in Neural Information Processing Systems (NeurIPS 2018)*, pages 842–852, 2018.

Alhussein Fawzi, Seyed-Mohsen Moosavi-Dezfooli, and Pascal Frossard. The robustness of deep networks: A geometrical perspective. *IEEE Signal Processing Magazine*, 34(6): 50–62, 2017.

Alhussein Fawzi, Seyed-Mohsen Moosavi-Dezfooli, Pascal Frossard, and Stefano Soatto. Empirical study of the topology and geometry of deep networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3762–3770, 2018.

FICO. The explainable machine learning challenge, 2018. URL `https://community.fico.com/s/explainable-machine-learning-challenge`.

Sorelle A Friedler, Carlos Scheidegger, Suresh Venkatasubramanian, Sonam Choudhary, Evan P Hamilton, and Derek Roth. A comparative study of fairness-enhancing interventions in machine learning. In *Proceedings of the Conference on Fairness, Accountability, and Transparency*, FAT* '19, pages 329–338, 2019.

Jerome Friedman, Trevor Hastie, and Robert Tibshirani. Sparse inverse covariance estimation with the graphical lasso. *Biostatistics*, 9(3):432–441, 2008.

Yarin Gal and Zoubin Ghahramani. Dropout as a Bayesian approximation: Representing model uncertainty in deep learning. In *International Conference on Machine Learning (ICML 2016)*, pages 1050–1059, 2016.

Amirata Ghorbani, Abubakar Abid, and James Zou. Interpretation of neural networks is fragile. *arXiv preprint arXiv:1710.10547*, 2017.

Koen Goetschalckx, P Wambacq, B Moons, and Marian Verhelst. Efficiently combining SVD, pruning, clustering and retraining for enhanced neural network compression. In *Proceedings of the 2018 International Workshop on Embedded and Mobile Deep Learning*, pages 1–6. ACM, 2018.

Gene H Golub and Charles F Van Loan. *Matrix Computations*. JHU Press, Baltimore, 4th edition, 2012.

Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

Caglar Gulcehre, Marcin Moczulski, Misha Denil, and Yoshua Bengio. Noisy activation functions. In *International Conference on Machine Learning (ICML 2016)*, pages 3059–3068, 2016.

Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q Weinberger. On calibration of modern neural networks. In *International Conference on Machine Learning (ICML 2017)*, pages 1321–1330, 2017.

William W Hager. Updating the inverse of a matrix. *SIAM Review*, 31(2):221–239, 1989.

Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems (NeurIPS 2015)*, pages 1135–1143, 2015.

Boris Hanin. Universal function approximation by deep neural nets with bounded width and ReLU activations. *arXiv preprint arXiv:1708.02691*, 2017.

Boris Hanin. Which neural net architectures give rise to exploding and vanishing gradients? In *Advances in Neural Information Processing Systems (NeurIPS 2018)*, pages 580–589, 2018.

Matthias Hein and Maksym Andriushchenko. Formal guarantees on the robustness of a classifier against adversarial manipulation. In *Advances in Neural Information Processing Systems (NeurIPS 2017)*, pages 2266–2276, 2017.

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

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.

Hanzhang Hu, John Langford, Rich Caruana, Saurajit Mukherjee, Eric Horvitz, and Debadeepta Dey. Efficient forward architecture search. *arXiv preprint arXiv:1905.13360*, 2019.

Shuai Huang, Jing Li, Liang Sun, Jun Liu, Teresa Wu, Kewei Chen, Adam Fleisher, Eric Reiman, and Jieping Ye. Learning brain connectivity of Alzheimer's disease from neuroimaging data. In *Advances in Neural Information Processing Systems (NeurIPS 2009)*, pages 808–816, 2009.

Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and $<$ 0.5 MB model size. *arXiv preprint arXiv:1602.07360*, 2016.

Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Logan Engstrom, Brandon Tran, and Aleksander Madry. Adversarial examples are not bugs, they are features. *arXiv preprint arXiv:1905.02175*, 2019.

Max Jaderberg, Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Synthetic data and artificial neural networks for natural scene text recognition. *arXiv preprint arXiv:1406.2227*, 2014.

