**UNIVERSITY OF SOUTHAMPTON**

Faculty of Engineering and Physical Sciences
School of Electronics and Computer Science
Vision, Learning, and Control

# Advancing differentiable program optimisation via novel first and second-order metrics, and adaptive optimisation strategies.

*by*

**Mark Tuddenham**

BSc. Hons.

ORCiD: 0000-0002-3428-4051

*A thesis for the degree of*
*Doctor of Philosophy*

March 2026

University of Southampton

Abstract

Faculty of Engineering and Physical Sciences
School of Electronics and Computer Science

Doctor of Philosophy

**Advancing differentiable program optimisation via novel first and second-order metrics, and adaptive optimisation strategies.**

by Mark Tuddenham

This thesis is on the optimisation of deep neural networks, how they behave during training, how they can be made more efficient, and how methods and ideas from classical optimisation vary when applied in a deep learning context.

The first part of the thesis is a first-order analysis of the training process of deep neural networks, with a focus on the long range structure of the learning process and how well this can be approximated by a stochastic process. We show that the learning process of deep neural networks can be reasonably well approximated by a stochastic process, and that this diffusion process can be used to understand the differences between the most popular optimisers including the artefacts from the update equations and the different convergence rates.

The second part of the thesis is on the second-order analysis of the training process of deep neural networks, with a focus on the curvature of the loss surface and how this can be used to improve the optimisation process.

In the third part we propose a new optimisation algorithm, called *orthogonalised stochastic gradient descent* (OSGD), which is based on the effect of introducing a diversification bias on the convolutional filters via orthonormalisation. And also show that the adaption from SGD to OSGD can be used to improve the convergence rate of other optimisers, including Adam and RMSProp. We show that this algorithm can be used to train deep neural networks with fewer epochs and better generalisation performance.

This work concludes with an overview of the results, a discussion of the implications of the work presented in this thesis, and some promising future directions of study.

# Contents

# List of Figures

# List of Tables

# Listings

# Declaration of Authorship

I declare that this thesis and the work presented in it is my own and has been generated by me as the result of my own original research.

I confirm that:

1. This work was done wholly or mainly while in candidature for a research degree at this University;
2. Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
3. Where I have consulted the published work of others, this is always clearly attributed;
4. Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
5. I have acknowledged all main sources of help;
6. Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
7. Parts of this work have been published as: M. Tuddenham, A. Prügel-Bennett, and J. Hare, "Quasi-newton's method in the class gradient defined high-curvature subspace," *arXiv preprint arXiv:2012.01938*, 2020, and M. Tuddenham, A. Prügel-Bennett, and J. Hare, "Orthogonalising gradients to speed up neural network optimisation," 2022. DOI: 10.48550/arxiv.2202.07052. [Online]. Available: https://arxiv.org/abs/2202.07052

Signed:..... Date:.....

## Acknowledgements

Thanks go to my supervisors, Adam Prügel-Bennett and Jonathan Hare, for their guidance and support; and the rest of the VLC group for the unrelentingly inquisitive yet relaxed atmosphere. To the University of Southampton for providing the facilities to conduct my research, especially the librarians for facilitating access to the literature and texts. And to the EPSRC for funding my PhD.

*To my parents, Karen and Andy, for their unwavering support and encouragement. None of this would have been possible without them.*

# Nomenclature

$S_{\mathbf{cos}}(\cdot,\cdot)$ Cosine similarity score $S_{\cos}(x,y) = \frac{\langle x,y\rangle}{\|x\|_2\|y\|_2}$. 80, 109, 110, 130, 131, 162

$\boldsymbol{I}$ The identity matrix. 12, 37, 38, 57, 59, 64, 68, 133, 156, 162

$\delta$ The Kronecker delta function, $\delta_{ij} = 1$ if $i = j$ and 0 otherwise. 98

$\eta$ The learning rate. 18–21, 41

$\mathcal{L}_\theta$ A loss function. 17–21, 33, 54, 58, 99

$\sigma$ An activation function. 15, 16, 161–163

$\theta$ Weight vector. ix, x, 15, 18–21, 33, 37, 39, 40, 43–47, 50, 51, 54, 56, 58, 64–66, 68–70, 80, 98, 99, 109, 114, 155, 156, 161, 162

$g$ The gradient vector of the loss function with respect to the weight vector, $g = \nabla\mathcal{L}_\theta(\cdot)$. 21, 99

# I

# Introduction

Optimisation is central to Deep Learning — it is the process by which the models go from being a random set of weights producing random, meaningless outputs to a model that can solve a problem, producing translations as good as humans [3], playing games better than humans [4], earning more money on the stock market than both humans and non–neural network based methods [5], and even diagnose medical aliments faster and more accurately than doctors [6].

Despite this, most deep learning practitioners do not think of optimisation first when they train deep learning models; instead, they focus on model architecture and data preparation and only tune the optimisation if the model fails to learn anything at all. Whilst the data, its augmentation, and the model are critical parts for performance[1], recent breakthroughs in deep learning show that problems can be solved better, and even new problems solved with new optimisers and architecture tricks designed to ease optimisation.

Since optimisation is the last step in obtaining a working model — barring advanced techniques such as ensembling and mixture of experts — it is affected by all the choices beforehand. Additionally, in most projects there is little guidance on how to improve the optimisation, and it is usually more cost-effective to improve the data or train a larger model. Optimisation methods, however, do play a vital role; the model architecture and data define what the model does, whilst optimisation dictates how well the model can do it.

Optimisation is not always ignored; tweaking the optimisation's hyper-parameters can often lead to a sizeable gain in performance, but the best results often stem from a huge investment in tuning these hyper-parameters, not least because any adjustment in the model, data, or non-optimisation hyper-parameters necessitates finding these hyper-parameters again. This becomes clear when looking at competitions for the

[1] We define performance as the model's accuracy under certain project-specific constraints, *e.g.* recall, precision, or compute limitations (training time, model size). For example, a self-driving AI might need to prioritise a low false-negative rate for stop sign detection.

fastest training times on standard problems; DAWNBench [7] has a record of two minutes thirty-eight[2] for reaching ninety-three per cent top-5 accuracy on ImageNet [8]; a problem that usually takes several days to train, with worse results. Speed training competitions prove that library-defaults are seldom ideal, both in terms of the model's architecture — simply using a large DenseNet [9] may get good accuracy, but it will also take a significant amount of time to train — and the optimisation algorithm choice with its hyper-parameters *e.g.* weight decay, learning rate, batch size, momentum, and learning rate schedule.

So what is the motivation behind understanding the optimisation better? Optimisation is important, hard to do, and not well understood. Efficient and good optimisation is important since it can lead to generally better models, better foundational models, quicker feedback on developing new ideas, more eco-friendly training due to reduced computation costs, and the ability to train on larger datasets. The optimisation of deep learning models is hard due to the curse of dimensionality, the need for many different techniques to even get a model to perform reasonably, and that many concepts in classical optimisation are not applicable to the deep learning setting. Hence, there is the possibility that a better understanding of high-dimensional error surfaces and the ways to traverse them will enable the creation of better non-convex optimisation algorithms, increasing model performance and training efficiency across many disciplines and domains.

## 1.1 Difficulties in deep learning optimisation

Optimisation can be crudely defined as the process of finding the best element from some set of available options based on some criterion. Its origin can be traced back to roughly the third century BC with the work of Euclid who wrote about finding the minimal distance between a point a line; but mainly begins during the $17^{\text{th}}$ and $18^{\text{th}}$ centuries.

Comparatively then, optimisation has been well studied for a much longer time than deep learning. Despite this, deep learning optimisation is based of the gradient descent algorithm, commonly attributed to Cauchy *et al.* [10] in 1847. We will expand on the difficulties of deep learning optimisation mentioned above in the following sections and how these make optimising deep neural networks substantially different from the problems studied in classical optimisation.

### 1.1.1 The complexity of neural networks

Deep neural networks are inherently complex systems, such that praxis often differs from theory. Firstly, the cost function is highly non-convex and often

[2] Although the competition was run from November 2017 to April 2018, this top entry is a later entry from March 2020

non-continuously differentiable, when using the popular Rectified Linear Unit (ReLU) [11], [12] or other piecewise functions. This non-convexity is a problem as it means that the optimisation problem could have many local minima, and so the optimisation algorithm could get stuck in a suboptimal solution, furthermore, there is no way of knowing if, assuming the model is at a minimum, it is at a low minimum. No matter how many times we train a model, we cannot know if we are at the global minimum, nor whether we are at a near-optimal local minimum, nor even if we are at a minimum — the lack of a feasible Hessian, see section 1.1.3, and presence of noise severely hinders our ability to judge our progress and possible outcomes. Moreover, even if we knew we were at a minimum, we would not know how many better local minimums exist or how much better they are. It is trivial to see if one set of weights is better than another but impossible to tell if there is a better parametrisation without trial and error, assuming both models have "converged".

General ablation studies are hard to perform as the components of deep learning models are so intertwined, *i.e.* no small subset of techniques can provide the majority of the performance; studies *ab initio* fail to replicate state-of-the-art results and studies ablating state-of-the-art are usually focused on one specific technique and don't look at how that technique compounds with ones already in the system. It is difficult to know what to include in any analysis of deep learning as many techniques, *e.g.* dropout, learning rate scheduling, batch normalisation, weight decay, or data augmentation, make small improvements — each seemingly non-critical — to the final performance of the model; but when they are all added up, they produce a model with excellent performance. Additionally, each one usually makes the analysis harder, so it is tempting to leave many out; however, if too many are left out, then we would not be analysing the type of system used in a real-world situation.

Due to a significant lack of a complete understanding, deep learning can sometimes be described as more of an art than a science. The myriad of ways that these components interact with each other and affect the optimisation is mostly unknown and so good results rely on the practitioner's experience.

We see here that the optimisation of deep models presents all the difficulties of classical optimisation and more.

### 1.1.2 Optimising proxies for performance and compute constraints

Computational bounds, specifically GPU or TPU memory, and the fact that datasets are huge, introduce stochasticity into the problem due to approximating the actual error function: we approximate the generalisation error by the training error and the training error by the mini-batch error. Additionally, we cannot disconnect the mathematical problem from the true problem, *i.e.* we do not want a low loss, but good performance on the task at hand. This is a problem as the mini-batch loss is somewhat

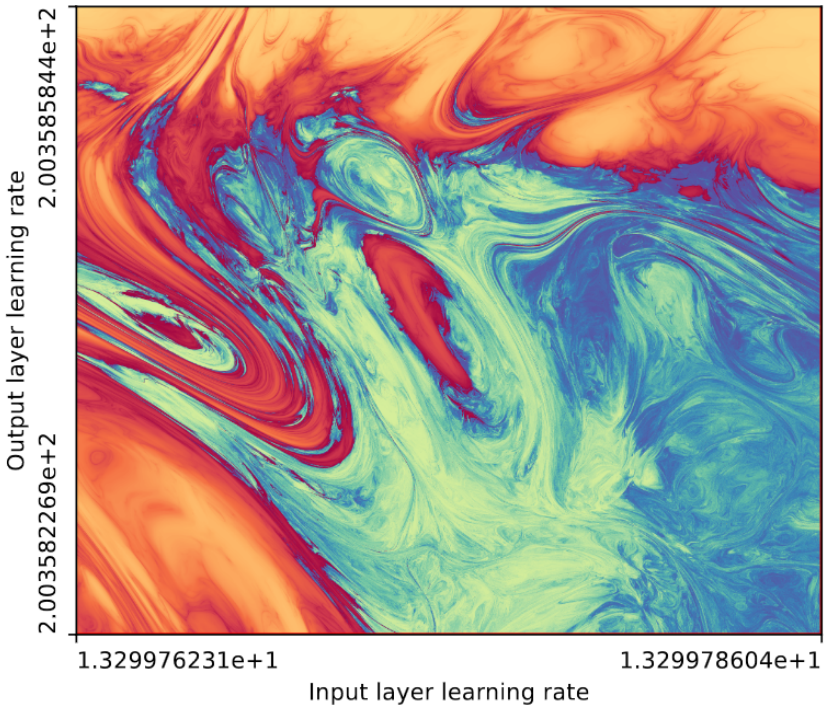



FIGURE 1.1: The boundary between convergence and divergence is fractal over hyper-parameter space. This is a grid search over neural network hyper-parameters where red points are divergent and blue convergent with the shade corresponding to the rate convergence or divergence, paler is faster. Figure taken from Sohl-Dickstein [13].

removed from the resulting performance, and so we cannot be sure that a low loss will result in good performance, it is quite probable that there exists a higher-loss model that would better solve the real world problem.

Another difference between the real world optimisation and theory is that the trillions of floating point calculations have numerical errors in them, for a smooth problem this would not be an issue, however in the very fractured landscape this difference may push the model into another partition. Sohl-Dickstein [13] show that the boundary between convergence and divergence is fractal over hyper-parameter space, see figure 1.1, and, due to the precision of the hyper-parameters, numerical errors will certainly push the model into a divergent region of the loss landscape.

### 1.1.3 The curse and boon of dimensionality

The curse of dimensionality [14], [15] is an oft-cited problem in classical optimisation that affects most optimisation problems that try to solve real-world problems with rich data streams, *i.e.* data with many features, or models with numerous parameters, but it is especially prevalent in deep learning. The curse of dimensionality is the name given to the fact that the volume of a high-dimensional space increases exponentially with the dimensionality. This means that, as the dimensionality increases, the number of samples needed to cover the space evenly also increases exponentially. This is a

problem for optimisation since there could be little to no information in the training data about a possible input. Additionally, the parameter space is extremely high dimensional, this is also a problem for optimisation as the amount of potential directions that the model could move in is larger than the amount of information given by a datum — if a huge portion of the parameter space makes little or no difference to a specific datum then it is hard for the optimiser to be able to pick the directions that helps the most overall.

The curse of dimensionality also means that the human intuition of space breaks down. A two-dimensional problem can be visualised as a relief map with the optimised parameters as the horizontal axes and the loss the terrain height, thus the common name "loss landscape"; in this view we can visualise an optimisation path as a vehicle's path through the said terrain.

At more than three dimensions, our intuition breaks down, and we end up with many counter-intuitive facts, for example, the volume of a unit sphere tends to 0 in high dimensions, but a unit cube does not [16, Chapter 1]. Therefore, it is difficult to justify decisions or give explanations for deep learning problems and solutions based on intuition alone.

Moreover, the dimensionality of the problem is so high we cannot feasibly compute the Hessian, and so second-order methods have to be discarded. It is possible to compute approximations of the Hessian or Hessian-vector products, but these are often also expensive to compute, and do not lead to big enough improvement in the training time. An additional problem with calculating the Hessian is that with ReLU-based activations the Taylor expansion is likely to be uninformative away from the expansion point.

The extreme high-dimensionality of deep learning, notwithstanding the problems it causes, is also a boon as it seems to be the reason why deep learning even works in the first place, allowing the models to generalise from very sparse data. Unfortunately, from the perspective of optimisation, it is only a curse as we do not understand how these models generalise so well and so cannot use more sophisticated optimisation methods.

### 1.1.4 That some classical optimisation theory is not applicable

Another difficulty is that some classical optimisation theory is not applicable to deep learning.

The classical notion of convergence is not applicable to deep learning, we will go into this in more detail in section 2.17, but the main point is that we never achieve the conditions for classical convergence, yet still consider the optimisation "converged" and stop iterating.

Deep learning optimisers use concepts close to classical optimisation, that are, however, different enough to make the classical theory not applicable. A prime example of this is weight decay and $L_2$ regularisation, which are used to prevent over-fitting, these are generally considered to be the same in the sense that they both reduce the norm of the model's weights; $L_2$ regularisation is the addition of the $L_2$ norm of the weights to the loss function, while weight decay is the multiplication of the weights by a decay term at each iteration. $L_2$ regularisation is the classical optimisation justification for weight decay; however they are not always the same [17], *e.g.* with adaptive methods, and so the classical theory and intuition does not necessarily hold with weight decay. Indeed, Zhang *et al.* [18] show that generally, "weight decay consistently outperform[s] $L_2$ regularization", which is nonsensical from the view of classical optimisation because there are now two processes updating the weights of the model and multiple objective learning is a significantly more difficult problem.

## 1.2 This work

While some optimisation methods are directly derived from classical works, *e.g.* momentum [19] and Nesterov's momentum [20], anything more involved, for example variations of Newton's method, fail due to the extremely high dimensions or from the problem's dearth of desirable properties. In addition, some methods were developed when neural networks were massively smaller and now that models are huge they are not as relevant; today Nesterov's momentum is not commonly used, it does not provide better results and is unstable more often than the basic momentum. To make practical improvement, the machine learning and optimisation community have designed other optimisation methods specifically for deep learning [21], but these often do not draw on classical optimisation above and beyond gradient descent and momentum.

We know second-order methods are impossible, but there exist derivative-free methods such as Bayesian optimisation, coordinate descent, grid search, and genetic algorithms as well. However, the first three of these suffer from the curse of dimensionality, while the latter often suffers from the network's coadaptation, where changing one node renders others ineffective. This is not to say that they never work; for instance, Such *et al.* [22] used genetic algorithms to train a reinforcement learning agent on Atari games, just that they are not easily generalisable algorithms. Because of this, this work will focus on gradient-based methods as they are the most widely used and practical.

In this light, we want to create a better understanding of high-dimensional error surfaces that will enable the creation of better non-convex optimisation algorithms

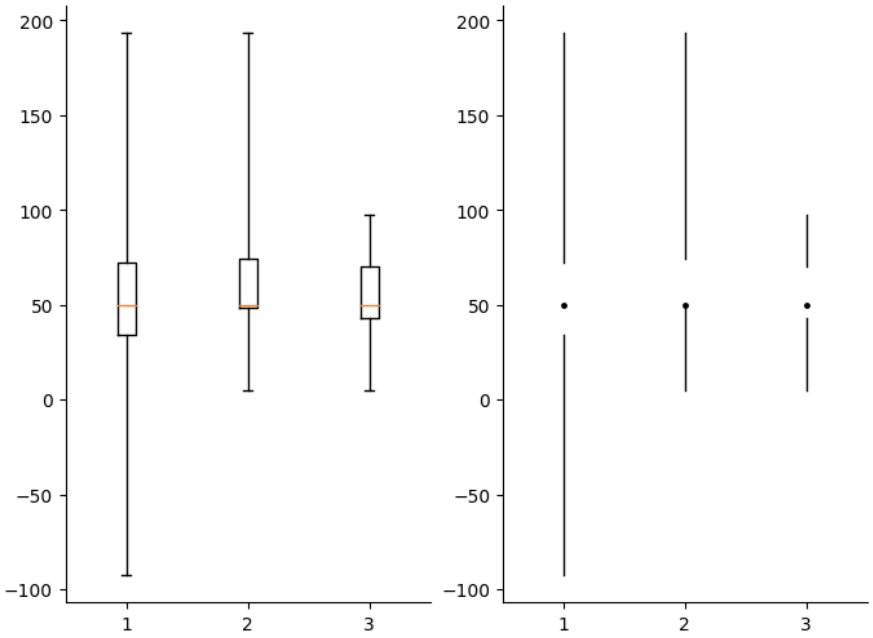



FIGURE 1.2: Comparison of Tukey's [23] standard box plot (left) and Tufte's [24] box plot used in this thesis (right) where it can be seen that Tufte's takes up less space while still remaining clear due to the removal of useless "data ink"[24]. Generated with listing B.1.

designed for neural networks, increasing model performance and training efficiency across many disciplines.

### 1.2.1 A note on figures

All figures in this thesis are created by the author unless otherwise stated. We use a compact variation of Tukey's [23] standard box plot designed by Edward Tufte [24, p. 124-5] to show the median, quartiles, and extrema of the data. A comparison of these styles is shown in figure 1.2 in case the reader has not seen Tufte's style before. We will see that this format is an improvement when there are many box plots on one chart axis.

### 1.2.2 How deep is deep learning?

In deep learning, we have used the word "deep" to describe more and more complex models over time. For example, three layers and one hundred parameters was considered a deep network in 1996 [25], in 2012, AlexNet [26] was released with eight layers and over sixty million parameters. Now, in 2023, we have models fifty layers deep and over one hundred and fifty billion parameters [27].

For this work, we need to strike a balance between feasibility and applicability: while Radford *et al.* [28]'s 1.5 billion parameter behemoth might be the go-to for text generation, training it hundreds or thousands of times to analyse its behaviour is

infeasible. However, analysing Babri and Tong [25]'s 175 parameter model will not yield many utilisable results as the small capacity of the model will not solve many problems for which people are now using deep learning. Thus, the models looked at in this work will have over ten layers and have tens to hundreds of millions of parameters. In terms of problems, we will be mainly concentrating on the well-known image classification problems ImageNet [8] and CIFAR-10 [29], using residual networks [30].

## 1.3 Research Goals

In this thesis we address the following research goals.

- Develop tools to analyse the paths taken by optimisers in high-dimensional spaces.
- Create metrics that can be used by new optimisers to improve their performance.
- Analysing the optimisation process of deep neural networks.
- Improving existing optimisation algorithms or develop a new optimisation algorithm.

## 1.4 Thesis structure

We conclude this preface with an overview of the thesis structure.

We begin in Chapter 2 by outlining the necessary background in both deep learning and optimisation. This chapter introduces key mathematical concepts such as convexity, smoothness, and Lipschitz continuity, which are central to understanding optimisation theory in classical settings. We then introduce the standard tools of deep learning—architectures like fully-connected and convolutional networks, regularisation techniques like batch normalisation, and core optimisation methods such as stochastic gradient descent (SGD), momentum, and adaptive optimisers including Adam and AdaGrad. We also review learning rate scheduling and survey convergence theory as it applies (or fails to apply) in modern deep learning practice. This sets the stage for understanding how and why classical optimisation tools break down in the high-dimensional, non-convex, and often chaotic landscapes of deep neural networks. This chapter is all literature review.

In Chapter 3, we shift from theoretical foundations to empirical analysis. We begin by critically examining earlier approaches to understanding neural network training, including visualisation techniques and randomised baselines. We argue that many of these are insufficiently sensitive to the geometry of optimisation, particularly in how

they represent the trajectory of the model through weight space. In response, we introduce new tools for quantifying "directedness" in optimisation—how efficiently the model's updates contribute to actual displacement in weight space. We define and develop a family of metrics rooted in geometric interpretations of the learning path, including cumulative path length, displacement, and their ratio over time. These are used to study a range of hypotheses about the nature of deep learning optimisation, including whether it resembles a random walk or an autoregressive process. In doing so, we offer a new lens for examining the temporal structure of training dynamics.

Chapter 4 builds on this by developing second-order tools for characterising the loss landscape and the behaviour of the optimiser. Since computing the full Hessian of a deep neural network is typically infeasible, we turn to scalable approximations such as Hessian-vector products, power iteration, and diagonal preconditioning. We discuss quasi-Newton methods like BFGS and SR1, and examine how well their assumptions hold in the deep learning regime. We also develop efficient methods to estimate dominant curvature directions via the Lanczos algorithm and study their consistency over time. This second-order information is integrated with our earlier directedness analysis, revealing how curvature structure interacts with optimisation dynamics throughout training. In particular, we show how the Hessian spectrum evolves during learning and how this can be exploited to better understand generalisation, flatness, and robustness. Our main contributions in this chapter are section 4.3 where we combine statements from the literature and show that the various assumptions in these, in aggregate, do not hold up; and section 4.4 where provide an efficient method for calculation an approximation to the top eigenvector of the Hessian of a deep neural network.

In Chapter 5, we introduce a novel optimiser modification based on gradient self-orthogonalisation. This method adapts the optimiser's updates to encourage diversity in the direction of successive gradients. We show how this idea can be realised via several mechanisms: soft regularisation, hard reparameterisation, and manifold-based projection. We situate this approach within a broader literature on orthogonality and multitask learning, and provide empirical results showing consistent improvements across a range of tasks, including CIFAR-10 image classification and Poisson variational autoencoders. Importantly, we also provide a careful analysis of when and why orthogonalisation helps, and when it may conflict with small-batch generalisation or limited curvature. This chapter, apart from the related work section is our second major contribution in this work.

Finally, Chapter 6 concludes the thesis by summarising the key contributions and insights, and suggesting directions for further research. These include extensions of the directedness framework to other domains, incorporation of higher-order approximations into lightweight optimisers, and explorations of the interplay between learning dynamics and model architecture.

# II

# A background of optimisation and deep networks

## 2.1 Optimisation problems

In optimisation we desire to minimise some function, $f$,

$$\min_x f(x), \tag{2.1}$$

where $x \in \mathbb{R}^n$ and $f : \mathbb{R}^n \to \mathbb{R}$. Since knowing everything about $f$ beforehand is often infeasible, we limit our algorithm to only knowing $f(x)$ at certain values of $x$ — which it gets to choose, called an evaluation of $f$ at $x$. An optimisation algorithm thus uses evaluated points to find iteratively better points. Since evaluations are not cheap in computation nor memory, we would like our algorithm to be efficient and use the least number of evaluations.

A point $x^\star$ is called a global minimum, or global optimum, when $f(x^\star) \leq f(x)$ for all $x$. Given that we only have local information, *i.e.* only at the points we have evaluated, of which we want there to be few, it is often overly difficult to tell if we are at a global minimum. Instead, if we can define a neighbourhood, $\mathcal{N}$, around a point $\hat{x}$, such that $f(\hat{x}) \leq f(x)$ for all $x \in \mathcal{N}$ then we call $\hat{x}$ a local minimum, or local optimum. While we are often content finding local minima instead of global minima, we obviously want minima where $f(x)$ is low, if there exist local minima with high evaluations then we can become trapped in these minima, failing to find the more desirable ones.

There are conditions for recognising if we are in a local or global minimum, however, as these involve calculating the Hessian, it is often infeasible to do so in a deep learning setting. If we have a good optimisation algorithm then we should end up sufficiently close to a minimum, we shall call this point the solution.

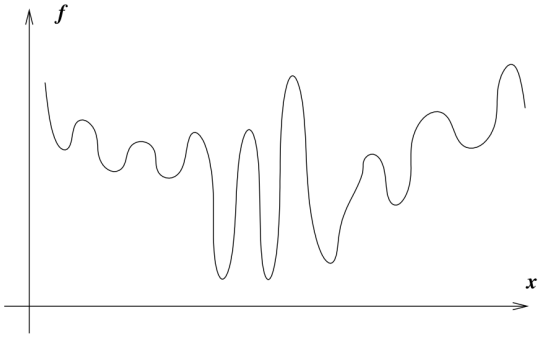



FIGURE 2.1: A function with many local minima where an optimisation may get trapped; taken from Nocedal and Wright [31]

## 2.2 Convexity

If $f$ is convex then all local minima are global minima, making convexity a desirable property since our optimisation algorithm cannot become stuck in local minima or at saddle points with a high value. Even though most problems in machine learning are non-convex [32], it is still useful to know since it can be used to prove optimisation methods work on simpler problems or can be used as an assumption that might lead to an algorithm that works on non-convex problems too.

A function, $f$, is convex if, for every pair of points, $x_1$ and $x_2$, and for every $\alpha \in [0, 1]$,

$$f(\alpha x_1 + (1 - \alpha)x_2) \leq \alpha f(x_1) + (1 - \alpha)f(x_2). \tag{2.2}$$

A function $f$ is strictly convex if the inequality is strict, *i.e.*

Since deep learning usually uses differentiable functions an equivalent definition of convexity for if the function is continuously differentiable can be more useful,

$$f(y) \geq f(x) + \langle \nabla f(x), y - x \rangle , \tag{2.3}$$

where $\langle \cdot, \cdot \rangle$ is the inner product and $\nabla f(x)$ is the gradient of $f$ at $x$. If the function is twice differentiable then it is convex if and only if the Hessian is positive semi-definite, *i.e.* $\nabla^2 f(x) \succeq 0$.

Additionally, a function is strongly convex if there exists a positive constant, $\alpha$, such that

$$f(y) \geq f(x) + \langle \nabla f(x), y - x \rangle + \frac{\alpha}{2}\|x - y\|_2^2. \tag{2.4}$$

Or, equivalently for twice differentiable functions, the Hessian is positive definite, *i.e.* $\nabla^2 f(x) \succeq m\boldsymbol{I} \succ 0$ for $m > 0$.

| Class | Properties | Example |
|---|---|---|
| $C^{-1}$ | Discontinuous | Heaviside step |
| $C^0$ | Continuous | $\lvert x \rvert$ |
| $C^1$ | Continuously differentiable | $x^2$ |
| $C^k$ | $k$-times continuously differentiable | $\lvert x \rvert^{k+1}$ |
| $C^\infty$ | Infinitely differentiable | $\exp(x)$ |

TABLE 2.1: Differentiability classes, their properties, and examples of functions in each class.

## 2.3 Smoothness

The smoothness of a function is measured by its differentiability class, $C^k$, where $k$ is the number of times the function is continuously differentiable. A function, $f$, is of class $C^k$ if its $k^{\text{th}}$ derivative is continuous. Generally, a smooth function means a function of class $C^\infty$ which means it is infinitely differentiable, *i.e.* it has derivatives of all orders. Table 2.1 shows the common differentiability classes and their properties.

## 2.4 Lipschitz

A similar property to smoothness is Lipschitz continuity, which is a bound on the growth of a function. A function, $f$, is $L$-Lipschitz continuous if

$$|f(x) - f(y)| \leq L\|x - y\|_2. \tag{2.5}$$

If the function is differentiable then it is Lipschitz continuous if and only if the gradient is bounded, *i.e.* $\|\nabla f(x)\|_2 \leq L$. Lipschitz continuity places a bound on a linear perturbation of the function, whereas strong smoothness places a bound on a quadratic perturbation of the function — $\|x - y\|_2$ versus $\|x - y\|_2^2$.

## 2.5 Line search vs trust region

An optimisation algorithm begins at an initial point $x_0$ and generates a sequence of iterates, $\{x_k\}_{k=0}^N$. The algorithm decides what point to evaluate next by looking at information about the current iterate and often from earlier iterates as well.

When calculating the next iterate there are two things to decide: the direction to step in, and the distance to step. A line search method first finds the direction and then searches for the minimum in that direction,

$$\min_{\alpha > 0} f(x_k + \alpha p_k). \tag{2.6}$$

In contrast, a trust region method first fixes the maximum distance, $\Delta$ called the radius, and then searches for the direction using some approximation, $\hat{f}$, of $f$,

$$\min_{p_k} \hat{f}(x_k + p_k), \tag{2.7}$$

where $\|p_k\| < \Delta$. $\hat{f}$ is designed to be easier to evaluate than $f$ as it only needs to approximate $f$ in the trust region. If we fail to optimise $\hat{f}$ it is likely our approximation is not good enough in the trust region, and so we reduce the trust region radius.

## 2.6 Convergence

Let $\{x_k\}_{k=0}^N$ be the sequence of iterates generated by an optimisation algorithm. The algorithm is said to converge to a point, $x^\star$, if the distance between the subsequent iterates and the solution goes to zero,

$$\lim_{k\to\infty} \|x_k - x^\star\| = 0. \tag{2.8}$$

If equation (2.8) holds, then the step length, $\|x_k - x_{k-1}\|$, must also go to zero. For gradient methods the step length is proportional to the gradient, $\|x_k - x_{k-1}\| \propto \|\nabla f(x_k)\|$ and so a gradient based algorithm can only converge to minima. We argue, in section 2.17, that in deep learning, the step length does not go to zero, and so the algorithm does not converge in the classical sense.

## 2.7 Stopping conditions

Now we have ways of finding the iterates, we need to define the criterion for stopping and deciding on a solution, Obviously, if the algorithm is monotonically decreasing, or we can calculate the convergence conditions — section 2.6, then we only need stop when we reach a minimum. If, however, the algorithm is non-monotonic then there are two main stopping conditions. Firstly, if the iterates do not reduce after a predetermined, $m$, number of steps, *i.e.* stop when

$$f(x_k) > f(x_{k-m}). \tag{2.9}$$

Secondly, simply fixing the total number of iterates, $N$, and taking the minimum.

$$\min_k \{x_k\}_{k=0}^N. \tag{2.10}$$

## 2.8 Steepest descent

The most obvious direction to use for line search is the direction of steepest descent, $-\nabla f(x_k)$. An advantage of this direction is that it only requires calculating the first derivative and not the second, however, any direction that makes an angle of less than $\frac{\pi}{2}$ radians with $-\nabla f(x_k)$ is certain to result in a decrease in $f$, given the step length is sufficiently small.

## 2.9 Deep Networks

Deep networks, at their most abstract, are simply a parametrised function $f : \mathbb{R}^p \to \mathbb{R}^q$ such that $m$ is made of multiple stacked parametrised functions,

$$f(x) = m_l(...m_1(m_0(x))), \tag{2.11}$$

here we have $l$ functions or layers, usually of order 10s or 100s; it is this stacked function composition that gives rise to the name deep network. We will denote the whole network $f$ to have parameters $\theta$ and each layer, $m_i$, to be parametrised by $\theta_i$.

While network architectures can become complex for some tasks, in this work we mostly focus on image classification with convolutional based architectures.
See MacKay [33, Chapter V] for the history and biological inspiration behind neural networks.

Below we will describe the most common layers used in convolutional deep networks.

### 2.9.1 Fully-connected layers

A fully-connected layer, in the biological analogy, is a layer where every neuron is connected to every neuron in the previous layer. Mathematically, a fully-connected layer is a linear transformation followed by a non-linear activation function,

$$\text{FC}(x) = \sigma(Wx + b), \tag{2.12}$$

where $\sigma$ is some non-linear activation function, commonly a ReLU.

### 2.9.2 Convolutional layers

Convolutional layers are inspired by the visual cortex, where neurons have a small receptive field and are connected to the neurons in the previous layer that are close to each other. First used in deep learning by LeCun *et al.* [34] for document classification, convolutional layers have been shown to be very effective at image classification tasks.

A convolutional layer is made up of a set of filters, each filter is a small $c \times c$ matrix, $W$, that is convolved across the input volume, $x$, producing an activation map, $a$,

$$a_{i,j} = \sum_{m=0}^{c-1} \sum_{n=0}^{c-1} x_{(i+m),(j+n)} W_{m,n}. \tag{2.13}$$

The activation map is then passed through a non-linear activation function, $\sigma$,

$$\text{conv}(x) = \sigma(a). \tag{2.14}$$

Ma and Lu [35] have shown that fully-connected layers can be transformed into convolutional layers, and *vice versa*, without loss of representational power. This means that the two types of layers are equivalent in terms of the functions they can represent, and so the choice of layer is more about the computational efficiency or output format of the layer.

### 2.9.3 Residual layers

He *et al.* [36] introduced the residual layer, which is a layer that adds the input of the layer whatever computation the layer does. For example a residual fully-connected layer is,

$$\text{residual}(x) = x + \sigma(Wx + b). \tag{2.15}$$

This is done to help with the vanishing gradient problem, where the gradient of the loss with respect to the input of the layer is very small or in the case of ReLU can be zero, and so the weights of the layer are barely changed or not updated at all. Additionally, it has been hypothesised that the residual layer prevents permutation symmetries as the residual function is not permutation symmetric while the linear or convolutional part is.

### 2.9.4 Batch Normalisation

Ioffe and Szegedy [37] introduced the batch normalisation layer, which is a layer that normalises its input to have zero mean and unit variance. This is done to help with the vanishing gradient problem and internal covariate shift — "the change in the distribution of network activations due to the change in network parameters during training" [37], since the input to the layer is normalised the gradient of the loss with respect to the input is not as likely to be very small, and the subsequent layer does not have to try and compensate for the change in distribution of the previous layers. Additionally, it has been shown to act as a regulariser, since the normalisation is done over the mini-batch, the layer is less likely to over-fit to the training data.

Batch normalisation is, in practice, added as another layer in the network before the one it is normalising for. Given a mini-batch of data each having $d$ dimensions,

$x = [x_0, x_1, \ldots, x_d]$, it is implemented by,

$$\mathrm{BN}(x)_i = \gamma \frac{x_i - \mathbb{E}[x_i]}{\sqrt{\mathrm{Var}[x_i] + \epsilon}} + \beta, \tag{2.16}$$

where $\gamma$ and $\beta$ are learnable parameters, $\mathbb{E}[x_i]$ is the mean of dimension $i$ over the mini-batch, $\mathrm{Var}[x_i]$ is the variance of dimension $i$ over the mini-batch, and $\epsilon$ is a small constant to prevent division by zero.

Interestingly, this layer also changes the optimisation behaviour by mixing the information between datums. That is, because the normalisation is done over the mini-batch, the output of a datum depends on the other data in the mini-batch.

### 2.9.5 Pooling layers

A pooling function calculated a summary statistic of the inputs nearby to the output location. The maximum pooling [38] layer summarises the maximum value within a rectangular window.

$$\max \mathrm{pool}(x)_{i,j} = \max_{-a \leq m \leq a, -b \leq n \leq b} x_{(i+m),(j+n)}, \tag{2.17}$$

where $a$ and $b$ are the height and width of the pooling window. Other examples of pooling functions are: the minimum, average, or $L_2$ norm of a rectangular window, or a weighted average of the distance from the central pixel. This summarisation makes the resulting representation invariant to small translations of the input, which can be a useful property if we care more about the existence of a feature in the input than the exact location of that feature.

## 2.10 Loss functions

Consider a classification problem with data, $\mathcal{D} = \left\{ (x^\mu, y^\mu) \,\middle|\, \mu \in [1, N] \right\}$ where $y^\mu$ is a one hot vector.

How do we form the function $f$ from equation (2.1) given that we have a model and some data such that we can use our optimisation algorithms to turn the model into a good function approximator?

We construct a function, $\mathcal{L}_\theta(x, y)$, called a loss function, that takes as inputs the data and gives a larger value the further away the model's output is from the ground truth vectors, by some criterion. For instance, and most commonly for image classification, the cross entropy loss [39],

$$\mathcal{L}_\theta(x^\mu, y^\mu) = -\sum_i y_i^\mu log(m(x^\mu)_i), \tag{2.18}$$

which measures the distance between the desired probability distribution, $y^\mu$, and the model's outputted distribution, $m(x^\mu)$.

Then, since our task has multiple points we sum the loss for every datum,

$$\mathcal{L}_\theta(\mathcal{D}) = -\sum_{(x,y)\in\mathcal{D}}\sum_i y_i log(m(x)_i). \tag{2.19}$$

Now we have a function whose minima results in a well performing model, to obtain this model we simply have to minimise this loss function with respect to the parameters

$$\underset{\theta}{\operatorname{argmin}}\, \mathcal{L}_\theta(\mathcal{D}). \tag{2.20}$$

## 2.11 Gradient Descent

There are many ways to choose the step length in a line search method, all involve a trade-off between the accuracy of the step length and the number of evaluations required to find it [31, Chapter 3]. We will not go into these methods, since, in deep learning, the evaluation of the model is assumed to be such an expensive operation that the optimal trade-off is doing no evaluations and having a step length proportional to the calculated update vector; this proportion is called the learning rate. In some ways, this is neither a line search nor a trust region method, since both step length and direction are fixed. Note this does not mean each step is the same length since we do not require the step direction, $p_k$ in equation (2.6), to be normal.

The update equation to get the next iterate using Gradient Descent (GD) is

$$\theta_{t+1} = \theta_t - \eta\nabla\mathcal{L}_\theta(\mathcal{D}). \tag{2.21}$$

See Bishop [39, Chapter 5] for more on how to obtain the gradient information of a deep network via backpropagation.

## 2.12 Stochastic Gradient Descent

Due to compute limitations it is often infeasible to compute the loss over every datum, so we approximate the total loss via the loss over a small subset of the data, know as a mini-batch

$$\theta_{t+1} = \theta_t - \eta\nabla\mathcal{L}_\theta(\mathcal{B}), \tag{2.22}$$

where $\mathcal{B} \subset \mathcal{D}$. Since this the mini-batch is sample randomly from $\mathcal{D}$, the mini-batch loss, $\mathcal{L}_\theta(\mathcal{B})$, is an approximation of the full loss, $\mathcal{L}_\theta\mathcal{D}$, and so it can be thought of as the full loss plus some noise $\mathcal{L}_\theta(\mathcal{B}) = \mathcal{L}_\theta(\mathcal{D}) + \epsilon$, hence this method of using

mini-batches is known as Stochastic Gradient Descent (SGD). The size of this noise depends on the size of the mini-batch, and how well the mini-batch represents the full dataset. The equivalent of a full gradient descent step is called an epoch, which is when the whole dataset has been seen once.

## 2.13 Momentum

Momentum, first introduced by Polyak [19] as a heavy ball method, follows the intuition that if the model has some intrinsic momentum as it rolls down the loss-landscape it will be less affected by the bumps, that is, adding a bias towards directions that have been shown to be beneficial reduces the noise of the optimised function.

To implement momentum we imagine the model has some velocity, $v_t$, and that the gradient acts on this velocity as acceleration

$$v_{t+1} = \gamma v_t + \nabla \mathcal{L}_\theta(\cdot), \tag{2.23}$$

analogously, the decay term, $\gamma$, can be thought of as friction, or some resistive force. Since displacement is velocity over time, we can formulate a version of SGD with momentum by choosing some time interval, $\eta$,

$$\theta_{t+1} = \theta_t - \eta v_t. \tag{2.24}$$

Stochastic Gradient Descent with Momentum (SGDM) can also be formulated as

$$v_{t+1} = \gamma v_t + \eta \nabla \mathcal{L}_\theta(\cdot) \tag{2.25}$$

$$\theta_{t+1} = \theta_t - v_t. \tag{2.26}$$

Figure 2.2 demonstrates one of the benefits of momentum, in poorly scaled problems the gradient is nearly perpendicular to the desired direction of travel because one of the dimensions has a much larger gradient; with SGD, the model will either bounce back and forth over the valley, unable to reach the minimum if the step size is too large; or, if the step size is small, it will rapidly fall to the bottom of the valley and then crawl slowly towards the minimum. Clearly, neither behaviour is desired when creating an efficient optimisation algorithm. With momentum, however, the anticorrelated directions in the gradient will cancel each other out in the velocity vector, while the correlated directions will stack together. This combination of constructive and destructive interference of the gradients means that the problem has effectively been rescaled.

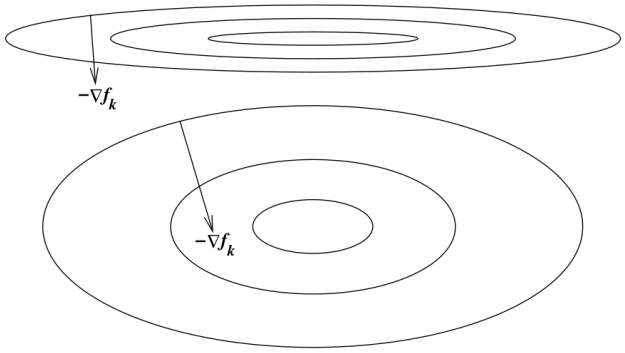



FIGURE 2.2: A comparison of poorly and well scaled minima shows the benefit of momentum; taken from Nocedal and Wright [31]

INPUT: $\eta > 0$, $\delta \geq 0$
VARIABLES: $s \in \mathbb{R}^d, H \in \mathbb{R}^{d\times d}$, $g_{1:t,i} \in \mathbb{R}^t$ for $i \in \{1,\ldots,d\}$
INITIALIZE $x_1 = 0$, $g_{1:0} = []$
FOR $t = 1$ to $T$
Suffer loss $f_t(x_t)$
Receive subgradient $g_t \in \partial f_t(x_t)$ of $f_t$ at $x_t$
UPDATE $g_{1:t} = [g_{1:t-1}\ g_t]$, $s_{t,i} = \|g_{1:t,i}\|_2$
SET $H_t = \delta I + \mathrm{diag}(s_t)$, $\psi_t(x) = \frac{1}{2}\langle x, H_t x\rangle$

Primal-Dual Subgradient Update (3):

$$x_{t+1} = \operatorname*{argmin}_{x\in\mathcal{X}} \left\{ \eta \left\langle \frac{1}{t}\sum_{\tau=1}^{t} g_\tau, x \right\rangle + \eta\varphi(x) + \frac{1}{t}\psi_t(x) \right\}.$$

Composite Mirror Descent Update (4):

$$x_{t+1} = \operatorname*{argmin}_{x\in\mathcal{X}} \left\{ \eta\langle g_t, x\rangle + \eta\varphi(x) + B_{\psi_t}(x, x_t) \right\}.$$

FIGURE 2.3: AdaGrad; taken from Duchi *et al.* [41]

In 1983 Nesterov [20] introduced an accelerated gradient method, which has provably better bounds for convex functions, $O(\tau^{-2})$ against $O(\tau^{-1})$ for GD, and can be written in the form of a momentum based algorithm [40],

$$v_{t+1} = \gamma v_t + \eta \nabla \mathcal{L}_{\theta+\gamma v_t} \tag{2.27}$$

$$\theta_{t+1} = \theta_t - v_t. \tag{2.28}$$

While the bounds are proven for the convex case, Nesterov's momentum is still effective in training deep networks.

## 2.14 Adaptive methods

### 2.14.1 AdaGrad

Duchi *et al.* [41] present a family of adaptive subgradient methods, called AdaGrad, that use an approximate Hessian to adapt the gradients to the problem's curvature, similar to the scaling mentioned above but with estimated second-order information;

intuitively, AdaGrad dynamically adapts the learning rate for each parameter. Importantly, AdaGrad does not require a smooth function or a positive definite Hessian as other similar methods do [42].

### 2.14.2 Adam

Kingma and Ba [21] proposed an adaptive method, called Adam — shown in algorithm 1, that has bias corrected momentum in addition to second moment estimation.

**Algorithm 1** Adam

**Require:** $\eta$ ▷ learning rate
**Require:** $\beta_1$ ▷ exponential decay rate for the first moment estimates
**Require:** $\beta_2$ ▷ exponential decay rate for the second moment estimates
**Require:** $\epsilon$ ▷ small constant to prevent division by zero
  $m_0 \leftarrow 0$
  $v_0 \leftarrow 0$
  **for** $t = 0$ to $T$ **do**
    $g_t \leftarrow \nabla \mathcal{L}_\theta(\cdot)$
    $m_{t+1} \leftarrow \beta_1 m_t + (1 - \beta_1) g_t$ ▷ update biased first moment estimate
    $v_{t+1} \leftarrow \beta_2 v_t + (1 - \beta_2) g_t^2$ ▷ update biased second raw moment estimate
    $\hat{m}_{t+1} \leftarrow \frac{m_{t+1}}{1-\beta_1^{t+1}}$ ▷ compute bias-corrected first moment estimate
    $\hat{v}_{t+1} \leftarrow \frac{v_{t+1}}{1-\beta_2^{t+1}}$ ▷ compute bias-corrected second raw moment estimate
    $\theta_{t+1} \leftarrow \theta_t - \eta \frac{\hat{m}_{t+1}}{\sqrt{\hat{v}_{t+1}} + \epsilon}$ ▷ update parameters
  **end**

Adam has been shown to work well on many tasks and is now considered the default optimiser since its adaption removes some of the need for hyper-parameter tuning. Convergence analysis of the Adam optimiser has been the subject of considerable scrutiny, as its adaptive moment-based update rule complicates traditional assumptions used in stochastic optimisation theory. While Adam was originally proposed by Kingma and Ba [21] with empirical success, it was later shown that the original algorithm can fail to converge even in simple convex settings [43]. This prompted the development of corrected variants such as AMSGrad, which introduces a non-decreasing second-moment accumulator to ensure convergence guarantees under convexity assumptions. Zou *et al.* [44], Wilson *et al.* [45], and Chen *et al.* [46] subsequently provided refined analyses showing that Adam can converge when certain conditions on step sizes and moment parameters are imposed—typically requiring diminishing learning rates or bounding the adaptive learning rates. For the non-convex setting, Zhou *et al.* [47] and Chen *et al.* [48] derived convergence to first-order stationary points under smoothness assumptions and bounded gradients. A general framework that unifies the analysis of Adam-type methods was introduced by Zaheer *et al.* [49], and further expanded by Défossez *et al.* [50], who proved

convergence with constant step sizes in non-convex scenarios given gradient noise control. Despite this progress, establishing tight and general convergence rates for Adam in practice remains an open problem, especially when batch noise, data sparsity, and non-smooth objectives are involved.

## 2.15 Learning rate scheduling

If we are optimising the function shown in figure 2.1 then we can see that if our step size is too big we will not be able to reach a minimum, since we will jump out of the low valued regions as soon as we jump into them due to the high gradient. To reach a good minimum, we might first have a high learning rate to skip over the high-valued minima on the outside of the function, and then lower our step size such that we are able to fall to the bottom of a minimum. This annealing of the step size allows the optimiser to traverse a highly non-convex function while still ending in a minimum.

This method, unfortunately, will pick a random minimum to settle in, if we want to explore all the minima then we could cycle the learning rate so that we keep jumping out of one minimum when the learning rate is high, then settling to the bottom of the local minimum we happen to be in when the step size is lowered. Recall that we can persist the parametrisations and value of the minima and choose the best one at the end.

There are many learning rate schedules used, see Schmidt *et al.* [51, Table 3] for an overview, however they tend to be more important for SGD based methods, since Adam intrinsically provides a form of step size annealing [21].

Inspired by learning rate decay schedules, Chen *et al.* [52] introduce the idea of momentum decay scheduling; this method decreases the momentum decay term, $\gamma$ in equation (2.26), so that the last steps of SGDM are nearly equivalent to SGD. They also use this momentum decay schedule with Adam to good results across a number of tasks. Confusingly, both the constant $\gamma$ and the process of reducing $\gamma$ are called momentum decay, since one decays the momentum vector, and the other decays $\gamma$ itself.

## 2.16 Deep Learning convergence in practice

Recent theoretical advancements have significantly enhanced our understanding of convergence guarantees in deep learning optimisation, addressing the limitations of earlier models and providing insights into the behaviour of neural networks during training. Traditional convergence analyses often relied on the Neural Tangent Kernel (NTK) framework, which assumes infinitely wide networks where the NTK remains constant during training. While this model offers valuable insights, its practical applicability is limited due to the unrealistic assumption of infinite width. Recent

research has aimed to extend convergence guarantees to more practical, finite-width networks.

[53] introduced an algorithm with global convergence guarantees under a verifiable "expressivity condition", applicable to practical deep networks beyond the NTK regime. This condition has been theoretically validated for fully-connected networks with specific architectures and empirically demonstrated for deep convolutional ResNets across various datasets.

Deep learning models often involve non-convex loss landscapes, posing challenges for traditional optimisation methods. Recent studies have focused on developing algorithms that ensure convergence in these complex landscapes. For instance, the Stochastic Gradient Descent Ascent (SGDA) [54] and Stochastic Extragradient methods (SEG) [55] have garnered significant interest for their efficiency in solving large-scale min-max optimisation and variational inequality problems prevalent in machine learning tasks.

Adaptive gradient methods like AdaGrad and Adam have been widely used for their ability to adjust learning rates dynamically. However, their convergence properties, especially in non-convex settings, have been a subject of ongoing research. Recent work [56] has introduced parameter-free variants of these algorithms with formal convergence guarantees. For example, AdaGrad++ and Adam++ [57] are novel adaptations that achieve comparable convergence rates to their predecessors without the need for predefined learning rate assumptions, enhancing their practical applicability.

There is a significant gap between the theoretical understanding of optimisation algorithms and their practical performance in deep learning. Recent studies [58] have re-evaluated commonly used assumptions, such as convexity and smoothness, to better align theoretical analyses with empirical observations. By developing new empirical metrics that compare real optimisation behaviour with analytically predicted behaviour, researchers aim to refine the theoretical frameworks governing optimisation in deep learning.

Federated learning introduces unique challenges for convergence due to data heterogeneity and communication constraints. Recent advancements [59] have proposed algorithms like Hybrid Federated Dual Coordinate Ascent (HyFDCA), which extend existing optimisation frameworks to the federated setting, providing provable convergence rates and improved empirical performance across various datasets.

In summary, theoretical advancements in convergence guarantees for deep learning optimisation have progressed beyond idealised models, addressing practical challenges in non-convex landscapes, adaptive methods, and federated settings.

These developments bridge the gap between theory and practice, offering algorithms with robust convergence properties applicable to real-world deep learning scenarios.

## 2.17 Deep Learning convergence in practice

Despite this, let us explain why we do not believe that in deep learning we have converged — even to a local optimum in the majority of cases. Firstly, we are not sure what convergence even means when using mini-batches as each mini-batch will have a different local optimum, and we have only sampled a small proportion of all possible mini-batches. We might assume that the learning rate is so small that we are effectively optimizing the loss for the whole training set, but this would suggest that the mini-batch size is irrelevant, which is usually not the case. Alternatively, we could say that we have converged when the gradient for the entire training set is zero; implying that the gradient for each mini-batch is zero. This is also one of the required conditions for a minimum in classical optimisation; however, empirically we have not observed a training loss curve where there is strong evidence that the gradient ever reaches zero, there is always a signification stochastic signal in the loss curve. And when measured, that the gradient magnitude reduces at all, with it sometimes increasing when the learning rate is dropped. While it might decrease in size, in every example we have seen, it is hard to argue it is zero. Furthermore, in our experience, when training any deep learning model for longer, we have always observed that the weights continue to change.

