The Australian National University
2600 ACT | Canberra | Australia

Australian National University

School of Computing

College of Systems and Society (CSS)

Efficient Saddle-Free Optimisation for Deep Learning

— Honours project (S1 2024–2025)

A thesis submitted for the degree
Bachelor of Philosophy (Science)

By:
Pranav Pativada

Supervisors:
Dr. Dylan Campbell
Dr. João F. Henriques

May 2025

Declaration:

I declare that this work:

- upholds the principles of academic integrity, as defined in the University Academic Misconduct Rules;
- is original, except where collaboration (for example group work) has been authorised in writing by the course convener in the class summary and/or Wattle site;
- is produced for the purposes of this assessment task and has not been submitted for assessment in any other context, except where authorised in writing by the course convener;
- gives appropriate acknowledgement of the ideas, scholarship and intellectual property of others insofar as these have been used;
- in no part involves copying, cheating, collusion, fabrication, plagiarism or recycling.

May, Pranav Pativada

Acknowledgements

I would like to thank my supervisor, Dr. Dylan Campbell, for his invaluable guidance and support throughout this project. Dylan was always available to answer my questions and provide constructive feedback on my work. His feedback was very helpful and his directions urged me to improve my skills and think deeply about my research. I am truly grateful for his continued support.

I would also like to thank my second supervisor, Dr. João Henriques, for his valuable insights and suggestions. João's expertise in optimisation and his breakdown of complex concepts was extremely helpful in developing my understanding of core concepts.

Finally, I would like to thank my friends and family for their support and encouragement. To Evan Markou, Sam Bahrami, and Mingda Xu, thank you for the fun discussions and keeping me company nearby the office. To my parents, thank you for your love and support.

Abstract

Substantial progress has been made in deep learning in recent years, with neural networks transforming our lives. However, optimisation in deep learning faces fundamental challenges as neural network landscapes are dominated by saddle points. Current first-order methods are efficient and scalable, but struggle in these regions and cannot converge quickly. Second-order methods are more effective, but are computationally intractable. This presents the fundamental question—how do we achieve the benefits of second-order methods while maintaining the efficiency of first-order methods?

In this thesis, we present KryBall, a novel optimisation algorithm that combines the best of first and second-order methods. We use a Krylov subspace approach, alongside Saddle-Free-Newton dynamics and a quadratic trust-region framework to efficiently incorporate second-order information. Our results demonstrate that KryBall achieves rapid convergence on ill-conditioned problems and binary classification, outperforming the state-of-the-art, and is generally competitive on image classification. We also provide an analysis on the deep learning optimisation landscape and demonstrate key theoretical properties of our approach.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 The need for optimisation

Optimisation is everywhere. In nature, physical systems self-organise towards a state of minimum energy. Light rays travel in the most efficient path to their destination. In society, businesses strive to maximise users and profit. Engineering processes aim to maximise efficiency and minimise waste.

Optimisation is paramount to the performance of many real-world systems and models. Within the last decade, optimisation has been key in enabling the deep learning revolution in the field of machine learning. This has resulted in state-of-the-art performance in tasks such as ImageNet for image classification (Krizhevsky et al., 2017) and AlphaFold for protein folding prediction (Jumper et al., 2021). Recently, with advances in hardware, large language models have been able to offer agentic experiences that enable human-like interactions (Chowdhery et al., 2023).

For these models to perform well, they need to learn complex concepts from a set of data and be able to generalise. The process of navigating the model to do this is the role of optimisation in deep learning. Given some *objective*, a quantitative measure of the model's performance, that is dependent on a set of *parameters*, the optimisation process is finding the right parameters that optimises the objective. For example, the objective for protein folding prediction could be to find a minimum energy conformation and the parameters could be the positions of the atoms of proteins and their orientations.

Throughout the years, many optimisation algorithms have been proposed for deep learning. At the most fundamental level, these can be categorised into either first-order or second-order methods. First-order methods guide the model by using information obtained from the *first derivative* of the objective. Second-order methods use information from the *second derivative*. Currently widely adopted and state-of-the-art optimisers, such as SGD (Robbins and Monro, 1951) and Adam (Kingma, 2014), are first-order

methods due to computational efficiency and scalability.

1.2 The curse of dimensionality and saddle points

However, the process of optimisation becomes increasingly difficult as the number of parameters increase. More specifically, the landscape of our models becomes very complex as we scale with dimensionality. An example is shown in Fig. 1.1. In these high-dimensional spaces, we see a phenomenon where there are exponentially more frequent *saddle points* than the points that locally optimise our objective (Dauphin et al., 2014).

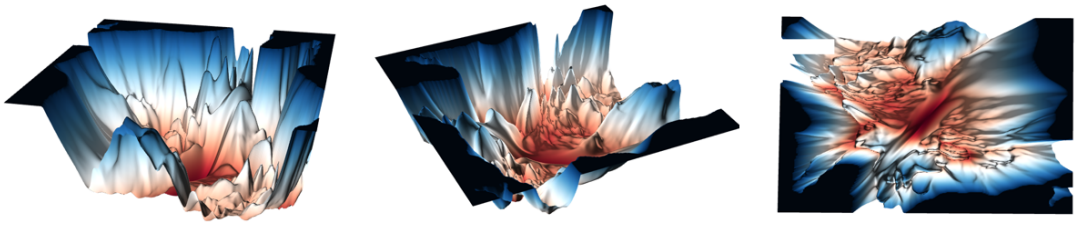


Figure 1.1: A visualisation of the loss landscape of a ResNet-56 model for image classification.

A saddle point is a flat region where along some dimensions, the second-order derivative of the objective is negative, but along others, it is positive. First-order methods slow down at saddle points since they perceive it as a flat region and do not have information about the curvature of the surrounding landscape. This hinders their convergence to a good solution.

Second-order methods, by incorporating this curvature information, can identify the nature of saddle points and navigate away from them more effectively. Despite this, it becomes computationally infeasible to process second-order information. This is because to obtain second-order information, we would require at least $O(N^2)$ space for N parameters. The computational cost of obtaining second-order information becomes prohibitive as N increases, which is the case given our deep learning setting. As a practical example, a ResNet-50 model has $N = 25 \times 10^6$ parameters. If we consider each parameter to be represented by 16-bits, we would require 1.25×10^{15} bytes, or 1.25 terabytes of memory to store the second-order information. This is in stark contrast to the 25 million bytes, or 25 megabytes, required for first-order information.

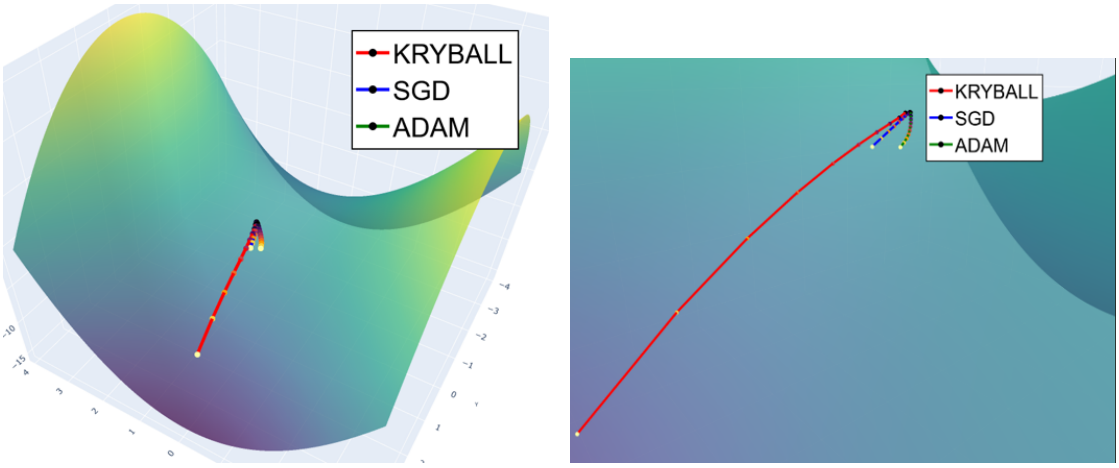
1.3 KryBall: the best of both worlds

The challenge is to therefore develop an optimisation algorithm that is computationally tractable and scalable, like first-order methods, but can also navigate complex loss landscapes, particularly by escaping saddle points quickly, like second-order methods. In this

thesis, we propose a new optimisation algorithm, KryBall, that combines the benefits of first-order and second-order methods. We summarise our approach below.

1. We use a *Krylov subspace* to approximate local curvature information as a low-dimensional subspace via efficient Hessian-vector products.
2. We analyse this subspace to understand the dominant geometric features of the local landscape.
3. We compute the *Saddle-Free Newton* direction within this subspace (Dauphin et al., 2014).
4. We combine this with first-order information such as the gradient and momentum in a *trust region* framework that uses a quadratic model approximation of the local landscape to get a combined, final search direction.

KryBall makes more informed steps than pure first-order methods, particularly in regions like saddle points, while remaining computationally tractable for deep learning. To motivate our method, we show an example in Fig. 1.2, where first-order methods such as Adam and SGD are hindered on a classic 2D horse saddle, but KryBall successfully escapes it quickly in fewer iterations.



(a) A classic 2D horse saddle with KryBall, SGD and Adam and their optimisation trajectories.

(b) A zoomed in version. KryBall successfully escapes the saddle point.

Figure 1.2: A toy example of our method, KryBall, successfully escaping the saddle point while first-order methods such as Adam and SGD are hindered.

1.4 Contributions

Our work focuses on the design and implementation of a novel optimisation algorithm for deep learning, KryBall. In this thesis, we present:

1. **The KryBall Optimisation Algorithm:** The proposal, design, and implementation of KryBall that combines Krylov subspace methods with the Saddle-Free-Newton and a trust-region framework. This enables efficient approximation of local curvature information and navigation of complex loss landscapes, while remaining computationally tractable for deep learning applications.

2. **Theoretical Analysis and Tools:** We provide analytical tools for assessing approximation quality through reconstruction error analysis and systematic investigation of loss landscape properties, including the effects of activation functions and problem conditioning on optimisation behaviour.

3. **Practical Implementation Framework:** The first N-dimensional subspace optimiser fully integrated with PyTorch as a drop-in replacement, alongside a modular experimental suite for optimiser research that integrates classic optimisation functions with modern deep learning workflows.

We evaluate KryBall extensively across diverse tasks including ill-conditioned function optimisation, binary classification, and image classification, with detailed sensitivity analysis and comparison against state-of-the-art optimisers.

1.5 Thesis outline

To present our contributions, this thesis is organised in the following manner:

- Chapter 2 provides a comprehensive background on optimisation. We formalise the optimisation problem and discuss the optimisation landscape from a mathematical perspective. This is followed by optimisation in deep learning, where first-order and second-order methods are explained. We then introduce methods for incorporating curvature information, such as Hessian-vector products, Krylov subspaces and trust region algorithms.

- Chapter 3 provides a comprehensive review of the relevant literature. This includes a survey of first-order and second-order optimisation algorithms used in deep learning. We compare these methods with our own.

- Chapter 4 presents the KryBall algorithm. We formally describe its components. This includes the Krylov subspace construction, the Saddle-Free-Newton computation, and integration with the trust-region framework.

- Chapter 5 presents our evaluations and analysis. We describe our experimental setup and benchmarking suite. This includes the datasets, model architectures, evaluation metrics, hyperparameter configurations, and comparison with state-of-the-art optimisers. This is followed by a sensitivity analysis. We then interpret our findings, analyse potential failure cases and unexpected behaviours, and address current limitations.

- Chapter 6 finishes the thesis. We summarise our contributions, discuss the implications of our work, and suggest future research.

This chapter has laid the outline for the thesis. We now proceed to Chapter 2 to establish the necessary mathematical context.

Chapter 2

Technical Background

In this chapter, we provide the relevant technical background required to understand our work. We start by introducing the optimisation problem in Sec. 2.1. This is continued by a mathematical formulation of the optimisation landscape in Sec. 2.2. We then discuss optimisation in deep learning in Sec. 2.3. We end this chapter with a discussion of computationally tractable methods for curvature exploitation in Sec. 2.4.

2.1 The Optimisation Problem

In this section, we formalise the optimisation problem. In the most fundamental case, we minimise an objective function f with respect to real-valued variables with no constraints. The formulation is

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

where

- $x \in \mathbb{R}^n$ is a real-valued vector with $n \geq 1$ components,
- $f : \mathbb{R}^n \to \mathbb{R}$ is a real-valued function,
- $f \in C^k$ s.t. $k \geq 1$, where C^k is the space of functions that are k times continuously differentiable.

We only have a local perspective of f, since it is usually expensive to evaluate. We know what f evaluates to on a limited set of points $x_0, x_1, \ldots, x_k$, in which we use this information to iteratively search for an optimal point x^* that minimises f. To do this, we must first understand the landscape of f and the scenarios that emerge when traversing it.

2.2 The Optimisation Landscape

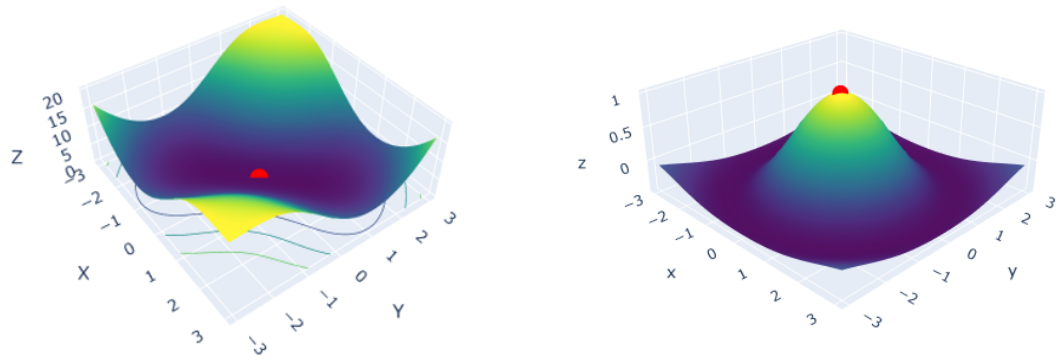
In this section, we introduce fundamental concepts that describe the landscape of f, which are needed to develop and analyse optimisation algorithms. We start by introducing the notion of critical points and how to recognise them in Sec. 2.2.1 and Sec. 2.2.2. We then discuss convexity in Sec. 2.2.3. This is followed by a discussion of ill-conditioning and non-smoothness in Sec. 2.2.4.

2.2.1 Critical Points

When exploring the landscape of an objective function, we are interested in identifying specific points of interest that characterise its features. These are called *critical points* (Goodfellow et al., 2016; Deisenroth et al., 2020). The most desirable of these are *global optima*, in which there are *global minimum* or *global maximum* (Nocedal and Wright, 2006). An example is provided in Fig. 2.1.

Definition 1 (Global Minimum)**.** A point x^* is a *global minimum* if $f(x^*) \leq f(x)$ for all x in the entire domain of f.

Definition 2 (Global Maximum)**.** A point x^* is a *global maximum* if $f(x^*) \geq f(x)$ for all x in the entire domain of f.



(a) A global minimum on $f(x, y) = (x - \sin(y))^2 + (y - \sin(x))^2$.

(b) A global maximum on $f(x, y) = \cos(x^2 + y^2)e^{-0.1(x^2+y^2)}$.

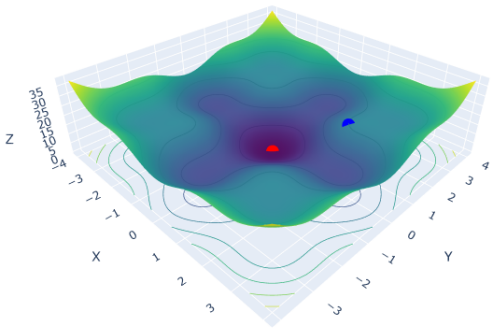
Figure 2.1: Examples of a global minimum and global maximum marked in red.

Finding such global optima is challenging, as we only have a limited set of information about f and are resource constrained when evaluating it. Thus, many optimisation algorithms aim to find *local optima*, which are points that are locally optimal. Similarly, there are *local minimum* or *local maximum* (Nocedal and Wright, 2006; Goodfellow et al., 2016). We provide examples in Fig. 2.2. We define these points with respect to a neighbourhood $\mathcal{N}$ of a point x.

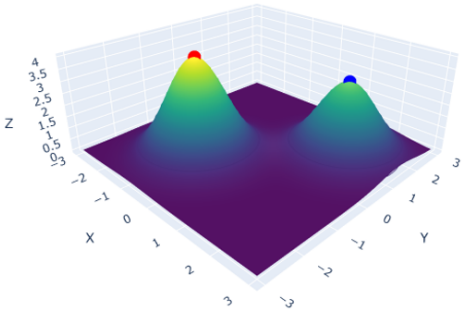
Definition 3 (Neighbourhood)**.** A neighbourhood $\mathcal{N}$ of a point $x \in \mathbb{R}^n$ is a set that contains an open ball centered at x.

Definition 4 (Local Minimum)**.** A point x^* is a *local minimum* if there exists a neighbourhood $\mathcal{N}$ around x^* such that $f(x^*) \leq f(x)$ for all $x \in \mathcal{N}$. It is a *strict local minimum* if instead $f(x^*) < f(x)$ for all $x \in \mathcal{N} \setminus \{x^*\}$.

Definition 5 (Local Maximum)**.** A point x^* is a *local maximum* if there exists a neighbourhood $\mathcal{N}$ around x^* such that $f(x^*) \geq f(x)$ for all $x \in \mathcal{N}$. It is a *strict local maximum* if instead $f(x^*) > f(x)$ for all $x \in \mathcal{N} \setminus \{x^*\}$.



(a) A local minimum on the egg crate function. $f(x, y) = (x^2 + y^2) + 5(\sin(x)^2 + \sin(y)^2)$.



(b) A local maximum on a multi-bump function. $f(x, y) = 3e^{-(x-1.5)^2-(y-1.5)^2} + 4e^{-(x+1)^2-(y+1)^2}$.

Figure 2.2: Examples of a local minimum and local maximum marked in blue. The global minimum and global maximum are marked in red for comparison.

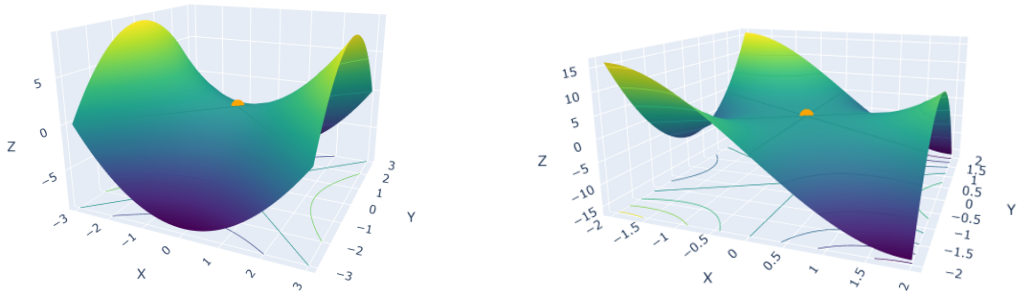
Beyond these, we have *saddle points*. These are points that are locally flat but are neither a local minimum nor a local maximum, as seen in Fig. 2.3. In any neighbourhood $\mathcal{N}$ around a saddle point, the function's value increases along some directions emanating from x^* and decreases along others. We formally define saddle points in Sec. 2.2.2.

2.2.2 Recognising Critical Points

For smooth and differentiable functions, we can recognise critical points using the first and second order information about f. Here, we introduce the necessary and sufficient conditions that we use to do this. We restrict our attention to the class of functions that are twice continuously differentiable, where $f \in C^2$ (Nocedal and Wright, 2006).

We start with a specific type of critical point, a *stationary point*. A key property of any stationary point x^* is that f is locally flat at x^*. This implies that its gradient—the vector of *first-order* partial derivatives, $\nabla f(x^*)$—must be zero (Goodfellow et al., 2016; Deisenroth et al., 2020).

