Optimisation & Generalisation
in Networks of Neurons

Thesis by
Jeremy Bernstein

In Partial Fulfillment of the Requirements for
the Degree of Doctor of Philosophy

Caltech

California Institute of Technology
Pasadena, California

2023
Defended September 23, 2022

ACKNOWLEDGEMENTS

ABSTRACT

The goal of this thesis is to develop the optimisation and generalisation theoretic foundations of learning in artificial neural networks. The thesis tackles two central questions. Given training data and a network architecture:

1) Which weight setting will generalise best to unseen data, and why?

2) What optimiser should be used to recover this weight setting?

On optimisation, an essential feature of neural network training is that the network weights affect the loss function only indirectly through their appearance in the network architecture. This thesis proposes a three-step framework for deriving novel "architecture aware" optimisation algorithms. The first step— termed *functional majorisation*—is to majorise a series expansion of the loss function in terms of functional perturbations. The second step is to derive *architectural perturbation bounds* that relate the size of functional perturbations to the size of weight perturbations. The third step is to substitute these architectural perturbation bounds into the functional majorisation of the loss and to obtain an optimisation algorithm via minimisation. This constitutes an application of the *majorise-minimise meta-algorithm* to neural networks.

On generalisation, a promising recent line of work has applied PAC-Bayes theory to derive non-vacuous generalisation guarantees for neural networks. Since these guarantees control the average risk of ensembles of networks, they do not address which individual network should generalise best. To close this gap, the thesis rekindles an old idea from the kernels literature: the *Bayes point machine*. A Bayes point machine is a single classifier that approximates the aggregate prediction of an ensemble of classifiers. Since aggregation reduces the variance of ensemble predictions, Bayes point machines tend to generalise better than other ensemble members. The thesis shows that the space of neural networks consistent with a training set concentrates on a Bayes point machine if both the network width and normalised margin are sent to infinity. This motivates the practice of returning a wide network of large normalised margin.

Potential applications of these ideas include novel methods for uncertainty quantification, more efficient numerical representations for neural hardware, and optimisers that transfer hyperparameters across learning problems.

TABLE OF CONTENTS

LIST OF FIGURES & TABLES

NOTATION

Measuring size

$\|x\|_2$ Euclidean norm of vector x

$\|W\|_F$ Frobenius norm of matrix W

$\|W\|_*$ operator norm of matrix W

$\|f\|_{\mathrm{RKHS}}$ reproducing kernel Hilbert space norm of function f

Describing data

$\mathcal{X}$ input space

$\mathcal{Y}$ output space

X collection of m train inputs $X = \{x_1, ..., x_m\} \in \mathcal{X}^m$

Y vector of m train labels $Y = [y_1, ..., y_m] \in \mathcal{Y}^m$

S train set $S = (X, Y) \equiv \{(x_1, y_1), ..., (x_m, y_m)\}$

f_X projected function $f_X = [f(x_1), ..., f(x_m)] \in \mathcal{Y}^m$

Working with kernels and Gaussian processes

$k(\cdot, \cdot)$ kernel function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$

$K_{XX'}$ Gram matrix $K_{XX'}^{ij} \coloneqq k(x_i, x_j')$

K_{xX} Gram vector $K_{xX}^i \coloneqq k(x, x_i)$

K_{xx} Gram scalar $K_{xx} \coloneqq k(x, x)$

Describing neural architecture

$\mathcal{W}$ weight space

L number of layers

d_l width of lth layer

$f(\cdot; \cdot)$ neural network $f : \mathbb{R}^{d_0} \times \mathcal{W} \to \mathbb{R}^{d_L}$

Part I

INTRODUCTION
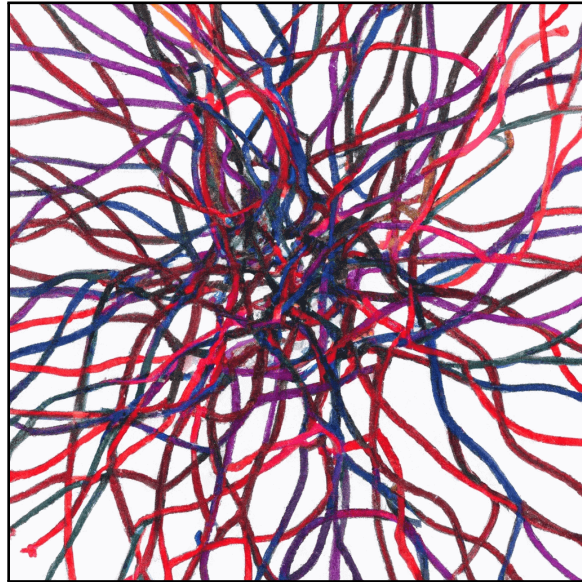
"It's much closer to Freud, the idea that there's this thin film of consciousness and deliberate reasoning and all this seething stuff underneath."

Geoffrey E. Hinton, 2018

Chapter 1

FINDING THE FOUNDATIONS

> This chapter introduces the central goal of this thesis: to find the foundations of optimisation and generalisation in artificial neural networks.

Research into artificial neural networks has drawn on various disciplines of science and engineering. Neuroscience and psychology have inspired basic learning frameworks (Sutton and Barto, 2018) as well as specific neural architectures (Fukushima, 1980). Computer engineering has yielded hardware accelerators that enable both experimentation and applications at larger and larger scale (Steinkraus et al., 2005). And mathematics and statistics offer the toolkits needed to understand the basic properties of learning systems.

From this breadth of scientific input, a paradigm known as *deep learning* has emerged (Schmidhuber, 2015; LeCun et al., 2015). Deep learning has been driving progress in machine learning applications across science and industry over the course of the last decade. While certain applications deviate from the following schema, at its core, deep learning involves three steps:

1) A large dataset of training examples is collected. These examples should in a sense span the richness of behaviour present in the task of interest.

2) An expressive neural network is constructed. A neural network consists of simple linear building blocks chained together and interspersed with simple elementwise nonlinearities to yield an overall highly complex and nonlinear function. The neural network is parameterised by the weights of the linear building blocks, and adjusting these weights adjusts the function that the network implements.

3) The error—otherwise known as the *loss*—of the network over the training examples is evaluated, and the mathematical gradient of this error with respect to the network's weights is computed. The weights are then adjusted according to this gradient so as to reduce the error. This step is iterated until the error on the training examples has been made small.

The fascinating aspect of this procedure, and of learning in general, is that minimising error on *train examples* is often sufficient to attain good performance on previously unseen *test examples*. Furthermore, there is a certain conceptual simplicity: deep learning is just gradient descent on a neural network's error over a set of examples. Despite this simplicity, some of the most basic questions surrounding its underlying mathematics are not resolved. For example:

⟨?⟩ *Optimisation.* Given the gradient of a neural network's error, how far and in which direction should the network weights be best adjusted?

⟨?⟩ *Generalisation.* Which of the functions that a neural network implements will perform best on unseen data. And why?

In practice, these questions are usually addressed by trial-and-error over a set of heuristic techniques. For instance, a few variants of gradient descent are known to work quite well for neural networks. On a given task, one such variant will often perform well, but it is not known in advance which it will be (Schmidt et al., 2021). Such trial-and-error has been highly successful—it is responsible for the wealth of deep learning applications that are seen today. Nevertheless, it is a contention of this thesis that, by answering these questions by way of a formal theory, there is potential both to simplify practical workflows as well as to unlock fundamentally new deep learning functionalities.

Part of the reason that these questions are still open is that researchers do not know which theoretical framework should be used to answer them. For optimisation, researchers have attempted to apply such varied frameworks as information geometry (Amari, 1998) and mirror descent (Azizan and Hassibi, 2019). For generalisation, the situation is similar (Valle-Pérez and Louis, 2020). The aim of this thesis, then, is to develop foundational frameworks and principles that should be used to study learning in artificial neural networks.

These principles could be useful to the machine learning practitioner, since they could provide her with learning algorithms that generalise better while requiring less arbitrary tuning of hyperparameters. They could be useful to the computer hardware engineer, since they could help him design chips that more effectively support learning. And the principles could be useful to the neuroscientist who is seeking to transfer ideas "upstream" to the study of biological neural networks.

The next two sections introduce the steps taken by this thesis toward tackling these questions of optimisation and generalisation in networks of neurons.

1.1   Optimisation via perturbation

Hyperparameter tuning is the bane of every deep learning practitioner's existence. A large number of optimisation algorithms have been proposed for neural networks (Schmidt et al., 2021), and each has a set of adjustable parameters known as *hyperparameters* that affect the performance of the method. The *learning rate* is the canonical example of a hyperparameter—this controls how strongly the network weights are adjusted in response to the gradient of the network's error. In the absence of compelling theoretical guidance on how to set the learning rate, best engineering practice is to try a logarithmic grid of possibilities and to see what works best (Goodfellow et al., 2016). This tuning process inflates the computational cost of applications since a network must be trained many times in order to find a single network that works well.

This thesis argues that the reason learning rate tuning in deep learning is so cumbersome is that a proper *perturbation analysis* of neural architecture is missing. Roughly, what this means is that there is not a simple, computationally tractable means of estimating how sensitive the network's function is to adjustments of its weights. And even given such a sensitivity measure, there is no way to apply it to derive optimisation algorithms for learning problems. To move beyond this situation, this thesis poses the following question:

> How far can the weights of a neural network be perturbed before
> the function of the network is damaged?

Answering this question is important for optimisation, since an optimiser must not damage the network that it is training. But the question could be of more general interest, too. It gets at the *precision* with which weights need to be stored, so it could be important for the computer hardware engineer to consider. Furthermore, the question could be interesting to the neuroscientist studying the dynamics of synaptic plasticity in living brains.

This question is tackled in Part II. Chapter 4 surveys classic iterative optimisation algorithms and shows how they may be put on common footing by way of a perturbation analysis operating in the *weight space* of the optimisation problem. Next, Chapter 5 restricts attention to machine learning optimisation

problems where the weights enter the optimisation problem via the machine learning model architecture. The chapter develops a novel technique termed *functional majorisation* that is essentially a perturbation analysis of the loss function operating in the function space of the machine learning model. Finally, Chapter 6 develops novel *architectural perturbation bounds* for deep neural networks. These bounds connect the size of weight perturbations to the size of the induced perturbation in the network function. They may be substituted into the functional majorisation of the loss function to obtain novel *architecture aware* optimisation methods for deep neural networks. These methods address how the learning rate should depend on details such as the depth of the neural network that is being trained.

In short, the thesis develops a perturbation analysis of deep networks through *architectural perturbation bounds*, and shows how these bounds interact with the neural network's error via *functional majorisation*. The resulting optimisation algorithms, obtained by minimising the functional majorisation of the error with respect to weight perturbations, constitute an application of the *majorise-minimise meta-algorithm* (Lange, 2016) to neural networks.

1.2   Generalisation via aggregation

What allows a machine learning model that has been fit to a finite set of training data to generalise to test examples that it has never seen before? This is, in a sense, the fundamental question of learning. This question is particularly interesting in the case of neural networks that are vastly *over-parameterised*, meaning that they have far more weights than training data. In this case, the neural network may have enough capacity to simply *memorise* its training data, without performing any useful computational processing that could lead to generalisation (Zhang et al., 2017). So why, when these kinds of vastly over-parameterised networks are trained, do they generalise regardless?

The study of generalisation in machine learning algorithms has a rich history. For instance, *uniform convergence theory*—dating back to the work of Vapnik and Chervonenkis (1971)—attempts to bound the difference between train and test error for all functions in the space of functions in which one is interested. Meanwhile *PAC-Bayes theory* (McAllester, 1999) provides another means of bounding the generalisation gap of machine learning algorithms. Unlike uniform convergence bounds, PAC-Bayes bounds are on the average generalisation gap

over a distribution of functions. PAC-Bayes bounds have been found to be significantly tighter than uniform convergence bounds (Seeger, 2002), while incorporating information about both the training set and the machine learning model architecture in a natural way.

Unfortunately, since PAC-Bayes bounds hold in expectation over distributions of functions, they say nothing about the generalisation of an individual function. To address this shortcoming, this thesis poses the following question:

> Given a neural network with the capacity to fit a set of training data in many ways, which of these functions should generalise best?

Answering this question is important for two practical reasons. First, most directly, in many applications one is interested in returning the single network that makes the best possible predictions. And second, less directly, in some applications one is interested in obtaining some measure of the uncertainty of the predictions of this best network. One idea for assessing uncertainty involves training an ensemble of networks in order to measure the variance across their predictions. But then it is important to have a means of ensuring that ensemble members do not all just collapse on to the single best generalising network.

These issues are tackled in Part III. Chapter 7 surveys some classic ideas in generalisation theory, including uniform convergence theory and PAC-Bayes theory. Chapter 8 develops a PAC-Bayes theory of Gaussian process classification. While this chapter contains little conceptual novelty in comparison to prior work (Seeger, 2002), the chapter derives some novel analytical results that will be useful later on. Finally, Chapter 9 rekindles the idea of the *Bayes point machine* (Herbrich et al., 2001). A Bayes point machine is a single classifier that approximates an ensemble's aggregate prediction. Since aggregation tends to improve ensemble performance, Bayes point machines are thought to generalise significantly better than other predictors. Via a detour through the neural network–Gaussian process correspondence (Neal, 1994), the thesis finds that maximising the *normalised margin* of a neural network's training predictions causes the network function to concentrate on a Bayes point machine.

The main conceptual agenda of this part of the thesis is to put forward a novel perspective on generalisation in artificial neural networks as arising from a specific form of approximate Bayesian inference. In particular, by leveraging

a statistical characterisation of the neural network function space known as the neural network–Gaussian process correspondence, it can be seen that a single neural network may itself approximate an aggregated predictor with good generalisation properties.

But before all that, the remaining chapters of this first part of the thesis will formally introduce the machine learning problem, as well as the technical tools needed to study it. Chapter 2 formally introduces neural networks, Gaussian processes and kernel methods, while Chapter 3 introduces various correspondences between these spaces of functions.

References

Shun-ichi Amari (1998). "Natural gradient works efficiently in learning". In: *Neural Computation.*

Navid Azizan and Babak Hassibi (2019). "Stochastic gradient/mirror descent: Minimax optimality and implicit regularization". In: *International Conference on Learning Representations.*

Kunihiko Fukushima (1980). "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position". In: *Biological Cybernetics.*

Ian Goodfellow, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning.* MIT Press.

Ralf Herbrich, Thore Graepel, and Colin Campbell (2001). "Bayes point machines". In: *Journal of Machine Learning Research.*

Kenneth Lange (2016). *MM Optimization Algorithms.* Society for Industrial and Applied Mathematics.

Yann LeCun, Yoshua Bengio, and Geoffrey E. Hinton (2015). "Deep learning". In: *Nature.*

David A McAllester (1999). "Some PAC-Bayesian theorems". In: *Machine Learning.*

Radford M. Neal (1994). "Bayesian learning for neural networks". Ph.D. thesis. Department of Computer Science, University of Toronto.

Jürgen Schmidhuber (2015). "Deep learning in neural networks: An overview". In: *Neural Networks.*

Robin M. Schmidt, Frank Schneider, and Philipp Hennig (2021). "Descending through a crowded valley—benchmarking deep learning optimizers". In: *International Conference on Machine Learning.*

Matthias Seeger (2002). "PAC-Bayesian generalisation error bounds for Gaussian process classification". In: *Journal of Machine Learning Research.*

Dave Steinkraus, Ian Buck, and Patrice Y. Simard (2005). "Using GPUs for machine learning algorithms". In: *International Conference on Document Analysis and Recognition.*

Richard S. Sutton and Andrew G. Barto (2018). *Reinforcement Learning: An Introduction.* MIT Press.

Guillermo Valle-Pérez and Ard A. Louis (2020). "Generalization bounds for deep learning". In: *arXiv:2012.04115.*

Vladimir N. Vapnik and Alexey Ya. Chervonenkis (1971). "On the uniform convergence of relative frequencies of events to their probabilities". In: *Theory of Probability & Its Applications.*

Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals (2017). "Understanding deep learning requires rethinking generalization". In: *International Conference on Learning Representations.*

Chapter 2

CONSTRUCTING SPACES OF FUNCTIONS

> This chapter provides an introduction to three popular machine learning techniques: kernel methods, Gaussian processes and neural networks. The material is expositional and is included for the reader's aid.

A machine learning algorithm uses training data to select a function from a space of functions. In order for this procedure to work well, it is important to construct an appropriate space of functions for the algorithm to select from. When choosing a space of functions, there are three main considerations:

1) *Leveraging prior information.* How can prior knowledge or belief about the structure of the data be incorporated into the function space?

2) *Measuring complexity.* How can one assess the complexity of a function within the space, in order to choose a simple explanation of the data?

3) *Computational efficiency.* How can one design a space of functions that is cheap to select from in the face of data?

This chapter surveys three popular techniques for constructing spaces of functions: kernel methods, Gaussian processes and neural networks. Each technique embodies a different philosophical approach to the above considerations. Kernel methods use *functional analysis* to incorporate prior knowledge and measure complexity. Gaussian processes employ the tools of *Bayesian probability*. While for neural networks, these considerations are mainly addressed via *empiricism*— the proof of a method's validity is in its pudding.

While kernel methods, Gaussian processes and neural networks each take a different approach to the above questions, correspondences exist between their respective function spaces which allow tools from one to be ported to another. These correspondences are surveyed in Chapter 3. The present chapter focuses on introducing the function spaces themselves, their complexity measures and their approaches to fitting data.

Before moving on to the techniques, it will first be useful to define the notion of *projecting* a function on to a set of inputs.

Definition 2.1 (Function projection). Given a function $f : \mathcal{X} \rightarrow \mathcal{Y}$ and a collection of m inputs $X = \{x_1, ..., x_m\}$, the *projection* f_X of the function f on to the input data X is given by:

$$f_X \coloneqq \left[f(x_1), ..., f(x_m) \right] \in \mathcal{Y}^m. \tag{2.1}$$

In words: the projection of a function on to a set of inputs is the vector of function outputs across the inputs. Given a function $f(\cdot, w) : \mathcal{X} \rightarrow \mathcal{Y}$ that is parameterised by a weight vector w, the projection on to X is denoted $f_X(w)$.

2.1 Kernel methods

A kernel $k(\cdot, \cdot)$ is a function that measures the degree of similarity between its two arguments. In particular, when $k(x, x')$ is large then inputs x and x' are similar, and when $k(x, x')$ is small then the two inputs are dissimilar. The notion of similarity is encoded by the choice of kernel function—a simple example being the Gaussian kernel:

Example 2.1 (Gaussian kernel). For a pair of inputs $x, x' \in \mathbb{R}^n$ and a length scale $\sigma > 0$, the Gaussian kernel is given by:

$$k_{\text{Gaussian}}(x, x') \coloneqq \exp \left(-\frac{\|x - x'\|_2^2}{2\sigma^2} \right). \tag{2.2}$$

Beyond measuring similarity, a kernel may be used to construct a space of functions. To see this, consider that the Gaussian kernel viewed as a function of x may be interpreted as an unnormalised Gaussian measure centred at the point x'. The basic idea is that one may construct a more-or-less arbitrary function by superposing many Gaussian kernels centred at different locations. This idea is illustrated in Figure 2.1. In accordance with this construction, the function $k_{\text{Gaussian}}(\cdot, x')$ is referred to as the *kernel basis function* centred at x'.

By superposing kernel basis functions in this way, any kernel $k(\cdot, \cdot)$ can be used to construct a space of functions. The functions are parameterised by the centres and strengths of each kernel basis function within the superposition. Such a space of functions is known as a *reproducing kernel Hilbert space* (RKHS). The remainder of this section will introduce the concept of an RKHS more formally, including how to measure the complexity of a function within an RKHS, and how to select a function from an RKHS in light of data.

$$f(x) = k(x, 0) \qquad\qquad f(x) = k(x, -1) - k(x, 1)$$
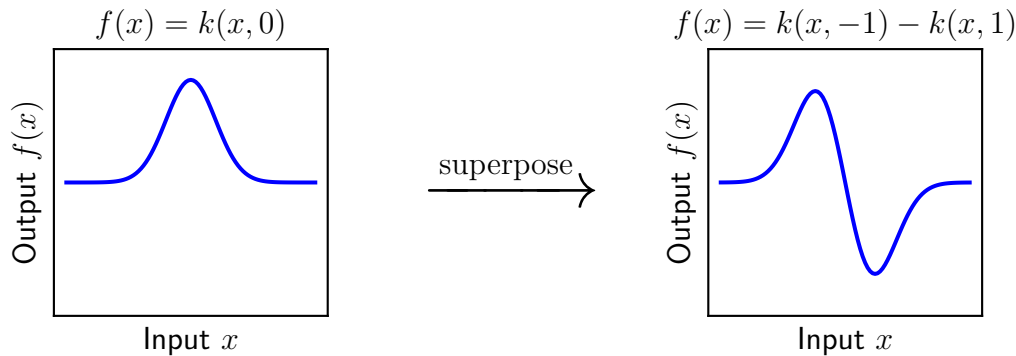
Figure 2.1: Constructing a function by superposing kernel basis functions. The left panel displays a kernel basis function for the Gaussian kernel (Example 2.1). The right panel shows how two kernel basis functions can be superposed to build a more complicated function. A *reproducing kernel Hilbert space* (Definition 2.5) consists of superpositions of arbitrarily many kernel basis functions.

Using a kernel to construct a space of functions

For a function $k(\cdot, \cdot)$ to qualify as a kernel, it must satisfy two conditions:

Definition 2.2 (Kernel). A function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is a *kernel* provided that:

i) k is *symmetric*: for any pair of inputs $x, x' \in \mathcal{X}$, $k(x, x') = k(x', x)$;

ii) k is *positive definite*: for any set of m distinct inputs $X = \{x_1, ..., x_m\}$, the corresponding *Gram matrix* $K_{XX}^{ij} := k(x_i, x_j)$ is positive definite.

These conditions imply that important computations involving the kernel are well-defined. For instance, the inverse K_{XX}^{-1} exists. Given Definition 2.2, it is simple to construct a space of functions by superposing kernel basis functions:

Definition 2.3 (Pre-RKHS). Given a kernel $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$, the *precursor* to a *reproducing kernel Hilbert space* (pre-RKHS) consists of all linear combinations of finitely many kernel basis functions—that is, all functions of the form:

$$f(\cdot) = \sum_{i=1}^{m} \alpha_i \, k(\cdot, x_i), \qquad\qquad (2.3)$$

for any weights $\alpha_1, ..., \alpha_m \in \mathbb{R}$, centres $x_1, ..., x_m \in \mathcal{X}$, and positive integer m.

One of the attractive features of a pre-RKHS is that one may measure the similarity of two functions within it by taking an inner product:

Definition 2.4 (Pre-RKHS inner product)**.** Given a kernel k and two functions $f(\cdot) = \sum_{i=1}^{m} \alpha_i \, k(\cdot, x_i)$ and $g(\cdot) = \sum_{i=1}^{m'} \beta_i \, k(\cdot, x_i')$, the *inner product* of f and g in the pre-RKHS induced by k is given by:

$$\langle f, g \rangle_{\text{RKHS}} := \sum_{i=1}^{m} \sum_{j=1}^{m'} \alpha_i k(x_i, x_j') \beta_j =: \alpha^\top K_{XX'} \beta.$$

It can be checked that, by the symmetry and positive definiteness of k, this definition serves as a valid inner product.

A standard treatment of kernel methods would now proceed to *complete* the pre-RKHS to obtain a true *Hilbert space* of functions. Completion essentially involves augmenting the pre-RKHS with certain limits of sequences of functions. This process adds significant technical overhead, while the pre-RKHS is already sufficient for the techniques studied in this thesis. As such, the term RKHS will be used herein merely as convenient shorthand for a pre-RKHS.

Definition 2.5 (RKHS)**.** By an abuse of terminology, an *RKHS* is a pre-RKHS.

Definition 2.6 (RKHS inner product)**.** By an abuse of terminology, an *RKHS inner product* is a pre-RKHS inner product.

Notice that the inner product of a function $f(\cdot) = \sum_{i=1}^{m} \alpha_i \, k(\cdot, x_i)$ with the kernel basis function $k(\cdot, x)$ *reproduces* the function evaluated at x:

$$\langle f, k(\cdot, x) \rangle_{\text{RKHS}} = \sum_{i=1}^{m} \alpha_i \, k(x, x_i) = f(x). \tag{2.4}$$

Equation 2.4 is known as the *reproducing property* of the RKHS.

Measuring complexity via RKHS norm

The RKHS inner product also leads to a natural tool for measuring the complexity of functions within the RKHS: the RKHS norm.

Definition 2.7 (RKHS norm)**.** Given a kernel k, the *RKHS norm* of a function $f(\cdot) = \sum_{i=1}^{m} \alpha_i \, k(\cdot, x_i)$ in the RKHS induced by k is given by:

$$\|f\|_{\text{RKHS}}^2 := \langle f, f \rangle_{\text{RKHS}} = \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i k(x_i, x_j) \alpha_j =: \alpha^\top K_{XX} \alpha.$$

To demonstrate that the RKHS norm constrains the complexity of a function within an RKHS, the following lemma shows that the RKHS norm limits how fast a function can vary as its input is varied:

Lemma 2.1. *Consider a function $f : \mathcal{X} \to \mathbb{R}$ in an RKHS induced by kernel k. For any two inputs $x, x' \in \mathcal{X}$, the variation in f satisfies:*

$$|f(x) - f(x')| \le \|f\|_{\mathrm{RKHS}} \cdot \mathrm{d}(x, x'), \tag{2.5}$$

where the distance function $\mathrm{d}(x, x') := \sqrt{k(x, x) + k(x', x') - 2 \cdot k(x, x')}$.