Kevin Jarrett, Koray Kavukcuoglu, Yann LeCun, et al. What is the best multi-stage architecture for object recognition? In *IEEE 12th International Conference on Computer Vision*, pages 2146–2153. IEEE, 2009.

Saumya Jetley, Nicholas Lord, and Philip Torr. With friends like these, who needs adversaries? In *Advances in Neural Information Processing Systems (NeurIPS 2018)*, pages 10749–10759, 2018.

Yang Jiang, Nigel Bosch, Ryan S Baker, Luc Paquette, Jaclyn Ocumpaugh, Juliana Ma Alexandra L Andres, Allison L Moore, and Gautam Biswas. Expert feature-engineering vs. Deep neural networks: Which is better for sensor-free affect detection? In *International Conference on Artificial Intelligence in Education*, pages 198–211. Springer, 2018.

Yiding Jiang, Dilip Krishnan, Hossein Mobahi, and Samy Bengio. Predicting the generalization gap in deep networks with margin distributions. In *International Conference on Learning Representations (ICLR 2019)*, 2019.

Steven G. Johnson. The NLopt nonlinear-optimization package, `http://ab-initio.mit.edu/nlopt`, 2014.

A. Kapoor, B. Lee, D. Tan, and E. Horvitz. Interactive optimization for steering machine classification. In *Proceedings of the 28th International Conference on Human Factors in Computing Systems*, 2010.

Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. Towards proving the adversarial robustness of deep neural networks. *arXiv preprint arXiv:1709.02802*, 2017.

Pang Wei Koh and Percy Liang. Understanding black-box predictions via influence functions. In *International Conference on Machine Learning (ICML 2017)*, pages 1885–1894, 2017.

Himabindu Lakkaraju and Cynthia Rudin. Learning cost-effective and interpretable treatment regimes. In *Artificial Intelligence and Statistics*, pages 166–175, 2017.

Himabindu Lakkaraju, Ece Kamar, Rich Caruana, and Jure Leskovec. Interpretable & explorable approximations of black box models. *arXiv preprint arXiv:1707.01154*, 2017.

Balaji Lakshminarayanan, Alexander Pritzel, and Charles Blundell. Simple and scalable predictive uncertainty estimation using deep ensembles. In *Advances in Neural Information Processing Systems (NeurIPS 2017)*, pages 6402–6413, 2017.

Clifford Lam and Jianqing Fan. Sparsistency and rates of convergence in large covariance matrix estimation. *Annals of Statistics*, 37(6B), 2009.

Nina L Lanza, Samuel M Clegg, Roger C Wiens, Rhonda E McInroy, Horton E Newsom, and Matthew D Deans. Examining natural rock varnish and weathering rinds with laser-induced breakdown spectroscopy for application to ChemCam on Mars. *Applied Optics*, 51(7), 2012.

Eugene L Lawler and David E Wood. Branch-and-bound methods: A survey. *Operations Research*, 14(4):699–719, 1966.

Tao Lei, Regina Barzilay, and Tommi Jaakkola. Rationalizing neural predictions. *arXiv preprint arXiv:1606.04155*, 2016.

Oscar Li, Hao Liu, Chaofan Chen, and Cynthia Rudin. Deep learning for case-based reasoning through prototypes: A neural network that explains its predictions. In *Proceedings of AAAI Conference on Artificial Intelligence*, 2018.

Richard P Lippmann. An introduction to computing with neural nets. *Artificial Neural Networks: Theoretical Concepts*, 4(2):4–22, 1987.

Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. In *European Conference on Computer Vision (ECCV)*, pages 19–34, 2018.

Han Liu, Xi Chen, Larry Wasserman, and John D Lafferty. Graph-valued regression. In *Advances in Neural Information Processing Systems (NeurIPS 2010)*, pages 1423–1431, 2010a.

Han Liu, Kathryn Roeder, and Larry Wasserman. Stability approach to regularization selection (StARS) for high dimensional graphical models. In *Advances in Neural Information Processing Systems (NeurIPS 2010)*, pages 1432–1440, 2010b.

Han Liu, Fang Han, and Cun-hui Zhang. Transelliptical graphical models. In *Advances in Neural Information Processing Systems (NeurIPS 2012)*, pages 800–808, 2012.

Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083*, 2017.