Definition 6 (Stationary Point)**.** A point x^* is a *stationary point* if f is continuously



(a) A saddle point on the horse saddle function. $f(x, y) = x^2 - y^2$.

(b) A saddle point on the monkey saddle function. $f(x, y) = x^3 - 3xy^2$.

Figure 2.3: Examples of two functions containing saddle points marked in orange.

differentiable at x^* and its gradient is zero:

$$\nabla f(x^*) = 0. \tag{2.2}$$

Given we are optimising f, we want to find a stationary point that is a local minimum. The following definition formalises the *necessary first-order conditions* for a local minimum (Nocedal and Wright, 2006).

Definition 7 (Necessary First-Order Conditions)**.** If a point x^* is a local minimum, and f is continuously differentiable at x^*, then $\nabla f(x^*) = 0$ and x^* is a stationary point.

We note that all stationary points are critical points, but not all critical points are stationary points. For example, a function may have a critical point at a point where the gradient is undefined. A simple case is when $f(x) = \text{abs}(x)$. This has a critical point at $x = 0$ where the gradient is undefined. However, given we restrict our attention to twice continuously differentiable functions, we do not need to consider these cases.

All global optima, local optima, and saddle points are stationary points, but not all stationary points are optima. To distinguish between them, we examine the function's local curvature at x^*, which is captured by the *Hessian matrix*—an $n \times n$ symmetric matrix of *second-order* partial derivatives of f, denoted $\nabla^2 f(x)$ for a point x (Goodfellow et al., 2016; Deisenroth et al., 2020; Nocedal and Wright, 2006). We abbreviate the Hessian matrix for a function f at a point x as H for convenience. The curvature information captured by H can be summarised by its *eigenvalues*. We denote the i-th eigenvalue of H, and more generally any $n \times n$ symmetric matrix A, as λ_i, where $i \in [1, n]$. We use these eigenvalues to characterise two important properties, *positive semidefiniteness* and *negative semidefiniteness*.

Definition 8 (Positive Semidefinite Matrix)**.** An $n \times n$ symmetric matrix A is *positive semidefinite* if all its eigenvalues are non-negative, where $\lambda_i \geq 0$ for all $i \in [1, n]$.

Definition 9 (Negative Semidefinite Matrix)**.** An $n \times n$ symmetric matrix A is *negative semidefinite* if all its eigenvalues are non-positive, where $\lambda_i \leq 0$ for all $i \in [1, n]$.

These properties can now be used to formalise the *necessary second-order conditions* to classify stationary points (Nocedal and Wright, 2006). We write these for a local minimum as follows.

Definition 10 (Necessary Second-Order Conditions)**.** If a point x^* is a local minimum, and f is twice continuously differentiable, then:

- $\nabla f(x^*) = 0$
- H is positive semi-definite at x^*.

Similarly, the above definition can be extended to a local maximum where H is negative semi-definite.

A special case is when H is *indefinite*.

Definition 11 (Indefinite Matrix)**.** An $n \times n$ symmetric matrix A is *indefinite* if it is not positive semidefinite and not negative semidefinite. That is, there exists $\lambda_i > 0$ and $\lambda_j < 0$ for some $i, j \in [1, n]$ and $i \neq j$.

Here, the function curves upwards in some directions and downwards in others. This is a saddle point.

Definition 12 (Saddle Point)**.** A stationary point x^* is a *saddle point* if H is indefinite at x^*.

Now, we can classify between local minima, local maxima, and saddle points based on whether H is positive/negative semi-definite or indefinite. However, we can still fall short of distinguishing between local minima/maxima and strict local minima/maxima. For example, if H is positive semidefinite at x^*, x^* could be a local minimum or a flat region that is not a strict minimum (Nocedal and Wright, 2006). Similarly, if H is negative semidefinite at x^*, we could be in a local maximum or a flat region. To distinguish between this, we consider two further properties—*positive definiteness* and *negative definiteness*, which are stronger conditions than positive and negative semi-definiteness.

Definition 13 (Positive Definite Matrix)**.** An $n \times n$ symmetric matrix A is *positive definite* if all its eigenvalues are positive, where $\lambda_i > 0$ for all $i \in [1, n]$.

Definition 14 (Negative Definite Matrix)**.** An $n \times n$ symmetric matrix A is *negative definite* if all its eigenvalues are negative, where $\lambda_i < 0$ for all $i \in [1, n]$.

We can now guarantee something stricter about the nature of x^*, that it is a strict local minimum/maximum. This guarantees that we will optimise our objective without being trapped in a flat region. We write these as the *sufficient second-order conditions* for optimisation (Nocedal and Wright, 2006). We formalise this for a strict local minimum as follows.

Definition 15 (Sufficient Second-Order Conditions)**.** If f is twice continuously differentiable, and $\nabla f(x^*) = 0$, and H is positive definite at x^*, then x^* is a strict local minimum.

Similarly, we can write the sufficient second-order conditions for a strict local maximum when H is negative definite at x^*.

We note that the sufficient second-order conditions are not necessary. A point x^* may satisfy the necessary second-order conditions but fail to satisfy the sufficient second-order conditions (Nocedal and Wright, 2006). For example, the function $f(x) = x^4$ has a strict local minimum at $x^* = 0$, but H vanishes here and is thus not positive definite at x^*.

2.2.3 Convexity

The property of convexity simplifies the task of finding optima. A function f is *convex* if geometrically, the line segment connecting any two points on the function's graph lies on or above the graph itself (Boyd and Vandenberghe, 2004; Nocedal and Wright, 2006; Deisenroth et al., 2020). We provide an illustration in Fig. 2.4, and formally define this as follows.

Definition 16 (Convex Function)**.** A function $f : D \to \mathbb{R}$, where $D \subseteq \mathbb{R}^n$ is a convex set, is *convex* if for any two points x_1, x_2 in its domain, and any scalar $t \in [0, 1]$:

$$f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2). \tag{2.3}$$

This is a special case of *Jensen's inequality*. Equivalently, a function is convex if H is positive semidefinite for all x in the domain of f given f is twice continuously differentiable.

Convex functions are particularly appealing because they possess global properties that simplify the optimisation process. We call these properties the *global optimality conditions* for convex functions (Boyd and Vandenberghe, 2004; Nocedal and Wright, 2006).

Definition 17 (Global Optimality Conditions for Convex Functions)**.** If f is convex, then:

- Any local minimum x^* is a global minimum of f.
- Any stationary point x^* is a global minimum of f given f is continuously differentiable.

This means that if we find a stationary point of a convex function, we have found the overall best possible solution. Additionally, given that H is guaranteed to be positive semidefinite for twice continuously differentiable convex functions, certain optimisation algorithms can guarantee convergence to a global minimum regardless of the initialisation point.

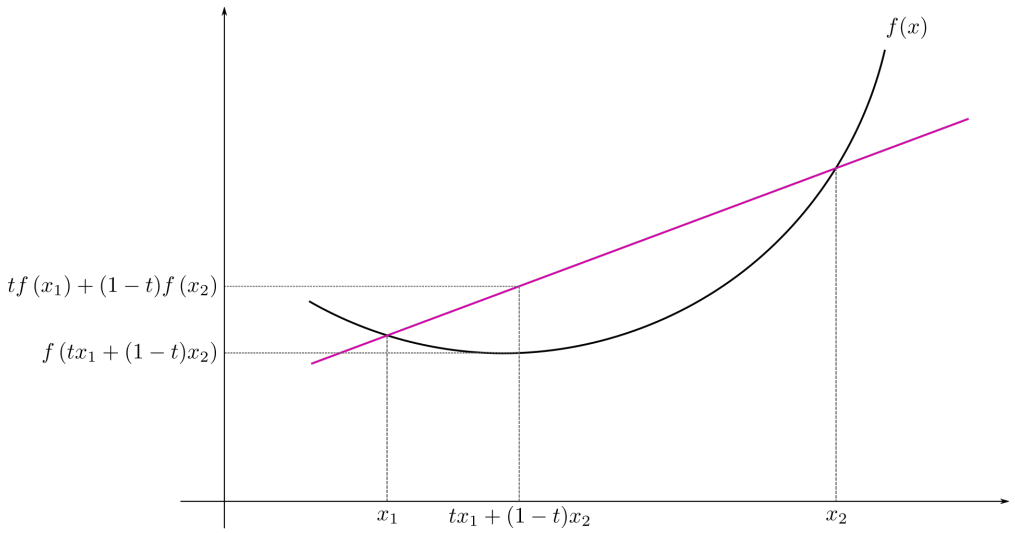


Figure 2.4: An illustration of a convex function. The line segment created by x_1 and x_2 clearly lies above the function.

We can further strengthen this by considering the property of *strict convexity*. A strictly convex function is one where the line segment connecting any two points on the function's graph lies strictly above the graph between those points (Boyd and Vandenberghe, 2004; Nocedal and Wright, 2006). We illustrate the difference between convex and strictly convex functions in Fig. 2.5. Formally, we define strictly convex functions as follows.

Definition 18 (Strictly Convex Function)**.** A function $f : \mathbb{R}^n \to \mathbb{R}$ is *strictly convex* if its domain is a convex set, and for any two distinct points x_1, x_2 such that $x_1 \neq x_2$ in its domain, and any scalar $t \in (0, 1)$:

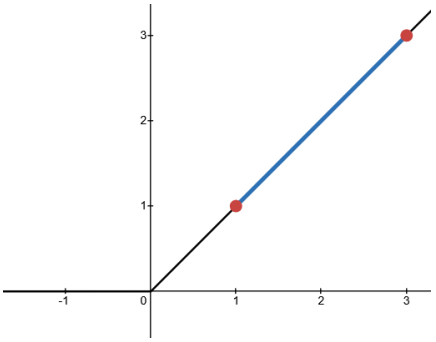
$$f(tx_1 + (1-t)x_2) < tf(x_1) + (1-t)f(x_2). \tag{2.4}$$

Equivalently, a function is strictly convex if H is positive definite for all x in the domain of f given f is twice continuously differentiable.

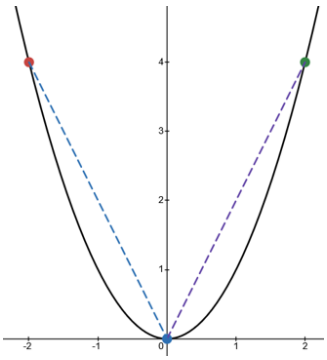
Strictly convex functions are a subset of convex functions. They inherit the same global optimality conditions, but with an additional *uniqueness* property that makes them incredibly easy to optimise (Nocedal and Wright, 2006).

Definition 19 (Uniqueness of Global Minimum)**.** If f is strictly convex, then there exists *at most one* local minimum of f. Consequently, if it exists, then it is the global minimum of f.

Thus, if we find any stationary point of a strictly convex function, we have found the global minimum (Nocedal and Wright, 2006). This is different from convex functions, where there could be multiple global minima. We provide an illustration of this in Fig. 2.6. This makes strictly convex functions extremely desirable for optimisation, since we have a guaranteed unique solution.



(a) The Rectified Linear Unit (ReLU) function, $f(x) = \max(0, x)$, is convex, but not strictly convex since we can pick two points where the line segment is not strictly above the function.



(b) The parabola function, $f(x) = x^2$, is strictly convex since for any two points, the line segment will always lie above the function.

Figure 2.5: The geometrical difference between a convex function and a strictly convex function.

2.2.4 Ill-Conditioning and Non-Smoothness

While convexity simplifies the optimisation process, two other characteristics, *ill-conditioning* and *non-smoothness*, significantly increase the difficulty instead. We now discuss these two characteristics and their influence the optimisation landscape.

Ill-Conditioning

An optimisation problem is *ill-conditioned* if the objective function f is highly sensitive to small changes to its parameters in the vicinity of a solution x^*. Geometrically, this corresponds to a landscape that is either elongated with narrow valleys, or steep ridges with large dropoffs, as illustrated in Fig. 2.7. To motivate the problem of ill-conditioning, we consider the deterministic Rosenbrock function

$$f(x, y) = (1 - x)^2 + 100(y - x^2)^2. \tag{2.5}$$

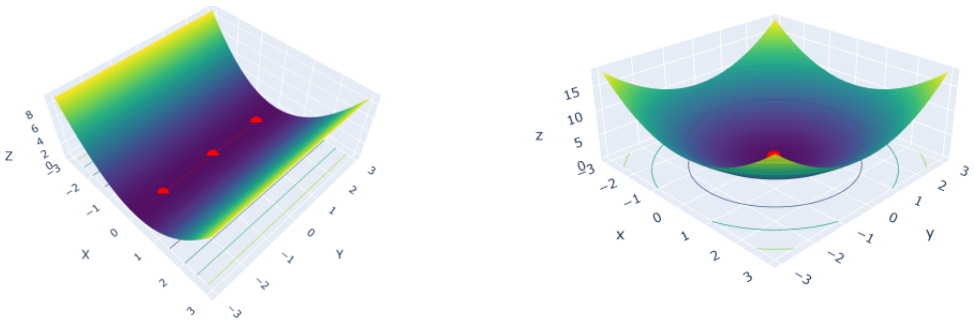
The global minimum for this function is at $(x, y) = (1, 1)$. Suppose we introduce a small perturbation $\epsilon_1 = 0.01$ to x and $\epsilon_2 = 0.02$ to y around the minimum. Our function evaluates to

$$f(x + \epsilon_1, y + \epsilon_2) = f(1 + 0.01, 1 + 0.02) \approx 1e - 4. \tag{2.6}$$

If we modify the perturbations to $\epsilon_1 = 0.02$ and $\epsilon_2 = -0.01$ instead, our function value now becomes

$$f(x + \epsilon_1, y + \epsilon_2) = f(1 + 0.02, 1 - 0.01) \approx 0.25. \tag{2.7}$$

We see a significant change in how our function behaves, where it is now 2500 times larger than before with only small changes applied to its input. This illustrates the problem



(a) Multiple global minima on the convex function $f(x, y) = x^2$.

(b) A unique global minimum on the strictly convex function $f(x, y) = x^2 + y^2$.

Figure 2.6: The difference between a convex function with multiple global minima and a strictly convex function with a unique global minimum.

of ill-conditioning, which makes it difficult for optimisation algorithms to converge to a solution. We describe the ill-conditioning of a problem in terms of the *condition number* (Belsley et al., 2005; Boyd and Vandenberghe, 2004).

Definition 20 (Condition Number)**.** For any non-singular square matrix $A \in \mathbb{R}^{n \times n}$, its *condition number* with respect to a given matrix norm $||\cdot||$ is defined as:

$$\kappa(A) = ||A|| \cdot ||A^{-1}||, \tag{2.8}$$

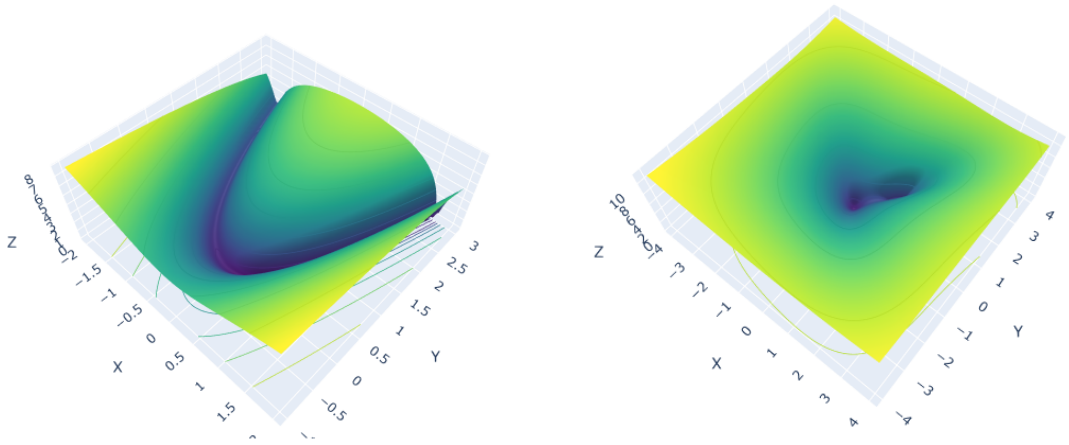
In the context of optimisation, if f is twice continuously differentiable at x and H is positive definite at x, then the condition number is the ratio of the largest eigenvalue $\lambda_{\max}$ to the smallest eigenvalue $\lambda_{\min}$ of H under the spectral norm $||\cdot||_2$.

$$\kappa(H) = ||H||_2||H^{-1}||_2 = \lambda_{\max}(H)\lambda_{\max}(H^{-1}) = \lambda_{\max}(H) \cdot \frac{1}{\lambda_{\min}(H)} = \frac{\lambda_{\max}(H)}{\lambda_{\min}(H)}. \tag{2.9}$$

In the Rosenbrock example, the condition number is 2508 at the minimum, which explains the behaviour we observed. Ill-conditioning poses significant challenges for many optimisation algorithms, especially those relying on gradient information. In these landscapes, finding solutions that are robust to perturbations is difficult. Algorithms may get stuck and make excessively small steps to counteract the ill-conditioning, which slows convergence, or they become unstable due to taking overly aggressive steps. Efficient navigation of these landscapes require algorithms to be *scale invariant* (Nocedal and Wright, 2006). We discuss these algorithms in more detail in Sec. 2.3.

Non-Smooth Problems

So far, we have talked about optimisation problems where the objective function f is smooth and usually twice continuously differentiable. However, many optimisation



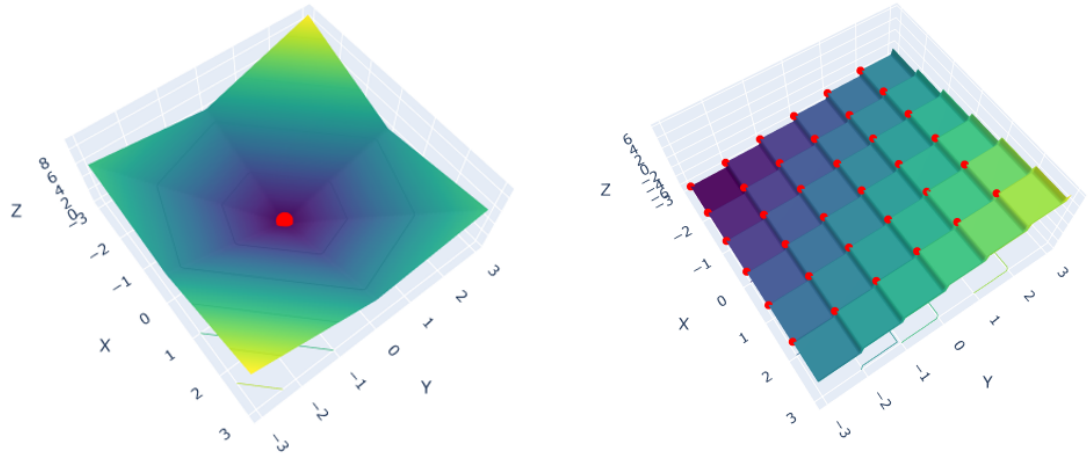
(a) The deterministic Rosenbrock function $f(x, y) = (1 - x)^2 + 100(y - x^2)^2$

(b) A simplified 2D analogue of the wood function $f(x, y) = 100 * (y - x^2)^2 + (1 - x)^2 + 90 * (y^2 - x)^2 + (1 - y)^2$

Figure 2.7: The optimisation landscape of two ill-conditioned functions. The Rosenbrock function (left) shows the narrow valley towards the minimum, whereas the 2D wood function (right) shows a steep dropoff as we approach the minimum.

problems that we encounter involve *non-smooth functions*. These are functions that possess points at which the function or its derivatives are not well-defined. We provide an example of such functions which have points of non-differentiability in Fig. 2.8.