Proof. By the reproducing property and the Cauchy-Schwarz inequality:

$$|f(x) - f(x')| = |\langle f, k(\cdot, x) - k(\cdot, x')\rangle_{\mathrm{RKHS}}|$$
$$\le \|f\|_{\mathrm{RKHS}} \cdot \|k(\cdot, x) - k(\cdot, x')\|_{\mathrm{RKHS}}.$$

The proof is completed by observing that, by the definition of RKHS norm, it holds that $\|k(\cdot, x) - k(\cdot, x')\|_{\mathrm{RKHS}} = \sqrt{k(x, x) + k(x', x') - 2 \cdot k(x, x')}$. $\qquad\square$

So a function's RKHS norm serves as a kind of *Lipschitz constant* for the function's continuity. The distance between inputs in this notion of continuity is measured according to a special distance function $\mathrm{d}(\cdot, \cdot)$ related to the degree of kernel similarity between inputs. To gain further intuition about this lemma, it may help to consider its specialisation to the Gaussian kernel:

Corollary 2.1. *Consider a function $f : \mathcal{X} \to \mathbb{R}$ in the RKHS induced by the Gaussian kernel (Definition 2.1) with length scale $\sigma = 1$. For any two inputs $x, x' \in \mathcal{X}$, the variation in f satisfies:*

$$|f(x) - f(x')| \le \sqrt{2} \cdot \|f\|_{\mathrm{RKHS}} \cdot \sqrt{1 - \mathrm{e}^{-\frac{1}{2}\|x - x'\|_2^2}}. \tag{2.6}$$

Fitting data subject to minimum RKHS norm

So far, this section has defined a space of functions called an RKHS, and a way to measure the complexity of functions in that space called the RKHS norm. It now makes sense to think of finding the least complex function in the RKHS that fits a set of data. This object admits a simple description, as follows:

Theorem 2.1 (Minimum RKHS norm kernel interpolation)**.** *Consider a set of m distinct training inputs $X = \{x_1, ..., x_m\}$ with respective training labels arranged into a vector $Y = [y_1, ..., y_m] \in \mathbb{R}^m$. Say that a function f interpolates (X, Y) if the projection f_X (Definition 2.1) satisfies $f_X = Y$. In the RKHS induced by kernel k, the interpolator $f_\star$ of (X, Y) with minimum RKHS norm is given by:*

$$f_\star(x) = \sum_{i=1}^{m} (K_{XX}^{-1} Y)_i \, k(x, x_i) =: K_{xX} K_{XX}^{-1} Y, \tag{2.7}$$

for Gram vector $K_{xX}^i := k(x, x_i)$ *and Gram matrix* $K_{XX}^{ij} := k(x_i, x_j)$.

Proof. To see that $f_\star$ interpolates (X, Y), observe that

$$[f_\star(x_1), ..., f_\star(x_m)] = K_{XX} K_{XX}^{-1} Y = Y.$$

To see that no interpolator exists with smaller RKHS norm, consider that another interpolator $g(\cdot)$ may be decomposed as:

$$g(\cdot) = f_\star(\cdot) + (g - f_\star)(\cdot).$$

But since $f_\star$ and g both interpolate the training set, for any training input x_i:

$$\langle g - f_\star, k(\cdot, x_i) \rangle_{\text{RKHS}} = g(x_i) - f_\star(x_i) = 0,$$

where the first equality follows by the reproducing property. So $g - f_\star$ is orthogonal to kernel basis functions $k(\cdot, x_1), ..., k(\cdot, x_m)$. Since $f_\star$ is constructed purely from those basis functions, this implies that $g - f_\star$ and $f_\star$ are themselves orthogonal: $\langle g - f_\star, f_\star \rangle_{\text{RKHS}} = 0$. In turn:

$$\|g\|_{\text{RKHS}}^2 = \|f_\star\|_{\text{RKHS}}^2 + \|g - f_\star\|_{\text{RKHS}}^2 \geq \|f_\star\|_{\text{RKHS}}^2, \tag{2.8}$$

which establishes the result. $\qquad\square$

The implication of this theorem is that given a training set $S = (X, Y)$ of m examples, the minimum RKHS norm interpolator $f_\star$ may be constructed by simple linear algebra operations $f_\star(x) = K_{xX} K_{XX}^{-1} Y$ involving kernel Gram matrix K_{XX} and Gram vector K_{xX}. The dominant computational cost is that of inverting an $m \times m$ matrix, which is naïvely $\mathcal{O}(m^3)$. For this reason, the cost of kernel methods is typically cubic in the size of the training set.

Furthermore, Equation 2.7 shows that the kernel interpolator of minimum RKHS norm may be represented using only finitely many kernel basis functions. As such, Theorem 2.1 is connected to a classic result known as the *representer theorem* (Schölkopf and Smola, 2001).

2.2 Gaussian processes

The previous section showed that a function space may be constructed by superposing kernel basis functions. But there are other means of constructing a function space from a kernel. In the case of *Gaussian processes*, a kernel is

used to place a probability measure over a space of functions. The functions themselves may be sampled from this measure.

A probability measure on function space can be thought of as encoding *prior belief* about which kinds of functions would have a good chance of explaining the data when it arrives. Phrased another way, functions of smaller prior probability might be considered more complex. An advantage of this approach is that, given the training data, a posterior distribution over functions consistent with the data may be constructed simply by turning the handle of Bayes' rule. This posterior provides not just a single prediction for a new data point, but a range of predictions each accompanied by a posterior probability.

Constructing a space of functions by random sampling

Given an input space $\mathcal{X}$, consider drawing a Gaussian random variable at each point $x \in \mathcal{X}$ and recording the value of each random draw as $f(x) \in \mathbb{R}$. To make life most interesting, one may choose not to draw these random variables independently, but rather from a joint Gaussian distribution. The covariance of this joint Gaussian could encode, for instance, that the closer an input x is to another input x', the more likely it is that $f(x)$ would be similar to $f(x')$. This construction, of jointly Gaussian *function values* $f(\cdot)$ with covariance structure based on similarity in the input space $\mathcal{X}$, is known as a Gaussian process:

Definition 2.8 (Gaussian process). Consider an input space $\mathcal{X}$ and a kernel $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$. If for any finite set of inputs $X = \{x_1, ..., x_m\} \in \mathcal{X}^m$, the distribution of function values $f(x_1), ..., f(x_m) \sim \text{NORMAL}(0, K_{XX})$, then the function f is drawn from a *Gaussian process* with covariance k: $f \sim \text{GP}(0, k)$.

This definition is said to establish the *prior measure* that a Gaussian process with a given kernel assigns to a space of functions.

Measuring complexity via probability

In principle, a Gaussian process provides a simple means to compare the complexity of functions. For instance, given two functions f_1 and f_2, one may simply declare the less likely of f_1 and f_2 under the Gaussian process prior to be more complex. To make this idea workable, one notices in Definition 2.8 that it is easier to assess the probability of a Gaussian process function when it is inspected only on a finite collection of inputs. As such, one could collect a collection of m inputs $X \in \mathcal{X}^m$ and compare the density that $\text{NORMAL}(0, K_{XX})$

assigns to $\{f_1(x)\}_{x \in X}$ versus $\{f_2(x)\}_{x \in X}$. This procedure is not entirely satisfactory, since the answer depends on the choice of the set of inputs X. Still it conveys the spirit of the idea that prior probability may be used to assess complexity.

That being said, the complexity of individual functions does not usually play a major role in discussion of Gaussian processes. *Distributions over functions* are considered to be more important, and Bayesians advocate for making predictions by integrating over these distributions of functions (Neal, 1994). The next section will discuss how a posterior distribution may be constructed from the Gaussian process prior in light of data.

Fitting data via Bayesian inference

An attractive feature of Gaussian processes is that, given data, one may use Bayes' rule to derive a posterior distribution over functions. Given a training sample $S = (X, Y)$, Bayes' rule states that:

$$\mathbb{P}[f \mid S] = \frac{\mathbb{P}[S \mid f] \cdot \mathbb{P}[f]}{\sum_{f'} \mathbb{P}[S \mid f'] \cdot \mathbb{P}[f']}. \tag{2.9}$$

There are three important quantities appearing in this equation:

1) The *prior* $\mathbb{P}[f]$ denotes the prior probability of function f.

2) The *likelihood* $\mathbb{P}[S \mid f]$ denotes how likely it is that a training sample $S = (X, Y)$ was obtained from a particular function f. The choice of likelihood is a modelling decision. The *zero-one likelihood* is common:

$$\mathbb{P}_{0/1}[S \mid f] := \mathbb{I}[f(X) = Y]. \tag{2.10}$$

3) The *posterior* $\mathbb{P}[f \mid S]$ denotes the probability of f in light of the training sample under the choice of likelihood.

Bayesian inference is generally intractable due to the high-dimensional summation or integration that appears in the denominator of Equation 2.9. But for Gaussian processes the required integrals are Gaussian in nature and often admit closed-form solutions. The following theorem is an important example:

Theorem 2.2. *Suppose that $f \sim \textsc{gp}(0, k)$. Conditioned on f interpolating dataset (X, Y), the distribution of f projected on to a fresh set of inputs X' is:*

$$f_{X'} \sim \textsc{normal}(K_{X'X} K_{XX}^{-1} Y, K_{X'X'} - K_{X'X} K_{XX}^{-1} K_{XX'}). \tag{2.11}$$
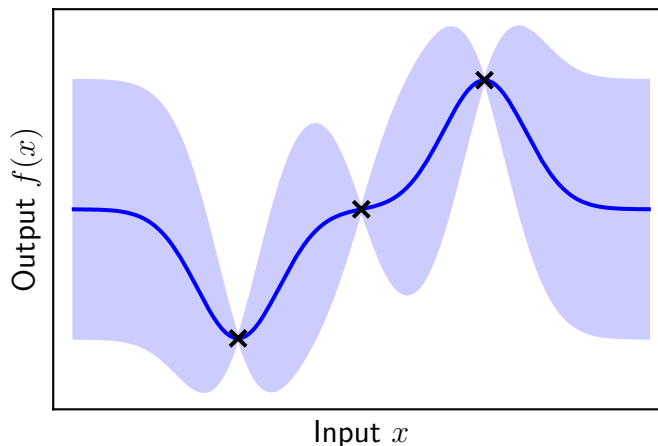
Figure 2.2: Posterior distribution of a Gaussian process, conditioned on fitting data marked by the black crosses. The kernel was set to the Gaussian kernel (Definition 2.1) and the likelihood was set to zero-one (Equation 2.10). The solid blue line depicts the posterior mean, and the shaded region represents ± 1 standard deviations about this mean.

Proof. Because $f \sim \mathrm{GP}(0, k)$, then $(f_X, f_{X'})$ is jointly Gaussian with covariance $K_{X \cup X', X \cup X'}$ under this prior. The conditional distribution of this multivariate Gaussian, given that $f_X = Y$, is given by Equation 2.11 (Bishop, 2006). $\square$

Theorem 2.2 allows one to sample functions from the Gaussian process conditioned on interpolating a set of m training examples. This process is illustrated in Figure 2.2. Just as was the case for kernel methods, the dominant computational cost is that of inverting the $m \times m$ kernel Gram matrix K_{XX} which is naïvely $\mathcal{O}(m^3)$. This means that Gaussian processes, like kernel methods, are said to have a cost that is cubic in the size of the training set.

2.3 Neural networks

The function spaces corresponding to kernel methods and Gaussian processes are both derived starting from a kernel function. This renders certain global properties of their function spaces amenable to analysis—for instance, one can write down the kernel interpolator that *globally minimises* the RKHS norm. Or one may write down a Gaussian process posterior distribution that includes within its support *all functions* that interpolate a particular training set. Neural networks eschew these properties, in favour of something else.

A neural network function space is constructed by composing many operators,

d_0 inputs d_1 units d_2 units   1 output
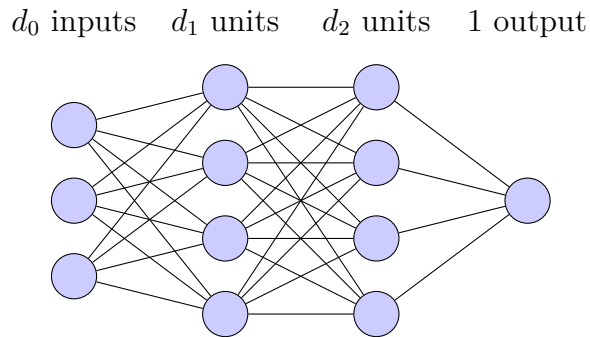
Figure 2.3:   A multilayer perceptron of depth $L = 3$.

where each operator has a set of weights. Adjusting the weights in each operator adjusts the function realised by the entire network. The choice of operators and how they are connected to each other is referred to as the *network architecture.*

Given a particular neural network architecture, it has so far been hard to characterise global properties of the space of functions that it realises, unlike kernel methods and Gaussian processes. Though some things can be said in certain limiting regimes, as will be discussed in Chapter 3.

Constructing a space of functions by composing parameterised operators
The simplest example of a neural network architecture is the *multilayer perceptron*—depicted in Figure 2.3. The multilayer perceptron comprises a composition of matrices interspersed by elementwise nonlinearities, meaning that it encapsulates many of the key features of more general neural networks. As such, it will serve as a *model organism* for detailed study in this thesis.

Definition 2.9 (Multilayer perceptron)**.** A *multilayer perceptron* f of depth L maps an input $x \in \mathbb{R}^{d_0}$ to an output $f(x; w) \in \mathbb{R}^{d_L}$ via the map:

$$f(x; w) \coloneqq W_L \circ (\varphi \circ W_{L-1}) \circ \cdots \circ (\varphi \circ W_1) \circ x. \tag{2.12}$$

In this expression φ denotes an elementwise nonlinearity, W_l denotes a matrix of dimension $d_l \times d_{l-1}$, and w denotes the tuple of L matrices $(W_1, ..., W_L)$.

Equation 2.12 provides a simple and direct means of constructing a space of functions. Without a nonlinearity, or with the nonlinearity set to the identity $\varphi \leftarrow \mathbf{I}$, the overall function would be linear. The canonical choice of nonlinearity is known as the *relu nonlinearity*:

Definition 2.10 (Relu nonlinearity)**.** The *relu nonlinearity* is given by:

$$\text{relu}(\cdot) := \max(0, \cdot). \tag{2.13}$$

So the relu nonlinearity retains the positive part of its input. It derives its name from the *rectified linear unit*. The relu nonlinearity is both simple and works well in applications (Nair and Hinton, 2010).

Measuring complexity via normalised margin

Unlike kernel methods and Gaussian processes, the tools for studying the complexity of neural network functions are not yet mature. Developing such tools and their understanding is one of the aims of this thesis. That said, there have been some fairly natural proposals. In the case of a binary classification problem, the *spectrally-normalised margin* is one such example:

Definition 2.11 (Spectrally-normalised margin)**.** Given a set of training data $S \in \{\mathbb{R}^{d_0} \times \pm 1\}^m$ and a multilayer perceptron $f : \mathbb{R}^{d_0} \times \mathcal{W} \to \mathbb{R}$ with matrices $w = (W_1, ..., W_L) \in \mathcal{W}$, the *spectrally-normalised margin* $\rho_\star$ is given by:

$$\rho_* := \min_{(x,y) \in S} \frac{f(x; w) \cdot y}{\|x\|_2 \cdot \prod_{l=1}^{L} \|W_l\|_*}. \tag{2.14}$$

The idea behind this definition is that, assuming that all training points are correctly classified, then the quantity $\min_{(x,y) \in S} f(x; w) \cdot y$ measures how close the closest training point is to being misclassified. But the problem with this measure is that a multilayer perceptron possesses various *rescaling symmetries*—in particular, scaling up the input or a weight matrix at any layer scales up the margin too. Normalising by the product of norms that appear in the denominator of Equation 2.14 yields a notion of margin that is invariant to these trivial rescaling symmetries. This definition of spectrally-normalised margin is related to one given by Bartlett et al. (2017).

Of course, there are other ways to measure margin modulo rescaling symmetries. For instance, the spectral norms appearing in Definition 2.11 measure the *largest* singular value of each weight matrix in the multilayer perceptron. One may just as well measure the *average* singular value. This motivates the following:

Definition 2.12 (Frobenius-normalised margin)**.** Given a set of training data $S \in \{\mathbb{R}^{d_0} \times \pm 1\}^m$ and a multilayer perceptron $f : \mathbb{R}^{d_0} \times \mathcal{W} \to \mathbb{R}$ with matrices

$w = (W_1, ..., W_L) \in \mathcal{W}$, let $\overline{d}_l := \min(d_l, d_{l-1})$ be the minimum dimension of matrix $W_l \in \mathbb{R}^{d_l \times d_{l-1}}$. Then the *Frobenius-normalised margin* ρ_F is given by:

$$\rho_F := \min_{(x,y) \in S} \frac{f(x; w) \cdot y}{\|x\|_2 \cdot \prod_{l=1}^{L} \|W_l\|_F / \sqrt{\overline{d}_l}}. \tag{2.15}$$

To understand this definition, note that the squared Frobenius norm $\|W_l\|_F^2$ of a matrix is equal to the sum of its squared singular values. Also, a matrix W_l has a number $\overline{d}_l$ of singular values in total. Therefore the Frobenius-normalised margin is just the spectrally-normalised margin with the largest singular value $\|W_l\|_*$ replaced by the root-mean-square singular value $\|W_l\|_F / \sqrt{\overline{d}_l}$. This definition is related to one given by Farhang et al. (2022).

But what do these notions of normalised margin have to do with the complexity of a neural network function? Consider a neural network that perfectly classifies a particular training set S, but with very small normalised margin. This network is, in a sense, *close* to a second network that misclassifies S. It would only take a small perturbation of the function realised by the former network to make it match the latter. Therefore, it seems reasonable that these two networks should have similar generalisation behaviour despite their differences on the training set. On the other hand, a network that classifies S perfectly and with large normalised margin is not close to a network that misclassifies S. Based on this line of thinking, it may seem reasonable to declare networks with small normalised margin *complex* on the grounds that they can mimic networks with different training error.

Of course, this is not a rigorous argument, and Definitions 2.11 and 2.12 are suggested only as candidate measures of complexity. Various modifications to these measures may work better. For example, one may consider replacing the minimum over the training set $\min_{(x,y) \in S}$ with the expectation $\mathbb{E}_{(x,y) \sim \text{UNIF}(S)}$. The role of these complexity measures in generalisation is studied further in Chapter 9.

Fitting data by gradient descent

To use a neural network in a machine learning application, one needs a way of selecting a neural network that fits a particular training set. In the classification setting, a natural goal is to seek a neural network that perfectly classifies the training set. This could be measured by the *zero-one loss*, say.

Definition 2.13 (Zero-one loss). For a neural network $f : \mathcal{X} \times \mathcal{W} \to \mathbb{R}$ and a training set $S \in (\mathcal{X} \times \pm 1)^m$, the *zero-one loss* of weight vector $w \in \mathcal{W}$ is:

$$\mathcal{L}_{0/1}(w) := \frac{1}{m} \sum_{(x,y) \in S} \mathbb{I}[\operatorname{sign} f(x; w) \neq y]. \tag{2.16}$$

Unfortunately, directly minimising the zero-one loss is not feasible, since its gradient with respect to the weights is either zero or undefined. Instead, a continuous proxy is used, such as the *square loss*:

Definition 2.14 (Square loss). For a neural network $f : \mathcal{X} \times \mathcal{W} \to \mathbb{R}$ and a training set $S \in (\mathcal{X} \times \pm 1)^m$, the *square loss* of weight vector $w \in \mathcal{W}$ is:

$$\mathcal{L}_2(w) := \frac{1}{2m} \sum_{(x,y) \in S} (f(x; w) - y)^2. \tag{2.17}$$

Observe that a neural network $f(\cdot, w)$ attaining square loss $\mathcal{L}_2(w) = 0$ also attains zero-one loss $\mathcal{L}_{0/1}(w) = 0$. But the square loss can be conveniently minimised via gradient descent. In spirit, such a procedure is given by:

$$w \leftarrow w - \eta \cdot \nabla_w \mathcal{L}_2(w), \tag{2.18}$$

where the constant $\eta > 0$ denotes the user-prescribed learning rate. For computational efficiency, the gradient of the loss with respect to only a sub-sample of data is typically used, rather than with respect to the full dataset. This is referred to as either *stochastic* or *mini-batch* gradient descent. The theory of *full-batch* gradient descent will be carefully studied in Part II.

The square loss also admits a compact description via function projection:

Proposition 2.1 (Square loss of projected function). *Consider a training set of m examples $S = (X, Y)$. Let $f_X(w)$ denote the projection (Definition 2.1) of neural network $f(\cdot; w)$ on to X. The square loss may be written:*

$$\mathcal{L}_2(w) = \frac{1}{2m} \cdot \|f_X(w) - Y\|_2^2. \tag{2.19}$$

Beyond square loss, other proxies for the zero-one loss are in use. For example:

Definition 2.15 (Logistic loss). For a neural network $f : \mathcal{X} \times \mathcal{W} \to \mathbb{R}$ and a training set $S \in (\mathcal{X} \times \pm 1)^m$, the *logistic loss* of weight vector $w \in \mathcal{W}$ is:

$$\mathcal{L}_{\log}(w) := \frac{1}{m} \sum_{(x,y) \in S} \log \left(1 + e^{-y \cdot f(x; w)}\right). \tag{2.20}$$

The logistic loss is *margin maximising* (Rosset et al., 2003). This means that in contrast to the square loss, which is minimised by setting the training outputs to fixed values, the logistic loss is reduced by making the outputs on correctly classified training points larger in magnitude.

A margin-maximising loss function would tend to increase the notions of normalised margin given in Definitions 2.11 and 2.12—at least provided the norms of the weights in the network do not themselves grow. To prevent weight norms growing, an *L2 penalty* is often added to the loss function:

Definition 2.16 (L2 penalty). Given a neural network $f : \mathcal{X} \times \mathcal{W} \to \mathcal{Y}$ with L layers and weight tuple $w = (W_1, ..., W_L) \in \mathcal{W}$, the *L2 penalty* is given by:

$$\|w\|_2^2 := \sum_{l=1}^{L} \|W_l\|_F^2. \tag{2.21}$$

In words, the L2 penalty penalises the size of the Frobenius norm of the weight matrix at each layer. To train a network, one might then minimise the following *L2-regularised* loss function:

$$\mathcal{L}_{\log}(w) + \lambda \cdot \|w\|_2^2. \tag{2.22}$$

This is often referred to as adding *weight decay*, since in the gradient descent update the size of the weights are damped by a factor $(1 - \eta\lambda)$ at each iteration:

$$w \leftarrow w \cdot (1 - \eta\lambda) - \eta \cdot \nabla_w \mathcal{L}_{\log}(w). \tag{2.23}$$

The final comment of this chapter is that it has so far been difficult to theoretically characterise the computational cost of neural network training, although efforts have been made to obtain empirical scaling laws (Kaplan et al., 2020). What can be said is that neural network training appears to overcome the unfavourable cubic cost of kernel methods and Gaussian processes.

References

Peter L. Bartlett, Dylan J. Foster, and Matus J. Telgarsky (2017). "Spectrally-normalized margin bounds for neural networks". In: *Neural Information Processing Systems*.

Christopher M. Bishop (2006). *Pattern Recognition and Machine Learning*. Springer-Verlag.

Alexander R. Farhang, Jeremy Bernstein, Kushal Tirumala, Yang Liu, and Yisong Yue (2022). "Investigating generalization by controlling normalized margin". In: *International Conference on Machine Learning.*

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei (2020). "Scaling laws for neural language models". In: *arXiv:2001.08361.*

Vinod Nair and Geoffrey E. Hinton (2010). "Rectified linear units improve restricted Boltzmann machines". In: *International Conference on Machine Learning.*

Radford M. Neal (1994). "Bayesian learning for neural networks". Ph.D. thesis. Department of Computer Science, University of Toronto.

Saharon Rosset, Ji Zhu, and Trevor Hastie (2003). "Margin maximizing loss functions". In: *Neural Information Processing Systems.*

Bernhard Schölkopf and Alexander J. Smola (2001). *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond.* MIT Press.

CORRESPONDENCES BETWEEN FUNCTION SPACES

> This chapter introduces various useful correspondences between kernel methods, Gaussian processes and neural networks. The material is expositional and is included for the reader's aid.

Kernel methods, Gaussian processes and neural networks were introduced in Chapter 2 as different philosophical approaches to constructing spaces of functions that may be used to fit data. Kernel methods construct functions by linearly combining kernel basis functions, and are thus amenable to a particular kind of functional analysis. Gaussian processes draw functions from a probability measure, allowing the laws of probability to be applied to both analyse complexity and fit data. Neural networks eschew these analytical considerations, preferring a very flexible means of building functions by composing parameterised operators that may be fit to data via gradient descent.

It may seem surprising, then, that multiple connections exist between these seemingly disparate spaces of functions. These *correspondences* provide a promising route toward better understanding each function space individually.

3.1 GP posterior mean is a kernel interpolator

One of the attractive features of a Gaussian process is that, after observing data, a full posterior distribution over possible explanations of the data may be obtained. But sometimes this full posterior distribution is overkill—for example, in some situations all that is needed is the single *best prediction* for a fresh test input. In these situations, it may be enough to simply return the mean of the posterior distribution (solid blue line in Figure 2.2).

Perhaps surprisingly, the mean of a Gaussian process posterior distribution is equivalent to the kernel interpolator of minimum RKHS norm, where the kernel is set to to the covariance function of the Gaussian process. Formally, suppose that $f \sim \mathrm{GP}(0, k)$. Conditioned on f interpolating a training set $S = (X, Y)$, by Theorem 2.2 the mean prediction on a fresh input x is given by:

$$\mathbb{E}\left[f(x) \mid f_X = Y\right] = K_{xX} K_{XX}^{-1} Y, \tag{3.1}$$

But by Theorem 2.1, this is equivalent to the minimum RKHS norm kernel interpolator of (X, Y) with kernel k.

3.2 GP posterior variance bounds the error of kernel interpolation

A slightly more subtle relationship between Gaussian processes and kernel methods connects the posterior variance of a Gaussian process to the worst case error of kernel interpolation. This correspondence is based on the fact that the Gaussian process posterior variance admits the following geometric interpretation in the corresponding RKHS:

Lemma 3.1 (Distance between a point and a subspace). *Consider an RKHS induced by kernel $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$. Given a set of inputs $X = \{x_1, ..., x_m\}$ and fresh input x, the shortest distance between the kernel basis function centred on x and the span of the kernel basis functions centred on points in X is given by:*

$$\operatorname{dist}^2\Big(k(\cdot, x), \operatorname{span}\{k(\cdot, x_i)\}_{i=1}^m\Big) := \min_{\alpha \in \mathbb{R}^m} \left\| k(\cdot, x) - \sum_{i=1}^m \alpha_i \, k(\cdot, x_i) \right\|_{\mathrm{RKHS}}^2$$

$$= K_{xx} - K_{xX} K_{XX}^{-1} K_{Xx}. \tag{3.2}$$

In words: the Gaussian process posterior variance (right-hand side of Equation 3.2) measures the shortest RKHS-distance from the kernel basis function centred at x and the span of the kernel basis functions centred on the training points.

Proof of Lemma 3.1. Evaluating the RKHS norm for arbitrary $\alpha \in \mathbb{R}^m$ yields:

$$\left\| k(\cdot, x) - \sum_{i=1}^m \alpha_i \, k(\cdot, x_i) \right\|_{\mathrm{RKHS}}^2 = k_{xx} - 2 \cdot \alpha^\top K_{Xx} + \alpha^\top K_{XX} \alpha.$$

Setting the derivative with respect to α to zero yields that $K_{Xx} = K_{XX} \alpha$ and hence the RKHS norm is minimised by setting $\alpha = K_{XX}^{-1} K_{Xx}$. Substituting this result back into the expression for the RKHS norm yields the result. $\square$

With Lemma 3.1 in hand, the next theorem follows readily. The result is related to Corollary 3.11 of Kanagawa et al. (2018).