Alexander Matyasko and Lap-Pui Chau. Margin maximization for robust classification using deep learning. In *International Joint Conference on Neural Networks*, pages 300–307. IEEE, 2017.

Nicolai Meinshausen and Peter Bühlmann. High-dimensional graphs and variable selection with the Lasso. *The Annals of Statistics*, 34(3):1436–1462, 2006.

Hrushikesh Mhaskar, Qianli Liao, and Tomaso Poggio. Learning functions: When is deep better than shallow. *arXiv preprint arXiv:1603.00988*, 2016.

Hrushikesh Narhar Mhaskar. Approximation properties of a multilayered feedforward artificial neural network. *Advances in Computational Mathematics*, 1(1):61–80, 1993.

Hossein Mobahi. Training recurrent neural networks by diffusion. *ArXiv e-prints*, abs/1601.04114, 2016.

Hossein Mobahi and John W Fisher III. A theoretical analysis of optimization by Gaussian continuation. In *AAAI Conference on Artificial Intelligence*, 2015.

Karthik Mohan, Mike Chung, Seungyeop Han, Daniela Witten, Su-In Lee, and Maryam Fazel. Structured learning of Gaussian graphical models. In *Advances in Neural Information Processing Systems (NeurIPS 2012)*, pages 620–628, 2012.

Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. Deepfool: A simple and accurate method to fool deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2574–2582, 2016.

Ramaravind Kommiya Mothilal, Amit Sharma, and Chenhao Tan. Explaining machine learning classifiers through diverse counterfactual explanations. *arXiv preprint arXiv:1905.07697*, 2019.

Kenton Murray and David Chiang. Auto-sizing neural networks: With applications to n-gram language models. *arXiv preprint arXiv:1508.05051*, 2015.

Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted Boltzmann machines. In *International Conference on Machine Learning (ICML 2010)*, pages 807–814, 2010.

Behnam Neyshabur, Srinadh Bhojanapalli, David McAllester, and Nati Srebro. Exploring generalization in deep learning. In *Advances in Neural Information Processing Systems (NeurIPS 2017)*, pages 5947–5956, 2017.

Behnam Neyshabur, Zhiyuan Li, Srinadh Bhojanapalli, Yann LeCun, and Nathan Srebro. The role of over-parametrization in generalization of neural networks. In *International Conference on Learning Representations (ICLR 2019)*, 2019.

Anh Nguyen, Jason Yosinski, and Jeff Clune. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *IEEE conference on Computer Vision and Pattern Recognition*, pages 427–436, 2015.

Jorge Nocedal and Stephen Wright. *Numerical Optimization*. Springer, New York, 2nd edition, 2006.

Steven J Nowlan and Geoffrey E Hinton. Simplifying neural networks by soft weight-sharing. *Neural Computation*, 4(4):473–493, 1992.

Dianne P. O'Leary. *Scientific Computing with Case Studies*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2009.

Joost AA Opschoor, Philipp C Petersen, and Christoph Schwab. Deep ReLU networks and high-order finite element methods. *SAM, ETH Zürich*, 2019.

Diane Oyen and Nina Lanza. Interactive discovery of chemical structure in ChemCam targets using Gaussian graphical models. In *Workshops of the International Joint Conference on Artificial Intelligence*, 2016.

Philipp Petersen and Felix Voigtlaender. Optimal approximation of piecewise smooth functions using deep ReLU neural networks. *Neural Networks*, 108:296–330, 2018.

Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean. Efficient neural architecture search via parameters sharing. In *International Conference on Machine Learning (ICML 2018)*, volume 80, pages 4095–4104, 2018.

Dimitris C Psichogios and Lyle H Ungar. SVD-NET: An algorithm that automatically selects network structure. *IEEE Transactions on Neural Networks*, 5(3):513–515, 1994.

Prajit Ramachandran, Barret Zoph, and Quoc V Le. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2018.

Garvesh Raskutti, Bin Yu, Martin J Wainwright, and Pradeep K Ravikumar. Model selection in Gaussian graphical models: High-dimensional consistency of ℓ_1-regularized MLE. In *Advances in Neural Information Processing Systems (NeurIPS 2009)*, pages 1329–1336, 2009.

Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. Why should I trust you?: Explaining the predictions of any classifier. In *International Conference on Knowledge Discovery and Data Mining*, pages 1135–1144. ACM, 2016.

Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. Anchors: High-precision model-agnostic explanations. In *AAAI Conference on Artificial Intelligence*, pages 1527–1535, 2018.

Adam J Rothman, Peter J Bickel, Elizaveta Levina, Ji Zhu, et al. Sparse permutation invariant covariance estimation. *Electronic Journal of Statistics*, 2, 2008.

Cynthia Rudin. Please stop explaining black box models for high stakes decisions. *arXiv preprint arXiv:1811.10154*, 2018.

Cynthia Rudin and Berk Ustun. Optimized scoring systems: Toward trust in machine learning for healthcare and criminal justice. *Interfaces*, 48(5):449–466, 2018.

David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Cognitive Modeling*, 5(3):1, 1988.

Chris Russell. Efficient search for diverse coherent explanations. In *Proceedings of the Conference on Fairness, Accountability, and Transparency*, FAT* '19, pages 20–28, 2019.

Ludwig Schmidt, Shibani Santurkar, Dimitris Tsipras, Kunal Talwar, and Aleksander Madry. Adversarially robust generalization requires more data. In *Advances in Neural Information Processing Systems (NeurIPS 2018)*, pages 5019–5031, 2018.

Hanie Sedghi, Vineet Gupta, and Philip M. Long. The singular values of convolutional layers. In *International Conference on Learning Representations (ICLR 2019)*, 2019.

Uri Shaham, Alexander Cloninger, and Ronald R Coifman. Provable approximation properties for deep neural networks. *Applied and Computational Harmonic Analysis*, 44(3): 537–557, 2018.

Adi Shamir, Itay Safran, Eyal Ronen, and Orr Dunkelman. A simple explanation for the existence of adversarial examples with small Hamming distance. *arXiv preprint arXiv:1901.10861*, 2019.

Aman Sinha, Hongseok Namkoong, and John Duchi. Certifiable distributional robustness with principled adversarial training. In *International Conference on Learning Representations (ICLR 2018)*, 2018.

Stephen M Smith, Karla L Miller, Gholamreza Salimi-Khorshidi, Matthew Webster, Christian F Beckmann, Thomas E Nichols, Joseph D Ramsey, and Mark W Woolrich. Network modelling methods for FMRI. *Neuroimage*, 54(2):875–891, 2011.

Jake Snell, Kevin Swersky, and Richard Zemel. Prototypical networks for few-shot learning. In *Advances in Neural Information Processing Systems (NeurIPS 2017)*, pages 4077–4087, 2017.

Alexander Spangher, Berk Ustun, and Yang Liu. Actionable recourse in linear classification. In *Proceedings of the 5th Workshop on Fairness, Accountability and Transparency in Machine Learning*, 2018.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002.

Gilbert Strang. *Linear Algebra and Learning from Data*. Wellesley-Cambridge Press, 2019.

Thomas Tanay and Lewis Griffin. A boundary tilting persepective on the phenomenon of adversarial examples. *arXiv preprint arXiv:1608.07690*, 2016.

Eu Jin Teoh, Kay Chen Tan, and Cheng Xiang. Estimating the number of hidden neurons in a feedforward network using the singular value decomposition. *IEEE Transactions on Neural Networks*, 17(6):1623–1629, 2006.

Dimitris Tsipras, Shibani Santurkar, Logan Engstrom, Alexander Turner, and Aleksander Madry. Robustness may be at odds with accuracy. In *International Conference on Learning Representations (ICLR 2019)*, 2019.

Ulrike Von Luxburg. A tutorial on spectral clustering. *Statistics and Computing*, 17(4): 395–416, 2007.

Andreas Wächter and Lorenz T Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106(1):25–57, 2006.

Sandra Wachter, Brent Mittelstadt, and Chris Russell. Counterfactual explanations without opening the black box: Automated decisions and the GDPR. *Harvard Journal of Law & Technology*, 31(2), 2018.

Tong Wang, Cynthia Rudin, Finale Velez-Doshi, Yimin Liu, Erica Klampfl, and Perry MacNeille. Bayesian rule sets for interpretable classification. In *IEEE International Conference on Data Mining*, pages 1269–1274, 2016.

Layne T Watson. Numerical linear algebra aspects of globally convergent homotopy methods. *SIAM Review*, 28(4):529–545, 1986.