Non-smoothness is common in machine learning, with terms such as ReLU and L1 regularisation being used in many problem settings (Goodfellow et al., 2016; Deisenroth et al., 2020). At points of non-differentiability, classical optimisation concepts that we have been discussing break down. The optimisation of non-smooth functions requires alternative theoretical frameworks and algorithms. While this is out of scope for this thesis, we briefly provide an overview of such methods in Chapter 3. In specific instances, we can reformulate the non-smooth problem into a smooth approximation. For example, the ReLU function has an equivalent smooth approximation given by the Softplus function, $f(x) = \log(1 + e^x)$ (Goodfellow et al., 2016) (Paszke et al., 2017). While non-smooth optimisation is out of scope for this thesis, we provide an overview of such methods in Chapter 3.



(a) The function $f(x, y) = \text{abs}(x) + \text{abs}(y)$ has an undefined gradient at (0,0).

(b) The step function $f(x, y) = \lfloor x \rfloor + \lfloor y \rfloor$ has undefined gradients at all integer points.

Figure 2.8: The optimisation landscape of two non-smooth functions, with points of non-differentiability highlighted in red.

2.3 Optimisation in Deep Learning

Deep learning introduces a new set of challenges to the optimisation landscape. In deep learning, we usually have a set of *input data* $x \in \mathbb{R}^n$ and a *target output* $y \in \mathbb{R}^m$. Our goal is to learn a mapping from x to y. This is represented by a *model*, which is a complex, highly parameterised function that is usually non-convex and can be thought of as a *universal function approximator* (Hornik et al., 1989). Through exposure to many examples in a *training set*, the model makes *predictions* $\hat{y}$ for inputs x, and is then evaluated on a *test set* with new, unseen inputs. Ideally, we would like our model to make accurate predictions on the test set, as this is a good indicator of generalisation performance. We optimise the model's *parameters*, $\theta \in \mathbb{R}^N$, by minimising a real-valued *loss function* $L : \mathbb{R}^N \to \mathbb{R}$ that measures model's performance. Typically, this is written as an average over the training set, such as

$$L(\theta) = \mathbb{E}_{(x,y)\sim\hat{p}_{data}} \mathcal{L}(f(x;\theta), y) = \mathbb{E}_{(x,y)\sim\hat{p}_{data}} \ell(\hat{y}, y), \tag{2.10}$$

where $\hat{y} = f(x;\theta)$, $\mathcal{L}$ is the per-example loss (such as mean-squared error or cross-entropy), and $\hat{p}_{data}$ is the empirical distribution of our training set. In practice, we usually sample a set of examples X_i, Y_i from the training set to compute $\hat{y}_i$ and L (Goodfellow et al., 2016; Paszke et al., 2017). We provide a general algorithm for deep learning optimisation in Algorithm 1.

Algorithm 1: General Model Optimisation

Input: Training data $\{(x_i, y_i)\}_{i=1}^n$, model $f(x;\theta)$ with initial parameters θ_0, loss function $\mathcal{L}$, number of iterations T

Output: Optimised parameters θ

$\theta \leftarrow \theta_0$
for $t = 1, 2, \ldots, T$ **do**
  Sample a set of examples (X_t, Y_t) from training data
  Compute loss $L_t(\theta) = \mathcal{L}(f(X_t;\theta), Y_t)$
  Find an update direction $\Delta\theta$
  $\theta \leftarrow \theta + \Delta\theta$
return θ

We limit our discussion of $\mathcal{L}$ to the *supervised* learning case, where we have a fixed input x and a corresponding target output y and the inputs to $\mathcal{L}$ are $\hat{y}$ and y. It is easy to extend this development to the *regularised* case, for example by including θ as an argument, or the *unsupervised* case by removing y.

In this section, we discuss the challenges of optimisation in deep learning. We start by discussing the behaviour of high-dimensional landscapes in Sec. 2.3.1. This is followed by a discussion of first-order methods in Sec. 2.3.2 and second-order methods in Sec. 2.3.3.

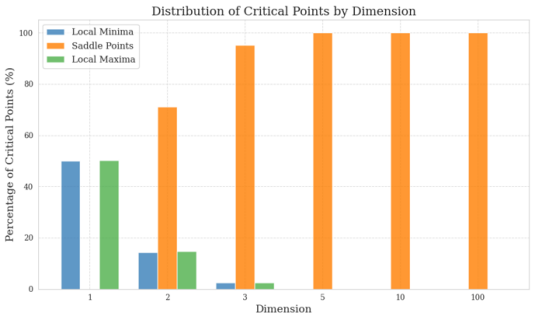
2.3.1 Challenges in High-Dimensional Landscapes

High-dimensional landscapes are complex and difficult to navigate. Recall that we saw an example of this with a highly parameterised high-dimensional ResNet model in Chapter 1. The geometric intuition derived from low-dimensional spaces is usually not applicable. We observe two key properties in high-dimensions.

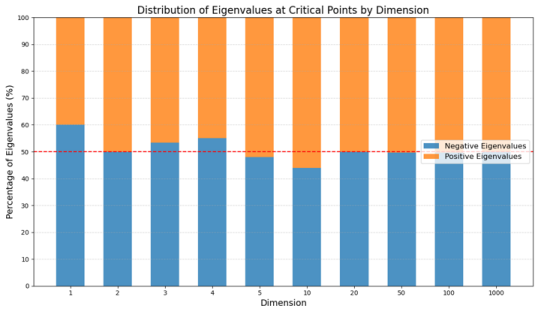
- *Proliferation of saddle points*: Saddle points are *exponentially* more likely than local minima as dimensionality N increases (Dauphin et al., 2014).
- *Local minima are close to global minimum*: Local minima in high dimensions are likely to have values very close to the global minimum (Dauphin et al., 2014; Choromanska et al., 2015).

We can understand the first property by analysing H in the context of our loss function, where we now have that $H = \nabla^2 L(\theta)$. As established in Sec. 2.2.2, a local minimum requires all H to be positive semidefinite, in which all eigenvalues are greater than or equal to zero. A saddle point however possesses both positive and negative eigenvalues. We note that for large random Gaussian matrices, the eigenvalue distribution follows *Wigner's semicircle law*, which states that as N increases, we observe the following (Wigner, 1958; Dauphin et al., 2014).

- An eigenvalue λ_i has an equal probability of $\frac{1}{2}$ to be positive or negative.
- Each eigenvalue's probability is approximately independent of others.



(a) The distribution of local minima, local maxima, and critical points of random Gaussian matrices of dimensions N.



(b) The distribution of the eigenvalues at critical points for various random Gaussian matrices of dimensions N.

Figure 2.9: The distribution of critical points and their eigenvalues for random Gaussian matrices of dimensions N.

Intuitively, we can consider the probability of each eigenvalue as akin to an independent fair coin toss. The probability of obtaining N non-negative eigenvalues diminishes exponentially with increasing N. The same goes for obtaining N non-positive eigenvalues. Consequently, obtaining N eigenvalues with mixed signs are far more probable, which explains the proliferation of saddle points, as seen in Fig. 2.9(a). As N increases, the eigenvalues follow the distribution of the semicircle law more, and we get approximately equal numbers of positive and negative eigenvalues. We see this in Fig. 2.9(b). While we have this case for large random Gaussian matrices, we note that this applies to the deep learning setting as well. Experimental evidence shows that the landscapes of deep learning models display many more saddle points than local minima (Dauphin et al., 2014). We provide a more detailed analysis of the deep learning optimisation landscape in Chapter 5.

The second property follows as a direct consequence of the first. Suppose we consider a local minima with a function value that is substantially higher than the global minimum. Given high dimensionality, it is very probable that there exists at least one eigenvalue that is negative which we can take to minimise the objective in some immediate neighbourhood. Such a point would then be a saddle point, offering a local escape direction. Given that local minima are so rare, we observe that they can only take a range of loss values, in which these are close to the global minimum (Choromanska et al., 2015; Dauphin et al., 2014; Goodfellow et al., 2016). This suggests that the challenge in deep learning optimisation is less about getting trapped in poor local minima and more about efficiently navigating the numerous saddle points that dominate the landscape (Goodfellow et al., 2016).

The dominance of saddle points changes the optimisation landscape and impedes the progress of optimisation algorithms. First-order methods that rely on gradient information, such as *gradient descent*, experience slowed convergence in these regions. Second-

order methods such as *Newton's method* also face problems, as in some cases they are actually attracted to saddle points, even though they have the benefit of having local curvature information and are scale invariant. We cover these in the next two sections Sec. 2.3.2 and Sec. 2.3.3.

2.3.2 First-Order Methods

Most optimisation algorithms in deep learning are *first-order methods*. First-order optimisation involves using the *gradient of the loss function* to iteratively update the model parameters θ (Goodfellow et al., 2016; Deisenroth et al., 2020). In the deep learning setting, this is defined as the vector of partial derivatives of L with respect to θ (Goodfellow et al., 2016).

Definition 21 (Gradient of the Loss Function)**.** The gradient of the loss function L with respect to the parameters θ is defined as:

$$\nabla L(\theta) = \left(\frac{\partial L}{\partial \theta_1}, \frac{\partial L}{\partial \theta_2}, \ldots, \frac{\partial L}{\partial \theta_N} \right). \tag{2.11}$$

We abbreviate this as g for brevity. The gradient vector points in the direction of steepest ascent of the loss function, and is thus the direction of most rapid change. We perform an optimisation step by moving in the *negative gradient* direction, which is the direction of *steepest descent* (Goodfellow et al., 2016; Deisenroth et al., 2020; Nocedal and Wright, 2006). The general update rule is given by:

$$\Delta\theta = -\alpha_t g_t, \tag{2.12}$$

$$\theta_{t+1} = \theta_t + \Delta\theta, \tag{2.13}$$

where θ_t denotes the model parameters at iteration t, g_t is the gradient evaluated at that point, and $\alpha_t > 0$ is the *learning rate*. The learning rate controls the step size of the steepest descent step.

First-order methods are computationally efficient and scalable. This is because the update can be performed with $O(N)$ space for a model with N parameters for a single sample of data (Goodfellow et al., 2016; Deisenroth et al., 2020). This makes them extremely appealing for deep learning, where models have millions or even billions of parameters and we have constraints on computational resources (Goodfellow et al., 2016). These methods are also highly parallelisable (Paszke et al., 2017). Modern GPU hardware and distributed computing with efficient low-level kernels enable gradient computations to be executed across multiple processing units, making them even more appealing (Paszke et al., 2017).

However, while simple and effective in many scenarios, the reliance of first-order methods on the gradient becomes problematic in high-dimensional landscapes that are characterised by a proliferation of saddle points, as briefly mentioned in Sec. 2.3.1. In these

regions of low curvature, the gradient magnitude is small. This leads to small, incremental updates to our parameters that result in slowed convergence. Conversely, in ill-conditioned landscapes as discussed in Sec. 2.2.4, the gradient can be noisy and unstable. This is particularly the case for some ill-conditioned problems, such as the *Rahimi-Recht function*, where the landscape becomes increasingly steep and majority of the eigenvalues tend towards infinity as we approach the minimum (Rahimi and Recht, 2017). This results in slowed convergence and suboptimal solutions being found, and we see this in the next section in Fig. 2.11. In the worst case, the possibility of divergence is also present, which we will see later on in our evaluation in Chapter 5.

Numerous variations of first-order methods, despite these challenges, have made them successful in deep learning and cemented them as the current state-of-the-art. These include the use of *stochastic gradients*, *momentum*, and *adaptive* methods (Robbins and Monro, 1951; Kingma, 2014). We provide a comprehensive overview of these optimisation algorithms in Chapter 3.

2.3.3 Second-Order Methods

Second-order methods use local curvature information to traverse the optimisation landscape. This is captured by the *Hessian of the loss function*, which is in contrast to first-order methods that solely rely on the gradient. We introduced the Hessian generally in Sec. 2.2.2. Here, we define it in the context of deep learning, where it is the matrix of second-order partial derivatives of L with respect to the parameters θ (Goodfellow et al., 2016).

Definition 22 (Hessian of the Loss Function)**.** The Hessian of the loss function L with respect to the parameters θ is the $N \times N$ matrix of second-order partial derivatives:

$$\nabla^2 L(\theta) = \left(\frac{\partial^2 L}{\partial \theta_i \partial \theta_j} \right)_{i,j=1}^{N} = \begin{pmatrix} \frac{\partial^2 L}{\partial \theta_1^2} & \frac{\partial^2 L}{\partial \theta_1 \partial \theta_2} & \cdots & \frac{\partial^2 L}{\partial \theta_1 \partial \theta_N} \\ \frac{\partial^2 L}{\partial \theta_2 \partial \theta_1} & \frac{\partial^2 L}{\partial \theta_2^2} & \cdots & \frac{\partial^2 L}{\partial \theta_2 \partial \theta_N} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 L}{\partial \theta_N \partial \theta_1} & \frac{\partial^2 L}{\partial \theta_N \partial \theta_2} & \cdots & \frac{\partial^2 L}{\partial \theta_N^2} \end{pmatrix}. \tag{2.14}$$

We denote this as H for brevity. Note that from here onwards, H specifically refers to $\nabla^2 L(\theta)$ rather than the general Hessian $\nabla^2 f(x)$ of an objective function previously defined in Sec. 2.2.2.

The foundation of many second-order methods is the approximation of the loss function using a *quadratic model* (Nocedal and Wright, 2006). This is derived by taking a second-order *Taylor expansion* of the loss function (Nocedal and Wright, 2006). We define a Taylor expansion in the one dimensional case as follows.

Definition 23 (Taylor Expansion)**.** Given a function $f : \mathbb{R} \to \mathbb{R}$ where $f \in C^p$, that is f is p times continuously differentiable, the Taylor expansion of f about $x = a$ is given

by:

$$f(x) \approx \sum_{k=0}^{p} \frac{f^{(k)}(a)}{k!}(x-a)^k \tag{2.15}$$

$$= f(a) + \frac{\nabla f(a)}{1!}(x-a) + \frac{\nabla^2 f(a)}{2!}(x-a)^2 + \cdots + \frac{\nabla^p f(a)}{p!}(x-a)^p. \tag{2.16}$$

Intuitively, the Taylor expansion of a function $f \in C^p$ at a point a is a degree-p polynomial approximation of $f(x)$ for x in a neighbourhood of a. As x approaches a, the approximation becomes more accurate. We call this a pth-order Taylor expansion of f at a. An illustration is provided in Fig. 2.10.

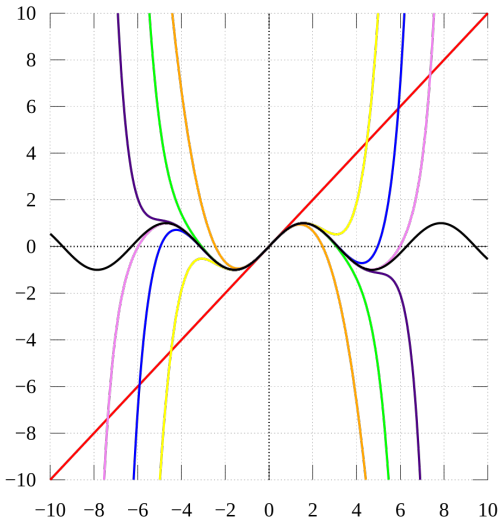


Figure 2.10: Given $f(x) = \sin(x)$, as the degree of the Taylor expansion increases, it better approximates the function around our chosen point $x = 0$. Shown here is the Taylor expansion of f where $p \in [1, 3, 5, 7, 9, 11, 13]$ at $x = 0$.

Second-order methods, as said in their name, use a second-order Taylor expansion (i.e., $p = 2$). In deep learning, we want to minimise the loss function L given our current parameters θ. This means we want to find a step $\Delta\theta$ such that $L(\theta + \Delta\theta) < L(\theta)$. We do this by taking a second-order Taylor expansion of L at θ to approximate $L(\theta + \Delta\theta)$ as follows.

$$L(\theta + \Delta\theta) \approx L(\theta) + \nabla L(\theta)^T(\theta + \Delta\theta - \theta) + \frac{1}{2}(\theta + \Delta\theta - \theta)^T \nabla^2 L(\theta)(\theta + \Delta\theta - \theta) \tag{2.17}$$

$$= L(\theta) + \nabla L(\theta)^T \Delta\theta + \frac{1}{2}\Delta\theta^T \nabla^2 L(\theta)\Delta\theta \tag{2.18}$$

$$= L(\theta) + g^T \Delta\theta + \frac{1}{2}\Delta\theta^T H \Delta\theta. \tag{2.19}$$

This gives our quadratic model m in terms of $\Delta\theta$. We refer to this as $m(\Delta\theta)$ in this case as $\Delta\theta$ is our step here that we want to find, though we note that this can be generalised to any step p. If H is positive definite, we then have a strictly convex quadratic function that we easily find the minimum of and thus solve for $\Delta\theta$. We do this by taking the derivative of the quadratic model with respect to $\Delta\theta$ and setting it to zero (Nocedal and Wright, 2006).

$$0 = \nabla_{\Delta\theta}\left(L(\theta) + g^T\Delta\theta + \frac{1}{2}\Delta\theta^T H\Delta\theta\right) \tag{2.20}$$

$$= g + H\Delta\theta. \tag{2.21}$$

Now, we solve for $\Delta\theta$, in which we get that:

$$\Delta\theta = -H^{-1}g. \tag{2.22}$$

This is known as the *Newton step* (Nocedal and Wright, 2006; Boyd and Vandenberghe, 2004; Deisenroth et al., 2020). In the context of an optimisation algorithm, we iteratively solve for $\Delta\theta_t$ by considering our model $m_t(\Delta\theta_t)$ at an iteration t with our current parameters θ_t. This gives us *Newton's method* (Nocedal and Wright, 2006; Boyd and Vandenberghe, 2004; Deisenroth et al., 2020).

$$\text{Solve } m_t(\Delta\theta_t) \text{ for } \Delta\theta_t \tag{2.23}$$

$$\Delta\theta_t = -H_t^{-1}g_t, \tag{2.24}$$

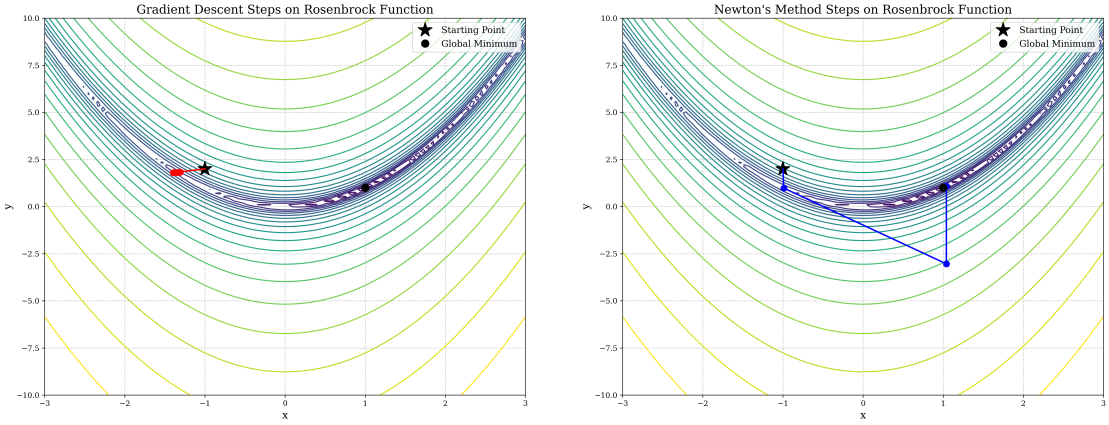
$$\theta_{t+1} = \theta_t + \Delta\theta_t. \tag{2.25}$$

where H_t is the Hessian evaluated at θ_t and g_t is as defined in Eq. (2.13).

The quadratic model and Newton's method are central to second-order optimisation methods. In particular, they offer a few key advantages over first-order methods.