Theorem 3.1 (Error of kernel interpolation). *Given a function g in an RKHS induced by kernel k and any set of distinct inputs X, the deviation between g and the minimum RKHS norm interpolator $f_\star$ of $(X, g(X))$ satisfies:*

$$|g(x) - f_\star(x)| \leq \sqrt{\|g\|_{\mathrm{RKHS}}^2 - \|f_\star\|_{\mathrm{RKHS}}^2} \cdot \sqrt{K_{xx} - K_{xX} K_{XX}^{-1} K_{Xx}}. \tag{3.3}$$

Proof. By the reproducing property, the fact that g and $f_\star$ agree on the training points, and the Cauchy-Schwarz inequality:

$$
\begin{aligned}
|g(x) - f_\star(x)| &= |\langle g - f_\star, k(\cdot, x)\rangle_{\text{RKHS}}| \\
&= \left| \langle g - f_\star, k(\cdot, x) - \sum_{i=1}^{m} \alpha_i \, k(\cdot, x_i)\rangle_{\text{RKHS}} \right| \text{ for all } \alpha \in \mathbb{R}^m \\
&\leq \|g - f_\star\|_{\text{RKHS}} \cdot \min_{\alpha \in \mathbb{R}^m} \left\| k(\cdot, x) - \sum_{i=1}^{m} \alpha_i \, k(\cdot, x_i) \right\|_{\text{RKHS}}.
\end{aligned}
$$

The result follows by observing that $\|g\|_{\text{RKHS}}^2 = \|f_\star\|_{\text{RKHS}}^2 + \|g - f_\star\|_{\text{RKHS}}^2$ by Equation 2.8, and an application of Lemma 3.1. $\qquad\square$

In words: if one is approximating an unknown function g by the the minimum RKHS norm interpolator $f_\star$ of a set of samples from g, then the error of this approximation is bounded by the posterior standard deviation of the corresponding Gaussian process fit to those samples, scaled by a constant depending on the difference in RKHS norm between g and approximation $f_\star$.

3.3 Making the GP posterior concentrate on a kernel interpolator

A particular kind of aggregate Gaussian process prediction—the posterior mean—is directly available via Equation 3.1. As mentioned in Section 3.1, the posterior mean may suffice for practical applications and often one may not bother computing the posterior variance.

But for other function spaces, it may be the case that the posterior distribution may only be accessed through samples—and these samples may be expensive to obtain. In fact, Chapter 9 will argue that a neural network trained to fit data by gradient descent is analogous to a *single draw* from a Gaussian process posterior distribution. In this situation, computing the posterior mean would involve averaging over many samples, which might be prohibitively expensive.

The following theorem observes that, for Gaussian processes, this issue may be circumvented—by forcing the entire posterior distribution to concentrate on its mean. In this case, *all* posterior samples may be trusted to faithfully report the mean. This mean is itself a kernel interpolator by the results of Section 3.1.

Theorem 3.2 (Posterior concentration). *Given a kernel k and a training sample (X, Y), define two constants: a "margin" $\gamma > 0$ and a "normalisation" $\tau^2 > 0$. Sample a function $f \sim \text{GP}(0, \tau^2 \cdot k)$ conditioned on f interpolating $(X, \gamma \cdot Y)$:*

$$
f \sim \text{GP}(0, \tau^2 \cdot k \mid f_X = \gamma \cdot Y). \tag{3.4}
$$

In the limit that the "normalised margin" $\gamma/\tau \to \infty$, then the rescaled function f/γ converges to the posterior mean:

$$f(x)/\gamma = K_{xX}K_{XX}^{-1}Y, \text{ with probability one.} \qquad (3.5)$$

Proof. By Theorem 2.2, the sampled function f evaluated at x follows:

$$f(x) \sim \text{NORMAL}\left(K_{xX}K_{XX}^{-1}(\gamma \cdot Y), \tau^2 \cdot (K_{xx} - K_{xX}K_{XX}^{-1}K_{Xx})\right).$$

Dividing through by γ then yields:

$$f(x)/\gamma \sim \text{NORMAL}\left(K_{xX}K_{XX}^{-1}Y, \frac{\tau^2}{\gamma^2} \cdot (K_{xx} - K_{xX}K_{XX}^{-1}K_{Xx})\right).$$

Taking the normalised margin $\gamma/\tau \to \infty$ causes the variance to vanish. $\qquad \square$

The language used in this theorem statement surrounding "normalised margin" is intended to evoke the concepts of normalised margin given in Definitions 2.11 and 2.12. This foreshadows a contribution of Chapter 9, where a formal connection is made between Theorem 3.2 and a notion of normalised margin in neural networks. This formal connection leverages a correspondence between neural networks and Gaussian processes, which is presented in the next section.

3.4 Neural network–Gaussian process correspondence

Informally, the *neural network–Gaussian process* (NNGP) correspondence states that the space of functions realised by a sufficiently wide neural network is equivalent to a Gaussian process. Since a key feature of a Gaussian process is that it assigns probabilities to functions, this informal statement is not yet fully meaningful. What is needed is a means of assigning probabilities to the functions realised by a neural network. The key idea is to consider randomly sampling the network weights to obtain a distribution over neural network functions. This leads to the more accurate—but still informal—statement:

> The distribution over functions realised by a wide enough neural network with randomly drawn weights is a Gaussian process.

While this correspondence may seem surprising, the mechanism by which it works ends up being fairly simple. Consider the lth layer of a neural network:

$$f_l(x) = W_l \cdot \varphi(f_{l-1}(x)), \qquad (3.6)$$

where $x \in \mathbb{R}^{d_0}$ is the network input, $f_l(x) \in \mathbb{R}^{d_l}$ is the layer output, $W_l \in \mathbb{R}^{d_l \times d_{l-1}}$ is the weight matrix, φ is the nonlinearity and $f_{l-1}(x) \in \mathbb{R}^{d_{l-1}}$ is the layer input. Suppose that the different components of $f_{l-1}(x)$ are iid random variables and that the entries of weight matrix W_l are also drawn iid. Then the components of the output $f_l(x)$, being the sum over a large number of iid contributions, are themselves iid Gaussian by the central limit theorem.

This idea is formalised in the following lemma, essentially due to Neal (1994).

Lemma 3.2 (NNGP correspondence). *For the neural network layer given by Equation 3.6, if the following conditions hold:*

(i) *Inputs: For every network input $x \in \mathbb{R}^{d_0}$, the components of the layer input $f_{l-1}(x) \in \mathbb{R}^{d_{l-1}}$ are iid with finite first and second moment.*

(ii) *Weights: Entries of W_l are iid with zero mean and variance $\sigma^2/d_{l-1} < \infty$.*

(iii) *Nonlinearity: For any random variable z with finite first and second moment, $\varphi(z)$ also has finite first and second moment.*

Then, in the limit of width $d_{l-1} \to \infty$, the distribution of layer outputs satisfies:

1) *IID outputs. For every input $x \in \mathbb{R}^{d_0}$, the components of the layer output $f_l(x)$ for that input are iid with finite first and second moment.*

2) *Gaussian outputs. For any collection of m network inputs $x_1, ..., x_m$, the output components $f_l^i(x_1), ..., f_l^i(x_m)$ are jointly Gaussian for $i = 1, ..., d_l$.*

Condition (ii) is satisfied simply by sampling the network weights iid Gaussian, say. Similarly, condition (iii) is easy to check for a given nonlinearity. Condition (i) on the layer inputs seems the most non-trivial. But notice that the lemma propagates this condition to the next layer via entailment 1). This means that provided condition (i) holds at the first layer, then recursive application of the lemma implies that condition (i) will hold at all layers.

Entailment 2) is equivalent to saying that the ith component of the layer output forms a Gaussian process (Definition 2.8). The kernel of this Gaussian process depends on the specific architecture of the network preceding the lth layer. For example, the kernel for multilayer perceptrons with relu nonlinearity is given by Theorem 3.3. But first, a proof of the lemma is given.

Proof of Lemma 3.2. To prove entailment 1), write output $f_l(x) \in \mathbb{R}^{d_l}$ as:

$$f_l(x) = \sum_{j=1}^{d_{l-1}} \left[W_l^{1j}, ..., W_l^{d_l j} \right] \cdot \varphi \left(f_{l-1}^j(x) \right) =: \sum_{j=1}^{d_{l-1}} v_j. \tag{3.7}$$

By conditions (i) and (ii), the summands v_j in Equation 3.7 are iid random vectors. To apply a central limit theorem, these random vectors must have finite mean and variance. Since the weights and inputs are independent, the mean is finite and is given by:

$$\mathbb{E}[v_j] = \mathbb{E}[W_l^{1j}, ..., \mathbb{E}\, W_l^{d_l j}] \cdot \mathbb{E}[\varphi(f_{l-1}^j(x))] = 0 \cdot \mathbb{E}[\varphi(f_{l-1}^j(x))] = 0.$$

This result relies on the quantity $\mathbb{E}[\varphi(f_{l-1}^j(x))]$ being finite by conditions (i) and (iii). Similarly, the variance is given by:

$$\mathbb{E}[v_j^i v_j^k] = \mathbb{E}[W_l^{ij} W_l^{kj}] \cdot \mathbb{E}[\varphi(f_{l-1}^j(x))^2] = \delta_{ik} \cdot \sigma^2/d_{l-1} \cdot \mathbb{E}[\varphi(f_{l-1}^j(x))^2],$$

where δ_{ik} is the Kronecker delta. Since $\sigma^2/d_{l-1} < \infty$ and, by conditions (i) and (iii), the quantity $\mathbb{E}[\varphi(f_{l-1}^j(x))^2]$ appearing on the right-hand side is finite too, then the variance of random vector v_j is finite. In turn, by the multivariate central limit theorem (Vaart, 1998), in the limit that $d_{l-1} \to \infty$, the layer output $f_l(x) \sim \text{NORMAL}(0, \sigma^2 \cdot \mathbb{E}[\varphi(f_{l-1}^1(x))^2] \cdot \mathbf{I})$. In particular, this implies that the components of $f_l(x)$ are iid with finite first and second moment.

Entailment 2) is established by a similar argument that considers the ith component of the layer output f_l projected on to m samples $X = \{x_1, ..., x_m\}$:

$$(f_l^i)_X = \sum_{j=1}^{d_{l-1}} W_l^{ij} \cdot \left[\varphi \left(f_{l-1}^j(x_1) \right), ..., \varphi \left(f_{l-1}^j(x_m) \right) \right]. \tag{3.8}$$

Again, by combining conditions (i), (ii) and (iii), the summands in Equation 3.8 are iid random vectors with finite mean and finite covariance. Then as $d_{l-1} \to \infty$, the distribution of $(f_l^i)_X$ is multivariate Gaussian—again by the multivariate central limit theorem. This completes the proof. $\square$

Working out the details of the neural network–Gaussian process correspondence for a specific network architecture involves computing the Gaussian process kernel that is induced by the given network topology and choice of nonlinearity. The following theorem demonstrates this process for the *model organism* of this thesis: the multilayer perceptron with relu nonlinearity. The essence of the following theorem appears in a paper by Lee et al. (2018), building on the work of Cho and Saul (2009).

Theorem 3.3 (NNGP for relu networks). *Consider a multilayer perceptron f (Definition 2.9) with L layers, output dimension $d_L = 1$ and nonlinearity:*

$$\varphi(\cdot) = \sqrt{2} \cdot \max(0, \cdot). \tag{3.9}$$

For each layer $l = 1, ..., L$, sample weight entries $W_l^{ij} \overset{\text{iid}}{\sim} \text{NORMAL}(0, 1/d_{l-1})$. Consider any collection X of m inputs constrained to the hypersphere of radius $\sqrt{d_0}$: $x_1, ..., x_m \in \sqrt{d_0} \cdot \mathbb{S}^{d_0-1}$. Then, as hidden layer widths $d_1, ..., d_{L-1} \to \infty$, the network outputs $f(x_1), ..., f(x_m)$ are jointly Gaussian with:

$$\mathbb{E}\left[f(x_i)\right] = 0; \tag{3.10}$$

$$\mathbb{E}\left[f(x_i)^2\right] = 1; \tag{3.11}$$

$$\mathbb{E}\left[f(x_i)f(x_j)\right] = \underbrace{h \circ ... \circ h}_{L-1 \ times}\left(\frac{x_i^\top x_j}{d_0}\right); \tag{3.12}$$

for all $x_i, x_j \in X$, and where $h(t) := \frac{1}{\pi}\left[\sqrt{1 - t^2} + t \cdot (\pi - \arccos t)\right]$.

The kernel appearing in Equation 3.12 is the *compositional arccosine kernel* (Cho and Saul, 2009). This kernel will be used in Chapter 9 to study generalisation in the multilayer perceptron.

Proof. First, Lemma 3.2 will be applied recursively over the layers of the network. Condition (ii) of Lemma 3.2 holds at all layers with $\sigma^2 = 1$, and condition (iii) holds trivially for the scaled relu nonlinearity of Equation 3.9. Condition (i) holds at the first layer since, in the notation of Equation 3.6,

$$\mathbb{E}[f_1(x)] = \mathbb{E}[W_1 \cdot x] = \mathbb{E}[W_1] \cdot x = 0,$$

$$\mathbb{E}[(f_1^i(x))^2] = \sum_{j,k=1}^{d_0} \mathbb{E}[W_1^{ij}W_1^{ik}] \cdot x^j x^k = \sum_{j=1}^{d_0} \mathbb{E}[(W_1^{ij})^2] \cdot (x^j)^2 = 1, \tag{3.13}$$

and φ preserves both iid-ness and finite-ness of the first and second moment. Then, by recursive application of Lemma 3.2, the outputs at any layer are jointly Gaussian. All that remains is to compute their moments.

For the ith component of layer l, the first moment $\mathbb{E}\left[f_l^i(x)\right] = 0$. This can be seen by taking the expectation of Equation 3.6 and using the fact that the W_l^{ij} are independent of the layer inputs and have mean zero.

Since any two layer outputs $f_l^i(x)$ and $f_l^i(x')$ are jointly Gaussian with mean zero, their distribution is completely described by the 2×2 covariance matrix:

$$\Sigma_l(x, x') := \begin{bmatrix} \rho_l(x, x) & \rho_l(x, x') \\ \rho_l(x, x') & \rho_l(x', x') \end{bmatrix},$$

where $\rho_l(x, x') := \mathbb{E}\left[f_l^i(x) f_l^i(x')\right]$ and the index i is unimportant since different components in the same layer have identical distributions.

The theorem statement will follow from an effort to express $\Sigma_l(x, x')$ in terms of $\Sigma_{l-1}(x, x')$ and then recursing back through the network. By Equation 3.6 and independence of the W_l^{ij}, the covariance $\rho_l(x, x')$ may be expressed as:

$$\rho_l(x, x') = \mathbb{E}\left[\varphi\left(f_{l-1}^j(x)\right) \varphi\left(f_{l-1}^j(x')\right)\right], \tag{3.14}$$

where j indexes an arbitrary component of layer $l - 1$. To make progress, it helps to first evaluate:

$$\rho_l(x, x) = \mathbb{E}\left[\varphi\left(f_{l-1}^j(x)\right)^2\right] = \tfrac{1}{2} \cdot 2 \cdot \rho_{l-1}(x, x),$$

which follows by the definition of φ and symmetry of the Gaussian expectation around zero. Then, by recursion:

$$\rho_l(x, x) = \rho_{l-1}(x, x) = \ldots = \rho_1(x, x) = 1,$$

where the final equality holds because $\mathbb{E}[(f_1^i(x))^2] = 1$ by Equation 3.13. Then the covariance Σ_{l-1} at layer $l - 1$ simplifies to:

$$\Sigma_{l-1}(x, x') = \begin{bmatrix} 1 & \rho_{l-1}(x, x') \\ \rho_{l-1}(x, x') & 1 \end{bmatrix}.$$

Equation 3.14 may now be used to express $\rho_l(x, x')$ in terms of $\rho_{l-1}(x, x')$. Dropping the (x, x') indexing for brevity:

$$\rho_l = \mathbb{E}_{u,v \sim \mathcal{N}(0, \Sigma_{l-1})}\left[\varphi\left(u\right) \varphi\left(v\right)\right]$$

$$= \frac{1}{\pi\sqrt{1 - \rho_{l-1}^2}} \iint_{u,v \geq 0} du\, dv\, \exp\left[-\frac{u^2 - 2\rho_{l-1}uv + v^2}{2(1 - \rho_{l-1}^2)}\right] uv.$$

By making the substitution $\rho_{l-1} = \cos\theta$, this integral becomes equivalent to $\frac{1}{\pi} \cdot J_1(\theta)$ as expressed in Cho and Saul (2009)'s Equation 15. Substituting in the evaluation of this integral (Cho and Saul, 2009, Equation 6), one obtains:

$$\rho_l(x, x') = h(\rho_{l-1}(x, x')). \tag{3.15}$$

What remains is to evaluate $\rho_1(x, x')$. Since $\mathbb{E}\left[W_1^{ij} W_1^{ik}\right] = \delta_{jk}/d_0$, this equals:

$$\rho_1(x, x') := \mathbb{E}\left[f_1^i(x) f_1^i(x')\right] = \sum_{j,k=1}^{d_0} \mathbb{E}\left[W_1^{ij} W_1^{ik}\right] x^j (x')^k = \frac{x^\top x'}{d_0}.$$

The proof is completed by combining this expression for $\rho_1(x, x')$ with the recurrence relation in Equation 3.15. $\qquad\square$

This completes the chapter on correspondences between function spaces.

References

Youngmin Cho and Lawrence K. Saul (2009). "Kernel methods for deep learning". In: *Neural Information Processing Systems.*

Motonobu Kanagawa, Philipp Hennig, Dino Sejdinovic, and Bharath K. Sriperumbudur (2018). "Gaussian processes and kernel methods: A review on connections and equivalences". In: *arXiv:1807.02582.*

Jaehoon Lee, Jascha Sohl-Dickstein, Jeffrey Pennington, Roman Novak, Sam Schoenholz, and Yasaman Bahri (2018). "Deep neural networks as Gaussian processes". In: *International Conference on Learning Representations.*

Radford M. Neal (1994). "Bayesian learning for neural networks". Ph.D. thesis. Department of Computer Science, University of Toronto.

Aad W. van der Vaart (1998). *Asymptotic Statistics.* Cambridge University Press.

Part II

OPTIMISATION
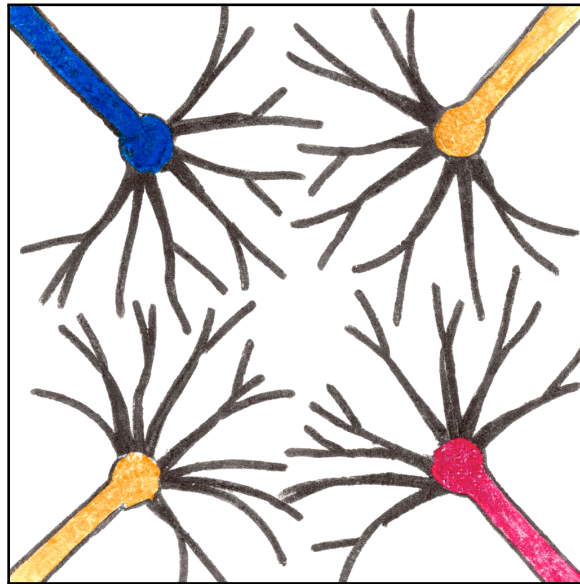
"*The number of open questions concerning steepest ascent, its ubiquity in the literature of optimization, and its continued use in computation might lead the unwary to think that it was a good thing to do in practice; but we think that in the art of computation it should be considered as a last resort, faute de mieux, as Cauchy might have said.*"

Philip Wolfe, 1969

Chapter 4

THE MAJORISE-MINIMISE META-ALGORITHM

> This chapter introduces several classic iterative optimisation methods. The derivations, while themselves not novel, are put on common footing by showing how each corresponds to a form of perturbation analysis.

This chapter surveys classic techniques for formally deriving gradient-based optimisation methods. The survey covers first-order methods—such as *gradient descent* and *mirror descent*—as well as second-order methods—such as the *cubic regularised* version of *Newton's method*. These varied techniques are put on a consistent theoretical footing by showing that one step of each method may be viewed as the minimisation of a particular local description of the loss function. In each case, this local description takes the form of a *perturbation bound* on the error of a *truncated perturbation expansion* of the loss. Such bounds, known as *majorisations*, assess the region within which the perturbation expansion can be trusted.

The optimisation methods described in this chapter make use of perturbation expansions and bounds related to the Taylor series expansion of the loss function $\mathcal{L}$ in the weight space $\mathcal{W}$ of the optimisation problem. As such, they may construct a perturbation $\Delta w \in \mathcal{W}$ to the weights $w \in \mathcal{W}$ by considering any of the following information:

1) *First-order information*: The gradient $\nabla_w \mathcal{L}(w)$ of the loss function.

2) *Second-order information*: The Hessian $\nabla_w^2 \mathcal{L}(w)$ of the loss function.

3) *Trust regions*: Bounds on the accuracy of the above information.

In contrast to the techniques considered in this chapter, Chapter 5 deals more closely with neural networks. There the loss $\mathcal{L}$ depends on the weights $w \in \mathcal{W}$ indirectly via the network architecture $f(\cdot; w) : \mathcal{X} \to \mathcal{Y}$, and it will help to model this dependence. But in this chapter, the loss function will be viewed as a straightforward function of the weights—formally, the loss $\mathcal{L} : \mathcal{W} \to \mathbb{R}$.

4.1 Perturbation analysis: Expansions and bounds

This section introduces the general ideas of perturbation analysis. Section 4.2 will show how these ideas may be used to solve optimisation problems.

Perturbation expansions

The simplest example of a perturbation expansion is a straightforward Taylor series. Provided a function $g : \mathbb{R} \to \mathbb{R}$ is analytic, it may be expanded about a point $x \in \mathbb{R}$ as a Taylor series in powers of some small perturbation $\delta x \in \mathbb{R}$:

$$g(x + \delta x) = g(x) + \frac{\partial g}{\partial x} \cdot \delta x + \frac{1}{2} \frac{\partial^2 g}{\partial x^2} \cdot \delta x^2 + ... \tag{4.1}$$

A *truncated* perturbation expansion refers to cutting off this series at some order. For instance, truncating Equation 4.1 yields:

$$g^{(2)}(x + \delta x) := g(x) + \frac{\partial g}{\partial x} \cdot \delta x + \frac{1}{2} \frac{\partial^2 g}{\partial x^2} \cdot \delta x^2, \tag{4.2}$$

where the superscript (2) indicates a second-order truncation.

Perturbation bounds

While the truncated perturbation expansion in Equation 4.2 is accurate for sufficiently small δx, the error in this approximation is unknown without computing the truncated part of the series. A *perturbation bound* deals with this issue by bounding the error in a truncated perturbation expansion. For example, if one can derive a function $h(x, \delta x)$ such that:

$$\left| g(x + \delta x) - g^{(2)}(x + \delta x) \right| \leq h(x, \delta x), \tag{4.3}$$

then Inequality 4.3 would constitute a perturbation bound. For the perturbation bound to be useful, one aspires to finding a bounding function $h(\cdot, \cdot)$ that:

1) provides a reasonably tight bound on the error in the truncated series.

2) is much cheaper to compute than the omitted terms in the truncation.

4.2 Majorise-minimise

In an iterative minimisation method, one is interested in selecting a perturbation Δw_* such that the loss after the perturbation, $\mathcal{L}(w + \Delta w_*)$, is smaller than the loss beforehand, $\mathcal{L}(w)$. A good starting point for the design of such a method is the Taylor expansion of the loss in weight perturbations Δw:

$$\mathcal{L}(w + \Delta w) = \mathcal{L}(w) + \nabla_w \mathcal{L}(w)^\top \Delta w + \frac{1}{2} \cdot \Delta w^\top \nabla_w^2 \mathcal{L}(w) \Delta w + ... \tag{4.4}$$

It is tempting to minimise, say, the first few terms in this Taylor expansion as a proxy for reducing the loss function. Formally, letting $\mathcal{L}^{(k)}(w + \Delta w)$ denote the Taylor expansion in Equation 4.4 truncated to kth order, one might be inclined to select a perturbation Δw_* via:

$$\Delta w_* = \arg\min_{\Delta w} \left[\mathcal{L}^{(k)}(w + \Delta w) \right]. \tag{4.5}$$

For instance, truncating to first order would correspond to the minimisation:

$$\Delta w_* = \arg\min_{\Delta w} \left[\mathcal{L}^{(1)}(w + \Delta w) \right] = \arg\min_{\Delta w} \left[\mathcal{L}(w) + \nabla_w \mathcal{L}(w)^\top \Delta w \right]. \tag{4.6}$$

Unfortunately, this procedure is not well-founded. To see this, observe that the minimand appearing in Equation 4.6 can be made arbitrarily negative by setting the perturbation Δw to point in the direction of the negative gradient $-\nabla_w \mathcal{L}(w)$ with an arbitrarily large magnitude. This holds even for bounded loss functions which cannot themselves be made arbitrarily negative.

The core issue being highlighted in the previous paragraph is that minimising a truncated series expansion can result in perturbations so large that the truncation is no longer a good approximation to the original loss function. This issue may be addressed by employing a special form of perturbation bound known as a *majorisation*:

Definition 4.1 (Majorisation). Given an analytic loss function $\mathcal{L} : \mathcal{W} \to \mathbb{R}$, let $\mathcal{L}^{(k)}(w + \Delta w)$ denote the Taylor expansion of $\mathcal{L}$ about w truncated to kth order. An analytic function $h : \mathcal{W} \times \mathcal{W} \to \mathbb{R}_{\geq 0}$ is a *majorisation* of $\mathcal{L}$ provided that:

(i) h gives a one-sided bound on the error of the truncated Taylor series:

$$\mathcal{L}(w + \Delta w) \leq \mathcal{L}^{(k)}(w + \Delta w) + h(w, \Delta w). \tag{4.7}$$

(ii) h is zero whenever the perturbation Δw is zero:

$$h(w, 0) = 0 \text{ for all } w \in \mathcal{W}. \tag{4.8}$$

Taken together, these two conditions imply that a majorisation provides an upper bound to the perturbed loss $\mathcal{L}(w + \Delta w)$ (Inequality 4.7) that lies *tangent* to the loss $\mathcal{L}$ at weight setting w. A graphical example is provided in Figure 4.1. The reason that such a construction is helpful is that minimising a majorisation will also reduce the original loss. This idea is also visualised in Figure 4.1.
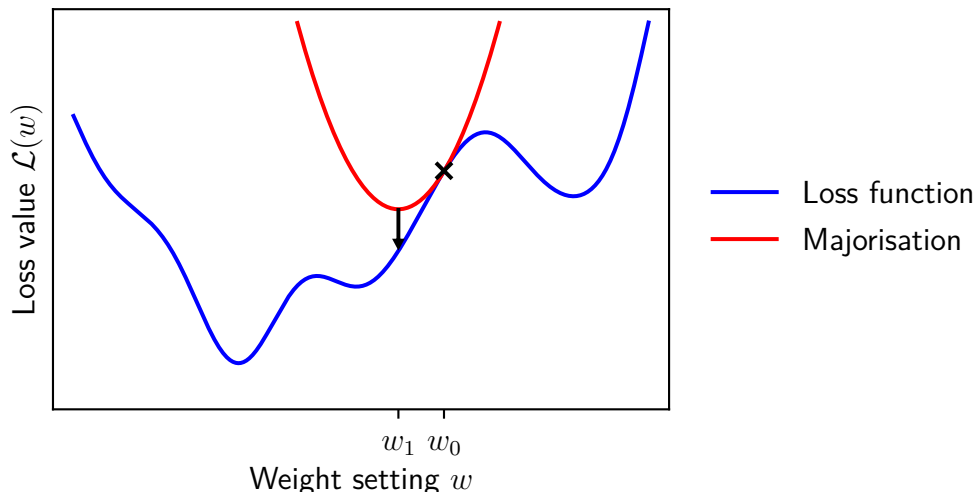
Figure 4.1: The majorise-minimise meta-algorithm. The blue curve denotes a loss function that one would like to reduce, starting from a point w_0. The red curve denotes a *majorisation* of the loss about w_0. Since the majorisation is both an upper bound to the loss that is also tangent at w_0, minimising the majorisation to obtain a new weight setting w_1 also reduces the original loss.