Putting aside this mini-batch and zero-magnitude gradient problem. Secondly, in a quadratic minimum using a quasi-Newton method such as conjugate gradient the time to reach the minimum would be the dimensionality of the space we are searching, *i.e.* the number of parameters. Considering that typical deep learning models often have millions or even hundred of millions of parameters; the number of updates would seem to be orders of magnitudes smaller than necessary for convergence in this ideal case. Given that we're using a weaker optimiser and are not in a convex, or even everywhere differentiable, search space, we would be very puzzled at why we would converge in this short time.

Thirdly, because a typical momentum decay rate is 0.9, it seems unlikely that the momentum buffer would have decayed to zero for the typical number of epochs being carried out. This means that even if the gradient was zero, the momentum would still be carrying the model forward.

Fourthly, practitioners often reduce their learning rate by a factor of 10 and find that the loss then rapidly decreases, suggesting that the parameters have not reached a minimum. Since the reduced learning rate begets a reduced step size, this new, lower loss, parametrisation is necessarily close to the parametrisation that was assumed to

have converged. Consequently, it is unlikely that the model had reached a minimum since there were better parametrisations close by.

In most instances, the training accuracy reaches 100% very rapidly — often after only a few epochs, but the loss continues falling for hundreds of epochs; as we said, we have never seen a case where the loss does not seem to be decreasing, albeit very slowly. Correctly classifying the training set is not a criterion for converging or reaching the global minimum.

Nonetheless, there exists the question of whether this is a fatal flaw. Clearly, from the perspective of classical optimisation, this seems very strange. However, training deep networks is, in our view, significantly different from most classical optimisation problems. In particular, given the size of the search space, we do not believe that we are ever near to converging only finding plateaux or neighbourhoods of good performance that the optimiser is unable to escape.

Moreover, it is not clear that convergence would even by desirable as early stopping is a well-known regularization strategy; we would argue that it is not done explicitly in deep learning regularly because networks are never close to converging. Thus, the classical notion of converging seems to us not to be a fatal flaw.

# III

# Analysis of learning

Here, we develop some first-order tools to analyse the loss surface of deep neural networks; how the model's path during learning resembles a random walk with a comparison to an Ornstein–Uhlenbeck process, and how the model never converges in parameter space. We then use this reformulation as a stochastic process to calculate directed the optimisation's path is.

Additionally, we look at how cohesive the gradient directions are, that is, do batch directions generally point in the same direction, both for the whole model and per layer.

## 3.1 Previous first order analyses

Most loss landscape and optimisation analyses tend to decrease the difficulty of the analysis. To do this they use simple models *e.g.* linear, use a more complex model but have to throw away much of the information, or make the model tiny to compute otherwise infeasible calculations.

### 3.1.1 Proven bounds are unrealistic

There have been many works that attempt to provide bounds on the convergence of the loss function, however, the bounds are always of little use to the practitioner. It is generally understood that neural networks become more well-behaved with lower learning rates, higher batch sizes, and that wider layers offer more generality, but putting a reasonable bound on these values has, so far, proven out of reach.

For instance, Du *et al.* [60] attempt to prove that gradient descent converges to a global minimum for neural networks. Firstly, their setup is overly optimistic. They previously [61] show that for a two-layer network, where only the second layer is optimised and with a learning rate small enough and layer width large enough, the prediction for each datum approaches its correct class label in linear time. This result

is then used on a deep network of arbitrary depth with all the layers being optimised. The bounds they reach are, unfortunately, not helpful for practical training.

Instead of focusing on the optimisation process itself, Nguyen and Hein [62] show that almost all minima are near-global minima as long as "the number of hidden units of one layer of the network is larger than the number of training points and the network structure from this layer on is pyramidal"; additionally they still require a fully connected network and analytic activation functions. This is a better bound than the previous as some networks, *e.g.* residual networks [63], have around the same order of magnitude of parameters in a layer as the CIFAR-10 dataset has data. However, the problems around defining convergence and achieving a zero gradient are still present; the fact that local minima are near global minima is less useful than it seems when we never find any minima during optimisation and just plateaux.

### 3.1.2 Black box methods

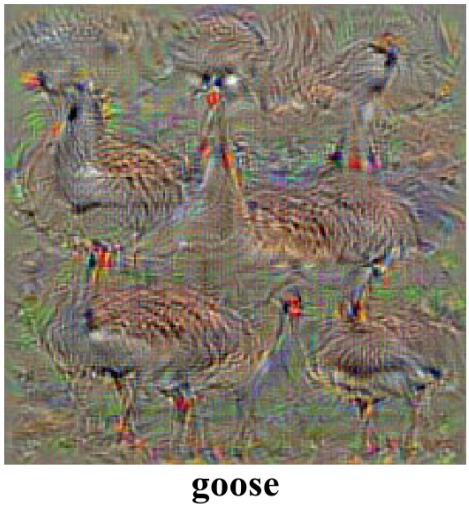
Deep networks, or any function can be analysed as a black box. Black box simply means that the internal workings of the function are not known, and no assumptions are made. Namely, just the input and output can be used to analyse the function. These methods are very interesting since they are applicable to any network.

Chen *et al.* [64] find adversarial examples, *i.e.* examples that are very similar to the original example but are misclassified, by minimising the distance from the initial sample and maximising the network's prediction to the target class. While this paper is not about the optimisation of the model itself, their success in creating adversarial examples shows that information about different networks can be extracted from the input and output alone.

The Ensemble Kalman Inversion (EKI) method has been updated to work with neural networks [65], successfully optimising deep networks on the MNIST dataset [66]. The EKI method attempts to optimise the inverse problem; it "samples particles from a prior distribution, and introduces a motion to move the particles around" [67]; eventually this method finds the minimizer of the cost function, and when the optimisation stops "the ensemble distribution of the particles resembles . . . the posterior distribution in the linear setting." [67]. This means that new models can be developed that use non-differentiable functions such as the Heaviside step function, and the optimisation will still work. EKI is easily parallelisable which is likely to be needed when working with large models and datasets since multiple models need to be modified in conjunction.

### 3.1.3 Visualisation

Visualisation analyses have to decrease the dimensionality of the data so that it can be plotted; moreover the model often needs to be evaluated many times to create a single




(A) Feature visualisation of the "goose" class.

(B) Saliency map of a tractor showing a highlighted area where the tractor lies and dark areas for the rest of the image.

FIGURE 3.1: Feature visualisation and saliency map taken from Simonyan *et al.* [68].

plot. To do this they use simple models *e.g.* linear, or, with more complex models but throwing away much of the information for a specific plot. Despite this, visualisations are a useful tool for understanding the model and the data it is trained on.

Feature visualisation is a method of creating images from the activations of a network. Simonyan *et al.* [68] use feature visualisation to create images that maximise the activation of a particular output neuron. This is done by starting with a random image and then, using backpropagation, updating the image to maximise the activation of the output neuron. They also create saliency maps, see figure 3.1b, which are grey scale images that show the relative parts of the image that the network is looking at to make its decision.

Instead of directly modifying the image, Nguyen *et al.* [69] use a generative model to create images that maximise the activation of a particular output neuron. This generative model is trained to create images that maximise the activation of the output neuron of the target network. Figure 3.2 shows the results of this method for CaffeNet trained in ImageNet. This method can be useful for figuring out what the network is looking for in the input images to classify it as a particular class. For example, the authors give the example of the lawn mower in figure 3.2 where the network appears to be looking for the grass below in addition to the mower itself.

To create the visualisation of the loss landscape in figure 3.3, Li *et al.* [70] sample from two random directions, and calculate the loss over this new basis. This reduction in


FIGURE 3.2: Feature visualisation taken from Nguyen *et al.* [69].

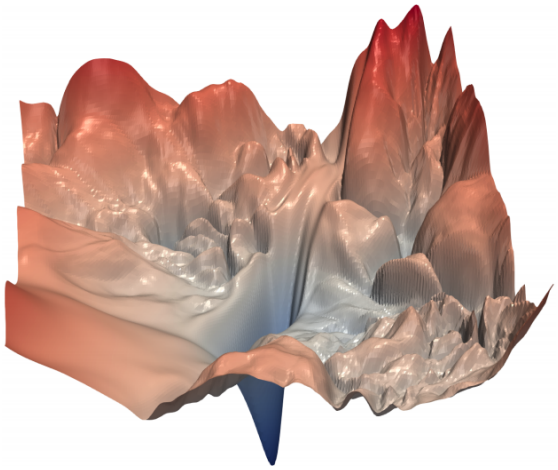


FIGURE 3.3: Loss landscape visualisation taken from Li *et al.* [70]

dimensionality allows the authors to plot the visualisation, however this can hide some of the complexity of the true loss landscape. If the reduced space is non-convex, then the original space is certainly non-convex; however, if the reduced space is convex in, then the $n$-dimensional space is only on average convex *i.e.* the average of the Hessian's eigenvalue is positive. Moreover, whilst this is a good way to view the current loss landscape, it does not provide an understanding of how the optimiser got to that point.

Li *et al.* [70] use Principal Component Analysis (PCA) to find the two most significant directions and track the model's path in only those directions — figure 3.4. These directions are found by performing PCA on the matrix $[\theta_0 - \theta_n, \ldots, \theta_{n-1} - \theta_n]$ where $\theta_0$ is the initial weights and $\theta_n$ is the final weights. These visualisations show a near-convex landscape, and that the optimiser is able to find the minimum in a relatively simple path. Also note the clear change in direction when the learning rate is decreased on the top-left plot; this is a good visualisation of the learning rate schedule in action and corresponds well to the drop on loss that is often seen when lowering the learning rate.

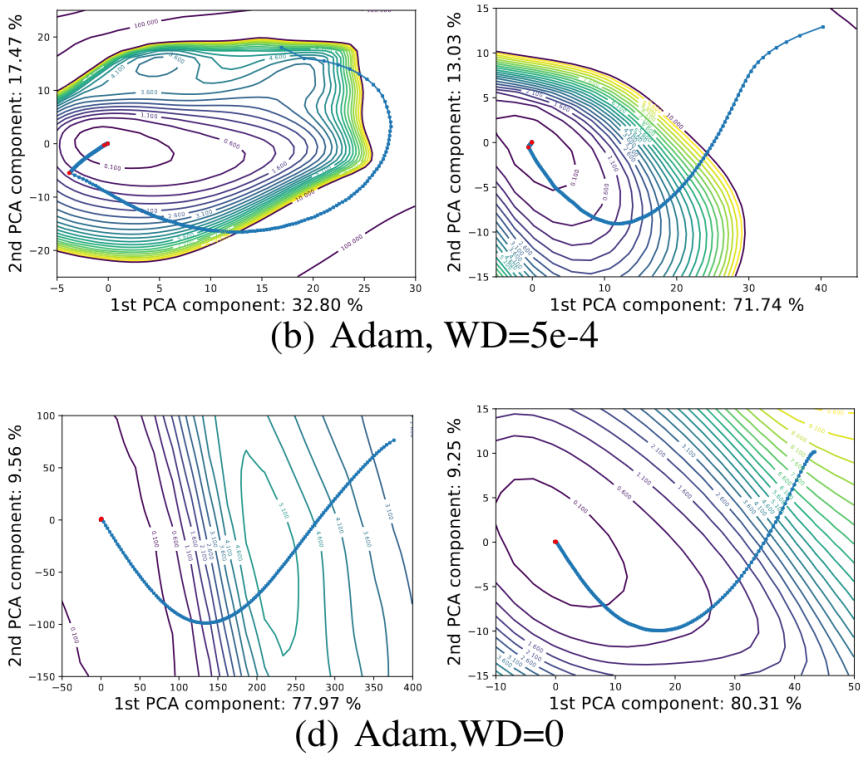



FIGURE 3.4: Visualisation of optimisation paths using PCA for a VGG-9 model. "Epochs where the learning rate was decreased are shown as red dots." Taken from Li *et al.* [70]

### 3.1.4 Representations

Shwartz-Ziv and Tishby [71] analyse the mutual information between the representations and both the inputs and outputs over time; mutual information is the intersection of the individual entropy of two variables defined by $I(X;Y) = D_{KL}(P_{(X,Y)} \parallel P_X \otimes P_Y)$. Figure 3.5, we see from the left plot that at the start of training the initial layers — blue and green — have a high mutual information with both the inputs and outputs, but the latter layers — orange and pink — do not share much information with either the inputs or outputs. This is expected as the as each randomly initialised layer add more noise and removes information from the input. As training progresses, the mutual information in every case increases; the network is learning to propagate the information from the input to the output.

Shwartz-Ziv and Tishby [71] interestingly note that at some point a regime shift occurs, and the mutual information between the input and representations, $I(X;T)$, starts to decrease. As the authors designed the dataset so that the mutual information between the input and output is approximately 0.99 bits, there is no advantage to the network keeping more than this. Therefore, the network starts to compress the representations. The network is overfitting to the training data — it is learning to ignore all the information that is not shared with the outputs in the training data.

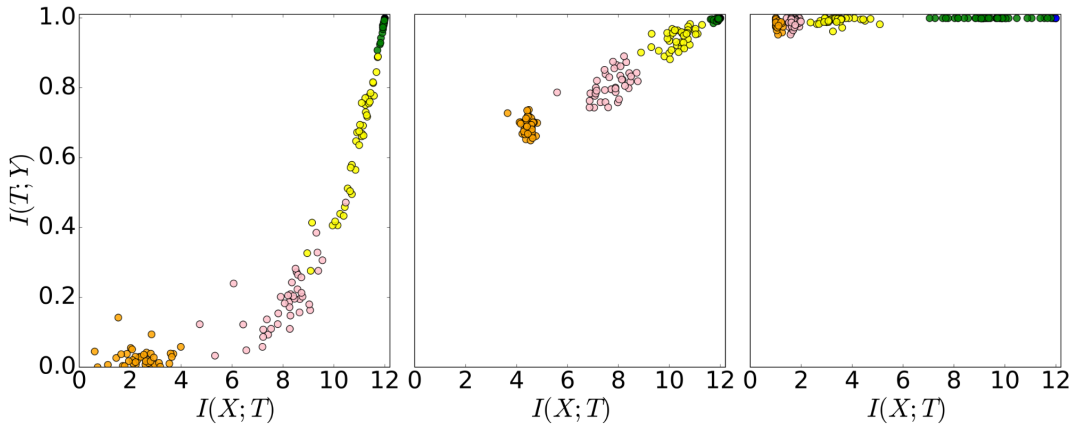



FIGURE 3.5: "Snapshots of layers (different colors) of 50 randomized networks during the SGD optimization process in the information plane (in bits): left - with the initial weights; center - at 400 epochs; right - after 9000 epochs. The reader is encouraged to view the full videos of this optimization process in the information plane at `https://goo.gl/rygyIT`". Taken from Shwartz-Ziv and Tishby [71].

Since we know that information from the input is being lost, it is interesting to look at the similarity between the representations. Nguyen *et al.* [72] compare the Centred Kernel Alignment (CKA) similarity between the representations different layers and how that is affected by the width of the layers and the depth of the network. They find that there is a block structure to the similarity matrix, that is a large number of adjacent layers have similar representations, figure 3.6. This is interesting as it shows how the network is over-parametrised and that not all the layers are needed. A small oversight is that ResNets have an inherent block structure, for CIFAR-10 the networks have three equally deep macro-layers, this can easily be seen in the plot for ResNet-164 in figure 3.6. Whilst there are smaller block structures shown, it would be interesting to know if these are present in other types of network.

### 3.1.5 Metrics

Various metrics can be created that provide an empirical measure of various properties of the loss landscape.

Wu *et al.* [73] create a metric that measures the roughness of the loss landscape by extending the concept of total variance to higher dimensions.

$$\begin{aligned} \mathrm{TV}(f) &= \sup \sum_{i=1}^{n-1} |f(x_{i+1}) - f(x_i)|, \\ &= \int_a^b \left|f'(x)\right| dx, \end{aligned}$$

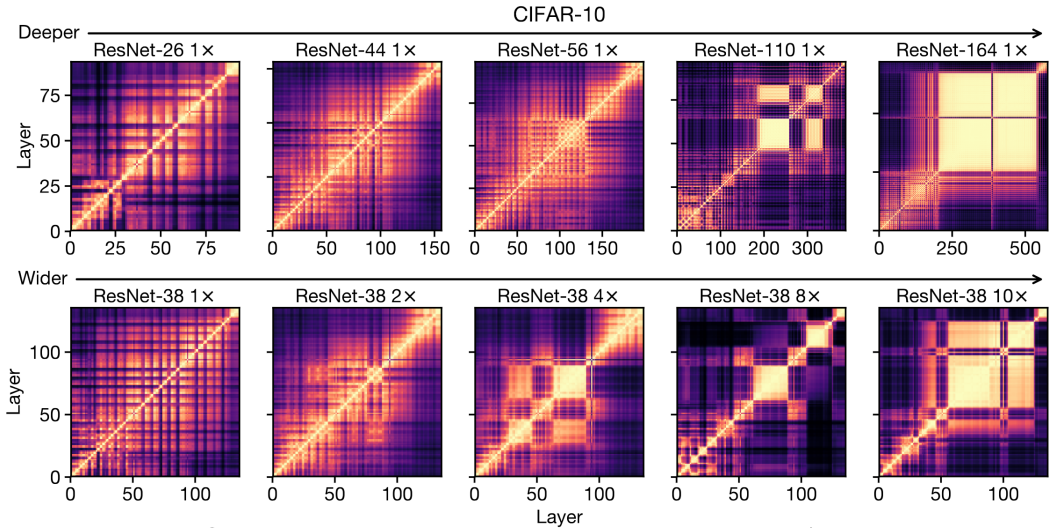



FIGURE 3.6: CKA similarity matrices between the representations of different layers of a network showing a block structure. Taken from Nguyen *et al.* [72] & cropped.

depending on if $f$ is absolutely continuous or not. To do this they look at a scaled version of total variation,

$$\mathrm{T}(f) = \frac{TV(f)}{(b-a)\left(\max\limits_{a\leq x\leq b} f(x) - \min\limits_{a\leq x\leq b} f(x)\right)},$$

on a distribution of Gaussian perturbations, $d$, with defined length, $F_d(s) = \mathcal{L}_\theta(\theta + sd)$. Their roughness index is then the standard deviation of $F_d$ over these random directions normalised by their expectation — $\mathcal{I}(\mathcal{L}_\theta, \theta) = \frac{\mathbf{std}_t F_d}{\mathbb{E}[F_d]}$. This metric is then applied to the problem of solving a Partial Differential Equation (PDE) with neural networks comparing the deep Ritz method [74] and the deep Galerkin method [75], and they find that the roughness of the loss landscape is related to the convergence of the optimisation algorithm — the Galerkin method has a lower roughness index along with a closer fit of the loss to the $L_2$ error of the PDE.

## 3.2 Measuring directedness in optimisation

This section develops new tools for analysing the geometry of optimisation paths in deep learning. We introduced directedness metrics, compared optimisation behaviour to stochastic processes, and established tools for estimating local curvature. These analyses reveal structure and the efficiencies nad inefficiencies in deep learning training that are not visible from scalar losses alone.

Optimisation of deep networks using SGD only considers one point of parameter space, to optimise better some methods that extend SGD can be thought of as adding

in more landscape information rather than just a single point. For instance, SGDM incorporates information from all the previous evaluations, and using the Hessian can be thought of as taking into account some small neighbourhood around the parameters.

These methods only add a small amount of local landscape information, but, more importantly, the information is only from where the optimiser has come from, assuming no information about the landscape ahead of it. Of course, factoring information about the whole landscape is impossible since, as discussed, evaluating the cost is an expensive operation. Therefore, as these methods have done, we must factor in only the most necessary and efficiently calculable information.

### 3.2.1 Path Distances

To understand the geometric structure of training, we can track how a model's parameters evolve through weight space during optimisation. Let $\theta_t \in \mathbb{R}^n$ denote the parameter vector at training step $t$. We define two complementary metrics. The total displacement from the initialisation point, $D_t(t)$, which measures the net change in parameters relative to the starting point. And the cumulative path length, $D_s(t)$, which sums the magnitudes of the parameter updates across all steps.

$$D_t(t) = \|\theta_t - \theta_1\|_2, \tag{3.1}$$

$$D_s(t) = \sum_{i=1}^{t} \|s_i\|_2, \tag{3.2}$$

where $s_i = \theta_i - \theta_{i-1}$ is the update step with $s_1 = \mathbf{0}$.

The cumulative path length $D_s(t)$ reflects the total effort expended by the optimiser, while the displacement $D_t(t)$ captures the net movement of the model in weight space.

### 3.2.2 Directedness

The ratio $\psi(t) = D_t / D_s$ thus encodes the efficiency of the optimisation, values close to 1 indicate highly aligned, efficient updates, whereas small values imply circuitous or redundant trajectories.

To keep from over-smoothing in practice, we measure this ratio over a fixed subset of time,

$$D_t(t - N) = \|\theta_t - \theta_{t-N}\|_2, \tag{3.3}$$

$$D_s(t - N) = \sum_{i=N}^{t} \|s_i\|_2. \tag{3.4}$$

Then, we define the directedness ratio to be,

$$\psi_N(t) = \frac{D_t(t-N)}{D_s(t-N)}. \tag{3.5}$$

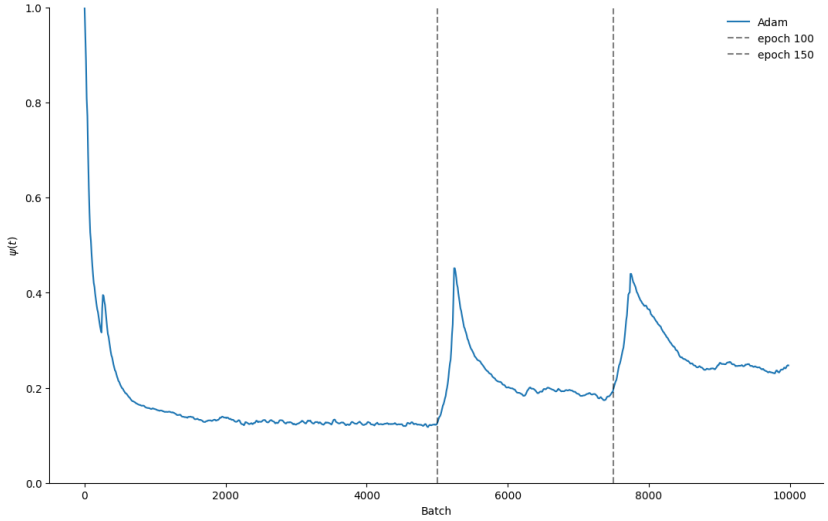
This framework is sensitive to both the local curvature of the loss surface and the consistency of gradient directions across time. In regions of high curvature or sharp minima, the optimiser may be forced to take small, oscillating steps that result in low displacement relative to path length. Similarly, frequent gradient misalignment—due to stochasticity, poor conditioning, or conflicting objectives—can lead to meandering trajectories with minimal net progress.

Consequently, monitoring $D_t(t)$, $D_s(t)$, and their ratio $\psi(t)$, provides insight not only into convergence behaviour but also into the geometry of learning, how straight, stable, or turbulent the training trajectory is in weight space.

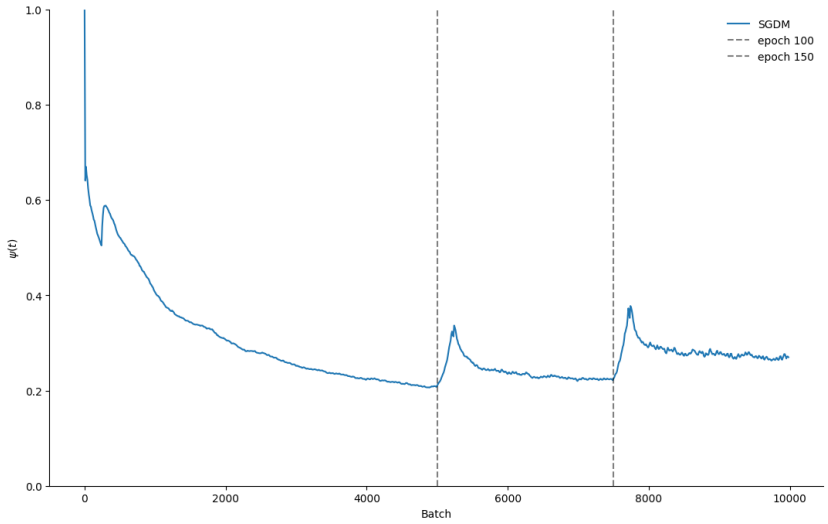
Since, by definition, $D_s(t) \geq D_t(t)$, $D_t(t) > 0$, and $D_s(t) > 0$, then $\psi \in [0, 1]$. The two extremes of the directedness ratio tell us about two extreme optimisation paths. Firstly, a maximal or aligned regime; we have a completely directed optimisation, $D_t(t) = D_s(t)$, then $\psi(t) = 1$. Under this regime every update points exactly towards the solution, and the optimisations trajectory is a linear interpolation of the initial position and the solution. Secondly, we have a confined regime, if we have a stationary optimisation path, *i.e.* each step is confined to some distance next to the initial point, $D_s(t) \gg D_t(t)$, then $\psi(t) \approx 0$. Consequently, the higher $\psi(t)$ is the more directed the optimisation is.

### 3.2.3 Directedness of deep network optimisation

We calculate the directedness ratio, $\psi$, for the deep network optimisation. Figures 3.7a and 3.7b depicts the directedness ratio for SGDM and Adam respectively. Figure 3.7 reveals several distinct phases of optimisation geometry; during early training, $t < 500$, $\psi(t)$ is initially high but rapidly decays, reflecting the optimiser's transition from large, directional updates to more curved and noisy dynamics as it begins to explore non-linear regions of the loss surface. In the middle of training $\psi(t)$ settles into a gradual decline, indicating increasing redundancy in the path as gradients become less aligned and weight updates contribute less to net displacement. And in the late stages of training $\psi(t)$ stabilizes at a low value, indicating persistent local oscillation or minor fine-tuning adjustments with limited directional progress—typical behaviour near convergence. Additionally, epoch transitions (vertical dashed lines at epochs 100 and 150) show sharp increases in $\psi(t)$ coinciding with learning rate drops, as the optimiser regains alignment and begins more directed movement through the now-narrower local landscape, before beginning the same pattern of early-mid-late




(A) $\psi$ for SGDM on a ResNet-20 trained using a batch size of 1024, learning-rate of 0.05, momentum of 0.85, weight-decay of $10^{-2}$, and a learning rate schedule of $\times 0.1$ at epochs 100, 150 for 200 epochs.




(B) $\psi$ for Adam on a ResNet-20 trained using a batch size of 1024, learning-rate of 0.01, beta one of 0.9, beta two of 0.99, weight-decay of $10^{-4}$, and a learning rate schedule of $\times 0.1$ at epochs 100, 150 for 200 epochs.

FIGURE 3.7: Time evolution of the local directedness efficiency metric $\psi_{250}(t)$ during training of a model with SGDM, over a rolling window of $N = 250$ steps revealing several distinct phases of optimisation geometry; early, mid, and late stages of training and a also the effects of learning rate scheduling. Note the artefacting at batch 250 is just from taking a window that is 250 batches wide.

training again. This is consistent with the intuition that learning rate drops help "straighten" the trajectory and refine movement toward a sharp or flat minimum.

Comparing the two plots of $\psi_{250}(t)$ for SGDM and Adam gives a perspective on the geometric behaviour of these optimisers during training. The initial trajectory of Adam exhibits a sharper initial drop in $\psi(t)$, quickly descending to around 0.15, whereas SGDM maintains a much higher $\psi(t)$ in the first 1000 batches. Meaning that

SGDM takes more directionally coherent steps early on, implying strong initial alignment of gradients. Adam's momentum and adaptivity inject more update variance early on — *i.e.* Adam more quickly finds a flat plateau. In the mid training, batch 1000-5000, SGDM maintains slightly higher $\psi$ throughout continuing to make more geometrically efficient progress through weight space. Adam, in contrast, has settled into a regime of smaller, oscillatory adjustments, suggesting more local adaptation and possibly less reliance on consistent gradient alignment. Adam's adaptive scaling may flatten curvature faster, leading to slower directional movement but better basin adaptation. This is not to say that Adam is performing worse, the earlier flattening also probably indicates that the model has reached a better performance earlier. Both optimisers exhibit sharp spikes in $\psi$ immediately following learning rate drops. SGDM's spikes are narrower; Adam's are sharper and more sustained, particularly after epoch 150. We interpret this as, learning rate drops re-stabilize both optimisers, increasing directional alignment (fewer overshoots). The larger spike for Adam suggests that its adaptive dynamics amplify the effect of LR drops. This confirms that LR scheduling can strongly reorient optimiser geometry, temporarily restoring alignment even late in training. Finally, at the end of training, SGDM stabilizes at a slightly higher $\psi$ than Adam indicating that SGDM continues to make slightly more directional progress, possibly due to its momentum carrying it across shallow valleys, whilst Adam appears to dampen movement, favouring local stabilization over traversal.

### 3.2.4 Random Walk

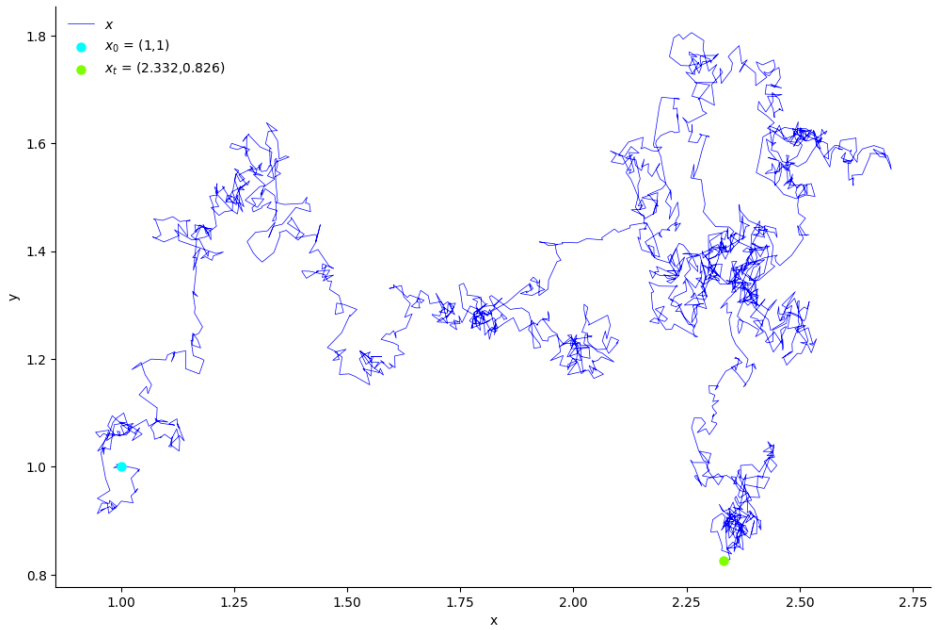



FIGURE 3.8: A realisation of a random walk in 2 dimensions.

To gain a better understanding of our directedness measure we calculate it for a random walk. A random walk is a stochastic process where the next state is a random step, independent of any previous state. The step is sampled from a normal distribution, *i.e.* $\theta_t = \theta_{t-1} + s_t$, where $s_t \sim \mathcal{N}(0, \sigma^2 \boldsymbol{I})$.

For a random walk, we calculate the expected cumulative increment distance,

$$\mathbb{E}\left[D_s(t)\right] = \mathbb{E}\left[\sum_{i=1}^{t} \|s_i\|_2\right] = \sum_{i=1}^{t} \mathbb{E}\left[\|s_i\|_2\right],$$

then, from lemma A.3 — the expected 2-norm of an n-dimensional random vector,

$$= \sum_{i=1}^{t} \sigma\sqrt{2}\, G(n) = t\sigma\sqrt{2}\, G(n). \tag{3.6}$$

Next, the expected total displacement for a random walk. Let $S = \sum_{i=1}^{t} s_i$,

$$\mathbb{E}\left[D_t(t)\right] = \mathbb{E}\left[\|\theta_t - \theta_1\|_2\right] = \mathbb{E}\left[\left\|\sum_{i=1}^{t} s_i\right\|_2\right] = \mathbb{E}\left[\|S\|_2\right]. \tag{3.7}$$

Since the elements, $S_i$, of $S$ are a sum of $t$ independent random variables, $S \sim \mathcal{N}(0, t\sigma^2 \boldsymbol{I})$, and lemma A.3,

$$\mathbb{E}\left[D_t(t)\right] = \sigma\sqrt{2t}\, G(n). \tag{3.8}$$

Finally, we calculate the directedness ratio, or the ratio between the expected displacement and the expected cumulative increment distance for a random walk.

$$\mathbb{E}\left[\psi(t)\right] = \mathbb{E}\left[\frac{D_t(t)}{D_s(t)}\right], \tag{3.9}$$

$$= \mathbb{E}\left[\frac{\left\|\sum_{i=1}^{N} s_i\right\|}{\sum_{i=1}^{N} \|s_i\|}\right] \tag{3.10}$$

This is a ratio of dependent random variables, the numerator is the norm of sum of Gaussians, $\sim \chi_d(\sqrt{N}\sigma)$, and the denominator is the sum of $N$ *i.i.d.* norms of Gaussians, $\sim \sum \chi_d(\sigma)$. This exact quantity has been studied in random walk theory and statistics. In particular, the expected ratio,

$$\mathbb{E}\left[\frac{\|\sum_{i=1}^{N} X_i\|}{\sum_{i=1}^{N} \|X_i\|}\right] = \frac{\mathbb{E}[\|\sum_{i=1}^{N} X_i\|]}{\mathbb{E}[\sum_{i=1}^{N} \|X_i\|]} + (\text{correction term}) \tag{3.11}$$

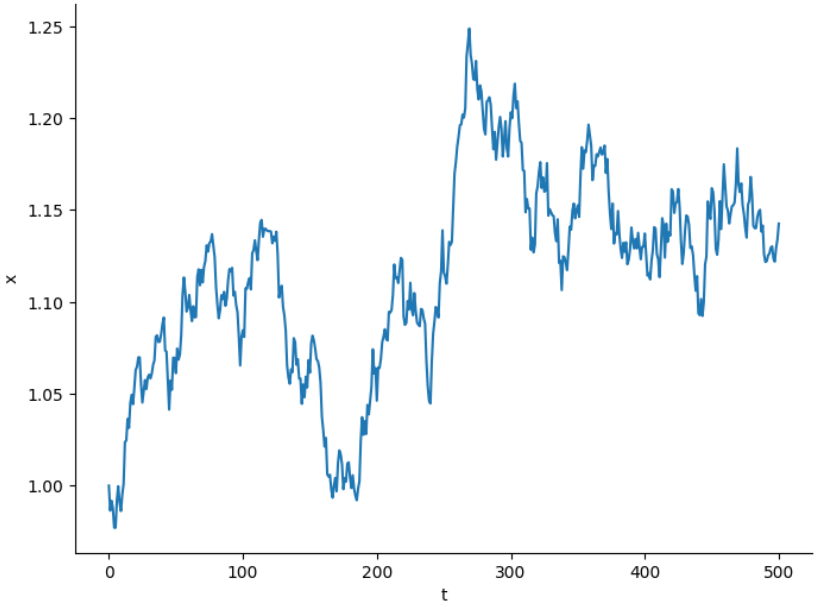



FIGURE 3.9: A random walk with $\theta_0 = 0$, $\sigma = 0.01$, and $n = 1$.

The correction is positive due to convexity of the ratio, $\mathbb{E}[X/Y] > \mathbb{E}[X]/\mathbb{E}[Y]$ when $X, Y > 0$ and positively correlated. There are numerical approximations [76], [77] that show,

$$\mathbb{E}\left[\frac{\|\sum_{i=1}^{N} X_i\|}{\sum_{i=1}^{N} \|X_i\|}\right] \approx \frac{1}{\sqrt{N}} \cdot \left(1 + \frac{c}{d}\right) \tag{3.12}$$

where $c$ is a constant (typically small), and $d$ is the dimension. So in high dimensions, this correction vanishes, and $1/\sqrt{N}$ becomes a good approximation.

Thus,

$$\mathbb{E}\left[\frac{D_t(t)}{D_s(t)}\right] \approx \mathbb{E}\left[\frac{\|\sum_{i=1}^{t} s_i\|}{\sum_{i=1}^{t} \|s_i\|}\right], \quad s_i \sim \mathcal{N}(0, \sigma^2 I), \tag{3.13}$$

$$\approx \frac{1}{\sqrt{t}}, \tag{3.14}$$

with the error vanishing as $t \to \infty$.

We simulate a random walk in 1 dimension with $\theta_0 = \mathbf{0}$, and $\sigma = 0.01$; showing, in figure 3.9, the random walk behaviour and, in figure 3.10, a 100 dimensional random walk where the expected distances are a good approximation for the simulated values. Additionally, we plot the expected directedness measure, $\psi(t) \approx \frac{1}{\sqrt{t}}$, in figure 3.11 showing that the value of $\psi$ is indeed $\frac{1}{\sqrt{t}}$ and that it tracks his value accurately, even from the beginning.

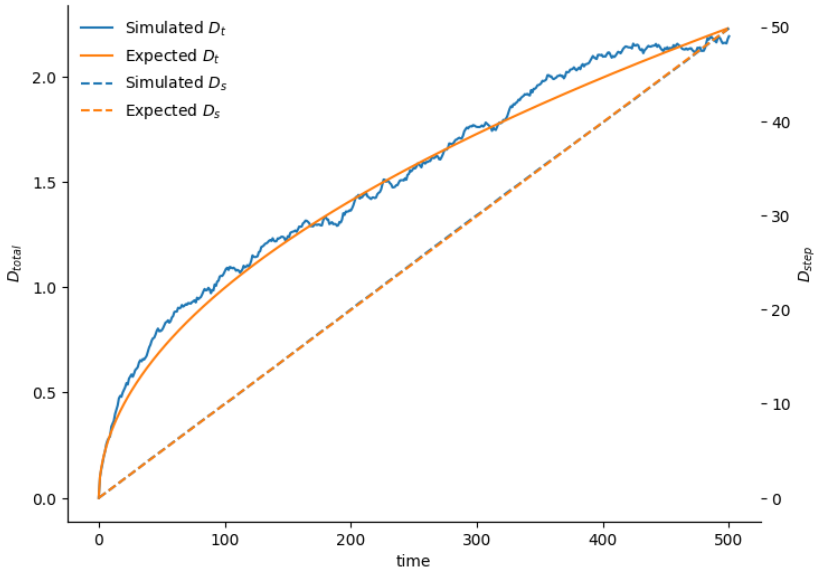



FIGURE 3.10: A random walk with $\theta_0 = \mathbf{0}$, $\sigma = 0.01$, and $n = 100$.

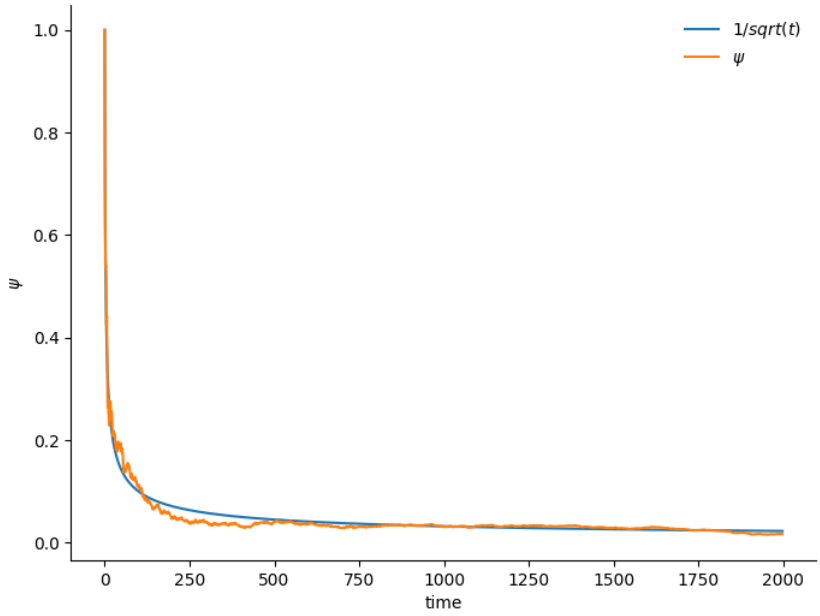



FIGURE 3.11: The directedness estimation, $\psi$ for a random walk process with $\sigma = 0.05$, and $n = 100$.

### 3.2.5 Deep Network Optimisation as a Random Walk

We train a ResNet-20 model on CIFAR10 using SGDM and Adam and plot $D_s$ and $D_t$ against the expected values of the random walk. To calculate the parameters for the expected values, we minimise, using the Nelder-Mead [78] method, the sum of the

mean squared errors of the estimated $D_s$ and $D_t$ values from the real values,

$$\min_{\sigma} \sum_{i=1}^{t} \gamma \left( D_s^{\text{model}}(i) - D_s^{\text{RW}}(i) \right)^2 + (1-\gamma) \left( D_t^{\text{model}}(i) - D_t^{\text{RW}}(i) \right)^2, \tag{3.15}$$

where $\gamma$ is a weighting factor between the two losses to give the same importance to both metrics. Figures 3.12a and 3.12b show the results of this experiment, where we see that the optimisation is directed for both the SGDM and Adam optimisers demonstrated by the displacement being much greater in practice than expected from a random walk.

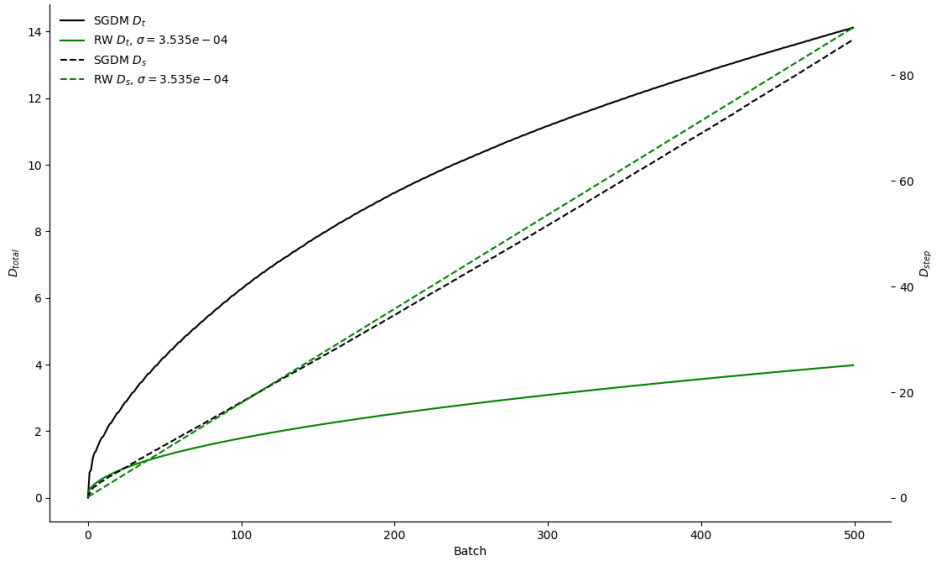
SGDM shows a shape more similar to the random walk, specifically, at the start, where Adam is extremely directed. The most likely reason is that this is an artefact of the adjusted momentum; both the momentum terms and velocity terms algorithm 1 are initialised to the zero vector, and so the first update term reduces to $\eta \frac{g_0}{\sqrt{g_0^2}+\epsilon}$ which is approximately $\eta \operatorname{sgn}(g_0)$, and so has a much larger norm than $g_0$ as each gradient element is usually significantly smaller than 1.

Since the optimisation is clearly directed, we can try to model it as a more nuanced stochastic process to model the optimisation.

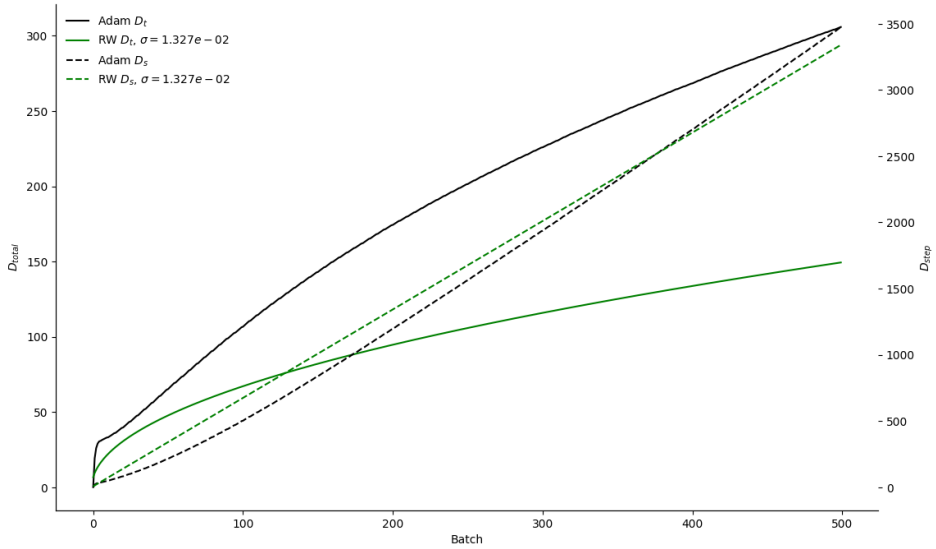
### 3.2.6 Auto-regressive Processes

While deep learning training dynamics are governed by high-dimensional, non-linear interactions between weights, gradients, and data, it is often useful to compare them against simple generative processes. One such baseline, in addition to random walks, is the first-order autoregressive, AR(1), process, a discrete-time linear dynamical system defined by $x_t = \alpha x_{t-1} + (1-\alpha)\mu + \epsilon_t$, with $\alpha \in (-1, 1)$ and *i.i.d.* noise $\epsilon_t \sim \mathcal{N}(0, \sigma^2 I)$. At a glance, the analogy may seem overly reductive, modern deep learning involves interacting layers of non-convex parameter transformations, curvature-sensitive gradient signals, and time-varying dynamics from learning rate schedules, momentum, and batch noise. Nevertheless, the AR(1) model provides a tractable and interpretable baseline for describing temporal dependence and correlation structure in weight evolution over training time.

Though clearly an oversimplification, this process captures a minimal form of temporal correlation and directional persistence, both of which are commonly observed in the trajectories of neural network parameters during training [70], [79]–[81]. Here, the autoregressive coefficient $\phi$ encodes temporal smoothness or inertia, akin to momentum terms in optimisers like SGDM or Adam, see [45] for a discussion on implicit momentum dynamics. Even in the absence of explicit momentum, correlations in gradient directions due to structured data and consistent local curvature can produce autoregressive-like behaviour. Moreover, several studies have noted that optimisation trajectories exhibit low effective dimensionality and




(A) $D_s$ and $D_t$ for SGDM on a ResNet-20 and the expected for a random walk in same dimension.




(B) $D_s$ and $D_t$ for Adam on a ResNet-20 and the expected for a random walk in same dimension.

FIGURE 3.12: Deep network optimisation measures plotted against the expectations for a random walk.

dominant modes of variation that can be approximated well with linear dynamical systems. For example, [82] and [83] show that high-dimensional training paths often lie on low-dimensional manifolds with strong temporal structure. In this context, the AR(1) model serves as a useful null hypothesis: it lets us ask whether observed directionality, curvature, or gradient alignment is stronger or weaker than what would be expected from a simple correlated random walk. Comparing real training runs to an AR(1) reference allows us to isolate non-trivial geometric features of deep learning dynamics from the baseline effects of correlated noise and incremental drift. Such comparisons make it possible to detect when training is more (or less)

structured, aligned, or efficient than what would be expected from even the simplest model with memory.

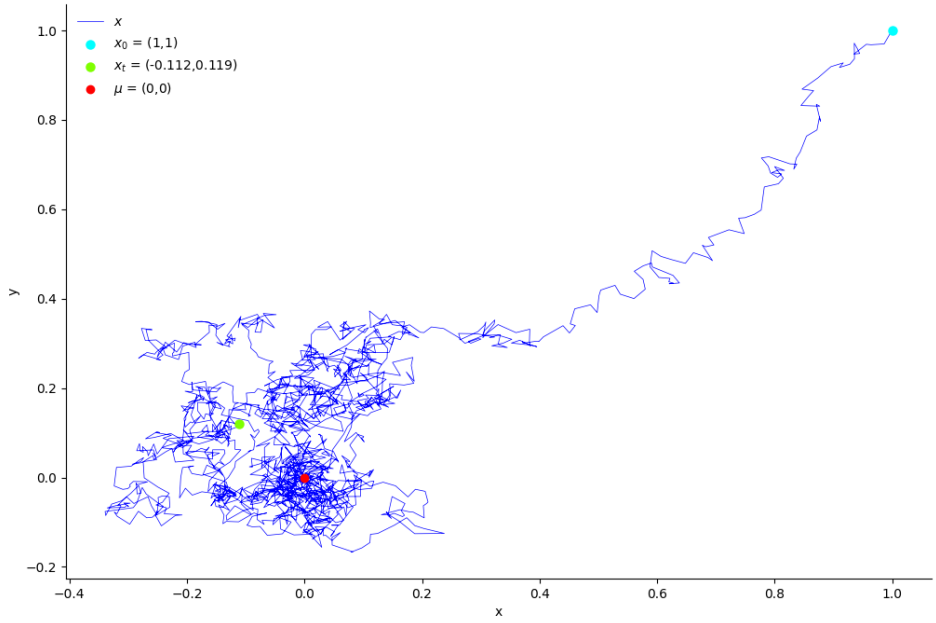



FIGURE 3.13: A realisation of an AR process in 2 dimensions.

An Auto-regressive (AR) process is a stochastic process that has a mean-reverting tendency. It satisfies three important properties.

- The Markov property means that the conditional distribution of current value of the process depends only on the previous value — and constants, but not on any additional information about the past; *i.e.* $\theta_t = f(\theta_{t-1})$.
- The Gaussian property means that every linear combination of a finite number of increments of the process is normally distributed *i.e.* $\sum_{i \subseteq T} s_i \sim \mathcal{N}(\mu', \sigma'^2)$.
- A stationary process is one whose joint probability distribution does not change when shifted in time or space.

Although an AR(1) process with vector elements is close to an AR(p) process, that is, a process in $p$ dimensions, we will only consider it an AR(1) process of $n$-dimensional vectors as the decay rate is identical for all parameters. Herein, AR will refer to an AR(1) process with $n$-dimensional vectors. We will consider the formulation of an AR(1) but where the variable $\theta$ is an $n$-dimensional vector; we introduce a condition on the decay parameter $|\alpha| < 1$, and a mean $\mu$; thus we define the update set of an AR process as,

$$\theta_t = (1 - \alpha)\mu + \alpha\theta_{t-1} + \epsilon_t. \tag{3.16}$$

Unrolling the updates for these processes,

$$\begin{aligned}
\theta_t &= (1-\alpha)\mu + \epsilon_t + \alpha\theta_{t-1}, \\
&= \sum_{i=1}^{t} \alpha^{t-i} \left((1-\alpha)\mu + \epsilon_i\right) + \alpha^t\theta_0, \\
&= (1-\alpha)\mu \sum_{i=1}^{t} \alpha^{t-i} + \sum_{i=1}^{t} \alpha^{t-i}\epsilon_i + \alpha^t\theta_0, \\
&= (1-\alpha)\mu \frac{\alpha^t - 1}{\alpha - 1} + \sum_{i=1}^{t} \alpha^{t-i}\epsilon_i + \alpha^t\theta_0, \\
&= \alpha^t\theta_0 + (1-\alpha^t)\mu + \sum_{i=1}^{t} \alpha^{t-i}\epsilon_i, \qquad (3.17) \\
&= \mu + \alpha^t(\theta_0 - \mu) + \sum_{i=1}^{t} \alpha^{t-i}\epsilon_i. \qquad (3.18)
\end{aligned}$$

We can interpret these unrolled equations in two ways. First, from equation (3.17) we see that the process is a noisy exponential interpolation between the initial value, $\theta_0$, and the mean, $\mu$. This is analogous to SGDM where the update is some decay of the weights, plus some signal plus some random noise. Secondly, equation (3.18) shows that the process converges noisily to the mean, $\mu$, by adding an exponentially weighted proportion of the difference of the initial state from the mean.