Roger C. Wiens, Sylvestre Maurice, Bruce Barraclough, Muriel Saccoccio, Walter C. Barkley, James F. Bell, Steve Bender, John Bernardin, Diana Blaney, Jennifer Blank, Marc Bouyé, Nathan Bridges, Nathan Bultman, Phillippe Caïs, Robert C. Clanton, Benton Clark, Samuel Clegg, Agnes Cousin, David Cremers, Alain Cros, Lauren DeFlores, Dorothea Delapp, Robert Dingler, Claude D'Uston, M. Darby Dyar, Tom Elliott, Don Enemark, Cecile Fabre, Mike Flores, Olivier Forni, Olivier Gasnault, Thomas Hale, Charles Hays, Ken Herkenhoff, Ed Kan, Laurel Kirkland, Driss Kouach, David Landis, Yves Langevin, Nina Lanza, Frank LaRocca, Jeremie Lasue, Joseph Latino, Daniel Limonadi, Chris Lindensmith, Cynthia Little, Nicolas Mangold, Gerard Manhes, Patrick Mauchien, Christopher McKay, Ed Miller, Joe Mooney, Richard V. Morris, Leland Morrison, Tony Nelson, Horton Newsom, Ann Ollila, Melanie Ott, Laurent Pares, René Perez, Franck Poitrasson, Cheryl Provost, Joseph W. Reiter, Tom Roberts, Frank Romero, Violaine Sautter, Steven Salazar, John J. Simmonds, Ralph Stiglich, Steven Storms, Nicolas Striebig, Jean-Jacques Thocaven, Tanner Trujillo, Mike Ulibarri, David Vaniman, Noah Warner, Rob Waterbury, Robert Whitaker, James Witt, and Belinda Wong-Swanson. The ChemCam instrument suite on the Mars Science Laboratory (MSL) rover: Body unit and combined system tests. *Space Science Reviews*, 170(1-4), 2012.

Martin Wistuba. Finding competitive network architectures within a day using UCT. *arXiv preprint arXiv:1712.07420*, 2017.

Yuhui Xu, Yuxi Li, Shuai Zhang, Wei Wen, Botao Wang, Yingyong Qi, Yiran Chen, Weiyao Lin, and Hongkai Xiong. Trained rank pruning for efficient deep neural networks. *arXiv preprint arXiv:1812.02402*, 2018.

Jian Xue, Jinyu Li, and Yifan Gong. Restructuring of deep neural network acoustic models with singular value decomposition. In *Interspeech*, pages 2365–2369, 2013.

Roozbeh Yousefzadeh and Dianne P O'Leary. Interpreting neural networks using flip points. *arXiv preprint arXiv:1903.08789*, 2019a.

Roozbeh Yousefzadeh and Dianne P O'Leary. Debugging trained machine learning models using flip points. In *ICLR 2019 Debugging Machine Learning Models Workshop*, 2019b. URL `https://debug-ml-iclr2019.github.io/cameraready/DebugML-19_paper_11.pdf`.

Roozbeh Yousefzadeh, Diane Oyen, and Nina Lanza. Learning diverse Gaussian graphical models and interpreting edges. In *Proceedings of the SIAM International Conference on Data Mining*, pages 405–413, 2019. doi: 10.1137/1.9781611975673.46.

Ming Yuan and Yi Lin. Model selection and estimation in the Gaussian graphical model. *Biometrika*, 94(1), 2007.

Tanja Zerenner, Petra Friederichs, Klaus Lehnertz, and Andreas Hense. A Gaussian graphical model approach to climate networks. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 24(2), 2014.

Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. *arXiv preprint arXiv:1611.03530*, 2016.

T. Zhao, H. Liu, K. Roeder, J. Lafferty, and L. Wasserman. The huge package for high-dimensional undirected graph estimation in R. *The Journal of Machine Learning Research*, 13(1), 2012.

Ding-Xuan Zhou. Universality of deep convolutional neural networks. *Applied and Computational Harmonic Analysis*, 2019.

Hao Zhou, Jose M Alvarez, and Fatih Porikli. Less is more: Towards compact CNNs. In *European Conference on Computer Vision*, pages 662–677. Springer, 2016.

Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 8697–8710, 2018.