Formally, if h is a majorisation and a perturbation Δw_* is selected via:

$$\Delta w_* = \arg\min_{\Delta w} \left[\mathcal{L}^{(k)}(w + \Delta w) + h(w, \Delta w) \right], \tag{4.9}$$

then $\mathcal{L}(w + \Delta w_*) \leq \mathcal{L}(w)$, with equality only when w was already a minimum.

This process, of minimising a tangent upper bound to a loss function, is known as the *majorise-minimise* meta-algorithm (Lange, 2016). It is a *meta-algorithm* in the sense that many different methods may be derived by following this procedure in different situations. Examples are given in the next section.

4.3 Instantiations of the meta-algorithm

First-order optimisation methods minimise a majorisation of the first-order Taylor expansion of the loss function. They therefore rely on first-order *gradient* information and do not have access to second-order *Hessian* information. Examples include basic gradient descent and also *mirror descent.*

Meanwhile, second-order optimisation methods minimise a majorisation of the second-order Taylor expansion of the loss function. Both first-order *gradient* information as well as second-order *Hessian* information are used. The hope is that including this extra information will lead to a more effective optimisation step. One example is the *cubic-regularised* version of *Newton's method.*

First example: Gradient descent

This subsection shows that gradient descent in its most basic form arises from the following majorisation of the first-order Taylor expansion of the loss:

$$\mathcal{L}(w + \Delta w) \leq \mathcal{L}(w) + \nabla_w \mathcal{L}(w)^\top \Delta w + \frac{\lambda}{2} \cdot \|\Delta w\|_2^2, \qquad (4.10)$$

for some constant $\lambda > 0$. This is a *Euclidean* majorisation, since the Euclidean norm $\|\cdot\|_2$ characterises the realm of validity of the first-order Taylor expansion.

One must ask, for which loss functions is Inequality 4.10 a valid majorisation? Here it helps to define the class of *gradient-Lipschitz* loss functions.

Definition 4.2 (Gradient-Lipschitz loss function). A loss function $\mathcal{L} : \mathbb{R}^d \to \mathbb{R}$ is *gradient-Lipschitz* with constant $\lambda > 0$ if:

$$\|\nabla_w \mathcal{L}(w + \Delta w) - \nabla_w \mathcal{L}(w)\|_2 \leq \lambda \cdot \|\Delta w\|_2. \qquad (4.11)$$

The following lemma shows that a gradient Lipschitz loss function is majorised according to Inequality 4.10.

Lemma 4.1 (Gradient-Lipschitz Euclidean majorisation). *Given that a loss function $\mathcal{L} : \mathbb{R}^d \to \mathbb{R}$ is gradient-Lipschitz with constant λ, then the Euclidean majorisation (Inequality 4.10) holds.*

Proof. By the fundamental theorem of calculus, the Cauchy-Schwarz inequality and finally the gradient-Lipschitz property:

$$\mathcal{L}(w + \Delta w) - \left[\mathcal{L}(w) + \nabla_w \mathcal{L}(w)^\top \Delta w \right]$$
$$= \int_0^1 dt \ (\nabla_w \mathcal{L}(w + t \cdot \Delta w) - \nabla_w \mathcal{L}(w))^\top \Delta w$$
$$\leq \int_0^1 dt \ \|\nabla_w \mathcal{L}(w + t \cdot \Delta w) - \nabla_w \mathcal{L}(w)\|_2 \cdot \|\Delta w\|_2$$
$$\leq \int_0^1 dt \ t \cdot \lambda \cdot \|\Delta w\|_2^2 = \frac{\lambda}{2} \cdot \|\Delta w\|_2^2.$$

The proof is completed by adding the first-order Taylor series to both sides. $\square$

Finally, the following theorem shows that the gradient descent optimisation algorithm emerges via minimisation of the majorisation in Inequality 4.10.

Theorem 4.1 (Gradient descent). *The following holds:*

$$\underset{\Delta w}{\arg\min} \left[\mathcal{L}(w) + \nabla_w \mathcal{L}(w)^\top \Delta w + \frac{\lambda}{2} \cdot \|\Delta w\|_2^2 \right] = -\frac{1}{\lambda} \cdot \nabla_w \mathcal{L}(w). \qquad (4.12)$$

Proof. Take the derivative of the minimand on the left-hand side with respect to Δw, set this derivative to zero, and solve for Δw. □

Second example: Mirror descent

The previous subsection showed that gradient descent in its most basic form arises from an intrinsically Euclidean majorisation of the loss. This subsection shows that a certain kind of *non-Euclidean* majorisation leads to a variant of gradient descent known as *mirror descent* (Nemirovsky and Yudin, 1983).

The kind of non-Euclidean majorisation in question has a particular form: it assesses the error in the first-order Taylor expansion of a convex function:

Definition 4.3 (Bregman divergence). Given a convex function $\psi : \mathcal{W} \to \mathbb{R}$, the *Bregman divergence* $h_\psi : \mathcal{W} \times \mathcal{W} \to \mathbb{R}$ corresponding to ψ is given by:

$$h_\psi(w, w + \Delta w) := \psi(w + \Delta w) - \left[\psi(w) + \nabla\psi(w)^\top \Delta w \right]. \qquad (4.13)$$

Note that by a basic property of convexity, $\psi(w + \Delta w)$ always lies above the tangent to ψ at w, and therefore $h_\psi \geq 0$. Also, it is quick to check that h_ψ satisfies the conditions of Definition 4.1 to be a valid majorisation of ψ.

One hopes to apply mirror descent in situations where the trust region of the loss function of interest $\mathcal{L}$ is well-modeled by h_ψ:

Assumption 4.1 (Majorisation via Bregman divergence). For a loss function $\mathcal{L} : \mathcal{W} \to \mathbb{R}$ and a convex function $\psi : \mathcal{W} \to \mathbb{R}$, assume that:

$$\mathcal{L}(w + \Delta w) \leq \mathcal{L}(w) + \nabla_w \mathcal{L}(w)^\top \Delta w + h_\psi(w, w + \Delta w). \qquad (4.14)$$

In words: the majorisation h_ψ of convex function ψ is assumed to also majorise the loss function $\mathcal{L}$. The reason that this assumption is interesting is that the corresponding majorise-minimise algorithm has a particularly elegant form:

Proposition 4.1 (Mirror descent). *Let perturbation Δw_* denote the minimiser of the majorisation given in Assumption 4.1:*

$$\Delta w_* := \arg\min_{\Delta w} \left[\mathcal{L}(w) + \nabla_w \mathcal{L}(w)^\top \Delta w + h_\psi(w, w + \Delta w) \right]. \qquad (4.15)$$

Then Δw_* satisfies the following first-order optimality condition:

$$\nabla\psi(w + \Delta w_*) = \nabla\psi(w) - \nabla_w \mathcal{L}(w). \qquad (4.16)$$

Furthermore, when $\nabla\psi$ is invertible, the optimal perturbation satisfies:

$$w + \Delta w_* = \nabla\psi^{-1}\left[\nabla\psi(w) - \nabla_w\mathcal{L}(w)\right]. \tag{4.17}$$

Equation 4.16 may be interpreted as a vanilla gradient descent update to optimisation variables that have been transformed by the map $\nabla\psi : \mathbb{R}^d \to \mathbb{R}^d$. So for the kind of non-Euclidean majorisation given in Assumption 4.1, optimisation still admits a simple additive structure when viewed in the *mirror domain* $\nabla\psi(\mathcal{W}) := \{\nabla\psi(w) : w \in \mathcal{W}\}$.

Mirror descent makes an important departure from the Euclidean structure of gradient descent by leveraging a particular kind of non-Euclidean majorisation given by a Bregman divergence (Assumption 4.1). But there may be other means of constructing non-Euclidean majorisations. Non-Euclidean majorisations directly tailored to learning problems will be explored in Chapters 5 and 6.

Third example: Cubic regularised Newton

A first attempt toward building a second-order optimisation method is known as *Newton's method.* Newton's method attempts to minimise the Taylor series expansion of the loss function truncated to second order:

$$\Delta w^* = \underset{\Delta w}{\arg\min}\left[\mathcal{L}(w) + \nabla_w\mathcal{L}(w)^\top\Delta w + \frac{1}{2}\cdot\Delta w^\top\nabla_w^2\mathcal{L}(w)\Delta w\right]. \tag{4.18}$$

Unfortunately, as was the case for minimising the first-order truncation (Equation 4.6), this procedure is generally not well-founded. For instance, if the Hessian $\nabla_w^2\mathcal{L}(w)$ has any negative eigenvalues, then the minimand appearing in Equation 4.18 can be made arbitrarily negative by selecting a Δw in the corresponding negative eigenspace, with arbitrarily large magnitude.

Again, this issue may be solved by a majorisation. Just as Lemma 4.1 showed that a gradient-Lipschitz loss function admits a quadratic majorisation of the first-order Taylor expansion, a Hessian-Lipschitz loss function admits a cubic majorisation of the second-order expansion. This insight leads to the *cubic regularised* version of Newton's method due to Nesterov and Polyak (2006):

$$\Delta w_* = \underset{\Delta w}{\arg\min}\left[\mathcal{L}^{(2)}(w + \Delta w) + \frac{\lambda}{6}\cdot\|\Delta w\|_2^3\right]. \tag{4.19}$$

The authors show how to solve this optimisation sub-problem, and provide some global convergence results for this method in certain settings.

Optimiser	Truncation order, k	Majorisation, h
gradient descent	$k = 1$	$\frac{\lambda}{2} \cdot \|\Delta w\|_2^2$
mirror descent	$k = 1$	$h_\psi(w, w + \Delta w)$
cubic regularised Newton	$k = 2$	$\frac{\lambda}{6} \cdot \|\Delta w\|_2^3$

Table 4.2: Optimisation methods and their corresponding majorisations. Each optimiser perturbs a weight vector $w \mapsto w + \Delta w_*$ where the perturbation Δw_* is selected by solving $\Delta w_* = \arg\min_{\Delta w}[\mathcal{L}^{(k)}(w + \Delta w) + h(w, \Delta w)]$. In this expression, $\mathcal{L}^{(k)}(w + \Delta w)$ refers to the Taylor series expansion of $\mathcal{L}(w + \Delta w)$ in perturbation Δw truncated to kth order.

4.4 Trade-off between computation and fidelity

This chapter has presented derivations of various optimisation methods, sum-marised in Table 4.2. The derivations work by majorising a truncated Taylor expansion in weight perturbations, and then minimising this majorisation. This process is formally described by Equation 4.9 and illustrated in Figure 4.1.

What has not been discussed is the computational complexity of this procedure. In general, there is a trade-off between the fidelity of any particular majorisation, and the computational cost of its evaluation. For instance, majorising a higher-order Taylor expansion may lead to a tighter perturbation bound with a larger region of validity, allowing each optimisation step to make more progress. But one needs to ask if this larger per-step improvement is worth the extra computational overhead of evaluating a higher-order perturbation bound.

A good case in point is first-order versus second-order methods. While the per-step improvement of a second-order method may well exceed that of a first-order method, the per-step cost of the second-order method may be prohibitively expensive. On a d-dimensional weight space, $\mathcal{W} = \mathbb{R}^d$:

1) The Hessian matrix $\nabla_w^2 \mathcal{L}(w) \in \mathbb{R}^{d \times d}$ requires $\mathcal{O}(d^2)$ memory to store.

2) The gradient vector $\nabla_w \mathcal{L}(w) \in \mathbb{R}^d$ requires $\mathcal{O}(d)$ memory to store.

So first-order methods may be preferable on high-dimensional weight spaces.

References

Kenneth Lange (2016). *MM Optimization Algorithms*. Society for Industrial and Applied Mathematics.

Arkady S. Nemirovsky and David B. Yudin (1983). *Problem complexity and method efficiency in optimization*. Wiley.

Yurii Nesterov and Boris Polyak (2006). "Cubic regularization of Newton method and its global performance". In: *Mathematical Programming*.

Chapter 5

MAJORISE-MINIMISE FOR LEARNING PROBLEMS

> This chapter introduces two novel techniques for machine learning optimisation problems: *functional majorisation* of the loss function and *architectural perturbation bounds* for machine learning models. Together these techniques allow the majorise-minimise meta-algorithm to be applied to generic learning problems. This chapter uses the techniques to re-derive gradient descent and the Gauss-Newton method. Chapter 6 will use the techniques to derive architecture aware optimisation algorithms.

While Chapter 4 dealt with somewhat generic optimisation algorithms, the focus of this chapter is a framework for deriving optimisation algorithms for machine learning problems. A central feature of this type of optimisation problem is that the loss function $\mathcal{L}$ is affected by the weight vector w only indirectly through its appearance in the machine learning model $f(\cdot; w)$. This presents an opportunity: one can design optimisation algorithms that leverage the architecture of the machine learning model $f(\cdot; w)$ in interesting and useful ways. This idea will be termed *architecture aware optimisation.*

This chapter makes an important conceptual shift from considering a weight perturbation Δw directly, as in Chapter 4, to first studying the functional perturbation Δf that is induced by the weight perturbation Δw:

$$\Delta f(\cdot) \coloneqq f(\cdot; w + \Delta w) - f(\cdot; w). \tag{5.1}$$

After making this shift, the following three-step framework is proposed for deriving architecture aware optimisation algorithms:

Step 1: *Functional majorisation of the loss.* Expand the loss function as a series in functional perturbations, and majorise this expansion in terms of the size of functional perturbations. Lemma 5.3 gives an example.

Step 2: *Architectural perturbation bounds.* Derive bounds that relate the size of functional perturbations to the size of weight perturbations by analysing the model architecture. Lemma 6.2 gives an example.

Step 3: *Majorise–minimise.* Substitute the architectural perturbation bounds into the functional majorisation of the loss and minimise with respect to the weight perturbation to obtain an optimisation algorithm.

For the case of linear regression, this framework turns out to reproduce the classic gradient descent algorithm. Under the assumption that functional perturbations are linear in weight perturbations, the framework reproduces the classic *Gauss-Newton method*, which is closely related to *natural gradient descent*. The real payoff comes in Chapter 6, where the framework is applied to deep neural networks—yielding architecture aware optimisation methods.

5.1 Expanding the loss as a series in functional perturbations

This section derives a novel series expansion of machine learning loss functions in terms of functional perturbations. The key idea is to Taylor expand the loss function in functional perturbations, and then to transform the linear terms in this expansion back to weight space. This last step renders the expansion more suitable for deriving optimisation algorithms that operate in weight space.

This expansion relies on the fact that a machine learning loss function $\mathcal{L}$ can be regarded either as a function of a weight vector w or a projected function $f_X(w)$ (Definition 2.1). With this in mind, this chapter will sometimes abuse notation by insisting that $\mathcal{L}(w) \equiv \mathcal{L}(f_X(w))$. Concretely, for square loss:

$$\mathcal{L}_2(w) = \frac{1}{2m} \cdot \|f_X(w) - Y\|_2^2 \quad \Longrightarrow \quad \mathcal{L}_2(f_X) = \frac{1}{2m} \cdot \|f_X - Y\|_2^2.$$

To derive this expansion, it will help to define a notion of functional perturbation projected over a set of inputs:

Definition 5.1 (Projected functional perturbation)**.** Given a function $f : \mathcal{X} \times \mathcal{W} \to \mathbb{R}$ and a collection of m inputs $X = \{x_1, ..., x_m\}$, the *projected functional perturbation* $\Delta f_X \in \mathbb{R}^m$ corresponding to unperturbed weight vector w and weight perturbation Δw is given by the difference of projections:

$$\Delta f_X := f_X(w + \Delta w) - f_X(w). \tag{5.2}$$

The projected functional perturbation Δf_X implicitly depends on both a weight vector w and a weight perturbation Δw, but this dependence is suppressed for brevity. Given this definition, the loss function may be expanded as follows:

Lemma 5.1 (Series expansion in functional perturbations). *Given a function $f : \mathcal{X} \times \mathcal{W} \to \mathbb{R}$, a set of training inputs $X = \{x_1, ..., x_m\}$, and a loss $\mathcal{L}$ that is an analytic function of the function space projection f_X, the following holds:*

$$\mathcal{L}(w + \Delta w) - \left[\mathcal{L}(w) + \nabla_w \mathcal{L}(w)^\top \Delta w\right]$$
$$= \nabla_{f_X} \mathcal{L}(f_X)^\top \left[\Delta f_X - \nabla_w f_X(w) \Delta w\right] + \tfrac{1}{2} \Delta f_X^\top \nabla_{f_X}^2 \mathcal{L}(f_X) \Delta f_X + ... \quad (5.3)$$

Proof. First, Taylor expand the loss in functional perturbations Δf_X:

$$\mathcal{L}(f_X + \Delta f_X) = \mathcal{L}(f_X) + \nabla_{f_X} \mathcal{L}(f_X)^\top \Delta f_X + \tfrac{1}{2} \Delta f_X^\top \nabla_{f_X}^2 \mathcal{L}(f_X) \Delta f_X + ...$$

Next, make the substitutions $\mathcal{L}(f_X + \Delta f_X) \equiv \mathcal{L}(w + \Delta w)$ and $\mathcal{L}(f_X) \equiv \mathcal{L}(w)$:

$$\mathcal{L}(w + \Delta w) = \mathcal{L}(w) + \nabla_{f_X} \mathcal{L}(f_X)^\top \Delta f_X + \tfrac{1}{2} \Delta f_X^\top \nabla_{f_X}^2 \mathcal{L}(f_X) \Delta f_X + ...$$

Finally, subtract $\left[\mathcal{L}(w) + \nabla_w \mathcal{L}(w)^\top \Delta w\right]$ from both sides and apply the chain rule $\nabla_w \mathcal{L}(w)^\top \Delta w = \nabla_{f_X} \mathcal{L}(f_X)^\top \nabla_w f_X \Delta w$ on the right-hand side. $\quad\square$

The requirement of Lemma 5.1 that the loss function be analytic in the function space projection f_X is mild. Most common loss functions—including square loss (Definition 2.14) and logistic loss (Definition 2.15) satisfy this requirement—even when the loss is not an analytic function of the weight vector w.

5.2 Functional majorisation of square loss

This section specialises the series expansion in functional perturbations (Lemma 5.1) to the case of square loss. This is a particularly convenient example to consider, since the series expansion terminates at second order.

Lemma 5.2 (Expanding square loss in functional perturbations). *The square loss (Proposition 2.1) admits the following expansion:*

$$\mathcal{L}_2(w + \Delta w) - \left[\mathcal{L}_2(w) + \nabla_w \mathcal{L}_2(w)^\top \Delta w\right]$$
$$= \frac{1}{m}(f_X - Y)^\top \left[\Delta f_X - \nabla_w f_X \Delta w\right] + \frac{1}{2m}\|\Delta f_X\|_2^2. \quad (5.4)$$

Proof. For the square loss, $\nabla_{f_X} \mathcal{L}(f_X) = \frac{1}{m}(f_X - Y)$ and $\nabla_{f_X}^2 \mathcal{L}(f_X) = \frac{1}{m} \cdot \mathbf{I}$. All higher-order derivatives with respect to f_X are zero. Substituting these relations in to Lemma 5.1 yields the result. $\quad\square$

By an application of the Cauchy-Schwarz inequality, Lemma 5.2 leads directly to the following majorisation of the square loss in terms of functional perturbations:

Lemma 5.3 (Functional majorisation of square loss)**.** *The square loss (Proposition 2.1) admits the following majorisation:*

$$\mathcal{L}_2(w + \Delta w) - \left[\mathcal{L}_2(w) + \nabla_w \mathcal{L}_2(w)^\top \Delta w\right]$$
$$\leq \frac{1}{m} \cdot \|f_X - Y\|_2 \cdot \|\Delta f_X - \nabla_w f_X \Delta w\|_2 + \frac{1}{2m} \cdot \|\Delta f_X\|_2^2. \quad (5.5)$$

It is worth drawing attention to the three important quantities appearing on the right-hand side of Inequality 5.5:

1) $\|f_X - Y\|_2$ measures the size of the current misfit of the training sample.

2) $\|\Delta f_X - \nabla_w f_X \Delta w\|_2$ measures the degree to which the projected functional perturbation Δf_X deviates from its linearisation in Δw.

3) $\|\Delta f_X\|_2^2$ measures the size of the projected functional perturbation.

While the current data misfit is usually easy to compute or estimate in a machine learning problem, the latter two quantities are considerably more subtle. This thesis suggests relating these quantities back to weight perturbations via *architectural perturbation bounds.* In turn, this will open the door to architecture aware optimisation algorithms.

5.3   First application: Deriving gradient descent

As a first complete example of the framework proposed in this chapter, this section derives architectural perturbation bounds for linear regression. Combining these bounds with the majorisation of square loss (Lemma 5.3) leads back to the simple gradient descent optimisation algorithm. While in practice one might prefer to use less rudimentary means of fitting a linear regressor, the power of the argument outlined here is that it will generalise to deep networks.

A linear regressor is a function $f : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ of the form:

$$f(x; w) := w^\top x. \quad (5.6)$$

Suppose one wishes to fit a linear regressor to data $X = \{x_1, ..., x_m\}$ by running iterative minimisation of the square loss (Example 2.1). Further, suppose that the data is constrained to the unit hypersphere: $x_1, ..., x_m \in \mathbb{S}^{d-1}$. Then one may leverage the following architectural perturbation bounds:

Lemma 5.4 (Architectural perturbation bounds for linear regression)**.** *Given a set X of m training inputs supported on the hypersphere $\sqrt{d} \cdot \mathbb{S}^{d-1}$, the linear regressor of Equation 5.6 satisfies:*

$$\|\Delta f_X\|_2 \leq \sqrt{md} \cdot \|\Delta w\|_2. \tag{5.7}$$

$$\|\Delta f_X - \nabla_w f_X \Delta w\|_2 = 0; \tag{5.8}$$

Proof. By linearity of Equation 5.6, the projected functional perturbation is:

$$\Delta f_X = (\Delta w^\top x_1, ..., \Delta w^\top x_m).$$

Inequality 5.7 follows from an application of the Cauchy-Schwarz inequality:

$$\|\Delta f_X\|_2^2 = \sum_{i=1}^m (\Delta w^\top x_i)^2 \leq \sum_{i=1}^m \|\Delta w\|_2^2 \cdot \|x_i\|_2^2 = md \cdot \|\Delta w\|_2^2,$$

where the last equality follows since $x_i \in \sqrt{d} \cdot \mathbb{S}^{d-1}$. Finally, Equation 5.8 follows by observing that:

$$\Delta f_X - \nabla_w f_X \Delta w = (\Delta w^\top x_1, ..., \Delta w^\top x_m) - (x_1, ..., x_m) \cdot \Delta w = 0.$$

This completes the proof. $\qquad\qquad\square$

By combining these architectural perturbation bounds (Lemma 5.4) with the functional majorisation of the square loss (Lemma 5.3), one obtains:

Theorem 5.1 (Majorisation of the square loss for linear regression)**.**

$$\left| \mathcal{L}_2(w + \Delta w) - \left[\mathcal{L}_2(w) + \nabla_w \mathcal{L}_2(w)^\top \Delta w \right] \right| \leq \frac{d}{2} \cdot \|\Delta w\|_2^2. \tag{5.9}$$

Then, by Theorem 4.1, the optimisation algorithm that minimises this majorisation with respect to Δw is gradient descent with step-size $1/d$.

5.4 Second application: Deriving the Gauss-Newton method

This section shows that the classic *Gauss-Newton method* (Björck, 1996), which is closely related to *natural gradient descent* (Pascanu and Bengio, 2014; Amari, 1998), may be derived under the proposed framework in a straightforward manner. In particular, the Gauss-Newton method is the minimiser of the functional majorisation of square loss (Lemma 5.3) under the assumption that functional perturbations are linear in weight perturbations:

Assumption 5.1 (Functional perturbations are linear). The functional perturbation Δf_X corresponding to a weight perturbation Δw is given by:

$$\Delta f_X = \nabla_w f_X \Delta w. \tag{5.10}$$

In words: Assumption 5.1 amounts to approximating the functional perturbation Δf_X by its Taylor series in weight perturbations Δw truncated to first order. When combined with the following definition, this assumption leads to very simple architectural perturbation bounds:

Definition 5.2 (Squared Jacobian). Consider a machine learning model $f : \mathcal{X} \times \mathcal{W} \to \mathbb{R}$ and a set of m training inputs $X = \{x_1, ..., x_m\}$. The squared Jacobian F_X is given by:

$$F_X := \frac{1}{m} \cdot \nabla_w f_X^\top \nabla_w f_X. \tag{5.11}$$

To make Defintion 5.2 more explicit, for a d-dimensional weight space $\mathcal{W} = \mathbb{R}^d$, the squared Jacobian is the $d \times d$ matrix whose (ik)th entry is given by:

$$F_X^{ik} = \frac{1}{m} \cdot \sum_{j=1}^{m} \frac{\partial f(x_j; w)}{\partial w_i} \cdot \frac{\partial f(x_j; w)}{\partial w_k}. \tag{5.12}$$

In the literature on natural gradient descent, the matrix F_X is connected to the *Fisher information matrix* of information geometry (Amari, 2016).