- *Local Curvature*: The quadratic model incorporates a more sophisticated understanding of the local curvature of the optimisation landscape compared to the gradient used by first-order methods, allowing for more informed steps.
- *Analytical Solution*: The quadratic model can be minimised analytically with respect to $\Delta\theta$. This gives a candidate for the optimal step.
- *Scale Invariance*: The Newton step is *scale-invariant*. This means its performance is not adversely affected by linear rescaling of the parameters θ (Nocedal and Wright, 2006). This is because it rescales the problem space through the use of H^{-1}, which naturally adapts to the local curvature of the objective function. We show an example of this with the Rosenbrock function in Fig. 2.11. This property makes Newton's method very effective for ill-conditioned problems, as mentioned before in Sec. 2.2.4. It can navigate elongated valleys or surfaces with disparate scaling across more effectively than first-order methods.

- *Quadratic Convergence*: The Newton step exhibits *quadratic convergence* (Nocedal and Wright, 2006; Boyd and Vandenberghe, 2004). Given that H is positive definite, this means we can converge quadratically to a local minimum if we are sufficiently close to it. This is much faster than first-order methods which have linear or sub-linear convergence rates (Nocedal and Wright, 2006; Boyd and Vandenberghe, 2004; Deisenroth et al., 2020).



(a) Trajectory of steepest descent on the Rosenbrock function with $\alpha = 0.01$.

(b) Trajectory of Newton's method on the Rosenbrock function.

Figure 2.11: Steepest descent and Newton's method on the Rosenbrock function for 10 steps. We see that the optimising using steepest descent direction makes slow progress, while Newton's method converges quickly in the first few steps.

However, despite these advantages, second-order methods face challenges in deep learning. This is primarily due to their computation cost. The first challenge is the storing the Hessian matrix H. As H is an $N \times N$ matrix, it requires $O(N^2)$ space to store (Goodfellow et al., 2016). This is infeasible to compute with large-scale models, which are very common in deep learning. The second challenge is the computation of H^{-1}, required for methods such as Newton's method. This requires $O(N^3)$ computation time. Additionally matrix inversion is difficult to parallelise and thus is not as friendly on modern GPU hardware (Goodfellow et al., 2016). This is significant since most of modern deep learning relies heavily on GPU parallelisation and distributed computing (Paszke et al., 2017). As such, the difficulty of pure Newton-style methods makes it impractical for large-scale deep learning. We note that this does not mean methods that rely on the quadratic approximation or inverse Hessian are useless, as there exist tractable methods that approximate them to capture local curvature. We discuss these in Section 2.4 and Chapter 3.

Beyond computational demands and their constraints, there is one significant problem with the Newton method, which is that it is *attracted* to saddle points (Dauphin et al., 2014). Consider the classic horse saddle $f(x, y) = x^2 - y^2$ as introduced in Fig. 2.3(a).

To understand the behaviour of Newton's method, we can examine how it acts along each of the two principal directions.

- Along the x-axis, we have the component function $g(x) = x^2$.
- Along the y-axis, we have the component function $h(y) = -y^2$.

Suppose we minimise $g(x)$ with Newton's method. We get that the gradient is $\nabla g(x) = 2x$ and the Hessian is $H_g = \nabla^2 g(x) = 2$ in the x-direction given we take the derivatives with respect to x. Then, the inverse Hessian is $\frac{1}{2}$. For a point x_0, we compute the Newton step to get the next iterate x_1 as follows:

$$x_1 = x_0 + \Delta x = x_0 - H_g^{-1} \nabla g(x_0) = x_0 - \frac{1}{2} \cdot 2x_0 = 0. \tag{2.26}$$

This correctly moves us to the local minimum of $g(x)$ at $x = 0$, and we only take one step to do this which demonstrates the fast convergence of Newton's method, especially given that $g(x)$ is convex.

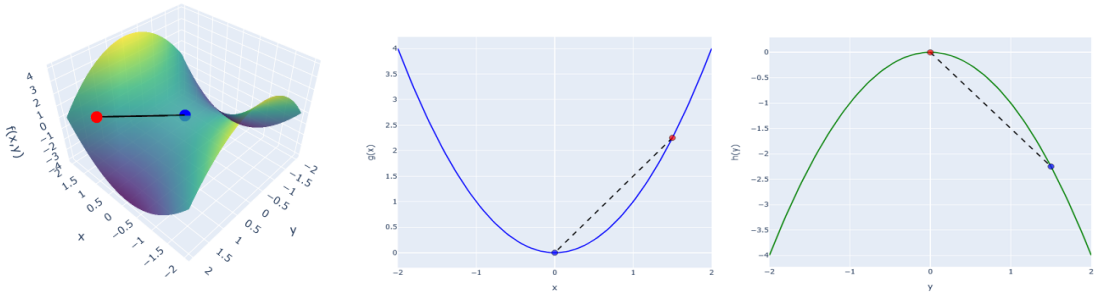
Now, we consider the behaviour along the y-axis for $h(y)$. The gradient is $\nabla h(y) = -2y$ and the Hessian is $H_h = \nabla^2 h(y) = -2$ in the y-direction given we take the derivatives with respect to y. The inverse Hessian is now $-\frac{1}{2}$. For a point y_0, if we compute the Newton step to get y_1, we observe the following.

$$y_1 = y_0 + \Delta y = y_0 - H_h^{-1} \nabla h(y_0) = y_0 - \left(-\frac{1}{2} \cdot -2y_0 \right) = y_0 - y_0 = 0. \tag{2.27}$$

In this case, the Newton step has taken us towards the *local maximum* of $h(y)$ at $y = 0$. We see that the Newton step actually goes in the wrong direction and does not minimise the function, which would involve taking the negative gradient direction as is done with gradient descent. As a result, our next set of iterates (x_1, y_1) are $(0, 0)$, the exact point that is a saddle for our original function f. If we then try to take a step from here, we will still remain at $(0, 0)$ as our gradient will be 0. Thus, we get stuck at the saddle point. We illustrate this in Fig. 2.12.

This is because Newton's method is attracted to saddle points when H is indefinite (Dauphin et al., 2014). In this example, the eigenvalues of H correspond to being $\lambda_1 = 2$ and $\lambda_2 = -2$. Intuitively, this means that along positive eigenvalue directions, the Newton step correctly moves towards minima, but along negative eigenvalue directions, it *moves away from the minima*. When H is instead negative definite, the Newton step will move towards the local maximum instead as our function is locally concave at the current point. This is a significant concern given the proliferation of saddle points we discussed in Sec. 2.3.1.

While it may seem like Newton's method and the quadratic model approximation are not a good fit for deep learning, multiple variants have extended these core ideas and have found success. These include *damped Newton* methods, *Quasi-Newton* methods, and *diagonal Hessian* approximations (Nocedal and Wright, 2006; Liu et al., 2023). We discuss these further in the next section and provide a detailed review in Chapter 3.



(a) The Newton step on the 2D horse saddle function.

(b) The Newton step on the x-axis component.

(c) The Newton step on the y-axis component.

Figure 2.12: Newton's method on the classic 2D horse saddle function. Given a starting point $(1.5, 1.5)$ in red, the Newton step is attracted towards the saddle points at $(0, 0)$ in blue. We see that for the x-axis component, it correctly minimises the component function $g(x)$, but for the y-axis component, it moves away from the minima instead.

2.4 Methods for Tractable Curvature Exploitation

In this section, we discuss strategies that can incorporate curvature information while being computationally tractable. We start by discussing Hessian-vector products in Sec. 2.4.1. This is followed by an introduction to Krylov subspaces in Sec. 2.4.2 and then trust-region methods in Sec. 2.4.3.

2.4.1 Hessian-vector Products

We saw previously in Sec. 2.3.3 that storing the Hessian matrix H or computing its inverse H^{-1} is infeasible in deep learning due to computational constraints. However, many optimisation algorithms incorporate curvature information without explicitly forming the Hessian. One way to do this is through *Hessian-vector* products (Pearlmutter, 1994; Dagréou et al., 2024).

Definition 24 (Hessian-vector Product)**.** Given a loss function $L(\theta)$ with parameters $\theta \in \mathbb{R}^N$, the Hessian-vector product of $H = \nabla^2 L(\theta)$ with a vector $v \in \mathbb{R}^N$ is defined as:

$$Hv = \nabla^2 L(\theta) \cdot v, \tag{2.28}$$

where $Hv \in R^N$.

We abbreviate Hessian-vector products as HVPs from here onwards. Intuitively, the HVP captures how g changes when we move in a specific direction v. It captures the local curvature of the optimisation landscape in this direction. The key insight is that we can compute HVPs efficiently without explicitly forming H by considering the directional derivative of g with respect to v. Instead of requiring $O(N^2)$ time and space, HVPs can

be computed with only two gradient evaluations (Pearlmutter, 1994; Dagréou et al., 2024; Martens et al., 2010). This is known as Pearlmutter's trick.

Definition 25 (Pearlmutter's Trick for Efficient HVPs)**.** The HVP is the directional derivative of g in the direction v,

$$Hv = \lim_{\epsilon \to 0} \frac{1}{\epsilon}[\nabla L(\theta + \epsilon v) - \nabla L(\theta)] = \nabla[\langle \nabla L(\cdot), v \rangle](\theta), \tag{2.29}$$

where $\langle \cdot, \cdot \rangle$ denotes the standard Euclidean inner product. See that this follows from the standard definition of the directional derivative for differentiable functions.

Based on this identity, HVPs can be computed efficiently with automatic differentiation systems using only $O(N)$ computation and storage (Paszke et al., 2017). This makes them comparable to standard gradient calculations. HVPs are valuable for deep learning optimisation as they provide access to curvature information while remaining computationally tractable. Modern deep learning frameworks implement HVPs efficiently using automatic differentiation modes such as *forward-mode* and *reverse-mode* (Paszke et al., 2017; Henriques et al., 2019). This allows for easy integration with deep learning tasks.

HVPs form the foundation of many iterative algorithms. We discuss these more generally in Chapter 3, and in the next section introduce the notion of Krylov subspaces, which leverages matrix-vector products like HVPs to allow for efficient optimisation.

2.4.2 Krylov Subspaces

In Sec. 2.3.1, we discussed the challenges of the high-dimensional landscapes present in deep learning. In Sec. 2.3.3, we followed this by discussing the problem of evaluating H or H^{-1}. We now introduce *Krylov subspaces*, which are a powerful iterative method that addresses these challenges (Nocedal and Wright, 2006; Gutknecht, 2007).

Consider a general linear system $Ax = b$ where $A \in R^{N \times N}$, and $x, b \in R^N$. In a high-dimensional system where $N \gg 1$, solving this is difficult and often intractable. An example of this is the Newton step, $\Delta\theta = H^{-1}g$, where we outlined this as a core limitation in the previous section. However, one way to approximate them is to replace the N-dimensional space with a much lower dimensional space M where $M \ll N$. This is the essence of Krylov subspaces.

Definition 26 (Krylov Subspace)**.** Given an $N \times N$ matrix A and a vector $u \in R^N$ where $u \neq 0$, we define the M-th *Krylov matrix* $K_M \in R^{N \times M}$ as

$$K_M(A, u) = \begin{bmatrix} u & Au & A^2u & \cdots & A^{M-1}u \end{bmatrix}. \tag{2.30}$$

Then, the subspace spanned by the columns of $K_M(A, u)$ is called the M-th *Krylov subspace*, $\mathcal{K}_M(A, u)$:

$$\mathcal{K}_M(A, u) = \text{span}\{u, Au, A^2u, \ldots, A^{M-1}u\}. \tag{2.31}$$

Generally, $\mathcal{K}_M$, which is the *rank* of K_M, equals M, though it may be smaller (Dagréou et al., 2024). Krylov matrices and their subspaces possess the following mathematical properties.

Definition 27 (Properties of Krylov Matrices and Krylov Subspaces)**.** Given the M-th Krylov matrix $K_M(A, u)$ and the M-th Krylov subspace $\mathcal{K}_M(A, u)$ generated by $A \in \mathbb{R}^{N\times N}$ and $u \in \mathbb{R}^N$, if we have an $x \in \mathcal{K}_M$, then following properties hold:

- $x = K_M z$ for some $z \in \mathbb{R}^M$.
- $x \in K_{M+1}$.
- $Ax \in \mathcal{K}_{M+1}$.
- $\mathcal{K}_M \subseteq \mathcal{K}_{M+1}$, and so Krylov subspaces are nested.

The appeal of generating $\mathcal{K}_M$ lies in its reliance on matrix-vector products Av, which can be computed efficiently. In general, the Krylov subspace can be stored with $O(MN)$ space for a Krylov dimension M (Nocedal and Wright, 2006; Paszke et al., 2017). We can now solve our original problem $Ax = b$ by equivalently replacing x with $K_M z$ as described above.

$$\min_{x\in\mathcal{K}_M} \|Ax - b\| = \min_{z\in\mathbb{R}^M} \|A(K_M z) - b\| = \min_{z\in\mathbb{R}^M} \|(AK_M)z - b\|. \tag{2.32}$$

This is a much lower dimensional problem which we can solve efficiently. Note that the natural seed vector for this problem is b, and so we can equivalently solve this in the Krylov subspace $\mathcal{K}_M(A, b)$.

While the columns of K_M in Eq. (2.30) form a basis for $\mathcal{K}_M$, the basis is often ill-conditioned. This is especially the case as M increases. This is because repeated matrix-vector products with matrix-powers A^k tend to concentrate in the direction of the *dominant eigenvector* of A (Gutknecht, 2007).

Definition 28 (Dominant Eigenvector of Matrix-Powers)**.** Given a matrix $A \in \mathbb{R}^{N\times N}$, suppose we have eigenvalues $\lambda_1, \lambda_2, \lambda_3, \ldots, \lambda_N$ such that

$$|\lambda_1| \geq |\lambda_2| \geq |\lambda_3| \geq \cdots \geq |\lambda_N|. \tag{2.33}$$

Then, for any vector $u \in \mathbb{R}^N$, $A^k u$ is eventually a scalar multiple of the *dominant eigenvector* of A as $k \to \infty$. The dominant eigenvector is the vector associated with the largest magnitude eigenvalue, λ_1.

As such, when M is large, each basis vector $A^k u$ in our Krylov subspace $\mathcal{K}_M$ is increasingly parallel to the dominant eigenvector of A. The basis vectors are thus increasingly linearly dependent and the condition number of K_M deteriorates. To overcome this, we instead construct the opposite of an ill-conditioned basis, an orthonormal basis for $\mathcal{K}_M$.

Consider *QR factorisation*, which factorises a matrix A into an orthogonal matrix Q and an upper triangular matrix R (Nocedal and Wright, 2006). We can then write K_M as:

$$K_M = \mathrm{QR}(K_M) = Q_M R_M = [q_1, q_2, \ldots, q_M] \begin{bmatrix} R_{11} & R_{12} & \cdots & R_{1M} \\ 0 & R_{22} & \cdots & R_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & R_{MM}. \end{bmatrix} \tag{2.34}$$

The vectors $q_1, q_2, \ldots, q_M$ are the orthonormal basis for $\mathcal{K}_M$. By Definition 27, we know that $Aq_M \in \mathcal{K}_{M+1}$, and therefore,

$$Aq_M = h_{1M} q_1 + h_{2M} q_2 + \cdots + h_{M+1,M} q_{M+1} \tag{2.35}$$

for some choice of the h_{iM} coefficients. Using orthonormality, we have:

$$q_i^T (Aq_M) = h_{iM}, \quad i = 1, \ldots, M. \tag{2.36}$$

We can then solve for $h_{M+1,M}$ and q_{M+1} as these are the only unknowns in Eq. (2.35), since we started by assuming that we know $q_1, \ldots, q_M$. Given they appear as a product, and we know that q_{M+1} is a unit vector due to orthonormality, they must be uniquely defined by the other terms in the equation. This results in the *Arnoldi iteration* algorithm, which computes Q_M (Gutknecht, 2007). We outline this below in Algorithm 2.

Algorithm 2: The Arnoldi Iteration

Input: Matrix $\mathbf{A}$, vector $\mathbf{u}$, Krylov dimension M
Output: Orthonormal basis Q_M for $\mathcal{K}_M(A, u)$

$\mathbf{q}_1 \leftarrow \mathbf{u}/\|\mathbf{u}\|$
for $m = 1, 2, \ldots, M$ **do**
  Find h_{im} for $i = 1, \ldots, m$ using Eq. (2.36)
  Compute $\mathbf{v} = (A\mathbf{q}_m) - h_{1m}\mathbf{q}_1 - h_{2m}\mathbf{q}_2 - \cdots - h_{mm}\mathbf{q}_m$ via Eq. (2.35)
  $h_{m+1,m} \leftarrow \|\mathbf{v}\|$
  $\mathbf{q}_{m+1} \leftarrow \mathbf{v}/h_{m+1,m}$

The Arnoldi iteration iteratively constructs nested orthonormal bases for nested Krylov subspaces. Although we have focused on finding Q_M, the h_{ij}, the coefficient values h_{ij} are key. For $j = 1, 2, \ldots, M$ in Eq. (2.35) we get that

$$
\begin{aligned}
AQ_M &= [Aq_1 \dots Aq_M] \\
&= [q_1 \dots q_{M+1}] \begin{bmatrix} h_{11} & h_{12} & \cdots & h_{1M} \\ h_{21} & h_{22} & \cdots & h_{2M} \\ & h_{32} & \ddots & \vdots \\ & & \ddots & h_{M,M} \\ & & & h_{M+1,M} \end{bmatrix} \\
&= Q_{M+1} H_M. \qquad (2.37)
\end{aligned}
$$

We call this the *fundamental identity of Krylov subspaces* (Gutknecht, 2007). The matrix H_M is of size $M+1 \times M$ and has a special structure where $h_{ij} = 0$ whenever $i > j+1$[1]. Note that generally speaking, although we have used the similar notation, H_M is not related to the Hessian H.

When we project Q_M onto the original matrix A, we obtain a reduced system $Q_M^T A Q_M = H_M$ of size $M \times M$ (Gutknecht, 2007). This is crucial as this acts as an approximation to the original matrix A, and thus the eigenvalues of H_M are approximations to the eigenvalues of A. If we consider the case when A is the Hessian H, then we can use this to approximate the Hessian eigenvalue spectrum and obtain local curvature information. As the Krylov dimension M becomes closer to N, H_M becomes a better approximation of A (Gutknecht, 2007). This property makes Krylov subspaces very powerful for optimisation, as we can work in this reduced subspace while still obtaining a good solution to our problem.

Krylov subspaces have been key in developing many successful numerical solvers and deep learning optimisers. For example, the *Krylov Subspace Descent* algorithm can construct sound approximations and incorporate local curvature to perform well in deep learning tasks (Vinyals and Povey, 2012). We discuss these further in Chapter 3.

2.4.3 Trust-Region Methods

We finish this chapter by introducing *trust-region methods*, another class of second-order optimisation algorithms that can utilise curvature information. Trust-region methods use the quadratic model as we discussed in Sec. 2.3.3 to define a *region* around the current iterate that they *trust* to be a good approximation of the objective function, hence the name *trust-region* (Nocedal and Wright, 2006).

We discussed the quadratic model in Sec. 2.3.3 in the specific case of deep learning. We defined it with respect to a loss function $L(\theta)$ and the model parameters θ using the explicit Hessian H. We now define the quadratic model in the general case to use for trust-region methods.

[1] This is usually the definition of an upper Hessenberg matrix. However H_M is not square in our case since it has an extra row. If we remove the last row, then it becomes an upper Hessenberg matrix.

Definition 29 (General Quadratic Model)**.** Given an objective function $f : \mathbb{R}^N \to \mathbb{R}$, the quadratic model is the second-order Taylor expansion of f around a point $x \in \mathbb{R}^N$,

$$m(p) = f(x) + g^T p + \frac{1}{2} p^T B p, \tag{2.38}$$

where $g = \nabla f(x)$ is the gradient, and B is a symmetric matrix that usually approximates the Hessian $\nabla^2 f(x)$.