Before we can calculate the expected distances for an AR process, we need to calculate the step, $s_t$, in terms of the starting conditions and noise,

$$\begin{aligned}
s_t &= \theta_t - \theta_{t-1}, \\
&= (1-\alpha)\mu + (\alpha-1)\theta_{t-1} + \epsilon_t,
\end{aligned}$$

Substituting equation (3.18),

$$\begin{aligned}
s_t &= (1-\alpha)\mu + (\alpha-1)\left(\mu + \alpha^{t-1}(\theta_0 - \mu) + \sum_{i=1}^{t-1} \alpha^{t-i-1}\epsilon_i\right) + \epsilon_t, \\
&= (\alpha-1)\alpha^{t-1}(\theta_0 - \mu) + (\alpha-1)\sum_{i=1}^{t-1} \alpha^{t-i-1}\epsilon_i + \epsilon_t. \qquad (3.19)
\end{aligned}$$

For an AR process, we calculate the expected cumulative increment distance $\mathbb{E}\left[D_s(t)\right]$. Starting with substituting equation (3.19) into equation (3.2),

$$\mathbb{E}\left[D_s(t)\right] = \mathbb{E}\left[\sum_{k=1}^{t} \|s_k\|_2\right] = \sum_{k=1}^{t} \mathbb{E}\left[\left\|(\alpha-1)\alpha^{k-1}(\theta_0 - \mu) + (\alpha-1)\sum_{i=1}^{k-1} \alpha^{k-i-1}\epsilon_i + \epsilon_k\right\|_2\right].$$

For simplicity, let $a_k = (\alpha-1)\alpha^{k-1}(\theta_0-\mu)$, and $b_k = (\alpha-1)\sum_{i=1}^{k-1}\alpha^{k-i-1}\epsilon_i$, then,

$$\begin{aligned}\mathbb{E}\left[D_s(t)\right] &= \sum_{k=1}^{t}\mathbb{E}\left[\|a_k+b_k+\epsilon_k\|_2\right],\\ &= \sum_{k=1}^{t}\mathbb{E}\left[\sqrt{\|a_k\|_2^2+\|b_k\|_2^2+\|\epsilon_k\|_2^2+2\left(a_k\cdot b_k+a_k\cdot\epsilon_k+b_k\cdot\epsilon_k\right)}\right].\end{aligned}$$

Using Jensen's inequality, that all $\epsilon_k$ are independent so that $\mathbb{E}\left[\epsilon_k\right]=0$, and that $\mathbb{E}\left[b_k\right]=0$ as $\epsilon_i$ has zero mean,

$$\mathbb{E}\left[D_s(t)\right] \leq \sum_{k=1}^{t}\sqrt{\|a_k\|_2^2+\mathbb{E}\left[\|b_k\|_2^2\right]+\mathbb{E}\left[\|\epsilon_k\|_2^2\right]}.$$

From lemmas A.4 and A.10 on the expected square norm of vectors and decayed vectors,

$$\begin{aligned}\mathbb{E}\left[D_s(t)\right] &\leq \sum_{k=1}^{t}\sqrt{\|(\alpha-1)\alpha^{k-1}(\theta_0-\mu)\|_2^2+(\alpha-1)^2\sigma^2 n\frac{\alpha^{2k-2}-1}{\alpha^2-1}+\sigma^2 n},\\ &\leq \sum_{k=1}^{t}\sqrt{(\alpha-1)^2\alpha^{2k-2}\|\theta_0-\mu\|_2^2+\sigma^2 n\left((\alpha-1)\frac{\alpha^{2k-2}-1}{\alpha+1}+1\right)}.\end{aligned}$$

To calculate the expected displacement $\mathbb{E}\left[D_t(t)\right]$ for an AR process, we start with substituting equation (3.18) into equation (3.1),

$$\begin{aligned}D_t(t) &= \|\theta_t-\theta_0\|_2,\\ &= \left\|(1-\alpha^t)(\mu-\theta_0)+\sum_{i=1}^{t}\alpha^{t-i}\epsilon_i\right\|_2,\\ &= \sqrt{(1-\alpha^t)^2(\mu-\theta_0)^2+\left(\sum_{i=1}^{t}\alpha^{t-i}\epsilon_i\right)^2+2(1-\alpha^t)(\mu-\theta_0)\sum_{i=1}^{t}\alpha^{t-i}\epsilon_i},\end{aligned}$$

Using Jensen's inequality and lemma A.10,

$$\begin{aligned}\mathbb{E}\left[D_t(t)\right] &\leq \sqrt{(1-\alpha^t)^2\|\mu-\theta_0\|_2^2+\mathbb{E}\left[\left\|\sum_{i=1}^{t}\alpha^{t-i}\epsilon_i\right\|_2^2\right]},\\ &\leq \sqrt{(1-\alpha^t)^2\|\mu-\theta_0\|_2^2+\sigma^2 n\frac{\alpha^{2t}-1}{\alpha^2-1}}.\end{aligned}$$

We simulate an AR process in 1 dimension, in figure 3.14; that demonstrates the reversion to the mean, and, in figure 3.15, a 100 dimensional AR process where the distance measures align well with their expected values.

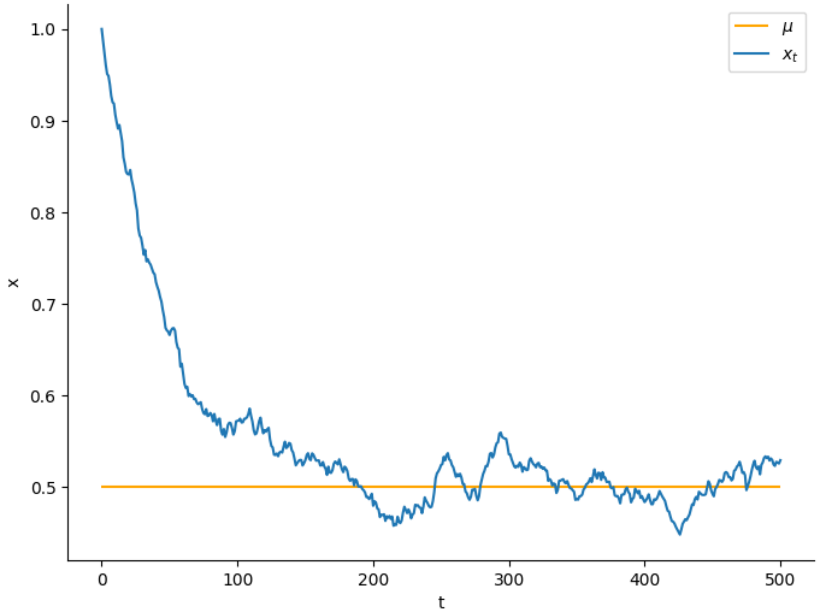



FIGURE 3.14: An AR process with $\theta_0 = 1$, $\mu = 0.5$, $\alpha = 0.98$, $\sigma = 0.005$, and $n = 1$.

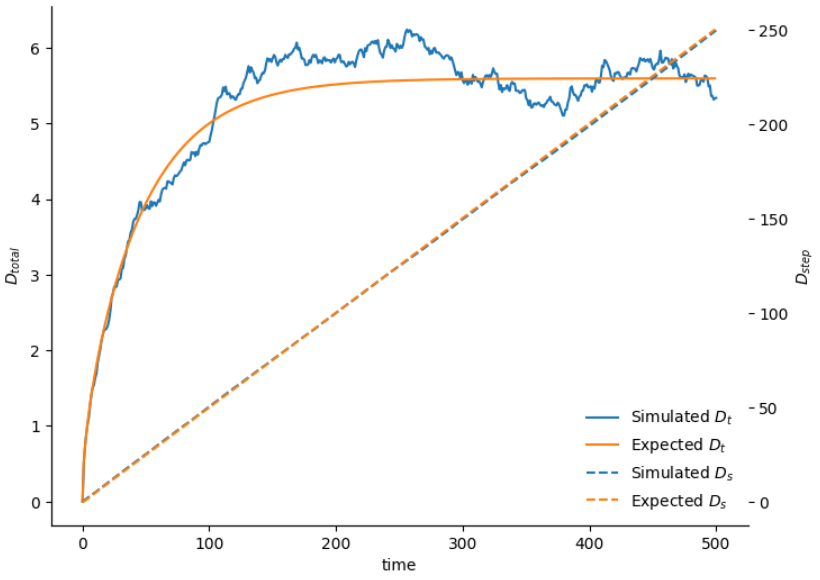
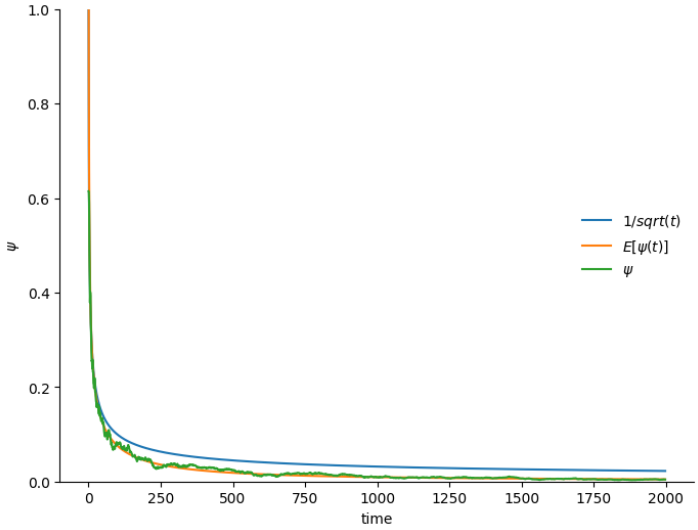


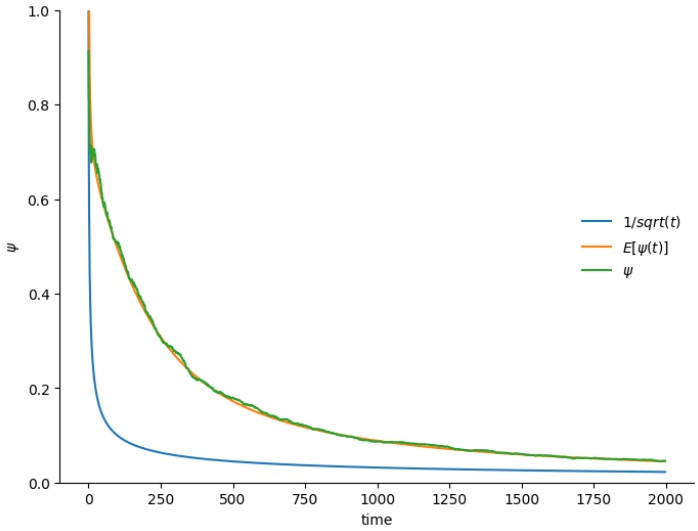

FIGURE 3.15: An AR process with $\theta_0 = \mathbf{1}$, $\mu = \mathbf{0.5}$, $\alpha = 0.98$, $\sigma = 0.05$, and $n = 100$.

These plots figure 3.16, show the geometric efficiency of AR trajectories, as quantified by $\psi(t)$, changes over time and as the noise scale $\sigma$ varies. The realized $\psi(t)$ (green) fluctuates early on but closely follows the expected value (orange) as $t$ increases, but both are below the random walk baseline ($1/\sqrt{t}$, blue) throughout. This indicates that when there is substantial noise, the AR process's directionality is limited and that it will oscillate widely around its mean.

With reduced noise, the trajectory becomes much shallower, and both the expected $\mathbb{E}[\psi(t)]$ and the empirical $\psi(t)$ remain substantially higher than the random walk line




(A) The directedness estimation, $\psi$ for an AR process with $\sigma = 1.5$, $\alpha = 0.99$, and $n = 100$.




(B) The directedness estimation, $\psi$ for an AR process with $\sigma = 0.1$, $\alpha = 0.99$, and $n = 100$.

FIGURE 3.16: Directedness measure for AR processes with differing levels of noise along with the expected $\psi$ value for both this AR process and a random walk. Panel (A) uses high noise ($\sigma = 1.5$), while (B) uses low noise ($\sigma = 0.1$). If the noise is low then the AR process dominates the expected random walk, but if the noise is high then the random walk has a higher $\psi$ than the AR process.

throughout. The decay of $\psi(t)$ is much slower than in figure 3.16b, demonstrating that lower stochasticity allows the autoregressive memory to dominate. When noise is small, the AR(1) process becomes increasingly directed, and the trajectory is nearly linear over its initial portion. The process retains long-range directional alignment, yielding sustained high values of $\psi(t)$, well above the baseline.

These results reinforce the idea that an AR process—even with moderate to high noise — produces trajectories with non-trivial directionality, and thus constitutes a meaningful baseline for comparison with deep learning training dynamics. In particular, if a training run exhibits $\psi(t)$ consistently below the RW reference, this

suggests excess curvature, oscillation, or lack of alignment—potentially due to poor conditioning, optimiser instability, or interaction with the loss landscape. If, however, it exceeds the RW baseline, that may indicate strong alignment or phase transitions in optimisation, *e.g.* during LR drops.

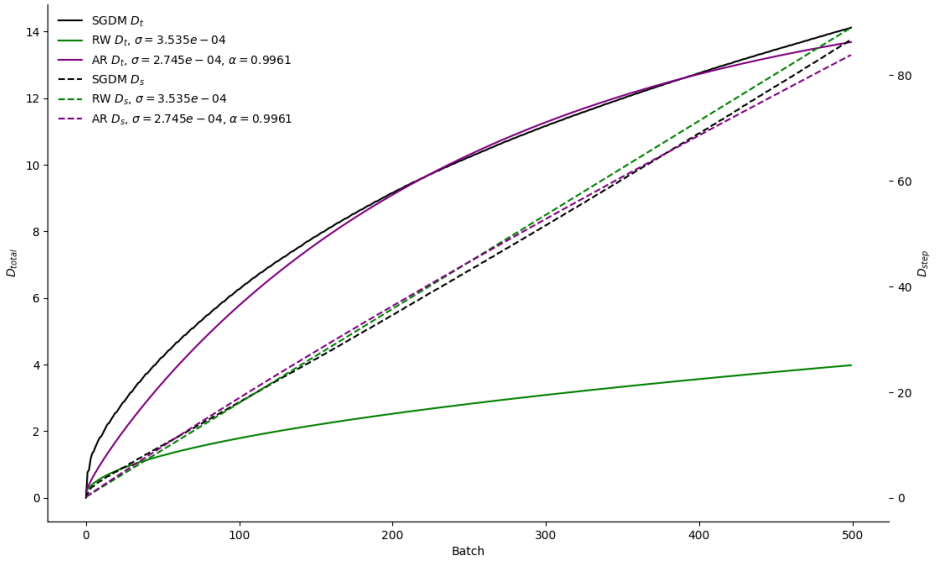
### 3.2.7 Deep Network Optimisation as an AR Process

Now we can also fit an AR process to the same deep learning optimisation process as in figures 3.12a and 3.12b and compare the results. To do this we again minimise the sum of the mean squared errors of the estimated $D_s$ and $D_t$ values from the real values, this time using L-BFGS-B. We plot the results in figures 3.17a and 3.17b, and we can see that the AR process is a much better fit than the random walk in terms of the magnitude of the values, as expected.

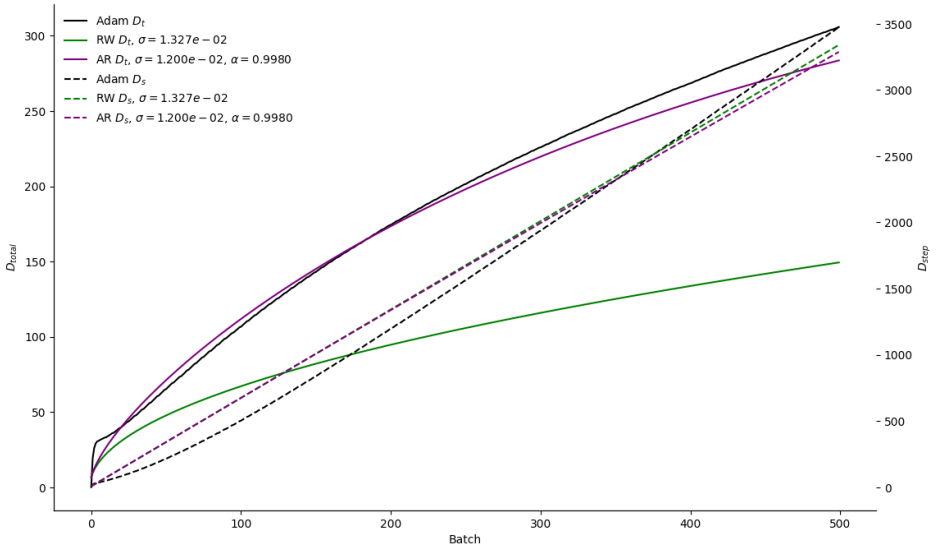
However, the AR process is not a perfect fit, as the shape of the total Euclidian distance is not quite the same between the expected values for an AR process and the values computed during optimisation. We check the match of shape of the total Euclidian distance for a random walk by setting $\gamma = 0$ for our minimisation equation (3.15) and find, figure 3.18 that curve of the total Euclidian distance is better fit by a random walk. This indicates that deep network optimisation is mean-reverting, but it is directed; that is, the optimisation moves in a non-random direction, but never starts oscillating around a point.

To illustrate this first case, let us set up an AR process so that each step takes it instantly to the mean and then applies the noise, *i.e.* $\alpha = 0$. Calculating the limit of $D_s(t)$ as $\alpha$ approaches 0,

$$\begin{aligned}
\lim_{\alpha\to 0} \mathbb{E}\left[D_s(t)\right] &= \lim_{\alpha\to 0} \sum_{k=1}^{t} \sqrt{(\alpha-1)^2\alpha^{2k-2}\|\theta_0-\mu\|_2^2 + \sigma^2 n\left((\alpha-1)\frac{\alpha^{2k-2}-1}{\alpha+1}+1\right)}, \\
&= \lim_{\alpha\to 0} \sqrt{(\alpha-1)^2\alpha^{2-2}\|\theta_0-\mu\|_2^2 + \sigma^2 n\left((\alpha-1)\frac{\alpha^{2-2}-1}{\alpha+1}+1\right)} \\
&\quad + \sum_{k=2}^{t} \sqrt{(\alpha-1)^2\alpha^{2k-2}\|\theta_0-\mu\|_2^2 + \sigma^2 n\left((\alpha-1)\frac{\alpha^{2k-2}-1}{\alpha+1}+1\right)}, \\
&= \lim_{\alpha\to 0} \sqrt{(\alpha-1)^2\alpha^{0}\|\theta_0-\mu\|_2^2 + \sigma^2 n\left((\alpha-1)\frac{\alpha^{0}-1}{\alpha+1}+1\right)} + (t-1)\sigma\sqrt{2n}, \\
&= \sqrt{\|\theta_0-\mu\|_2^2 + \sigma^2 n} + (t-1)\sigma\sqrt{2n}.
\end{aligned}$$




(A) $D_s$ and $D_t$ for SGDM on a ResNet-20 and the expected values for both an AR process and random walk in same dimension.




(B) $D_s$ and $D_t$ for Adam on a ResNet-20 and the expected values for both an AR process and random walk in same dimension.

FIGURE 3.17: Deep network optimisation measures plotted against the expectations for am AR processes.

And for $D_t(t)$,

$$\lim_{\alpha \to 0} \mathbb{E}\left[D_t(t)\right] = \lim_{\alpha \to 0} \sqrt{(1-\alpha^t)^2 \|\mu - \theta_0\|_2^2 + \sigma^2 n \frac{\alpha^{2t}-1}{\alpha^2 - 1}}\,,$$
$$= \sqrt{\|\mu - \theta_0\|_2^2 + \sigma^2 n}\,.$$

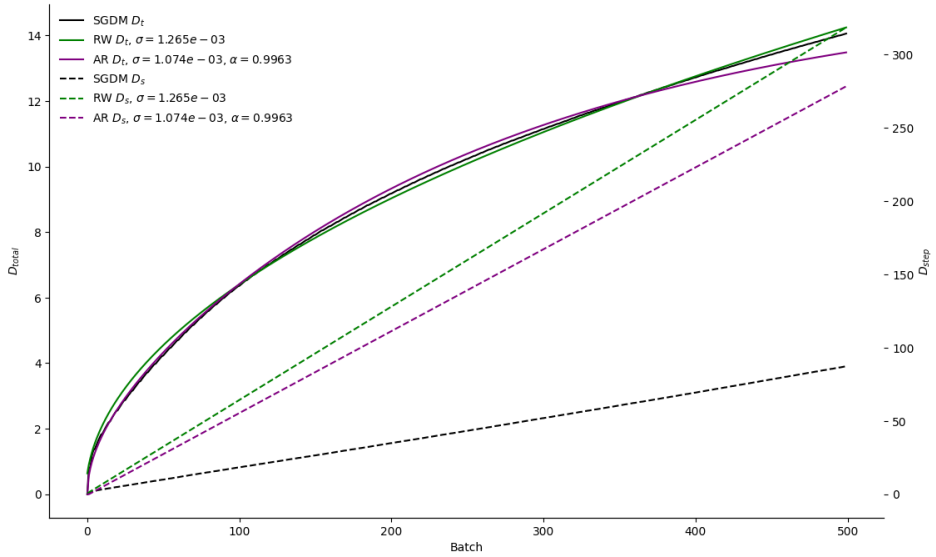


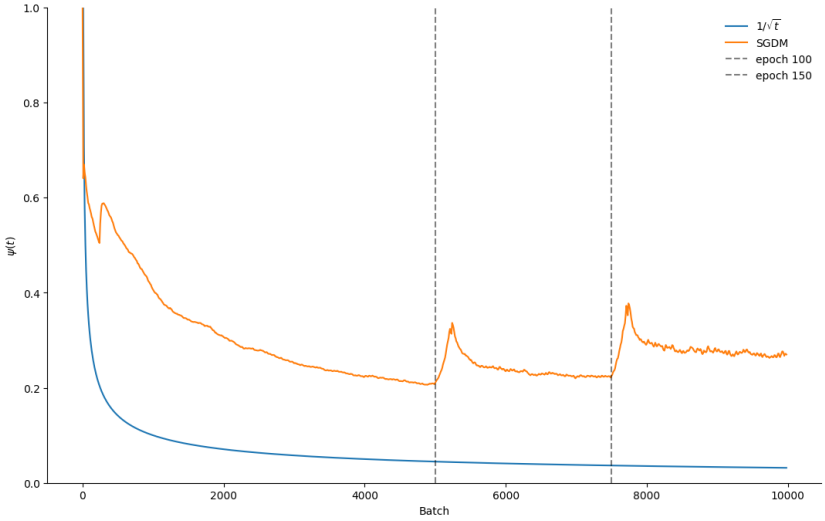

FIGURE 3.18: $D_s$ and $D_t$ for SGDM on a ResNet-20 and the expected values for both an AR process and random walk in same dimension. Curve fitted with $\gamma = 0$.

Then,

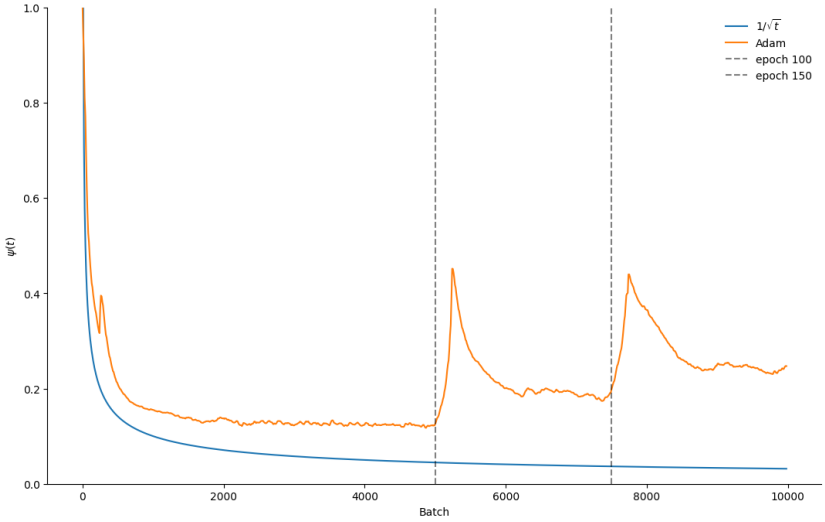
$$\lim_{\alpha \to 0} \frac{\mathbb{E}\left[D_t(t)\right]}{\mathbb{E}\left[D_s(t)\right]} = \frac{\sqrt{\|\mu - \theta_0\|_2^2 + \sigma^2 n}}{\sqrt{\|\theta_0 - \mu\|_2^2 + \sigma^2 n} + (t-1)\sigma\sqrt{2n}},$$

Which is maximised when $\|\mu - \theta_0\|_2^2 \gg (t-1)\sigma\sqrt{2n}$, *i.e.* when the distance to the solution from initialisation is much greater than the noise levels, and minimised *v.v.*

With this understanding, we can add the expected random walk behaviour into our empirical calculation of $\psi$ for neural network training. The $\psi(t)$ metric, figure 3.19, shows that SGDM maintains higher directional efficiency than Adam throughout training, with smoother decay and more consistent progress through weight space. Adam exhibits a sharper early drop in $\psi$, indicating a more quick descent into a minimum and then less coherent movement, but responds more strongly to learning rate drops with brief spikes in efficiency. Both optimisers show $\psi$ well above the random walk baseline, confirming non-trivial alignment in their trajectories; similar to the AR process with low noise. SGDM tends to make steadier, more directed updates, while Adam's adaptivity induces greater curvature sensitivity and local oscillation.




(A) $\psi$ for SGDM on a ResNet-20 trained using a batch size of 1024, learning-rate of 0.05, momentum of 0.85, weight-decay of $10^{-2}$, and a learning rate schedule of $\times 0.1$ at epochs 100, 150 for 200 epochs.




(B) $\psi$ for Adam on a ResNet-20 trained using a batch size of 1024, learning-rate of 0.01, beta one of 0.9, beta two of 0.99, weight-decay of $10^{-4}$, and a learning rate schedule of $\times 0.1$ at epochs 100, 150 for 200 epochs.

FIGURE 3.19: Time evolution of the local directedness efficiency metric $\psi_{250}(t)$ during training of a model with SGDM, over a rolling window of $N = 250$ steps revealing several distinct phases of optimisation geometry; early, mid, and late stages of training and a also the effects of learning rate scheduling. Note the artefacting at batch 250 is just from taking a window that is 250 batches wide.

# IV

# Second order methods

In this chapter we develop some tools to extract second-order information from large networks where it is computationally infeasible to compute the full Hessian, integrate them into our analysis, and use this information to attempt to improve optimisation. We will look at approximating the Hessian with a low-rank version by assuming that the gradients related to each class form the high-loss subspace; next we try and extract approximations to the eigensystem and integrate them into SGD.

## 4.1 Literature review

The dimensionality of the problem is so high we cannot feasibly compute the Hessian, and approximations are also often prohibitively expensive for the information gain.

In optimisation, a general rule of thumb for the problem size for using quasi-Newton's method is in the thousands of parameters, but the smallest model we use in this work has over twenty thousand. For Multi-layered Perceptron (MLP)s of three layers and hundreds of parameters we could conceivably use second-order methods, however, for current models that order in the millions or billions[27] of parameters conventional second-order methods, such as Newton's method are intractable. However, while quasi-Newton's methods have been shown to work on a small network [84], they are also infeasible on a significantly sized model when calculating the second-order information the classical way.

### 4.1.1 The Hessian

The Hessian is the extension of the second derivative to function of multiple variables, as such it is a measure of the curvature of the loss function. The Hessian is the matrix of second partial derivatives of the loss function, and it is a size $p \times p$ matrix where $p$

is the number of parameters in the model. That is,

$$H_{ij} = \frac{\partial^2 \mathcal{L}_\theta}{\partial \theta_i \partial \theta_j} \tag{4.1}$$

Classically, the Hessian can be used to identify the minimum of a function by checking that it is positive-definite when $\nabla f(x) = 0$.

In the context of optimisation, the Hessian can be used to find the minimum of a convex function by using Newton's method. Newton's method is an iterative root finding method,

$$x_{t+1} = x_t - \frac{f(x_t)}{f'(x_t)}, \tag{4.2}$$

solves $f(x) = 0$. Generalising this to find a minimum, *i.e.* $f'(x) = 0$, and to a loss function of multiple variables, $\mathcal{L}_\theta$, we have

$$\theta_{t+1} = \theta_t - H^{-1} \nabla \mathcal{L}_\theta. \tag{4.3}$$

Unfortunately, in the non-convex case, Newton's method can not differentiate between different types of stationary points and so will converge to a saddle points, minimums, or maximums. Extensions to this method have been proposed to ensure the minimisation of a non-convex function, such as the Gauss-Newton[85] method and the Levenberg-Marquardt[86], [87] method. Unfortunately, as the Hessian is a size $p \times p$ matrix, it is infeasible to compute for large models by simply taking the derivatives *wrt.* each parameter.

### 4.1.2 Quasi-Newton methods

A quasi-Newton optimisation method is simply any method that is based of Newton's method and uses an approximation to the Hessian. That is, they still assume that the function can be approximated by a quadratic near the minimum, but they do not require the Hessian to be computed.

Since the optimisation methods are iterative, the Hessian approximation is usually updated at each iteration based off the new gradient vector. Quasi-Newton methods generalise the secant method [31] as they compute the Hessian approximation, $B_k$, from the secant equation,

$$B_{k+1}(\theta_{k+1} - \theta_k) = \nabla f_{k+1} - \nabla f_k, \tag{4.4}$$

$$B_{k+1} s_k = y_k, \tag{4.5}$$

where $s_k$ is the displacement, and $y_k$ is the change in the gradient. The secant equation requires $B_{k+1}$ to be symmetric positive definite which is not guaranteed in a non-convex optimisation problem, only if $s_k^\top y - k > 0$. To fix this we calculate some

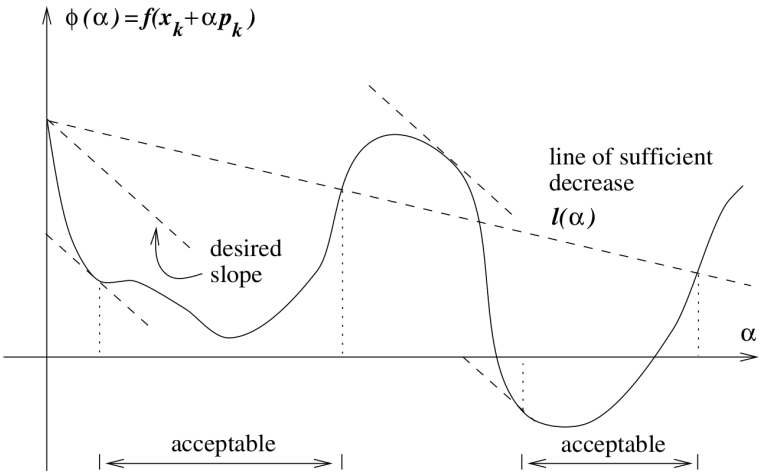



FIGURE 4.1: The Wolfe conditions. Taken from Nocedal and Wright [31].

step size $\alpha$ via a line search in the direction $p_k = -B_k^{-1}\nabla f_k$ such that the Wolfe conditions are satisfied.

The Wolfe conditions [88], [89],

$$f(x_k + \alpha_k p_k) \leq f(x_k) + c_1 \alpha_k \nabla f_k^T p_k, \tag{4.6}$$

$$\nabla f(x_k + \alpha_k p_k)^T p_k \geq c_2 \nabla f_k^T p_k, \tag{4.7}$$

$$\text{with } 0 < c_1 < c_2 < 1. \tag{4.8}$$

are a pair of requirements, one on the gradient, and one on the curvature, that ensure that the step size is sufficient but not too large, figure 4.1. Firstly, the Armijo condition — that the step size is not too large, says that the step must reduce the loss function by an amount at least proportional to the step size; and secondly, the curvature condition — that the step size is sufficiently large, says that the step must increase the gradient by at least some proportion.

The secant equation is under-determined, *i.e.* has multiple solutions, and so quasi-Newton methods must choose how they constrain the solution, typically by adding a simple low-rank update to the current estimate of the Hessian by choosing the solution $B_{k+1}$ that is closest to the current estimation.

Another advantage of quasi-Newton methods is that they can be used to approximate instead of computing the Hessian and then inverting it, the inverted matrix can usually be approximated directly. The most common quasi-Newton algorithms are the Symmetric Rank-1 algorithm (SR1), the Broyden-Fletcher-Goldfarb-Shanno algorithm (BFGS), and its low-memory extension Limited-memory Broyden-Fletcher-Goldfarb-Shanno algorithm (L-BFGS).

#### 4.1.2.1 BFGS, L-BFGS, and SR1

The BFGS closely followed the original quasi-Newton method, the Davidon-Fletcher-Powell algorithm (DFP) method, and was developed by Broyden [90], Fletcher [91], Goldfarb [92], and Shanno [93]. This simultaneous discovery happened because BFGS is almost identical to DFP, instead of requiring $B_k$ to be symmetric positive definite, BFGS requires $H_k = B_k^{-1}$ to be symmetric positive definite and satisfy the secant equation *mutatis mutandis*,

$$H_{k+1} y_k = s_k. \tag{4.9}$$

The motivation for this is that the secant condition ensures that the Hessian approximation matches at least one observed curvature pair. BFGS builds the best possible $H_{k+1}$, in the least-change sense, that satisfies this. The BFGS method is the most popular quasi-Newton method, and is the default method in many optimisation libraries, *e.g.* SciPy [94] and MATLAB [95]; algorithm 2 presents the necessary steps for computing minimums with the BFGS.

**Algorithm 2** The BFGS method for unconstrained optimisation.

**Require:** $\theta_0$ ▷ Starting point
**Require:** $H_0$ ▷ Initial approximation to the Hessian, often the identity matrix
**Require:** $\epsilon > 0$ ▷ convergence tolerance
  $k \leftarrow 0$
  **while** $\|\nabla f_k\| > \epsilon$ **do**
    $\rho_k \leftarrow -H_k \nabla f_k$ ▷ Search direction
    $\alpha_k \leftarrow$ line search$(\theta_k, p_k)$ ▷ Find $\alpha_k$ that satisfies the Wolfe conditions
    $\theta_{k+1} \leftarrow \theta_k + \alpha_k \rho_k$ ▷ Update the parameters
    $s_k \leftarrow \theta_{k+1} - \theta_k$
    $y_k \leftarrow \nabla f_{k+1} - \nabla f_k$
    $H_{k+1} \leftarrow (I - \rho_k s_k y_k^\top) H_k (I - \rho_k y_k s_k^\top) + \rho_k s_k s_k^\top$ ▷ Update the inverse Hessian approximation
    $k \leftarrow k + 1$
  **end**

BFGS decides the secant equation by choosing the solution that is closest to the current estimate,

$$\min_B \|B - B_k\|,$$
$$\textit{subject to } B = B^\top \textit{ and } B s_k = y_k,$$

yielding the update equation in algorithm 2. When transformed via the Sherman-Morrison formula, this update equation becomes,

$$B_{k+1} = B_k - \frac{B_k s_k s_k^\top B_k}{s_k^\top B_k s_k} + \frac{y_k y_k^\top}{s_k^\top y_k}, \tag{4.10}$$

which is a rank-two update to the current estimate of the Hessian.

The convergence of BFGS has been extensively studied. In the convex setting, it has been shown that BFGS with exact line search converges super-linearly when applied to twice continuously differentiable functions with Lipschitz-continuous gradients [31], [96]. The seminal result by Powell [97] proved that the BFGS method generates a sequence $\{\theta_k\}$ converging to a minimizer of $f$, and that the convergence is super-linear provided that the sequence $\{H_k\}$ remains bounded and that $f$ is strongly convex in a neighbourhood of the minimizer. In the non-convex case, more recent results [98] show that BFGS retains global convergence properties under weaker assumptions when combined with inexact line search conditions. Moreover, when used with safeguards e.g. damped BFGS[99], it avoids generating indefinite Hessian approximations and maintains descent directions even in non-convex landscapes.

The SR1 method asks if there is a rank-one update, defined as $B_{k+1} = B_k + \sigma vv^\top$, that satisfies the secant equation. Substituting this into the secant equation gives,

$$B_{k+1} = B_k + \frac{(y_k - B_k s_k)(y_k - B_k s_k)^\top}{(y_k - B_k s_k)^\top s_k}. \tag{4.11}$$

This update does not guarantee that if $B_k$ is positive definite then $B_{k+1}$ is also positive definite, and so the SR1 method is not as popular as BFGS for line search methods, however, it has been shown useful in trust-region methods. SR1 also has another drawback, the denominator can be close to zero indicating that there is no rank-1 update that satisfies the secant equation. Nevertheless, there are strategies to prevent this and SR1 remains a popular method.

The L-BFGS method is a low-memory version of BFGS that does not require the full $p \times p$ matrix to be stored, instead it stores the last $m$ pairs of $s_k$ and $y_k$ and uses them to approximate the Hessian. This method is useful for large problems where the Hessian is infeasible to store. L-BFGS follows the same algorithm as BFGS but uses algorithm 3 to compute the Hessian-gradient product, where the $H_k^0$ is a new initialisation of the Hessian at each iteration, often set to $H_k^0 = \gamma_k \boldsymbol{I}$, where

$$\gamma_{k+1} = \frac{s_k^\top y_k}{y_k^\top y_k}.$$

### 4.1.3 Diagonal approximation

Some of the aforementioned methods rely on the inverse of the Hessian; this is much easier to compute if the Hessian is approximated as a diagonal matrix, *i.e.* one where all the off-diagonal are zero, $H_{i\neq j} = 0$. This can be done in $O(p)$ time [39, Chapter 5.4.1] instead of $O(p^2)$, and is a common approximation [100], [101] since

**Algorithm 3** The L-BFGS update.

**Require:** $H_k^0$ ▷ Initial approximation to the Hessian
**Require:** $m$ ▷ Number of pairs to store
  $q \leftarrow \nabla f_k$
  $\alpha_i \leftarrow$ array of length $m$
  $\rho_i \leftarrow$ array of length $m$
  **for** $i = k - 1$ to $k - m$ **do**
    $\alpha_i \leftarrow \rho_i s_i^\top q$
    $q \leftarrow q - \alpha_i y_i$
  $r \leftarrow H_k^0 q$
  **for** $i = k - m$ to $k - 1$ **do**
    $\beta \leftarrow \rho_i y_i^\top r$
    $r \leftarrow r + s_i(\alpha_i - \beta)$
  $H_{k+1}\nabla f_k \leftarrow r$

neural networks have a large amount of parameters. Unfortunately, the approximation that the off-diagonal elements are zero is not always accurate enough to make this approximation useful.

### 4.1.4 Hessian-vector product

A common use of the Hessian is to multiply it by a vector, since storing and calculating a large matrix like the Hessian is difficulty, a different method for directly computing the Hessian-vector product is used instead. This is done by noting that

$$v^\top H = v^\top \nabla(\nabla \mathcal{L}_\theta) \tag{4.12}$$

Pearlmutter [102] denote the operation $v^\top \nabla$ as $\mathcal{R}\{\cdot\}$, and show that it can be computed in $O(p)$ time. In addition, they show that the inverse Hessian can be computed by minimising $\|Hv - b\|_2^2$, giving $v = H^{-1}b$; or if the Hessian is positive definite, $\frac{1}{2}v^\top Hv + v \cdot b$.

Yao *et al.* [103] show another method to compute $Hv$, and also provide a package for PyTorch [104] to do so on neural networks. They do this by calculating the derivative of the inner product of the gradients and the vector, $v$,

$$\frac{\partial g^\top v}{\partial \theta} = \frac{\partial g^\top}{\partial \theta} v + g^\top \frac{\partial v}{\partial \theta} = \frac{\partial g^\top}{\partial \theta} v = Hv,$$

via the chain rule and the independence of $v$ and $\theta$. The product of the gradients, $g$ and $v$ is a scalar, and so the derivative of this with respect to $\theta$ can be computed in one pass using reverse accumulation automatic differentiation.

Martens *et al.* [105] call their use of this technique a Hessian-free optimisation method since the Hessian is never explicitly computed; their method is an inexact

quasi-Newton's method using a damped Hessian and a Conjugate Gradient (CG) inner optimisation step. Inexact — the CG method is stopped before convergence; and damped — the Hessian is regularised by adding a multiple of the identity matrix to it via a function utilising the Hessian-vector product, $B(x) = (H + \lambda \boldsymbol{I})x = Hx + \lambda x$ stopping the CG method moving out of the region being well-approximated by the quadratic. This method is verified on an auto-encoder on MNIST [34].

### 4.1.5 Lanczos algorithm

The Lanczos algorithm [106] is a method that helps to compute the $k$ most extreme eigenvalues of a large, symmetric matrix. Given a symmetric matrix $A$, a number of iterations, $m$, the Lanczos algorithm, algorithm 4, computes the $m \times m$ tridiagonal matrix $T = V^\star AV$ where $V$ is an orthonormal $n \times m$ matrix.

**Algorithm 4** Lanczos algorithm [107, Algorithm 6.5] for Hermitian matrices.

**Require:** $A \in \mathbb{R}^{n \times n}$
**Require:** $m \in \mathbb{N}^+$
  $v_0 \leftarrow 0$
  $\beta_0 \leftarrow 0$
  $v_1 \leftarrow$ random vector with $\|v_1\| = 1$
  **for** $i = 1$ to $m$ **do**
    $w_i \leftarrow Av_i - \beta_i v_{i-1}$
    $\alpha_i \leftarrow \langle w_i, v_i \rangle$
    $w_i \leftarrow w_i - \alpha_i v_i$
    $\beta_{i+1} \leftarrow \|w_i\|_2$
    $v_{i+1} \leftarrow w_i / \beta_{i+1}$ ▷ Normalised $w_i$
  $V \leftarrow [v_1, v_2, \ldots, v_m]$
  $T_{ii} \leftarrow \alpha_i$ and $T_{i,i\pm1} \leftarrow \beta_i$ ▷ $\alpha$ on the diagonal, $\beta$ on the outer bands.

Whilst this algorithm does not compute the eigenvalues itself, it does transform the eigen-decomposition problem of $A$ into the easier decomposition of $T$. If $\lambda$ is an eigenvalue of $A$ and $x$ is an eigenvector of $T$ also with eigenvalue $\lambda$, then

$$\begin{aligned} Tx &= \lambda x \\ V^\star AVx &= \lambda x \\ AVx &= \lambda Vx \\ Ay &= \lambda y \end{aligned}$$

that is, we can transform the eigenvectors of $T$ into the Ritz vectors of $A$, *i.e.* the approximate eigenvectors, by left-multiplying by $V$. This simplifies the problem in two ways, firstly there exist specialised algorithms for decomposing tridiagonal matrices or other algorithms are known to converge faster, *e.g.* the QR algorithm, and secondly, as $m << n$ the matrix to find the eigenvalue for is much smaller.

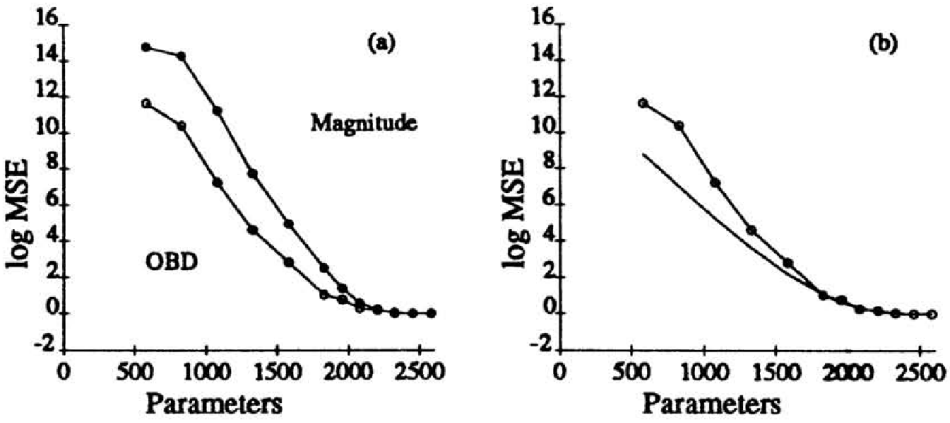



FIGURE 4.2: Optimal brain damage: the diagonal of the Hessian is used to prune the network. Taken from LeCun *et al.* [101].

Note that this method works extremely well with the Hessian-vector product trick, as the Lanczos algorithm only truly requires the matrix-vector product, $Av_i$ and not $A$ itself. This version, algorithm 4, has numerical stability issues due to the normalisation so a different form is used in practice, although the method remains the same.

The convergence properties of the Lanczos algorithm are closely linked to polynomial approximation theory, the Ritz values, approximated eigenvalues, converge rapidly to the extremal eigenvalues of the matrix, especially when the spectrum is well separated. Convergence is typically super-linear for the extreme eigenvalues and stagnates for interior ones. A rigorous convergence analysis is given by Paige [108], who showed that in exact arithmetic, the Lanczos method produces an orthonormal basis of the Krylov subspace and that the eigenvalues of the resulting tridiagonal matrix, Ritz values, converge to the extremal eigenvalues of the original matrix. Further results, including the role of the Chebyshev polynomial basis in explaining the convergence rate, are covered in [107], [109]. In practice, numerical round-off error leads to loss of orthogonality, which affects convergence and requires reorthogonalisation strategies or restarting schemes, as discussed by [110], [111].

## 4.2 Literature review — Application to neural networks

In neural networks, the Hessian has been used to create faster optimisers [112]. Whilst, LeCun *et al.* [101] identify the least significant parameters of a network in order to prune them — figure 4.2; they measure a parameter's saliency by approximating the change in loss for this perturbation of the weight vector by a Taylor expansion, parameters that do not change the loss much can be pruned without damaging the network.

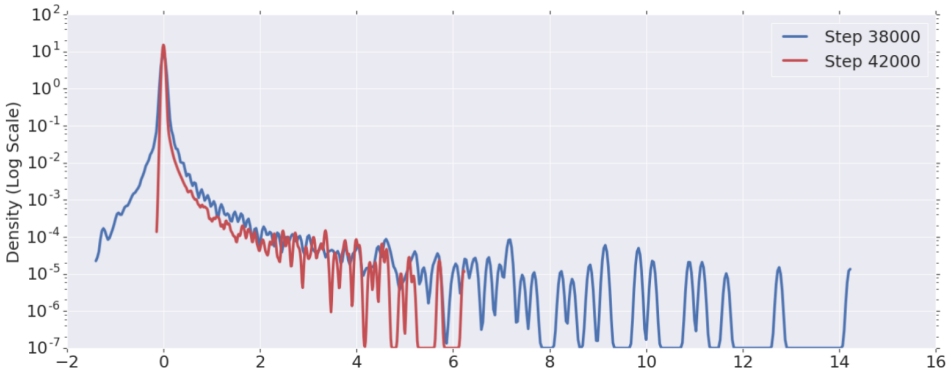



FIGURE 4.3: "Spectral densities of Resnet-32 preceding and following a learning rate decrease (at step 40000). The Hessian prior to the learning rate drop appears sharper." Taken from Ghorbani *et al.* [81].

It is also heavily used in Bayesian neural networks, where the Hessian is used to approximate the posterior distribution of the weights [113] via the Laplacian.

### 4.2.1 Eigensystem analyses

To study the curvature and roughness of the loss function during training Ghorbani *et al.* [81] estimate the density of the eigenvalues. To produce their eigen-density curve, $\phi(t)$, they use the Lanczos method to compute the eigenvalues of the Hessian via the Hessian-vector trick, and then calculate the average over the Gaussian-relaxed Dirac delta distributions for each eigenvalue,

$$\phi(t) = \frac{1}{n}\sum_{i=1}^{n}\delta(t-\lambda_i), \tag{4.13}$$

"relax[ing] the problem by convolving with a Gaussian density of variance $\sigma^2$",

$$\phi_\sigma(t) = \frac{1}{n\sigma\sqrt{2\pi}}\sum_{i=1}^{n}\exp\left(-\frac{(t-\lambda_i)^2}{2\sigma^2}\right). \tag{4.14}$$

They then evaluate this density curve at several points during training to see how the curvature of the loss function changes. For instance, they find that the eigenvalues of the Hessian are sharper before a learning rate decrease, figure 4.3, that batch normalisation reduces large outliers, and that residual connections increase the eigenvalues. This last finding is in opposition to Li *et al.* [70], previously mentioned in section 3.1.3, who produced visualisations that showed that residual connections increase the smoothness of the loss function. Although the Hessian-vector trick provides a drastic speed-up, Ghorbani *et al.* [81] still find that this computation is the bottleneck of their method as for "a Resnet-18 on ImageNet, running a single draw takes about half the time of training the model"[81].

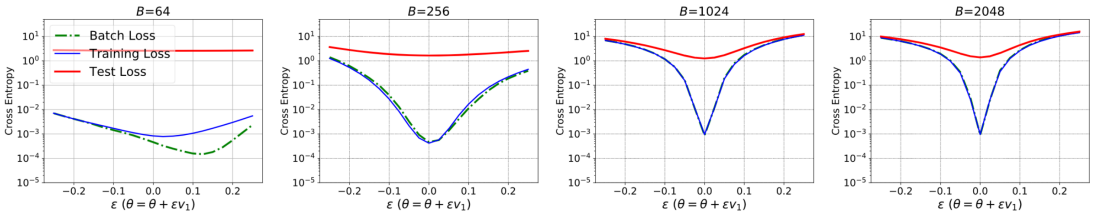



FIGURE 4.4: "The landscape of the loss is shown along the dominant eigenvector, $v_1$, of the Hessian for C1 on CIFAR-10 dataset. Here $\epsilon$ is a scalar that perturbs the model parameters along $v_1$" Taken from Yao *et al.* [114].

Other studies compute the exact Hessian by backpropagating twice; Yao *et al.* [114] use this to analyse the Hessian spectrum to compare large and small sized mini-batch training. In doing this they show "that despite the arguments regarding prevalence of saddle-points plaguing optimization [...], that is actually not the problem with large batch size training, even when batch size is increased to the gradient descent limit". Additionally, they show that large mini-batch training has a sharper loss landscape than small mini-batch training, and that this is consistent on both the MNIST and CIFAR-10 datasets. Figure 4.4 shows the landscape of the loss along the dominant eigenvector of the Hessian reinforcing the greater sharpness of large mini-batch optimisation.

### 4.2.2 Saddle points and stationary points

Dauphin *et al.* [115] try to validate Bray and Dean's [116] calculation that the majority of stationary points are saddle points in high dimensions for neural networks. Bray and Dean [116] show that the eigenvalues of the Hessian at critical points are distributed as a Wigner semicircle [117] that is shifted proportional to the error. Global minima are shifted so far that all the eigenvalues are positive and saddle points are shifted so that some are positive and some are negative. As the error increases, the semicircle shifts to the right, and so saddle points become more prevalent with more negative eigenvalues; since the distribution is a semicircle these saddle points also have a large amount of zero eigenvalues indicating large plateau surrounding these points. If these plateaux exist in neural network training then they could slow the convergence of the optimisation and make it appear to have converged.

Whilst Dauphin *et al.*'s [115] results are favourable to this theory, they are not conclusive as they are only "interested in [...] a single layer MLP [...] on a down-sampled version of MNIST and CIFAR-10" [115]. Both datasets were down-sampled to $10 \times 10$ pixels, additionally they train a model with seven hidden layers to showcase the SGD being caught in a plateau and then their saddle-free method successfully navigating itself out of the same loss region. Seven hidden layers can be considered deep, but it is the number of parameters rather than the depth of the network that makes the Hessian infeasible, this number is absent but likely very

small. For the experimental validation of the prevalence of saddle points, their use of a single-layer MLP on scaled-down MNIST is unconvincing for today's networks, especially compared to more recent studies, *e.g.* the aforementioned Yao *et al.* [114].

Moreover, even though the motivation of this saddle-free Newton method is that in high-dimensional space the vast majority of stationary points are saddle points and not local minima, this might not be the case for the sub-space that is optimised over. This result is for the complete weight-space, whereas, in reality, it is trivial to enter a low-loss subspace with any optimiser and hyper-parameter set and the stationary points are more likely to be minima when the loss is low [116]. That is not to say that there is not a proliferation of saddle points in the low-loss subspace, but simply that the region where the most challenging part of the optimisation happens may have different properties to the whole space.

## 4.3 High-curvature directions

This section introduces a quasi-Newton's method where we combine statements from the literature about the Hessian eigenstructure seen during training and how this relates to the clustering of gradients of a class and show that the various assumptions in these, in aggregate, do not hold up

Classification problems using deep learning have been shown to have a high-curvature subspace in the loss landscape equal in dimension to the number of classes. Moreover, this subspace corresponds to the subspace spanned by the logit gradients for each class. An obvious strategy to speed up optimisation would be to use Newton's method in the high-curvature subspace and stochastic gradient descent in the co-space. We show that a naive implementation actually slows down convergence, and we speculate why this might be.