With Definition 5.2 in hand, the following lemma is immediate:

Lemma 5.5 (Architectural perturbation bounds for linear functional perturbations). *Under Assumption 5.1, given a set of m inputs X, the following hold:*

$$\|\Delta f_X\|_2 = \sqrt{m} \cdot \sqrt{\Delta w^\top F_X \Delta w}; \tag{5.13}$$

$$\|\Delta f_X - \nabla_w f_X \Delta w\|_2 = 0. \tag{5.14}$$

Referring to the results in Lemma 5.5 as architectural perturbation *bounds* is technically a misnomer since these results are actually *equalities*. The thesis persists with this misnomer to emphasise the connection to Lemmas 5.4 and 6.2. These architectural perturbation bounds lead to the following majorisation:

Lemma 5.6 (Majorisation of square loss for linear functional perturbations). *Under Assumption 5.1, the square loss (Proposition 2.19) admits majorisation:*

$$\mathcal{L}_2(w + \Delta w) - \left[\mathcal{L}_2(w) + \nabla_w \mathcal{L}_2(w)^\top \Delta w \right] \leq \frac{1}{2} \Delta w^\top F_X \Delta w. \tag{5.15}$$

Proof. Substitute the results of Lemma 5.5 into Lemma 5.3. □

Finally, the Gauss-Newton method is obtained:

Theorem 5.2 (Gauss-Newton method). *Under Assumption 5.1, Lemma 5.6 produced a majorisation of square loss given by Inequality 5.15. The minimiser of this majorisation is as follows:*

$$\arg\min_{\Delta w} \left[\mathcal{L}_2(w) + \nabla_w \mathcal{L}_2(w)^\top \Delta w + \frac{1}{2} \cdot \Delta w^\top F_X \Delta w \right] = -F_X^{-1} \cdot \nabla_w \mathcal{L}_2(w).$$

(5.16)

Proof. Set to zero the derivative of the minimand on the left-hand side with respect to Δw: $\nabla_w \mathcal{L}_2(w) + F_X \Delta w = 0$. Solve for Δw to yield the result. □

In summary: this chapter developed a framework for deriving optimisation algorithms for generic learning problems. The framework involves minimising a functional majorisation of the loss. Architectural perturbation bounds are employed to relate functional perturbation to weight perturbations. The chapter concluded by demonstrating that both gradient descent and the Gauss-Newton method may be re-derived under this framework. The next chapter will apply this framework to deep neural networks.

References

Shun-ichi Amari (1998). "Natural gradient works efficiently in learning". In: *Neural Computation.*

– (2016). *Information Geometry and Its Applications.* Springer.

Åke Björck (1996). *Numerical Methods for Least Squares Problems.* Society for Industrial and Applied Mathematics.

Razvan Pascanu and Yoshua Bengio (2014). "Revisiting natural gradient for deep networks". In: *International Conference on Learning Representations.*

Chapter 6

MAJORISE-MINIMISE FOR DEEP NETWORKS

> This chapter derives architectural perturbation bounds for deep linear networks. When combined with the functional majorisation of a loss function, the bounds yield novel *architecture aware* optimisation methods.

Chapter 5 developed a framework for deriving optimisation algorithms for generic machine learning problems. In essence, the framework describes a majorise-minimise meta-algorithm (Lange, 2016) for composite optimisation problems that apply an error measure to a function projected on to data.

The present chapter specialises this framework to machine learning problems that involve deep neural networks. Again, the framework works in three steps:

Step 1: Majorise a series expansion of the loss function in perturbations to the network output projected on to the training set.

Step 2: Derive architectural perturbation bounds that express the sensitivity of the network output to weight perturbations. The form of these bounds depends on details such as the width and depth of the network.

Step 3: Substitute the architectural perturbation bounds into the majorisation of the loss and minimise to obtain an optimisation algorithm.

A note on related work

Cohen et al. (2021) express concern that majorise-minimise style optimisation theories (or ones based on Majorisation 4.10 in particular) may be too pessimistic for deep learning. On the contrary, this chapter obtains various useful architectural scaling rules for learning rate from a majorise-minimise analysis.

While the depth scaling relation presented in this chapter is an original contribution of the thesis author and collaborators (Bernstein et al., 2020), the width scaling relation was first derived separately via the *tensor programs* framework (Yang et al., 2021). That framework amounts to a perturbation analysis of neural networks with random weights operating in the asymptotic limit of

infinite width. The width scaling relation in this chapter resulted from discussions with Greg Yang about how to reconcile the tensor programs framework with the non-asymptotic, non-random framework presented in this thesis. The experiments in Figures 6.1 and 6.2 were run as part of that collaboration.

6.1 The deep linear network

This chapter deals mainly with deep networks with identity nonlinearity $\varphi = \mathbf{I}$.

Definition 6.1 (Deep linear network). A *deep linear network f* of depth L maps an input $x \in \mathbb{R}^{d_0}$ to an output $f(x; w) \in \mathbb{R}^{d_L}$ via L matrix multiplications:

$$f(x; w) := W_L W_{L-1} \ldots W_1 \cdot x. \tag{6.1}$$

In this expression, w denotes the tuple $w = (W_1, ..., W_L)$ for $W_l \in \mathbb{R}^{d_l \times d_{l-1}}$.

This choice of *non-nonlinearity* greatly simplifies the analysis. Although architectural perturbation bounds have been obtained for more general nonlinearities (Bernstein et al., 2020), the results obtained for deep linear networks in this chapter were already found to transfer experimentally to deep relu networks (Figure 6.1). It is important to note that the optimisation landscape of the deep linear network is still non-convex due to the product of weight matrices.

The following definition will aid the analysis of deep linear networks:

Definition 6.2 (Output scale of the deep linear network). The *output scale* of a deep linear network f with weight matrices $w = (W_1, ..., W_L)$ is given by:

$$F(w) := \sqrt{d_0} \cdot \prod_{l=1}^{L} \|W_l\|_*. \tag{6.2}$$

The output scale provides a simple bound on the magnitude of network outputs:

Lemma 6.1 (Output bound). *For a deep linear network $f(\cdot, w)$ and all hyperspherically constrained inputs $x \in \sqrt{d_0} \cdot \mathbb{S}^{d_0-1}$, the output magnitude obeys:*

$$\|f(x; w)\|_2 \leq F(w). \tag{6.3}$$

Proof. Recursively extract operator norms from the matrix product:

$$\|f(x; w)\|_2 = \|W_L W_{L-1} \ldots W_1 \cdot x\|_2$$

$$\leq \|W_L\|_* \cdot \|W_{L-1} \ldots W_1 \cdot x\|_2 \leq \ldots \leq \prod_{l=1}^{L} \|W_l\|_* \cdot \|x\|_2.$$

Finally, substituting $\|x\|_2 = \sqrt{d_0}$ completes the proof. $\square$

6.2 Architectural perturbation bounds for deep linear networks

The deep linear network admits the following architectural perturbation bounds:

Lemma 6.2 (Architectural perturbation bounds for the deep linear network)**.** *Consider perturbing the weights of a deep linear network $f : \mathbb{R}^{d_0} \times \mathcal{W} \to \mathbb{R}$ from $w = (W_1, ..., W_L) \in \mathcal{W}$ to $w + \Delta w = (W_1 + \Delta W_1, ..., W_L + \Delta W_L) \in \mathcal{W}$. For any collection of m inputs $X \in (\sqrt{d_0} \cdot \mathbb{S}^{d_0 - 1})^m$, the following bounds hold:*

$$\frac{\|\Delta f_X\|_2}{\sqrt{m}} \le F(w) \cdot \left[\prod_{l=1}^{L} \left(1 + \frac{\|\Delta W_l\|_*}{\|W_l\|_*} \right) - 1 \right]; \tag{6.4}$$

$$\frac{\|\Delta f_X - \nabla_w f_X \Delta w\|_2}{\sqrt{m}} \le F(w) \cdot \left[\prod_{l=1}^{L} \left(1 + \frac{\|\Delta W_l\|_*}{\|W_l\|_*} \right) - 1 - \sum_{l=1}^{L} \frac{\|\Delta W_l\|_*}{\|W_l\|_*} \right]. \tag{6.5}$$

These architectural perturbation bounds involve a product over the relative size of the perturbation to each network layer, reflecting the product structure of the network itself. At large depth, the bounds are roughly exponential in the layerwise perturbation size.

While the best way to understand the results is by working out a few examples (for two, three and four layer networks, say) a formal proof is now given.

Proof of Lemma 6.2. The result is shown by induction over depth L for a network with multiple outputs $d_L > 1$ and a single input $X = \{x\}$. The result for multiple inputs $m > 1$ and a single output $d_L = 1$ is then immediate.

For the base case $L = 1$, the relevant network to consider is given by $f(x; w) = W_1 \cdot x$. Observe that $\nabla_w f_{\{x\}} \Delta w = \Delta W_1 \cdot x$, and also:

$$\Delta f_{\{x\}} := f(x; w + \Delta w) - f(x; w) = (W_1 + \Delta W_1) \cdot x - W_1 \cdot x = \Delta W_1 \cdot x.$$

The base case is established by noting $\|\Delta f_{\{x\}} - \nabla_w f_{\{x\}} \Delta w\|_2 = 0$, and also:

$$\|\Delta f_{\{x\}}\|_2 \le \|\Delta W_1\|_* \cdot \|x\|_2 = \|\Delta W_1\|_* \cdot \sqrt{d_0}.$$

For the inductive step, the relevant network is given by $f(x; w) = W_L ... W_1 \cdot x$.

To tackle Inequality 6.4, observe that:

$$\|\Delta f_{\{x\}}\|_2 := \|f(x; w + \Delta w) - f(x; w)\|_2$$
$$= \|(W_L + \Delta W_L)...(W_1 + \Delta W_1) \cdot x - W_L...W_1 \cdot x\|_2$$
$$= \|(W_L + \Delta W_l) \cdot [(W_{L-1} + \Delta W_{L-1})...(W_1 + \Delta W_1) \cdot x - W_{L-1}...W_1 \cdot x]$$
$$+ \Delta W_L W_{L-1}...W_1 \cdot x\|_2$$
$$\leq (\|W_L\|_* + \|\Delta W_L\|_*) \cdot \|(W_{L-1} + \Delta W_{L-1})...(W_1 + \Delta W_1)x - W_{L-1}...W_1 x\|_2$$
$$+ \|\Delta W_L\|_* \cdot \|W_{L-1}\|_* \cdot ... \cdot \|W_1\|_* \cdot \sqrt{d_0},$$

where the last line follows by several applications of the triangle inequality and the operator norm bound. Then, by the inductive hypothesis:

$$\|\Delta f_{\{x\}}\|_2 \leq (\|W_L\|_* + \|\Delta W_L\|_*) \cdot \frac{F(w)}{\|W_L\|_*} \cdot \left[\prod_{l=1}^{L-1}\left(1 + \frac{\|\Delta W_l\|_*}{\|W_l\|_*}\right) - 1\right]$$
$$+ \|\Delta W_L\|_* \cdot \|W_{L-1}\|_* \cdot ... \cdot \|W_1\|_* \cdot \sqrt{d_0}$$
$$= \left(1 + \frac{\|\Delta W_L\|_*}{\|W_L\|_*}\right) \cdot F(w) \cdot \left[\prod_{l=1}^{L-1}\left(1 + \frac{\|\Delta W_l\|_*}{\|W_l\|_*}\right) - 1\right]$$
$$+ \frac{\|\Delta W_L\|_*}{\|W_L\|_*} \cdot F(w)$$
$$= F(w) \cdot \left[\prod_{l=1}^{L}\left(1 + \frac{\|\Delta W_l\|_*}{\|W_l\|_*}\right) - 1\right],$$

which establishes Inequality 6.4. Next, to tackle Inequality 6.5, observe that:

$$\|\Delta f_{\{x\}} - \nabla_w f_{\{x\}} \Delta w\|_2 := \|f(x; w + \Delta w) - f(x; w) - \nabla_w f_{\{x\}} \Delta w\|_2$$
$$= \Big\|(W_L + \Delta W_L)...(W_1 + \Delta W_1) \cdot x - W_L...W_1 \cdot x$$
$$- \sum_{l=1}^{L} W_L...W_{l+1}\Delta W_l W_{l-1}...W_1 \cdot x\Big\|_2$$
$$= \Big\|(W_L + \Delta W_L) \cdot \Big[(W_{L-1} + \Delta W_{L-1})...(W_1 + \Delta W_1) \cdot x - W_{L-1}...W_1 \cdot x$$
$$- \sum_{l=1}^{L-1} W_{L-1}...W_{l+1}\Delta W_l W_{l-1}...W_1 \cdot x\Big]$$
$$+ \Delta W_L \sum_{l=1}^{L-1} W_{L-1}...W_{l+1}\Delta W_l W_{l-1}...W_1 \cdot x\Big\|_2$$
$$\leq (\|W_L\|_* + \|\Delta W_L\|_*) \cdot \Big\|(W_{L-1} + \Delta W_{L-1})...(W_1 + \Delta W_1)x - W_{L-1}...W_1 x$$
$$- \sum_{l=1}^{L-1} W_{L-1}...W_{l+1}\Delta W_l W_{l-1}...W_1 \cdot x\Big\|_2$$
$$+ \|\Delta W_L\|_* \cdot \sum_{l=1}^{L-1} \|W_{L-1}\|_* ... \|W_{l+1}\|_* \|\Delta W_l\|_* \|W_{l-1}\|_* ... \|W_1\|_* \cdot \sqrt{d_0},$$

where the last line follows by several applications of the triangle inequality and the operator norm bound. Then, by the inductive hypothesis:

$$\|\Delta f_{\{x\}} - \nabla_w f_{\{x\}} \Delta w\|_2$$

$$\leq (\|W_L\|_* + \|\Delta W_L\|_*) \cdot \frac{F(w)}{\|W_L\|_*} \cdot \left[\prod_{l=1}^{L-1} \left(1 + \frac{\|\Delta W_l\|_*}{\|W_l\|_*} \right) - 1 - \sum_{l=1}^{L-1} \frac{\|\Delta W_l\|_*}{\|W_l\|_*} \right]$$

$$\quad + \|\Delta W_L\|_* \cdot \frac{F(w)}{\|W_L\|_*} \cdot \sum_{l=1}^{L-1} \frac{\|\Delta W_l\|_*}{\|W_l\|_*}$$

$$= \left(1 + \frac{\|\Delta W_L\|_*}{\|W_L\|_*} \right) \cdot F(w) \cdot \left[\prod_{l=1}^{L-1} \left(1 + \frac{\|\Delta W_l\|_*}{\|W_l\|_*} \right) - 1 - \sum_{l=1}^{L-1} \frac{\|\Delta W_l\|_*}{\|W_l\|_*} \right]$$

$$\quad + \frac{\|\Delta W_L\|_*}{\|W_L\|_*} \cdot F(w) \cdot \sum_{l=1}^{L-1} \frac{\|\Delta W_l\|_*}{\|W_l\|_*}$$

$$= F(w) \cdot \left[\prod_{l=1}^{L} \left(1 + \frac{\|\Delta W_l\|_*}{\|W_l\|_*} \right) - 1 - \sum_{l=1}^{L} \frac{\|\Delta W_l\|_*}{\|W_l\|_*} \right],$$

which establishes Inequality 6.5 and completes the proof. □

6.3   Majorise-minimise for deep linear networks

This section converts the architectural perturbation bounds of Lemma 6.2 into an optimisation algorithm. The algorithm is *architecture aware* in the sense that it automatically accounts for details of the network architecture such as the scale of the weights, the number of layers and the width of each layer.

Solving the full majorise-minimise problem obtained via the architectural perturbation bounds of Lemma 6.2 is challenging, due to the many degrees of freedom in how perturbation strengths could be assigned to different layers. To simplify matters, a restricted solution is presented under the following ansatz:

Ansatz 6.1 (Equal layerwise updates). For some $\eta > 0$ that is independent of layer, the perturbation ΔW_l to the weight matrix W_l at layer l is given by:

$$\Delta W_l = -\eta \cdot \frac{1}{L} \cdot \|W_l\|_* \cdot \frac{\nabla_{W_l} \mathcal{L}(w)}{\|\nabla_{W_l} \mathcal{L}(w)\|_*}. \tag{6.6}$$

The content of this ansatz is that across all layers $l = 1, ..., L$, the perturbation ΔW_l is aligned with the negative gradient and has relative magnitude $\|\Delta W_l\|_* / \|W_l\|_* = \eta/L$ independent of layer. The factor of $1/L$ is only included for later convenience—it could just as well be folded into the factor of η. Under Ansatz 6.1, the majorise-minimise problem is reduced to solving for a single variable $\eta > 0$. The architectural perturbation bounds simplify as follows:

Lemma 6.3 (Architectural perturbation bounds under equal layerwise updates).
*Consider perturbing the weights of a deep linear network $f : \mathbb{R}^{d_0} \times \mathcal{W} \to \mathbb{R}$
from $w = (W_1, ..., W_L) \in \mathcal{W}$ to $w + \Delta w = (W_1 + \Delta W_1, ..., W_L + \Delta W_L) \in \mathcal{W}$.
For any collection of m inputs $X \in (\sqrt{d_0} \cdot \mathbb{S}^{d_0-1})^m$, under Ansatz 6.1:*

$$\|\Delta f_X\|_2 \le \sqrt{m} \cdot F(w) \cdot [\exp \eta - 1]; \tag{6.7}$$

$$\|\Delta f_X - \nabla_w f_X \Delta w\|_2 \le \sqrt{m} \cdot F(w) \cdot [\exp \eta - \eta - 1]. \tag{6.8}$$

Proof. Under Ansatz 6.1:

$$\prod_{l=1}^{L} \left(1 + \frac{\|\Delta W_l\|_*}{\|W_l\|_*}\right) = \left(1 + \frac{\eta}{L}\right)^L \le \lim_{L \to \infty} \left(1 + \frac{\eta}{L}\right)^L = \exp \eta,$$

$$\sum_{l=1}^{L} \frac{\|\Delta W_l\|_*}{\|W_l\|_*} = L \cdot \frac{\eta}{L} = \eta.$$

Substituting these relations into Lemma 6.2 yields the results. $\qquad\square$

These architectural perturbation bounds may be combined with the functional
majorisation of square loss (Lemma 5.3) to obtain:

Lemma 6.4 (Majorisation of square loss under equal layerwise updates). *Under
Ansatz 6.1, the square loss of a deep linear network with m training inputs
$X \in (\sqrt{d_0} \cdot \mathbb{S}^{d_0-1})^m$ and corresponding label vector $Y \in \mathbb{R}^m$ satisfies:*

$$\mathcal{L}_2(w + \Delta w) - \left[\mathcal{L}_2(w) + \nabla_w \mathcal{L}_2(w)^\top \Delta w\right]$$
$$\le \frac{1}{2} \cdot F(w) \cdot \left(F(w) + \frac{\|Y\|_2}{\sqrt{m}}\right) \cdot [\exp(2\eta) - 2\eta - 1]. \tag{6.9}$$

Proof. Substituting the architectural perturbation bounds from Lemma 6.3
into Lemma 5.3 yields:

$$\mathcal{L}_2(w + \Delta w) - \left[\mathcal{L}_2(w) + \nabla_w \mathcal{L}_2(w)^\top \Delta w\right]$$
$$\le \frac{\|f_X - Y\|_2}{\sqrt{m}} \cdot F(w) \cdot [\exp \eta - \eta - 1] + \frac{1}{2} \cdot F(w)^2 \cdot [\exp \eta - 1]^2.$$

To simplify this expression, one can observe that:

$$\|f_X - Y\|_2 \le \|f_X\|_2 + \|Y\|_2 \le \sqrt{m} \cdot F(w) + \|Y\|_2,$$

where the last inequality follows from Lemma 6.1. When combined with the
relaxation that $F(W)^2 \le F(w) \cdot (F(w) + \|Y\|_2/\sqrt{m})$, the result is obtained. $\quad\square$

With the majorisation of Lemma 6.4 in hand, the majorise-minimise principle may be applied as follows:

Theorem 6.1 (Log learning rates). *Lemma 6.3's majorisation of square loss for deep linear networks under Ansatz 6.1 is minimised by setting η to:*

$$\eta_\star := \frac{1}{2} \log \left(1 + \frac{1}{F(w) \left(F(w) + \frac{\|Y\|_2}{\sqrt{m}} \right)} \cdot \frac{1}{L} \sum_{l=1}^{L} \|W_l\|_* \frac{\|\nabla_{W_l} \mathcal{L}_2(w)\|_F^2}{\|\nabla_{W_l} \mathcal{L}_2(w)\|_*} \right). \quad (6.10)$$

Proof. Under Ansatz 6.1, the first-order Taylor expansion of square loss is:

$$\mathcal{L}_2^{(1)}(w + \Delta w) := \mathcal{L}_2(w) + \sum_{l=1}^{L} \nabla_{W_l} \mathcal{L}_2(w)^\top \Delta W_l$$

$$= \mathcal{L}_2(w) - \frac{\eta}{L} \sum_{l=1}^{L} \|W_l\|_* \cdot \frac{\|\nabla_{W_l} \mathcal{L}_2(w)\|_F^2}{\|\nabla_{W_l} \mathcal{L}_2(w)\|_*}.$$

Substituting this form of the first-order Taylor expansion into the majorisation of Lemma 6.4 implies that $\mathcal{L}_2(w + \Delta w)$ is upper bounded by:

$$\mathcal{L}_2(w) - \frac{\eta}{L} \sum_{l=1}^{L} \|W_l\|_* \frac{\|\nabla_{W_l} \mathcal{L}_2(w)\|_F^2}{\|\nabla_{W_l} \mathcal{L}_2(w)\|_*} + \frac{F(w)}{2} \left(F(w) + \frac{\|Y\|_2}{\sqrt{m}} \right) [\exp(2\eta) - 2\eta - 1].$$

Setting the derivative of this expression with respect to η to zero yields:

$$\frac{1}{L} \sum_{l=1}^{L} \|W_l\|_* \frac{\|\nabla_{W_l} \mathcal{L}_2(w)\|_F^2}{\|\nabla_{W_l} \mathcal{L}_2(w)\|_*} = F(w) \left(F(w) + \frac{\|Y\|_2}{\sqrt{m}} \right) [\exp(2\eta) - 1].$$

Finally, solving for η yields the result. $\qquad \square$

Theorem 6.1 was derived in close collaboration with Kevin Huang. In short, the theorem suggests a learning rule where layer l is perturbed via:

$$W_l \mapsto W_l - \eta_\star \cdot \frac{1}{L} \cdot \|W_l\|_* \cdot \frac{\nabla_{W_l} \mathcal{L}(w)}{\|\nabla_{W_l} \mathcal{L}(w)\|_*}, \quad (6.11)$$

with $\eta_\star$ given by Equation 6.10. A curious aspect of this update is that the scale of the gradient only enters logarithmically through the $\eta_\star$ term. This may explain why popular neural net optimisers, such as *Adam* (Kingma and Ba, 2015), more-or-less completely remove the gradient scale from their update.

Another feature of Update 6.11 is that explicit dependence on both the network depth L and the scale of the weight matrices $\|W_l\|_\star$ are encoded. But, as of yet, there is no explicit dependence on the network width. This omission is rectified in the next subsection.

Width scaling

Bernstein et al. (2020) assumed—without real evidence—that the weight matrices and gradients of a deep network have roughly the same conditioning. In turn, this meant that the architecture aware optimisation method developed in their paper does not scale properly with network width. A better conditioning assumption was employed in a paper by Yang et al. (2021):

Assumption 6.1 (Weight matrix and gradient conditioning). For all $l = 1, ..., L$:

$$\|W_l\|_* = \frac{\|W_l\|_F}{\sqrt{\min(d_l, d_{l-1})}};\tag{6.12}$$

$$\|\nabla_{W_l}\mathcal{L}(w)\|_* = \|\nabla_{W_l}\mathcal{L}(w)\|_F.\tag{6.13}$$

To understand this assumption, one needs to be familiar with the following aspect of matrix conditioning. Given a matrix $A \in \mathbb{R}^{d_l \times d_{l-1}}$ with $\overline{d} := \min(d_l, d_{l-1})$ singular values denoted $\sigma_1, ..., \sigma_{\overline{d}}$, the norms of A satisfy:

$$\|A\|_F^2 = \sum_{i=1}^{\overline{d}} \sigma_i^2 \geq \max_{i \in \{1, ..., \overline{d}\}} \sigma_i^2 = \|A\|_*^2.\tag{6.14}$$

This means that $\|A\|_F / \sqrt{\min(d_l, d_{l-1})}$ reports the root-mean-square singular value of A, while the operator norm $\|A\|_*$ reports the largest singular value.

Under this interpretation of matrix norms, Equation 6.12 is stating that the largest singular value of weight matrix W_l is equal to the average singular value—meaning that W_l is *well-conditioned*. The justification for this assumption is that the weight matrices in a deep network are typically initialised randomly, and random matrices are fairly well-conditioned.

On the other hand, Equation 6.13 states that the operator norm and Frobenius norm of the gradient $\nabla_{W_l}\mathcal{L}(w)$ at layer l are equal. By Equation 6.14, this happens when $\nabla_{W_l}\mathcal{L}(w)$ has only one non-zero singular value—meaning that the gradient is very *low rank*. The justification for this assumption is that the gradient is, in a sense, an *optimal* object: it reports the perturbation direction that elicits the largest change in loss. This makes it reasonable that the gradient would not spread itself too thin in the sense of rank. This intuition is simple to prove for the gradient of the loss over a single training input x, which may be written directly as the rank-one outer product $\nabla_{W_l}\mathcal{L} = \nabla_{f_l(x)}\mathcal{L} \otimes \varphi(f_{l-1}(x))$ in the notation of Equation 3.6. Rigorously extending this argument to larger numbers of training examples appears challenging.

Combining Assumption 6.1 with Theorem 6.1 leads to the following update:

Theorem 6.2 (Architecture aware deep network update)**.** *Lemma 6.3's majorisation of square loss for deep linear networks under Ansatz 6.1 and Assumption 6.1 is minimised by the following update:*

$$W_l \mapsto W_l - \eta_\dagger \cdot \frac{1}{L} \cdot \frac{\|W_l\|_F}{\sqrt{\min(d_l, d_{l-1})}} \cdot \frac{\nabla_{W_l}\mathcal{L}(w)}{\|\nabla_{W_l}\mathcal{L}(w)\|_F}, \qquad (6.15)$$

where the learning rate $\eta_\dagger$ is given by:

$$\eta_\dagger := \frac{1}{2}\log\left(1 + \frac{\frac{1}{L}\sum_{l=1}^L \frac{\|W_l\|_F}{\sqrt{\min(d_l, d_{l-1})}} \cdot \|\nabla_{W_l}\mathcal{L}_2(w)\|_F}{F(w)\left(F(w) + \frac{\|Y\|_2}{\sqrt{m}}\right)}\right). \qquad (6.16)$$

Proof. The result follows by substituting Assumption 6.1 into Theorem 6.1. $\square$

Update 6.15 explicitly depends on the width, depth and weight scale of the deep network. For this reason, Theorem 6.2 is tagged *architecture aware*. The theorem unifies various heuristic and theoretical ideas explored in the literature:

1) *Relative updates.* The gradient is rescaled by $\|W_l\|_F/\|\nabla_{W_l}\mathcal{L}(w)\|_F$. This means that the magnitude of the update is in proportion to the magnitude of the weight matrix to which it is applied. Such a scaling was proposed on heuristic grounds by You et al. (2017) and explored theoretically by Bernstein et al. (2020). It also relates to ideas explored by Carbonnelle and Vleeschouwer (2019) and Liu et al. (2021).