If we consider f as the loss function $L(\theta)$ and p as $\Delta\theta$, then this is the same as Eq. (2.19) when $B = H$. Trust-region methods employ this quadratic model to iteratively solve a *constrained optimisation problem* that is the *trust-region subproblem* (Nocedal and Wright, 2006). A constrained optimisation problem is one where we must adhere to constraints on our parameters (Boyd and Vandenberghe, 2004). Here, we optimise f subject to the constraint that our step size $||p||$ is less then a given *trust-region radius* Δ.

Definition 30 (The Trust-Region Subproblem)**.** Given a function $f : \mathbb{R}^N \to \mathbb{R}$ and a trust-region radius $\Delta > 0$, the trust-region subproblem is to find a step p that solves

$$\min_{p \in \mathbb{R}^N} m(p) \quad \text{subject to} \quad \|p\| \leq \Delta, \tag{2.39}$$

where $\| \cdot \|$ is usually the Euclidean norm.

The trust-region radius Δ controls the maximum size of our step p. It represents the region in which we trust our quadratic model to be a good approximation of the objective function. If Δ is too small, then we make little progress. If Δ is too large, our quadratic model we may overshoot and take a poor step. When we take a step p, we consider the ratio between the actual and prediction reduction in the objective function (Nocedal and Wright, 2006). We denote this ratio as ρ.

$$\rho = \frac{f(x) - f(x+p)}{m(0) - m(p)} = \frac{\text{actual reduction}}{\text{predicted reduction}}, \tag{2.40}$$

For our step p, we want ρ to be always non-negative, as this guarantees that p is a descent direction (Nocedal and Wright, 2006). We can characterise the behaviour of the trust-region based on the value of ρ as follows:

- *Rejection*: If ρ is negative, then $f(x+p) > f(x)$ and so our step does not minimise the objective function. In this case, we *reject* the step (Nocedal and Wright, 2006).
- *Acceptance and Expansion*: If ρ is close to 1, then our model m is a good approximation of f and we can accept p. If we also have that $\|p\| = \Delta$, then we *expand* our trust-region radius Δ as this indicates that the step has reached the boundary of our trust-region (Nocedal and Wright, 2006).
- *Acceptance and Shrinkage*: If ρ is close to 0, then our model is a poor approximation of f and we should *shrink* our trust-region radius Δ (Nocedal and Wright, 2006).

To turn this into an optimisation algorithm, we iteratively solve the trust-region subproblem at the current iterate x_t and update our trust-region radius Δ_t. We illustrate the algorithm in Algorithm 3.

Algorithm 3: Trust-Region Method

Input: Initial point x_0, initial trust-region radius $\Delta_0 > 0$, maximum radius $\hat{\Delta} > 0$, thresholds $0 < c_1 < c_2 < 1$, reduction factor $\gamma_1 \in (0, 1)$, expansion factor $\gamma_2 > 1$, maximum iterations T

Output: Approximate local minimiser x_T

$x \leftarrow x_0$
$\Delta \leftarrow \Delta_0$
for $t = 0, 1, 2, \ldots, T$ **do**
  Construct the quadratic model $m_t(p)$ in Eq. (2.38)
  Obtain p_t by solving the trust-region subproblem in Eq. (2.39)
  Compute the ratio ρ_t in Eq. (2.40)
  if $\rho_t < 0$ **then**
    $p_t \leftarrow 0$
    $\Delta \leftarrow \gamma_1 \Delta$
  else if $\rho_t < c_1$ **then**
    $\Delta \leftarrow \gamma_1 \Delta$
  else if $\rho_t > c_2$ ***and*** $\|p_t\| = \Delta$ **then**
    $\Delta \leftarrow \min(\gamma_2 \Delta, \hat{\Delta})$
  $x_{t+1} \leftarrow x_t + p_t$
return x_T

The core of this algorithm is the trust-region subproblem. We discussed the solution to this in the deep learning case in Sec. 2.3.3 when $B = H$ to compute the Newton step. We also made an important note that the Newton step as a result of the quadratic model is not always a descent direction when H is not positive semidefinite. We now adapt this for the general case, in which we want to solve

$$(B + \lambda I)p^* = -g, \tag{2.41}$$

where $\lambda \geq 0$ and p^* is the solution. As a result the following conditions must be met if p^* is the solution.

Definition 31 (Trust-Region Subproblem Conditions)**.** If p^* is the solution to Eq. (2.41), then the following[2] must be satisfied:

- $\|p^*\| \leq \Delta$.
- $B + \lambda I$ is positive semidefinite.
- $\lambda(\Delta - \|p^*\|) = 0$.

[2] These are essentially the *Karush-Kuhn-Tucker* (KKT) conditions for the trust-region subproblem.

These conditions characterise the exact solution, but solving this exactly is computationally expensive as discussed in Sec. 2.3.3. In practice, we use approximation methods. A popular approach for approximately solving the trust-region subproblem is *two-dimensional subspace optimisation*, where we restrict ourselves to a low dimensional subspace (Nocedal and Wright, 2006). We now have that

$$\min_{p \in \mathcal{S}} m(p) \quad \text{subject to} \quad \|p\| \leq \Delta, \tag{2.42}$$

where $\mathcal{S} = \text{span}\{-g, B^{-1}g\}$ is the two-dimensional subspace (Nocedal and Wright, 2006). The first direction, $-g$, is the steepest descent direction which guarantees reduction. The second direction $B^{-1}g$ is also a solution when B is positive definite. In the case it is not, we can then change the subspace to be $\text{span}\{-g, (B + \alpha I)^{-1}g\}$ where $\alpha \geq \lambda_{min}(B)$. Intuitively, this searches for a solution that is a linear combination of the descent direction *and* the approximate Newton direction. Since we operate in subspace of size 2, this becomes extremely inexpensive to solve. Subspace optimisation often is close to the exact solution of Eq. (2.41). We can also extend two-dimensional subspace optimisation to higher dimensions. For example, it may be beneficial to find a solution within the subspace $\text{span}\{-g, B^{-1}g, z\}$ where z is an average of past gradients (known as the momentum).

With appropriate choices of B, trust-region methods poses strong convergence guarantees. These include:

- *Global Convergence*: Trust-region methods converge globally to stationary points regardless of the starting point, given B is uniformly bounded and g is Lipschitz continuous (Nocedal and Wright, 2006) [3].
- *Protection against negative curvature*: Given we have a model approximation that we trust around a radius, we are protected against directions of negative curvature that might cause to diverge. This is since we will always stop at the boundary of the trust region.
- *Second-order necessary conditions*: When $B = H$, trust-region methods guarantee convergence to points satisfying the second-order necessary conditions as seen in Definition 10 (Nocedal and Wright, 2006).
- *Superlinear or quadratic convergence*: Trust-region methods achieve superlinear or quadratic convergence rates near solutions satisfying second-order sufficient conditions as seen in Definition 15, when using accurate Hessian approximations.

These properties make trust-region methods particularly useful in deep learning, as they offer a good balance between rapid progress, stability, and computational efficiency.

In this chapter, we have covered a complete overview on the fundamentals of optimisation, optimisation in deep learning, and how to efficiently incorporate curvature

[3] A function is Lipschitz continuous if there exists a constant L such that the slope between any two points is bounded by L, meaning the function cannot change too rapidly anywhere in its domain

information for optimisation algorithms. We now discuss these algorithms and perform a comprehensive literature review of the state-of-the-art and foundational methods.

Chapter 3

Literature Review

The work in this thesis is heavily inspired by the advances in deep learning optimisation. In this chapter, we explore the major academic works that have played a significant role in the field of optimisation. In Sec. 3.1, we provide an overview of first-order methods and their categorisations. This is followed by a discussion of second-order methods and their categorisations in Sec. 3.2. We then look at other techniques in the field, namely non-smooth optimisation and meta-learning in Sec. 3.3 and Sec. 3.4. At the end of our discussion for each group of works in these respective sections, we will compare and contrast them to our own contributions, which are detailed in the next chapter.

3.1 First-Order Methods

In the previous chapter, we saw that an optimisation algorithm in deep learning finds a set of model parameters θ^* that minimises the loss function $L(\theta)$ introduced in Eq. (2.10). We covered in Sec. 2.3.2 that first-order methods are characterised by computing the first order $\nabla_\theta L$, or g, to use in optimisation. We now look at the different types of first-order methods, including *gradient-based* methods, *momentum* methods, and *adaptive* methods.

3.1.1 Gradient-Based Methods

Gradient Descent: One of the earliest methods deployed to solve such optimisation problems is gradient descent (GD). We introduced the concept of steepest descent in Sec. 2.3.2, where we saw the parameters being iteratively updated with a scalar multiple, the learning rate α_t, of the steepest descent direction, the negative gradient $-g$ at each step given an iteration t. This gives $\theta_{t+1} = \theta_t - \alpha_t g_t$. Recall that the learning rate α controls the step size of each iteration (Ruder, 2016). GD gradually approaches a local optimum of our loss function L, and given appropriate α, reaches this sub-linearly

if the loss function is *convex*, converging linearly if it is *strongly convex* (Nocedal and Wright, 2006). However, GD is expensive. In many machine learning tasks, the loss function is evaluated with N samples, each with dimensionality D. Thus, GD requires $O(ND)$ per iteration to evaluate. This becomes infeasible very quickly as N and D scale, and disallows online updates and limits model adaptability (Ruder, 2016). While some parallelisation methods were proposed to combat this, the scalability issue still remains (Alspector et al., 1992; Nocedal and Wright, 2006).

Stochastic Gradient Descent: To address the limitations of GD, stochastic gradient descent (SGD) was proposed. Instead of evaluating g using all N, a random sample $n \in N$ is picked and instead a stochastic gradient $\hat{g}$, an unbiased estimate of the real gradient, is computed to update θ (Robbins and Monro, 1951). SGD achieves the same convergence rates as GD given L satisfies the same convexity conditions (Johnson and Zhang, 2013; Nemirovski et al., 2009). Each iteration reduces from $O(ND)$ to $O(D)$ given only one sample is used. This overcomes the disadvantage of GD in two ways: online updates can be performed, and convergence can be accelerated, as an optimal θ can be found using $n << N$ samples (Johnson and Zhang, 2013; Nemirovski et al., 2009). However, the stochastic gradients used in SGD are inherently noisy and fluctuate. These fluctuations can be beneficial, as they allow SGD to jump from one place to another. This behavior is particularly useful in high-dimensional spaces, where we introduced in Sec. 2.3.1 that most local minima are approximately equal to global minima. If SGD gets trapped in regions such as saddle points where it is hard to make progress, the fluctuations can help escape them.

On the other hand, these fluctuations can also lead to large variance in the gradient estimates (Sun et al., 2019). This can make the optimisation process unstable and slow convergence. Convergence is also affected by the learning rate α. Setting a too low α will slow convergence, and setting a too high α can hinder convergence all together, either overshooting or oscillating at the minimum. Choosing a good α usually requires manual tuning, which is arduous and compute heavy.

Mini-batch Stochastic Gradient Descent: The compromise between SGD and GD resulted in mini-batch stochastic gradient descent (mini-batch SGD) (Robbins and Monro, 1951). mini-batch SGD splits the N samples into $m << N$ i.i.d *batches* (usually ranging from 64 to 256). θ is updated each iteration by with the stochastic batch gradient based off one batch. Over time, all batches are processed. This reduces the variance in the gradients and makes convergence more stable, which helps optimisation speed, though we note that the α parameter is still needed. mini-batch SGD is now a mainstream technique used for optimising machine learning models. To align with the standard in the literature, we will refer to mini-batch SGD as SGD from now on.

Our work is different from pure gradient-based methods such as GD and SGD. In particular, our work does not only use the gradient direction, but incorporates this with second-order information. Our step direction is instead a linear combination of the gradient direction with other useful directions minimised in a trust-region framework. Our

work also does not need a learning rate like these methods, as we calculate the coefficients from our trust-region framework. We note that our work is similar to SGD in that we sample a mini-batch for each iteration to compute the relevant information.

3.1.2 Momentum Methods

Momentum SGD: Many works have been proposed to improve upon SGD, one of which is momentum (MSGD) (Polyak, 1964). The momentum method augments SGD with a *momentum* variable z, a decaying average of past gradients controlled by a *momentum factor* β (Polyak, 1964; Henriques et al., 2019). The update is then performed using z as,

$$z_{t+1} = \beta z_t - g_t \tag{3.1}$$
$$\theta_{t+1} = \theta_t + \alpha z_{t+1}. \tag{3.2}$$

This encourages progress along consistent, but small gradient directions. MSGD can converge faster, remains more stable under changing α, and is more resistant if our loss function is poorly scaled. However, this adds another hyperparameter to tune, and the same problems as seen with α arise. Setting a β too low will not result in improvements, and setting it too high can result in overshooting.

Nesterov Accelerated SGD: A small improvement over MSGD is Nesterov-Accelerated SGD (NAG). NAG first updates θ with z, and then calculates the gradient based on the updated θ (Nesterov, 1983; Sutskever et al., 2013),

$$\tilde{\theta}_t = \theta_t + \beta z_t \tag{3.3}$$
$$z_{t+1} = \beta z_t - \nabla_\theta F(\tilde{\theta}_t) \tag{3.4}$$
$$\theta_{t+1} = \theta_t + \alpha z_{t+1}. \tag{3.5}$$

Updating based on the future position of θ includes more gradient information in comparison to traditional momentum, which can provide a better direction (Nesterov, 1983; Dozat, 2016). NAG can also result in superlinear convergence when there is no stochasticity (Nesterov, 1983; Sutskever et al., 2013).

Our work incorporates momentum, but it is different from pure momentum-based methods since we also incorporate second-order information. Additionally to emphasise again, we do not need a learning rate like these methods. Our work uses second-order information to complement the momentum and the gradient in a trust-region framework. As such, our work has the benefit of momentum methods, while also having faster convergence from the guarantees of second-order methods.

3.1.3 Adaptive Methods

AdaGrad: To tackle the problem of setting an appropriate α, adaptive methods that adjust or scale α directly per parameter were proposed Duchi et al. (2011). The AdaGrad

algorithm keeps a history of previous gradients and adjusts the learning rate dynamically (Duchi et al., 2011). It accumulates the historical gradients via squaring, then scales the current gradient (Duchi et al., 2011),

$$V_t = \sqrt{\sum_{i=1}^{t} g_i \odot g_i} \tag{3.6}$$

$$\theta_{t+1} = \theta_t + \alpha \frac{g_t}{V_t + \epsilon}, \tag{3.7}$$

where $\odot$ denotes element-wise multiplication and the division is also performed element-wise. Note that the ϵ term is added for numerical stability. The result of this is that only an initial α needs to be set, as during each update the learning rate will adapt due to scaling. Though, as $t \to \infty$, $V_t \to \infty$, and the learning rate is driven towards zero as the denominator in the update term becomes very large (Ruder, 2016; Geoff, 2012; Zeiler, 2012). This makes the parameter updates ineffective. This is especially the case in non-convex settings and regions such as saddle points, as the learning rate will be very quickly driven to zero before making sufficient progress.

RMSProp and AdaDelta: To address this, instead of accumulating all of the gradients, RMSProp takes inspiration from momentum (Ruder, 2016; Geoff, 2012). It uses an exponential decaying moving average to weight the past squared gradients using a decay parameter p (Geoff, 2012; Kingma, 2014). The update rule is then,

$$V_t = \sqrt{pV_{t-1} + (1-p)g_t^2}. \tag{3.8}$$

This simulates having a window size w, where gradients outside of this window are forgotten (Geoff, 2012; Kingma, 2014). It priorities more recent gradients and so the learning rate is more stable and does not diminish with time. AdaDelta improves upon this and also keeps track of the change in updates, $\Delta(\theta_t)$, performed (Zeiler, 2012). This results in

$$U_t = \sqrt{pU_{t-1} + (1-p)(\Delta(\theta_t))^2}. \tag{3.9}$$

AdaDelta then uses these accumulative updates at each iteration to perform the actual update which is

$$\theta_{t+1} = \theta_t - \frac{U_{t-1}}{V_t + \epsilon} g_t. \tag{3.10}$$

The key observation here is the absence of α. This is a step up from most SGD methods, which require the α parameter. The algorithm is also suitable in non-convex settings since there is no diminishing learning rate (Zeiler, 2012). However, the absence of α is replaced with p and ϵ which still need to be tuned, though these are much less sensitive.

Adam: Perhaps the most successful algorithm for optimisation is Adaptive Moment Estimation (Adam) (Kingma, 2014). Adam maintains two moving averages: one for the

gradients themselves, m_t, and the same V_t used in RMSProp/AdaDelta (Kingma, 2014). Each of these have an associated decay term β_1 and β_2, which weight the gradients and the history (Kingma, 2014).

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{3.11}$$

$$V_t = \sqrt{\beta_2 V_{t-1} + (1 - \beta_2) g_t^2} \tag{3.12}$$

$$\hat{m}_t = \frac{m_t}{1 - {\beta_1}^t} \tag{3.13}$$

$$\hat{V}_t = \frac{V_t}{1 - {\beta_2}^t} \tag{3.14}$$

$$\theta_{t+1} = \theta_t - \alpha \frac{\hat{m}_t}{\hat{V}_t + \epsilon} \tag{3.15}$$

We call m_t and V_t the *first* and *second* moment estimates respectively, with β_1 and β_2 being the first and second moment hyper-parameters (Kingma, 2014). Adam has a few key advantages. It uses momentum and exponential decay to smooth out gradients, which makes the optimisation process more stable. It also has bias correction applied to $\hat{m}_t$ and $\hat{V}_t$ to allow for proper initialisation (Kingma, 2014). This ensures they are not biased towards their starting estimates. This is important during early stages of optimisation, as there is insufficient data to estimate the moments. Adam is stable and effective in practice, and works well with both convex and non-convex loss functions. Adam is the current state-of-the-art optimisation method in deep learning. However, it lacks theoretical guarantees and convergence is not well understood (Reddi et al., 2019). Adam has been shown to fail in simple one dimensional convex settings (Reddi et al., 2019). It additionally needs two extra parameters, β_1 and β_2, in comparison to traditional SGD, which need to be tuned.

Our work is similar in concept to these adaptive methods. While these methods adjust the learning rates based on historical gradient information, we instead use a trust-region framework. The step sizes are determined by trust-region optimisation. We use second-order information, and do not need a learning rate, or first and second moment hyperparameters, and also have convergence guarantees.

3.1.4 Adam Variants

AdaMax: Several variants have been proposed to address Adam's limitations. The first variant is AdaMax. AdaMax replaces the L_2 norm in V_t with the L_∞ norm (Kingma, 2014). Specifically, instead of maintaining the average of the squared gradients V_t, AdaMax keeps track of the maximum absolute value of the past gradients (Kingma,

2014).

$$u_t = \max(\beta_2 \cdot u_{t-1}, |g_t|) \tag{3.16}$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{3.17}$$

$$\theta_{t+1} = \theta_t - \alpha \cdot \frac{\hat{m}_t}{u_t}, \tag{3.18}$$

where we do a component wise max and component wise division. Here, the second moment estimate $\hat{V}_t$ is replaced with u_t. This is done to make the algorithm more robust, as $u_1, u_2, \ldots, u_t$ are influenced by fewer gradients and thus there is less noise. This is especially the case for when the gradients are large and can become numerically unstable (Kingma, 2014). Note that no bias correction is also needed for u_t.