### 4.3.1 Second-order methods

Directions of high curvature limit the step size that can be taken in gradient descent. To see this, consider a quadratic minimum defined by a Hessian, $H$,

$$f(\theta) = \frac{1}{2}(\theta - \theta^*)^\mathsf{T} H\,(\theta - \theta^*). \tag{4.15}$$

We can write the Hessian matrix as its eigen-decomposition

$$H = \sum_i \lambda_i v_i\, v_i^\mathsf{T}, \tag{4.16}$$

where $\lambda_i$ is a measure of the curvature in the direction $v_i$. For $\theta^*$ to be a minimum we require $\lambda_i > 0$ for all $i$. If we perform gradient descent,

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla f(\theta^{(t)}) = \theta^{(t)} - \eta H (\theta^{(t)} - \theta^*), \tag{4.17}$$

then

$$\theta^{(t+1)} - \theta^* = (\boldsymbol{I} - \eta H) \left(\theta^{(t)} - \theta^*\right) = (\boldsymbol{I} - \eta H)^t \left(\theta^{(1)} - \theta^*\right), \tag{4.18}$$

$$= \sum_i v_i (1 - \eta \lambda_i)^t v_i^\mathsf{T} \left(\theta^{(1)} - \theta^*\right). \tag{4.19}$$

Thus, gradient descent will diverge exponentially fast unless $\eta \leq 2/\lambda_{\max}$.

Second-order methods are a form of preconditioned gradient methods using second derivatives; these methods work well for ill-conditioned problems as the Hessian "automatically normalize[s] the ill-conditioned problem by stretching and contracting" [118] the directions corresponding to its eigenvectors. However, the difficulty in calculating accurate second-order information holds this class of methods back from improving general deep learning optimisation. If there are only a small number of directions in weight space with high curvature, then we can precondition the gradient in only those directions by using a low-rank Hessian, since the low-rank version is an approximation of the true Hessian this is a quasi-Newton's method.

### 4.3.2 Logit gradients

Gur-Ari *et al.* [82] show that gradient descent is mostly contained in a small subspace, so although the problem is high-dimensional, the optimisation is limited by a low-dimensional high-curvature subspace. Intuitively, we can think that classifiers are made up of some separator and then a linear discriminator — the last fully-connected layer — such that the separator's job is to push the representations of the different classes away from each other in latent space; this would give us $C$ significant, and thus high-curvature, directions. In practice, the amount of curvature fades away rather than dropping suddenly after $C$ directions. That subspace, spanned by the logit gradients, intersects with the top-$C$ eigenvectors of the Hessian [119], [120]. If there are some identifiable directions in weight space with high curvature, then we can significantly speed up optimisation by performing Newton's method in this high-curvature subspace and a standard first-order method in the low-curvature dual subspace.

We propose a method that uses this overlap of class gradients and high-curvature directions to compute an approximate Hessian from the class gradients

### 4.3.3 Setting

Consider a classification problem with data, $\mathcal{D} = \{(x^\mu, y^\mu) \mid \mu \in [1, N]\}$ where $y^\mu$ is a $C$-dimensional one hot vector, as in Fort and Ganguli [121].

We will look at some deep network with parameters, $\theta$, that calculates logits, $z_k^\mu$, which gives class probabilities, $p_k^\mu$, via a softmax

$$p_k^\mu = \frac{\exp(z_k^\mu)}{\sum_{\ell=1}^C \exp(z_\ell^\mu)}, \tag{4.20}$$

and train this network with a cross-entropy loss, $\mathcal{L}^\mu(\theta)$ for an input $(x^\mu, y^\mu) \in \mathcal{D}$.

$$\mathcal{L}^\mu(\theta) = -\sum_{k=1}^C y_k^\mu \log(p_k^\mu) = \sum_{k=1}^C y_k^\mu \left( \log\left( \sum_{\ell=1}^C \exp(z_\ell^\mu) \right) - z_k^\mu \right). \tag{4.21}$$

The mini-batch loss is

$$\mathcal{L}(\theta) = \frac{1}{|\mathcal{B}|} \sum_{\mu \in \mathcal{B}} \mathcal{L}^\mu(\theta) = \frac{1}{|\mathcal{B}|} \sum_{\mu \in \mathcal{B}} \sum_{k=1}^C y_k^\mu \left( \log\left( \sum_{\ell=1}^C \exp(z_\ell^\mu) \right) - z_k^\mu \right). \tag{4.22}$$

Then, for any single parameter $\theta_\alpha$ we have its gradient, $g_\alpha^{\mathcal{B}}$, for some mini-batch

$$g_\alpha^{\mathcal{B}} = \frac{\partial \mathcal{L}(\theta)}{\partial \theta_\alpha} = \frac{1}{|\mathcal{B}|} \sum_{\mu \in \mathcal{B}} \sum_{k=1}^C (p_k^\mu - y_k^\mu) \frac{\partial z_k^\mu}{\partial \theta_\alpha}, \tag{4.23}$$

and so the mini-batch Hessian, $H^\beta$, has components

$$\begin{aligned} H_{\alpha\beta}^{\mathcal{B}} = \frac{\partial^2 \mathcal{L}(\theta)}{\partial \theta_\alpha \partial \theta_\beta} = & \frac{1}{|\mathcal{B}|} \sum_{\mu \in \mathcal{B}} \sum_{k=1}^C \sum_{\ell=1}^C p_k^\mu \left( \delta_{k\ell} - p_\ell^\mu \right) \frac{\partial z_k^\mu}{\partial \theta_\alpha} \frac{\partial z_\ell^\mu}{\partial \theta_\beta} \\ & + \frac{1}{|\mathcal{B}|} \sum_{\mu \in \mathcal{B}} \sum_{k=1}^C (p_k^\mu - y_k^\mu) \frac{\partial^2 z_k^\mu}{\partial \theta_\alpha \partial \theta_\beta} \end{aligned} \tag{4.24}$$

**Assumption 4.1.** *- The logit curvatures, $\frac{\partial^2 z_k^\mu}{\partial \theta_\alpha \partial \theta_\beta}$, are small compared to the logit gradients, $\frac{\partial z_k^\mu}{\partial \theta_\alpha}$.*

There is both empirical and theoretical evidence for assumption 4.1 — that the curvatures of the logits, $\frac{\partial^2 z_k^\mu}{\partial \theta_\alpha \partial \theta_\beta}$, are very small; small enough that the first term of equation (4.24), with the logit gradients, dominates the second term, with the logit curvature. Empirically, [122] show that a small neural network has an eigenspectrum that is a bulk distribution of small eigenvalues, and $C$ large outlier eigenvalues. Furthermore, [81] demonstrate that the spectrum of the logit gradient term alone is similar to the outliers of the Hessian eigenvalues and that the spectrum of the logit curvatures term is similar to the low-values bulk eigenvalues of the Hessian. This

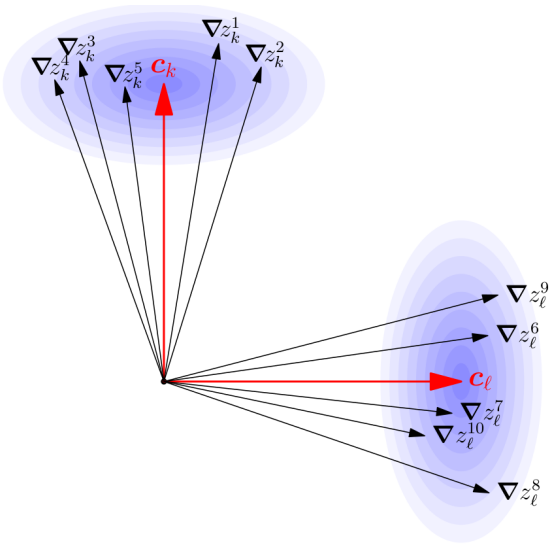



FIGURE 4.5: Illustration of clustered class gradients showing that they cluster more tightly with other gradient vectors from the same class than they do with gradients from a different class.

bulk and outlier structure is well understood in random matrices, [123], if we have the sum of a low-rank matrix with a full-rank matrix whereby the scale of the eigenvalues of the low-rank matrix are large compared to the full-rank matrix; then the sum will follow this bulk and outlier structure. Theoretical justification of this structure comes from Neural Tangent Kernels [124] where their training regime is shown to only depend linearly on the gradients [125], *i.e.* the curvature of the logits is exactly 0. This is only the case for infinite width networks, however, given the large widths of the networks that are used in practice, it is sensible to expect the first term in equation (4.24) to dominate the second. Thus, we will continue with an approximation of the Hessian by dropping the second term.

#### 4.3.3.1 Class gradients

Fort and Ganguli [121] define the class gradient as

$$c_k = \frac{1}{|\mathcal{D}_k|} \sum_{\mu \in \mathcal{D}_k} \nabla z_k^\mu, \tag{4.25}$$

where $\mathcal{D}_k$ is the set of training examples belonging to class $k$ and claim that these are highly correlated, figure 4.5 illustrates what this might look like.

The logit gradients can be decomposed as the class gradients plus some error for that batch,

$$\nabla z_k^\mu = c_k + \epsilon_k^\mu. \tag{4.26}$$

Substituting the logit gradient decomposition into the Hessian we get

$$H \approx \sum_{k=1}^{C} \left( \frac{1}{|\mathcal{B}|} \sum_{\mu \in \mathcal{B}} p_k^\mu (1 - p_k^\mu) \right) c_k c_k^\mathsf{T} + \frac{1}{|\mathcal{B}|} \sum_{\mu \in \mathcal{B}} \sum_{k=1}^{C} \sum_{\ell=1}^{C} p_k^\mu \left( \delta_{k\ell} - p_\ell^\mu \right) \epsilon_k^\mu \, \epsilon_\ell^\mu. \tag{4.27}$$

**Assumption 4.2.** - *The error term in equation* (4.27) *is sufficiently small.*

Here, we state the assumption that the error term in equation (4.27) is negligible. In other words, the epsilon term in equation (4.26) is difference of a specific logit from the mean logit for that class; for this to be small, the logit gradients must be tightly clustered. As long as this error term is sufficiently small the eigensystem of the Hessian remains sufficiently unchanged [126], and, following Fort and Ganguli [121], we ignore the epsilon term. [121] provide empirical evidence for this by measuring the cosine similarity between all pairs of different logits of the same class and showing a high similarity; additionally, [127] also cluster the logit gradients showing "that, empirically, the variation within each cluster is small compared to the variations between the clusters".

Thus, a low-rank approximation of the Hessian is the first term in equation (4.27), as it is the sum of the outer product of the $C$ class mean vectors, now denoted $H^{\text{Low Rank}}$.

### 4.3.4 Quasi-Newton's method

**Assumption 4.3.** - *The class gradients, $c_k$, are orthogonal — i.e. their inner product is 0, $\langle c_k, c_l \rangle = 0$.*

Taking assumption 4.3 — that the class gradients are orthogonal we have

$$H^{\text{Low Rank}} = \sum_{k=1}^{C} \lambda_k v_k v_k^\mathsf{T}, \tag{4.28}$$

where $v_k = c_k / \|c_k\|$ and

$$\lambda_k = \frac{1}{|\mathcal{B}|} \sum_{\mu \in \mathcal{B}} p_k^\mu (1 - p_k^\mu) \|c_k\|^2. \tag{4.29}$$

As this is a sum of non-negative elements it is positive definite which is a necessary condition for Newton's method to move downhill. As $H^{\text{Low Rank}}$ is singular it has no inverse, however, we can define a generalised inverse

$$\left(H^{\text{Low Rank}}\right)^\dagger = \sum_{k=1}^{C} \frac{1}{\lambda_k} v_k v_k^\mathsf{T}, \tag{4.30}$$

that is, $x = (H^{\text{Low Rank}})^{\dagger} a$ is the vector with minimum 2-norm such that $H^{\text{Low Rank}} x = a$.

This inverse can then be used to perform Newton's method in the high-curvature subspace,

$$\theta' \leftarrow \theta - (H^{\text{Low Rank}})^{\dagger} g^{\mathcal{B}} = \theta - \sum_{k=1}^{C} \frac{1}{\lambda_k} (v_k^{\mathsf{T}} g^{\mathcal{B}}) v_k. \tag{4.31}$$

However, as information is still learned outside this subspace [121] it is reasonable to perform stochastic gradient descent in the orthogonal subspace as well. To perform stochastic gradient descent in the orthogonal subspace we can use the projection operator

$$\mathbf{P} = \boldsymbol{I} - \sum_{k=1}^{C} v_k \, v_k^{\mathsf{T}} \tag{4.32}$$

applied to the gradient $\theta' \leftarrow \theta - \eta \mathbf{P} g^{\mathcal{B}}$, where $\eta$ is a learning rate which can be higher than normal as we are in a low-curvature sub-space. Then, putting these two steps together, we get the update equation

$$\theta' \leftarrow \theta - \eta g^{\mathcal{B}} - \sum_{k=1}^{C} \left( \frac{1}{\lambda_k} - \eta \right) (v_k^{\mathsf{T}} g^{\mathcal{B}}) \, v_k. \tag{4.33}$$

To lower the stochasticity of the estimation of the eigensystem of $H^{\text{Low Rank}}$ we use an exponential moving average and root-mean-square for the eigenvalues

$$c_k^{(t)} = \frac{1-\gamma}{1-\gamma^t} \sum_{i=1}^{t} \gamma^{t-i} c_k^{\mathcal{B}^{(i)}}, \qquad c_k^{\mathcal{B}^{(t)}} = \frac{1}{|\mathcal{B}_k^{(t)}|} \sum_{\mu \in \mathcal{B}_k^t} \nabla z_k^{\mu}, \tag{4.34}$$

$$\lambda_k^{(t)} = \sqrt{\frac{1-\gamma}{1-\gamma^t} \sum_{i=1}^{t} \gamma^{t-i} (\lambda_k^{\mathcal{B}^{(i)}})^2}, \qquad \lambda_k^{\mathcal{B}^{(t)}} = \frac{1}{|\mathcal{B}^{(t)}|} \sum_{\mu \in \mathcal{B}^{(t)}} p_k^{\mu} (1 - p_k^{\mu}) \left\| c_k^{\mathcal{B}^{(t)}} \right\|^2, \tag{4.35}$$

where $\mathcal{B}_k^{(t)}$ is the set of examples in the $t^{th}$ mini-batch, $\mathcal{B}^{(t)}$, that belong to class $k$. This gives us a hyper-parameter, $\gamma$, which needs to take into account how rapidly the high-curvature subspace changes during learning; Gur-Ari *et al.* [82] show that the subspace does not markedly change during training, thus, we can set $\gamma$ close to one. We initialise $c_k^{(0)} = \mathbf{0}$ but $\lambda_k^{(0)} = 1$ since we use $1/\lambda_k$ in our update.

### 4.3.5 Comparison to SGD

Figure 4.6 shows the typical loss when training a ResNet-9 (6.5M parameters) with SGDM and with our quasi-Newton's method, equation (4.33) on CIFAR-10. We can see that the final performance is similar; however, the quasi-Newton's method is much more volatile, slower to improve in general, and does end on average with lower accuracy. One would expect to be able to use a higher learning rate with the

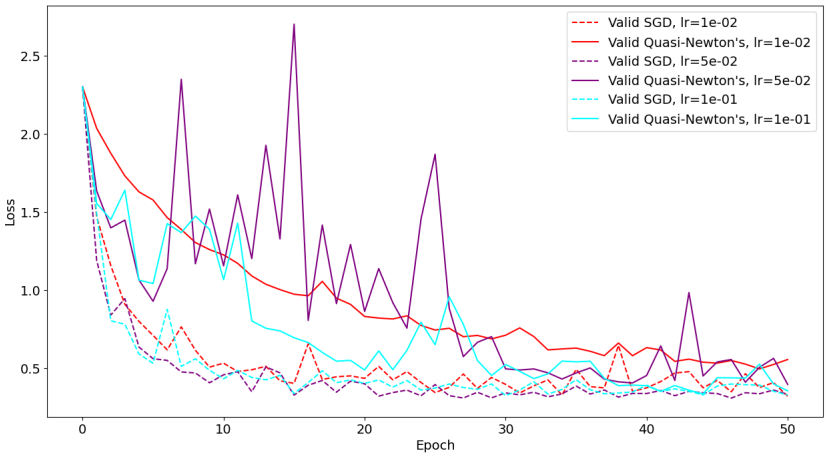



FIGURE 4.6: Validation loss for both SGDM and quasi-Newton's method of a ResNet-9 on CIFAR-10, they were both trained with the same hyper-parameters: batch size of 1024, learning-rate of 0.01, momentum of 0.0, weight-decay of $5 \times 10^{-4}$, for 50 epochs.

quasi-Newton's method as that applies to the low-curvature subspace; however, this is not true, and both methods diverge with $\eta > 10^{-1}$.

### 4.3.6 Validating assumptions

During our derivation of this quasi-Newton's method, we made three assumptions. In this section we validate these assumptions, with our own methods for assumption 4.1 and assumption 4.3, and with the same method as [121] for assumption 4.2.

Firstly, assumption 4.1, that the logit curvature is small. For a network with $P$ parameters we calculate both the mean logit magnitude, $f_k^\mu$, and the mean logit curvature magnitude, $u_k^\mu$, and find their ratio,

$$f_k^\mu = \frac{1}{P}\left|\frac{\partial z_k^\mu}{\partial \theta}\right|_1, \tag{4.36}$$

$$u_k^\mu = \frac{2}{P(P-1)}\left|\text{triu}\left(\frac{\partial^2 z_k^\mu}{\partial \theta^2}\right)\right|_1, \tag{4.37}$$

$$q_k^{\text{LC}} = \frac{1}{N_k}\sum_{\mu \in k}\frac{u_k^\mu}{(f_k^\mu)^2}, \tag{4.38}$$

$$q^{\text{LC}} = \frac{1}{C}\sum_{k=1}^{C} q_k^{\text{LC}}. \tag{4.39}$$

If the said ratio is low, $q^{\text{LC}} \ll 1$, then we can conclude that the logit gradients do dominate the logit curvatures. We can see, figure 4.7, that this ratio is indeed very small, and remains small for the full duration of training. Additionally, the initial logit curvature is higher which is to expected as the initialisation is random, however, even

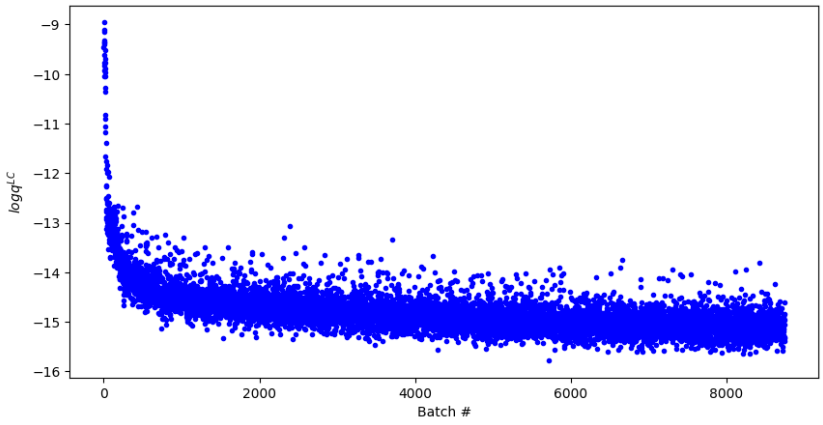



FIGURE 4.7: Graph of the ratio of logit curvature to logits gradients, $q^{\text{LC}}$, plotted for a training run of a ResNet-9 over 50 epochs. The logit curvature metric is plotted on a log scale as the ratio is small. That the ratio is so small indicates that the second differential term is dominated by the first differential term.

this is still small enough to show that the logit gradient term dominates the logit curvature term.

To validate assumption 4.2, we repeat the experiment done in [121] and cluster the logit gradients, that is, we take the mean cosine between logits with the same class,

$$q^{\text{SL}} = \frac{1}{C}\sum_{k=1}^{C}\frac{1}{N(N-1)}\sum_{\mu\neq v}\cos\left(\frac{\delta z_k^{\mu}}{\delta\theta}, \frac{\delta z_k^{v}}{\delta\theta}\right). \tag{4.40}$$

Here we see, as in [121], the logit gradients are clustered per class, figure 4.8; whilst the clustering relaxes during training, the similarity remains high. Additionally, this can be contrasted with the validation for the next assumption — looking at the similarity of the class gradients, equation (4.41), which are lower, figure 4.9, implying that the logit gradients are are more correlated with other logit gradients for data in the same class than they are for logit gradients of different classes.

The final assumption, assumption 4.3 can be validated by calculating the mean cosine similarity between the class gradients of different classes,

$$q^{\text{CG}} = \frac{1}{C(C-1)}\sum_{k=1}^{C}\sum_{l<k}\cos\left(c_k, c_l\right). \tag{4.41}$$

Figure 4.9 shows that the class gradients are not orthogonal and that, due to the high-dimensionality, there are significant overlaps between the mean class logit gradients for this model. This is contrary to the results in [121] who find a much higher orthogonality between their class gradients; the most likely reason for this is the difference in architecture in the two networks. [121] observe orthogonality "with fully-connected and convolutional networks, with ReLU and tanh non-linearites, at

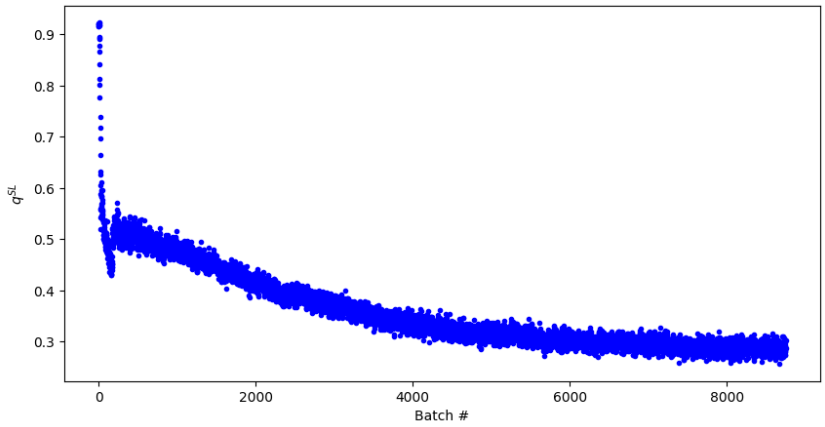



FIGURE 4.8: Showing the clustering of the logit gradients, $q^{\mathrm{SL}}$, plotted for a training run of a ResNet-9 over 50 epochs. Although the similarity reduces throughout training the similarity remains high due to the large dimensionality of the model's weights. The values shown here match well with those found by [121]

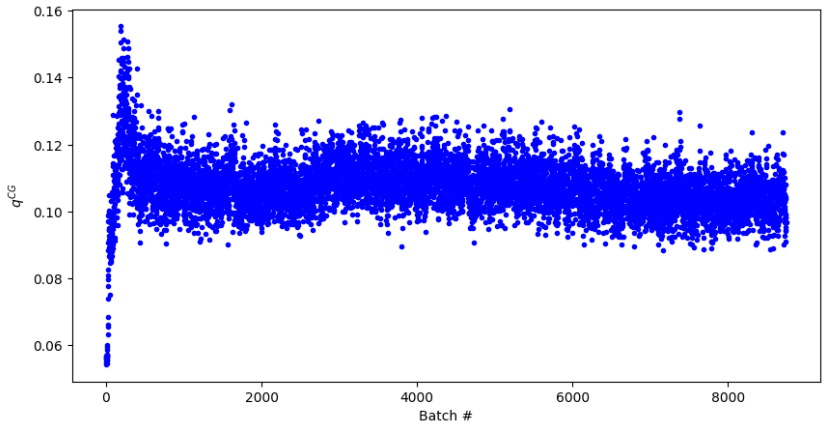



FIGURE 4.9: The inter-class cosine correlation $q^{\mathrm{GC}}$ for a ResNet-9 trained on CIFAR10, plotted for each batch during a 50-epoch training run. The class gradient similarity is higher than reported in other works [121], and are large enough to doubt the assumption that the class gradients are orthogonal.

different stages of training (including initialization), and on different datasets.", our work, in contrast, uses residual networks which have batch normalisation and skip connections. section 4.3.7.2 explains why this lack of orthogonality might be the reason for the method's lack of performance.

### 4.3.7 Discussion

We have a few potential explanations for this inconsistency between theory and praxis: there is too much noise per mini-batch, that the intersection of the class vector subspace and the top-$C$ eigenvector subspace is not high enough, or that rapid gradient descent loses information. The poor performance of the proposed method is both disappointing and mysterious. More information can be found in appendix E.

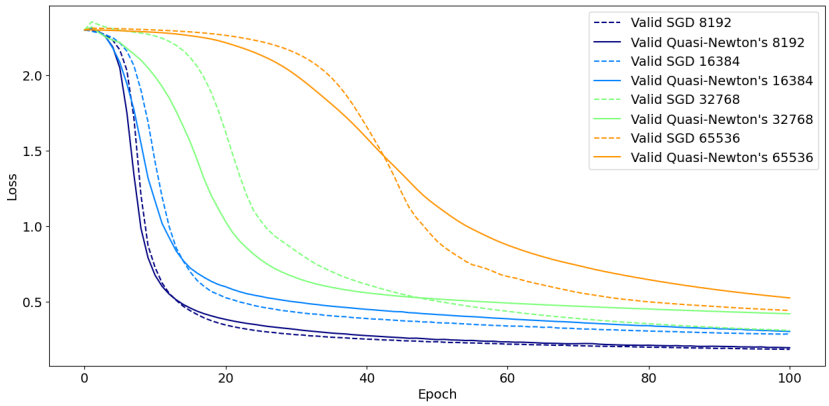



FIGURE 4.10: Validation loss curves on MNIST for different batch sizes using standard SGD and a quasi-Newton method, with learning rate fixed at $\eta = 10^{-2}$. Each curve shows the validation loss as a function of training epoch, averaged over multiple runs. SGD consistently outperforms the quasi-Newton's method at all batch sizes, but not in the early epochs. However, the performance gap narrows as the batch size decreases, with both methods converging to similar loss values for the small batch sizes. Notably, for batch sizes $\geq$ 32768, SGD exhibits significantly delayed convergence, while the quasi-Newton method maintains sharper descent early in training.

We have outlined a few potential explanations for this inconsistency between theory and praxis below.

#### 4.3.7.1 Too much noise

One obvious reason to explore is if a curve is noisy, then Hessian will capture the noise as local curvature meaning that our estimation will be spurious and oppose learning.

While we have taken steps to reduce noise, *e.g.* using the exponential moving average, there might still be too much stochasticity in each mini-batch given that the mini-batch error is an approximation of the training error which itself is an approximation of the generalisation error. Indeed, Granziol [128] shows that "the extremal eigenvalues of the batch Hessian are larger than those of the empirical Hessian" due to the noise in the mini-batch.

To see if inter-batch noise causes the lack of improvement, we have trained a small CNN on MNIST so that full-batch training is feasible. From figure 4.10 we can see that, although our quasi-Newton's method may start learning faster than SGDM, there comes a crossover point where SGDM overtakes and continues to have better performance regardless of the batch size[1]. Smaller batches have their crossover point in the first epoch due to the increased number of steps and so are not shown in figure 4.10 to illustrate this effect. Thus, we can conclude that the noise incurred due to batch training is not a root of the performance deficit.

[1]Since MNIST only has 50,000 training samples the batch size of 65,535 is just full batch.

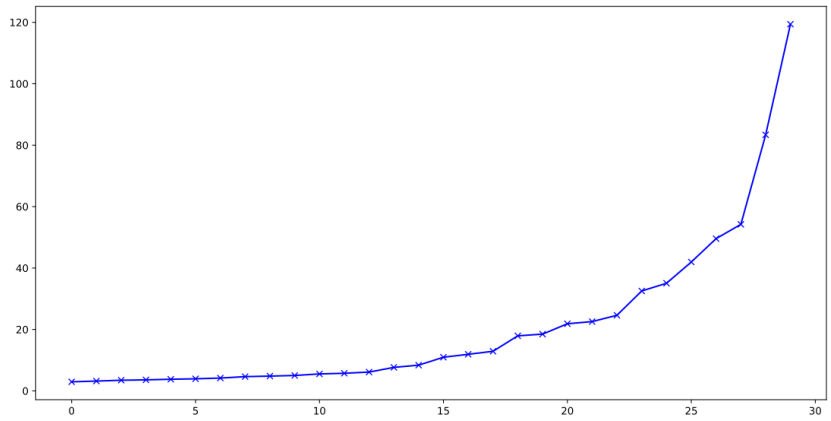



FIGURE 4.11: Top 30 eigenvalues of the true Hessian matrix for a small convolutional neural network with approximately 60,000 parameters. The eigenvalues are sorted in ascending order and plotted on a linear scale. This spectrum reveals a highly anisotropic curvature structure: most eigenvalues are clustered near zero, indicating flat directions in the loss landscape, while a small number of large eigenvalues dominate, corresponding to sharp directions. The rapid growth in the upper tail suggests a high condition number and potential instability in second-order optimisation unless curvature is carefully regularised or approximated. This motivates the use of curvature-aware methods that adapt to this spectral skew, such as quasi-Newton or subspace-based techniques.

#### 4.3.7.2 Non-intersection

If the class vector subspace does not totally cover the top eigenvector subspace, as is indicated by the high cosine between the class gradients, figure 4.9, then we are still leaving at least one high-curvature direction not preconditioned. This would also explain the inability to increase the learning rate.

It is interesting to note from figure 4.11 that, knowing $C$, it seems a reasonable cut-off point for the top-curvature subspace, but if given only the eigenspectrum it would be difficult to guess the number of classes. This might indicate that $C$ is not the best cut off point for the high-curvature subspace. It is possible that the class vectors actually have no particular relation to the high-curvature subspace but are simply an arbitrary way of splitting the cumulative gradient into some approximately orthogonal component vectors. Gur-Ari *et al.* [82], however, argue that it is only the top-$C$ eigenvectors that are preserved over time.

#### 4.3.7.3 Gradient information loss

It may be that the structure and non-convexity of the landscape is such that rapid gradient descent actually slows down the search. That is, this method pushes the low-curvature dimensions into a flat region of similar loss, where there is less gradient information, before they can be optimised into a low-loss region, thus slowing down the overall optimisation.

### 4.3.8 Conclusions

Common experience is that, when done right, optimisation strategies that compensate for different curvatures in the loss landscape can lead to a significant speed-up. Deep learning, however, rules out many traditional methods because of the vast dimensionality of the search space. Despite the failure of our naive implementation, the recognition that there exists a relatively low-dimensional subspace of high curvature which can be identified relatively easily suggests that it may be possible to improve on current methods. In this section we have shown a method for using the class gradients to precondition gradient descent which firstly highlights the need to identify the high-curvature subspace with reduced noise such that instabilities are not introduced into the optimisation, and secondly highlights the extent to which assumptions must be accurate in order to reduce the cumulative effect of the additional noise these assumptions impose on their approximations.

## 4.4 Estimating the top eigenvector of the Hessian

This section introduces an efficient method for calculation an approximation to the top eigenvector of the Hessian of a deep neural network via a de-noised power iteration method.

We briefly mentioned, in section 4.1, how calculating the Hessian is intractable and calculating its eigenvalues almost so. Yao *et al.* [129] provide methods for calculating the top eigenvalues, computing an oracle for the Hessian vector product: $\frac{\partial g_\theta^\top v}{\partial \theta} = Hv$, the trace using Hutchinson's method [130], and the spectral density of the Hessian via Stochastic Lanczos Quadrature (SLQ) [131], however, while all of these methods are possible to compute on sizeable networks, they require a large amount of computation, and thus unlikely to be able to provide enough information to be used directly in the optimisation. That is, the information it generates doesn't provide a benefit above and beyond simply using that compute to perform more steps of gradient descent. Therefore, it would be desirable to be able to estimate the Hessian with information that we have already calculated *i.e.* only the gradients. We show, in this section, how to find the top eigenvector of the Hessian from the gradients using a power iteration method. Code for this section can be found in appendix F.

### 4.4.1 Power iteration of the Hessian

The power iteration method is a simple iterative method to find the top eigenvector and eigenvalue of a diagonalisable matrix. It is based on the fact that if $v$ is an eigenvector of $A$ with largest eigenvalue $\lambda$, *i.e.* $Av = \lambda v$; then $A^k u$ will be an

approximation of $v$ for large $k$ and $u$ a random vector. This is because we can write any vector, $x \in \mathbb{R}^n$, as a linear combination of some eigenbasis, $\{v_1, \ldots, v_n\}$,

$$x = \sum_i \alpha_i v_i. \tag{4.42}$$

We can also think of $A$ as a linear transform, $T : \mathbb{R}^n \to \mathbb{R}^n$,

$$T(x) = \sum_i \lambda_i \alpha_i v_i, \tag{4.43}$$

and write the definition of its eigenvectors as, $T(v_i) = \lambda_i v_i$. Repeated application of $T$ gives,

$$T^k(x) = \sum_i \lambda_i^k \alpha_i v_i. \tag{4.44}$$

With a large enough $k$, the largest eigenvalue will dominate the sum and initial linear combination parameters. Thus, we can approximate the top eigenvector as,

$$T^k(x) \approx \pm \lambda^k v, \tag{4.45}$$

where $\lambda = \max_i \lambda_i$ is the largest eigenvalue and $v$ its corresponding eigenvector — called the top eigenvector.

Since the Hessian is an $n \times n$ real symmetric matrix it has an eigenbasis and thus can be power iterated. Therefore, in order to find its top eigenvector we can multiply a random vector by some large power of the Hessian. In practice, we do this iteratively to find a reasonable value of $p$ and as $\lambda^p$ may grow large enough to cause numerical issues we normalise at each iteration. This method also gives an estimate of the top eigenvalue by taking the difference in norms between iterations.

We can see that the top eigenvalue estimated by the power iteration method converges to the true value and the top eigenvector also converges to the true value, figure 4.12. With this run we have obtained a cosine similarity of $\approx 1$ between the true and estimated eigenvectors however, since the original linear coefficient — $\alpha_i$ — may be negative, we could also obtain an estimate of $-v$ and the cosine similarity would be $\approx -1$.

### 4.4.2 Estimating from gradients

Unfortunately, as we have already seen in previous sections, we do not have the Hessian available to us. Instead, we must use the gradient, which we can write as the product of the Hessian, $H$, and a random $n$-dimensional column vector, $r$,

$$g = Hr, \tag{4.46}$$

assuming the loss function is quadratic.

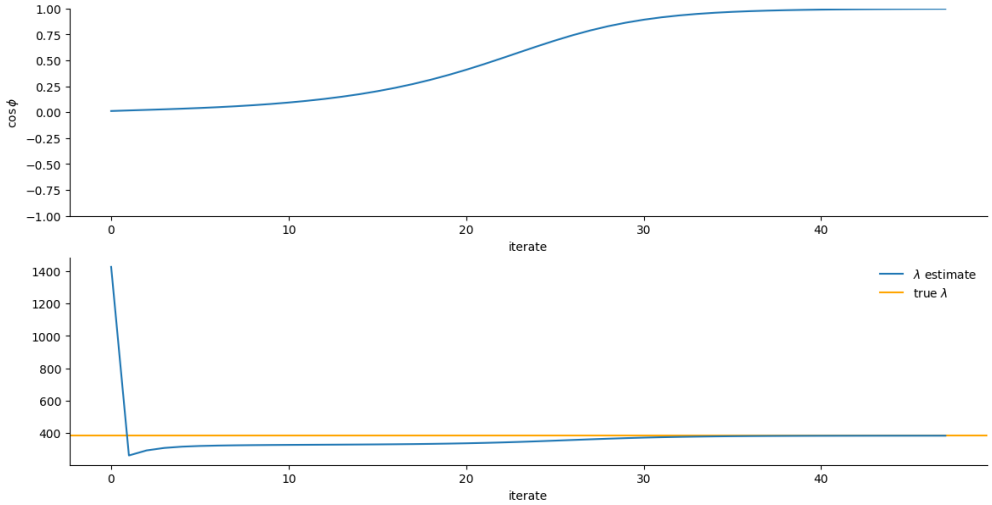



FIGURE 4.12: Estimating the top eigenvalue and eigenvector of a known matrix using the power iteration method. $\cos\phi$ is the cosine between the true eigenvector and the estimated one. The top eigenvalue is also estimated by the norm of the pre-normalised iterate. The true top eigenvalue is shown in orange.

When second-order information is inaccessible due to memory or compute constraints, one strategy is to estimate properties of the Hessian indirectly by leveraging first-order observations. Specifically, if gradients are treated as stochastic estimates drawn from varying directions in parameter space (*e.g.* due to minibatch sampling or noise injection), then one can interpret them as the product of a random direction with the underlying curvature matrix: $g = Hv$, where $v$ is an (approximately) random vector and $H$ is the true Hessian. This interpretation connects to a long-standing idea in numerical linear algebra: that random projections can reveal spectral properties of a matrix *cf.* Hutchinson [130] and Avron and Toledo [132]. In particular, the outer product $gg^\top = Hvv^\top H$ acts as a sketch of the form $H^2$, and its expectation over random $v$ yields useful information about the eigenstructure of $H$, especially its leading directions. This framework also underpins stochastic Lanczos and power iteration techniques used in neural networks to estimate top eigenvalues of the Hessian [103], [127]. Motivated by this, we propose to treat gradients as random projections of the Hessian and aggregate them to reconstruct partial curvature information, without requiring explicit computation or storage of second-order tensors.

**Theorem 4.4.** *The gradient of a quadratic function at a point is the product of the Hessian and the vector from the point to the minimum of the function.*

*Proof of theorem 4.4.* Let $f(x) = x^\top Ax$ be a quadratic function with positive semi-definite $A$, which can also be written in summation form as,

$$\begin{aligned} f(x) &= \sum_{i=1}^{d}\sum_{j=1}^{d} A_{ij}x_i x_j, \\ &= \sum_{i=1}^{d}(A_{ii}x_i^2 + \sum_{j\neq i} A_{ij}x_i x_j). \end{aligned} \tag{4.47}$$

Taking partial derivatives with respect to $x_k$ gives,

$$\begin{aligned} \frac{\partial f}{\partial x_k} &= 2A_{kk}x_k + \sum_{j\neq k} A_{jk}x_j + \sum_{j\neq k} A_{kj}x_j, \\ &= \sum_{j=1}^{d} A_{jk}x_j + \sum_{j=1}^{d} A_{kj}x_j. \end{aligned} \tag{4.48}$$

Assembling the partial derivatives into a vector gives,

$$\begin{aligned} \nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_d} \end{bmatrix} &= \begin{bmatrix} \sum_{j=1}^{d} A_{1j}x_j + \sum_{j=1}^{d} A_{j1}x_j \\ \vdots \\ \sum_{j=1}^{d} A_{dj}x_j + \sum_{j=1}^{d} A_{jd}x_j \end{bmatrix} \\ &= \begin{bmatrix} \sum_{j=1}^{d} A_{1j}x_j \\ \vdots \\ \sum_{j=1}^{d} A_{dj}x_j \end{bmatrix} + \begin{bmatrix} \sum_{j=1}^{d} A_{j1}x_j \\ \vdots \\ \sum_{j=1}^{d} A_{jd}x_j \end{bmatrix} \\ &= Ax + A^\top x, \\ &= (A + A^\top)x, \\ &= 2Ax, \end{aligned} \tag{4.49}$$

where the last line follows from the fact that $A$ is symmetric.

To get the Hessian of this function we take the second partial derivatives,

$$\frac{\partial f}{\partial x_k} = \sum_{i=1}^{d}(A_{ii}x_i^2 + \sum_{j\neq i} A_{ij}x_i x_j,$$

and so the second partial derivatives are constants,

$$\frac{\partial^2 f}{\partial x_k \partial x_{k'}} = A_{kk'} + A_{k'k}. \tag{4.50}$$

Thus,

$$\begin{aligned}\nabla^2 f(x) &= \begin{bmatrix} A_{11}+A_{11} & A_{12}+A_{21} & \cdots & A_{1d}+A_{d1} \\ A_{21}+A_{12} & A_{22}+A_{22} & \cdots & A_{2d}+A_{d2} \\ \vdots & \vdots & \ddots & \vdots \\ A_{d1}+A_{1d} & A_{d2}+A_{2d} & \cdots & A_{dd}+A_{dd} \end{bmatrix}, \\ &= A + A^\top, \\ &= 2A. \end{aligned} \tag{4.51}$$

Generalising both of these to the quadratic function $f(x) = \frac{1}{2}x^\top Ax + b^\top x + c$ gives,

$$\begin{aligned}\nabla f(x) &= Ax + b, \\ \nabla^2 f(x) &= A, \end{aligned} \tag{4.52}$$

and so

$$\nabla f(x) = A(x + A^{-1}b). \tag{4.53}$$

Since, $A$ is positive semi-definite, $f(x)$ has a minimum at $x^\star = -A^{-1}b$. Thus,

$$\nabla f(x) = A(x - x^\star), \tag{4.54}$$

where $A$ is the Hessian of $f(x)$. □

Since we do not know the value of the solution $x^\star$, as that would defeat the point of optimisation, we can assume that we are some random distance from the solution and let $r = x - x^\star$, even though we do not know the properties of this random vector, we can construct a matrix proportional to the square of the Hessian,

$$gg^\top = (Hr)(Hr)^\top,$$

as $H$ is symmetric,

$$\begin{aligned} &= Hrr^\top H, \\ &\approx H^2. \end{aligned} \tag{4.55}$$

Thus, we can find the top eigenvector of $H^2$ by the method above. Intuitively, we can see that the top eigenvector of $H$ is the same as of $H^2$ as we are performing power iteration of $H$ but skipping every other iterate. Unfortunately, as $gg^\top$ is only approximately equal to $H^2$ we can only really estimate the top eigenvalue if we know the expected noise in the random vector, otherwise each application contains an additional scaling of unknown size. This random value does not affect the top

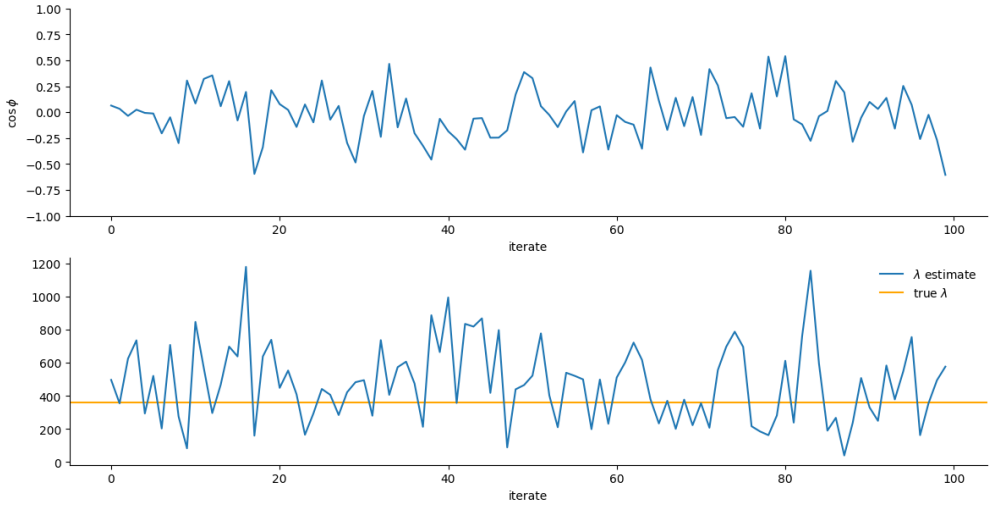



FIGURE 4.13: Estimating the top eigenvalue and eigenvector of a matrix using the power iteration method using only its product with a random vector. $\cos\phi$ is the cosine between the true eigenvector and the estimated one. The top eigenvalue is also estimated by the norm of the pre-normalised iterate. The true top eigenvalue is shown in orange.

eigenvector since we normalise our iterate, but the noise is included in the eigenvalue estimate.

To increase the performance of this method during training, we only perform one power iteration instead of performing many iterations of the power iteration method, this does not affect the accuracy a lot since we perform many iterations in the long run. Figure 4.13 reveals that this method does not work since neither value is converging. We will need to remove some of the noise from the estimation.

### 4.4.3 Noise reduction

While it seems that there is too much noise in the estimation, we note that the variance of the estimation is not related to the variance of the randomness of the random vector since the estimates are normalised; if we let $\hat{g} = H(\sigma r)$ then $\hat{g}\hat{g}^\top = H\sigma r\sigma r^\top H = \sigma^2 gg^\top$ which clearly gets normalised out. Instead, the noise-like property instead comes from the fact that $rr^\top$ is an outer product yielding rank-1 matrix and thus has only one non-zero eigenvalue — which is the square of the norm of $r$ — which reduces $gg^\top$ to a rank-1 matrix removing the information about its eigensystem. Let us assume that $r \sim \mathcal{N}(0, \sigma^2 I)$, is a $p$-dimensional vector, then $rr^\top$ is a rank-1 $p \times p$ matrix with a single non-zero eigenvalue of $\sigma^2 p$ — lemma A.4. Next, since we are working with two iterations of the Hessian, the top eigenvalue of $H^2$ is $\lambda^2$ and so the top eigenvector of the Hessian is the square root of our estimation.

To fix the noise issue due to the low rank matrix we can stack several gradients so that $r$ is now an $n \times s$ matrix to make $gg^t$ a rank-$min(s, n)$ matrix.

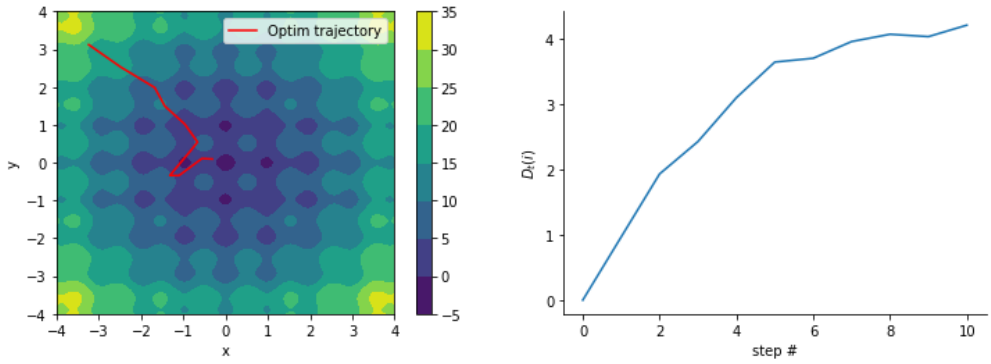
However, for this to be valid we need the gradients to be consistent over time, and also reasonably consistent within each batch.

#### 4.4.3.1 Temporal consistency

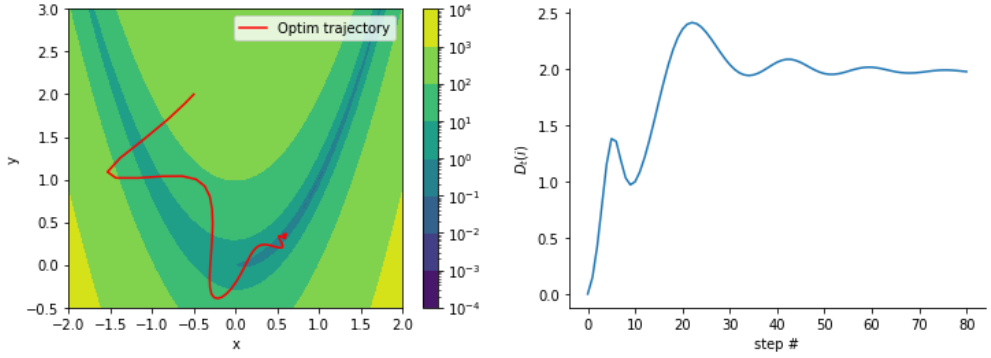
We look at whether the gradients always point in the general direction of the solution throughout training.

Often, the optimisation trajectory is not directly towards the minimum because the functions are more complex than a simple slope. The general shape of the trajectory shows us the overall pattern of the function, *i.e.* if the surface is dominated by a big undulating hill, as with Rosenbrock's function – figure 4.14b, or if it is dominated by many smaller obstructions, as Rastrigin's function – figure 4.14a.

We see that for the hilly surface, the trajectory is substantially curved and the total Euclidian distance, equation (3.1), fluctuates, but for the obstruction-dominated landscape the trajectory is generally straight and the Euclidian distance is consistently increasing until the solution — subject to the optimiser being able to overcome the obstructions. Obviously, for both of these classical functions, the distance will stabilise as the model settles or gets trapped in a minimum.




(A) Rastrigin's function, $a = 2$




(B) Rosenbrock's function, $a = 1, b = 100$

FIGURE 4.14: Classic optimisation test functions solved with SGDM, with the total Euclidian distance, equation (3.1), over time of their optimisation trajectories

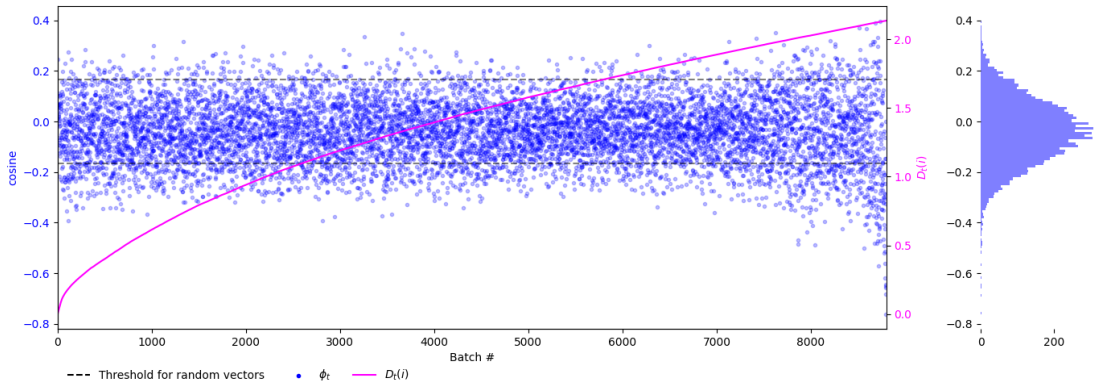
Now, to see if the gradients always point in the general direction of the solution, we measure the cosine similarity, $S_{\cos}(\cdot, \cdot)$, between the direction to the solution and the gradient,

$$\phi_t = S_{\cos}(\theta_\tau - \theta_t, \nabla\theta_t), \tag{4.56}$$

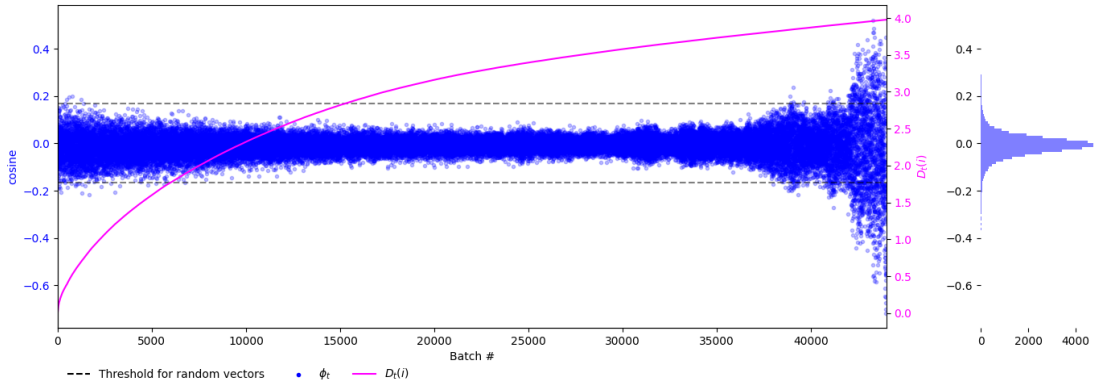
for a batch $t$ and final batch $\tau$.

As we update our model in the negative gradient direction, we would like the gradients to be as anticorrelated as possible with the solution, indicating a fast optimisation path. However, since this is unintuitive to talk about, we will talk as if the gradients are negative *i.e.* a gradient points towards the solution if it is anticorrelated with the direction from the current position to the solution.

If the weights move through a linear interpolation of the initialisation and the solution, $\alpha\theta_\tau + (1-\alpha)\theta_0$, $\alpha \in [0, 1]$, then $\phi_t = -1$ for all $\alpha$. Note, there are some artefacting with this metric, for example, for SGD $\phi_{\tau-1} = -1$ by definition, however, these do not affect the overall view.




(A) 200 Epochs




(B) 1000 epochs

FIGURE 4.15: $\phi_t$ for the filters of the second convolutional layer of a ResNet-18 on CIFAR10. The pattern holds regardless of the length of training — the same rough pattern is seen for 200 epochs as for 1000.

We plot $\phi$ for each filter in a convolutional layer of a ResNet-18; figure 4.15 establishes that the trajectory is mostly random with respect to the solution, that is, there are

almost as many gradients that are correlated with the direction to the solution as there are anticorrelated ones. Indeed, very few of the gradients pass our threshold for random vectors from appendix A.1.4. Importantly, however, we see that, the gradient does, on average, point towards the solution; illustrated by the peak of the distribution being negative and not at zero. One might assume, from the substantially increased variance of $\phi_t$ at the end of training, that the model is oscillating around a minimum, but this is an artefact of the fact that the absolute distances to the solution are smaller, so the cosine is more sensitive.