2) *Depth scaling.* Scaling the perturbation strength like $1/L$ for networks of depth L was proposed on theoretical grounds by Bernstein et al. (2020).

3) *Width scaling.* Scaling the perturbation size by $\|W_l\|_F/\sqrt{\min(d_l, d_{l-1})}$ relates to a theoretical technique proposed by Yang et al. (2021).

4) *Adaptive gradient clipping.* The logarithmic dependence of the update on the gradient scale relates to a heuristic technique known as *adaptive gradient clipping* (Brock et al., 2021) which clips the gradient once its magnitude surpasses a certain threshold.

6.4   Experimental tests with relu networks

The performance of Update 6.15 was tested for training multilayer perceptrons (Definition 2.9) with relu nonlinearity and of varying width and depth. The performance of the update was measured as a function of $\eta_\dagger$, where $\eta_\dagger$ was held constant during training. Testing the logarithmic form of $\eta_\dagger$ (Equation 6.16) is part of ongoing research with Kevin Huang, Chris Mingard and Yisong Yue.

The networks were trained on the CIFAR-10 dataset (Krizhevsky, 2009). CIFAR-10 consists of sixty thousand 32px by 32px RGB input images that each fall into one of ten classes. This means that each image is described by $32 \times 32 \times 3 = 3072 =: d_0$ real numbers along with a class index in $\{1, ..., 10\}$. The input images were pre-processed as follows: each image was flattened to lie in $\mathbb{R}^{d_0}$, centred to have mean zero and then projected on to the hypersphere of radius $\sqrt{d_0}$. The nonlinearity φ was set to $\varphi(\cdot) = \sqrt{2} \cdot \max(0, \cdot)$. The weight matrices were initialised iid Gaussian and scaled such that the root mean square singular value at each layer was approximately one. The loss function was set to measure the square loss between the 10-dimensional network output and a one-hot encoding of the class index. The networks were trained for 19 epochs with 1000 train images used to compute the gradient at each step.

The results are presented in Figure 6.1. As can be seen in that figure, the train loss as a function of learning rate $\eta_\dagger$ is quite stable as both the width and depth of the network are varied. For comparison, Figure 6.2 displays the behaviour without explicit depth scaling, by plotting train loss as a function of $\eta_\dagger/L$. As can be seen, this causes performance to drift with depth.
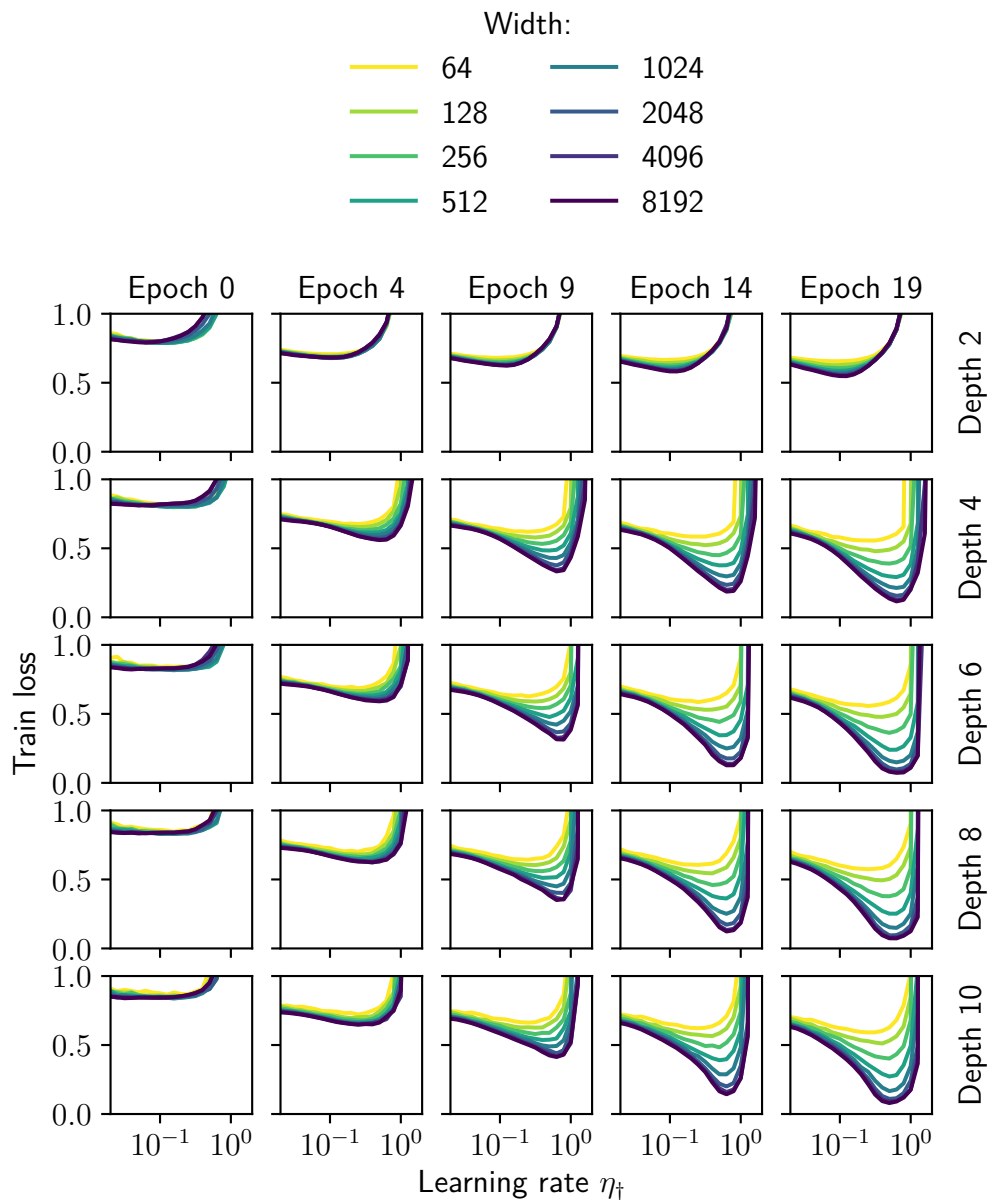
Figure 6.1: Learning rate transfer across width and depth. Update 6.15 was used to train relu multilayer perceptrons of varying width and depth on the CIFAR-10 dataset. As can be seen, training performance was quite stable as a function of learning rate $\eta_\dagger$, as both width and depth were varied.
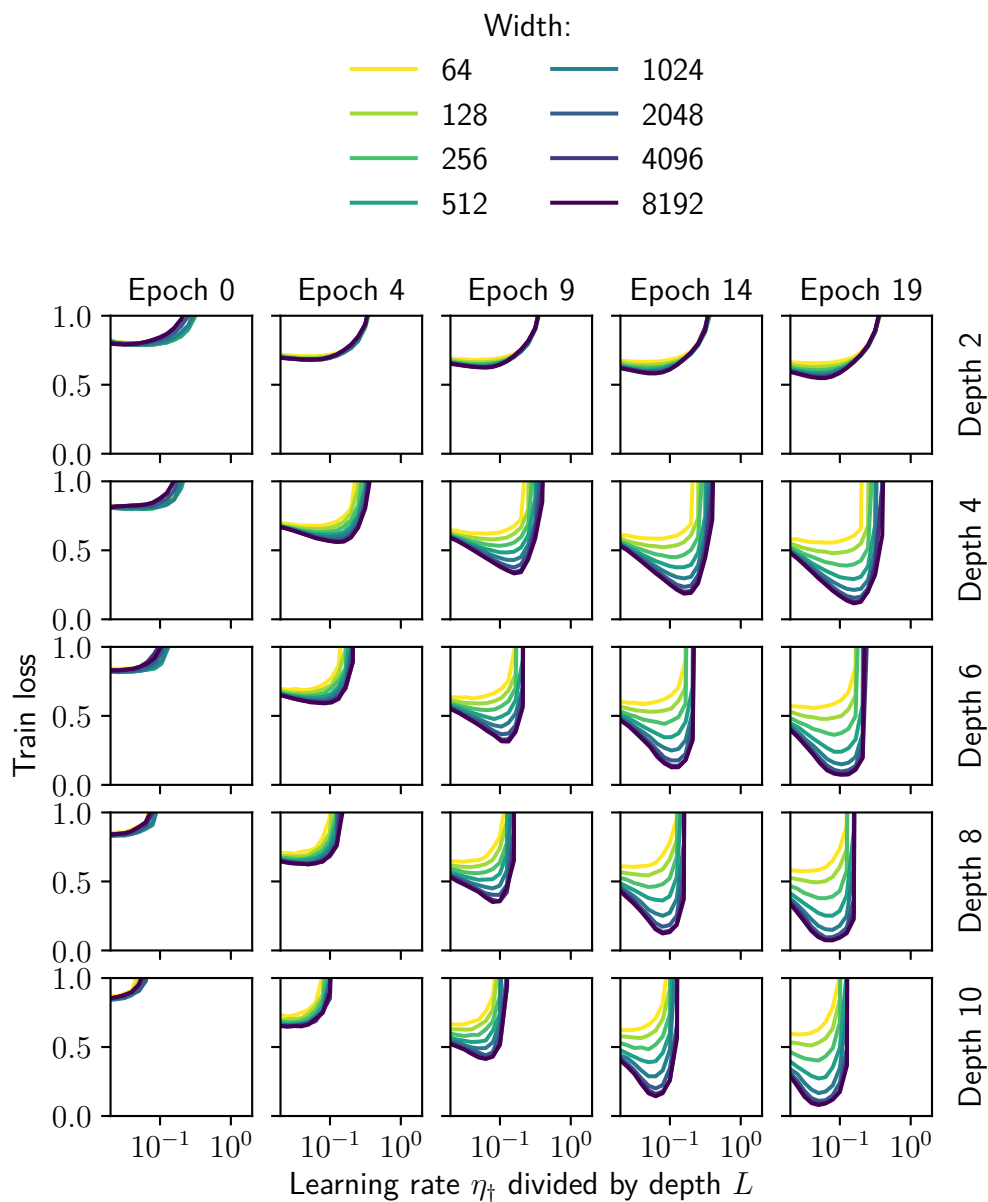
Figure 6.2: Learning rate transfer without explicit depth scaling. The same results as Figure 6.1 are plotted, except as a function of $\eta_\dagger/L$. This displays the behaviour of Update 6.15 with the explicit depth scaling removed. As can be seen, the tuning curves shift left for increasing depth.

References

Jeremy Bernstein, Arash Vahdat, Yisong Yue, and Ming-Yu Liu (2020). "On the distance between two neural networks and the stability of learning". In: *Neural Information Processing Systems*.

Andy Brock, Soham De, Samuel L. Smith, and Karen Simonyan (2021). "High-performance large-scale image recognition without normalization". In: *International Conference on Machine Learning*.

Simon Carbonnelle and Christophe De Vleeschouwer (2019). "Layer rotation: A surprisingly simple indicator of generalization in deep networks?" In: *ICML Workshop on Identifying and Understanding Deep Learning Phenomena*.

Jeremy Cohen, Simran Kaur, Yuanzhi Li, J. Zico Kolter, and Ameet Talwalkar (2021). "Gradient descent on neural networks typically occurs at the edge of stability". In: *International Conference on Learning Representations*.

Diederik P. Kingma and Jimmy Ba (2015). "Adam: A method for stochastic optimization". In: *International Conference on Learning Representations*.

Alex Krizhevsky (2009). *Learning multiple layers of features from tiny images*. Technical report. University of Toronto.

Kenneth Lange (2016). *MM Optimization Algorithms*. Society for Industrial and Applied Mathematics.

Yang Liu, Jeremy Bernstein, Markus Meister, and Yisong Yue (2021). "Learning by turning: Neural architecture aware optimisation". In: *International Conference on Machine Learning*.

Greg Yang, Edward J. Hu, Igor Babuschkin, Szymon Sidor, Xiaodong Liu, David Farhi, Nick Ryder, Jakub Pachocki, Weizhu Chen, and Jianfeng Gao (2021). "Tuning large neural networks via zero-shot hyperparameter transfer". In: *Neural Information Processing Systems*.

Yang You, Igor Gitman, and Boris Ginsburg (2017). *Scaling SGD batch size to 32K for ImageNet training*. Technical report. University of California, Berkeley.

Part III

GENERALISATION
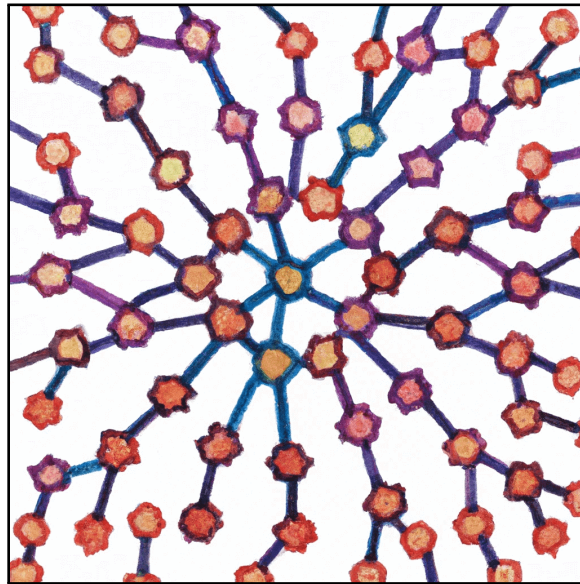
"[The brain's] underlying physical storage devices are capable of a thousand to a million times the capacity manifest in learned behavior... Possibly we should not be looking for models and mechanisms that produce storage economies, but rather ones in which marvels are produced by profligate use of capacity."

Thomas K. Landauer, 1986

Chapter 7

CLASSIC GENERALISATION THEORIES

> This chapter provides an introduction to three classic generalisation theories: uniform convergence, algorithmic stability and PAC-Bayes. The material is expositional and is included for the reader's aid.

The goal of this chapter is to review three classic theoretical frameworks for reasoning about generalisation of machine learning algorithms. The intention is not to provide a comprehensive survey, but more to provide an introduction to the landscape of thought surrounding generalisation. For each framework, the chapter discusses its general relevance to neural networks, including key hurdles that may need to be overcome to make the approach workable.

The three frameworks that are considered are:

1) *Uniform convergence.* This framework seeks to assess the amount of training data needed such that train error converges to test error for all functions within a function space. To achieve this, the techniques tend to rely on limiting the size of the function space in some way.

2) *Algorithmic stability.* This framework makes an important departure from uniform convergence by seeking to characterise generalisation of only those functions returned by a learning algorithm. In particular, the framework uses the insensitivity of the learning algorithm to modifications of its training set in order to guarantee generalisation.

3) *PAC-Bayes.* This framework also departs from uniform convergence by assessing generalisation only of a user-specified ensemble of functions within the function space. What's more, the guarantee is on the average generalisation error over the ensemble, meaning that a small fraction of ensemble members are allowed to generalise poorly. The guarantee is good when the *posterior measure* that defines the ensemble is sufficiently close to a *prior measure* chosen before seeing the training data.

7.1 Uniform convergence

Uniform convergence is a classic framework for studying generalisation in machine learning. The name refers to the idea of test error converging to train error *for all* functions in a function class. The framework dates back at least to the work of Vapnik and Chervonenkis (1971).

The mechanism for proving uniform convergence bounds relies on the size of a function space being, in a sense, *small* compared to the number of training examples. Since function spaces of interest may naïvely contain infinitely many functions, one needs to be careful about how one counts functions.

In the case of binary classification, a suitable means of counting functions is to count the number of distinct binary labellings that the function space is able to assign to a finite set of inputs. This is known as the *shattering coefficient*, which given the notion of *function projection* (Definition 2.1), is easy to define:

Definition 7.1 (Shattering coefficient). Given a machine learning model $f : \mathcal{X} \times \mathcal{W} \to \mathbb{R}$, the *shattering coefficient* $\mathcal{N}(f, m)$ is the largest number of binary labellings that f can realise on any set of m training inputs $X \in \mathcal{X}^m$:

$$\mathcal{N}(f, m) := \max_{X \in \mathcal{X}^m} \#\{\operatorname{sign} f_X(w) \mid w \in \mathcal{W}\}. \tag{7.1}$$

Since there are only 2^m possible binary labellings of m points, it holds that $\mathcal{N}(f, m) \leq 2^m$. A VC (Vapnik-Chervonenkis) bound relies on the machine learning model f being so limited in its capacity that $\mathcal{N}(f, m) \ll 2^m$:

Theorem 7.1 (VC bound). *Consider a machine learning model $f : \mathcal{X} \times \mathcal{W} \to \mathbb{R}$ that makes classifications by binarising its output $f \mapsto \operatorname{sign} f$. For a training set $S \in (\mathcal{X} \times \{\pm 1\})^m$ drawn iid from a data distribution $\mathcal{D}$, let $\mathcal{L}_{0/1}^S(w)$ denote the zero-one loss of weight vector $w \in \mathcal{W}$ with respect to training set S, and define $\mathcal{L}_{0/1}^{\mathcal{D}}(w) := \mathbb{E}_{x,y \sim \mathcal{D}} \mathbb{I}[\operatorname{sign} f(x, w) \neq y]$ to be the average misclassification rate over $\mathcal{D}$. Then for a fraction $1 - \delta$ of such training sets drawn in this way:*

$$\textit{For all } w \in \mathcal{W}, \quad \mathcal{L}_{0/1}^{\mathcal{D}}(w) \leq \mathcal{L}_{0/1}^S(w) + \sqrt{\frac{8}{m} \left(\log \mathcal{N}(f, 2m) + \log \frac{4}{\delta} \right)}. \tag{7.2}$$

This example of a VC bound is taken from the textbook of Schölkopf and Smola (2001). A core feature of this bound is that the amount by which the train error can exceed the test error is bounded *for all* weight vectors. Provided that the shattering coefficient saturates at some number of training examples m, then the bound is $\mathcal{O}(1/\sqrt{m})$.

Applicability to neural networks

The relevance of a result like Theorem 7.1 for deep learning practice is unclear. Often, neural networks operate in a regime where they can realise essentially any labelling of their training set (Zhang et al., 2017). This means that the shattering coefficient $\mathcal{N}(f, m) \approx 2^m$. In turn, the complexity term $\log \mathcal{N}(f, 2m)/m$ appearing in Theorem 7.1 does not decay as m increases.

One could resolve this problem by restricting the neural network function space, excluding the kinds of highly expressive function that allow the shattering coefficient $\mathcal{N}(f, 2m)$ to approach 2^m. It is not obvious how to accomplish this, although there have been proposals. For instance, Bartlett et al. (2017) propose an approach based on restricting to neural network functions with bounded layerwise spectral norms. Such approaches have not yet led to non-vacuous generalisation guarantees for deep neural networks.

7.2 Algorithmic stability

The distinguishing feature of a uniform convergence bound is that the bound holds for all functions in the function space. Usually, in practice, one is interested in the generalisation performance of only a single function—the one that is returned by the learning algorithm. If the learning algorithm has special properties, then perhaps the function that it returns could generalise significantly better than other functions in the function space. This motivates the framework of *algorithmic stability* (Bousquet and Elisseeff, 2002). To introduce this framework, it first helps to formally define the learning algorithm:

Definition 7.2 (Learning algorithm). A *learning algorithm A* is a deterministic mapping $S \mapsto A(S)$ from a training set S to a weight vector $A(S)$.

In practice, a learning algorithm need not be deterministic. For instance, the order in which the training set is fed into the algorithm will often affect the returned weight vector. But Definition 7.2 glosses over this detail.

Algorithmic stability provides generalisation guarantees for learning algorithms that obey some kind of stability property. The following definition establishes a notion of stability for learning algorithms with respect to manipulating a training set S by removing the ith example $S \mapsto S^{\backslash i}$:

Definition 7.3 (Uniform stability). Let $\ell(w, z) \in \mathbb{R}$ denote the loss incurred by weight vector $w \in \mathcal{W}$ on a single training example $z \in (\mathcal{X} \times \mathcal{Y})$. A learning

algorithm A has *uniform stability* β with respect to loss ℓ if the following holds: for all training sets $S \in (\mathcal{X} \times \mathcal{Y})^m$ and training examples $i \in \{1, ..., m\}$,

$$\sup_{z \in \mathcal{X} \times \mathcal{Y}} \left| \ell\big(A(S), z\big) - \ell\big(A(S^{\backslash i}), z\big) \right| < \beta. \tag{7.3}$$

In words: if for any training set, removing any training example leads to a small change in loss on any test point, then the learning algorithm is uniformly stable. This definition leads to the following generalisation bound for regression:

Theorem 7.2 (Uniform stability bound). *Let A be a learning algorithm with uniform stability β with respect to a bounded loss $0 \leq \ell(\cdot, \cdot) \leq 1$. For a data distribution $\mathcal{D}$, define the test loss $\mathcal{L}_{\mathcal{D}}(w) := \mathbb{E}_{z \sim \mathcal{D}} \, \ell(w, z)$ and train loss $\mathcal{L}_S(w) := \mathbb{E}_{z \sim UNIF(S)} \, \ell(w, z)$. Then, for a fraction $1 - \delta$ of training sets $S \overset{iid}{\sim} \mathcal{D}^m$:*

$$\mathcal{L}_{\mathcal{D}}(A(S)) \leq \mathcal{L}_S(A(S)) + 2\beta + (4m\beta + 1) \cdot \sqrt{\frac{\log 1/\delta}{2m}}. \tag{7.4}$$

This theorem is due to Bousquet and Elisseeff (2002). The theorem is most interesting for a learning algorithm with uniform stability $\beta = \mathcal{O}(1/m)$, for which the complexity term in Equation 7.4 decays like $\mathcal{O}(1/\sqrt{m})$, similar to the VC bound of Theorem 7.1.

Applicability to neural networks

It is not obvious whether neural networks satisfy a stability condition such as Definition 7.3 at a non-trivial level β. Sometimes neural networks are seemingly able to interpolate any training set (Zhang et al., 2017), meaning that they are highly sensitive to the inclusion or exclusion of any particular data point.

To make this framework workable for neural networks, one would need to produce a stability condition that is satisfied by a neural network. This could potentially be a weaker notion of stability than uniform stability, and some alternatives are proposed in the paper by Bousquet and Elisseeff (2002).

7.3 PAC-Bayes

PAC-Bayes theory presents another approach to introducing algorithm dependence into a generalisation bound. The theory makes a conceptual shift from the frameworks of both uniform convergence as well as algorithmic stability. It is worth introducing this new perspective in three steps:

Step 1: *Specify prior belief.* Before seeing the training data, assign a prior measure P to the function space. The idea is that functions that are believed more likely to explain the training data (upon its arrival) should be assigned higher probability under this prior measure.

Step 2: *Given data, construct a posterior.* Once the training data has arrived, a posterior measure Q should be constructed that assigns higher probability to functions that now seem more likely. The posterior is, in a sense, *algorithm dependent*: the choice of which functions to include in the posterior constitutes the learning algorithm. Although it can be, the posterior Q need not be set to the Bayesian posterior for prior P.

Step 3: *Measure distance between prior and posterior.* According to PAC-Bayes theory, the closer the posterior Q is to the prior P, the better the functions in the posterior Q will generalise on average. The idea is that if the posterior and prior are very similar, then less information was extracted from the training data. To quantify this, a distance on probability measures is required, such as the KL divergence $\mathrm{KL}(Q||P)$.

These three steps underlie the following theorem of Langford and Seeger (2001):

Theorem 7.3 (PAC-Bayes bound)**.** *Let P be a prior over functions realised by a classifier and let $\mathcal{D}$ denote the data distribution. For a fraction $1 - \delta$ of training sets $S \overset{iid}{\sim} \mathcal{D}^m$, the following holds for all posterior distributions Q:*

$$\mathrm{KL}(\text{BERN}_{S,Q}||\text{BERN}_{\mathcal{D},Q}) \leq \frac{\mathrm{KL}(Q||P) + \log(2m/\delta)}{m - 1}, \qquad (7.5)$$

where $\text{BERN}_{S,Q}$ is Bernoulli with probability $\mathbb{P}_{w \sim Q,(x,y) \sim \text{UNIF}(S)}[\mathrm{sign}\, f(x; w) \neq y]$ and $\text{BERN}_{\mathcal{D},Q}$ is Bernoulli with probability $\mathbb{P}_{w \sim Q,(x,y) \sim \mathcal{D}}[\mathrm{sign}\, f(x; w) \neq y]$.

In words: the KL divergence appearing on the left-hand side measures the distance between train error and test error averaged over the posterior. This is upper bounded by the KL divergence between prior and posterior (plus a logarithmic confidence term) divided by the number of training examples.

The following corollary may be slightly easier to parse than Theorem 7.3. It specialises to the *realisable* setting of functions that attain zero train error.

Corollary 7.1 (Realisable PAC-Bayes). *Let P be a prior over functions realised by a classifier. For a fraction $1 - \delta$ of training sets $S \overset{iid}{\sim} \mathcal{D}^m$, the following holds for all posterior distributions Q over functions that correctly classify S:*

$$\mathbb{P}_{w \sim Q, (x,y) \sim \mathcal{D}}[\operatorname{sign} f(x; w) \neq y] \leq 1 - \exp\left[-\frac{\operatorname{KL}(Q||P) + \log(2m/\delta)}{m - 1}\right]. \quad (7.6)$$

Proof. For a posterior Q that correctly classifies S, the Bernoulli random variable $\textsc{bern}_{S,Q}$ is zero with probability one. In turn, this implies that:

$$\operatorname{KL}(\textsc{bern}_{S,Q}||\textsc{bern}_{\mathcal{D},Q}) = -\log(1 - \mathbb{P}_{w \sim Q, (x,y) \sim \mathcal{D}}[\operatorname{sign} f(x; w) \neq y]).$$

Substituting this relation into Theorem 7.3 and rearranging yields the result. □

Applicability to neural networks

PAC-Bayes has been shown to yield non-vacuous generalisation guarantees both for neural networks (Dziugaite and Roy, 2017) and for infinitely wide neural networks (Valle-Pérez and Louis, 2020; Bernstein et al., 2022). Indeed, by inspection of Corollary 7.1, when $\operatorname{KL}(Q||P)$ is finite, the right-hand side of Inequality 7.6 is smaller than one and the bound is never vacuous.