NAdam: Another variant of Adam is Nesterov Adam (NAdam), which extends Adam with the NAG method seen in Sec. 3.1.2 (Dozat, 2016). NAdam uses the NAG update by performing the *Nesterov* trick - in which the previous first moment is replaced with the currently calculated first moment similar to NAG (Nesterov, 1983; Dozat, 2016). Condensing the Adam update equations from Eq. (3.11) to Eq. (3.15) in Eq. (3.19), the following update is changed from

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\hat{V}_t + \epsilon} \left(\beta_1 \mathbf{\hat{m}_{t-1}} + \frac{(1 - \beta_1) g_t}{1 - (\beta_1)^t} \right) \tag{3.19}$$

to

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\hat{V}_t + \epsilon} \left(\beta_1 \mathbf{\hat{m}_t} + \frac{(1 - \beta_1) g_t}{1 - (\beta_1)^t} \right). \tag{3.20}$$

Like NAG, NAdam uses the future position of θ to calculate the gradient and incorporate more information in the update. NAdam has been shown to yield sizeable improvements in performance in comparison to Adam on MNIST and Word2Vec tasks (Dozat, 2016).

AdamW: Decoupled weight decay regularisation, or AdamW, is a further variant of Adam (Loshchilov, 2017). Normally, applying L2 regularisation in Adam involves appending the regularisation term to the gradient update as follows,

$$g_t = \nabla_\theta F(\theta_t) + \lambda \theta_{t-1}. \tag{3.21}$$

However, upon rescaling, the effect of the regularisation is not properly accounted for and large g_t do not get regularised as much as they should (Loshchilov, 2017). To address this, AdamW decouples the weight decay and applies it alongside the update.

$$\theta_{t+1} = \theta_t - \alpha \left(\frac{\hat{m}_t}{\hat{V}_t + \epsilon} + \lambda \theta_t \right). \tag{3.22}$$

This is because in adaptive gradient methods, L2 regularisation in the gradient update is *not* the same as weight decay - unlike SGD (Loshchilov, 2017). Adam is not effective with

L2 regularisation, and so AdamW substantially benefits from this decoupling, improving in performance on CIFAR10 and ImageNet-32 tasks (Loshchilov, 2017). AdamW is widely used in practice and is also a current state-of-the-art optimisation method.

AMSGrad: A particularly well-known issue with Adam is the lack of convergence guarantees. One limitation is that in certain scenarios, Adam aggressively increases the learning rate, even when the algorithm is close to the optimum (Reddi et al., 2019). This is because the second moment term V_t can grow indefinitely, and there is a very high dependence on β_2. For particular β_2, highly suboptimal solutions can be reached in the case of simple convex settings (Reddi et al., 2019).

AMSGrad modifies this by keeping a bound on $\hat{V}_t$ by taking the maximum for the update rule (Reddi et al., 2019).

$$\hat{V}_t = \max(\hat{V}_{t-1}, V_t) \tag{3.23}$$

This guarantees that the learning rate is not increased indefinitely, as each V_t is guaranteed to be non decreasing and at least as large as the previous V_t's (Reddi et al., 2019). By ensuring this, drastic and aggressive learning rate increases are prevented, and the algorithm convergences better and is more stable. However, AMSGrad is still dependent on β_2, and cannot show convergence guarantees on any arbitrary β_2 (Taniguchi et al., 2024). Some works have demonstrated that problem dependent tuning of β_2 can lead to convergence, but this still requires manual tuning (Taniguchi et al., 2024).

ADOPT: To address the dependency of convergence on β_2, ADaptive Gradient Method with the OPTimal Convergence Rate (ADOPT) was most recently proposed (Taniguchi et al., 2024). ADOPT shows that by modifying the update rules of Adam, convergence can be attained for any $\beta_2 \in [0, 1]$. The non-convergence of Adam can be attributed to the V_t needing to be conditionally independent of the current gradient g_t (Taniguchi et al., 2024). In a normal Adam update as in Eq. (3.15), the conditional independence criteria is not satisfied. This is because $\hat{V}_t$ contains information about g_t, and so convergence breaks. ADOPT fixes this by modifying the order of the update and the normalisation (Taniguchi et al., 2024). Specifically, the normalisation is applied to the current gradient in a modified update rule of m_t,

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \frac{g_t}{\hat{V}_{t-1} + \epsilon^2}. \tag{3.24}$$

Here, the previous second moment, V_{t-1} is used to ensure this conditional independence with g_t when scaling and so convergence is guaranteed. In practice, ADOPT uses $\max(\hat{V}_{t-1}, \epsilon)$ instead of $\hat{V}_{t-1} + \epsilon^2$ for the normalisation for performance benefits (Taniguchi et al., 2024). The parameter update is shortly applied after as

$$\theta_{t+1} = \theta_t - \alpha \cdot m_{t+1}. \tag{3.25}$$

ADOPT outperforms Adam, AdamW, and AMSGrad on MNIST and CIFAR10 image classification tasks (Taniguchi et al., 2024). It achieves substantial improvements in large

language model pretraining and finetuning tasks over Adam and the rest of the variants as well (Taniguchi et al., 2024).

Our work presents a different paradigm from these Adam variants. While these variants focus on the convergence and reliance on tuning, as we mentioned in the previous section, our work does not need explicit learning rate and moment hyperparameters. Our work has theoretical convergence guarantees due to our trust-region framework, whereas these works address the problem of Adam's convergence rather than exhibiting these guarantees.

3.2 Second-Order Methods

As discussed in Chapter 2, first-order methods are effective for many optimisation problems, but fall short when the optimisation landscape is ill-conditioned or challenging. We saw that second-order methods address these problems in Sec. 2.3.3, as they find or approximate the Hessian H. We now discuss the different types of second order methods.

3.2.1 Newton and Quasi-Newton Methods

Damped and Regularised Newton Methods: We saw in Sec. 2.3.3 that while Newton's method offers rapid quadratic convergence, it may not produce a descent direction, when the Hessian H is not positive definite or when far from a solution. Damped and regularised Newton methods improve the robustness and global convergence properties of the standard Newton step. *Step-size damping* introduces the learning rate α_t to scale the Newton direction, similar to first-order methods we saw in Sec. 3.1 (Sun et al., 2019):

$$\theta_{t+1} = \theta_t - \alpha_t H_t^{-1} g_t. \tag{3.26}$$

The step length α_t is typically chosen using a line search at each iteration to ensure sufficient decrease. This prevents overly aggressive steps and improves stability. Another approach is *regularisation damping*. Instead of scaling the step, this method modifies the Hessian matrix H_t directly to ensure it is sufficiently positive definite. A common technique is to add a constant λ to the eigenvalues of H_t before inversion. This yields the update:

$$H_t = V \Lambda V^T \tag{3.27}$$

$$B_t = (V(\Lambda + \lambda I)V^T)^{-1} \tag{3.28}$$

$$\theta_{t+1} = \theta_t - B_t g_t. \tag{3.29}$$

This damps the eigenvalues of H_t, ensuring the modified Hessian is well-conditioned. The parameter λ_t can be adjusted adaptively,and instead of adding, we can scale the eigenvalues of H_t. As we noted in Sec. 2.4.3, this formulation is closer to the solution of the trust-region subproblem. Both damping strategies enhance the reliability of Newton's method, making it more applicable to the non-convex landscapes. However, we

note that the computational cost is still a problem. With regularisation damping, we need to explicitly perform an eigendecomposition of H_t and then reconstruct our modified Hessian. This is intractable for large H_t as the eigendecomposition is $O(N^3)$. Thus, while these methods increase the stability of Newton's method while maintaining the convergence properties, the computation intractability still remains.

BFGS: We address the computational infeasibility of Newton methods with Quasi-Newton methods. Quasi-Newton methods compute an explicit approximation of the Hessian, rather than computing it directly. The Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm generates a sequence of matrices to estimate the Hessian (Nocedal and Wright, 2006). It iteratively refines the estimate using gradient information and rank-two updates at each iteration (Nocedal and Wright, 2006). The estimate is always positive-definite. Through experimental testing and analysis, these estimates are quite good. BFGS also has effective self-correcting properties. If an estimate becomes poor at one iteration, within the next few it will correct itself (Nocedal and Wright, 2006). BFGS reduces the complexity to $O(N^2)$ for computation time (Nocedal and Wright, 2006; Sun et al., 2019).

A common variant of BFGS, Limited BFGS (LBFGS), reduces the memory requirement by storing k vectors instead of retaining the full $n \times n$ approximations (Nocedal and Wright, 2006; Sun et al., 2019).

SR1: The Symmetric Rank-One (SR1) method further tackles the infeasibility problem. It approximates the inverse Hessian using the difference between a history of gradients and positions, while ensuring symmetry in the update (Nocedal and Wright, 2006). The key advantage of SR1 is that it uses only rank-updates, but also does not guarantee positive definiteness (Nocedal and Wright, 2006). This makes it beneficial in non-convex settings, where positive definiteness guarantee is hard to maintain. Albeit at the cost of stability, it helps the algorithm capture more curvature information.

Stochastic Quasi-Newton with Trust Region: BFGS and SR1 methods can be adapted into a stochastic setting with a trust-region framework. sL-BFGS-TR and sL-SR1-TR were proposed as methods following this setting. These methods compute the Hessian approximation D_t at each iteration using either L-BFGS or SR1. This D_t is then used in the second-order approximation of L to solve the trust-region subproblem. Results for sL-BFGS-TR and sL-SR1-TR show that they are efficient and are able to keep up with Adam on MNIST and CIFAR10 tasks (Yousefi and Martínez, 2023). Specifically, as batch size increases, these algorithms are faster than Adam on models such as LeNet, ResNet and a 3C3L (3 Convolution 3 Linear) ConvNet (Yousefi and Martínez, 2023). Together with an effective mini-batch sampling strategy, sL-BFGS-TR and sL-SR1-TR converge faster than Adam (Yousefi and Martínez, 2023). This shows that Quasi-Newton methods can be effective in deep learning, and can be used in a stochastic setting, even if contrary to popular belief.

Our work differs from traditional Quasi-Newton methods like BFGS, LBFGS, and SR1 in its approach to curvature information. Our work instead directly utilises HVPs to

build an approximation instead of using rank-one or rank-two updates. Our work also incorporates momentum and the gradient in a trust-region framework to get our step direction, instead of relying purely on the second-order approximation. Additionally, while the sL-BFGS-TR and sL-SR1-TR methods use a trust-region, our approach is different. We use subspace optimisation to solve the trust-region subproblem and to find a linear combination of useful directions, whereas these methods instead use other methods to solve the subproblem and instead maintain an explicit Hessian approximation matrix.

3.2.2 Diagonal Hessian Estimation

AdaHessian: As Quasi-Newton methods aim to estimate the inverse Hessian, improvements have been made to use diagonal hessian estimates as well. AdaHessian modifies the second moment from Adam to use the squared diagonal estimate of the Hessian (Yao et al., 2021).

$$\hat{D}_t \approx \text{diag}(H_t) \tag{3.30}$$

$$V_t = \sqrt{\beta_2 V_{t-1} + (1-\beta_2)\hat{D}_t \hat{D}_t} \tag{3.31}$$

This brings down the space complexity to $O(D)$, and makes it more feasible to use incorporate second order information (Yao et al., 2021). Though, as a trade-off, an extra backward pass is required per iteration. AdaHessian performs well in image classification, NLP, and recommender system tasks (Yao et al., 2021).

Sophia: A more effective approach to using diagonal estimates was proposed by Second-order Clipped Stochastic Optimisation (Sophia) (Liu et al., 2023). Sophia uses a diagonal estimate of the Hessian as a preconditioner to the first moment in the update rule alongside a clipping mechanism. This was motivated by heterogenous curvature in deep learning models, in which traditional optimisers struggle with (Liu et al., 2023).

Sophia first computes the diagonal estimate of the Hessian, $\bar{D}_t$, every k iterations using either the Hutchinson or Gauss-Newton-Bartlett estimation methods via an estimator E (Liu et al., 2023).

$$\hat{D}_t = E(H_t) \quad E \in \{\text{Hutchinson, Gauss-Newton-Bartlett}\} \tag{3.32}$$

$$\bar{D}_t = \begin{cases} \beta_2 \bar{D}_{t-k} + (1-\beta_2)\hat{D}_t & \text{if } t \mod k = 1, \\ \bar{D}_{t-1} & \text{otherwise} \end{cases} \tag{3.33}$$

This is then put into a clipping function that controls the worst case size update, in which the update rule is

$$\theta_{t+1} = \theta_t - \alpha \cdot \text{clip}\left(\frac{m_t}{\max(\gamma \cdot \bar{D}_t, \epsilon)}, \rho\right) \tag{3.34}$$

where $\text{clip}(z, p) = \max(\min(z, p), -p)$ and γ, ρ are hyper-parameters that function as a scaling factor and a clipping threshold respectively (Liu et al., 2023). We saw in

Sec. 2.3.3 that incorporating second-order information through the Newton step can result in moving in the wrong direction. To counter this, Sophia considers only the positive entries of $\bar{D}_t$, and clips each coordinate to a maximum step size of p (where $p = 1$ usually) (Liu et al., 2023). If any entry of $\bar{D}_t$ is negative, Sophia falls back to momentum signSGD. This is a method in which the update is scaled by the sign of the gradient for each component, ignoring the magnitude. This ensures that in the worst case, the update is controlled with a size of p, improving stability. On language modelling tasks, Sophia converges quicker and achieves better performance in comparison to AdamW Liu et al. (2023).

Our work is different since it utilises curvature information from Hessian-vector products, instead of relying on diagonal Hessian estimates. However, it is similar in that we use a clipping threshold to clip the eigenvalues of our Hessian approximation. We also can optionally use diagonal estimates to precondition the vector used for the HVPs when building our approximation. Our work is more stable than Sophia, as it offers a richer representation of the local curvature. We also use a trust-region framework which these works do not employ.

3.2.3 HVP-based Methods

Hessian-Free Optimisation: Hessian-free optimisation (HF) uses HVPs to incorporate second-order information (Nocedal and Wright, 2006; Martens et al., 2010). Similar to Newton's method, a search direction is computed by solving the following linear system, where α_t is a step size computed to guarantee sufficient decrease (Martens et al., 2010).

$$H_t d_t = -g_t \tag{3.35}$$

$$\theta_{t+1} = \theta_t - \alpha_t d_t. \tag{3.36}$$

The system in Eq. (3.36) is solved with the conjugate-gradient solver (CG), an iterative method that solves linear systems. This computes the HVP without using the Hessian, which reduces the space to $O(N)$ and results in $O(KN)$ time, where $K << N$ controls the number of iterations used for CG (Martens et al., 2010). However, the CG solver can be very unstable when H_t is not positive definite. Thus, it breaks down when dealing with noise and stochasticity. Ill-conditioned and non-convex problems also add to these issues.

CurveBall: Combining fast HVPs and curvature information with the heavy-ball framework was introduced with CurveBall, hence its name. CurveBall uses a quadratic approximation like the Newton method, and solves the optimisation problem by solving the same linear system as in Eq. (3.36). Instead of using CG or matrix inversion, it uses GD to optimise on d_t and finds Δd_t Henriques et al. (2019).

$$\Delta d_{t+1} = H_t d_t + g_t \tag{3.37}$$

$$z_{t+1} = \beta z_t - \alpha_2 \Delta d_{t+1} \tag{3.38}$$

$$\theta_{t+1} \leftarrow \theta_t + \alpha_1 z_{t+1} \tag{3.39}$$

The advantage to this is that there is no restriction on H. The algorithm can work in non-convex and ill-conditioned problems too, as demonstrated by it's performance on Rosenbrock and the Rahimi-Recht function (Henriques et al., 2019). The Δd_t variable keeps track of how H_t and g_t change as θ evolves, which incorporates the curvature. To amortize cost, it interleaves updates between θ and Δd_{t+1} (Henriques et al., 2019). With the use of fast HVPs through forward-mode (FMAD) and backward-mode automatic differentiation (RMAD), it requires only two passes of back-propagation (Henriques et al., 2019), as we discussed in Sec. 2.4.1. This makes it highly efficient and scalable, in which it demonstrates better performance on MNIST and CIFAR10 classification tasks (Henriques et al., 2019).

Our method is similar to these works since the core of it is using HVPs. However, we do not use CG or GD to optimise and solve the linear system present in Eq. (3.36). The problem we solve is similar, but not the same as Eq. (3.36), as ours is a regularised/damped version of it. We also solve this problem in a low dimensional subspace generated through Krylov subspaces, and then compute useful search directions from this and embed them in a trust-region framework.

3.2.4 Krylov Subspace Methods

Krylov Subspace Descent: Approximate solutions to optimisation problems in low dimensional subspaces are also a popular strategy to involve curvature information. Krylov Subspace Descent (KSD) constructs a basis of vectors $\mathcal{K}_m$ and then finds a search direction in $\mathcal{K}_m$ using BFGS (Vinyals and Povey, 2012). The Krylov basis is constructed with diagonal Hessian estimate preconditioning (Vinyals and Povey, 2012).

$$\mathcal{K}_m = \{(D^{-1}H)^k D^{-1} g | \quad 0 \leq k \leq m\} \tag{3.40}$$

During optimisation, this subspace is converted into a new non-orthogonal subspace $\hat{\mathcal{K}_m}$, where BFGS is run to find to $\hat{K}v$ as the search direction, where $\hat{K} \in \hat{\mathcal{K}_m}$ (Vinyals and Povey, 2012). KSD exhibits lower training error and faster running time in comparison to HF on MNIST (Vinyals and Povey, 2012). It also has no assumptions on H, unlike HF which has stability issues if H is not positive semidefinite.

Our method is quite similar to Krylov Subspace Descent. Both approaches construct a low-dimensional subspace using HVPs, but the subsequent utilisation of this subspace differs. KSD applies an optimiser like BFGS within the generated Krylov subspace to find a search direction. In contrast, our method instead embeds search directions found within this subspace alongside the gradient and the momentum in a trust-region. We thus derive our update from the trust-region rather than directly optimising within the initial Krylov subspace.

3.3 Non-smooth Optimisation

For some problems, there may be conditions on the loss function such that it is non-smooth and that g is intractable to calculate. We now look at optimisation methods that address this case.

Coordinate Descent: Coordinate Descent (CD) aims to perform one dimensional search across along each axis direction of θ, till all directions are chosen properly to find optimal θ. Naively, a set of bases $E = e_1, ..., e_D$ are chosen where $D = \dim(\theta)$. The parameter space is broken down into individual dimensions or coordinates, and optimization proceeds along each direction sequentially (Conn et al., 2009). As such, the update at the jth dimension is given by

$$\theta_j^{t+1} = \arg\min_{\theta_j \in \mathbb{R}} L(\theta_1^{t+1}, \ldots, \theta_{j-1}^{t+1}, \theta_j, \theta_{j+1}^{t}, \ldots, \theta_D^{t}). \tag{3.41}$$

This guarantees convergence, as L is set to decrease or stay the same at each iteration, and the convergence of CD is similar to that of GD (Conn et al., 2009). The main difference to GD is that each update is always axis aligned, whereas the g in GD may not be aligned with any $e \in E$. Given the algorithm focuses on one dimension at a time, the update is simple, even for complex problems. It is useful for in settings where there is low D or sparse structures. While highly impractical for deep learning settings, as D is high and we are not given the sparsity structure, setting a coordinate basis may be beneficial to accelerate convergence.

Our method primarily targets smooth objective functions, as is the case for most deep learning tasks. The specific challenges of non-smooth optimisation are outside its direct scope. We do note that the iterative improvements are shared, as our method generally improves in decreasing the objective function in each iteration. While our method does not target non-smooth objective functions, reformulating it into a smooth approximation may work.

3.4 Meta-Learning Discovery

A new way to solve optimisation problems is to consider abstracting away the problem. Meta-learning discovery aim to formulate algorithm search such that optimisation methods are found as a result.