The cosine metric will bias towards capturing information about the low-curvature directions as they travel a greater Euclidian distance than the high-curvature directions, so causing greater changes in cosine distance. To gain some intuition of the loss-landscape, we have also plotted the total Euclidian distance travelled on the same plot to see if there is movement from and then towards the initial weights. figure 4.15 reveals that the total Euclidian distance travelled is continuously increasing reinforcing our previous result that the model is not oscillating around a minimum as this would require the Euclidian distance to oscillate. That most of the Euclidian distance is travelled at the beginning of training implies that initially there are many directions with consistently large gradients, figure 4.15 also shows that the loss-landscape has many small obstructions as its Euclidian distance over time is similar to Rastrigin's figure 4.14a.

#### 4.4.3.2 Batch consistency

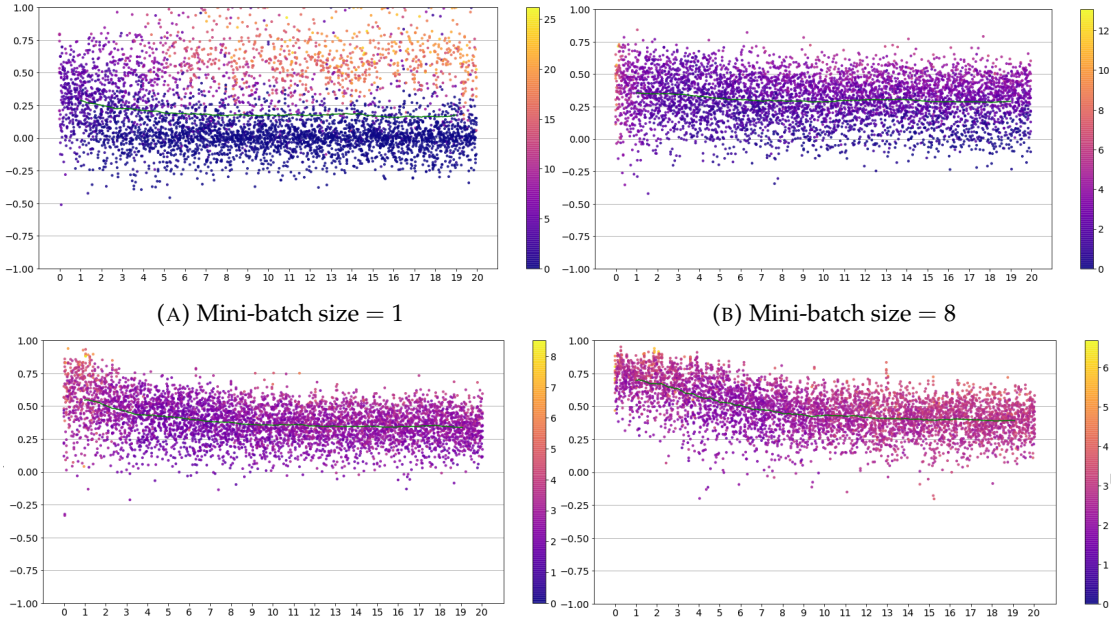
The noise inherent in different mini-batches will cause their gradients to point in different directions. If the landscape is rugged, then that noise will have more significance, and the gradients' directions will be spread out. If the landscape is smooth, then that noise will be less significant, and the gradients will be more cohesive.

Thus, if we have a measure of gradient cohesion, we can use it as a proxy for the loss landscape's ruggedness. Let the measure be the average angle between the gradients of several mini-batches and their mean gradient.

With the weights fixed, we calculate,

$$G = \begin{bmatrix} \vdots & & \vdots \\ g_1 & \cdots & g_d \\ \vdots & & \vdots \end{bmatrix}, \tag{4.57}$$

where $g_i$ is the column-vector gradient of the $i^{\text{th}}$ mini-batch. $G$ is an $n \times d$ matrix where $n$ is the number of model parameters and $d$ is the number of mini-batches sampled.



(A) Mini-batch size = 1

(B) Mini-batch size = 8

(C) Mini-batch size = 64

(D) Mini-batch size = 256

FIGURE 4.16: Effect of mini-batch size on gradient cohesion. $\cos \psi_i$, plotted on the $y$-axis, for $n = 10$ and sampled 25 times per epoch. $x$-axis is the epoch and the colour scale shows the magnitude of the gradient.

Then the measure is

$$\mathcal{C} = \frac{1}{d} \left\langle \frac{\mu^\top G}{\|\mu\|_2 \|G\|_2} \cdot \mathbf{1} \right\rangle, \tag{4.58}$$

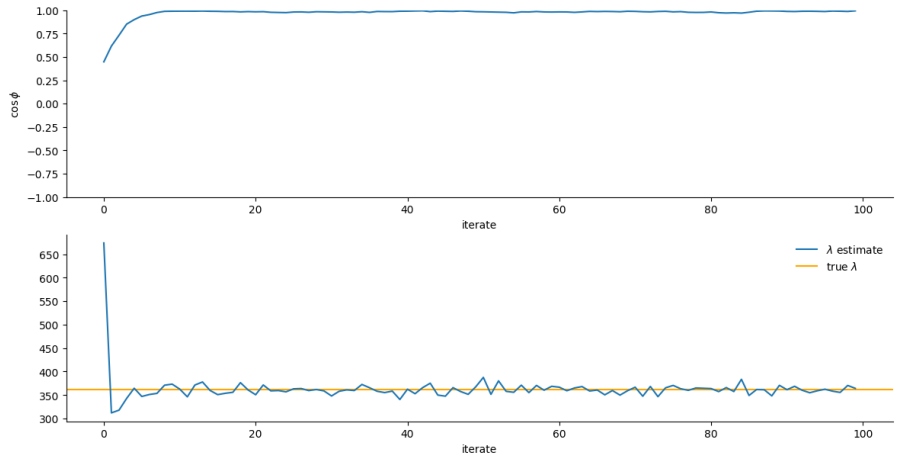
where, $\mu$, is the mean of the gradients, $\mu = \frac{1}{n} \sum_{i=1}^{n} g_i$ and $\mathbf{1}$ is a vector of ones.

As a feasibility check, we can increase the mini-batch size and, since the stochasticity reduces, the gradients should be more cohesive *i.e.* $\mathcal{C}$ closer to 1.

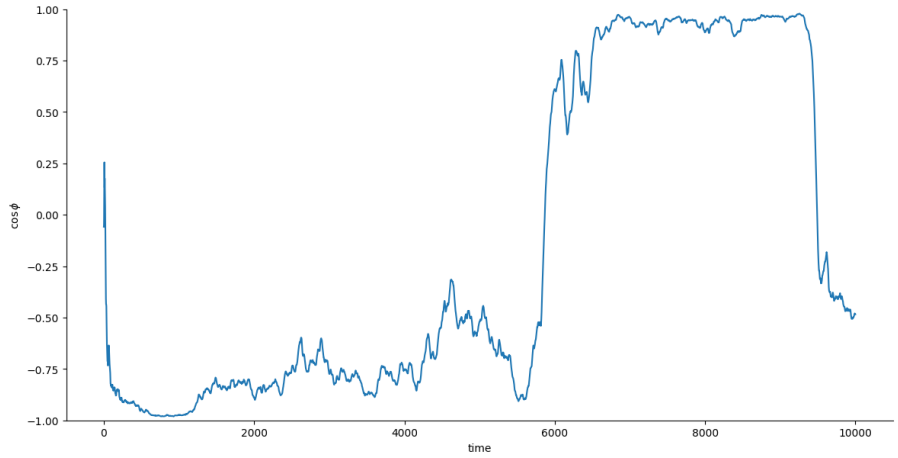
From figure 4.16 we can see that the larger the mini-batch size, the higher the average cohesion and the smaller the variance; indicating that our assumption, that larger mini-batches decrease the amount of noise, is correct. Moreover, we can see how errors dominate the gradient from figure 4.16a as there are gradients with $\psi_i \approx 1$ since their magnitude is much larger than the other samples'. This also reveals how the gradients become less cohesive over time; there seems to be a very obvious direction that is shared by all the examples; the cohesion degrades. This could be that the classes become separated or just that the variance in the input data increases relative to the model's capacity to the point that the network has difficulty learning them.

#### 4.4.3.3 Denoised

From the previous two sections we see that the gradients are both consistent both within the mini-batch and over time. Therefore, its is valid to stack the gradients to remove noise.




(A) Estimating the top eigenvalue and eigenvector of a matrix using the power iteration method using its product with 1000 random vectors. $\cos\phi$ is the cosine between the true eigenvector and the estimated one. The top eigenvalue is also estimated by the norm of the pre-normalised iterate. The true top eigenvalue is shown in orange.




(B) Estimating the top eigenvalue and eigenvector of a matrix using the power iteration method using a rolling mean of its product with 1000 random vectors, where only one vector is replaced every iterate. $\cos\phi$ is the cosine between the true eigenvector and the estimated one.

FIGURE 4.17

Now that we have removed some of the noise from the estimation, figure 4.17a, by working with $s = 1000$ gradients at a time, the top eigenvector is converging to the true value, and the estimated eigenvalue converges to $\|v\|_2 = \frac{\sigma^2\lambda^2}{s}$ as expected. To use this while training we do not want to have to pause the model at each step and recalculate a whole sample of gradients as this would be terribly inefficient, thus at each optimisation step we will replace only one item of the sample with the new gradient and use the last $s$ gradients in this iteration. Figure 4.17b shows that the top eigenvector and eigenvalue are still converging to the true values, however, they are significantly more unstable. We can see that there is some oscillation due to the randomness in the gradients, but if we use a larger sample size we can reduce the oscillations and get a more consistently accurate estimation — figure 4.18. The results here do not guarantee performance for a real situation as the Hessian is static.

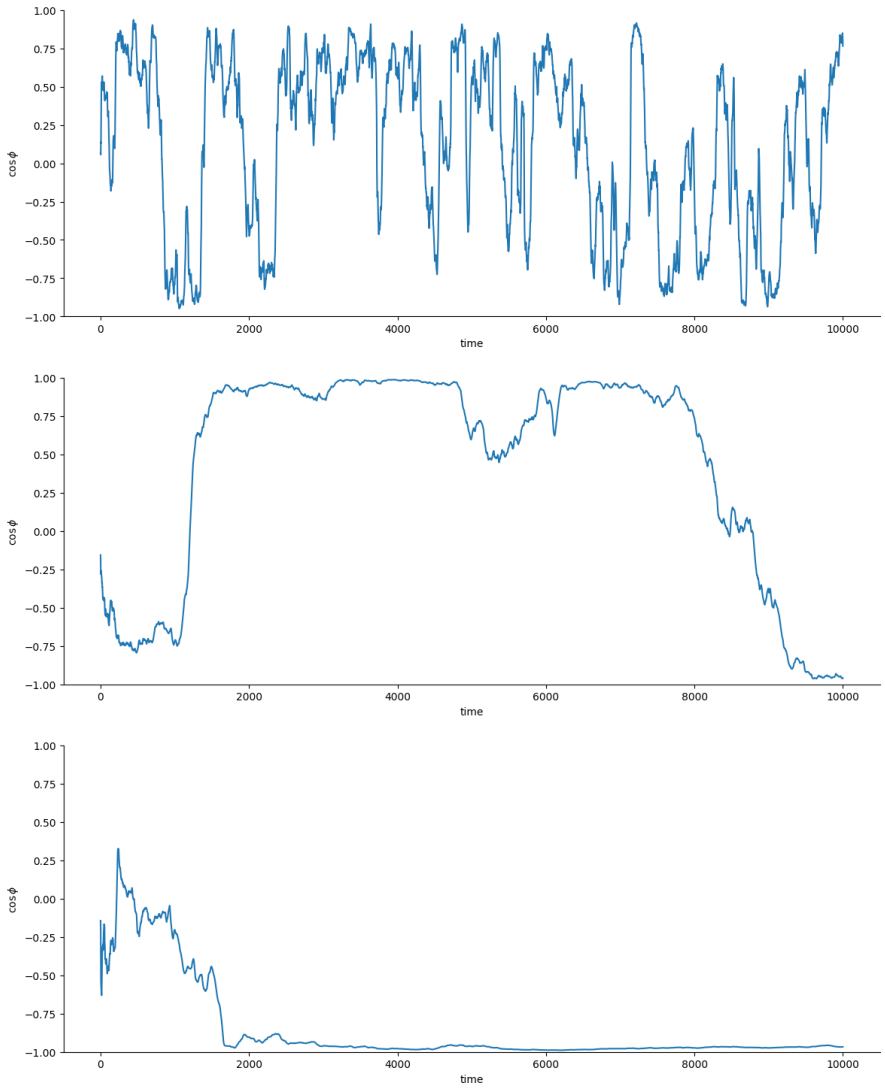



FIGURE 4.18: Estimating the top eigenvalue and eigenvector of a matrix using the power iteration method using a rolling mean of its product with 100, 1000, 10000 random vectors, where only one vector is replaced every iterate. $\cos\phi$ is the cosine between the true eigenvector and the estimated one.

### 4.4.4 Estimating with a Deep Network

We now run this estimation on a deep network, and show two ways to use the calculated values in the optimisation. While training a ResNet-20 on CIFAR-10, we calculate the top eigenvector and eigenvalue of the Hessian of the loss function as described above using a sample size of 250.

First, we use the estimated top eigenvalue to adjust the step size of the optimisation algorithm. As we calculated in section 4.3.1 the optimal step size for the gradient descent algorithm is bounded by the top eigenvalue — $\eta \leq \frac{2}{\lambda}$. Thus, we set the step size of the optimisation to be equal to this value using our estimated top eigenvalue. We find that the training suffers from catastrophic blow-ups using this method as it

only takes one large step size to significantly upset the model. To counteract this, we can clip the step size to be less than the provided learning rate, $\min\left\{\eta, \frac{2}{\lambda}\right\}$.

Alternatively, we can use the estimated top eigenvector to adjust the step size by calculating the projection of the eigenvector onto the gradients. The projection is given by,

$$\text{proj}_g(v) = \frac{g \cdot v}{\|g\|_2^2} g. \tag{4.59}$$

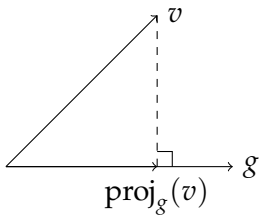



FIGURE 4.19: The projection of the top eigenvector onto the gradient.

However, there are two problems with this, firstly, as we are estimating $\lambda^2$ we do not know if we have estimated $v$ or $-v$, therefore we must use the absolute value of the projection to ensure the step is aligned with the gradient. Secondly, the top eigenvector is the direction of the largest change in the gradient of the loss function, and therefore if the gradient is aligned with this direction we need to take a smaller step than if it is orthogonal; since the eigenvector is of unit length we can simply subtract the absolute dot product from 1. Lastly, the magnitude of the gradient is also important information to keep so we adjust the step size to be 1 minus the absolute cosine between the eigenvector and the gradient, which is also 1 minus the norm of the projection.

$$\begin{aligned}
\eta &= 1 - \left| \frac{g^\top v}{\|g\|_2 \|v\|_2} \right|, \\
&= 1 - \left\| \text{proj}_g(v) \right\|_2, \\
&= 1 - \frac{\left| g^\top v \right|}{\|g\|_2}.
\end{aligned} \tag{4.60}$$

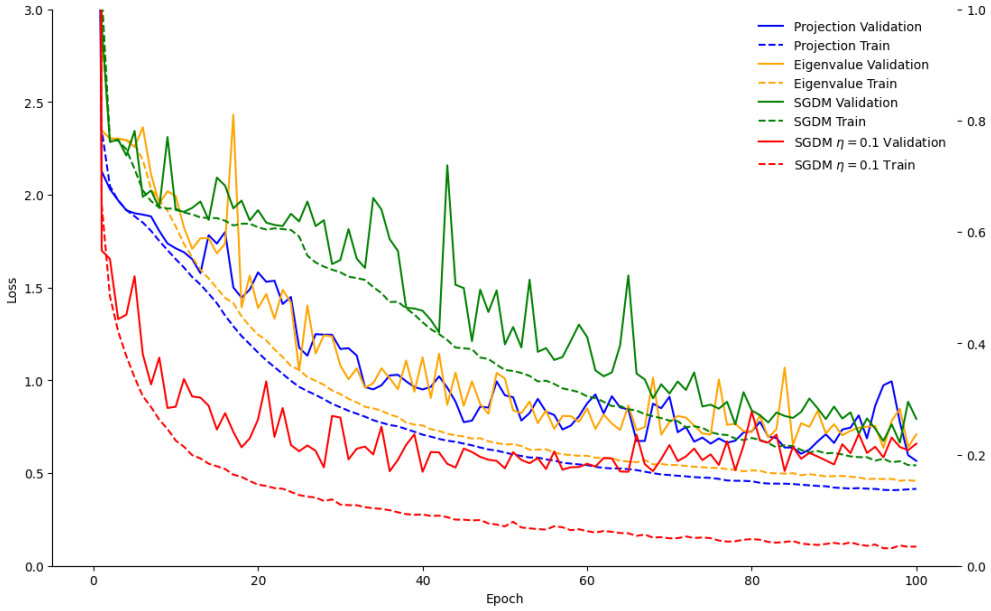
Again, to avoid large step sizes we can clip the step size to be less than the provided learning rate; pseudocode for both of these methods is shown in algorithm 5.

To experiment we train a ResNet-20 on CIFAR-10 using a learning rate — or learning rate bound — of 1; both methods perform better than SGD,figure 4.20, with this high step size, indicating that the methods are indeed limiting the step size for when the loss landscape is highly curved, and that this is beneficial. However, when compared with a better learning rate for SGD, 0.1, we find that these eigen-estimation methods perform suboptimally, most notably, at the start of training when the model is most sensitive to moving in the wrong direction. Overall there is slightly less noise in the

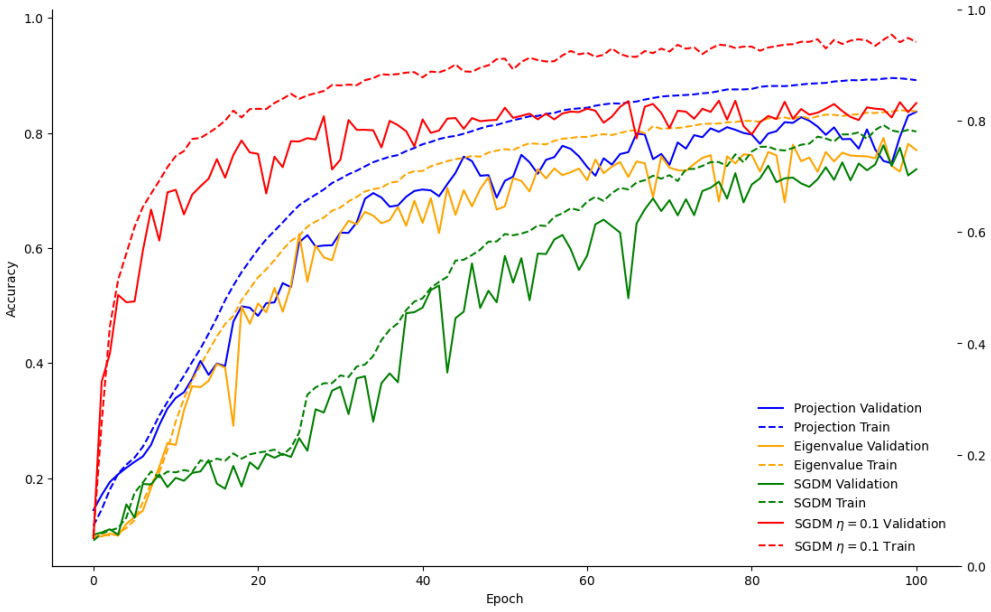
**Algorithm 5** Estimating the top eigenvector of the Hessian using the power iteration method

**Require:** $\eta' > 0$ ▷ Step size limit
**Require:** $p > 0$ ▷ Number of parameters
**Require:** $s > 0$ ▷ Sample size
$v \leftarrow \mathbb{R}^p,\ v_i \sim \mathcal{N}(0,1)$ ▷ Initial random vector for top eigenvector
$S \leftarrow \mathbb{R}^{p \times n},\ S_{ij} \sim \mathcal{N}(0,1)$ ▷ Initial random matrix for gradients
**for** each mini-batch, $i$ **do**
  $g \leftarrow \nabla L(\theta_i)$ ▷ Get gradients
  $S_{i \bmod s} \leftarrow g$ ▷ Replace the oldest gradient with new gradient
  $v \leftarrow SS^\top v$ ▷ Power iteration
  $\lambda \leftarrow \sqrt{\frac{\|v\|_2}{s}}$ ▷ Estimate top eigenvalue
  $v \leftarrow \frac{v}{\|v\|_2}$ ▷ Normalise top eigenvector
  $p \leftarrow 1 - \frac{\left|g^\top v\right|}{\|g\|_2}$ ▷ Projection based step size
  **if** eigenvalue based step size **then** ▷ Clipped step size
    $\eta \leftarrow \min\left\{\eta', \frac{2}{\lambda}\right\}$
  **else if** projection based step size **then**
    $\eta \leftarrow \min\left\{\eta', p\right\}$

train set for the eigen-estimation methods, but the validation curves show similar smoothness to the lower learning rate SGD run. These methods do not perform better than normal SGD, likely because the estimated eigensystem is not accurate enough, although it does reduce the oscillations in accuracy and loss while training, figure 4.20. This indicates that these methods could perform better if a way to better estimate the magnitude of the noise is found.




(A) Loss




(B) Accuracy

FIGURE 4.20: Accuracy and loss of a ResNet-20 on CIFAR-10 while training using the top eigenvalue and top eigenvector of the Hessian. The top eigenvalue is used to adjust the step size of the optimisation algorithm, and the top eigenvector is used to adjust the step size by calculating the projection of the eigenvector onto the gradients. A single run is shown so as to display the amount of noise in each method. Best viewed in colour.

# V

# Adapting current optimisers using gradient self-orthogonalisation

This chapter uses the insights from the previous analyses, especially that the directionality measure for SGDM that shows a period of low direction at the beginning of learning to develop a method that exploits the inherent structure of neural networks leading to an adaption to current optimisers using orthogonalisation that improves convergence speed whilst maintaining performance.

The no-free-lunch theorem [133] "establish[s] that for any algorithm, any elevated performance over one class of problems is offset by performance over another class". In other words, an optimisation algorithm works on one problem because it is exploiting the structure of that problem, and so it works well on problems with that structure, and not of problems that lack it. This chapter is then also based of the fact that the structure of neural networks is also a part of the structure of the problem, and so we can exploit this to improve the performance of optimisation algorithms on neural networks.

## 5.1 Introduction

Neural network layers are made up of several identical, but differently parametrised, components. Layers consist of several components so that they can provide a diverse set of intermediary representations to the next layer. For simplicity and ease of understanding, we will particularise these components as convolutional filters in our explanations, however, the same arguments apply to any layer with multiple components, *e.g.* heads in a multi-headed attention layer or the vectors in a dense layer. Despite this reason for multiple filters in a convolutional layer, there is no constraint or bias — other than the implicit bias from the cost function — to learning different parametrisations. This is undesirable since at the start of learning one might

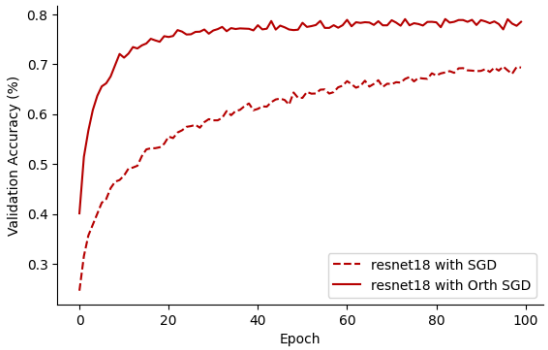



FIGURE 5.1: An example of the speed-up obtained by orthogonalising the gradients on CIFAR-10.

expect all the filters to learn the same parametrisation — the parametrisation that provides the most information to the next layer — and we show this to be the case in section 5.5.1. This co-learning problem, in aggregate, supplies duplicate information to the next layer. We introduce a diversification bias in the form of orthogonalised gradients and find a resultant speed-up in learning and sometimes improved performance, see figure 5.1.

Our novel contributions include this new optimisation method, thorough testing on CIFAR-10 and ImageNet, additional testing on a semi-supervised learning method, and experiments to support our hypothesis.

In section 5.3 we detail the method and in section 5.4 the results to give an understanding of how this method works and its capabilities. Then, in section 5.5, we provide experimental justifications and supporting experiments for this method along with finer details of the implementation and limitations.

## 5.2 Related Work

The idea of leveraging orthogonality in neural network training has emerged in multiple contexts, with differing motivations. Broadly, these approaches can be categorised into two camps: orthogonalisation of optimiser update, typically for multi-task learning or training stability, and orthogonalisation of weights, generally for optimisation dynamics, generalisation, and representation disentanglement.

### 5.2.1 Orthogonalisation for Multi-task learning

Gradient orthogonalisation has been notably explored in the multi-task learning literature. In multi-task learning, gradient interference between tasks can hinder performance. For instance, Yu *et al.* [134] introduced the Projecting Conflicting Gradients (PCGrad) algorithm, which modifies gradients by projecting task-specific gradients to be orthogonal to one another when in conflict. This encourages

task-specific updates to avoid destructive interference and promotes learning of distinct task representations.

Assume a shared model with parameters $\theta$, optimized over $T$ tasks. Then in a multi-task problem, each task $i \in \{1, \ldots, T\}$ has its own loss function $\mathcal{L}_i(\theta)$. The overall goal is to minimize a combination of these task losses. A naive approach is to use the sum or average:

$$\mathcal{L}_{\text{total}}(\theta) = \sum_{i=1}^{T} \mathcal{L}_i(\theta)$$

The gradient used to update $\theta$ is then,

$$g = \sum_{i=1}^{T} \nabla \mathcal{L}_i(\theta)$$

However, when tasks are *conflicting*, this sum can result in suboptimal updates. In particular, if $\nabla \mathcal{L}_i \cdot \nabla \mathcal{L}_j < 0$, then the gradient directions oppose each other—potentially slowing convergence or degrading performance.

PCGrad modifies each task gradient $g_i = \nabla \mathcal{L}_i$ to reduce conflicts by projecting it away from any conflicting task $g_j$,

$$\tilde{g}_i \leftarrow \begin{cases} g_i - \frac{\langle g_i, g_j \rangle}{\|g_j\|^2} g_j & \text{if } \langle g_i, g_j \rangle < 0 \\ g_i & \text{otherwise} \end{cases}$$

This is done pairwise for all $i \neq j$, potentially across multiple other task gradients. After all necessary projections are applied, the final update is usually the average of the modified gradients,

$$g = \sum_{i=1}^{T} \tilde{g}_i$$

where $\tilde{g}_i$ is the projected gradient for task $i$. This projection ensures that the updated gradient $g_i$ is orthogonal or positively correlated to $g_j$, mitigating interference and promoting more effective multi-task learning. This formulation preserves useful signal while removing conflicting components, and empirically improves learning in tasks with partially overlapping or competing objectives.

However, this line of work focuses on balancing multiple objectives, rather than improving learning efficiency or generalisation in a single-task setting.

In contrast, our work focuses on the role of orthogonalisation in single-task performance, where the benefits are subtler but potentially broader: inducing better conditioned optimisation dynamics, encouraging disentangled internal representations, and biasing learning toward functionally diverse features.

A large body of literature investigates weight orthogonality as a means to stabilise and regularise deep network training. Several empirical studies have demonstrated that constraining weights to be orthogonal can improve convergence and generalisation, particularly in very deep networks [135]–[137]. For example, Xie *et al.* [136] showed that enforcing strict orthogonality during training allowed deep convolutional networks to be trained effectively without residual connections—an architectural crutch generally considered essential for mitigating vanishing gradients. Similarly, Bansal *et al.* [135] argued that orthogonal initialisation and regularisation of weights can reduce covariance shift and improve gradient flow, yielding better training dynamics and final accuracy.

There are two main ways that this orthogonality constraint is satisfied. First, a regularisation approach which is a soft constraint, allowing the optimiser to explore solutions near the orthogonal subspace without explicit manifold constraints. This is simpler, computationally cheaper, but does not guarantee strict orthogonality. Second, a manifold optimisation approach which strictly constrains solutions to be orthogonal throughout training, preserving desirable mathematical properties but increasing complexity and computational overhead due to manifold projections and specialized optimisation techniques. The choice between them typically depends on the application's requirements and the computational resources available.

### 5.2.2 Orthogonality through Regularisation — Soft Constraints

For the regularisation approach, [138], [139] propose adding a penalty term to the loss function to encourage weight matrices to remain orthogonal during training. Mathematically, given a weight matrix $W \in \mathbb{R}^{m \times n}$, the penalty term added to the loss function is typically formulated as follows:

$$\mathcal{L}_{\text{ortho}}(W) = \lambda \|W^\top W - I\|_F^2$$

where $\|\cdot\|_F$ denotes the Frobenius norm, $I$ is the identity matrix of appropriate dimension, $\lambda$ is a hyper-parameter that controls the regularisation strength.

The term $W^\top W - I$ quantifies deviation from perfect orthogonality, and the Frobenius norm measures the total magnitude of this deviation. When the weight matrix is perfectly orthogonal, we have $W^\top W = I$, meaning the columns of $W$ are mutually orthonormal. Minimizing this penalty encourages weights to lie close to the orthogonal subspace, which has favourable properties, such as norm preservation and stable gradient dynamics.

The optimisation loss then becomes,

$$\mathcal{L}_{\text{total}}(\theta) = \mathcal{L}_{\text{original}}(\theta) + \lambda \sum_{l \in L} \|W_l^\top W_l - I\|_F^2,$$

where, $L$ indexes layers where orthogonality is enforced. Unfortunately, there is an obvious problem with this additional loss, namely that it encourages the columns of $W \in \mathbb{R}^{m \times n}$ to be orthonormal. However, this formulation makes a critical assumption — that it's possible for $W^\top W \approx I$. That is only true if $W$ is undercomplete or square, *i.e.* if $W \in \mathbb{R}^{m \times n}$ and $m \geq n$.

The matrix $W^\top W$ (an $n \times n$ matrix) is the Gram matrix of the columns of $W$.

Now, if $m \geq n$, then the columns can in principle be orthonormal — this is the classic Stiefel manifold case. But if $m < n$, then the rank of $W$ is at most $m$, and so the rank of $W^\top W$ is also at most $m$.

Since the identity matrix $I_n \in \mathbb{R}^{n \times n}$ has full rank $n$, and $W^\top W$ has rank $\leq m < n$, then $W^\top W$ can never be close to $I_n$ in Frobenius norm, no matter what you do.

This means that minimizing $\|W^\top W - I\|_F^2$ is inherently biased in the overcomplete case, you're penalizing something that cannot be satisfied. The optimiser may respond by shrinking the weights to small norms, to get $W^\top W \approx 0$, rather than learning diverse features. The regularise then acts more like a norm constraint than a decorrelation objective.

To address this, [135] proposed Double Soft Orthogonality Regularisation (DSO), which adds a second symmetric term:

$$\mathcal{L}_{\text{DSO}} = \lambda_1 \|W^\top W - I\|_F^2 + \lambda_2 \|W W^\top - I\|_F^2$$

This now enforces soft orthogonality on both columns and rows, $W^\top W \approx I$ encourages the columns to be orthonormal, if $m \geq n$, and $W W^\top \approx I$ encourages the rows to be orthonormal, if $m \leq n$. This dual formulation can partially compensate for the limitations of the rank condition by regularizing both sides. It effectively tries to spread out the weight directions in both input and output space. Still, it's only a partial fix. DSO improves numerical conditioning and encourages diversity, but doesn't directly control the most important aspect of representation degeneracy, feature redundancy.

[135] then propose a sharper criterion based off Mutual Coherence (MC) [140], instead of trying to match a matrix to identity, directly minimize the pairwise correlation between columns,

$$\mathcal{L}_{\text{MC}} = \lambda \max_{i \neq j} \frac{\left| w_i^\top w_j \right|}{\|w_i\| \cdot \|w_j\|} \tag{5.1}$$

$$= \lambda \left\| W^\top W - I \right\|_\infty. \tag{5.2}$$

This is the maximum cosine similarity between all distinct column pairs in $W$, also known as mutual coherence, it measures the worst-case angular correlation between features.

This is an improvement over DSO as the formulation does not assume full-rank Gram matrices. Additionally, it is scale-invariant — normalising by vector norms eliminates the bias toward shrinking weights, and finally, it directly penalizes redundant feature directions, *i.e.* overlapping filters or neurons, which is often the real cause of poor generalisation or inefficient learning. Instead of trying to make $W^\top W$ close to identity, which assumes equal norm and orthogonality, the MC regulariser tries to maximize the angular diversity of the learned features, which is a much more attainable and relevant geometric goal in deep networks.

### 5.2.3 Orthogonality through Reparametrisation — Hard Constraints via Manifold Optimisation

For the manifold optimisation approach, [137] propose maintaining strict orthogonality constraints through parameter reparametrisation methods. In particular, they consider parameters constrained directly onto the Stiefel manifold, the set of orthonormal matrices,

$$\mathrm{St}(m, n) = \{W \in \mathbb{R}^{m \times n} : W^\top W = I_n\}$$

A commonly used approach is to represent the orthogonal weights explicitly is via the exponential map or a Cayley transform. For example, using the exponential map representation [141], a standard geometric optimisation technique, consider a skew-symmetric matrix $A$ (*i.e.* $A^\top = -A$). Then the matrix exponential gives an orthogonal matrix,

$$W = \exp(A)\,, \quad \text{where } A^\top = -A$$

Another popular alternative is the Cayley transform [142],

$$W = (I + A)(I - A)^{-1}, \quad \text{where } A^\top = -A$$

This transformation is computationally efficient for optimisation, as it avoids expensive exponential operations while preserving strict orthogonality.

Optimisation under this hard constraint setting becomes a constrained optimisation problem on the Stiefel manifold. The update rule for parameters must preserve membership on the manifold. Thus, gradient-based optimisation is adapted to a Riemannian gradient step, given the Euclidean gradient $\nabla\mathcal{L}$, the Riemannian gradient on the Stiefel manifold is the projection onto the tangent space at $W$:

$$\nabla_{\mathrm{St}}\mathcal{L}(W) = \nabla\mathcal{L}(W) - W\mathrm{sym}(W^\top \nabla\mathcal{L}(W))$$

where, $\text{sym}(X) = \frac{1}{2}(X + X^\top)$ is the symmetric part of a matrix $X$. Then the update becomes,

$$W \leftarrow \text{Retract}_{\text{St}}\left(W - \eta \nabla_{\text{St}} \mathcal{L}(W)\right)$$

where the retraction operator $\text{Retract}_{\text{St}}(\cdot)$ projects the update back onto the Stiefel manifold, typically via QR decomposition or another orthonormalisation procedure.

Strictly enforcing orthogonality this way has robust theoretical support [142], [143], orthogonality ensures gradients neither vanish nor explode exponentially across layers as it preserves the norm of backpropagated gradients, stabilizing deeper network training, and training directly on the Stiefel manifold improves the conditioning of optimisation landscapes and has been empirically observed to accelerate convergence, especially in recurrent architectures [142]. Additionally, [144] provided rigorous justification for orthogonal weight matrices in terms of norm preservation, gradient stability, and expressivity bounds. They position orthogonality as a form of implicit regularisation, potentially constraining the learned functions to be more stable and interpretable.

To see the norm preserving effect, consider a linear layer with square weight matrix $W$ and input $x$. The output is $y = Wx$. During backpropagation, the gradient with respect to the input is $\nabla_x \mathcal{L} = W^T \nabla_y \mathcal{L}$. If $W$ is orthogonal, $W^T W = I$, then,

$$\begin{aligned}
\|\nabla_x \mathcal{L}\|^2 &= \|W^\top \nabla_y \mathcal{L}\|^2, \\
&= (W^\top \nabla_y \mathcal{L})^\top (W^\top \nabla_y \mathcal{L}), \\
&= (\nabla_y \mathcal{L})^\top W W^\top \nabla_y \mathcal{L}, \\
&= \|\nabla_y \mathcal{L}\|^2, \\
\|\nabla_x \mathcal{L}\| &= \|\nabla_y \mathcal{L}\|
\end{aligned}$$

This equality shows that the gradient norm is preserved, preventing issues like vanishing or exploding gradients and facilitating deeper network training, however this is only guaranteed in the square case.

### 5.2.4 Summary

While orthogonality offers theoretical and empirical benefits, enforcing strict orthogonality can be challenging and may limit the expressiveness of the model. Soft constraints or regularisation terms provide a balance, encouraging orthogonality without strictly enforcing it, thus allowing the network to learn more flexible representations.

Moreover, computational overhead introduced by enforcing orthogonality should be considered. Techniques like the Cayley transform or Householder reflections offer computationally efficient ways to maintain orthogonality during training.

Incorporating orthogonality into neural network training, whether through gradient modification or weight constraints, has been shown to enhance training dynamics and generalisation, but the choice of method and the degree of enforcement should be carefully tailored to the specific application and computational resources available.

However, methods that directly enforce orthogonality on weights suffer from several limitations. First, they can introduce significant training instability due to the non-convex nature of the orthogonality constraint manifold, interdependence across layers, and the need for projection or reparametrisation steps during training [137]. Second, they constrain the optimiser to explore only a strict subset of the weight space—effectively transforming the problem into a form of constrained optimisation. Whilst this can improve training in some settings, it also reduces the expressive flexibility of the model. In contrast, much of deep learning's success has stemmed from its ability to leverage vast, over-parametrised weight spaces, which facilitate better empirical risk minimisation and robustness to local minima [145], [146].

Thus, rather than restricting the parameter space of the model, we propose to bias the learning dynamics themselves through gradient orthogonalisation. By modifying the gradients before they are consumed by the optimiser, we steer the updates toward more diverse and less redundant directions, without hard constraints on the weights. This approach introduces a soft bias—analogous in spirit to weight decay, which encourages but does not enforce small parameter norms. Such soft constraints have proven to be both theoretically appealing and empirically robust in deep learning, offering a regularising influence without curtailing the optimiser's ability to explore the solution manifold.

Finally, we note that our method is optimiser-agnostic and architectural-flexible, in contrast to weight-orthogonalisation methods that are often tied to specific reparametrisations or require special care when used with adaptive optimisers like Adam. By staying in an unconstrained weight space, we preserve compatibility with a wide range of existing training techniques while injecting a bias toward orthogonal learning trajectories — a property we show empirically to result in faster convergence and competitive final performance.

## 5.3 Method

### 5.3.1 Problem Conjecture

We can avoid this co-learning problem by self-orthogonalisation, *i.e.* orthogonalising the gradients of the filters with respect to each other, using the weight reshaping shown by Jia *et al.* [144]; that is, for a convolutional kernel $K \in \mathbb{R}^{c_{\text{out}} \times c_{\text{in}} \times h \times w}$, it can be reshaped into $W \in \mathbb{R}^{c_{\text{out}} \times (c_{\text{in}} \cdot h \cdot w)}$, speeding up the learning of the lower information filters at the beginning of learning, as it is now more difficult for them to learn the

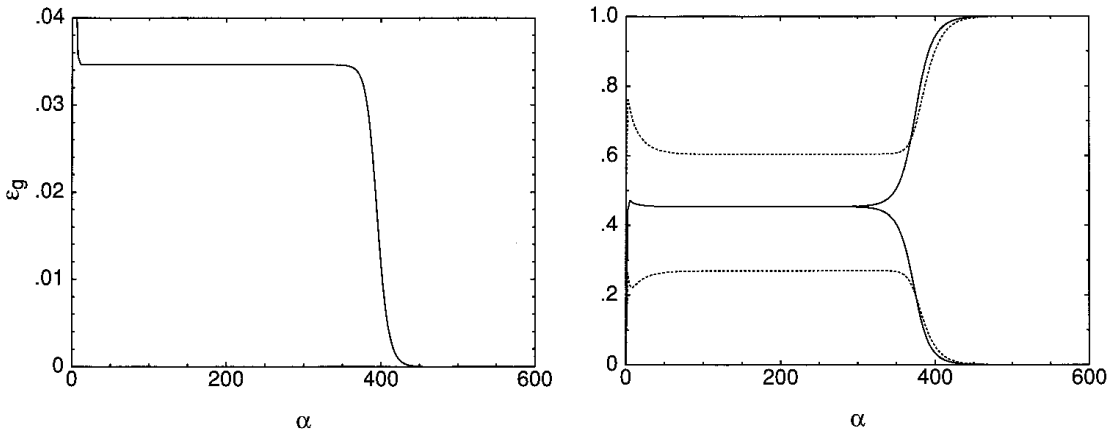



FIGURE 5.2: "Time evolution of the generalization error (a) and the order parameters ... (b) ... in the ... learning scenario with an isotropic teacher" [147]. Taken from Biehl *et al.* [147]. Where figure (b) shows the plateau at the beginning of learning where the two nodes learn the same value.

same parametrisation. What we will show is that doing this significantly speeds up learning on several tasks across different data sets. Gradient self-orthogonalising may provide benefits in addition to avoiding this co-learning problem since it will continuously ensure information from unusual datum are still applied to the filters. Note that this method does not restrict the features to an orthogonal sub-space since a composition of orthogonal updates is not necessarily orthogonal itself, and with momentum the updates are likely not orthogonal themselves.

At initialisation, the filters in a neural network layer activate on noise and so, to start, all the filters will attempt to learn the same function, the parametrisation with the highest discriminatory power. The most discriminatory parametrisation is the one that results in the best performing network if we reduce the layer to have only one filter. All the filters will attempt to learn this parametrisation first since it will reduce the loss most significantly, *i.e.* the gradient component will be largest in this direction, this is inefficient as all the components will attempt to co-learn this one parametrisation and will have to, at some point, pivot and learn a different function later on — not only wasting the initial updates, but making it harder to learn the next parametrisation.

This is analogous to Biehl *et al.*'s [147] observation that for a tiny MLP ($\mathbb{R}^N \to \mathbb{R}^2 \to \mathbb{R}$) trained from an identically-structured teacher MLP that the student model will initially learn hidden nodes with high correlation to both of the teacher nodes simultaneously [147, Figure 1], reproduced here in figure 5.2. "[T]he student vectors are almost identical and have — apart from small deviations — the same overlap with each teacher vector." [147], then, after some time at this plateau, they diverge to correlate separately with the teacher's hidden nodes.

### 5.3.2 Gradient Self-orthogonalisation

Given a neural network, $f$, with $L$ layers, made from filters, or components, $c$,

$$f_l(x) = [c_{l1}(x), c_{l2}(x), \ldots, c_{lN_l}(x)], \tag{5.3}$$

where $N_l$ is the number of filters in layer $l$, $c_l : \mathbb{R}^{S_{l-1} \times N_{l-1}} \to \mathbb{R}^{S_l}$ is a parametrised function and $c_{li}$ denotes $c_l$ parametrised with $\theta_{li} \in \mathbb{R}^{P_l}$ giving $f_l : \mathbb{R}^{S_{l-1} \times N_{l-1}} \to \mathbb{R}^{S_l \times N_l}$ parametrised by $\theta_l \in \mathbb{R}^{P_l \times N_l}$.

Let

$$G_l = [\nabla c_{l1}, \nabla c_{l2}, \ldots, \nabla c_{lN_l}], \tag{5.4}$$

be the $P_l \times N_l$ matrix of the filters' gradients.

Then, to find the nearest orthonormal matrix to $G_l$, we find the matrix, $O_l$, that minimises the Frobenius norm of its difference from $G_l$

$$\min_{O_l} \|O_l - G_l\|_F \quad \text{subject to } \forall i, j : \left\langle O_{li}, O_{lj} \right\rangle = \delta_{ij}, \tag{5.5}$$

where $\delta_{ij}$ is the Kronecker delta function. The solution is the product of the left and right singular vector matrices from the Singular Value Decomposition (SVD) of $G_l$ [148],

$$G_l = U_l \Sigma_l V_l^{\mathsf{T}}, \tag{5.6}$$
$$O_l = U_l V_l^{\mathsf{T}}. \tag{5.7}$$

Thus, we can adjust a first-order gradient descent method, such as SGDM [19], to make steps where the components are pushed in orthogonal directions,

$$v_l^{(t+1)} = \gamma v_l^{(t)} + \eta O_l^{(t)}, \text{ and} \tag{5.8}$$
$$\theta_l^{(t+1)} = \theta_l^{(t)} - v_l^{(t+1)}, \tag{5.9}$$

where $v_l$ is the velocity matrix, $t \in \mathbb{Z}^{0+}$ is the time, $\gamma$ is the momentum decay term, and $\eta$ is the step size. We call this method Orthogonal Stochastic Gradient Descent with Momentum (Orthogonal-SGDM). There are many variants of first-order optimisation algorithms, each with different preconditions, guarantees, and computation profiles. Our modification is completely portable and can clearly be applied in a black-box fashion to any first-order optimisation algorithm by replacing the gradients with $O_l^{(t)}$ before the calculation of the next iterate.

Pseudocode for the orthogonalised algorithm can be found in algorithm 6. Python code for creating orthogonal optimisers in PyTorch is provided at `https://github.com/MarkTuddenham/Orthogonal-Optimisers`. And python code

for the experiments in this chapter is provided at
https://github.com/MarkTuddenham/Orthogonalised-Gradients

**Algorithm 6 Orthogonalised Optimisers**
Pseudocode showing the use of gradient self-orthogonalisation with a standard optimiser, here the optimiser takes the current weights and gradients as inputs and produces the weight updates as output while holding some internal state.

**Require:** $f : \Theta \times \Theta \to \Theta$ ▷ An optimiser, where $\Theta$ is the model state space
  **for** $t = 0$ to $T$ **do**
    $g_t \in \Theta \leftarrow \nabla \mathcal{L}_\theta(\cdot)$
    $g'_t \leftarrow g_t$
    **for** $p = 0$ to $P$ **do** ▷ For each parameter group
      **if** $g_t[p]$ is the weights in a convolutional or linear layer **then**
        $G \leftarrow \text{flatten}_1(g_t[p])$ ▷ Flatten tensor starting from dimension 1
        $U, \Sigma, V^T \leftarrow \text{SVD}(G)$
        $O \leftarrow UV^T$
        $g'_t[p] \leftarrow \text{reshape\_as}(O, g_t[p])$ ▷ Reshape orthogonalised gradients to their original shape
      **end**
    **end**
    $\theta_{t+1} \leftarrow \theta_t - f(\theta_t, g'_t)$ ▷ update parameters using provided optimiser
  **end**

## 5.4 Results

### 5.4.1 Residual Networks on CIFAR-10

Firstly, we test the efficacy of our method in reproducing the results of a resnet-20 from the original ResNet paper [30] on CIFAR-10 [29]. In this work we are concerned with just the full pre-activation version of residual networks and the original identity mappings [63] for shortcut connections.

The resnet-20 was trained on the full train split — 40k images — of CIFAR10 with a batch size of 1024. We ran a grid search over the following hyper-parameter sets, learning rate $\in \{1e{-}4, 1e{-}3, 1e{-}2, 1e{-}1, 5e{-}5, 5e{-}4, 5e{-}3, 5e{-}2, 5e{-}1\}$, momentum $\in \{0.5, 0.75, 0.8, 0.85, 0.9, 0.95\}$ and weight decay $\in \{1e{-}4, 1e{-}3, 1e{-}2\}$. For all the Adam variants the second beta was fixed at 0.99 and the epsilon value was fixed at PyTorch's default — $1e{-}8$, experiments with gradient clipping were always set at 1.0. Additionally, both cosine annealing and multi-step, at epoch 100 and 150 with $\gamma = 0.1$, learning rate scheduling were tried with the multi-step scheduling coming out on top in all cases.

To compare against AdamW we trained a resnet-18 instead of a resnet-20 as the decoupled weight decay in AdamW is more suited to tasks where the model is vastly over-parametrised and as such failed to train the resnet-20 to any reasonable

TABLE 5.1: Test loss and accuracy of a ResNet-20 trained with different optimisers across five runs on CIFAR-10. † are the results taken directly from He *et al.* [30]. SGDM uses the hyper-parameter set: batch size of 1024, learning-rate of 0.05, momentum of 0.85, weight-decay of $10^{-2}$, and a learning rate schedule of $\times 0.1$ at epochs 100, 150 for 200 epochs. Adam uses the hyper-parameter set: batch size of 1024, learning-rate of 0.01, beta one of 0.9, beta two of 0.99, weight-decay of $10^{-4}$, and a learning rate schedule of $\times 0.1$ at epochs 100, 150 for 200 epochs. SGDM with gradient clipping uses the hyper-parameter set: batch size of 1024, learning-rate of 0.05, momentum of 0.85, weight-decay of $10^{-4}$, a max gradient of 1.0, and a learning rate schedule of $\times 0.1$ at epochs 100, 150 for 200 epochs. Adam with gradient clipping uses the hyper-parameter set: batch size of 1024, learning-rate of 0.01, beta one of 0.9, beta two of 0.99, weight-decay of $10^{-4}$, a max gradient of 1.0, and a learning rate schedule of $\times 0.1$ at epochs 100, 150 for 200 epochs. Results for AdamW train a resnet-18 instead as AdamW is more suited to tasks where the model is vastly over-parametrised. AdamW with gradient clipping uses the hyper-parameter set: batch size of 128, learning-rate of $10^{-5}$, beta one of 0.9, beta two of 0.99, weight-decay of $10^{-3}$, a max gradient of 1.0, and a learning rate schedule of $\times 0.1$ at epochs 100, 150 for 200 epochs. Orthogonal-AdamW with gradient clipping uses the hyper-parameter set: batch size of 128, learning-rate of $10^{-5}$, beta one of 0.9, beta two of 0.99, weight-decay of $10^{-3}$, a max gradient of 1.0, and a learning rate schedule of $\times 0.1$ at epochs 100, 150 for 200 epochs. NB. there are two bold results for the two networks, AdamW on resnet18s and the rest on resnet20, since resnet18 is larger it is expected to perform better.

| Optimiser | Test Loss | Test Acc (%) |
|---|---|---|
| SGDM† | — | 91.25 |
| SGDM | $0.3062 \pm 0.0036$ | $91.45 \pm 0.09$ |
| Adam | $0.4230 \pm 0.0039$ | $91.67 \pm 0.10$ |
| SGDM w/ grad clipping | $0.5025 \pm 0.0073$ | $86.60 \pm 0.05$ |
| Adam w/ grad clipping | $0.4322 \pm 0.0091$ | $91.73 \pm 0.23$ |
| Orthogonal-SGDM | $0.3553 \pm 0.0045$ | $\mathbf{92.08} \pm 0.05$ |
| Orthogonal-Adam | $0.6192 \pm 0.0029$ | $90.86 \pm 0.21$ |
| Orthogonal-SGDM w/ grad clipping | $0.7259 \pm 0.0108$ | $90.06 \pm 0.10$ |
| Orthogonal-Adam w/ grad clipping | $0.6487 \pm 0.0118$ | $90.73 \pm 0.07$ |
| AdamW w/ grad clipping | $0.2926 \pm 0.0101$ | $\mathbf{93.23} \pm 0.17$ |
| Orthogonal-AdamW w/ grad clipping | $0.7364 \pm 0.0078$ | $89.58 \pm 0.12$ |

performance. For this we used a fixed batch size of 128, learning rate $\in \{1e{-}6, 1e{-}5, 1e{-}4, 1e{-}3\}$, beta one $\in \{0.5, 0.75, 0.8, 0.85, 0.9, 0.95\}$, beta two of 0.99, weight decay $\in \{1e{-}6, 1e{-}5, 1e{-}4, 1e{-}3, 1e{-}2\}$, a maximum gradient of 1.0, and a multi-step learning rate schedule of $\times 0.1$ at epochs 100, 150 for 200 epochs.

Each of these experiments were run twice and from this the set with the best final validation accuracy was chosen and run three more times, producing the results in table 5.1.