One of the main problems with applying Theorem 7.3 or Corollary 7.1 to neural networks is that there is a mismatch between what is bounded and what matters in a standard machine learning problem. In particular, PAC-Bayes bounds are on the average test error over an ensemble of functions, described by posterior distribution Q. But in a standard machine learning scenario, one is most interested in the generalisation error of an individual function. Steps towards resolving this issue will be taken in Chapter 9.

References

Peter L. Bartlett, Dylan J. Foster, and Matus Telgarsky (2017). "Spectrally-normalized margin bounds for neural networks". In: *Neural Information Processing Systems.*

Jeremy Bernstein, Alexander R. Farhang, and Yisong Yue (2022). "Max-margin neural networks as Bayes point machines". In preparation.

Olivier Bousquet and André Elisseeff (2002). "Stability and generalization". In: *Journal of Machine Learning Research.*

Gintare Karolina Dziugaite and Daniel M. Roy (2017). "Computing nonvacuous generalization bounds for deep (stochastic) neural networks with many more parameters than training data". In: *Uncertainty in Artificial Intelligence.*

John Langford and Matthias Seeger (2001). *Bounds for averaging classifiers.* Technical report. Carnegie Mellon University.

Bernhard Schölkopf and Alexander J. Smola (2001). *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond.* MIT Press.

Guillermo Valle-Pérez and Ard A. Louis (2020). "Generalization bounds for deep learning". In: *arXiv:2012.04115.*

Vladimir N. Vapnik and Alexey Ya. Chervonenkis (1971). "On the uniform convergence of relative frequencies of events to their probabilities". In: *Theory of Probability & Its Applications.*

Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals (2017). "Understanding deep learning requires rethinking generalization". In: *International Conference on Learning Representations.*

Chapter 8

PAC-BAYES FOR GAUSSIAN PROCESSES

> This chapter derives PAC-Bayes risk bounds for Gaussian process classi-
> fication. While the specific calculations are novel, the general framework
> is due to Seeger (2002). The calculations will be useful in Chapter 9.

Gaussian processes already contain many of the perplexing phenomena present
in neural networks. They represent a function space of essentially unlimited
expressivity, that nevertheless tends to generalise well when fit to data.

PAC-Bayes theory applies naturally to Gaussian process classification since
Gaussian processes come equipped with prior and posterior measures that
can be used in the PAC-Bayes bound. Seeger (2002) developed an extensive
PAC-Bayesian treatment of Gaussian process classification. Those results apply
to a very flexible set of methods known as *approximate Gaussian process clas-
sifiers*. Essentially, these methods use a Gaussian distribution to approximate
an intractable Bayesian posterior. Since the PAC-Bayes bound applies to
any posterior and not just the Bayesian posterior, PAC-Bayes theory can be
straightforwardly applied to control the error of these approximate methods.

The focus of this chapter is on developing a simple and direct PAC-Bayesian
treatment of Gaussian process classification for use in Chapter 9.

8.1 Gaussian process classification

There is a simple way to convert a Gaussian process into a binary classifier.
Given a training sample $S = (X, Y)$, one can simply draw functions from the
prior $f \sim \mathrm{GP}(0, k)$ until a function is found that correctly classifies the training
sample: $\mathrm{sign}\, f_X = Y$. This function can then be evaluated on novel test points.

To develop a PAC-Bayesian generalisation theory of this procedure, one needs
to write down the corresponding posterior and compute its KL-divergence with
the prior. To ease this process, it is helpful to decompose the prior into two
separate pieces, each defined implicitly via a sampling procedure:

Definition 8.1 (GP binary classifier prior)**.** To construct the *prior* $P_{\mathrm{GP}}(\cdot)$ of a GP

binary classifier, take train inputs X and any other inputs X' and decompose:

$$P_{\mathrm{GP}}(f_{X'}, f_X) = P_{\mathrm{GP}}(f_{X'}|f_X) \cdot P_{\mathrm{GP}}(f_X). \qquad (8.1)$$

Now, define each piece of this decomposition separately:

1) The measure $P_{\mathrm{GP}}(f_X)$ corresponds to sampling train outputs:

$$f_X \sim \mathrm{NORMAL}(0, K_{XX}). \qquad (8.2)$$

2) The measure $P_{\mathrm{GP}}(f_{X'}|f_X)$ corresponds to sampling the other outputs:

$$f_{X'} \sim \mathrm{NORMAL}(K_{X'X}K_{XX}^{-1}f_X, K_{X'X'} - K_{X'X}K_{XX}^{-1}K_{XX'}). \qquad (8.3)$$

One might worry that this definition of the prior depends on the training sample X, which is expressly not allowed. But this dependence is illusory: by Theorem 2.2, this definition is equivalent to sampling $f_{X'}, f_X \sim \mathrm{NORMAL}(0, K_{X'\cup X \, X'\cup X})$. Breaking the sampling procedure in two like this is helpful for making a direct comparison with the following definition of the posterior:

Definition 8.2 (GP binary classifier posterior). To construct the *posterior* $Q_{\mathrm{GP}}(\cdot)$ of a GP binary classifier, for train inputs X and other inputs X', decompose:

$$Q_{\mathrm{GP}}(f_{X'}, f_X) = Q_{\mathrm{GP}}(f_{X'}|f_X) \cdot Q_{\mathrm{GP}}(f_X). \qquad (8.4)$$

Now, define each piece of this decomposition separately:

1) The measure $Q_{\mathrm{GP}}(f_X)$ corresponds to sampling train outputs:

$$f_X \sim \mathrm{NORMAL}(0, K_{XX} \mid \mathrm{sign}\, f_X = Y). \qquad (8.5)$$

2) The measure $Q_{\mathrm{GP}}(f_{X'}|f_X) := P_{\mathrm{GP}}(f_{X'}|f_X)$ from Definition 8.1.

The posterior is identical to the prior, except that the distribution on train outputs is truncated to the orthant $\{f_X \in \mathbb{R}^m \mid \mathrm{sign}\, f_X = Y\}$.

It will also turn out to be both computationally and analytically convenient to define the following approximate posterior distribution:

Definition 8.3 (GP binary classifier spherised posterior)**.** For train inputs X and other inputs X', the *spherised posterior* $Q_{\mathrm{sph}}(\cdot)$ is defined by first decomposing:

$$Q_{\mathrm{sph}}(f_{X'}, f_X) = Q_{\mathrm{sph}}(f_{X'}|f_X) \cdot Q_{\mathrm{sph}}(f_X). \qquad (8.6)$$

And, next, defining each piece of this decomposition separately:

1) The measure $Q_{\mathrm{sph}}(f_X)$ corresponds to sampling train outputs:

$$f_X \sim \mathrm{NORMAL}(0, \mathbf{I} \cdot |K_{XX}|^{1/m} \mid \mathrm{sign}\, f_X = Y). \qquad (8.7)$$

2) The measure $Q_{\mathrm{sph}}(f_{X'}|f_X) \coloneqq P_{\mathrm{GP}}(f_{X'}|f_X)$ from Definition 8.1.

The spherised posterior modifies the posterior of Definition 8.2 by replacing the truncated Gaussian distribution over train outputs $Q_{\mathrm{GP}}(f_X)$ with a truncated spherical Gaussian distribution $Q_{\mathrm{sph}}(f_X)$. The factor $|K_{XX}|^{1/m}$ is included to match the variance scales of these two distributions.

8.2 The KL divergence between prior and posterior

To construct PAC-Bayes bounds, one needs to compute the KL-divergence between these prior and posterior measures over functions. A lemma will help:

Lemma 8.1 (Chain rule for the KL divergence)**.** *Let Q and P denote measures over functions $f : \mathcal{X} \to \mathbb{R}$, for a finite input space $\mathcal{X}$. For a collection of m inputs $X \in \mathcal{X}^m$, let X^{c} denote all inputs excluding X: $X^{\mathrm{c}} \coloneqq \mathcal{X} \setminus X$. Then:*

$$\mathrm{KL}\big(Q(f) \,||\, P(f)\big) = \mathrm{KL}\big(Q(f_{X^{\mathrm{c}}}|f_X) \,||\, P(f_{X^{\mathrm{c}}}|f_X)\big) + \mathrm{KL}\big(Q(f_X) \,||\, P(f_X)\big). \qquad (8.8)$$

Proof. First, a full function $f \equiv f_{\mathcal{X}} \equiv (f_{X^{\mathrm{c}}}, f_X)$. The result follows by substituting the chain rule for joint probability into the definition of the KL divergence, separating the logarithm, and recognising the two KL divergences:

$$
\begin{aligned}
\mathrm{KL}(Q(f) \,||\, P(f)) &= \int \mathrm{d}f\, Q(f) \log \frac{Q(f)}{P(f)} \\
&= \int \mathrm{d}f_X\, Q(f_X) \int \mathrm{d}f_{X^{\mathrm{c}}}\, Q(f_{X^{\mathrm{c}}}|f_X) \log \frac{Q(f_{X^{\mathrm{c}}}|f_X) \cdot Q(f_X)}{P(f_{X^{\mathrm{c}}}|f_X) \cdot P(f_X)} \\
&= \int \mathrm{d}f_X\, Q(f_X) \left[\mathrm{KL}(Q(f_{X^{\mathrm{c}}}|f_X) \,||\, P(f_{X^{\mathrm{c}}}|f_X)) + \log \frac{Q(f_X)}{P(f_X)} \right] \\
&= \mathrm{KL}(Q(f_{X^{\mathrm{c}}}|f_X) \,||\, P(f_{X^{\mathrm{c}}}|f_X)) + \mathrm{KL}(Q(f_X) \,||\, P(f_X)).
\end{aligned}
$$

The proof is complete. $\qquad\square$

Note that machine learning methods implemented on computers use finite input spaces. It is possible to generalise the lemma beyond finite input spaces, but this requires concepts from measure theory such as *Radon-Nikodym derivatives* and the *disintegration theorem*, which are beyond the scope of this thesis.

The relevance of this result is that, since the second sampling steps of Definitions 8.1, 8.2 and 8.3 are identical, it holds that:

$$\text{KL}(Q_{\text{GP}}(f_{X^c}|f_X) \,||\, P(f_{X^c}|f_X)) = \text{KL}(Q_{\text{sph}}(f_{X^c}|f_X) \,||\, P(f_{X^c}|f_X)) = 0. \quad (8.9)$$

In turn, by Lemma 8.1, this implies that:

$$\text{KL}(Q_{\text{GP}}(f) \,||\, P(f)) = \text{KL}(Q_{\text{GP}}(f_X) \,||\, P(f_X)); \quad (8.10)$$

$$\text{KL}(Q_{\text{sph}}(f) \,||\, P(f)) = \text{KL}(Q_{\text{sph}}(f_X) \,||\, P(f_X)). \quad (8.11)$$

These relations mean that, in order to evaluate the KL divergence between the full prior and posterior, one only needs to evaluate the KL divergence between the prior and posterior restricted to the training inputs. To derive these KL divergences, it is first helpful to define two quantities:

Definition 8.4 (Gaussian orthant probability). Given a training set $S = (X, Y)$ with binary labels $Y \in \{\pm 1\}^m$, the *Gaussian orthant probability* P_Y is:

$$P_Y := \mathbb{P}_{f_X \sim P_{\text{GP}}}[\text{sign}\, f_X = Y]. \quad (8.12)$$

This is termed an *orthant probability* because for a training set X consisting of m examples, the set $\{f_X \in \mathbb{R}^m \,|\, \text{sign}\, f_X = Y\}$ is an orthant of $\mathbb{R}^m$.

Definition 8.5 (Kernel complexity). Given a kernel k and a training set $S = (X, Y)$, the *kernel complexity* $\mathcal{A}(k, X, Y)$ is given by:

$$\mathcal{A}(k, X, Y) := m \cdot \left(\log 2 - \frac{1}{2}\right) +$$
$$|K_{XX}|^{1/m} \cdot \left[\left(\frac{1}{2} - \frac{1}{\pi}\right) \text{tr}\, K_{XX}^{-1} + \frac{1}{\pi} Y^T K_{XX}^{-1} Y\right]. \quad (8.13)$$

Given these definitions, the following KL divergences may be obtained:

Lemma 8.2 (KL divergences for Gaussian process classification). *Given a training set $S = (X, Y)$ with binary labels $Y \in \{\pm 1\}^m$ and a kernel k:*

$$\text{KL}(Q_{\text{GP}} \,||\, P_{\text{GP}}) = \log(1/P_Y) \le \mathcal{A}(k, X, Y) = \text{KL}(Q_{\text{sph}} \,||\, P_{\text{GP}}). \quad (8.14)$$

Proof. First, by Lemma 8.1 and the observation that the KL divergences from both $Q_{\mathrm{GP}}(f_{X^c}|f_X)$ and $Q_{\mathrm{sph}}(f_{X^c}|f_X)$ to $P_{\mathrm{GP}}(f_{X^c}|f_X)$ are zero, it is enough to relate $\mathrm{KL}(Q_{\mathrm{GP}}(f_X) \parallel P_{\mathrm{GP}}(f_X))$ to $\mathrm{KL}(Q_{\mathrm{sph}}(f_X) \parallel P_{\mathrm{GP}}(f_X))$.

To establish the first equality, observe that since $Q_{\mathrm{GP}}(f_X)$ and $P_{\mathrm{GP}}(f_X)$ differ on the support of $Q_{\mathrm{GP}}(f_X)$ only by normalisation constant P_Y, it holds that:

$$\mathrm{KL}(Q_{\mathrm{GP}}(f_X) \parallel P_{\mathrm{GP}}(f_X)) = \mathbb{E}_{f_X \sim Q_{\mathrm{GP}}} \log(1/P_Y) = \log(1/P_Y).$$

The last equality is derived by first observing that:

$$\mathrm{KL}(Q_{\mathrm{sph}}(f_X) \parallel P_{\mathrm{GP}}(f_X)) = \mathbb{E}_{f_X \sim Q_{\mathrm{sph}}} \log \frac{2^n \cdot e^{-\frac{1}{2}\|f_X\|_2^2 \cdot |K_{XX}|^{-1/m}}}{e^{-\frac{1}{2}f_X^\top K_{XX}^{-1} f_X}}$$

$$= n \log 2 + \tfrac{1}{2} \mathbb{E}_{f_X \sim Q_{\mathrm{sph}}}[f_X^\top (K_{XX}^{-1} - \mathbf{I}|K_{XX}|^{-1/m}) f_X].$$

To obtain $\mathcal{A}(k, X, Y)$, one must substitute in the following identity for half-Normal random variables:

$$\mathbb{E}_{f_X \sim Q_{\mathrm{sph}}}[f_X^i \cdot f_X^j] = |K_{XX}|^{1/m} \cdot \left[\delta_{ij} + \tfrac{2}{\pi} Y_i Y_j (1 - \delta_{ij})\right].$$

Finally, the inequality follows via:

$$\mathrm{KL}(Q_{\mathrm{sph}}(f_X) \parallel P_{\mathrm{GP}}(f_X)) = \mathbb{E}_{f_X \sim Q_{\mathrm{sph}}} \left[\log \frac{Q_{\mathrm{sph}}(f_X)}{Q_{\mathrm{GP}}(f_X)} + \log \frac{Q_{\mathrm{GP}}(f_X)}{P_{\mathrm{GP}}(f_X)}\right]$$

$$= \mathrm{KL}(Q_{\mathrm{sph}}(f_X) \parallel Q_{\mathrm{GP}}(f_X)) + \log(1/P_Y),$$

and noting that $\mathrm{KL}(Q_{\mathrm{sph}}(f_X) \parallel Q_{\mathrm{GP}}(f_X)) \geq 0$. $\qquad\square$

8.3   PAC-Bayes bounds

Given the results in Sections 8.1 and 8.2, it is now a simple matter to write down PAC-Bayesian generalisation bounds for a Gaussian process binary classifier:

Theorem 8.1 (PAC-Bayes for Gaussian process classification). *Given a kernel k and a training sample $S = (X, Y)$, recall the definitions of the kernel complexity $\mathcal{A}$ (Definition 8.5) and the Gaussian orthant probability P_Y (Definition 8.4). Given a data distribution $\mathcal{D}$ over $\mathcal{X} \times \{\pm 1\}$, for a fraction $1 - \delta$ of training sets $S \stackrel{iid}{\sim} \mathcal{D}^m$, the following bounds hold simultaneously:*

$$\mathbb{P}_{f \sim Q_{\mathrm{GP}}, (x,y) \sim \mathcal{D}}[\mathrm{sign}\, f(x) \neq y] \leq 1 - \exp\left[-\frac{\log 1/P_Y + \log(2m/\delta)}{m - 1}\right] \tag{8.15}$$

$$\leq 1 - \exp\left[-\frac{\mathcal{A}(k, X, Y) + \log(2m/\delta)}{m - 1}\right]; \tag{8.16}$$

$$\mathbb{P}_{f \sim Q_{\mathrm{sph}}, (x,y) \sim \mathcal{D}}[\mathrm{sign}\, f(x) \neq y] \leq 1 - \exp\left[-\frac{\mathcal{A}(k, X, Y) + \log(2m/\delta)}{m - 1}\right]. \tag{8.17}$$

Proof. Instantiate Theorem 7.3 with prior $P_{\mathrm{GP}}(f)$, the two posteriors $Q_{\mathrm{GP}}(f)$ and $Q_{\mathrm{sph}}(f)$ and the KL divergences from Lemma 8.2. $\qquad\square$

According to Theorem 8.1, the posterior Q_{GP} enjoys a tighter risk bound than the spherised posterior Q_{sph}. So why introduce the spherised posterior? There are two reasons:

1) It is significantly easier to sample from the spherised posterior (Definition 8.3) than the original posterior (Definition 8.2).

2) Since the kernel complexity measure $\mathcal{A}(k, X, Y)$ is a closed-form analytical expression, whereas the Gaussian orthant probability P_Y requires computing a high-dimensional integral, Inequality 8.17 is much easier to evaluate than Inequality 8.15.

These results will be used in Chapter 9 to study generalisation in neural networks through the lens of the neural network–Gaussian process correspondence.

References

Matthias Seeger (2002). "PAC-Bayesian generalisation error bounds for Gaussian process classification". In: *Journal of Machine Learning Research.*

Chapter 9

NEURAL NETWORKS AS BAYES POINT MACHINES

> This chapter introduces a novel correspondence between neural networks and kernels. In particular, as both width and normalised margin are sent to infinity, the neural network function space concentrates on a particular kernel classifier. This kernel classifier aggregates over all infinitely wide neural networks that correctly classify the train sample. This offers a new, potentially fruitful perspective as to why neural networks generalise.

Chapter 8 derived PAC-Bayes bounds for Gaussian process classification. These bounds may be readily transferred to neural networks—albeit infinitely wide ones—by leveraging the neural network–Gaussian process correspondence from Section 3.4. Having these bounds is certainly nice, and they have been found to be non-vacuous (Seeger, 2002; Valle-Pérez and Louis, 2020). But what would be nicer is having the bounds inform some aspect of deep learning practice. For instance: could the bounds tell which single function should generalise best?

There is a problem, however. Theorem 8.1 bounds the misclassification rate of the Gaussian process averaged over posterior draws. This leaves room for individual draws to generalise either significantly worse or significantly better than average. Then which single function should be returned in practice?

To answer this question, the chapter takes a detour through *Bayesian classification strategies*. A Bayesian classification strategy is a rule for combining an input x with a posterior distribution Q over classifiers to yield a prediction. The most natural of these strategies involve either randomly sampling a single posterior function or aggregating over the posterior. Bayesian wisdom, and also certain technical results (Lacasse et al., 2007), suggest that aggregation over the posterior should perform best. This is for the simple reason that aggregation removes the variance incurred by sampling a posterior function.

But the trouble with aggregation is that it is expensive. Naïvely, it involves either integrating or summing over lots of posterior functions. To get the benefits of aggregation without the associated cost, this chapter picks up on an

old idea from the kernels literature. A *Bayes point machine* (Herbrich et al., 2001) is a single posterior sample that, by itself, approximates the posterior aggregate. In the context of neural networks, this suggests the question:

> Can a single neural network report the aggregated classification of an ensemble of networks?

Given the expressive power of the neural network function class, it might seem reasonable that the answer could be yes. In that case:

> How can such a network be found?

This chapter attempts to resolve these two questions. The chapter argues that, in the limit of large width and large normalised margin, the entire space of neural networks that interpolate a training set *concentrates* on a kernel classifier that itself aggregates over a posterior distribution. This implies that a single wide neural network trained to large normalised margin will attain the same reduction in the variance of its predictions as an aggregated Bayesian method.

9.1 Bayesian classification strategies

From a Bayesian perspective, there are three natural ways to use a posterior Q over functions to classify a fresh input $x \in \mathcal{X}$. The first is the random strategy:

Definition 9.1 (Gibbs classifier). The *Gibbs classifier* returns a random draw:

$$f_{\text{Gibbs}}(x) := \text{sign}\, f(x) \text{ for } f \sim Q. \tag{9.1}$$

PAC-Bayes (Theorem 7.3) bounds the probability that the Gibbs classifier misclassifies a randomly drawn test point. But the Gibbs classifier, being random, has the misfortune of containing variance. A Bayesian would like to deal with this issue by integrating, or *aggregating*, over the posterior, and thus removing this variance. This motivates the second strategy:

Definition 9.2 (Bayes classifier). The *Bayes classifier* returns the majority vote:

$$f_{\text{Bayes}}(x) := \text{sign}\, \mathbb{E}_{f \sim Q}\, \text{sign}\, f(x). \tag{9.2}$$

The majority is one form of aggregation. The third strategy employs another:

Definition 9.3 (BPM classifier)**.** The *BPM classifier* returns the simple average:

$$f_{\mathrm{BPM}}(x) := \mathrm{sign}\,\mathbb{E}_{f\sim Q}\,f(x). \tag{9.3}$$

The abbreviation BPM is short for *Bayes point machine.* Two observations motivate this terminology:

1) The BPM classifier is obtained by reversing the order of the sign and expectation operators in the Bayes classifier:

$$\underbrace{\mathrm{sign}\,\mathbb{E}_{f\sim Q}\,\overset{\frown}{\mathrm{sign}}f(x)}_{\text{Bayes classifier}} \approx \underbrace{\mathrm{sign}\,\mathbb{E}_{f\sim Q}\,f(x)}_{\text{BPM classifier}}. \tag{9.4}$$

This operator exchange amounts to approximating the ensemble majority by a single point in function space: the ensemble centre-of-mass. Approximation 9.4 is referred to as the *the BPM approximation.* The quality of this approximation will be considered in this chapter.

2) Suppose that the classifier has *hidden linearity* when represented in weight space. In particular, consider classifier $f_\varphi(x; w) := \varphi(x)^\top w$, where φ is an arbitrary nonlinear input embedding. Then:

$$\underbrace{\mathrm{sign}\,\mathbb{E}_{w\sim Q}\,f_\varphi(x; w)}_{\text{BPM classifier}} = \mathrm{sign}\,f_\varphi(x; \underbrace{\mathbb{E}_{w\sim Q}\,w}_{\text{weight space centre-of-mass}}). \tag{9.5}$$

In words: for linear classifiers, the BPM classifier is equivalent to a single point in weight space: the posterior Q's weight space centre-of-mass.

Of course, since the sign function is nonlinear, the BPM approximation is not always correct. Herbrich (2001) calls it a *trick.* Is the approximation ever correct? In the case of hidden linearity (Equation 9.5), the approximation is correct when over half the ensemble agrees with the centre-of-mass on an input. This happens, for example, when the posterior Q is point symmetric about the centre-of-mass (Herbrich, 2001). But point symmetry is a strong assumption that does not hold for, say, the posterior of a GP classifier (Definition 8.2).

The next section presents a novel result on the quality of the BPM approximation. This result leverages a novel connection between Bayesian classification strategies and certain objects of study in convex geometry (Grünbaum, 1960) and social choice theory (Caplin and Nalebuff, 1991). The result leads to a novel bound on the generalisation error of the BPM classifier.

9.2 Relationships between classification strategies

This section presents relations, one of which is novel, between the test error of the three classification strategies introduced in the previous section. In each case, the test error is measured over a data distribution $\mathcal{D}$ on $\mathcal{X} \times \{\pm 1\}$. It will help to formally define the three notions of error considered:

First, the Gibbs error measures the misclassification rate averaged over both the data distribution and the posterior:

Definition 9.4 (Gibbs error). The *Gibbs error* $\varepsilon_{\text{Gibbs}} \in [0, 1]$ is given by:

$$\varepsilon_{\text{Gibbs}} := \mathbb{E}_{f \sim Q} \, \mathbb{E}_{(x,y) \sim \mathcal{D}} \, \mathbb{I}\Big[\operatorname{sign} f(x) \neq y\Big]. \tag{9.6}$$

Meanwhile, the Bayes error measures the misclassification rate of the posterior majority, averaged over the data distribution:

Definition 9.5 (Bayes error). The *Bayes error* $\varepsilon_{\text{Bayes}} \in [0, 1]$ is given by:

$$\varepsilon_{\text{Bayes}} := \mathbb{E}_{(x,y) \sim \mathcal{D}} \, \mathbb{I}\Big[f_{\text{Bayes}}(x) \neq y\Big]; \tag{9.7}$$

Finally, the BPM error measures the misclassification rate of the posterior mean, averaged over the data distribution:

Definition 9.6 (BPM error). The *BPM error* $\varepsilon_{\text{BPM}} \in [0, 1]$ is given by:

$$\varepsilon_{\text{BPM}} := \mathbb{E}_{(x,y) \sim \mathcal{D}} \, \mathbb{I}\Big[f_{\text{BPM}}(x) \neq y\Big]. \tag{9.8}$$

Various relationships exist between these three notions of error. A classic example is that the Bayes error cannot be more than twice the Gibbs error:

Lemma 9.1 (Pessimistic Gibbs–Bayes). *For any ensemble of classifiers Q,*

$$\varepsilon_{\text{Bayes}} \leq 2 \cdot \varepsilon_{\text{Gibbs}}.$$

Proof. First, consider the Bayes and Gibbs errors on a single datapoint (x, y):

$$\varepsilon_{\text{Bayes}}(x, y) := \mathbb{I}\left[\operatorname{sign} \mathbb{E}_{f \sim Q} \operatorname{sign} f(x) \neq y\right];$$
$$\varepsilon_{\text{Gibbs}}(x, y) := \mathbb{E}_{f \sim Q} \, \mathbb{I}\left[\operatorname{sign} f(x) \neq y\right].$$

When the Bayes classifier is correct, $\varepsilon_{\text{Bayes}}(x, y) = 0$. When the Bayes classifier is incorrect, $\varepsilon_{\text{Bayes}}(x, y) = 1$ and $\varepsilon_{\text{Gibbs}}(x, y) \geq 1/2$. In either case:

$$\varepsilon_{\text{Bayes}}(x, y) \leq 2 \cdot \varepsilon_{\text{Gibbs}}(x, y).$$

Taking the expectation over $(x, y) \sim \mathcal{D}$ yields the result. $\qquad \square$

This result is tagged *pessimistic* since one often expects the Bayes classifier to significantly outperform the Gibbs classifier: $\varepsilon_{\text{Bayes}} \ll \varepsilon_{\text{Gibbs}}$. This is because the Gibbs classifier is noisy, whereas the Bayes classifier aggregates over this noise. For this reason, Seeger (2002) referred to Lemma 9.1 as *crude.*

A potentially less crude relationship is given by the following lemma:

Lemma 9.2 (Optimistic Gibbs–Bayes). *Define the average Gibbs agreement:*

$$\alpha_{\text{Gibbs}} := \mathbb{E}_{x \sim \mathcal{D}} \left[\left[\mathbb{E}_{f \sim Q} \operatorname{sign} f(x) \right]^2 \right] \in [0, 1].$$

Then, for any ensemble of classifiers Q,

$$\varepsilon_{\text{Bayes}} \leq 1 - \frac{(1 - 2 \cdot \varepsilon_{\text{Gibbs}})^2}{\alpha_{\text{Gibbs}}}.$$

This result is usually known by a different name: *the C-bound.* Its proof is given by Lacasse et al. (2007). The result is tagged *optimistic* since it is capable of expressing that the Bayes classifier can significantly outperform the Gibbs classifier: $\varepsilon_{\text{Bayes}} \ll \varepsilon_{\text{Gibbs}}$. In particular, this happens when the ensemble members make very noisy predictions, such that the Gibbs error $\varepsilon_{\text{Gibbs}}$ is large but the Gibbs agreement α_{Gibbs} is small.