Lion: Evo**L**ved Si**g**n M**o**me**n**tum (Lion) was discovered via a meta-learning approach (Chen et al., 2024). It uses momentum and the sign operation to update parameters.

$$\text{update} = \text{sign}(\beta_1 m_{t-1} + (1 - \beta_1) g_t) \tag{3.42}$$

$$\theta_{t+1} = \theta_t - \alpha \cdot \text{update} \tag{3.43}$$

$$m_t = \beta_2 m_{t-1} + (1 - \beta_2) g_t \tag{3.44}$$

Unlike adaptive methods, there is no second moment V_t, and so no normalisation on the first moment is performed. Thus, Lion contributes a fixed size update to θ at each

iteration, only scaling by the output of the sign function. This is similar to signSGD, mentioned in Sec. 3.2.2 under Sophia. Lion is advantageous given that it is simple and only tracks the momentum. This halves the memory requirement, making it more efficient. Lion gains up to a 2.3x speedup on AdamW (Chen et al., 2024). It outperforms AdamW on image classification, vision-language contrastive learning, diffusion modelling and language modelling and pre-training tasks (Chen et al., 2024). However, we note that this is the case only on transformer models that Lion is very effective. Further evaluations proving Lion's effectiveness on non-transformer based models are needed to confirm its effectiveness.

Our method is different entirely from the paradigm of meta-learning since we do not formulate optimisation as an algorithm search. Our method is similar to Lion since we use momentum, but we do not perform a signed update. We also do not use a learning rate or momentum hyperparameters, as previously mentioned. We use HVPs and a trust-region to approximate the local curvature, and our update step is a combination of the momentum, gradient, and other useful directions.

In this chapter, we have covered in detail the major works for optimisation in deep learning and how our work fits alongside them. We now introduce our work, KryBall, in the next chapter.

Chapter 4

The KryBall Optimisation Algorithm

In this chapter, we present our method, the KryBall optimisation algorithm. We first introduce the concept of the Saddle-Free-Newton that our method builds upon on in Sec. 4.1. We then present our algorithm, which includes core components such as Krylov subspace construction, the computation of the Saddle-Free-Newton, and a trust-region framework. This is detailed in Sec. 4.2. We then present the main algorithm in Sec. 4.3.

4.1 Saddle-Free-Newton

We build upon a key work in optimisation, the *Saddle-Free-Newton* (SFN) method (Dauphin et al., 2014). Recall in Sec. 3.2, we showed that the pure Newton method can be attracted to saddle points due to optimising in a non-convex setting. To address this problem, the SFN method was proposed (Dauphin et al., 2014). The SFN is essentially a regularised Newton method, similar to that of Damped Newton methods we saw in Sec. 3.2.1. However, instead of applying step-size or regularisation damping, the SFN method instead applies the $|\cdot|$ (abs) function onto the *eigenvalues* of the Hessian Dauphin et al. (2014). This results in an adjusted Hessian which is *guaranteed* to be positive semi-definite, and thus avoids being attracted to saddle points.

Definition 32 (Saddle-Free-Newton Step)**.** Given a Hessian H and a gradient g evaluated at our current parameters θ, we define the Saddle-Free-Newton step $\Delta\theta_{SFN}$ as:

$$H = V\Lambda V^T \tag{4.1}$$

$$H^{SFN} = V|\Lambda|V^T \tag{4.2}$$

$$\Delta\theta_{SFN} = -(H^{SFN})^{-1}g, \tag{4.3}$$

where Λ is the diagonal matrix of eigenvalues, and $|\cdot|$ is the element-wise absolute value applied to each element of Λ.

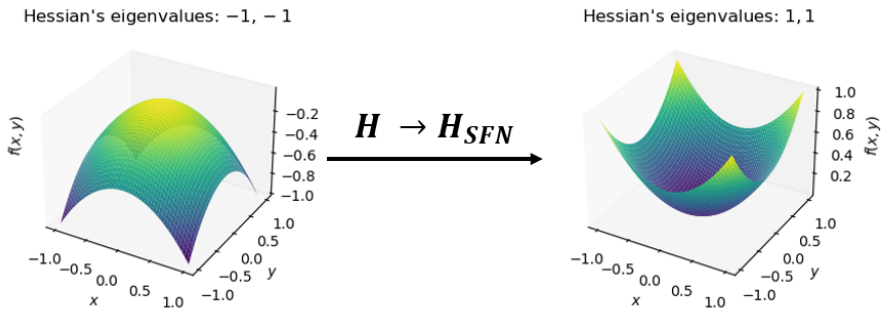


Figure 4.1: The crux of the SFN step: A non positive-semi-definite H is now be guaranteed to be positive semi-definite as we manipulate the curvature information by applying $|\cdot|$ to the eigenvalues Λ of H to form H^{SFN}.

We use the notation $\|H\|$ and H^{SFN} interchangeably to refer to the reconstructed Hessian that is now guaranteed to be positive semi-definite. We illustrate this change in Fig. 4.1.

As an example, consider once again our 2D horse saddle function from Sec. 2.3.3. Recall we split the problem into two cases to optimise: $g(x) = x^2$ and $h(y) = -y^2$. We computed that $H_g = \nabla^2 g(x) = 2$ in the x-direction and $H_h = \nabla^2 h(y) = -2$ in the y-direction given our starting iterates $(x_0, y_0) = (1.5, 1.5)$. Our problem with Newton's method was that H_h had a negative eigenvalue, which made our Hessian indefinite and thus in the y-direction, we moved towards the local maximum instead. Now, we consider the SFN step and its behaviour.

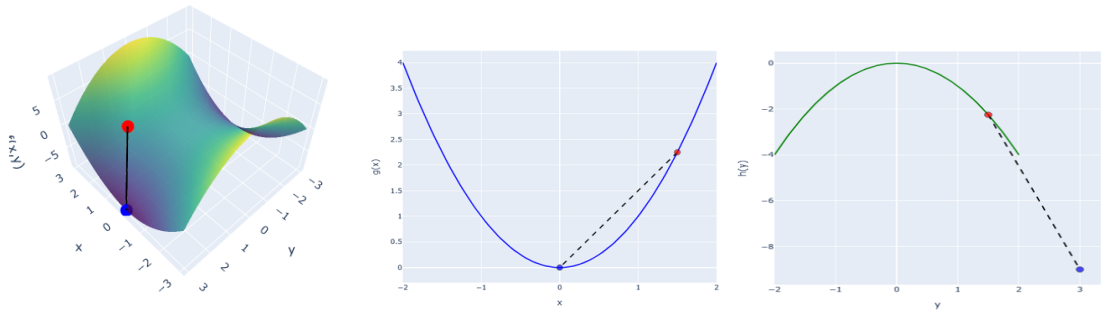
- In the x-direction, we have $H_g = 2$. Applying $|\cdot|$ to the eigenvalues, we get $H^{SFN} = 2$. This is the same as the Newton step, which behaves correctly for this case, so $x_1 = 0$.
- In the y-direction, we have $H_h = -2$. Applying $|\cdot|$ to the eigenvalues, we get $H^{SFN} = 2$. Now, we have that

$$y_1 = y_0 + \Delta y = y_0 - (H^{SFN})^{-1}\nabla h(y_0) = y_0 - \frac{1}{2}\cdot(-2y_0) = y_0 + y_0 = 2y_0. \quad (4.4)$$

  Given we have $y_0 = 1.5$, our next iterate is $y_1 = 2 \cdot 1.5 = 3$. This is the correct behaviour, and we have successfully moved *away* from the saddle point and follow the negative gradient direction.

As a result, our next set of iterates $(x_1, y_1) = (0, 3)$ now evaluates to $f(0, 3) = -9$. Thus, we have moved away from the saddle point. Subsequently applying more SFN steps will optimise f more until we diverge, since the function is unbounded. We illustrate our discussion in Fig. 2.12.

However, while the SFN method addresses the issue, the core limitation is still the computational intractability. To compute the SFN, we must do an eigendecomposition



(a) The SFN step on the 2D horse saddle function. (b) The SFN step on the x-axis component (c) The SFN step on the y-axis component,

Figure 4.2: SFN on the classic 2D horse saddle function. Unlike the Newton step, the SFN step is not attracted to the saddle points. Given a starting point $(1.5, 1.5)$ in red, the SFN step is able to move away from the saddle point and decrease the objective function. For both the x-axis and y-axis components, the SFN step is able to move in the correct direction.

of the Hessian, apply our abs function, reconstruct it, and then take the inverse. This sequence of steps is infeasible for deep learning.

4.2 Components of KryBall

This raises the question: how can we compute the SFN step efficiently, especially given the proliferation of saddle points? We thus introduce KryBall, a method that extends the SFN method and uses its advantages while being computationally tractable for deep learning. KryBall is a second-order method that uses Krylov subspaces, the SFN step, and a trust-region framework. At its core, KryBall consists of two main components:

- **SFN Computation**: The efficient computation of the SFN step direction.
- **Trust-Region Framework**: The integration of the SFN step and other useful components into a generalised trust-region framework.

In this section, we detail these two core components. We first discuss how to compute the SFN step direction in Sec. 4.2.1. We then present how we integrate this into a trust-region framework in Sec. 4.2.2. Finally, we present our main algorithm in Sec. 4.3.

4.2.1 Computation of the Saddle-Free-Newton

We introduced Krylov subspaces in Sec. 2.4.2. We now use them to their best ability in our method. Instead of computing and storing the full Hessian, we can generate a Krylov subspace with the Hessian H and the gradient g. This gives us

$$\mathcal{K}_M(H, g) = \text{span}\{g, Hg, H^2g, \dots, H^{M-1}g\}. \tag{4.5}$$

Recall that the Arnoldi iteration not only generates an orthonormal basis Q_M for $\mathcal{K}_M(H, g)$, but also the coefficient matrix H_M. We know that this is a projection of the original space that we work in. That is, we have the following relationship,

$$Q_M^T H Q_M = H_M, \tag{4.6}$$

where H_M is the *projected Hessian*. Thus, we now have a low-dimensional approximation of our original space H through the projected Hessian H_M. We then use this to compute the SFN step direction.

$$H_M = V_M \Lambda_M V_M^T \tag{4.7}$$

$$H_M^{SFN} = V_M |\Lambda_M| V_M^T \tag{4.8}$$

$$\Delta\theta_{SFN} = -(H_M^{SFN})^{-1} q_0, \quad q_0 = \frac{g}{\|g\|}, \tag{4.9}$$

The gradient projected into this subspace is $g_M = Q_M^T g_t$. Since $q_0 = g_t / \|g_t\|_2$, g_M is simply $\|g_t\|_2 e_1$, where e_1 is the first standard basis vector in $\mathbb{R}^M$. Note that we assume H_M is symmetric here even though we have not explicitly assumed this in our Arnoldi iteration. For this case, we do not use the last row of H_M and thus our H_M goes from being a $M+1 \times M+1$ matrix to a $M \times M$ matrix. We do this since we only need the first M eigenvalues of H_M to compute the SFN step. After we compute the SFN step, we then project it back into the original space as follows,

$$\Delta\theta_{SFN} = Q_M \Delta\theta_{SFN}. \tag{4.10}$$

We now have our SFN direction that we can use to optimise our model. In KryBall, we offer an almost one to one implementation of this. However, we make one slight modification and add a clipping parameter ϵ for numerical stability. This ensures that the step direction and norm are not too large, and thus we do not diverge from the original space. We perform the clipping after we apply the abs operator to the eigenvalues. Formally, we write this as,

$$H_M^{SFN} = V_M \text{clip}(|\Lambda_M|, \epsilon) V_M^T \tag{4.11}$$

$$\Delta\theta_{SFN} = -(H_M^{SFN})^{-1} q_0, \tag{4.12}$$

where the clipping function $\text{clip}(x, \epsilon)$ clips any eigenvalues of H_M that are less than ϵ to ϵ.

4.2.2 N-Dimensional Subspace Optimisation

Having computed the SFN direction $\Delta\theta_{SFN}$, we now integrate into a robust step selection procedure using N-dimensional subspace optimisation. We introduced briefly 2D subspace optimisation in Sec. 2.4.3. The general principles are extended to the N-dimensional case.

We define a search subspace $\mathcal{S}_k$ as a k-dimensional subspace that is spanned by a set of k direction vectors. These vectors are chosen to capture both first-order and approximated second-order information about the optimisation landscape. Typically, the basis vectors for $\mathcal{S}_k$ include:

- The normalised current gradient: $g_t/\|g_t\|_2$.
- The normalised SFN direction: $\Delta\theta_{SFN}/\|\Delta\theta_{SFN}\|_2$, as computed in Sec. 4.2.1.
- The normalised momentum vector: $z_t/\|z_t\|_2$.

Let $X_t = [x_1, x_2, \ldots, x_k]$ be the matrix whose columns are these k orthonormalised basis vectors. Any potential step p_t from the current iterate θ_t is therefore restricted to this subspace, meaning $p_t = X_t\alpha$ for some coefficient vector $\alpha \in \mathbb{R}^k$.

The core of the trust-region method is to minimise a quadratic model $m_t(p)$ of the loss function $L(\theta_t + p)$ within this subspace $\mathcal{S}_k$, subject to the constraint that the step p_t remains within a trust region of radius Δ_t. The quadratic model is given by:

$$m_t(p) = L(\theta_t) + g_t^T p + \frac{1}{2} p^T H_t p. \tag{4.13}$$

Substituting $p = X_t\alpha$, the subproblem becomes finding α^* that solves:

$$\min_{\alpha\in\mathbb{R}^k} \quad (X_t^T g_t)^T \alpha + \frac{1}{2}\alpha^T (X_t^T H_t X_t)\alpha \tag{4.14}$$

$$\text{subject to} \quad \|X_t\alpha\|_2^2 \leq \Delta_t^2. \tag{4.15}$$

Let $b_k = X_t^T g_t$ be the projection of the gradient onto the subspace directions, and $A_k = X_t^T H_t X_t$ be the projection of the Hessian onto the subspace. We can compute A_k efficiently using HVPs $H_t x_i$ for each basis vector x_i in X_t. The objective function in Eq. (4.14) then simplifies to $b_k^T\alpha + \frac{1}{2}\alpha^T A_k \alpha$.

This k-dimensional trust-region subproblem is solved to find the optimal coefficients α^*. The resulting step in the full parameter space is $p_t^* = X_t\alpha^*$. We then follow suite with the standard trust-region algorithm and update our radius based on the reduction ratio. If we cannot ensure sufficient reduction, we reject the step and adapt our radius accordingly and try again. This ensures that the algorithm adapts to the reliability of the quadratic model, promoting stable and efficient convergence.

4.3 Main Algorithm

We now bring together the previously described components: the efficient computation of the SFN direction via Krylov subspaces, and its integration into an N-dimensional subspace optimisation using a trust-region framework. We present the complete KryBall optimisation algorithm in Algorithm 4.

The algorithm iteratively builds a low-dimensional quadratic model of the objective function. It leverages curvature information approximated via the SFN direction computed

Algorithm 4: The KryBall Optimisation Algorithm

Input: Initial parameters θ_0, objective function $L(\theta)$, Krylov dimension M, trust-region parameters (initial radius Δ_0, max radius Δ_{max}, adaptation rules η_1, η_2), Krylov refresh rate $r_{refresh}$, momentum factor β

Output: Optimised parameters θ^*

Initialise $\theta \leftarrow \theta_0$, trust-region radius $\Delta \leftarrow \Delta_0$
Initialise momentum vector $z \leftarrow 0$
for $t = 0, 1, 2, \ldots$ *until convergence* **do**
    Compute current gradient $g_t = \nabla L(\theta_t)$
    if $t \pmod{r_{refresh}} == 0$ **then**
        Construct orthonormal Q_M and H_M from the Arnoldi iteration for $\mathcal{K}_M(H_t, g_t)$
        Compute and normalise $\Delta\theta_{SFN}$
    Define search subspace directions $X_S = [g_t/\|g_t\|_2, z_t/\|z_t\|_2]$
    if $t \pmod{r_{refresh}} == 0$ **then**
        Add SFN direction $\Delta\theta_{SFN}$ to X_S
    else
        Add basis vectors of Q_M to X_S
    Project gradient $b_k = X_t^T g_t$
    Project Hessian $A_k = X_t^T H_t X_t$
    Solve the trust-region subproblem to find α^*
    Compute candidate step $p_t = X_t \alpha^*$
    Compute the reduction ρ_t
    Accept or reject the step based on ρ_t and $\|p_t\|$
    Update Δ_{t+1} based on ρ_t, $\|p_t\|$, and Δ_{max}
    Update momentum vector $z_{t+1} \leftarrow \beta z_t + g_{t+1}$
    $\theta \leftarrow \theta_{t+1}$, $\Delta \leftarrow \Delta_{t+1}$
return θ

within a Krylov subspace. An optimal step is then determined within this subspace using the trust-region framework, which adaptively adjusts the step size to ensure reliable progress.

Note that we have a refresh parameter $r_{refresh}$ which controls how often we update our Krylov basis. This is primarily for efficiency reasons, as it is expensive to compute the Arnoldi iteration every single time. As such, when we do not compute the Arnoldi iteration, we leverage the basis vectors previously computed and add these into our trust-region. In practice, we find that an appropriate choice of $r_{refresh}$ is usually performant even if it is not set to 1.

In this chapter, we detailed the innerworkings of our method, the KryBall optimiser. We discussed the need for methods like the SFN. We then introduced our core ideas of efficiently computing the SFN and integrating it into a trust-region framework, resulting in our algorithm. We now present an empirical evaluation of KryBall in the next chapter, assessing its performance in a range of tasks.

Chapter 5

Evaluation and Analysis

In this chapter, we present our evaluation, findings, and analysis of our method, the KryBall optimisation algorithm. We start by discussing the implementation of KryBall in 5.1. We then present the experimental setup in 5.2, which includes the datasets, models, metrics, hyperparameter settings, and tasks to be evaluated. This is followed by our results in 5.3, where we include our empirical evaluations and sensitivity analysis. We then interpret our findings and do some further analysis in 5.4. Finally, we discuss the limitations of our method in 5.5.

5.1 Implementation

Most machine learning workflows and tasks are implemented entirely in Python, and leverage the PyTorch library (Paszke et al., 2017). This is because PyTorch has efficient low-level kernels that allow for automatic differentation and tensor operations (Paszke et al., 2017). We design our optimiser to be integrated seamlessly with PyTorch. As such, KryBall can act as a drop-in replacement for other PyTorch optimisers in deep learning tasks.

5.1.1 Adhering to the PyTorch Optimiser API

Our implementation adheres to the PyTorch optimiser API while simplifying the standard training workflow. In a typical training setup in PyTorch, users must explicitly call the following functions (Paszke et al., 2017):

- `.backward()` to create the computation graph and the associated gradients.
- `.zero_grad()` to clear previous gradients.
- `.step()` to perform the optimisation step and update the model parameters.

We instead consolidate these into just a single `step` function. This makes it cleaner than the current workflow while maintaining full compatibility with the PyTorch ecosystem.

In addition, we provide an interpretable way to track necessary information. In PyTorch, optimiser parameters such as the momentum and adaptive learning rates are associated with individual parameter blocks and buried within the optimiser's internal state dictionaries (Paszke et al., 2017). This makes it difficult to understand how they behave during training. Our implementation exposes these quantities directly as accessible vectors, allowing researchers to easily inspect gradients, momentum terms, and other optimisation variables. This facilitates deeper insights into the optimisation process and enables more effective understanding and debugging.

5.1.2 Function Integration with PyTorch

We also introduce a novel test suite that integrates classic functions (Rosenbrock, 2D Saddle, Monkey Saddle etc.) with PyTorch. Currently, most existing implementations that offer direct optimisation of just functions do not integrate with PyTorch. Instead, they all use numerical solvers and approximate the first and second-order derivatives. This is because PyTorch workflows are primarily for deep learning, and simple function optimisation is not a focus. To our knowledge, there exists only a few libraries that try and integrate function optimisation with deep learning. However, these are outdated, lack maintainability, and do not use modern PyTorch techniques such as model JIT compilation to improve speed. We thus implement an easy way to optimise any function with PyTorch optimisers such as SGD, Adam and LBFGS that adheres to the standard training loop. We show an example in Algorithm 5.