Table 5.2 shows the main advantage of gradient self-orthogonalisation, that is, a decrease in the number of iterates to obtain a certain accuracy. We can see that, across most pairings of accuracy checkpoints and optimiser that the orthogonal version reaches the said accuracy in less epochs, the major exception to this being Adam in the

TABLE 5.2: Number of Epochs Required to Reach a Given Validation Accuracy for Different Optimizers. This table presents the number of training epochs required for various optimisers to reach specific validation accuracy thresholds (50%, 75%, 80%, 85%, and 90%). The optimisers compared include SGDM, Adam, and their respective variants with gradient clipping, as well as their orthogonalised versions. The hyper-parameter set for these are laid out in table 5.1. Baseline optimisers (SGDM and Adam): Adam generally reaches milestones faster than SGDM, particularly at higher accuracy thresholds. Applying gradient clipping to SGDM reduces the required epochs for 50%-80% accuracy but fails to train to the highest threshold (90%). For Adam, gradient clipping does not significantly accelerate learning. Orthogonal variants: Orthogonal-SGDM reaches 50% validation accuracy after just 1 epoch, significantly faster than other optimisers, and achieves 90% in 101 epochs, matching standard Adam. Orthogonal-Adam exhibits even greater efficiency in early training stages, reaching 75% in just 5 epochs and 80% in 8 epochs, the fastest amongst all methods. The combination of orthogonal gradients and gradient clipping generally leads to more stable training, though with some trade-offs at higher accuracy levels. For instance, Orthogonal-Adam with gradient clipping reaches 85% in just 18 epochs.

| Optimiser | 50% | 75% | 80% | 85% | 90% |
|---|---|---|---|---|---|
| SGDM | 4 | 44 | 100 | 100 | 150 |
| Adam | 4 | 17 | 20 | 88 | **101** |
| SGDM w/ grad clipping | 4 | 23 | 50 | 101 | — |
| Adam w/ grad clipping | 4 | 14 | 27 | 101 | 104 |
| Orthogonal-SGDM | **1** | 9 | 48 | 100 | **101** |
| Orthogonal-Adam | 2 | **5** | **8** | 22 | 105 |
| Orthogonal-SGDM w/ grad clipping | 2 | 7 | 11 | 26 | 108 |
| Orthogonal-Adam w/ grad clipping | 2 | 7 | 9 | **18** | 109 |

late stages of training, where Adam reaches 90% accuracy after 101 epochs whilst Orthogonal-Adam reaches 90% accuracy after 105. The decrease is most notable at the start of training where some orthogonal optimisers take 4-5x fewer epochs to reach 80/85% accuracy. Note also the rapid decrease in loss at the 100 epoch mark; this is due to the learning rate scheduling that reduces the loss for the 100th epoch. This phenomenon appears noticeably for Orthogonal-SGDM, however, it is not as apparent for the other orthogonalised optimisers as these reach 85% accuracy with their starting learning rates.

While faster convergence is valuable, it's crucial that the new optimiser matches the final performance of established methods, as accelerated training without comparable terminal accuracy risks trading speed for suboptimal generalization; table 5.1 shows that the orthogonalised algorithms are also close in final performance with their original counterparts. For instance, we can reproduce the performance given in the original paper [30] with our setup, and our experiments on resnet-20 are all within one percentage point or above their non orthogonalised versions and are in the same ballpark as the state-of-the-art results [149]. Unfortunately, the experiments with AdamW on a resnet-18 fall further short of the benchmark performance, this is likly due to the effect of small batch sizes on the orthogonalised optimisers which we go into more detail in section 5.5.7.

These results suggest that the gradient self-orthogonalisation modification, especially when combined with Adam, can significantly enhance training efficiency, particularly in the early-to-mid stages of optimisation whilst retaining similar end-of-training performance.

#### 5.4.1.1 Untuned optimisation

In this section we look at untuned optimisation, that is, instead of choosing the hyper-parameter set and model that achieves the best validation loss, we look at a set experiments holistically. Bellow are two experiments that look over a suite of models and common starting hyper-parameter sets.

Firstly, we train a suite of models on the CIFAR-10 data set with a mini-batch size of 1024, learning-rate of $10^{-2}$, momentum of 0.9, and a weight decay of $5 \times 10^{-4}$ for 100 epochs, and plot the results in figures 5.3 and 5.4 and table 5.3. Orthogonal-SGDM is more efficient and achieves better test accuracy than SGDM for every model we trained in this experiment. The validation curves, figures 5.5 and 5.6, follow the training curves, figures 5.3 and 5.4, and have the same patterns, this implies that Orthogonal-SGDM exhibits the same generalisation performance as SGDM. More importantly though, we can see that the model learns much faster at the beginning of training, as shown by figure 5.3, this means that we do not need as many epochs to get to a well-performing network. While still requiring hyper-parameter tuning, this can allow researchers to more quickly iterate on ideas and model architectures. This is especially good in light of the large data sets that new models are being trained on, where they are trained for only a few epochs, or less [27]. The performance of the residual networks designed for ImageNet [8] ResNet-(18, 34, 50) under SGDM get worse with the increasing model's size. The original ResNet authors, He *et al.* [30], note that unnecessarily large networks may over-fit on a small data set such as CIFAR-10. However, with Orthogonal-SGDM, these models do not suffer from this over-parametrisation problem and, in clear contrast to SGDM, even slightly improve in performance as the models get bigger. This agnosticism to over-parametrisation helps alleviate the need for the practitioner to tune a model's architecture to the task at hand to achieve good feedback on their experiments.

Next, we compare SGDM, Orthogonal-SGDM, Adam, and Orthogonal-Adam over several hyper-parameter sets, table 5.4. Amongst our hyper-parameter sets we include the default hyper-parameters for both of the most common frameworks, *i.e.* PyTorch [104] and TensorFlow [150], which are the same. Note, both frameworks have a default momentum of 0 as they view SGDM as an extension of SGD so we show both 0 and the momentum value they suggest in their tutorials.

[1] As described in appendix D.0.1
[2] Model same as in He *et al.* [30]
[3] From `https://pytorch.org/vision/0.9/models.html`

TABLE 5.3: Test loss and accuracy across a suite of models on CIFAR-10 comparing normal SGDM with Orthogonal-SGDM, standard error across five runs.

| | Test Loss | | Test Accuracy (%) | |
|---|---|---|---|---|
| | SGDM | Orthogonal-SGDM | SGDM | Orthogonal-SGDM |
| BasicCNN[1] | 0.7603 ± 0.0061 | 0.6808 ± 0.0038 | 73.60 ± 0.19 | **76.67** ± 0.10 |
| ResNet-20[2] | 0.6728 ± 0.0301 | 0.6766 ± 0.0155 | 79.14 ± 0.62 | **87.12** ± 0.12 |
| ResNet-44[2] | 0.7000 ± 0.0166 | 0.7600 ± 0.0299 | 79.81 ± 0.37 | **88.12** ± 0.20 |
| ResNet-18[3] | 0.9656 ± 0.0104 | 0.8427 ± 0.0121 | 77.01 ± 0.21 | **84.68** ± 0.12 |
| ResNet-34[3] | 1.0468 ± 0.0134 | 0.7087 ± 0.0165 | 75.86 ± 0.26 | **85.42** ± 0.33 |
| ResNet-50[3] | 1.2304 ± 0.0462 | 0.6797 ± 0.0235 | 67.99 ± 0.73 | **86.51** ± 0.12 |
| DenseNet-121[3] | 1.0027 ± 0.0132 | 0.8669 ± 0.0132 | 75.26 ± 0.30 | **84.34** ± 0.15 |
| DenseNet-161[3] | 1.1399 ± 0.0096 | 1.1688 ± 0.1960 | 75.81 ± 0.20 | **85.51** ± 0.19 |
| ResNeXt-50_32x4d[3] | 1.2470 ± 0.0254 | 0.6669 ± 0.0223 | 68.73 ± 0.30 | **86.37** ± 0.24 |
| Wide_ResNet-50_2[3] | 1.4141 ± 0.0337 | 0.7018 ± 0.0091 | 69.42 ± 0.33 | **87.30** ± 0.12 |

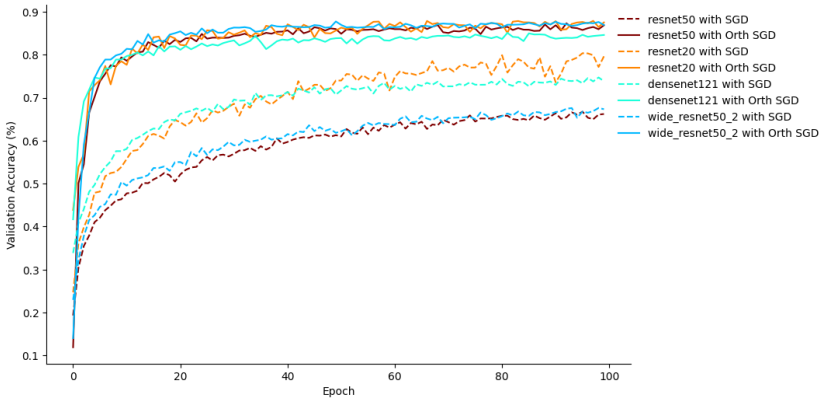



FIGURE 5.3: Validation accuracy from one run of SGDM vs Orthogonal-SGDM for a selection of models. Full plot in appendix G.1. Best viewed in colour.

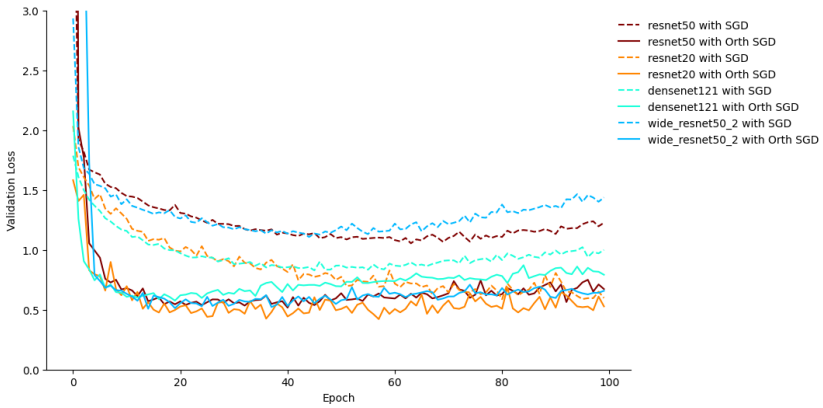



FIGURE 5.4: Validation losses from one run of SGDM vs Orthogonal-SGDM for a selection of models. Full plot in appendix G.1. Best viewed in colour.

From table 5.4 we see that, in general, the orthogonalised versions of SGDM and Adam outperform their original counterparts and that they survive a wider range of hyper-parameter values. However, none of these values match state-of-the-art results,

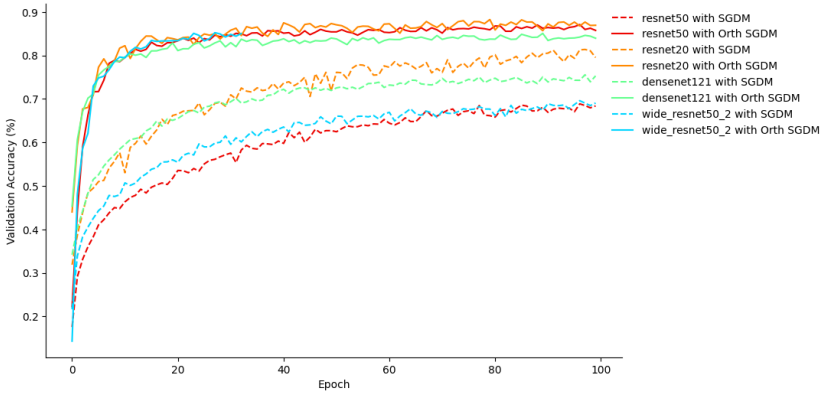



FIGURE 5.5: Train accuracy from one run of SGDM vs Orthogonal-SGDM for a selection of models. Best viewed in colour.

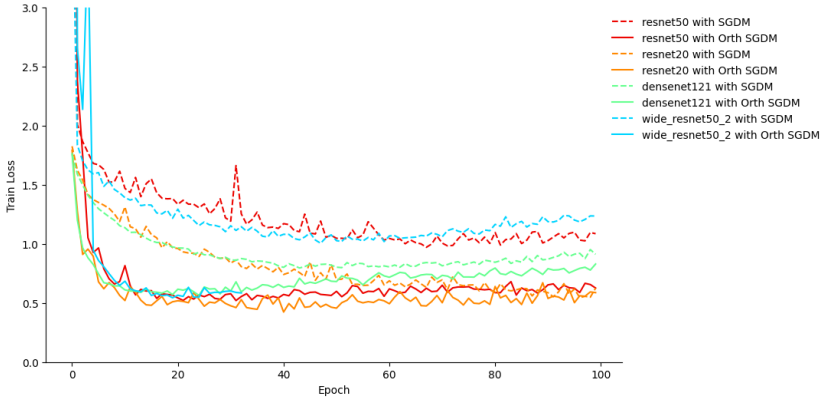



FIGURE 5.6: Train losses from one run of SGDM vs Orthogonal-SGDM for a selection of models. Best viewed in colour.

showing that tuning is still necessary for best results. An additional benefit presented by table 5.4 is a potential reduction in memory needed to train models. All first order method have to store $2n$ parameters, where $n$ is the size of the model, the model itself and the calculated gradients; however, when an optimiser uses an extra buffer we need an additional $n$ memory. All oft-used optimisers also use an extra buffer to store either some cumulation of previous gradients (usually called the momentum buffer), *e.g.* SGDM, taking $3n$-memory, or a buffer of some cumulation of the squared gradients, *e.g.* AdaGrad or RMSProp, again requiring $3n$-memory; in some cases the optimiser stores both these buffers *e.g.* Adam and AdamW using $4n$-memory. Orthogonal-SGDM and Orthogonal-Adam are able to obtain good results with 0 momentum, table 5.4, which means that this training run can be implemented using only $2n$-memory, this would be useful for training huge models – for example foundational language models — which need to be trained in a distributed fashion; for which the primary impediment is the communication cost and the reduction in memory needed to store the optimisation state will reduce synchronisation difficulties.

TABLE 5.4: Test accuracy, and standard error across five runs, for a suite of hyper-parameter sets on CIFAR-10 on a ResNet-20 for 100 epochs using a batch size of 1024 and a weight-decay of $5 \times 10^{-4}$, accuracy across five runs. For Adam $\beta_2 = 0.999$. We point out the default parameters for the frameworks with †; both frameworks have a default momentum of 0 as they view SGDM as an extension of SGD so we show both 0 and the momentum value they suggest in their tutorials.

| | | SGDM | | Adam | |
|---|---|---|---|---|---|
| LR | Mom | Original | Orthogonal | Original | Orthogonal |
| $10^{-1}$ | 0.95 | 83.59± 2.09 | **85.58**± 0.98 | 38.95± 8.95 | 76.84± 1.53 |
| $10^{-2}$ | 0.95 | 82.66± 1.02 | **87.72**± 0.44 | 74.23± 2.38 | 86.48± 0.17 |
| $10^{-3}$ | 0.95 | 66.59± 0.44 | **85.88**± 0.33 | 83.08± 0.76 | 85.12± 0.06 |
| $10^{-4}$ | 0.95 | 41.88±0.05 | **77.93**±0.20 | 73.73±0.20 | 76.46±0.14 |
| $10^{-1}$ | 0.9 | 82.52± 1.16 | **85.06**± 0.47 | 28.26± 7.16 | 73.62± 2.96 |
| $10^{-2}$ | 0.9 | †79.96± 0.48 | **87.44**± 0.25 | 73.46± 1.19 | 85.26± 0.38 |
| $10^{-3}$ | 0.9 | 60.69± 0.18 | 84.67± 0.21 | †83.16± 0.66 | **85.25**± 0.31 |
| $10^{-4}$ | 0.9 | 36.83±0.05 | 70.40±0.16 | 72.99±0.09 | **76.31**±0.16 |
| $10^{-1}$ | 0.8 | 84.16± 0.43 | **86.01**± 0.74 | 27.50± 6.76 | 71.88± 4.25 |
| $10^{-2}$ | 0.8 | 77.42± 0.98 | **87.18**± 0.12 | 72.60± 1.76 | 86.75± 0.26 |
| $10^{-3}$ | 0.8 | 53.21± 0.43 | 82.95± 0.40 | 80.89± 1.93 | **85.52**± 0.29 |
| $10^{-4}$ | 0.8 | 31.92±0.08 | 61.90±0.09 | 73.16±0.21 | **76.58**±0.12 |
| $10^{-1}$ | 0.5 | 80.08± 0.36 | **87.37**± 0.18 | 18.39± 4.84 | 72.10± 2.83 |
| $10^{-2}$ | 0.5 | 68.64± 1.05 | **86.05**± 0.10 | 71.62± 1.95 | 84.22± 0.69 |
| $10^{-3}$ | 0.5 | 43.51± 1.02 | 78.68± 0.77 | 81.67± 1.05 | **84.21**± 0.43 |
| $10^{-4}$ | 0.5 | 25.73±0.05 | 51.89±0.06 | 72.50±0.03 | **76.65**±0.20 |
| $10^{-1}$ | 0 | 74.42± 0.63 | **83.80**± 0.32 | 10.00± 0.00 | 55.14± 6.97 |
| $10^{-2}$ | 0 | †53.42± 0.20 | **82.18**± 0.55 | 60.47± 3.70 | 80.50± 1.04 |
| $10^{-3}$ | 0 | 32.41± 0.57 | 63.25± 0.57 | 76.76± 2.09 | **83.45**± 0.40 |
| $10^{-4}$ | 0 | 21.17± 0.07 | 43.59± 0.10 | 71.55± 0.14 | **76.24**± 0.17 |

### 5.4.2 ImageNet, Barlow Twins, & Poisson-VAE

While we do not have access to the compute power to train very accurate ImageNet models, we can show that Orthogonal-SGDM has the same properties on a larger dataset that it does on the smaller CIFAR-10.

Figure 5.7 demonstrates that Orthogonal-SGDM also works on a large data set such as ImageNet [8]; using a ResNet-34, mini-batch size of 1024, learning rate of $10^{-2}$, momentum of 0.9, and a weight decay of $5 \times 10^{-4}$, for 100 epochs. SGDM achieves a test accuracy of 61.9% and a test loss of 1.565 while Orthogonal-SGDM achieves 67.5% and 1.383 respectively. While these results are some distance from the capabilities of the model they still demonstrate a significant speed-up and improvement from using Orthogonal-SGDM, especially at the start of learning, comparable to the CIFAR-10 results; and further reinforces how a dearth of hyper-parameter tuning impedes performance.

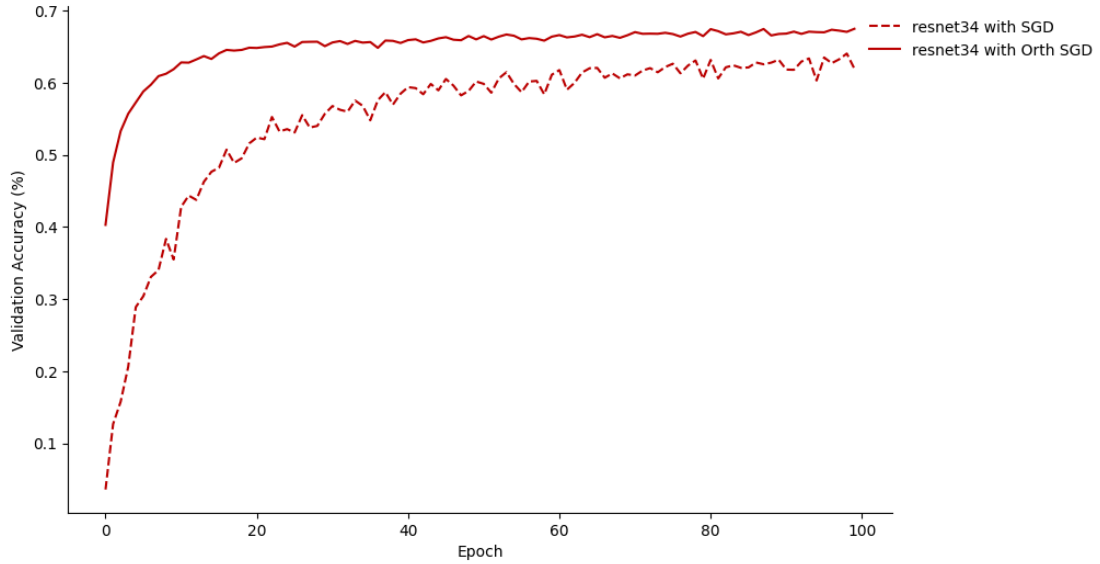



FIGURE 5.7: Validation accuracy of SGDM vs Orthogonal-SGDM on ImageNet for a ResNet-34, mini-batch size of 1024, learning rate of $10^{-2}$, momentum of 0.9, and a weight decay of $5 \times 10^{-4}$, for 100 epochs.

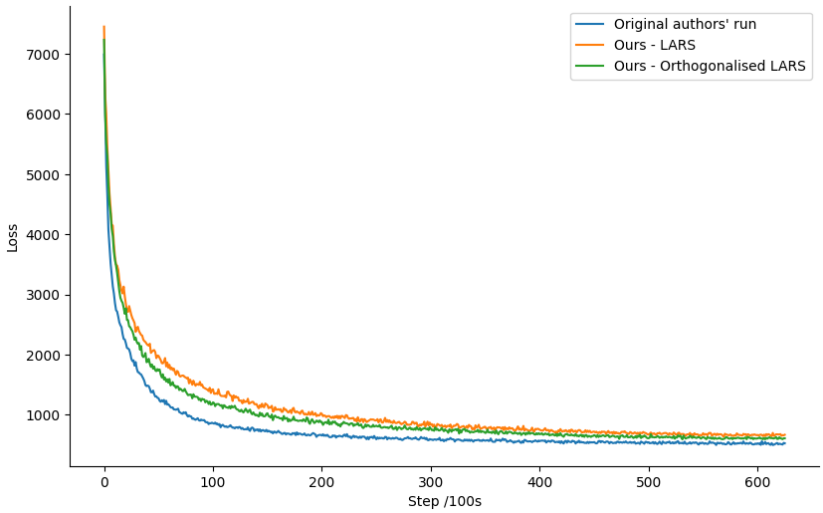



FIGURE 5.8: Barlow Twins loss during the unsupervised phase using LARS and Orthogonal-LARS on ImageNet

Barlow Twins [151] is a self-supervised method that uses "the cross-correlation matrix between the outputs of two identical networks fed with distorted versions of a sample" to avoid collapsing to trivial solutions. While the authors do provide code, we could not replicate the performance of their results by running it. To train within our compute limitations we used a mini-batch size of 1024 instead of 2048 however this should not significantly affect the results since "Barlow Twins does not require large batches" [151]. Additionally, as Barlow Twins uses the Layer-wise Adaptive Rate Scaling (LARS) algorithm [152], which is designed to adjust the learning rate based on the ratio between the magnitudes of the gradients and weights, there should be no

noticeable slow-down, or speed-up, in learning due to the mini-batch size. We do not orthogonalise the gradients for the dense layers (see section 5.5.6).

Orthogonalising gradients is a natural complement to LARS because it addresses a key limitation in how LARS modulates update magnitudes. LARS adjusts each layer's update direction by scaling the gradient norm to match the parameter norm, effectively controlling learning rate at the layer level,

$$\Delta\theta^{(l)} \propto \frac{\|\theta^{(l)}\|}{\|g^{(l)}\|} \cdot g^{(l)}$$

But this assumes that the direction of the gradient is meaningful — *i.e.* , that it is not redundant or dominated by a small subspace. Orthogonalising gradients between layer components helps preserve directional diversity, ensuring that updates span a richer set of directions, gradient information does not collapse into degenerate or highly aligned subspaces, and the per-layer scaling in LARS remains sensitive to meaningful geometric structure rather than noise or redundant drift. In effect, orthogonalisation improves gradient isotropy, which makes the layer-wise norm-based scaling in LARS more faithful to the true geometry of the optimisation landscape, particularly in over-parametrised models where gradients often align and collapse.

Comparing our own runs, we establish that self-orthogonalising the gradients before the LARS algorithm does speed up learning as shown in figure 5.8, in agreement with previous experiments. This is evidence that orthogonalising gradients is also beneficial for self-supervised learning and, moreover, it is likely that most first-order optimisation algorithms can be improved in this way.

Lastly, we apply our method to a Poisson Variational Autoencoder (P-VAE) [153], which is a generative model tailored for count-valued data, such as neural spike trains, word counts, or pixel intensities in low-bit-depth images. It extends the standard VAE framework by replacing the Gaussian likelihood in the decoder with a Poisson distribution, enabling a more natural modelling of sparse, non-negative observations. However, training VAEs with non-Gaussian likelihoods—particularly discrete or heavy-tailed ones like the Poisson—poses significant optimisation challenges due to high variance in the gradient estimators and complex curvature in the ELBO landscape.

In figure 5.9, we compare standard Adam with its orthogonalised variant on the task of training a P-VAE, note for VAEs we only orthogonalise the gradients for the encoder. Orthogonal-Adam exhibits a lower initial loss and converges faster than standard Adam, suggesting more efficient use of early gradients. While both methods reach similar asymptotic performance, the orthogonal method consistently achieves slightly better loss in the mid-to-late training regime. This indicates that enforcing

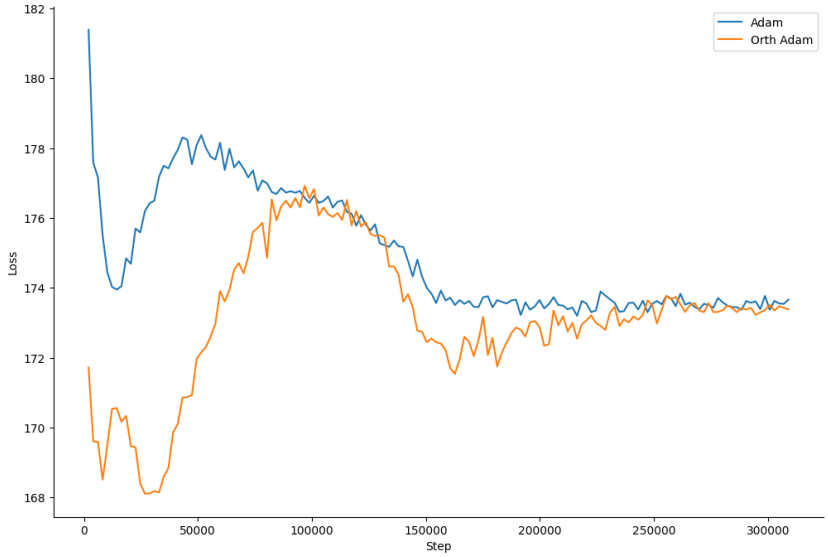



FIGURE 5.9: Evaluation MSE reconstruction loss over training steps for a Poisson Variational Autoencoder (P-VAE), comparing standard Adam (blue) and its orthogonalised variant (orange). The x-axis represents training steps, and the y-axis shows the MSE on held-out data. The orthogonalised optimiser achieves a lower reconstruction error earlier in training and maintains a consistent edge over standard Adam throughout most of the trajectory. The gap is most pronounced between steps 100k and 200k, suggesting more efficient navigation of the complex, high-variance optimisation landscape typical of P-VAEs. This supports the hypothesis that orthogonal gradient updates improve stability and convergence when training models with discrete or heavy-tailed likelihoods.

local directional diversity may help overcome transient curvature barriers or suboptimal basins that standard Adam drifts into. These results are consistent with our theoretical motivation, by encouraging updates to span a more diverse subspace, the model avoids redundant search directions and exhibits more direct convergence, especially in the presence of noisy or highly correlated gradients, as commonly observed in latent-variable models like the P-VAE

## 5.5 Analysis of the method

### 5.5.1 Similarity of component parametrisation

We hypothesised in section 5.1 that neural networks have components with parametrisations resulting in disproportionally large discriminatory powers and that the components would co-learn as a result of this. To see if there is evidence of the filers co-learning, we calculate the pair-wise cosine similarity of the weights.

Specifically, we look at the mean of the absolute cosine of all distinct pairs of different filters' parameters,

$$\frac{2}{N_l(N_l-1)}\sum_{i<j}\left|S_{\cos}\left((\theta_l)_i,(\theta_l)_j\right)\right|, \tag{5.10}$$

for some layer $l$.

We confirm, in figure 5.10, that with a high learning rate, $\eta = 1$, the similarity between the filters of various convolutional layers in a ResNet-20 on CIFAR-10 are significant (see appendix A.1.4 for details on this cosine threshold). Additionally, there is an initial spike in similarity that would indicate multiple components co-learning the same parametrisation, before being separated slowly during the remainder of training. Indeed, when we contrast with Orthogonal-SGDM, we see that the components' parameters are less similar than they were with SGDM and do not include the initial spike.

However, when the learning rate is lower, $\eta = 0.01$, the roles are reversed figure 5.11. We hypothesise that this is due to the minimal distance that SGDM traverses in weight space especially contrasted with Orthogonal-SGDM which adapts to a much larger step size. In addition, some co-learning may be important to have some similar or near-similar components to provide the latter layers with variations on themes *e.g.* when visualising GoogLeNet Olah *et al.* [154] find several different "floppy ear" detectors which are clearly similarly parametrised filters.

### 5.5.2 Diversified intermediary representations

Along with different parametrisations we also desire different intermediary latent features, a model will perform better if its layers output $N$ different representations as opposed to $N$ similar ones.

Let $x_l$ be the resulting representations from the intermediary layers,

$$x_l = f_l(x_{l-1}) \tag{5.11}$$

where $x_0$ is the input.

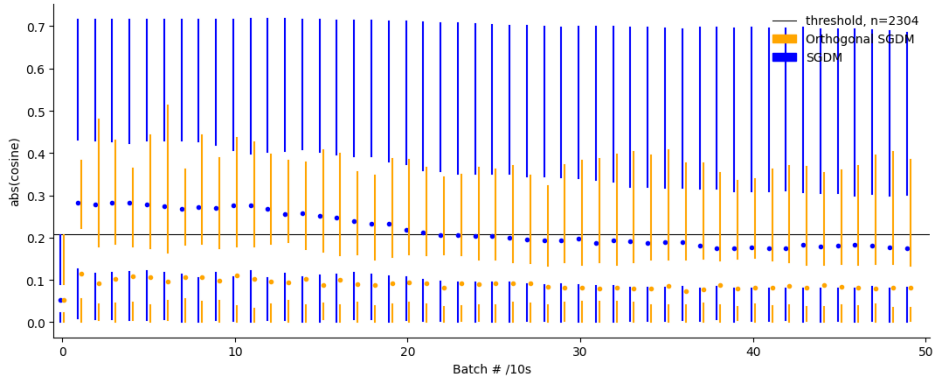
We look at the statistics of the absolute cosine of all distinct pairs of different latent features,

$$R_l = \left\{ \left| S_{\cos}\left( (x_l)_i, (x_l)_j \right) \right| \mid i < j \right\}. \tag{5.12}$$

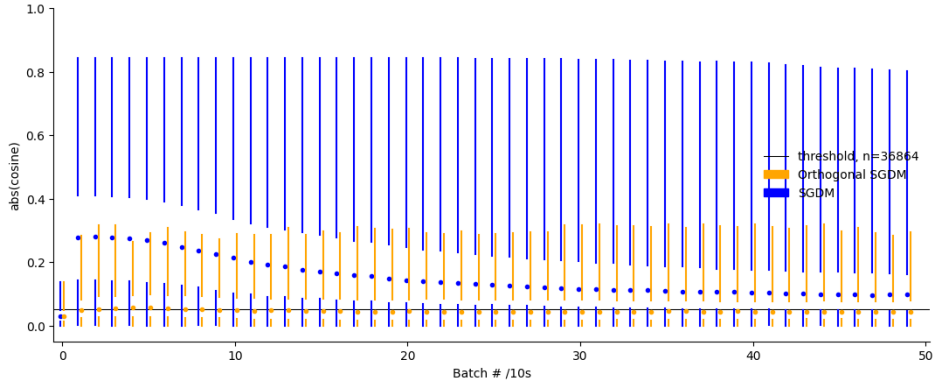
and see that the representations have smaller cosines, *i.e.* they are more diverse, when using Orthogonal-SGDM versus SGDM — figure 5.12. In addition, Orthogonal-SGDM shows a decline in cosine similarity throughout training while SGDM does not; this is likely due to Orthogonal-SGDM having a higher regularisation effect and indicates that more information is being passed to the next layer as the network is trained.

### 5.5.3 Gradient normalisation

When we perform SVD on the reshaped gradient tensor, we obtain an orthonormal matrix, since this changes the magnitude of the resultant update vector we look at the




(A) Distribution of the cosine matrix for second convolution of the third block in the first layer of a ResNet-20.

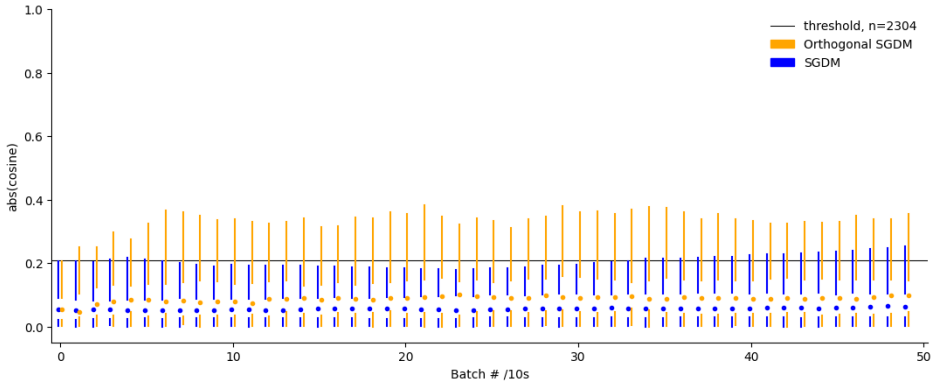



(B) Distribution of the cosine matrix for the first convolution of the second block in the fourth layer of a ResNet-20.

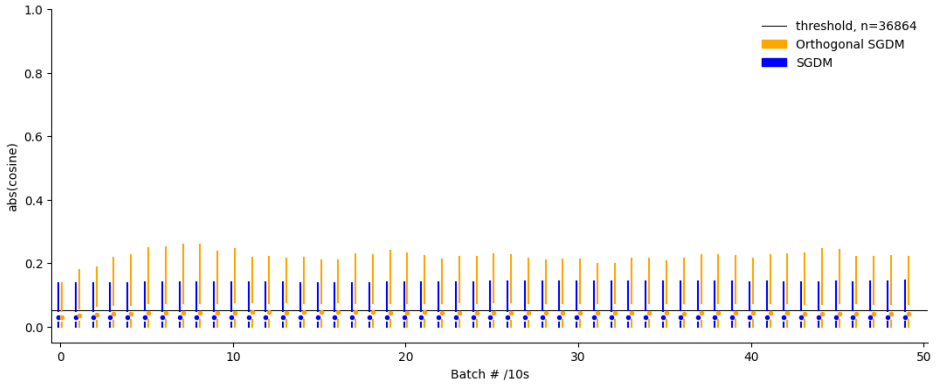
FIGURE 5.10: Box plots showing the distribution of cosine similarities between gradient vectors in selected convolutional layers of a ResNet-20 during training with learning rate $\eta = 1$. Each box represents the distribution of pairwise cosine values at a given training batch, computed from gradients within the specified layer. Panel (A) corresponds to the second convolution of the third block in the first layer, and Panel (B) to the first convolution of the second block in the fourth layer. The blue boxes denote standard SGDM, while orange boxes show gradients under the orthogonalised SGDM update. The horizontal dashed line indicates the random vector similarity threshold for the corresponding dimensionality. Orthogonal SGDM consistently produces gradient updates with significantly lower mutual cosine similarity (*i.e.* , higher angular diversity), demonstrating the effectiveness of the method in reducing directional redundancy across batches.

effect of this normalisation. Normalised-SGDM (N-SGDM) [155] provides an improvement in non-convex optimisation since it is difficult to get stuck in a local minimum as the step size is not dependent on the gradient magnitude. However, it hinders convergence to a global minimum since there is no way of shortening the step size; however, deep learning is highly non-convex and is unlikely to be optimised to a global minimum. Therefore, it stands to reason that normalising the gradient would speed up the optimisation of deep networks.

In addition, we compare N-SGDM to normalising the gradients per component — *i.e.* normalising the columns of $G_l$, equation (5.4) without orthogonalising them —




(A) Distribution of the cosine set, $R_l$, for second convolution of the third block in the first layer of a ResNet-20.




(B) Distribution of the cosine set, $R_l$, for the first convolution of the second block in the fourth layer of a ResNet-20.

FIGURE 5.11: Box plots showing the distribution of the cosine similarity set $R_l$, which consists of the absolute values of pairwise cosine similarities between gradient vectors within layer $l$, over the course of training on a ResNet-20 with learning rate $\eta = 0.01$. Each box captures the distribution at a given batch (in increments of 10), allowing comparison of directional similarity under different optimisers. Panel (A) shows results for the second convolution of the third block in the first layer; Panel (B) shows the first convolution of the second block in the fourth layer. In both cases, orange box plots represent the Orthogonal SGDM method, while blue box plots correspond to standard SGDM. The horizontal line denotes the expected cosine similarity between random unit vectors in the corresponding parameter space (based on dimensionality $n$). Orthogonal SGDM consistently yields a higher and more dispersed distribution of cosine similarities, indicating its explicit deviation from random-gradient alignment and a structural bias toward increased angular diversity in updates.

Component Normalised SGDM (CN-SGDM). N-SGDM improves over SGDM, and CN-SGDM might improve over N-SGDM except for the oft-case where it diverges. Finally, Orthogonal-SGDM obtains the best solutions while remaining stable on all the models.

[4] As described in appendix D.0.1

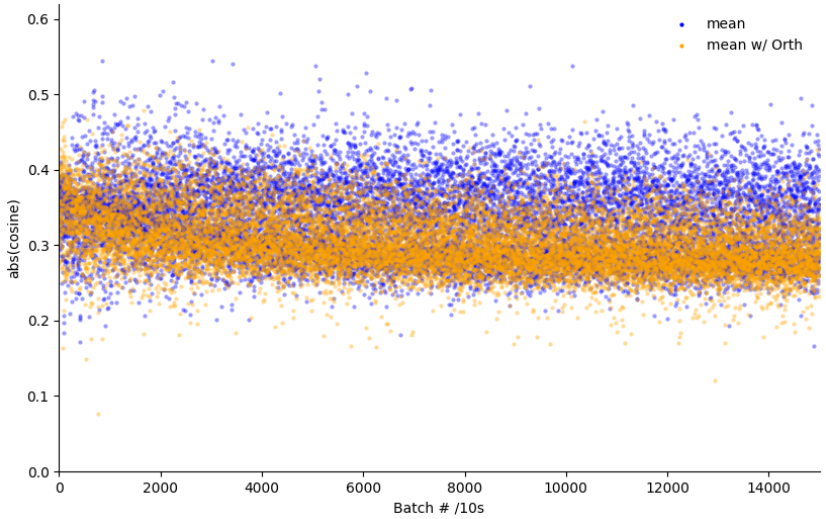



FIGURE 5.12: Mean of the absolute cosine of all distinct pairs of different intermediary representations, $\mathbb{E}[R_l]$, $l \in \{1,2,3\}$, for all layers of a BasicCNN trained on CIFAR-10 as in section 5.4.1.

TABLE 5.5: Test accuracy for several models trained with SGDM, Normalised-SGDM, Component Normalised SGDM, and Orthogonal-SGDM; trained as in section 5.4.1.

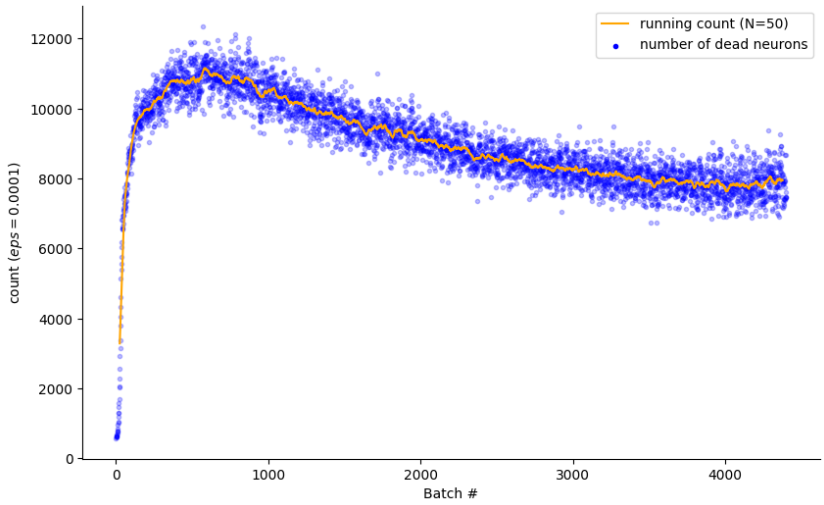
| | SGDM | N-SGDM | CN-SGDM | Orthogonal-SGDM |
|---|---|---|---|---|
| BasicCNN[4] | 73.68± 0.27 | 73.72± 0.45 | 74.53± 0.32 | **76.75**± 0.23 |
| ResNet-18 | 76.83± 0.22 | 78.94± 0.19 | 0.00± 0.00 | **84.94**± 0.10 |
| ResNet-50 | 69.35± 0.30 | 79.35± 0.21 | 0.00± 0.00 | **86.59**± 0.10 |
| ResNet-44 | 79.73± 1.27 | 83.60± 0.77 | 84.44± 0.55 | **87.49**± 0.39 |
| densenet121 | 75.45± 0.20 | 79.06± 0.04 | 0.00± 0.00 | **84.86**± 0.07 |

### 5.5.4 Disabled parameters

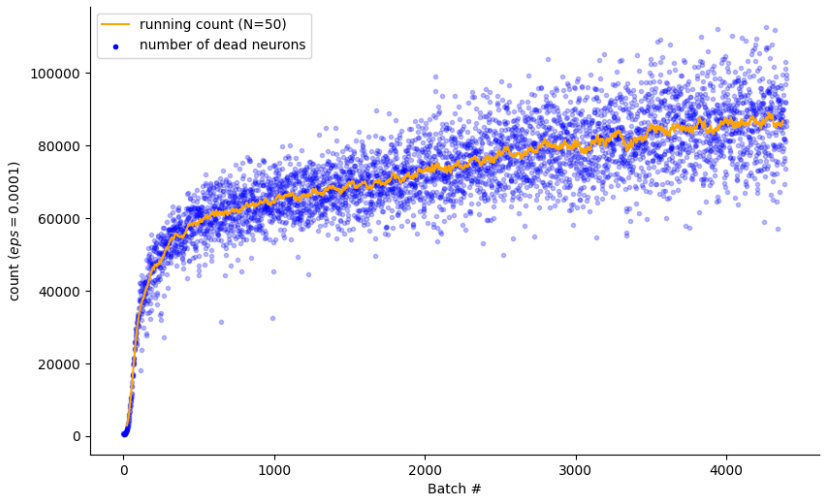
Disabled parameters occur when the activation function has a part with zero gradient, *e.g.* a ReLU. If the result of the activation remains in this part, then the gradients of all the preceding parameters will be zero and prevented from learning, this can limit the model's capacity, however, parameters whose activation function doesn't "activate" for a single batch, called Temporarily Disabled Parameters (TDP), can be beneficial and act as a regulariser, similar to dropout. To detect TDP, we simply look for parameters with zero gradient. Comparing the amount of TDP produced by SGDM versus Orthogonal-SGDM — figures 5.13a and 5.13b respectively — shows that Orthogonal-SGDM ends with around and order of magnitude more TDP. This implies a much higher regularisation which helps to explain Orthogonal-SGDM's insensitivity to over-parametrisation.

### 5.5.5 Decomposition implementation

While the QR decomposition is the most common orthogonalisation method; here, it is, in practice, less stable than SVD as the gradients are rank deficient [156, Section 3.5],




(A) With SGDM.




(B) With Orthogonal-SGDM.

FIGURE 5.13: Number of TDP for the layer "layer2[1].conv2" of a ResNet-50 trained as in section 5.4.1.

*i.e.* they have at least one small singular value. Orthogonal-SGDM has a longer wall time than SGDM because of the added expense of the SVD which has non-linear time complexity in the matrix size. In practice, we have found that the calculation of the SVD is either more than made up for by the speed-up in iterates or a prohibitively expensive cost, with dense layers being the largest and thus most problematic.

Whilst there exist methods for computing an approximate SVD which are faster, we have used PyTorch's default implementation since we are more concerned with Orthogonal-SGDM's performance and efficiency in iterates and not in wall time. Even so the overhead is small, training a ResNet-20 as in section 5.4.1 takes 790.3 seconds with 96.4 of them taken up by the SVD calculation — an increase of 12.2% over normal SGDM. While this is a not insignificant amount of time we can see that our method can take less than 5% of the number of epochs to reach the same accuracy — figure 5.3.

It is doubtful that convergence of the SVD is needed, so a custom matrix orthogonalisation algorithm, that has the required stability but remains fast and approximate, will reduce the computation overhead significantly and may allow previously infeasible networks to be optimised using Orthogonal-SGDM. However, we note that even with a more suitable implementation, our method would still bias towards many smaller layers for a deeper, thinner network.

### 5.5.6 Fully connected layers

Fully connected or dense layers also fit our component model from equation (5.3) where the components are based on the inner product of the input and the parametrisation,

$$c_{li}(x) = \sigma(\langle \text{flatten}(x), \theta_{li} \rangle), \tag{5.13}$$

where $\sigma$ is an activation function, $S_l = 1$ giving $f_l : \mathbb{R}^{S_{l-1} \times N_{l-1}} \to \mathbb{R}^{N_l}$ and $\theta_l \in \mathbb{R}^{S_{l-1} \cdot N_{l-1} \times N_l}$ as desired [157]. Intuitively, each column of the weight matrix acts as a linear map resulting in one item in the output vector. Thus, the gradients of fully connected layers can also be self-orthogonalised.

TABLE 5.6: Test accuracies and losses on CIFAR-10 when orthogonalising all layers vs orthogonalising just the convolutional layers. Trained as in section 5.4.1, standard error across five runs.

| Accuracy | SGDM | Orthogonal-SGDM | Conv Orthogonal-SGDM |
|---|---|---|---|
| BasicCNN | 73.60 ± 0.19 | 76.67 ± 0.10 | **76.80** ± 0.18 |
| ResNet-34 | 75.86 ± 0.26 | 85.42 ± 0.33 | **85.68** ± 0.21 |
| ResNet-20 | 79.14 ± 0.62 | 87.12 ± 0.12 | **87.70** ± 0.40 |
| **Loss** | **SGDM** | **Orthogonal-SGDM** | **Conv Orthogonal-SGDM** |
| BasicCNN | 0.7603 ± 0.0061 | 0.6808 ± 0.0038 | 0.6732 ± 0.0041 |
| ResNet-34 | 1.0468 ± 0.0134 | 0.7087 ± 0.0165 | 0.6268 ± 0.0105 |
| ResNet-20 | 0.6728 ± 0.0301 | 0.6766 ± 0.0155 | 0.4824 ± 0.0225 |

As noted above, section 5.5.5, the extra wall time is dominated by the largest parameter, this is often the final dense layer; table 5.6 shows that, for CIFAR-10, only orthogonalising the convolutional layers does not reduce performance. While both the error rates actually decrease for Conv-Orthogonal-SGDM we hesitate to say that orthogonalising dense layers is detrimental since the results are within the margin of experimental error; additionally these networks only have a dense final classification layer which is qualitatively different from intermediary dense layers, and only making up a small proportion of the performance of the network.

### 5.5.7 Limitations with small mini-batch sizes

Orthogonal-SGDM does not perform as well as SGDM when the mini-batch size is minuscule, figure 5.14, due to the increased levels of noise in the gradients which is compounded during SVD. Large mini-batches reduce this noise by making the gradient vector more consistent between mini-batches, and thus the resulting orthogonalisation is also more similar.

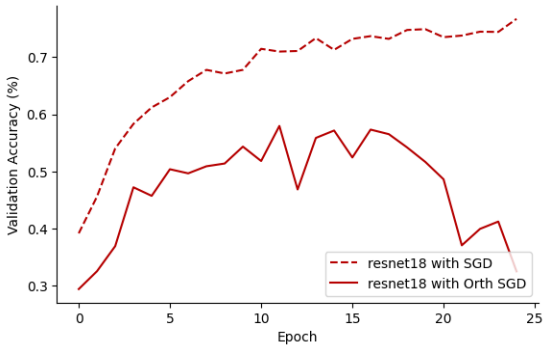



FIGURE 5.14: CIFAR-10 with mini-batch size=4 trained as in section 5.4.1.

Orthogonal-SGDM starts to outperform SGDM with a mini-batch size of 32 on a ResNet-20 for CIFAR-10. Few models need such small mini-batch sizes, but if they do then SGDM might be a more suitable optimisation algorithm; however, if small mini-batches are needed for memory or GPU bounds, then this can be easily offset using batch accumulation, where multiple forward passes are performed sequentially and summed before the backwards pass, effectively increasing the mini-batch size seen by the optimiser. In addition to the learning collapse, the time taken by SVD is only dependent on the parameter size and not the mini-batch size, as the gradients are accumulated before the SVD, so increasing the number of mini-batches per epoch also increases the wall time to train.

## 5.6 Discussion

Adam has found its place as a reliable optimiser that works over a wide variety of hyper-parameter sets, yielding consistent performance with little fine-tuning needed, such that most of the performance is gained under most hyper-parameter sets.

In this work we have laid out a new adaption to optimisation methods, tested it on different models and data sets, showing state-of-the-art results and out of the box robustness to hyper-parameter choice and over-parametrised models. Not only is Orthogonal-SGDM better for image classification it also has practical application in problems such as object detection and semantic segmentation since they make use of a pre-trained image classification backbone.

SGDM with hyper-parameter tuning is still more studied, but Orthogonal-SGDM can be an excellent method for quick verification of models or for prototyping — when we want quick validation that a method or technique is tractable before seeking to optimise the best possible model. However, as more data set sizes are growing more models are being trained on fewer to less than one epoch of data [27] leading to an severely limited ability to tune the hyper-parameters and quicker feedback on training progress is needed.

Lastly, we mentioned briefly in section 5.1 how attention heads fit our component model but, since they are beyond the scope of this work, we will explore the potential gain in using Orthogonal-SGDM with them in future work, and expect a similarly large gain will be obtained. Using Orthogonal-SGDM to train language models is a promising avenue for future work, as it is a very computationally expensive task.

# VI

# Concluding remarks

## 6.1 Our results

In this thesis we have produced results in these three areas of deep network optimisation.

- First order metrics to understand the behaviour of the optimisation
- Extracting curvature information from the already-computed gradients
- Creating new optimisation methods using deep network specific features

In the first we looked into the trends during optimisation and have created metrics for the directedness of optimisation via stochastic process analysis and cosine similarity; in addition, we created a roughness metric via measuring the gradient similarity. In the second we have looked at implementing a quasi-Newton's method assuming the average gradient for each class is a one of the top-k eigenvectors of the Hessian. We also show that it is possible to estimate the top-k eigenvalues using the power iteration method on the gradients. In the third, we have looked at self-orthogonalising the gradients and the resulting positive effect of the optimisation rate and model performance.

## 6.2 Discussion

In this work we have found that the optimisation of deep networks is not as simple as it is often portrayed in the practical literature, and that there are many properties that can be calculated to indicate what the optimisation is doing.

Firstly, models are very directed in learning but do not converge. Many of our results on first order statistics are counter-intuitive and so are often ignored or simplified out of the problem. What we have developed to date implies what is going on, but the

results do not always cohere, for example, the random distribution of the gradients, figure 4.15, with the monotonic loss when doing a linear interpolation of the start and solution [158].

Next, second order information can be extracted from the gradients, but it is unclear how to use this information to improve the optimisation as there is a seemingly insurmountable amount of noise in both the top-eigenvector and the quasi-Newton methods.

Lastly, exploiting the inherent structure of the models can lead to improvements in the optimisation, and that normalising the gradients, *i.e.* increasing their magnitude, leads to a more efficient optimisation as long as it does not cause the loss to blow up.

Our most significant result is the adaption to any current optimisation algorithm that speeds it up. This is a significant result as it is a simple, low computational cost, change to the optimisation algorithm that can be applied to any problem and is constantly better.

As we said in the preface, understanding these areas and results can and have lead to immense improvements in optimising deep networks across many disciplines and domains.

## 6.3 Future work

There are many promising areas for future work in optimisation and deep learning; some of which we have outlined below.

### 6.3.1 Extensions of this work

Herein we specify some future work that builds immediately from this work.

- Apply our analysis techniques to more variations of networks, optimisers, and problems (NLP, generative) *inc.* ablations of various features like dropout, batch norm, skip connections.
- Our quasi-Newton's method, section 4.3, has a simple method to calculate the Hessian approximation, given that this method does not work well, we would be interested to know if a BFGS analogue, *i.e.* limiting the step size with the Wolfe conditions, will improve performance; and if so, is the additional line search computationally prohibitive?
- Estimating the eigenspectrum via calculating the cumulants from the powers of the gradients, see appendix H for a beginning.

- The Lanczos method can be used to calculate the top-k eigenvectors, not just the top single. Can our method, section 4.4, that is also based of power iteration be modified to produce more than one top eigenvector.
- The main problem faced in using second order information is the extreme amount of noise in the estimations. Reducing this noise is the main problem to solve in order to use second order information; this could be done by trying to average over more data, or by combining different ways of estimation the information together.
- Test our method that self-orthogonalises the gradients on the transformer [159] architecture and other models that use self-attention.
- Since the loss is dominated by miss-classifications, section 4.4.3.2, does sampling with a bias to previously misclassified data improve learning speed? Is there an optimal ratio of misclassified vs correctly classified data for a batch?

### 6.3.2 Other areas of research in the optimisation of deep networks

Several adjacent areas of research in the optimisation of deep networks have been identified during the course of this work but were not explored, and are outlined below.