This thesis proves a novel relationship, analogous to Lemma 9.1, between the BPM error and the Gibbs error. The result leverages the following relationship between (sub)majorities and averages:

Lemma 9.3 (Weighted Grünbaum's inequality). *Let Q be a log-concave probability density supported on a convex subset of $\mathbb{R}^d$ with positive volume. Let $\mu \in \mathbb{R}^d$ denote the mean $\mu := \mathbb{E}_{w \sim Q} w$. Then for any vector $x \in \mathbb{R}^d$:*

$$\mathbb{P}_{w \sim Q} \left[\operatorname{sign}[w^\top x] = \operatorname{sign}[\mu^\top x] \right] \geq 1/e.$$

In words: for any input x, a fraction of at least $1/e \approx 36\%$ of the distribution reports the same binary classification as the mean. This result is due to economists Caplin and Nalebuff (1991), who were working in social choice theory. Their interest was in understanding what fraction of an electorate can disagree with the most average individual voter. The result generalises an inequality of Grünbaum (1960) on mass partitions in convex geometry.

Lemma 9.3 leads directly to the following analogue of Lemma 9.1:

Lemma 9.4 (Pessimistic Gibbs–BPM). *Consider an ensemble of classifiers whose distribution at all inputs $x \in \mathcal{X}$ follows:*

$$f_\varphi(x; w) = w^\top \varphi(x), \qquad w \sim Q,$$

for arbitrary nonlinear input embedding φ, and log-concave probability density Q supported on a convex subset of $\mathbb{R}^d$ with positive volume. Then:

$$\varepsilon_{\mathrm{BPM}} \leq \mathrm{e} \cdot \varepsilon_{\mathrm{Gibbs}}.$$

Proof. The proof mirrors the structure of the proof of Lemma 9.1. First, consider the BPM and Gibbs error on a single datapoint (x, y):

$$\varepsilon_{\mathrm{BPM}}(x, y) := \mathbb{I}\left[\operatorname{sign} \mathbb{E}_{w \sim Q} \, w^\top \varphi(x) \neq y\right];$$

$$\varepsilon_{\mathrm{Gibbs}}(x, y) := \mathbb{E}_{w \sim Q} \, \mathbb{I}\left[\operatorname{sign} w^\top \varphi(x) \neq y\right].$$

When the BPM classifier is correct, $\varepsilon_{\mathrm{BPM}}(x, y) = 0$. When the BPM classifier errs, $\varepsilon_{\mathrm{BPM}}(x, y) = 1$ and $\varepsilon_{\mathrm{Gibbs}}(x, y) \geq 1/e$ by Lemma 9.3. In either case:

$$\varepsilon_{\mathrm{BPM}}(x, y) \leq \mathrm{e} \cdot \varepsilon_{\mathrm{Gibbs}}(x, y).$$

Taking the expectation over $(x, y) \sim \mathcal{D}$ yields the result. $\qquad \square$

So, under the stated conditions of Lemma 9.4, the BPM error cannot be more than e times the Gibbs error. The result is tagged *pessimistic* since, in practice, one might expect the BPM classifier to perform significantly better than the noisy Gibbs classifier: $\varepsilon_{\mathrm{BPM}} \ll \varepsilon_{\mathrm{Gibbs}}$. While proving this intuition appears to be an open problem, the rest of this section provides one potential route.

The idea is that, when the Gibbs classifier is very noisy, Lemma 9.2 suggests a more optimistic relationship between the Gibbs and Bayes errors. But if the BPM approximation (Approximation 9.4) is good, then the BPM classifier should inherit the same favourable properties as the Bayes classifier. To pursue this idea, it will help to formalise the BPM approximation error:

Definition 9.7 (BPM approximation error). The *BPM approximation error Δ* is defined as follows:

$$\Delta := \mathbb{E}_{(x,y) \sim \mathcal{D}} \, \mathbb{I}\left[f_{\mathrm{BPM}}(x) \neq f_{\mathrm{Bayes}}(x)\right]. \tag{9.9}$$

So the BPM approximation error Δ measures at what rate the BPM classifier and the Bayes classifier disagree. The BPM approximation error Δ relates the BPM and Bayes errors as follows:

Lemma 9.5 (Bayes–BPM). *For any ensemble of classifiers Q,*

$$\varepsilon_{\mathrm{BPM}} \leq \varepsilon_{\mathrm{Bayes}} + \Delta.$$

Proof. First consider the BPM error, Bayes error and BPM approximation error on a single datapoint (x, y):

$$\varepsilon_{\mathrm{BPM}}(x, y) := \mathbb{I}\left[f_{\mathrm{BPM}}(x) \neq y\right];$$
$$\varepsilon_{\mathrm{Bayes}}(x, y) := \mathbb{I}\left[f_{\mathrm{Bayes}}(x) \neq y\right];$$
$$\Delta(x, y) := \mathbb{I}\left[f_{\mathrm{BPM}}(x) \neq f_{\mathrm{Bayes}}(x)\right].$$

When the BPM classifier is correct, $\varepsilon_{\mathrm{BPM}}(x, y) = 0$. Otherwise, $\varepsilon_{\mathrm{BPM}}(x, y) = 1$ and either $\varepsilon_{\mathrm{Bayes}}(x, y) = 1$ and $\Delta(x, y) = 0$ or vice versa. Thus:

$$\varepsilon_{\mathrm{BPM}}(x, y) \leq \varepsilon_{\mathrm{Bayes}}(x, y) + \Delta(x, y).$$

Taking the expectation over $(x, y) \sim \mathcal{D}$ yields the result. $\qquad\square$

Lemmas 9.2 and 9.5 may be directly combined to yield the following result:

Lemma 9.6 (Optimistic Gibbs–BPM). *Let α_{Gibbs} denote the average Gibbs agreement (Lemma 9.2) and let Δ denote the BPM approximation error. Then:*

$$\varepsilon_{\mathrm{BPM}} \leq 1 - \frac{(1 - 2 \cdot \varepsilon_{\mathrm{Gibbs}})^2}{\alpha_{\mathrm{Gibbs}}} + \Delta.$$

In words: when the BPM classifier is a good approximation to the Bayes classifier, and when the Gibbs classifier is noisy such that the Gibbs error is large but the Gibbs agreement is small, then the BPM classifier can substantially outperform the Gibbs classifier.

9.3 Kernel interpolation as a Bayes point machine

This section shows that kernel interpolators of minimum RKHS norm (Equation 2.7) are themselves Bayes point machine classifiers. While, in itself, this observation is not novel (Seeger, 2002), it yields novel PAC-Bayes generalisation bounds for kernel interpolators when combined with the novel Lemma 9.4.

To begin, the BPM classifier of the Gaussian process posterior Q_{GP} is the sign of the kernel interpolator of *centre-of-mass labels* $\mathbb{E}_{f_X \sim Q_{\mathrm{GP}}} f_X$:

Lemma 9.7 (BPM of a Gaussian process classifier is a kernel interpolator). *For the Gaussian process classification posterior Q_{GP} (Definition 8.2),*

$$f_{\mathrm{BPM}}(x) = \mathrm{sign}[K_{xX}K_{XX}^{-1}\,\mathbb{E}_{f_X \sim Q_{\mathrm{GP}}}\,f_X]. \tag{9.10}$$

Proof. The Gibbs classifier of Q_{GP} classifies a test point x in three steps:

$$\text{Sample train outputs:}\quad f_X \sim \mathrm{NORMAL}\!\left(0, K_{XX} \mid \mathrm{sign}\, f_X = Y\right);$$
$$\text{Sample noise:}\quad \xi \sim \mathrm{NORMAL}\left(0, K_{xx} - K_{xX}K_{XX}^{-1}K_{Xx}\right);$$
$$\text{Return:}\quad \mathrm{sign}[K_{xX}K_{XX}^{-1}f_X + \xi].$$

Then, recalling that the BPM classifier is obtained by reversing the order of sign and expectation in the Bayes classifier, the BPM classifier is given by:

$$\mathrm{sign}\,\overset{\frown}{\mathbb{E}_{\xi,f_X}\,\mathrm{sign}}[K_{xX}K_{XX}^{-1}f_X + \xi] \qquad\qquad \text{(Bayes classifier)}$$
$$\approx \mathrm{sign}\,\mathbb{E}_{\xi,f_X}[K_{xX}K_{XX}^{-1}f_X + \xi] \qquad\qquad \text{(BPM classifier)}$$
$$= \mathrm{sign}\,K_{xX}K_{XX}^{-1}\,\mathbb{E}_{f_X \sim Q_{\mathrm{GP}}}\,f_X. \qquad\qquad \text{(kernel interpolator)}$$

This completes the proof. $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\square$

And second, the BPM of the spherised Gaussian process posterior Q_{sph} is the sign of the kernel interpolator of *centroidal labels Y*:

Lemma 9.8 (BPM of a spherised Gaussian process classifier is a kernel interpolator). *For the spherised posterior Q_{sph} (Definition 8.3),*

$$f_{\mathrm{BPM}}(x) = \mathrm{sign}[K_{xX}K_{XX}^{-1}Y]. \tag{9.11}$$

Proof. The Gibbs classifier of Q_{sph} classifies a test point x in three steps:

$$\text{Sample train outputs:}\quad f_X \sim \mathrm{NORMAL}\!\left(0, \mathbf{I} \cdot |K_{XX}|^{1/m} \mid \mathrm{sign}\, f_X = Y\right);$$
$$\text{Sample noise:}\quad \xi \sim \mathrm{NORMAL}\left(0, K_{xx} - K_{xX}K_{XX}^{-1}K_{Xx}\right);$$
$$\text{Return:}\quad \mathrm{sign}[K_{xX}K_{XX}^{-1}f_X + \xi].$$

Then the BPM classifier is given by exchanging operators in the Bayes classifier:

$$\mathrm{sign}\,\overset{\frown}{\mathbb{E}_{\xi,f_X}\,\mathrm{sign}}[K_{xX}K_{XX}^{-1}f_X + \xi] \qquad\qquad\qquad \text{(Bayes classifier)}$$
$$\approx \mathrm{sign}\,\mathbb{E}_{\xi,f_X}[K_{xX}K_{XX}^{-1}f_X + \xi] \qquad\qquad\qquad \text{(BPM classifier)}$$
$$= \mathrm{sign}\,K_{xX}K_{XX}^{-1}\,\mathbb{E}_{f_X \sim Q_{\mathrm{sph}}}\,f_X = \mathrm{sign}\,K_{xX}K_{XX}^{-1}Y. \qquad \text{(kernel interpolator)}$$

This completes the proof. $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\square$

These results lead to the following PAC-Bayesian generalisation guarantees for minimum RKHS norm kernel interpolation.

Theorem 9.1 (Kernel PAC-Bayes). *Given a kernel k and a training sample $S = (X, Y)$, recall the definitions of the kernel complexity $\mathcal{A}$ (Definition 8.5) and the Gaussian orthant probability P_Y (Definition 8.4). Let $\mathcal{D}$ be a data distribution over $\mathcal{X} \times \{\pm 1\}$. For the minimum RKHS norm kernel interpolator of data sample (X, Υ) given by $f_\Upsilon(x) := K_{xX} K_{XX}^{-1} \Upsilon$, define the test error:*

$$\varepsilon[\Upsilon] := \mathbb{E}_{(x,y) \sim \mathcal{D}} \, \mathbb{I}[\text{sign } f_\Upsilon(x) \neq y].$$

Then, with probability $1 - \delta$ over a training sample $(X, Y) \overset{iid}{\sim} \mathcal{D}^m$, the following bounds hold simultaneously:

$$\varepsilon[\mathbb{E}_{f_X \sim Q_{\text{GP}}} f_X] \leq \mathrm{e} \cdot \left[1 - \exp\left(-\frac{\log 1/P_Y + \log(2m/\delta)}{m - 1}\right)\right] \tag{9.12}$$

$$\leq \mathrm{e} \cdot \left[1 - \exp\left(-\frac{\mathcal{A}(k, X, Y) + \log(2m/\delta)}{m - 1}\right)\right]; \tag{9.13}$$

$$\varepsilon[Y] \leq \mathrm{e} \cdot \left[1 - \exp\left(-\frac{\mathcal{A}(k, X, Y) + \log(2m/\delta)}{m - 1}\right)\right]. \tag{9.14}$$

Proof. First, consider the Gaussian process posterior Q_{GP} (Definition 8.2). The Gibbs error of this classifier is bounded by Inequalities 8.15 and 8.16. Observe that for all inputs $x \in \mathcal{X}$, the value $f(x) \sim Q_{\text{GP}}$ follows:

$$f(x) = (f_X, z)^\top (K_{XX}^{-1} K_{Xx}, K_{xx} - K_{xX} K_{XX}^{-1} K_{Xx}),$$

for $f_X \sim Q_{\text{GP}}$ and $z \sim \text{NORMAL}(0, 1)$. But this distribution over (f_X, z) is log-concave and supported on a convex subset of $\mathbb{R}^{m+1}$ with positive volume. Therefore, Inequalities 9.12 and 9.13 follow from Inequalities 8.15 and 8.16 by Lemmas 9.4 and 9.7. Inequality 9.14 follows by applying the same argument but switching Q_{GP} for Q_{sph}. $\qquad\square$

9.4 Max-margin neural networks as Bayes point machines

This section presents the novel argument that infinitely wide neural networks, fit to large normalised margin, are Bayes point machine classifiers. The argument works by considering what happens to the neural network–Gaussian process posterior distribution as a notion of normalised margin is taken large. In short, this posterior distribution concentrates on a minimum RKHS norm kernel interpolator, which is itself a Bayes point machine by the results of Section 9.3.

Consider a hyperspherical input space $\mathcal{X} = \sqrt{d_0} \cdot \mathbb{S}^{d_0 - 1}$ and a multilayer perceptron (Definition 2.9) with L layers and scaled relu nonlinearity $\varphi(\cdot) = \sqrt{2} \cdot \max(0, \cdot)$. For each layer $l = 1, ..., L$, consider sampling the weight matrix at that layer $W_l \in \mathbb{R}^{d_l \times d_{l-1}}$ from the distribution NORMAL$(0, \mathbf{I}/d_{l-1})$. Then, as the layer widths $d_1, ..., d_{l-1} \to \infty$, the corresponding functions are distributed:

$$f \sim \text{GP}(0, k_{\text{arccos}}), \tag{9.15}$$

where the kernel k_{arccos} is the compositional arccosine kernel of Theorem 3.3.

Now suppose that, at all layers $l = 1, ..., L$, the weight matrices are instead sampled $W_l \sim$ NORMAL$(0, \sigma^2 \cdot \mathbf{I}/d_{l-1})$ for a choice of "normalisation" $\sigma > 0$. The core observation is that the relu multilayer perceptron is homogeneous of degree L in its weight vector, meaning that $f(x; \sigma \cdot w) = \sigma^L \cdot f(x; w)$ for all inputs $x \in \mathcal{X}$. This means that, as the layer widths $d_1, ..., d_{l-1} \to \infty$, the resulting functions are distributed according to:

$$f \sim \text{GP}(0, \sigma^{2L} \cdot k_{\text{arccos}}). \tag{9.16}$$

Next, condition this distribution of functions on interpolating a training sample $(X, \gamma \cdot Y)$, where the labels Y were scaled by a "margin" $\gamma > 0$:

$$f \sim \text{GP}(0, \sigma^{2L} \cdot k_{\text{arccos}} \mid f_X = \gamma \cdot Y). \tag{9.17}$$

By Theorem 3.2, the distribution of f/γ concentrates on the kernel interpolator $x \mapsto K_{xX} K_{XX}^{-1} Y$ in the limit that the "normalised margin" γ/σ^L is sent to infinity. In this expression, K_{xX} and K_{XX} are the Gram vector and Gram matrix corresponding to the unscaled kernel k_{arccos}.

In summary: by defining a notion of *normalised margin* for the neural network–Gaussian process posterior, and taking this normalised margin to infinity, the posterior concentrates on a minimum RKHS norm kernel interpolator. This function is itself a Bayes point machine by the results of Section 9.3.

This behaviour was tested experimentally, and the results are displayed in Figure 9.1. The plots show the test accuracy of both the NNGP posterior as a function of normalised margin (Equation 9.17), and also large but finite width multilayer perceptrons trained by a variant of gradient descent as a function of Frobenius-normalised margin (Definition 2.12). Qualitatively similar behaviour was observed in both cases: the average of many functions of small normalised margin attained similar accuracy to one function of large normalised margin.
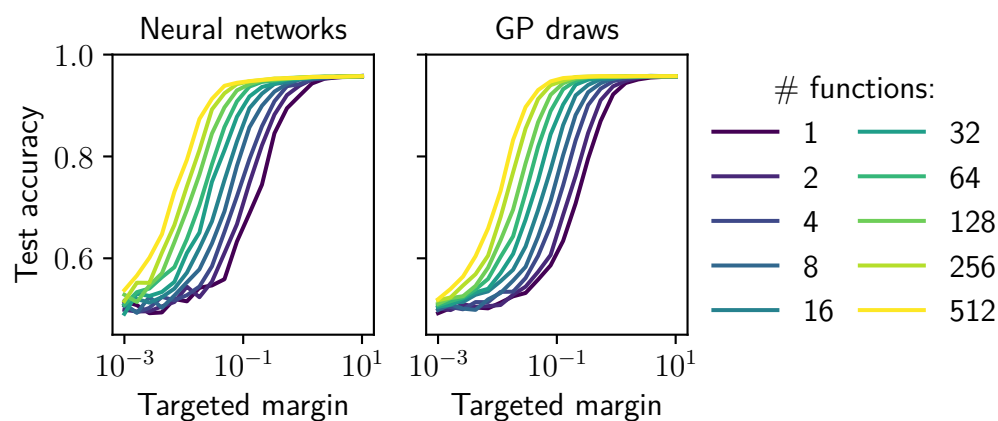
Figure 9.1: Test accuracy as a function of normalised margin, for both width-2048 neural networks (left) and neural network–Gaussian processes (right). The task was binary classification of MNIST digits (LeCun et al., 1998) using a 5-layer multilayer perceptron. Each curve shows the test accuracy of the average over a number of functions, where each function in the average attains the specified value of normalised margin. The main behaviour visible is that one function of large normalised margin appears equivalent to the average of many functions of small normalised margin. The NNGP experiments used Equation 9.17 to control normalised margin. The neural network experiments controlled Frobenius-normalised margin by minimising square loss with respect to rescaled labels, and projecting every weight matrix to a hypersphere of fixed Frobenius norm at each iteration.

9.5 Empirical comparisons between classification strategies

This section reports an experimental comparison of various binary classification strategies for both Gaussian processes and neural networks. The task was binary classification of MNIST digits (LeCun et al., 1998) using a 7-layer relu multilayer perceptron. The width was set to either 1000 or infinity (via the neural network–Gaussian process correspondence).

For neural network–Gaussian processes, the classification strategies tested were:

1) *Gibbs classifier:* the sign of a random posterior sample.

2) *Bayes classifier:* the majority vote over the posterior.

3) *BPM classifier:* the sign of the minimum RKHS norm kernel interpolator.

The spherised Gaussian process posterior (Definition 8.3) was used for reasons of computational tractability. Also, several generalisation bounds for Gaussian processes and kernel classifiers are plotted:

1) *Rademacher bound:* a uniform convergence bound for kernel classifiers (Bartlett and Mendelson, 2002, Theorem 21).

2) *Gibbs bound:* Inequality 8.17 of Theorem 8.1.

3) *BPM bound:* Inequality 9.14 of Theorem 9.1.

For finite width neural networks, the classification strategies considered were:

1) *Gibbs classifier:* train a randomly initialised network to fit the train sample to small Frobenius-normalised margin.

2) *Bayes classifier:* take the majority vote over 501 networks trained from different random initialisations to small Frobenius-normalised margin.

3) *BPM classifier:* train a randomly initialised network to fit the train sample to large Frobenius-normalised margin.

The results for Gaussian processes and kernel classifiers are presented in Figure 9.2, while the results for neural networks are presented in Figure 9.3. The results support the idea that minimum RKHS norm kernel interpolators, and large Frobenius-normalised margin neural networks, are Bayes point machines.
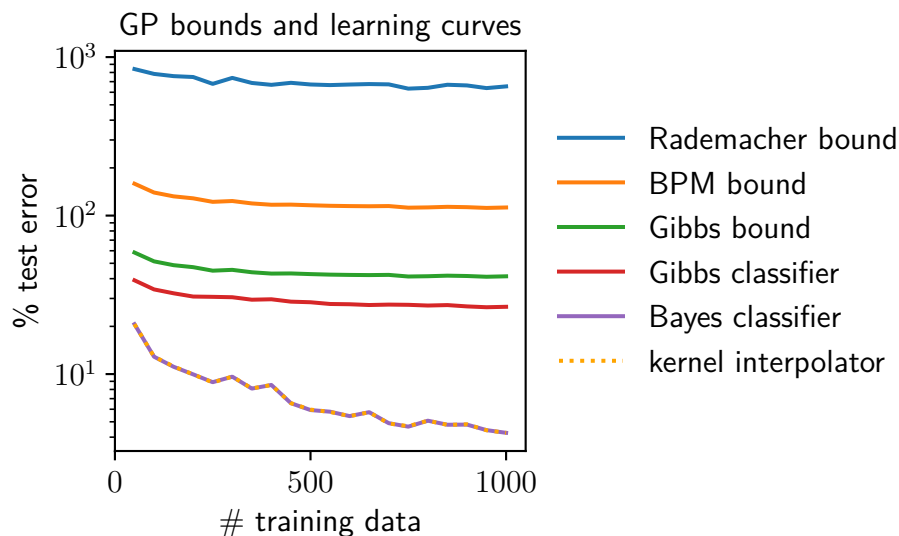
Figure 9.2: Testing classification strategies for Gaussian processes and kernels, on an MNIST (LeCun et al., 1998) binary classification task. The Bayes classifier and kernel interpolator attain indistinguishable performance, supporting the claim that minimum RKHS norm kernel interpolation is a Bayes point machine. Despite the kernel interpolator significantly outperforming the Gibbs classifier in practice, the order of the Gibbs and BPM bounds are reversed. Still, the BPM bound is substantially smaller than the Rademacher bound.
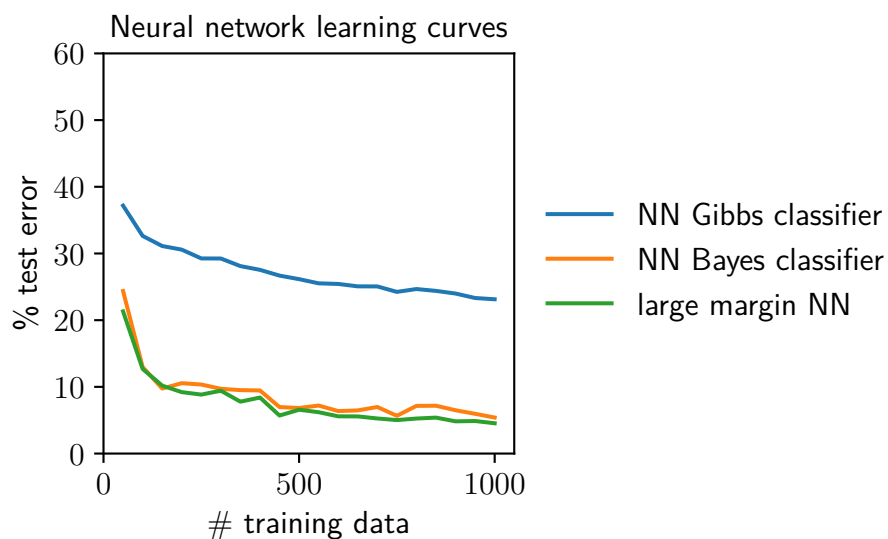

Figure 9.3: Testing classification strategies for neural networks, on the same task as Figure 9.2. The Bayes classifier reports a majority vote over 501 small margin networks. It attains similar (though not identical) performance to a single neural network of large Frobenius-normalised margin. This supports the idea that large margin neural networks are Bayes point machines. Both classifiers substantially outperform the corresponding Gibbs classifier.

References

Peter L. Bartlett and Shahar Mendelson (2002). "Rademacher and Gaussian complexities: Risk bounds and structural results". In: *Journal of Machine Learning Research.*

Andrew Caplin and Barry Nalebuff (1991). "Aggregation and social choice: A mean voter theorem". In: *Econometrica.*

Branko Grünbaum (1960). "Partitions of mass-distributions and of convex bodies by hyperplanes". In: *Pacific Journal of Mathematics.*

Ralf Herbrich (2001). *Learning Kernel Classifiers: Theory and Algorithms.* MIT Press.

Ralf Herbrich, Thore Graepel, and Colin Campbell (2001). "Bayes point machines". In: *Journal of Machine Learning Research.*

Alexandre Lacasse, François Laviolette, Mario Marchand, Pascal Germain, and Nicolas Usunier (2007). "PAC-Bayes bounds for the risk of the majority vote and the variance of the Gibbs classifier". In: *Neural Information Processing Systems.*

Yann LeCun, Corinna Cortes, and Christopher J.C. Burges (1998). *MNIST handwritten digit database.*

Matthias Seeger (2002). "PAC-Bayesian generalisation error bounds for Gaussian process classification". In: *Journal of Machine Learning Research.*

Guillermo Valle-Pérez and Ard A. Louis (2020). "Generalization bounds for deep learning". In: *arXiv:2012.04115.*