- How does distributed training affect the optimisation? — a lot of distributed training is done such that the optimisation is equivalent to the non-distributed version, however, Dean *et al.* [160] use out of order updates to train; which is also similar to federated learning. What is the effect of this on the optimisation and its trajectory.
- Symmetries in the loss landscape — how do symmetries in the loss landscape affect the optimisation? How can we use these symmetries to improve the optimisation? See appendix I for a beginning.
- Non-convergence — do models ever stop learning, can we separate the decrease in loss from the noise.
- Finding ensembles in a single training run — can we find ensembles in a single training run by creating a high-noise optimisation and taking multiple checkpoints at low-loss positions. Is this categorically different from optimising from different initialisations; *i.e.* there is a significant overlap in the way the models "work" for every point in an optimisation run. Is there an optimisation algorithm that, although it does not produce the best model, produces several models simultaneously that can be ensembled to a better model in the same or fewer iterates?

# Glossary

**AdaGrad** An optimiser that uses ADAptive GRADient estimation. ix, 20, 21, 105

**Adam** An optimiser that uses ADAptive Moment estimation. xv, xvi, 21, 22, 35–37, 39–42, 49, 52, 100–103, 105, 108, 116

**AdamW** An optimiser that uses ADAptive Moment estimation with Weight decay. xv, 100, 102, 105

**AR** Auto-regressive. vi, x, 43–50, 129

**BFGS** Broyden-Fletcher-Goldfarb-Shanno algorithm. 55–57, 118

**CG** Conjugate Gradient. 59

**CKA** Centred Kernel Alignment. ix, 32, 33

**CN-SGDM** Component Normalised SGDM. xvi, 112

**DFP** Davidon-Fletcher-Powell algorithm. 56

**DSO** Double Soft Orthogonality Regularisation. 93, 94

**EKI** Ensemble Kalman Inversion. 28

**Gauss-Newton** An extension of Newton's method for non-linear least squares problems. 54

**GD** Gradient Descent. 18, 20

**L-BFGS** Limited-memory Broyden-Fletcher-Goldfarb-Shanno algorithm. 55, 57, 58

**LARS** Layer-wise Adaptive Rate Scaling. xii, 106, 107

**Levenberg-Marquardt** A minimisation algorithm that combines gradient descent and the Gauss-Newton method. 54

**MC** Mutual Coherence. 93, 94

**MLP** Multi-layered Perceptron. 53, 97

**N-SGDM** Normalised-SGDM. xvi, 111, 112

**NTK** Neural Tangent Kernel. 22

**Orthogonal-Adam** An orthogonalised version of ADAptive Moment estimation. xvi, 100, 101, 103, 105, 108

**Orthogonal-AdamW** An orthogonalised version of ADAptive Moment estimation with Weight decay. xv, 100

**Orthogonal-SGDM** Orthogonal Stochastic Gradient Descent with Momentum. xii, xvi, 99–106, 109, 110, 112–116

**P-VAE** Poisson Variational Autoencoder. 107, 108

**PCA** Principal Component Analysis. ix, 30, 31

**PCGrad** Projecting Conflicting Gradients. 90, 91

**PDE** Partial Differential Equation. 33

**ReLU** Rectified Linear Unit. 3, 5, 15, 16, 112, 163, 164

**RMSProp** An optimiser that uses Root Mean Square Propagation. 105

**SGD** Stochastic Gradient Descent. xvi, 19, 22, 34, 53, 63, 80, 87, 103, 105

**SGDM** Stochastic Gradient Descent with Momentum. x–xiii, xv, xvi, 19, 22, 34–37, 39–42, 44, 49, 50, 52, 69, 72, 81, 89, 98, 100–106, 109–116, 143, 153

**SLQ** Stochastic Lanczos Quadrature. 74

**SR1** Symmetric Rank-1 algorithm. 55, 57

**SVD** Singular Value Decomposition. 98, 111, 113–115

**TDP** Temporarily Disabled Parameters. xiii, 113

# References


[1] M. Tuddenham, A. Prügel-Bennett, and J. Hare, "Quasi-newton's method in the class gradient defined high-curvature subspace," *arXiv preprint arXiv:2012.01938*, 2020.

[2] ——, "Orthogonalising gradients to speed up neural network optimisation," 2022. DOI: 10.48550/arxiv.2202.07052. [Online]. Available: https://arxiv.org/abs/2202.07052.

[3] H. Hassan, A. Aue, C. Chen, V. Chowdhary, J. Clark, C. Federmann, X. Huang, M. Junczys-Dowmunt, W. Lewis, M. Li, *et al.*, "Achieving human parity on automatic chinese to english news translation," *arXiv preprint arXiv:1803.05567*, 2018.

[4] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, *et al.*, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *arXiv preprint arXiv:1712.01815*, 2017.

[5] C. Li, L. Shen, and G. Qian, "Online hybrid neural network for stock price prediction: A case study of high-frequency stock trading in the chinese market," *Econometrics*, vol. 11, no. 2, p. 13, 2023.

[6] M. Tian, Z. Shen, X. Wu, K. Wei, and Y. Liu, "The application of artificial intelligence in medical diagnostics: A new frontier," *Academic Journal of Science and Technology*, vol. 8, no. 2, pp. 57–61, 2023.

[7] C. Coleman, D. Narayanan, D. Kang, T. Zhao, J. Zhang, L. Nardi, P. Bailis, K. Olukotun, C. Ré, and M. Zaharia, "Dawnbench: An end-to-end deep learning benchmark and competition," *Training*, vol. 100, no. 101, p. 102, 2017. [Online]. Available: https://dawn.cs.stanford.edu/benchmark/.

[8] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*, Ieee, 2009, pp. 248–255.

[9] G. Huang, Z. Liu, G. Pleiss, L. Van Der Maaten, and K. Weinberger, "Convolutional networks with dense connectivity," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2019.

[10] A. Cauchy *et al.*, "Méthode générale pour la résolution des systemes d' équations simultanées," *Comp. Rend. Sci. Paris*, vol. 25, no. 1847, pp. 536–538, 1847.

[11] K. Fukushima, "Cognitron: A self-organizing multilayered neural network," *Biological cybernetics*, vol. 20, no. 3-4, pp. 121–136, 1975.

[12] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 807–814.

[13] J. Sohl-Dickstein, "The boundary of neural network trainability is fractal," *arXiv preprint arXiv:2402.06184*, 2024.

[14] R. E. Bellman, "Dynamic programming," 1957.

[15] R. Bellman, *Adaptive Control Processes: A Guided Tour*, ser. Princeton Legacy Library. Princeton University Press, 1961, ISBN: 9780691079011. [Online]. Available: https://books.google.co.uk/books?id=POAmAAAAMAAJ.

[16] J. Hopcroft and R. Kannan, *Foundations of data science*. 2014.

[17] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," *arXiv preprint arXiv:1711.05101*, 2017.

[18] G. Zhang, C. Wang, B. Xu, and R. Grosse, "Three mechanisms of weight decay regularization," *arXiv preprint arXiv:1810.12281*, 2018.

[19] B. T. Polyak, "Some methods of speeding up the convergence of iteration methods," *Ussr computational mathematics and mathematical physics*, vol. 4, no. 5, pp. 1–17, 1964.

[20] Y. E. Nesterov, "A method of solving a convex programming problem with convergence rate $o(1/k^2)$," in *Dokl. akad. nauk Sssr*, vol. 269, 1983, pp. 543–547.

[21] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[22] F. P. Such, V. Madhavan, E. Conti, J. Lehman, K. O. Stanley, and J. Clune, "Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning," *arXiv preprint arXiv:1712.06567*, 2017.

[23] J. W. Tukey, *Exploratory data analysis*. Addison–Wesley, Reading, MA, 1977.

[24] E. R. Tufte, *The visual display of quantitative information*. Graphics press LLC, Cheshire, Connecticut, 2001.

[25] H. A. Babri and Y. Tong, "Deep feedforward networks: Application to pattern recognition," in *Proceedings of International Conference on Neural Networks ( ICNN'96)*, Ieee, vol. 3, 1996, pp. 1422–1426.

[26] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[27] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, "Language models are few-shot learners," *arXiv preprint arXiv:2005.14165*, 2020.

[28] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," *OpenAI Blog*, vol. 1, no. 8, p. 9, 2019.

[29] A. Krizhevsky, G. Hinton, *et al.*, "Learning multiple layers of features from tiny images," 2009.

[30] K. He, X. Zhang, S. Ren, and J. Sun, *Deep residual learning for image recognition. corr abs/1512.03385 ( 2015)*, 2015.

[31] J. Nocedal and S. Wright, *Numerical optimization*. Springer Science & Business Media, 2006.

[32] P. Jain and P. Kar, "Non-convex optimization for machine learning," *arXiv preprint arXiv:1712.07897*, 2017.

[33] D. J. C. MacKay, *Information theory, inference, and learning algorithms*. Cambridge University Press, 2003, p. 628, ISBN: 0521642981.

[34] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[35] W. Ma and J. Lu, "An equivalence of fully connected layer and convolutional layer," *arXiv preprint arXiv:1712.01252*, 2017.

[36] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[37] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International conference on machine learning*, Pmlr, 2015, pp. 448–456.

[38] Zhou and Chellappa, "Computation of optical flow using a neural network," in *IEEE 1988 international conference on neural networks*, IEEE, 1988, pp. 71–78.

[39] C. M. Bishop, *Pattern recognition and machine learning*. springer, 2006.

[40] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *International conference on machine learning*, Pmlr, 2013, pp. 1139–1147.

[41] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization.," *Journal of machine learning research*, vol. 12, no. 7, 2011.

[42] A. Bordes, L. Bottou, and P. Gallinari, "Sgd-qn: Careful quasi-newton stochastic gradient descent," *Journal of Machine Learning Research*, vol. 10, pp. 1737–1754, 2009.

[43] S. J. Reddi, S. Kale, and S. Kumar, "On the convergence of adam and beyond," 2019.

[44] F. Zou, L. Shen, Z. Jie, W. Zhang, and W. Liu, "A sufficient condition for convergences of adam and rmsprop," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 11 127–11 135.

[45] A. C. Wilson, R. Roelofs, M. Stern, N. Srebro, and B. Recht, "The marginal value of adaptive gradient methods in machine learning," *Advances in neural information processing systems*, vol. 30, 2017.

[46] J. Chen, D. Zhou, Y. Tang, Z. Yang, Y. Cao, and Q. Gu, "Closing the generalization gap of adaptive gradient methods in training deep neural networks," *arXiv preprint arXiv:1806.06763*, 2018.

[47] D. Zhou, J. Chen, Y. Cao, Z. Yang, and Q. Gu, "On the convergence of adaptive gradient methods for nonconvex optimization," *arXiv preprint arXiv:1808.05671*, 2018.

[48] X. Chen, S. Liu, R. Sun, and M. Hong, "On the convergence of a class of adam-type algorithms for non-convex optimization," *arXiv preprint arXiv:1808.02941*, 2018.

[49] M. Zaheer, S. Reddi, D. Sachan, S. Kale, and S. Kumar, "Adaptive methods for nonconvex optimization," *Advances in neural information processing systems*, vol. 31, 2018.

[50] A. Défossez, L. Bottou, F. Bach, and N. Usunier, "A simple convergence proof of adam and adagrad," *arXiv preprint arXiv:2003.02395*, 2020.

[51] R. M. Schmidt, F. Schneider, and P. Hennig, "Descending through a crowded valley–benchmarking deep learning optimizers," *arXiv preprint arXiv:2007.01547*, 2020.

[52] J. Chen, C. Wolfe, Z. Li, and A. Kyrillidis, "Demon: Momentum decay for improved neural network training," *arXiv preprint arXiv:1910.04952*, 2019.

[53] K. Kawaguchi and Q. Sun, "A recipe for global convergence guarantee in deep neural networks," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 35, 2021, pp. 8074–8082.

[54] A. Beznosikov, E. Gorbunov, H. Berard, and N. Loizou, "Stochastic gradient descent-ascent: Unified theory and new efficient methods," in *International conference on artificial intelligence and statistics*, PMLR, 2023, pp. 172–235.

[55] Y.-G. Hsieh, F. Iutzeler, J. Malick, and P. Mertikopoulos, "On the convergence of single-call stochastic extra-gradient methods," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[56] R. Ward, X. Wu, and L. Bottou, "Adagrad stepsizes: Sharp convergence over nonconvex landscapes, from any initialization," *arXiv preprint arXiv:1806.01811*, vol. 2, 2018.

[57] Y. Tao, H. Yuan, X. Zhou, Y. Cao, and Q. Gu, "Towards simple and provable parameter-free adaptive gradient methods," *arXiv preprint arXiv:2412.19444*, 2024.

[58] H. Tran, Q. Zhang, and A. Cutkosky, "Reevaluating theoretical analysis methods for optimization in deep learning,"

[59] T. Overman, G. Blum, and D. Klabjan, "A primal-dual algorithm for hybrid federated learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, 2024, pp. 14 482–14 489.

[60] S. Du, J. Lee, H. Li, L. Wang, and X. Zhai, "Gradient descent finds global minima of deep neural networks," in *International conference on machine learning*, PMLR, 2019, pp. 1675–1685.

[61] S. S. Du, X. Zhai, B. Poczos, and A. Singh, "Gradient descent provably optimizes over-parameterized neural networks," *arXiv preprint arXiv:1810.02054*, 2018.

[62] Q. Nguyen and M. Hein, "The loss surface of deep and wide neural networks," in *International conference on machine learning*, PMLR, 2017, pp. 2603–2612.

[63] K. He, X. Zhang, S. Ren, and J. Sun, "Identity mappings in deep residual networks," in *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part IV 14*, Springer, 2016, pp. 630–645.

[64] P.-Y. Chen, H. Zhang, Y. Sharma, J. Yi, and C.-J. Hsieh, "Zoo: Zeroth order optimization based black-box attacks to deep neural networks without training substitute models," in *Proceedings of the 10th ACM workshop on artificial intelligence and security*, 2017, pp. 15–26.

[65] N. B. Kovachki and A. M. Stuart, "Ensemble kalman inversion: A derivative-free technique for machine learning tasks," *Inverse Problems*, vol. 35, no. 9, p. 095 005, 2019.

[66] L. Deng, "The mnist database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[67] Z. Ding and Q. Li, "Ensemble kalman inversion: Mean-field limit and convergence analysis," *Statistics and computing*, vol. 31, pp. 1–21, 2021.

[68] K. Simonyan, A. Vedaldi, and A. Zisserman, "Deep inside convolutional networks: Visualising image classification models and saliency maps," *arXiv preprint arXiv:1312.6034*, 2013.

[69] A. Nguyen, A. Dosovitskiy, J. Yosinski, T. Brox, and J. Clune, "Synthesizing the preferred inputs for neurons in neural networks via deep generator networks," *Advances in neural information processing systems*, vol. 29, 2016.

[70] H. Li, Z. Xu, G. Taylor, C. Studer, and T. Goldstein, "Visualizing the loss landscape of neural nets," in *Advances in Neural Information Processing Systems*, 2018, pp. 6389–6399.

[71] R. Shwartz-Ziv and N. Tishby, "Opening the black box of deep neural networks via information," *arXiv preprint arXiv:1703.00810*, 2017.

[72] T. Nguyen, M. Raghu, and S. Kornblith, "Do wide and deep networks learn the same things? uncovering how neural network representations vary with width and depth," *arXiv preprint arXiv:2010.15327*, 2020.

[73] K. Wu, X. Jian, R. Du, J. Chen, and X. Zhou, "Roughness index for loss landscapes of neural network models of partial differential equations," in *2023 IEEE International Conference on Big Data (BigData)*, IEEE, 2023, pp. 966–975.

[74] B. Yu *et al.*, "The deep ritz method: A deep learning-based numerical algorithm for solving variational problems," *Communications in Mathematics and Statistics*, vol. 6, no. 1, pp. 1–12, 2018.

[75] J. Sirignano and K. Spiliopoulos, "Dgm: A deep learning algorithm for solving partial differential equations," *Journal of computational physics*, vol. 375, pp. 1339–1364, 2018.

[76] A. J. Stam, "Some inequalities satisfied by the quantities of information of fisher and shannon," *Information and Control*, vol. 2, no. 2, pp. 101–112, 1959.

[77] R. Y. Rubinstein and D. P. Kroese, *Simulation and the Monte Carlo method*. John Wiley & Sons, 2016.

[78] J. A. Nelder and R. Mead, "A simplex method for function minimization," *The computer journal*, vol. 7, no. 4, pp. 308–313, 1965.

[79] S. Jastrzebski, M. Szymczak, S. Fort, D. Arpit, J. Tabor, K. Cho, and K. Geras, "The break-even point on optimization trajectories of deep neural networks," *arXiv preprint arXiv:2002.09572*, 2020.

[80] S. Fort, H. Hu, and B. Lakshminarayanan, "Deep ensembles: A loss landscape perspective," *arXiv preprint arXiv:1912.02757*, 2019.

[81] B. Ghorbani, S. Krishnan, and Y. Xiao, "An investigation into neural net optimization via hessian eigenvalue density," in *International Conference on Machine Learning*, PMLR, 2019, pp. 2232–2241.

[82] G. Gur-Ari, D. A. Roberts, and E. Dyer, "Gradient descent happens in a tiny subspace," *arXiv preprint arXiv:1812.04754*, 2018. [Online]. Available: http://arxiv.org/abs/1812.04754.

[83] L. Liu, X. Liu, J. Gao, W. Chen, and J. Han, "Understanding the difficulty of training transformers," *arXiv preprint arXiv:2004.08249*, 2020.

[84] Y. Li and H. Liu, "Implementation of stochastic quasi-newton's method in pytorch," *arXiv preprint arXiv:1805.02338*, 2018.

[85] Å. Björck, *Numerical methods for least squares problems*. SIAM, 2024.

[86] D. W. Marquardt, "An algorithm for least-squares estimation of nonlinear parameters," *Journal of the society for Industrial and Applied Mathematics*, vol. 11, no. 2, pp. 431–441, 1963.

[87] K. Levenberg, "A method for the solution of certain non-linear problems in least squares," *Quarterly of applied mathematics*, vol. 2, no. 2, pp. 164–168, 1944.

[88] P. Wolfe, "Convergence conditions for ascent methods," *SIAM review*, vol. 11, no. 2, pp. 226–235, 1969.

[89] ——, "Convergence conditions for ascent methods. ii: Some corrections," *SIAM review*, vol. 13, no. 2, pp. 185–188, 1971.

[90] C. G. Broyden, "The convergence of a class of double-rank minimization algorithms 2. the new algorithm," *IMA Journal of Applied Mathematics*, vol. 6, no. 3, pp. 222–231, 1970.

[91] R. Fletcher, "A new approach to variable metric algorithms," *The Computer Journal*, vol. 13, no. 3, pp. 317–322, 1970.

[92] D. Goldfarb, "A family of variable-metric methods derived by variational means," *Mathematics of Computation*, vol. 24, no. 109, pp. 23–26, 1970.

[93] D. F. Shanno, "Conditioning of quasi-newton methods for function minimization," *Mathematics of Computation*, vol. 24, no. 111, pp. 647–656, 1970.

[94] E. Jones, T. Oliphant, P. Peterson, *et al.*, "Scipy: Open source scientific tools for python," 2001.

[95] T. M. Inc., *Matlab optimization toolbox version: 9.4 (r2022b)*, Natick, Massachusetts, United States, 2022. [Online]. Available: https://www.mathworks.com.

[96] J. E. Dennis Jr and J. J. Moré, "Quasi-newton methods, motivation and theory," *SIAM review*, vol. 19, no. 1, pp. 46–89, 1977.

[97] M. J. Powell, "Some global convergence properties of a variable metric algorithm for minimization without exact line searches," *Nonlinear programming*, vol. 9, no. 1, pp. 53–72, 1976.

[98] D.-H. Li and M. Fukushima, "On the global convergence of the bfgs method for nonconvex unconstrained optimization problems," *SIAM Journal on Optimization*, vol. 11, no. 4, pp. 1054–1064, 2001.

[99] R. H. Byrd, J. Nocedal, and Y.-X. Yuan, "Global convergence of a cass of quasi-newton methods on convex problems," *SIAM Journal on Numerical Analysis*, vol. 24, no. 5, pp. 1171–1190, 1987.

[100] Ricotti, Ragazzini, and Martinelli, "Learning of word stress in a sub-optimal second order back-propagation neural network," in *IEEE 1988 International Conference on Neural Networks*, IEEE, 1988, pp. 355–361.

[101] Y. LeCun, J. Denker, and S. Solla, "Optimal brain damage," *Advances in neural information processing systems*, vol. 2, 1989.

[102] B. A. Pearlmutter, "Fast exact multiplication by the hessian," *Neural computation*, vol. 6, no. 1, pp. 147–160, 1994.

[103] Z. Yao, A. Gholami, K. Keutzer, and M. W. Mahoney, "Pyhessian: Neural networks through the lens of the hessian," in *2020 IEEE international conference on big data (Big data)*, IEEE, 2020, pp. 581–590.

[104] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[105] J. Martens *et al.*, "Deep learning via hessian-free optimization.," in *Icml*, vol. 27, 2010, pp. 735–742.

[106] C. Lanczos, "An iteration method for the solution of the eigenvalue problem of linear differential and integral operators," 1950.

[107] Y. Saad, *Numerical methods for large eigenvalue problems: revised edition*. SIAM, 2011.

[108] C. C. Paige, "The computation of eigenvalues and eigenvectors of very large sparse matrices.," PhD thesis, University of London, 1971.

[109] B. N. Parlett, *The symmetric eigenvalue problem*. SIAM, 1998.

[110] J. K. Cullum and R. A. Willoughby, *Lanczos algorithms for large symmetric eigenvalue computations: Vol. I: Theory*. SIAM, 2002.

[111] H. D. Simon, "The lanczos algorithm with partial reorthogonalization," *Mathematics of computation*, vol. 42, no. 165, pp. 115–142, 1984.

[112] A. Likas and A. Stafylopatis, "Training the random neural network using quasi-newton methods," *European Journal of Operational Research*, vol. 126, no. 2, pp. 331–339, 2000.

[113] D. J. MacKay, "A practical bayesian framework for backpropagation networks," *Neural computation*, vol. 4, no. 3, pp. 448–472, 1992.

[114] Z. Yao, A. Gholami, Q. Lei, K. Keutzer, and M. W. Mahoney, "Hessian-based analysis of large batch training and robustness to adversaries," *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[115] Y. N. Dauphin, R. Pascanu, C. Gulcehre, K. Cho, S. Ganguli, and Y. Bengio, "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization," in *Advances in neural information processing systems*, 2014, pp. 2933–2941.

[116] A. J. Bray and D. S. Dean, "Statistics of critical points of gaussian fields on large-dimensional spaces," *Physical review letters*, vol. 98, no. 15, p. 150 201, 2007.

[117] E. P. Wigner, "On the distribution of the roots of certain symmetric matrices," *Annals of Mathematics*, vol. 67, no. 2, pp. 325–327, 1958.

[118] Z. Yao, A. Gholami, S. Shen, K. Keutzer, and M. W. Mahoney, "Adahessian: An adaptive second order optimizer for machine learning," *arXiv preprint arXiv:2006.00719*, 2020.

[119] V. Papyan, "Measurements of three-level hierarchical structure in the outliers in the spectrum of deepnet hessians," *36th International Conference on Machine Learning, ICML 2019*, vol. 2019-June, pp. 8819–8830, Jan. 2019. [Online]. Available: http://arxiv.org/abs/1901.08244.

[120] B. Ghorbani, S. Krishnan, and Y. Xiao, "An investigation into neural net optimization via hessian eigenvalue density," *36th International Conference on Machine Learning, ICML 2019*, vol. 2019-June, pp. 4039–4052, Jan. 2019. [Online]. Available: http://arxiv.org/abs/1901.10159.

[121] S. Fort and S. Ganguli, "Emergent properties of the local geometry of neural loss landscapes," Oct. 2019. [Online]. Available: http://arxiv.org/abs/1910.05929.

[122] L. Sagun, U. Evci, V. U. Guney, Y. Dauphin, and L. Bottou, "Empirical analysis of the hessian of over-parametrized neural networks," *arXiv preprint arXiv:1706.04454*, 2017.

[123] F. Benaych-Georges and R. R. Nadakuditi, "The eigenvalues and eigenvectors of finite, low rank perturbations of large random matrices," *Advances in Mathematics*, vol. 227, no. 1, pp. 494–521, 2011.

[124] A. Jacot, F. Gabriel, and C. Hongler, "Neural tangent kernel: Convergence and generalization in neural networks," *Advances in neural information processing systems*, vol. 31, 2018.

[125] J. Lee, L. Xiao, S. Schoenholz, Y. Bahri, R. Novak, J. Sohl-Dickstein, and J. Pennington, "Wide neural networks of any depth evolve as linear models under gradient descent," *Advances in neural information processing systems*, vol. 32, 2019.

[126] F. Benaych-Georges and R. R. Nadakuditi, "The eigenvalues and eigenvectors of finite, low rank perturbations of large random matrices," *Advances in Mathematics*, vol. 227, pp. 494–521, 1 May 2011, ISSN: 10902082. DOI: 10.1016/j.aim.2011.02.007.

[127] V. Papyan, "Measurements of three-level hierarchical structure in the outliers in the spectrum of deepnet hessians," *arXiv preprint arXiv:1901.08244*, 2019.

[128] D. Granziol, "Curvature is key: Sub-sampled loss surfaces and the implications for large batch training," Jun. 2020. [Online]. Available: http://arxiv.org/abs/2006.09092.

[129] Z. Yao, A. Gholami, K. Keutzer, and M. Mahoney, "Pyhessian: Neural networks through the lens of the hessian," *arXiv preprint arXiv:1912.07145*, 2019.

[130] M. F. Hutchinson, "A stochastic estimator of the trace of the influence matrix for laplacian smoothing splines," *Communications in Statistics-Simulation and Computation*, vol. 18, no. 3, pp. 1059–1076, 1989.

[131] G. H. Golub and G. Meurant, *Matrices, moments and quadrature with applications*. Princeton University Press, 2009, vol. 30.

[132] H. Avron and S. Toledo, "Randomized algorithms for estimating the trace of an implicit symmetric positive semi-definite matrix," *Journal of the ACM (JACM)*, vol. 58, no. 2, pp. 1–34, 2011.

[133] D. H. Wolpert and W. G. Macready, "No free lunch theorems for optimization," *IEEE transactions on evolutionary computation*, vol. 1, no. 1, pp. 67–82, 1997.

[134] T. Yu, S. Kumar, A. Gupta, S. Levine, K. Hausman, and C. Finn, "Gradient surgery for multi-task learning," *arXiv preprint arXiv:2001.06782*, 2020.

[135] N. Bansal, X. Chen, and Z. Wang, "Can we gain more from orthogonality regularizations in training deep cnns?" *arXiv preprint arXiv:1810.09102*, 2018.

[136] D. Xie, J. Xiong, and S. Pu, "All you need is beyond a good init: Exploring better solution for training extremely deep convolutional neural networks with orthonormality and modulation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 6176–6185.

[137] L. Huang, X. Liu, B. Lang, and A. W. Yu, "Orthogonal weight normalization: Solution to optimization over multiple dependent stiefel manifolds in deep neural networks," *arXiv preprint arXiv:1709.06079*, 2017.

[138] R. Balestriero *et al.*, "A spline theory of deep learning," in *international conference on machine learning*, PMLR, 2018, pp. 374–383.

[139] R. Balestriero and R. G. Baraniuk, "Mad max: Affine spline insights into deep learning," *Proceedings of the IEEE*, vol. 109, no. 5, pp. 704–727, 2020.

[140] D. L. Donoho, "Compressed sensing," *IEEE Transactions on information theory*, vol. 52, no. 4, pp. 1289–1306, 2006.

[141] S. Hyland and G. Rätsch, "Learning unitary operators with help from u (n)," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 31, 2017.

[142] K. Helfrich, D. Willmott, and Q. Ye, "Orthogonal recurrent neural networks with scaled Cayley transform," in *Proceedings of the 35th International Conference on Machine Learning*, J. Dy and A. Krause, Eds., ser. Proceedings of Machine Learning Research, vol. 80, PMLR, Oct. 2018, pp. 1969–1978. [Online]. Available: https://proceedings.mlr.press/v80/helfrich18a.html.

[143] M. Henaff, A. Szlam, and Y. LeCun, "Recurrent orthogonal networks and long-memory tasks," in *International Conference on Machine Learning*, PMLR, 2016, pp. 2034–2042.

[144] K. Jia, S. Li, Y. Wen, T. Liu, and D. Tao, "Orthogonal deep neural networks," *IEEE transactions on pattern analysis and machine intelligence*, 2019.

[145] B. Neyshabur, S. Bhojanapalli, D. McAllester, and N. Srebro, "Exploring generalization in deep learning," *Advances in neural information processing systems*, vol. 30, 2017.

[146] K. Than and N. Vu, "Generalization of gans and overparameterized models under lipschitz continuity," *arXiv preprint arXiv:2104.02388*, 2021.

[147] M. Biehl, P. Riegler, and C. Wöhler, "Transient dynamics of on-line learning in two-layered neural networks," *Journal of Physics A: Mathematical and General*, vol. 29, no. 16, p. 4769, 1996.

[148] L. N. Trefethen and D. Bau III, *Numerical linear algebra*. Siam, 1997, vol. 50.

[149] [Online]. Available: https://paperswithcode.com/sota/image-classification-on-cifar-10.

[150] M. Abadi, "Tensorflow: Learning functions at scale," in *Proceedings of the 21st ACM SIGPLAN international conference on functional programming*, 2016, pp. 1–1.

[151] J. Zbontar, L. Jing, I. Misra, Y. LeCun, and S. phane Deny, *Barlow twins: Self-supervised learning via redundancy reduction*, 2021.

[152] Y. You, I. Gitman, and B. Ginsburg, *Large batch training of convolutional networks*, 2017.

[153] H. Vafaii, D. Galor, and J. Yates, "Poisson variational autoencoder," *Advances in Neural Information Processing Systems*, vol. 37, pp. 44 871–44 906, 2024.

[154] C. Olah, A. Satyanarayan, I. Johnson, S. Carter, L. Schubert, K. Ye, and A. Mordvintsev, "The building blocks of interpretability," *Distill*, vol. 3, no. 3, e10, 2018.

[155] Y. Nesterov, *Introductory lectures on convex optimization: A basic course*. Springer Science & Business Media, 2003, vol. 87.

[156] J. W. Demmel, *Applied numerical linear algebra*. Siam, 1997.

[157] J. Wang, Y. Chen, R. Chakraborty, and S. X. Yu, "Orthogonal convolutional neural networks," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2020, pp. 11 505–11 515.

[158] J. Lucas, J. Bae, M. R. Zhang, S. Fort, R. Zemel, and R. Grosse, *Analyzing monotonic linear interpolation in neural network loss landscapes*, 2021.

[159] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[160] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, *et al.*, "Large scale distributed deep networks," *Advances in neural information processing systems*, vol. 25, pp. 1223–1231, 2012.

[161] M. Abramowitz and I. Stegun, "Handbook of mathematical functions (ninth corrected printing)," *National Bureau of Standards*, 1970.

[162] J. Stirling, *Methodus differentialis: sive tractatus de summatione et interpolatione serierum infinitarum*. 1730.

[163] A. Papoulis and S. U. Pillai, *Probability, random variables, and stochastic processes*. McGraw-Hill Europe: New York, NY, USA, 1991.

[164] P. Whittle, *Probability*. Penguin, Library of University Mathematics, 1970.

[165] P. Diaconis and D. Freedman, "Asymptotics of graphical projection pursuit," *The annals of statistics*, pp. 793–815, 1984.

[166] T. Cai, J. Fan, and T. Jiang, "Distributions of angles in random packing on spheres," *The Journal of Machine Learning Research*, vol. 14, no. 1, pp. 1837–1864, 2013.

[167] V. Badrinarayanan, B. Mishra, and R. Cipolla, "Symmetry-invariant optimization in deep networks," *arXiv preprint arXiv:1511.01754*, 2015.

[168] A. E. Orhan and X. Pitkow, "Skip connections eliminate singularities," *arXiv preprint arXiv:1701.09175*, 2017.

# I

# Mathematics for section 3.2

## A.1 Some preliminary values on random vectors

Because of the aforementioned problems in analysing high-dimensional models, we have sought to create some tools that will allow us to gain information about the loss landscape of high-dimensional models in real-world scenarios. We start by laying out some mathematical preliminaries which will be used in sections 3.2.4 and 3.2.6 to compare the empirical performance against what would be expected for a random walk or an auto-regressive process.

For mathematical convenience later on, let us define a function

$$G(n) = \frac{\Gamma(\frac{n+1}{2})}{\Gamma(\frac{n}{2})} \tag{A.1}$$

It would be good to study a few properties of this function, since we will be using it a lot, and often approximating it.

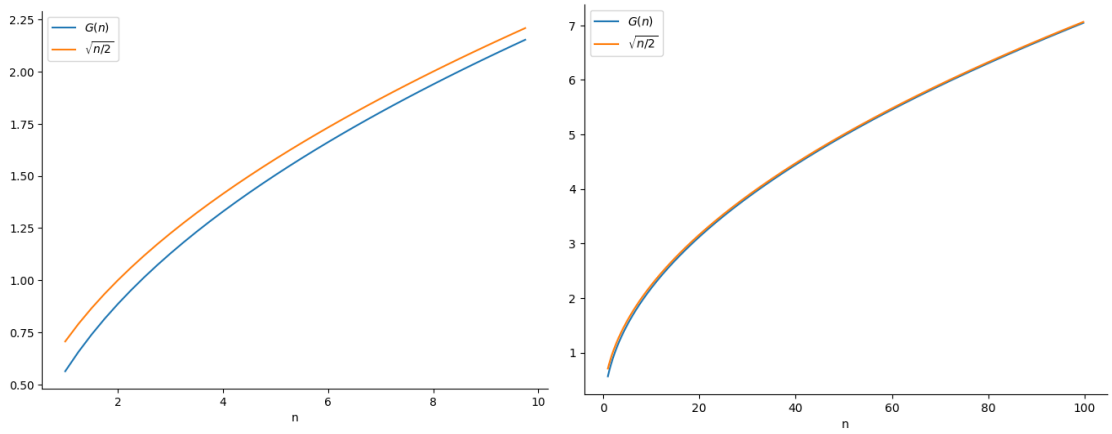
**Corollary A.1.** *$G(n)$ is monotonically increasing for $n > 0$.*

*Proof of corollary A.1.* The gamma function can be defined in terms of a recurrence relation [161, p256 §6.1.15]

$$\Gamma(n) = (n-1)\Gamma(n-1)$$

Then,

$$G(n) = \frac{\Gamma(\frac{n+1}{2})}{\Gamma(\frac{n}{2})} = \frac{\frac{n-1}{2}\Gamma(\frac{n-1}{2})}{(\frac{n}{2}-1)\Gamma(\frac{n}{2}-1)} = \frac{n-1}{n-2}G(n-1).$$




(A) Plot of $G(n)$ vs $\sqrt{n/2}$ up to 10.

(B) Plot of $G(n)$ vs $\sqrt{n/2}$ up to 100.

FIGURE A.1: Comparison of $G(n)$ and $\sqrt{n/2}$ for small and large $n$.

Since $\frac{n-1}{n-2} > 0$ for $n > 2$, $G(n)$ is monotonically increasing *iff* $G(1) > 0$ and $G(2) > G(1)$,

$$G(1) = \frac{\Gamma(\frac{2}{2})}{\Gamma(\frac{1}{2})} = \frac{1}{\sqrt{\pi}} \approx 0.564,$$
$$G(2) = \frac{\Gamma(\frac{3}{2})}{\Gamma(1)} = \Gamma\left(\frac{3}{2}\right) \approx 0.886.$$

□

We can also define an approximation of $G(n)$ for large $n$ by applying Stirling's approximation [162] [161, p257 §6.1.37] for the gamma function,

$$\Gamma(z) \approx z^{z-\frac{1}{2}} e^{-z} (2\pi)^{\frac{1}{2}}. \tag{A.2}$$

**Corollary A.2.** *$G(n) \approx \sqrt{\frac{n}{2}}$, for large $n$.*

*Proof of corollary A.2.* Substituting Sterling's formula, equation (A.2), into equation (A.1) gives

$$\begin{aligned} G(n) = \frac{\Gamma(\frac{n+1}{2})}{\Gamma(\frac{n}{2})} &\approx \frac{\sqrt{\frac{n}{2}}}{\sqrt{\frac{n+1}{2}}} \left(\frac{n+1}{2e}\right)^{\frac{n+1}{2}} \left(\frac{n}{2e}\right)^{-\frac{n}{2}}, \\ &\approx \sqrt{\frac{n}{n+1}} \sqrt{\frac{n+1}{2e}} \left(\frac{n+1}{2e}\right)^{\frac{n}{2}} \left(\frac{n}{2e}\right)^{-\frac{n}{2}}, \\ &\approx \sqrt{\frac{n}{n+1}} \sqrt{\frac{n+1}{2e}} \left(\frac{n+1}{n}\right)^{\frac{n}{2}}, \\ &\approx \sqrt{\frac{n}{2e}} \left(1 + \frac{1}{n}\right)^{\frac{n}{2}}. \end{aligned}$$

When $n$ is large,

$$\lim_{n\to\infty}\left(1+\frac{1}{n}\right)^{\frac{n}{2}}=\sqrt{e},$$

Thus,

$$G(n)\approx\sqrt{\frac{n}{2}}. \tag{A.3}$$

□

Unfortunately, equation (A.3) is not a very good approximation for small $n$, as we can see from the left plot figure A.1 — code to generate these plots is given in appendix C, listing C.1. Even for large $n$, the approximation is not accurate enough to be substituted into the various expressions for $G(n)$ in this work; for example, we will see expressions for variance that, if this approximation is used, result in zero. More terms in Stirling's approximation could be used to improve the approximation, but this would make the expressions more complicated for limited gain in accuracy; and, computationally, existing implementations of the gamma function will be better optimised for numerical accuracy, especially when they are often computed in log space. However, it is useful for understanding the asymptotic behaviour of $G(n)$.

### A.1.1 Norms of random vectors

Deep learning models consist of many layers of which each could have a different number and shape of parameters. This leads to a variety of different architectures which complicate any analysis. To simplify the analysis, we will consider the parameters of a model as a single vector. Subsequently, to better interpret learning, we need to understand the behaviour of random high-dimensional vectors — specifically their norms — to differentiate it from the true learning behaviour.

**Lemma A.3.** *The expected 2-norm of an $n$-dimensional random vector, $\epsilon$, with elements sampled i.i.d. from $\mathcal{N}(0,\sigma^2)$ is $\sigma\sqrt{2}G(n)$.*

*Proof of lemma A.3.* Let $\epsilon_i\sim\mathcal{N}(0,\sigma^2)$, and $\zeta_i\sim\mathcal{N}(0,1)$ be a standard Gaussian such that $\epsilon_i=\sigma\zeta_i$, then

$$\mathbb{E}\left[\|\epsilon\|_2\right]=\mathbb{E}\left[\sqrt{\sum_{i=1}^{n}\epsilon_i^2}\right]=\mathbb{E}\left[\sqrt{\sum_{i=1}^{n}(\sigma\zeta_i)^2}\right]=\sigma\mathbb{E}\left[\sqrt{\sum_{i=1}^{n}\zeta_i^2}\right].$$

The squared norm of $n$ standard normally distributed variables is a chi-squared distribution, $\chi^2_n$ [163, p96], so

$$\mathbb{E}\left[\sqrt{\sum_{i=1}^{n}\zeta_i^2}\right] = \mathbb{E}\left[\sqrt{\chi^2_n}\right]. \tag{A.4}$$

From the law of the unconscious statistician [164, p31],

$$\begin{aligned}\mathbb{E}\left[\|\epsilon\|_2\right] &= \sigma\int_0^\infty \frac{\sqrt{x}}{2^{n/2}\Gamma(n/2)}x^{n/2-1}e^{-x/2}dx,\\ &= \sigma\sqrt{2}\left(\frac{\Gamma(\frac{n+1}{2})}{\Gamma(\frac{n}{2})}\right),\\ &= \sigma\sqrt{2}G(n).\end{aligned} \tag{A.5}$$

□

Code for empirical confirmation is given in appendix C, listing C.4.

**Lemma A.4.** *The expected square of the* 2*-norm of an n-dimensional random vector, $\epsilon$, with elements sampled i.i.d. from $\mathcal{N}(0, \sigma^2)$ is $\sigma^2 n$*

*Proof of lemma A.4.* Following the same lines as in lemma A.3,

$$\mathbb{E}\left[\|\epsilon\|_2^2\right] = \mathbb{E}\left[\sum_{i=1}^{n}\epsilon_i^2\right] = \sigma^2\mathbb{E}\left[\chi^2_n\right] = \sigma^2 n. \tag{A.6}$$

□

**Corollary A.5.** *The expected* 2*-norm of an n-dimensional random vector, $\epsilon$, with elements sampled i.i.d. from $\mathcal{N}(0, \sigma^2)$ is bounded by $\sigma\sqrt{n}$.*

*Proof of corollary A.5.* Applying Jensen's inequality, $\mathbb{E}\left[f(X)\right] \leq f(\mathbb{E}\left[X\right])$ for concave $f$, to equation (A.4) gives

$$\mathbb{E}\left[\|\epsilon\|_2\right] = \sigma\mathbb{E}\left[\sqrt{\chi^2_n}\right] \leq= \sigma\sqrt{\mathbb{E}\left[\chi^2_n\right]} = \sigma\sqrt{n} \tag{A.7}$$

□

We can use corollary A.5 and the true value, equation (A.5), to bound $G(n)$ as follows,

$$\begin{aligned}\mathbb{E}\left[\|\epsilon\|_2\right] &\leq \sigma\sqrt{n},\\ \sigma\sqrt{2}G(n) &\leq \sigma\sqrt{n},\\ G(n) &\leq \sqrt{\frac{n}{2}},\end{aligned}$$

showing that our approximation using Sterling's formula, equation (A.3), is also the upper bound for $G(n)$.

Next, since optimisation involves summing gradients, we need to also know the variance of these random vector norms.

**Lemma A.6.** *The variance of the* 2*-norm of an n-dimensional random vector, $\epsilon$, with elements sampled i.i.d. from $\mathcal{N}(0, \sigma^2)$ is $\sigma^2\left(n - 2G(n)^2\right)$.*

*Proof of lemma A.6.* Using the expectation of the norm, lemma A.3, and its square, lemma A.4, we have

$$\begin{aligned}\text{Var}\left[\|\epsilon\|_2\right] &= \mathbb{E}\left[\|\epsilon\|_2^2\right] - \mathbb{E}\left[\|\epsilon\|_2\right]^2, \\ &= \sigma^2 n - \left(\sigma\sqrt{2}G(n)\right)^2, \\ &= \sigma^2\left(n - 2G(n)^2\right). \end{aligned} \tag{A.8}$$

□

**Lemma A.7.** *The variance of the square of the* 2*-norm of an n-dimensional random vector, $\epsilon$, with elements sampled i.i.d. from $\mathcal{N}(0, \sigma^2)$ is $2\sigma^4 n$.*

*Proof of lemma A.7.* Let $Q \sim \chi^2_n$, then

$$\begin{aligned}\text{Var}\left[\|\epsilon\|_2^2\right] &= \mathbb{E}\left[\|\epsilon\|_2^4\right] - \mathbb{E}\left[\|\epsilon\|_2^2\right]^2, \\ &= \sigma^4\mathbb{E}\left[Q^2\right] - \sigma^4 n^2.\end{aligned}$$

From the law of the unconscious statistician [164, p31], and as the support of chi squared is $x \in [0, \infty)$.

$$\begin{aligned}\mathbb{E}\left[Q^2\right] &= \int_0^\infty x^2 f_Q(x)dx, \\ &= \int_0^\infty x^2 \frac{1}{2^{n/2}\Gamma(n/2)} x^{n/2-1}e^{-x/2}dx, \\ &= \frac{1}{2^{n/2}\Gamma(n/2)}\int_0^\infty x^{n/2+1}e^{-x/2}dx, \\ &= \frac{1}{2^{n/2}\Gamma(n/2)}\left[-2^{n/2+2}\Gamma\left(\frac{n}{2}+2, \frac{x}{2}\right)\right]_0^\infty,\end{aligned}$$

where $\Gamma(a, x)$ is the upper incomplete gamma function [161, p260 §6.5.3]. Continuing,

$$\begin{aligned}
\mathbb{E}\left[Q^2\right] &= \frac{4}{\Gamma(n/2)}\left[-\Gamma\left(\frac{n}{2}+2, \frac{x}{2}\right)\right]_0^\infty, \\
&= \frac{4}{\Gamma(n/2)}\Gamma\left(\frac{n}{2}+2\right), \\
&= \frac{4}{\Gamma(n/2)}\left(\frac{n}{2}+1\right)\Gamma\left(\frac{n}{2}+1\right), \\
&= 4\left(\frac{n}{2}+1\right)\frac{n}{2}, \\
&= n^2+2n. \\
\mathrm{Var}\left[\|\epsilon\|_2^2\right] &= \sigma^4\left(n^2+2n\right) - \sigma^4 n^2, \\
&= 2\sigma^4 n. \qquad \text{(A.9)}
\end{aligned}$$

□

### A.1.2 Norm of a perturbed random vector

Once the gradients are summed, they are then multiplied by the learning rate — which corresponds to a scaling of the variance — before being added to the parameters, or in this model, the non-random weight vector.

**Lemma A.8.** *The expected 2-norm of $x+\epsilon$, where $\epsilon$ is an $n$-dimensional random vector with elements sampled i.i.d. from $\mathcal{N}(0, \sigma^2)$ is less than or equal to $\sqrt{\|x\|_2^2+\sigma^2 n}$.*

*Proof of lemma A.8.*

$$\begin{aligned}
\|x+\epsilon\|_2 &= \sqrt{\sum_{i=1}^{n}(x_i+\epsilon_i)^2} \\
&= \sqrt{\|x\|_2^2+\|\epsilon\|_2^2+2\sum_{i=1}^{n}x_i\epsilon_i}
\end{aligned}$$

Using Jensen's inequality, lemma A.4, and that $\mathbb{E}[\epsilon_i]=0$.

$$\begin{aligned}
\mathbb{E}\left[\|x+\epsilon\|_2\right] &\leq \sqrt{\|x\|_2^2+\mathbb{E}\left[\|\epsilon\|_2^2\right]+2\sum_{i=1}^{n}x_i\mathbb{E}\left[\epsilon_i\right]} \\
&\leq \sqrt{\|x\|_2^2+\sigma^2 n}. \qquad \text{(A.10)}
\end{aligned}$$

□

### A.1.3 Norm of a sum of random vectors

We will also look at decaying sums of random vectors, since almost all optimisation algorithms use some form of momentum. Additionally, these results will be key to analysing optimisation as an Auto-regressive process – section 3.2.6.

**Lemma A.9.** *The expected norm of the sum of k n-dimensional random vectors, with elements sampled i.i.d. from* $\mathcal{N}(0, \sigma^2)$ *is* $\sigma\sqrt{2k}G(n)$.

*Proof of lemma A.9.* Let $\zeta_j = \sum_{i=1}^{k} \epsilon_{ij}$, and $\zeta_i \sim \mathcal{N}(0, k\sigma^2)$, then from lemma A.3

$$\mathbb{E}\left[\left\|\sum_{i=1}^{k} \epsilon_i\right\|\right] = \mathbb{E}\left[\|\zeta\|\right] = \sigma\sqrt{2k}G(n) \tag{A.11}$$

□

**Lemma A.10.** *The expected norm of the* $\lambda$*-decayed sum of k n-dimensional random vectors, with elements sampled i.i.d. from* $\mathcal{N}(0, \sigma^2)$ *is* $\sigma\sqrt{2\frac{\lambda^{2k}-1}{\lambda^2-1}}G(n)$, *and the expected square norm is* $\sigma^2 n\frac{\lambda^{2k}-1}{\lambda^2-1}$.

*Proof of lemma A.10.* First, the expected $\lambda$-decayed sum of $k$ scalars, $x_i$, is

$$\mathbb{E}\left[\sum_{i=1}^{k} \lambda^{i-1}x_i\right] = \sum_{i=1}^{k} \lambda^{i-1}\mathbb{E}\left[x_i\right] = 0. \tag{A.12}$$

Next, the variance of a $\lambda$-decayed sum of $k$ scalars is

$$\begin{aligned}
\mathrm{Var}\left[\sum_{i=1}^{k} \lambda^{i-1}x_i\right] &= \mathbb{E}\left[\left(\sum_{i=1}^{k} \lambda^{i-1}x_i\right)^2\right] - \mathbb{E}\left[\sum_{i=1}^{k} \lambda^{i-1}x_i\right]^2, \\
&= \mathbb{E}\left[\sum_{i=1}^{k} \lambda^{2(i-1)}x_i^2\right], \\
&= \sum_{i=1}^{k} \lambda^{2(i-1)}\mathbb{E}\left[x_i^2\right], \\
&= \sum_{i=1}^{k} \lambda^{2(i-1)}\left(\mathrm{Var}\left[x_i\right] + \mathbb{E}\left[x_i\right]^2\right), \\
&= \sum_{i=1}^{k} \lambda^{2(i-1)}\sigma^2, \\
&= \sigma^2\frac{1-\lambda^{2k}}{1-\lambda^2}.
\end{aligned} \tag{A.13}$$

Now, let $\zeta_j = \sum_{i=1}^{k} \lambda^{i-1}\epsilon_{ij}$; and so, from equations (A.12) and (A.13), $\zeta_i \sim \mathcal{N}(0, \sigma^2\frac{\lambda^{2k}-1}{\lambda^2-1})$. Then from lemma A.3,

$$\mathbb{E}\left[\left\|\sum_{i=1}^{k}\lambda^{i-1}\epsilon_i\right\|_2\right] = \mathbb{E}\left[\|\zeta\|_2\right] = \sigma\sqrt{2\frac{1-\lambda^{2k}}{1-\lambda^2}}\,G(n). \tag{A.14}$$

Lastly, using lemma A.4, the squared norm of the $\lambda$-decayed sum of $k$ $n$-dimensional random vectors,

$$\mathbb{E}\left[\left\|\sum_{i=1}^{k}\lambda^{i-1}\epsilon_i\right\|_2^2\right] = \mathbb{E}\left[\|\zeta\|_2^2\right] = \sigma^2 n\frac{1-\lambda^{2k}}{1-\lambda^2}. \tag{A.15}$$

□

Experimental code is given in appendix C, listing C.2.

### A.1.4 Cosine similarity metric

We use the cosine similarity metric

$$S_{\cos}(x,y) = \cos(\theta) = \frac{\langle x\cdot y\rangle}{\|x\|_2\|x\|_2}, \tag{A.16}$$

in this work since it allows the comparison of the directions of high-dimensional vectors, and so obtain insight about the surface we are optimising on. However, we note that the significance of the cosine between two random vectors depends on their size.

**Theorem A.11.** *The expected cosine between two random vectors, $x$ and $y$, with elements sampled i.i.d. from $D \sim \mathcal{N}(0, \sigma^2)$ is 0.*

*Proof of theorem A.11.*

$$\mathbb{E}\left[S_{\cos}(x,y)\right] = \mathbb{E}\left[\frac{x\cdot y}{\|x\|_2\|y\|_2}\right]. \tag{A.17}$$

Since the numerator is a product of two symmetric distributions, and the denominator is strictly positive; the resulting distribution is also symmetric and thus has an expectation of 0. □

**Theorem A.12.** *The variance of the cosine between two n-dimensional random vectors, $x$ and $y$, with elements sampled i.i.d. from $\mathcal{N}(0, \sigma^2)$ is $\frac{1}{n}$.*

*Proof of theorem A.12.*

$$S_{\cos}(x,y) = \frac{x^\top y}{\|x\|\|y\|} = \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum x_i^2}\sqrt{\sum y_i^2}}$$

$$\mathrm{Var}[S_{\cos}(x,y)] = \mathbb{E}\left[\left(\frac{\sum x_i y_i}{\|x\|\|y\|}\right)^2\right] = \mathbb{E}\left[\frac{(\sum x_i y_i)^2}{\sum x_i^2 \sum y_i^2}\right]$$

Then, from[165], [166], The expected squared cosine between two *i.i.d.* standard Gaussian vectors in $\mathbb{R}^n$ is,

$$\mathbb{E}[S_{\cos}(x,y)^2] = \frac{1}{n}$$

This follows from explicit evaluation of the joint distribution of dot products and norms for Gaussian vectors. Thus,

$$\mathrm{Var}[S_{\cos}(x,y)] = \frac{1}{n}.$$

□

Experimental code for these results is given in appendix C, listing C.3.

Let us find the threshold, $k$, for which the probability that the cosine between two random vectors is greater than $k$ is less than $p$; *i.e.* the probability that the two vectors are not independently randomly distributed is $1-p$.

**Theorem A.13.** *The probability that the cosine between two n-dimensional random vectors, x and y, with elements sampled i.i.d. from $\mathcal{N}(0, \sigma^2)$ is greater than k is bounded by $\frac{1}{nk^2}$.*

*Proof of theorem A.13.* From Chebichev's inequality [164, p.39], for a random variable $X$, and $k > 0$,

$$\mathrm{P}\left[|X - \mathbb{E}[X]| \geq k\right] \leq \frac{\mathrm{Var}[X]}{k^2},$$

and so from theorems A.11 and A.12,

$$\mathrm{P}\left[|S_{\cos}(x,y)| \geq k\right] \leq \frac{1}{nk^2}. \tag{A.18}$$

□

Rearranging theorem A.13 to find the threshold, $k$, such that the probability of the cosine being greater than $k$ is less than 1%,

$$\mathrm{P}\left[|S_{\cos}(x,y)| \geq k\right] \leq p.$$
$$k = \frac{1}{\sqrt{np}}. \tag{A.19}$$

Then, with $p = 0.01$,

$$k = \frac{10}{\sqrt{n}}. \tag{A.20}$$

This threshold is important because a cosine similarity of 0.1 might seem small, but it is larger than our threshold for vectors with more than $10,000$ dimensions. That is, it is unlikely that two vectors of dimension greater than $10,000$ with a cosine similarity of 0.1 or greater were constructed independently. This is a weak inequality as both Chernoff bounds and Hoeffding's inequality could be used here to get a tighter bound, but this is sufficient for our purposes in showing how a small cosine similarity is significant in high dimensions.

## A.2 Random Walks

**Lemma A.14.** *For a random walk, the variance of the cumulative increment distance* $Var\left[D_s(t)\right]$ *is approximately* $t\sigma^2(n - 2G(n)^2)$.

*Proof of lemma A.14.* First, the expectation of the square of the cumulative increment distance is,

$$\mathbb{E}\left[D_s(t)^2\right] = \mathbb{E}\left[\left(\sum_{i=1}^{t}\|s_i\|_2\right)^2\right] = \mathbb{E}\left[\sum_{i=1}^{t}\|s_i\|_2^2\right] + \sum_{i\neq j}\mathbb{E}\left[\|s_i\|_2\right]\mathbb{E}\left[\left\|s_j\right\|_2\right]. \tag{A.21}$$

Using lemmas A.3 and A.4,

$$\begin{aligned}\mathbb{E}\left[D_s(t)^2\right] &= tn\sigma^2 + \sum_{i\neq j}\mathbb{E}\left[\|s_i\|_2\right]\mathbb{E}\left[\left\|s_j\right\|_2\right],\\ &= tn\sigma^2 + \sum_{i\neq j}\left(\sigma\sqrt{2}G(n)\right)^2,\\ &= tn\sigma^2 + t(t-1)2\sigma^2G(n)^2.\end{aligned}$$

Now the variance is,

$$\begin{aligned}\mathrm{Var}\left[D_s(t)\right] &= \mathbb{E}\left[D_s(t)^2\right] - \mathbb{E}\left[D_s(t)\right]^2,\\ &= tn\sigma^2 + t(t-1)2\sigma^2G(n)^2 - \left(t\sigma\sqrt{2}G(n)\right)^2,\\ &= tn\sigma^2 - 2t\sigma^2G(n)^2,\\ &= t\sigma^2(n - 2G(n)^2).\end{aligned} \tag{A.22}$$

□

**Lemma A.15.** *For a random walk, the variance of the displacement* $Var\left[D_t(t)\right]$ *is* $\sigma^2t\left(n - 2G(n)^2\right)$.

*Proof of lemma A.15.* Let $S = \sum_{i=1}^{t} s_i$,

$$\text{Var}\left[D_t(t)\right] = \text{Var}\left[\|\theta_t - \theta_1\|_2\right] = \text{Var}\left[\left\|\sum_{i=1}^{t} s_i\right\|_2\right] = \text{Var}\left[\|S\|_2\right]. \tag{A.23}$$

Since the elements, $S_i$, of $S$ are a sum of $t$ independent random variables, $S \sim \mathcal{N}(0, t\sigma^2 \boldsymbol{I})$, and lemma A.6,

$$\text{Var}\left[D_t(t)\right] = \sigma^2 t \left(n - 2G(n)^2\right). \tag{A.24}$$

□

# II

# Source for chapter 1

```
#! /usr/bin/env python

from __future__ import annotations
from typing import Any
import matplotlib.pyplot as plt
import numpy as np


def bplot(
    ax: plt.Axes,
    pos: int,
    data: list[float],
    offset: float = 0,
    s: float = 1,
    **plot_kwargs: Any,  # noqa: ANN401
) -> None:
    """Plot a single box plot from a list of data."""
    x = np.array(data)
    v000 = x.min()
    v025 = np.percentile(x, 25)
    v050 = np.median(x)
    v075 = np.percentile(x, 75)
    v100 = x.max()

    line = ax.plot(
        [pos + offset, pos + offset],
        [v000, v025],
        linewidth=s / 2,
        **plot_kwargs,
    )
    if "color" not in plot_kwargs:
        plot_kwargs["color"] = line[0].get_color()
    ax.plot([pos + offset, pos + offset], [v075, v100], linewidth=s / 2, **plot_kwargs)
    ax.scatter([pos + offset], [v050], s=s * 3, **plot_kwargs)


# Adapted from
# https://matplotlib.org/3.1.1/gallery/pyplots/boxplot_demo_pyplot.html

# Fixing random state for reproducibility
np.random.seed(19680801)
```

```
# fake up some data
spread = np.random.rand(50) * 100
center = np.ones(25) * 50
flier_high = np.random.rand(10) * 100 + 100
flier_low = np.random.rand(10) * -100
data1 = np.concatenate((spread, center, flier_high, flier_low))
data2 = np.concatenate((spread, center, flier_high))
data3 = np.concatenate((spread, center))


fig, axs = plt.subplots(1, 2, figsize=(8,6))
# axs[0].set_title("Tukey's Box Plot")
axs[0].boxplot(data1, positions=[1], whis=100)
axs[0].boxplot(data2, positions=[2], whis=100)
axs[0].boxplot(data3, positions=[3], whis=100)

# axs[1].set_title("Tufte's Box Plot")
bplot(axs[1], 1, data1, s=2, color="black")
bplot(axs[1], 2, data2, s=2, color="black")
bplot(axs[1], 3, data3, s=2, color="black")

axs[0].spines["right"].set_color(None)
axs[0].spines["top"].set_color(None)
axs[1].spines["right"].set_color(None)
axs[1].spines["top"].set_color(None)
axs[1].set_xticks(range(4))
axs[1].set_xlim((0.5, 3.5))

fig.savefig("box_plots.png", bbox_inches="tight" )
```

LISTING B.1: Source for box plot style comparison.

# III

# Experimental code for appendix A.1

```
import math
import torch as th
import os
from matplotlib import pyplot as plt

# Use double precision to delay numerical errors
# when computing gammas with a large n
th.set_default_dtype(th.float64)

# Compute the gamma functions in log space to avoid computational errors
def G(n):
    lgamma_n2_p1 = th.tensor(n).add(1).div(2).lgamma()
    lgamma_n2 = th.tensor(n).div(2).lgamma()
    return (lgamma_n2_p1 - lgamma_n2).exp()

def G_approx(n):
    return math.sqrt(n/2)
    # # A better approximation for some reason
    # # maybe it partially takes into account more terms in the sterling approx
    # return math.sqrt(n/2-1/4)


def plot(x_max=10):
    scale = 4
    x = x_max*scale
    xs = [n/scale for n in range(scale, x)]
    Gs = [G(n) for n in xs]
    G_approxs = [G_approx(n) for n in xs]

    fig, axs = plt.subplots(1, 1, figsize=(8, 6))
    axs.set_xlabel("n")
    axs.spines["right"].set_color(None)
    axs.spines["top"].set_color(None)
    axs.plot(xs, Gs, label="$G(n)$")
    axs.plot(xs, G_approxs, label="$\sqrt{n/2}$")
    axs.legend()
    dir_path = os.path.dirname(os.path.realpath(__file__))
    fig.savefig(dir_path + f"/G_vs_G_approx_{x_max}.png", bbox_inches="tight")

plot(10)
plot(100)
```

LISTING C.1: Comparison of $G(n)$ and the approximation of $G(n)$ using Sterling's formula, corollary A.2.

```
import math
from statistics import variance
import torch as th

n = 1000
decay = 0.3
count = 100  # k
sigma = 1.78


def calc():
    total = th.zeros(n)
    a = th.ones(1)
    for _ in range(count):
        rand = sigma * th.randn(n)
        total.add_(a * rand)
        a.mul_(decay)
    return total.norm().item()


# Compute the gamma functions in log space to avoid computational errors
def G(n):
    lgamma_n2_p1 = th.tensor(n).add(1).div(2).lgamma()
    lgamma_n2 = th.tensor(n).div(2).lgamma()
    return (lgamma_n2_p1 - lgamma_n2).exp()


exp = sigma * math.sqrt(2*(decay**(2*count) - 1)/(decay**2 -1)) * G(n)

# Compute the value `experiment_count` times to get a mean and var
experiment_count = 200
totals = [calc() for _ in range(experiment_count)]
totals_avg = sum(totals) / experiment_count
totals_var = variance(totals)

print(
    f"Expectation:    {exp}\n"
    f"Computed avg:   {totals_avg}\n"
    f"Computed var:   {totals_var}\n"
)
```

LISTING C.2: Simulation of the expected norm of the $\lambda$-decayed sum of $k$ $n$-dimensional random vectors, with elements sampled *i.i.d.* from $\mathcal{N}(0, \sigma^2)$, lemma A.10.

```
import torch as th
from torch.nn.functional import cosine_similarity as cosine

n = 3

count = 100000
cosines = []
for _ in range(count):
    x = th.randn(n)
    y = th.randn(n)
```

```
    cosines.append(cosine(x,y,0))

cosines = th.tensor(cosines)
avg = cosines.mean()
var = cosines.var()

print(f"Expectation:  0\n"
      f"Average:      {avg}\n"
      f"Var:          {1/n}\n"
      f"Empircal Var: {var}"
      )
```

LISTING C.3: Simulation of the expected value and variance of the cosine between two $n$-dimensional random vectors, theorems A.11 and A.12

```
import math
import torch as th

# Use double precision to delay numerical errors
# when computing gammas with a large n
th.set_default_dtype(th.float64)

# n = 542_346
# n = 5426
n = 346

# Compute the average norm of a random vector empirically
count = 1000
items = []
for _ in range(count):
    a = th.randn(n)
    # Equivalent to: a.dot(a).sum().sqrt()
    items.append(a.norm())
items = th.tensor(items)
avg_norm = items.mean()
computed_norm_var = items.var()


# Compute the gamma functions in log space to avoid computational errors
def G(n):
    lgamma_n2_p1 = th.tensor(n).add(1).div(2).lgamma()
    lgamma_n2 = th.tensor(n).div(2).lgamma()
    return (lgamma_n2_p1 - lgamma_n2).exp()

expected_norm = math.sqrt(2) * G(n)


# Bound from Jensen's inequality
bound = math.sqrt(n)

var = n - 2*G(n)**2


print(
    f"Computed norm:   {avg_norm}\n"
    f"Expected norm:   {expected_norm}\n"
    f"Expected bound:  {bound}\n"
    f"Computed Var:    {computed_norm_var}\n"
    f"Var:             {var}"
```

```
)
```

LISTING C.4: Simulation of the expected 2-norm of an $n$-dimensional random vector, lemma A.3

# IV

# Model Summaries

### D.0.1 BasicCNN

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 16, 16]             896
       BatchNorm2d-2           [-1, 32, 16, 16]              64
            Conv2d-3             [-1, 32, 8, 8]           9,248
       BatchNorm2d-4             [-1, 32, 8, 8]              64
            Conv2d-5             [-1, 32, 4, 4]           9,248
       BatchNorm2d-6             [-1, 32, 4, 4]              64
            Linear-7                   [-1, 10]           5,130
          BasicCNN-8                   [-1, 10]               0
================================================================
Total params: 24,714
Trainable params: 24,714
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 0.16
Params size (MB): 0.09
Estimated Total Size (MB): 0.27
----------------------------------------------------------------
```

# V

# Appendices for the quasi-Newton method in the class-gradient sub-space

## E.1 Comparison of SGD and quasi-Newton's

| $\eta$ | Quasi-Newton | SGDM |
|---|---|---|
| 1e-1 | 88 | 92 |
| 5e-2 | 87 | 92 |
| 1e-2 | 82 | 90 |

TABLE E.1: Accuracy (%)

| $\eta$ | Quasi-Newton | SGDM |
|---|---|---|
| 1e-1 | 0.3574 | 0.3304 |
| 5e-2 | 0.3985 | 0.3346 |
| 1e-2 | 0.5362 | 0.3411 |

TABLE E.2: Loss

## E.2 On the Hessian

The main disadvantages of most second-order methods are the iteration time complexity, as computing the Hessian is $\mathcal{O}(N^2)$ and then computing its inverse is $\mathcal{O}(N^3)$, and the increased space complexity, $\mathcal{O}(N^2)$, to store the Hessian. The quasi-Newton's method avoids these pitfalls as it computes an approximation of the inverse Hessian in the same timescale as the gradient, $\mathcal{O}(N)$ and, since our Hessian approximation is formed from outer products, we can calculate a Hessian-vector-product without the quadratic space cost ($H = hh^\mathsf{T} \rightarrow Hz = (h^\mathsf{T} z)h$). However, it is still infeasible in its current form because it requires $C$ backward passes and storing $c_k$ means we require $C + 1$ times as much space as SGD.

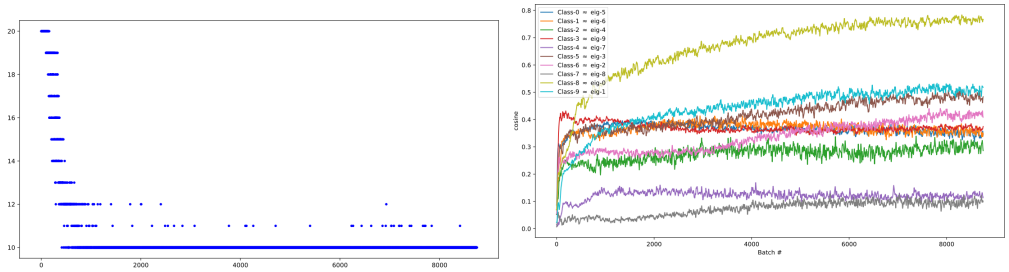
Incidentally, it is slightly odd to talk of a Hessian in a landscape with a discontinuous derivative. With ReLU activations we have a continuous piecewise linear surface

which has zero Hessians almost everywhere. But even in one dimension, if we approximate a quadratic by a piecewise linear curve, then, although the second derivative is zero almost everywhere, we can still diverge using gradient descent if the step size is too big since the function is non-locally curved.

## E.3 Extent of subspace intersection

Although there is general consensus that the class vector directions align with the top eigenvectors of the Hessian, we experimentally check to what degree this is true since this is a probable reason for the worse performance of the quasi-Newton's method.

We collect the gradient directions for each class in the mini-batch, $c_k^{\mathcal{B}^{(t)}}$ while training a simple 60k parameter CNN on CIFAR-10. At the end of training (50 epochs) we calculate the full eigensystem of the trained model. We can then plot the correlation and check whether the directions are largely stable.




(A) Rank of the combined matrix for each batch.

(B) Class gradients assigned to the top-$C$ eigenvectors at the end of training.

FIGURE E.1: Similarity measures between the class gradients and the top-$C$ eigenvectors

The cosine-maximised linear assignment for the top-$C$ eigenvectors to the class gradient directions is $\approx 0.36$, and most of the class gradients have a significant cosine similarity with a top eigenvector figure E.1b. This is a significant correlation since random 60k-dimensional vectors have an expected cosine of order $10^{-3}$.

We can also look at the rank of the combined matrix

$$[v_1, \ldots, v_C,\ e_0, \ldots, e_C], \tag{E.1}$$

where $e_i$ is the top $i^{\text{th}}$ true eigenvector of the fully trained model. If the subspace spanned by the top-$C$ eigenvectors is the same as that which is spanned by the class gradients then the rank of this matrix should be $C$. We see, from figure E.1a, that the rank quickly declines from $2C$, *i.e.* no intersection, to consistently being $C$. However, there are several batches when the rank increases by one or two implying that these batches have a greater amount of noise.

The rank of the combined matrix is a strong indicator that it is covered, but the noise may introduce spurious results due to the precision error in calculating the rank.

## E.4 Hyperparameters

For all the experiments we choose a learning rate of 0.1 (unless otherwise specified), basic momentum of with a decay parameter of 0.9, and a gamma 0.9 for the quasi-Newton's method, and 0 weight decay or regularization. These values were not optimised.

# VI

# Appendices for estimating the top eigenvector of the Hessian

```
#!/usr/bin/env python
# coding: utf-8

# python 3.8.5
import torch # v1.5.1
import numpy as np #v1.24.4
import matplotlib.pyplot as plt # v3.7.4

import math
from functools import partial

torch.set_printoptions(precision=None, threshold=None, edgeitems=None, linewidth=100, profile=None, s

def l1_norm(v, sum_dim=0):
    return torch.sum(torch.abs(v), dim=sum_dim)

def l2_norm(v, sum_dim=0):
    return torch.sqrt(torch.sum(torch.pow(v, 2), dim=sum_dim))

def li_norm(v1, v2=None):
    if v2 is not None:
        v1 = v1.abs() - v2.abs()
    return torch.max(torch.abs(v1))

def lp_norm(v, p, sum_dim=0):
    return torch.pow(torch.sum(torch.pow(v, p), dim=sum_dim), (1/p))

# Estimating the top eigenvector of the Hessian
size = 100, 100
X = torch.randn(size)
H = X.T@X

true_eig = H.eig(eigenvectors=True)
true_top_eigvec = true_eig.eigenvectors[:,0]
true_top_eigval = true_eig.eigenvalues[0,0]

print(f'True top eigenvalue: {true_top_eigval:.2f}')
# print(f'True top eigenvector:\n{true_top_eigvec}')
```

```
## Estimating directly from the Hessian
H_top_eigvec = torch.randn(size[1])
H_top_eigval = 0
H_delta_eigval = 1
coss = []
eigvals = []
for i in range(100):
    H_top_eigvec = H@H_top_eigvec

    H_delta_eigval = (H_top_eigval - l2_norm(H_top_eigvec)).abs()
    H_top_eigval = l2_norm(H_top_eigvec)
    H_top_eigvec /= H_top_eigval # normalise eigenvec

    eigvals.append(H_top_eigval)
    coss.append(true_top_eigvec@H_top_eigvec)

    if H_delta_eigval < 0.05: break

fig = plt.figure(figsize=(14, 7))
ax = fig.add_subplot(211)
ax.plot(coss)
ax.set_ylim((-1,1))
ax.set_ylabel(r'$\cos\phi$')
ax.set_xlabel('iterate')
ax.spines["right"].set_color(None)
ax.spines["top"].set_color(None)
ax = fig.add_subplot(212)
# ax.set_ylabel(r'$\lambda$')
ax.set_xlabel('iterate')
ax.spines["right"].set_color(None)
ax.spines["top"].set_color(None)
ax.plot(eigvals, label=r"$\lambda$ estimate")
ax.axhline(true_top_eigval, color="orange", zorder=-1, label=r"true $\lambda$")
ax.legend(loc="best", frameon=False)
fig.savefig("top_eig_known_H.png", bbox_inches="tight")


print("Known H")
print(f'$L_\infty = {li_norm(true_top_eigvec, H_top_eigvec):.3f}$')
print(f'$\cos\phi = {true_top_eigvec@H_top_eigvec:.3f}$')

## Estimating from gradients
eps = 0.1
G_top_eigvec = torch.randn(size[1])
coss = []
eigvals = []
for i in range(100):
    r = eps*torch.randn(size[1], 1)
    G = H@r
    G2 = G@G.T
    G_top_eigvec = G2@G_top_eigvec
    G_top_eigval = l2_norm(G_top_eigvec)
    G_top_eigvec /= G_top_eigval
    eigvals.append(math.sqrt(G_top_eigval)/eps)
    coss.append(true_top_eigvec@G_top_eigvec)


fig = plt.figure(figsize=(14, 7))
ax = fig.add_subplot(211)
```

```
ax.plot(coss)
ax.set_ylim((-1,1))
ax.set_ylabel(r'$\cos\phi$')
ax.set_xlabel('iterate')
ax.spines["right"].set_color(None)
ax.spines["top"].set_color(None)
ax = fig.add_subplot(212)
ax.set_xlabel('iterate')
ax.spines["right"].set_color(None)
ax.spines["top"].set_color(None)
ax.plot(eigvals, label=r"$\lambda$ estimate")
ax.axhline(true_top_eigval, color="orange", zorder=-1, label=r"true $\lambda$")
ax.legend(loc="best", frameon=False)
fig.savefig("top_eig_from_gradients.png", bbox_inches="tight")

print("Estimating from gradients")
print(f'$L_\infty = {li_norm(true_top_eigvec, G_top_eigvec):.3f}$')
print(f'$\cos\phi = {true_top_eigvec@G_top_eigvec:.3f}$')

## Improving accuracy with a larger sample size of gradients
sample_size = 1000
eps = 0.1
G_top_eigvec = torch.randn(size[1])
coss = []
eigvals = []
for i in range(100):
    r = eps*torch.randn(size[1], sample_size)
    G = H@r
    G2 = G@G.T
    G_top_eigvec = G2@G_top_eigvec
    G_top_eigval = l2_norm(G_top_eigvec)
    G_top_eigvec /= G_top_eigval
    eigvals.append(math.sqrt(G_top_eigval/sample_size)/eps)
    coss.append(true_top_eigvec@G_top_eigvec)

fig = plt.figure(figsize=(14, 7))
ax = fig.add_subplot(211)
ax.plot(coss)
ax.set_ylim((-1,1))
ax.set_ylabel(r'$\cos\phi$')
ax.set_xlabel('iterate')
ax.spines["right"].set_color(None)
ax.spines["top"].set_color(None)
ax = fig.add_subplot(212)
# ax.set_ylabel(r'$\lambda$')
ax.set_xlabel('iterate')
ax.spines["right"].set_color(None)
ax.spines["top"].set_color(None)
ax.plot(eigvals, label=r"$\lambda$ estimate")
ax.axhline(true_top_eigval, color="orange", zorder=-1, label=r"true $\lambda$")
ax.legend(loc="best", frameon=False)
fig.savefig("top_eig_from_gradients_large_sample.png", bbox_inches="tight")

print("Estimating from many gradients")
print(f'$L_\infty = {li_norm(true_top_eigvec, G_top_eigvec):.3f}$')
print(f'$\cos\phi = {true_top_eigvec@G_top_eigvec:.3f}$')


## Estimating from iterative gradients
```

```
# (rolling window of last `sample_size` grads)

def iterative_gradients(sample_size = 100):
    coss = []
    GI_top_eigvec = torch.randn(size[1])
    G = torch.randn(size[1], sample_size)
    for i in range(10_000): # for each datum
        G[:, i%sample_size] = H@torch.randn(size[1])
        G2 = G@G.T
        GI_top_eigvec = G2@GI_top_eigvec
        GI_top_eigval = l2_norm(GI_top_eigvec)
        GI_top_eigvec /= GI_top_eigval

        coss.append(true_top_eigvec@GI_top_eigvec)

    return GI_top_eigvec, coss

GI_top_eigvec, coss = iterative_gradients()

print("Estimating from iterated gradients")
print(f'$L_\infty = {li_norm(true_top_eigvec, GI_top_eigvec):.3f}$')
print(f'$\cos\phi = {true_top_eigvec@GI_top_eigvec:.3f}$')


def plot_cos_phis(ax, cos_phis, s=1000):
    ax.plot(cos_phis)
    ax.set_ylim((-1,1))
    ax.set_ylabel(r'$\cos\phi$')
    ax.set_xlabel('time')
    ax.spines["right"].set_color(None)
    ax.spines["top"].set_color(None)

fig = plt.figure(figsize=(14, 7))
ax = fig.add_subplot(111)
plot_cos_phis(ax, coss)
fig.savefig("top_eig_cos_phis.png", bbox_inches="tight")


fig = plt.figure(figsize=(14, 18))

sample_size = 100
ax = fig.add_subplot(311)
plot_cos_phis(ax, iterative_gradients(sample_size)[1], sample_size)
sample_size = 1000
ax = fig.add_subplot(312)
plot_cos_phis(ax, iterative_gradients(sample_size)[1], sample_size)
sample_size = 10_000
ax = fig.add_subplot(313)
plot_cos_phis(ax, iterative_gradients(sample_size)[1], sample_size)
fig.savefig("top_eig_cos_phis_compare.png", bbox_inches="tight")


## Estimating from iterative gradients with gradient descent
step_size = 1e-2

GID_top_eigvec = torch.randn(size[1])
coss = []
eigvals = []
for i in range(10_000): # for each datum
```

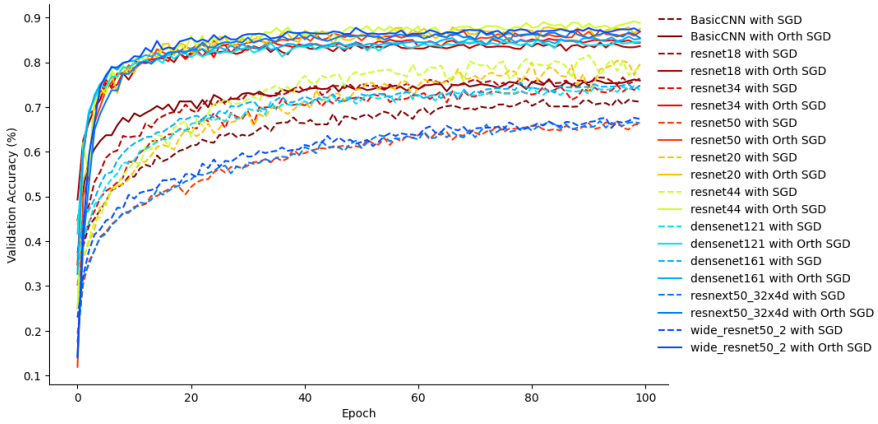
```
    r = torch.randn(size[1], 1)
    G = H@r
    # %TODO: scaling
    # G /= G.norm()
    G2 = G@G.T
    # G2 /= li_norm(G2) # normalise to keep scaling low
    # Gradient Descent
    GID_top_eigvec -= step_size * (GID_top_eigvec - G2@GID_top_eigvec)
    GID_top_eigval = l2_norm(GID_top_eigvec) # todo: have to get norm of G2@top_vec not this
    GID_top_eigvec /= GID_top_eigval
    coss.append(true_top_eigvec@GID_top_eigvec)
    eigvals.append(math.sqrt(GID_top_eigval))

fig = plt.figure(figsize=(14, 7))
ax = fig.add_subplot(111)
ax.plot(coss)
ax.set_ylim((-1,1))
ax.set_ylabel(r'$\cos\phi$')
ax.set_xlabel('time')
ax.spines["right"].set_color(None)
ax.spines["top"].set_color(None)
fig.savefig("top_eig_cos_phis_sgd.png", bbox_inches="tight")

print("Estimating from iterated gradients with gradient descent")
print(f'$L_\infty = {li_norm(true_top_eigvec, GID_top_eigvec):.3f}$')
print(f'$\cos\phi = {true_top_eigvec@GID_top_eigvec:.3f}$')
```

LISTING F.1: Code to estimate the top eigenvector of the Hessian using the power iteration method.

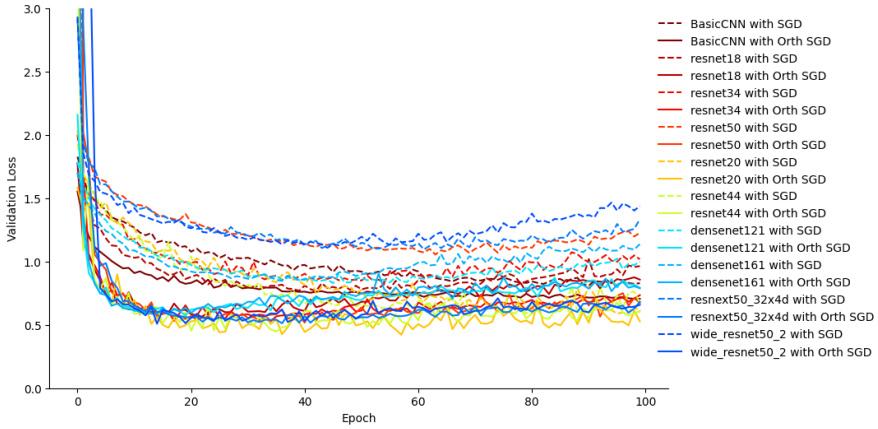
# VII

# Appendices for orthogonal optimisers

## G.1 Full results plot




(A) Validation Accuracy




(B) Validation Loss

FIGURE G.1: SGDM vs Orthogonal SGDM

# VIII

# Approximate inference of the eigenspectrum of the Hessian

Assume we are in an $n$-dimensional quadratic minim problem, i.e. the loss function is given by

$$\mathcal{L}(\theta) = \frac{1}{2}(\theta - \theta^\star)^T H(\theta - \theta^\star), \tag{H.1}$$

such that the Hessian $H$ is symmetric and positive semi-definite. We are interested in the eigenspectrum of the Hessian, i.e. the set of eigenvalues $\lambda_1, \ldots, \lambda_n$ of $H$.

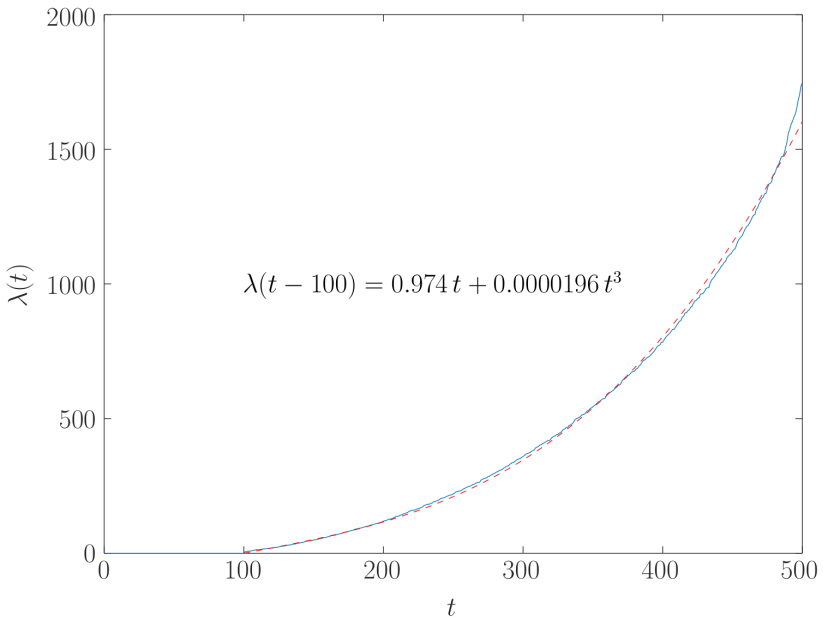



FIGURE H.1: The eigenvalues of a generated Hessian, $H$, in ascending order. The blue line are the true eigenvalues, and the red line is a fitted cubic. The Hessian was generated $H = X^\top X$ where $X$ is a $500 \times 400$ random matrix.

We can decompose $H$ as $H = V\Lambda V^T$, where $V$ is an orthogonal matrix with columns $v_1, \ldots, v_n$ being the eigenvectors of $H$, and $\Lambda$ is a diagonal matrix with diagonal entries $\lambda_1, \ldots, \lambda_n$ being the corresponding eigenvalues.

## H.1 Probability of squared gradients

The gradient of the loss function at $\theta$ is given by

$$\nabla\mathcal{L}(\theta) = H(\theta - \theta^\star). \tag{H.2}$$

Let us consider the random variable $\theta = \theta^\star + \epsilon$, where $\epsilon \sim \mathcal{N}(0, \boldsymbol{I})$ is a Gaussian noise. Then, we are interested in the distribution of the squared norm of the gradients,

$$\begin{aligned} s &= \|\nabla\mathcal{L}(\theta)\|^2, \\ &= \|H((\theta^\star + \epsilon) - \theta^\star)\|^2, \\ &= \|H\epsilon\|^2, \\ &= \epsilon^T H^2 \epsilon. \end{aligned} \tag{H.3}$$

The Fourier transform of this distribution is given by,

$$\begin{aligned} \tilde{f}(\omega) &= \mathbb{E}\left[\exp(i\omega s)\right], \\ &= \mathbb{E}\left[\exp\left(i\omega\epsilon^T H^2\epsilon\right)\right], \end{aligned}$$

writing the expectation explicitly,

$$\begin{aligned} &= \left(\prod_i \int \frac{d\epsilon_i}{\sqrt{2\pi}}\right) \exp\left(-\frac{1}{2}\|\epsilon\|^2\right) \exp\left(i\omega\epsilon^T H^2\epsilon\right), \\ &= \left(\prod_i \int \frac{d\epsilon_i}{\sqrt{2\pi}}\right) \exp\left(-\frac{1}{2}\epsilon^T\left(\boldsymbol{I} - 2i\omega H^2\right)\epsilon\right), \end{aligned} \tag{H.4}$$

as $H^2 = V\Lambda^2 V^T$, and since $V$ is orthogonal, $V^T V = \boldsymbol{I}$,

$$= \left(\prod_i \int \frac{d\epsilon_i}{\sqrt{2\pi}}\right) \exp\left(-\frac{1}{2}\epsilon^T V(\boldsymbol{I} + 2i\omega\Lambda^2)V^T\epsilon\right). \tag{H.5}$$

Let us perform a change of variables with $x = V^T\epsilon$, the Jacobian of the transformation is $V$ with determinant 1, and the Fourier becomes,

$$\tilde{f}(\omega) = \left(\prod_i \int \frac{dx_i}{\sqrt{2\pi}}\right) \exp\left(-\frac{1}{2}x^T\left(\boldsymbol{I} + 2i\omega\Lambda^2\right)x\right),$$

and since $\Lambda^2$ is diagonal,

$$= \prod_i \int \exp\left(-\frac{1}{2}x_i^2\right) \exp\left(-i\omega\lambda_i^2 x_i^2\right) \frac{dx_i}{\sqrt{2\pi}},$$

using $\int \exp\left(-\frac{1}{2}ax^2\right) dx = \frac{1}{\sqrt{a}}$,

$$\begin{aligned} &= \prod_i \frac{1}{\sqrt{1 + 2i\omega\lambda_i^2}}, \\ &= \exp\left(-\frac{1}{2}\sum_i \log\left(1 + 2i\omega\lambda_i^2\right)\right). \end{aligned} \tag{H.6}$$

Note that we only need to sum over the non-zero eigenvalues since $\log(1) = 0$.

Now, we can use the cumulant generating function, $K(\omega) = \log\left(\tilde{f}(\omega)\right)$,

$$\begin{aligned} K(\omega) &= \log\left(\tilde{f}(\omega)\right) \\ &= \sum_{n=1} \frac{i^n\omega^n}{n!}\kappa_n, \end{aligned}$$

where $\kappa_n$ are the cumulants, $\kappa_1$ is the mean, $\kappa_2$ is the variance, $\kappa_3$ is the third central moment, etc.

To show this,

$$\begin{aligned} \tilde{f}(\omega) &= \int f(s)\exp(i\omega s)\, ds, \\ &= \int f(s) \sum_{n=1} \frac{(i\omega s)^n}{n!} ds, \\ &= \sum_{n=1} \frac{(i\omega)^n}{n!}\mu_n, \end{aligned}$$

where $\mu_n = \mathbb{E}\left[s^n\right]$ is the $n^{\text{th}}$ moment of $f(s)$,

$$\begin{aligned} \log\left(\tilde{f}(\omega)\right) &= \log\left(\sum_{n=1} \frac{(i\omega)^n}{n!}\mu_n\right), \\ &= \log\left(1 + i\omega\mu_1 - \frac{1}{2}\omega^2\mu_2 - \frac{1}{3!}i\omega^3\mu_3 + \ldots\right), \\ &= \omega\mu_1 - \frac{1}{2}\omega^2(\mu_2 - \mu_1^2) - \frac{1}{3!}i\omega^3(\mu_3 - 2\mu_2 - \mu_1 + 2\mu_1^3) + \ldots, \\ &= i\omega\kappa_1 - \frac{1}{2}\omega^2\kappa_2 - \frac{1}{3!}i\omega^3\kappa_3 + \ldots. \end{aligned}$$

In addition, from equation (H.6),

$$\begin{aligned}\log\big(\tilde{f}(\omega)\big) &= -\frac{1}{2}\sum_i \log\big(1+2i\omega\lambda_i^2\big)\,,\\ &= i\omega\sum_i \lambda_i^2 - \omega^2\sum_i \lambda_i^4 - \frac{4}{3}i\omega^3\sum_i \lambda_i^6 + \ldots.\end{aligned}$$

Thus, comparing the two expressions,

$$\begin{aligned}\kappa_1 &= \sum_i \lambda_i^2,\\ \kappa_2 &= 2\sum_i \lambda_i^4,\\ \kappa_3 &= 8\sum_i \lambda_i^6.\end{aligned}$$

Computing this for a random matrix $H = X^T X$, where $X$ is a $500 \times 400$ random matrix shown in figure H.1, we get,

- $\kappa_1 = 1.7799 \times 10^8$ whilst $\sum_i \lambda_i^2 = 1.7842 \times 10^8$,
- $\kappa_2 = 4.9891 \times 10^{14}$ whilst $2\sum_i \lambda_i^4 = 4.9788 \times 10^{14}$,
- $\kappa_3 = 3.3612 \times 10^{21}$ whilst $8\sum_i \lambda_i^6 = 3.7424 \times 10^{21}$,

and figure H.2 shows the histogram.

## H.2 Approximating the eigenspectrum

We need to compute,

$$\sum_{i=1}^{n^\star} \lambda_i^{2k}, \tag{H.7}$$

where $n^\star$ is the number of non-zero eigenvalues, to calculate the $k^{\text{th}}$ cumulant.

Let $\lambda(t) = \lambda_{n^\star t}$ then,

$$\sum_{i=1}^{n} \lambda_i^{2k} \approx n^\star \int_0^1 \lambda(t)^{2k} dt, \tag{H.8}$$

and let us approximate $\lambda(t)$ as a cubic polynomial,

$$\lambda(t) = at + bt^3. \tag{H.9}$$

(This seems to be a good approximation for random matrices, as shown in figure H.1, but other equations may lead to better results in practice.)

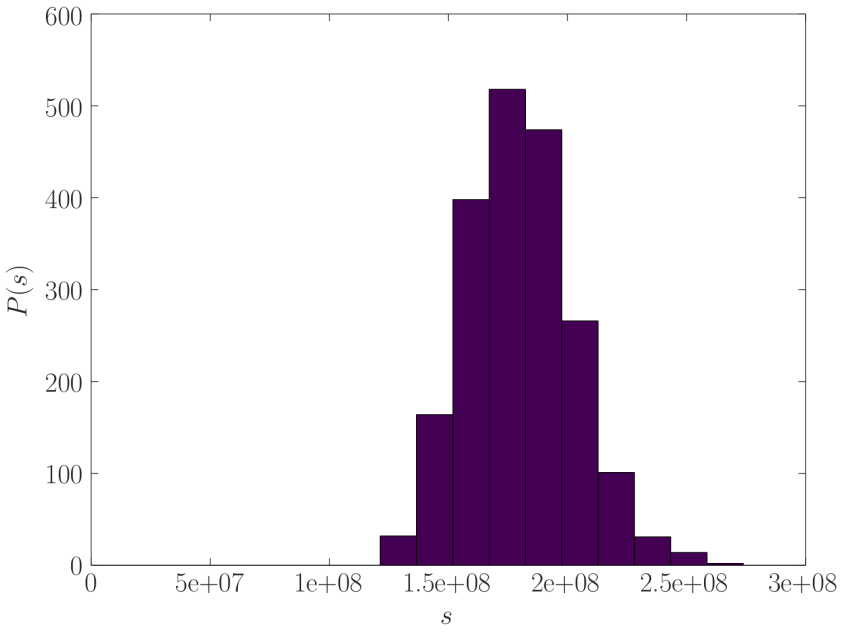



FIGURE H.2: Histogram of the squared gradients of a generated Hessian, $H = X^\top X$ where $X$ is a $500 \times 400$ random matrix.

Then,

$$\sum_{i=1}^{n} \lambda_i^{2k} \approx n^\star \left( \frac{a^2}{3} + \frac{2ab}{5} a^2 + \frac{b^2}{7} \right) = K_1,$$
$$2 \sum_{i=1}^{n} \lambda_i^{4} \approx n^\star \left( \frac{4a^2b^2}{3} + \frac{8a^3b}{7} + \frac{2a^4}{5} + \frac{8ab^3}{11} + \frac{2b^4}{12} \right) = K_2,$$
$$8 \sum_{i=1}^{n} \lambda_i^{6} \approx n^\star \left( \frac{120a^4b^2}{11} + \frac{160a^3b^3}{13} + 8a^2b^4 + \frac{16a^5b}{3} + \frac{8a^6}{7} + \frac{48ab^5}{17} + \frac{8b^6}{19} \right) = K_3.$$

## H.3 Future work

We probably want to model our estimation errors, $\sigma_n$, and use Bayes' rule,

$$f(n^\star, a, b | \kappa_1, \kappa_2, \kappa_3) = \frac{\prod_{n=1}^{m} \mathcal{N}(\kappa_n | K_n, \sigma_n^2) f(n^\star, a, b)}{f(\kappa_1, \kappa_2, \kappa_3)}, \tag{H.10}$$

meaning we want to put a prior on $n^\star, a, b$ or use maximum likelihood.

# IX

# Symmetries and Regularisation

What is the effect of symmetries on the optimisation of a neural network?

Symmetries in a network's weight space [39, Chapter 5.1.1] can lead to adverse effects and different minima from the same initial weights. Symmetries are believed to slow down optimisation as they introduce saddle points in an otherwise convex setting., Thus, much focus has been given to reducing the number of symmetries in a neural network, for example, skip connections [30], however, Badrinarayanan *et al.* [167] show that not all symmetries are as simple as a permutation.

## I.1 Linear and non-linear symmetries

If we have a function,

$$f(x) = \theta_2\theta_1 x \tag{I.1}$$

then we can also construct this exact function with a completely different set of weight matrices

$$f(x) = \hat{\theta_2}\hat{\theta_1}x = (\theta_2 A^{-1})(A\theta_1)x. \tag{I.2}$$

However, this only applies when $f(x)$ is linear, if we apply an element-wise non-linear function, $\sigma(\cdot)$, after the first weight, then

$$\theta_2\sigma(\theta_1 x) = \hat{\theta_2}\sigma(\hat{\theta_1}x), \tag{I.3}$$

if and only if $A$ is a permutation matrix, hereon denoted by $\pi$.

## I.2 Non Symmetry-breaking skip connections

Let

$$f(x) = \theta_2\sigma(x + \theta_1 x). \tag{I.4}$$

Given $\pi$ is a permutation, clearly

$$f(x) \neq \theta_2 \pi^{-1} \sigma(x + \pi\theta_1 x), \tag{I.5}$$

however, we can construct a matrix

$$A = (\pi\theta_1 - \boldsymbol{I})\theta_1^{-1}, \tag{I.6}$$

so that

$$f(x) = \theta_2 \pi^{-1} \sigma(x + A\theta_1 x) \tag{I.7}$$

given $\theta_1$ is non-degenerate. Thus, we have constructed a new symmetry direction from the old: $(\pi - \boldsymbol{I})\theta_1 - \boldsymbol{I}$ instead of $(\pi - \boldsymbol{I})\theta_1$. This shows why implementations span their skip connections across multiple layers: there must be at least one non-linearity between the weight matrix and the skip addition. Intuitively, since the addition is a linear operation, we can construct some matrix to be its inverse.

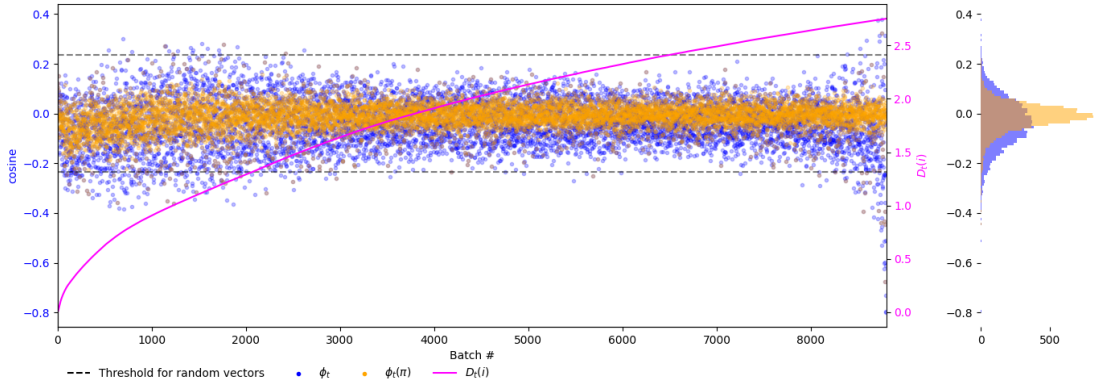
## I.3 Effect on the learning dynamics

We now consider the effect of these symmetries on the learning dynamics. We know these symmetries will introduce saddle points in the loss landscape but is the model actually affected by them? That is, does the model reside wholly in one of the symmetrical subspaces far enough from the saddle point borders? If the model traverses across one of these saddle points then the optimisation will definitely be disrupted, but if the model merely skirts around these saddle points then the optimisation will be slowed but still give approximately the same solution. Since it's often said that resnets work well since they break these symmetries [168], we expect adding skip connections to a basic CNN to reduce the effect of symmetries comared to the original CNN.

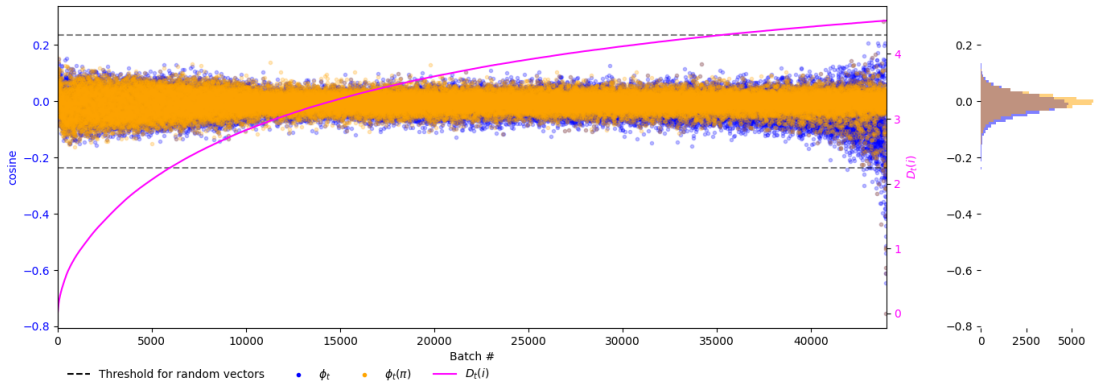
To discover if the model is affected by other symmetries we measure the cosine distance of the gradient to both the solution, $\phi_t$, equation (4.56), and a permuted version of the solution

$$\phi_t(\pi) = S_{\cos}\left(\pi\theta_\tau - \theta_t, \nabla\theta_t\right). \tag{I.8}$$

figure I.1a demonstrates that while most of the time the symmetry has less effect on the gradient, there are some batches where the gradient is more anticorrelated with the symmetry than the actual solution. Being approximately Gaussian distributed around zero, we could conclude that all of the symmetries are simply random noise, however, we can see that there are distributional shifts during training. For example, in figure I.1a the initial distribution if $\phi_t(\pi)$ is not centred at zero, but at a negative value; and in figure I.1b also displays this shift and a reducing of variance over time. Again, we see the peak of the distribution of $\phi_t$ is negative while the peak of $\phi_t(\pi)$ is at zero.




(A) 200 Epochs




(B) 1000 epochs

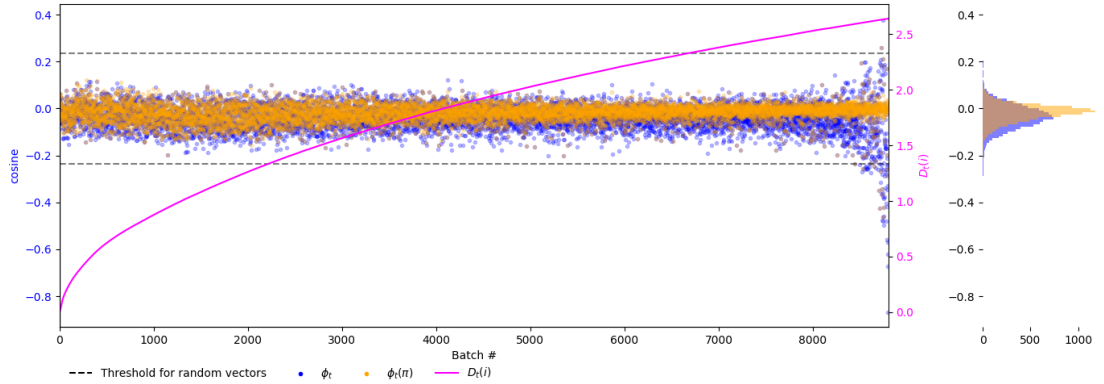
FIGURE I.1: $\phi_t$ and $\phi_t(\pi)$ for the second convolutional layer of a basic CNN on CIFAR10

With added skip connections, figure I.2, the distributional shifts of $\phi_t(\pi)$ have been removed, indicating that $\phi_t(\pi)$ is closer to a random variable. Unfortunately, in figure I.2 there are some points that exceed the random variable threshold it seems that permutation symmetries have not been eliminated entirely. Indeed, since we have only measured one arbitrary permutation per batch, it may be that there is at least one symmetry that is more anticorrelated to the gradient than the solution.

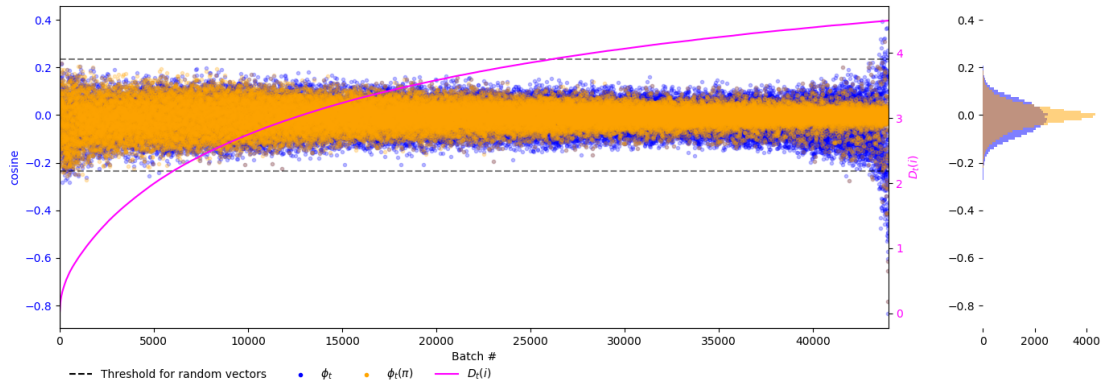
## I.4 Theories to investigate

Below we outline some theories we have on symmetries in deep neural networks that we will be investigating in the future.

We mentioned in appendix I.1 that, after the application of a non-linear function, we required the symmetry to be a permutation. However, this is for an arbitrary non-linear function; if we know the specific non-linearity we might be able to relax this requirement. For instance, if $\sigma(\cdot)$ is the ReLU function then equation (I.3) holds if and only if $A$ is non-negative monomial. This introduces a notion of the size of these symmetries — the biggest would be the complete linear transform, $A$, in equation (I.2),




(A) 200 Epochs




(B) 1000 epochs

FIGURE I.2: $\phi_t$ and $\phi_t(\pi)$ for the second convolutional layer of a basic CNN with skip connections on CIFAR10

the next biggest, the non-negative monomial for ReLUs, and the smallest the permutation symmetry for an arbitrary non-linearity. It should be investigated as to what the difference in effect is between these various sized of symmetry.

One of the easiest ways to remove symmetries is with regularisation, so it may be tempting to use it as a panacea. But do we want to remove *every* symmetry? You can view symmetries as providing multiple paths to a point in weight space; therefore, reducing the number of symmetries will reduce the number of available paths to a particular solution. In this light, it seems possible that reducing symmetries could harm the optimisation. If they are indeed a problem then does breaking them predominantly affect the generalisation performance or the optimisation performance, *i.e.* does it make it easier to train a model or lead to better results?

Orhan and Pitkow [168] reveal that skip connections help beyond simply breaking the symmetries, but they conclude that the breaking of symmetries does help the optimisation. In appendix I.2 we saw that it is possible to create a skip connection without breaking any symmetries. Therefore, we can test the extent to which a skip connection helps beyond breaking symmetries, by comparing normal skip connections and ones that don't break symmetries.