Algorithm 5: Optimising a function with PyTorch optimisers

```
1 T = 100
2 model = Rosenbrock()
3 optimiser = torch.optim.LBFGS(model.parameters())
4 for t = 1, 2, ..., T do
5     optimiser.zero_grad()
6     loss = model()
7     loss.backward()
8     optimiser.step()
```

5.1.3 The Trust-Region Framework

A core contribution that we present is a novel implementation of N-dimensional subspace optimisation that solves the trust-region subproblem. To our knowledge, this is the first general-purpose implementation tailored for deep learning applications within the PyTorch ecosystem. Many trust-region frameworks either employ the dogleg method, or 2D subspace optimisation, but also are not integrated with PyTorch. We thus implement

a general-purpose N-dimensional subspace optimiser that can be used with any PyTorch optimiser, and more generally any task that involves the trust-region subproblem.

5.1.4 Loss Landscape Sampling

The final part of our contributions that we implement is the ability to inspect the loss landscape of deep learning training tasks. We implement a sampler that can be used to sample trajectories from the loss landscape of a deep learning model. Given a particular point in training, our sampler can investigate the local geometry by performing an exhaustive line search along the optimiser's step direction. This allows researchers to visualise and analyse the characteristics of the loss landscape in the vicinity of the optimisation trajectory for a particular optimiser. We use this to analyse the trajectory of optimisers for different deep learning tasks in Sec. 5.4.

5.2 Experimental Setup

To evaluate KryBall, we must first define a range of tasks that accurately reflect the challenges faced by deep learning models. In this section, we define these tasks with respect to the datasets, models, metrics, and hyperparameter tuning setups. We consider the following tasks as our suite: ill-conditioned functions, binary classification, and image classification.

5.2.1 Baselines

Before we introduce our tasks, we first define our baselines. We select MSGD and Adam as our baseline optimisers. These are state-of-the-art first-order optimisers that we find outcompetes other optimisers in literature. MSGD is a simple and foundational approach that is well-suited for many deep learning tasks. Adam is more of a modern and practical baseline, and has been the standard optimiser in practice ever since it was introduced. We also use LBFGS as a second-order baseline, as it exhibits good performance across many tasks and is also a popular optimiser in practice. However, we note that we only use LBFGS where it is tractable, and so for the image classification task in Sec. 5.2.4, we do not use LBFGS to remain consistent.

5.2.2 Task 1: Ill-Conditioned Function Optimisation

The first set of tasks involves minimising ill-conditioned objective functions. We choose the Rosenbrock function and implement a noisy, stochastic variant of it, and the Rahimi-Recht function, which we briefly mentioned in Sec. 2.2.4 of Chapter 2.

Stochastic Rosenbrock Function

We implement a stochastic variant of the Rosenbrock function from Henriques et al. (2019). We have that $\mathcal{R} : \mathbb{R}^2 \to \mathbb{R}$:

$$\mathcal{R}(x, y) = (1 - x)^2 + 100\epsilon_i(y - x^2)^2, \tag{5.1}$$

where at each evaluation of the function, a noise sample ϵ_i is drawn from a uniform distribution $\mathcal{U}[\lambda_1, \lambda_2]$ with $\lambda_1 \leq \lambda_2$. When $\lambda_1 = \lambda_2 = 1$, we can recover the deterministic Rosenbrock function with its well-known minimum at $(x, y) = (1, 1)$. To assess robustness to noise, we compare each optimiser on both the deterministic formulation and two stochastic variants with different noise regimes.

Rahimi-Recht Function

The second function is an ill-conditioned linear regression problem introduced by Rahimi and Recht (Rahimi and Recht, 2017). This involves training a neural network with two linear layers to approximate a linear map with a high condition number. The objective function is

$$\mathcal{L}(W) = \|\hat{y} - y_{true}\|^2 = \|W_L W_{L-1} \cdots W_1 x - Ax\|^2, \tag{5.2}$$

where A is an ill-conditioned matrix with condition number κ, and W_i are the weight matrices of the linear network. We construct the true linear map $A_{true} \in \mathbb{R}^{m \times d}$ with singular values linearly spaced between 1 and κ (Rahimi and Recht, 2017). We generate $n = 1000$ random input samples and compute the corresponding outputs using this map. The behaviour of this function is that as we get closer to the minima, it becomes increasingly ill-conditioned. Specifically, the eigenvalues of the Hessian tend towards infinity.

Evaluation Setup and Metrics

We evaluate MSGD, Adam, LBFGS, and KryBall on the Rosenbrock and the Rahimi-Recht functions with the following chosen ill-conditioning parameters:

- **Deterministic Rosenbrock**: $\lambda_1 = \lambda_2 = 1$.
- **Stochastic Noisy Rosenbrock**: $\lambda_1 = 0$, $\lambda_2 = 1$.
- **Stochastic Noisy Rosenbrock**: $\lambda_1 = 0$, $\lambda_2 = 3$.
- **Slightly Ill-Conditioned Rahimi-Recht**: $\kappa = 10$.
- **Highly Ill-Conditioned Rahimi-Recht**: $\kappa = 1e5$.

Each optimiser is tuned on each function for 20 sweeps, where each sweep has 100 epochs. We describe in detail our hyperparameter tuning setup in Sec. 5.2.5. We then use the best performing configuration for each optimiser, and then run all optimisers with a budget of 2000 epochs until they converge within an ϵ of the solution or use up the computational budget. For Rosenbrock, we choose $\epsilon = 1 \times 10^{-4}$ and for Rahimi-Recht

we choose $\epsilon = 1 \times 10^{-2}$. Here, we are measuring the optimisers ability to converge to the solution while handling noise and ill-conditioning.

5.2.3 Task 2: XOR Classification

The next task we consider is a simple case of binary classification with the XOR function. This is a good task to bridge between the ill-conditioned functions and more complex recognition tasks as it introduces non-linearity. The XOR task cannot be linearly separated, and needs a non-linear activation function. Our dataset is simple and small. It includes all 4 combinations of inputs to the XOR function, a 2D vector of zeros and ones, and the corresponding labels in which they evaluate to.

Evaluation Setup and Metrics

For this task, we use a simple MLP with 3 layers, consisting of one input layer, a hidden layer with a hidden dimension of size 8, and an output layer. Our activation function is the Softplus function, and our loss function is the Binary Cross Entropy (BCE) loss (Mao et al., 2023). The BCE loss is defined as

$$\mathcal{L}(y, \hat{y}) = -\frac{1}{N} \sum_{i=1}^{N} \left[y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i) \right], \tag{5.3}$$

and measures the difference between the predicted probability distribution and the actual distribution which consists of the ground truth labels for a binary classification task (Mao et al., 2023). This is useful for the XOR case since our output is binary.

We test MSGD, Adam, LBFGS, and KryBall on this task. Each optimiser is tuned for 20 sweeps, each with 100 epochs like the previous task. We use the best performing parameters, and then run each optimiser on the XOR task with a budget of 100 epochs. Given this task is quite simple, and our chosen model is more than capable, we measure the number of epochs it takes to converge. Our convergence criteria here is to reach perfect accuracy, that is an accuracy of 1.0, and completely minimise the loss function (reach 0 loss). This means we have perfectly predicted the XOR problem. In practice, most models and most optimisers are capable of this, and thus we instead focus on convergence speed.

5.2.4 Task 3: Image Classification

Our third and final task is a suite of image classification problems. Here, given an image I that is usually either a binary image or an RGB image, we want to classify it into one of C classes. We thus use a variety of datasets to formulate our image classification problem.

MNIST-1D

MNIST-1D is a scaled down one-dimensional version analogue of the classic MNIST handwritten digit dataset (Greydanus and Kobak, 2020). It consists of 40-dimensional time series signals that represent the handwritten digits from 0 to 9. These are constructed through procedural generation and involve random transformations such as padding, translation and scaling operations (Greydanus and Kobak, 2020). There are 5000 total samples in MNIST-1D, where 4000 are allocated for training and 1000 for testing (Greydanus and Kobak, 2020). Each example in MNIST-1D is labeled with its corresponding digit class (0–9). This estbalishes a 10-class classification problem where the ground truth represents the original digit template from which each signal was derived.

CIFAR-10

CIFAR-10 is widely used dataset for image classification. It consists of 60000 RBG images that are of size $32 \times 32 \times 3$ (Krizhevsky et al., 2009). These are distributed across 10 distinct object classes: irplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. The dataset comprises 50000 training images and 10000 test images (Krizhevsky et al., 2009). Each class contains exactly 6000 images, where the train and test split is 5000 and 1000 respectively. The ground truth for this dataset is a single-label classification that indicates the primary object category present in each image (Krizhevsky et al., 2009). In CIFAR-10, all images are labelled. We note that a superset of CIFAR-10 is CIFAR-100, which instead contains 100 classes.

Evaluation Setup and Metrics

We evaluate our optimisers on the MNIST-1D and CIFAR-10 datasets. We categorise our experiments into three different groups which are based on the model we use. We now describe each of these groups.

- **MNIST-1D MLP**: We use the simple MLP we had from the XOR task. Here, we modify the hidden dimension to be of size 256. This is consistent with the original implementation for MNIST-1D in (Greydanus and Kobak, 2020), and offers a good baseline for this task.
- **CIFAR-10 CNN**: The first model we use to evaluate on CIFAR-10 is a 3-layer Convolutional Neural Network (CNN) with 2 fully connected layers. Here, we use three convolutional layers with 32, 32 and 64 output channels, each using 5×5 kernels with padding. Each convolution is followed by batch normalisation, our choice of activation and average pooling. We then follow with two fully connected layers to predict out output.
- **CIFAR-10 ResNet-18**: We also evaluate using a stronger model, a ResNet-18, introduced by (He et al., 2016). This deeper architecture employs residual connections to enable training of networks with 18 layers, where each layer consists

of four residual blocks with skip connections. ResNet-18 is a baseline architecture for CIFAR-10, and offers good performance in which we can evaluate our optimisers on.

For each model, we use the Softplus activation as done previously.

For this image classification task, we employ the Categorical Cross Entropy (CCE) loss function (Mao et al., 2023). This generalises the BCE loss to multi-class classification problems, extending it to handle C classes (Mao et al., 2023). This is done by comparing the predicted probability distribution over all classes with the true one-hot encoded label distribution. The CCE loss is defined as

$$\mathcal{L}(y, \hat{y}) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{c=1}^{C} y_{i,c} \log \hat{y}_{i,c}. \tag{5.4}$$

Here, N is the number of samples, C is the number of classes, and $y_{i,c}$ is the true label, where it is 1 if sample i belongs to class c or 0 otherwise, and $\hat{y}_{i,c}$ is the predicted probability for sample i belonging to class c. CCE measures the difference between the predicted probability distribution and the actual distribution. This makes it well-suited for CIFAR-10 with given its 10 distinct object categories.

Our metrics for this task are the test accuracy and the training loss. Specifically, we look at the peak test accuracy and final test accuracy achieved during training. We also consider the final training. This shows how well our optimiser is able to train these different models on different datasets.

Training Setup

Our training setup is distinct from other tasks. Here, we sample mini-batches during training since evaluating on the whole dataset is infeasible. This is unlike our previous tasks, which were not highly parameterised. We now detail the batch size and the number of epochs for each task.

- **MNIST-1D MLP**: The batch size is 512 and we train for 100 epochs.
- **CIFAR-10 CNN**: The batch size is 256 and we train for 40 epochs.
- **CIFAR-10 ResNet-18**: The batch size is 128 and we train for 40 epochs.

5.2.5 Hyperparameter Tuning Setup

All of our experiments are tuned with Bayesian hyperparameter search combined with the Hyperband early stopping algorithm (Li et al., 2018). Hyperband is a bandit-based algorithm that allocates resources to promising configurations while eliminating poorly performing ones early in training (Li et al., 2018). We choose a halving factor $\eta = 3$, which eliminates the worst-performing configurations, retaining only the top $\frac{1}{3}$ of candidates at each successive halving round. This significantly reduces computational

cost by avoiding full training of suboptimal hyperparameter combinations. We set a minimum iteration threshold of 10 epochs before any configuration can be terminated, ensuring sufficient training time for meaningful performance evaluation.

- **MSGD**: Learning rate $\alpha \in [1 \times 10^{-4}, 1]$ under a log uniform distribution, momentum $\beta \in [0.8, 0.99]$ under a uniform distribution.
- **Adam**: Learning rate $\alpha \in [1 \times 10^{-4}, 1]$ under a log uniform distribution, betas $\beta_1 \in [0.9, 0.999]$ and $\beta_2 \in [0.99, 0.99999]$ under a uniform distribution.
- **LBFGS**: Learning rate $\alpha \in [1 \times 10^{-4}, 1]$ under a log uniform distribution.
- **KryBall**: Krylov subspace size $k \in [1, 10]$ under an integer uniform distribution, krylov refresh rate $r_{refresh} \in [1, 10]$ under an integer uniform distribution.

5.2.6 Experimental Infrastructure

We run our evaluations and experiments on a single machine that consists of an NVIDIA RTX 3070 GPU with 8GB VRAM, and a Ryzen 5 3600 CPU with 6 cores, with a clock speed of 3.6 GHZ. We also have 32GB DDR4 memory of available. For our software, we use Python 3.12.4, and PyTorch 2.4.0 on a Linux Subsystem. This is complemented by CUDA version 12.6. We use the Weights and Biases platform to track and run all of our experiments and tune our hyperparameters. We run our experiments with maximum GPU memory usage, and our GPU utilisation was monitored to be around 0.30 in all runs. This is consistent with GPU utilisation rate for mainstream machine learning workloads.

5.3 Results

In this section, we present our empirical evaluations of KryBall. We present the comparisons of KryBall with other optimisers for each of the tasks we have defined in Sec. 5.2.2, Sec. 5.2.3, and Sec. 5.2.4. We then present our sensitivity analysis in Sec. 2.3.3. For each task, we discuss the results and analyse our findings.

5.3.1 Task 1: Ill-Conditioned Function Optimisation

Table 5.1 summarises our results of optimising ill-conditioned functions. Here, we present the epochs until convergence, where we defined our criteria earlier in Sec. 5.2.2. We see that KryBall and LBFGS outclass MSGD and Adam when optimising the Rosenbrock function. Specifically, LBFGS converges the fastest regardless of the Rosenbrock variant. This is followed by KryBall, which is competitive with LBFGS. However, we note that LBFGS represents the *ideal convergence* in practice, since for each iteration it evaluates the function multiple times and backtracks. This is contrast to our method, which only evaluates the function once. Thus, we see that our method is competitive in practice with the ideal case.

Table 5.1: Epochs until convergence for each optimiser on the Rosenbrock and Rahimi-Recht functions. The best values denote fastest convergence, and are bolded and highlighted in green. Divergence is represented by N/A and highlighted in red.

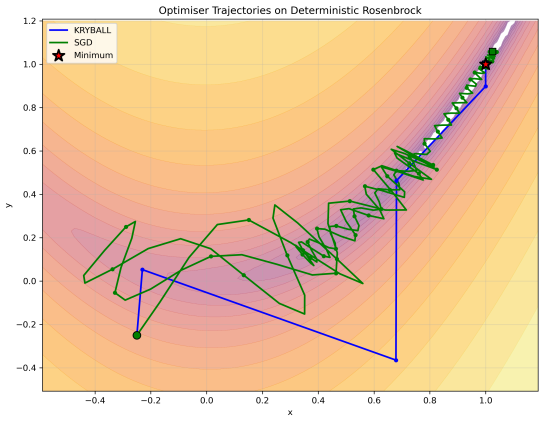
	Epochs Until Convergence ↓				
	Rosenbrock			**Rahimi-Recht**	
	$\mathcal{U}[1,1]$	$\mathcal{U}[0,1]$	$\mathcal{U}[0,3]$	$\kappa = 10$	$\kappa = 1 \times 10^5$
MSGD	248	624	1587	94	N/A
Adam	249	981	1978	242	1720
LBFGS	**5**	**5**	**5**	**6**	N/A
KryBall	**5**	13	16	51	**52**

This is further evidenced by the optimiser trajectories for the Rosenbrock variants in Fig. 5.1. KryBall and LBFGS are impacted little by the ill-conditioning and are able to converge. LBFGS is able to converge very quickly as it takes very large steps in the optimisation landscape. It also takes a near perfect straight line path and can correct itself. We see this in Fig. 5.1(a) as it overshoots the solution, but then re-evaluates and reaches the solution. This is a testament to LBFGS's backtracking and ability to evaluate the function multiple times. Our method also demonsrates this ability, and as it gets closer to the solution takes a path similar to LBFGS.

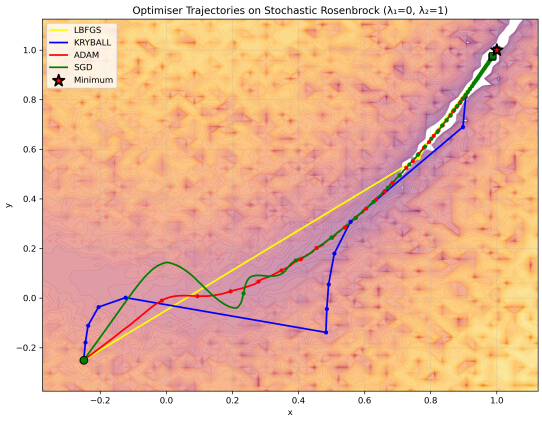
On the other hand, we see that MSGD and Adam make little progress and can be unstable. In Fig. 5.1(a), both MSGD and Adam become unstable when trying to take large steps. In Fig. 5.1(b) and Fig. 5.1(c), MSGD and Adam make very little progress per iteration. This shows that for ill-conditioned functions, MSGD and Adam are not able to take large steps and remain stable. This supports our hypothesis that ill-conditioned functions are a problem for first-order methods, as we discussed in Chapter 2.

For the Rahimi-Recht function, all optimisers are able to reach the minimum when the problem is slightly ill-conditioned. Fig. 5.2(a) and Fig. 5.2(b) show the loss curves for the two instances of the Rahimi-Recht function. Our problem of interest is Fig. 5.2(b), where the function is severely ill-conditioned. We see in Table 5.1 that only KryBall and Adam are able to converge, while MSGD and LBFGS diverge. Adam converges very slowly and exhibits instability as it approaches the minimum. However, it does not diverge like MSGD. We suspect that Adam's adaptive learning rate is advantageous for highly ill-conditioned functions, as it can adaptively scale each parameter. This is beneficial since, as we approach the minimum, the eigenvalues of the Hessian become increasingly large. MSGD's constant momentum scaling is thus insufficient to handle this eigenvalue spread, which explains why it diverges so rapidly.

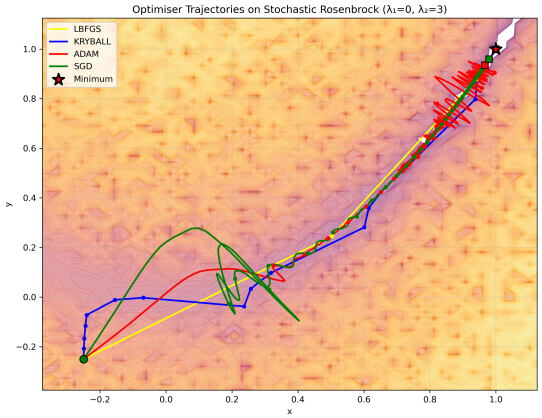
We also suspect LBFGS diverges because its limited-memory Hessian approximation becomes increasingly inaccurate when the function is highly ill-conditioned, leading to poor search directions. This is unlike KryBall, which maintains reliable curvature in-



(a) The optimiser trajectories on the deterministic Rosenbrock function with $\lambda_1 = 2$ and $\lambda_2 = 1$.



(b) The optimiser trajectories on the stochastic Rosenbrock function with $\lambda_1 = 0$ and $\lambda_2 = 1$.

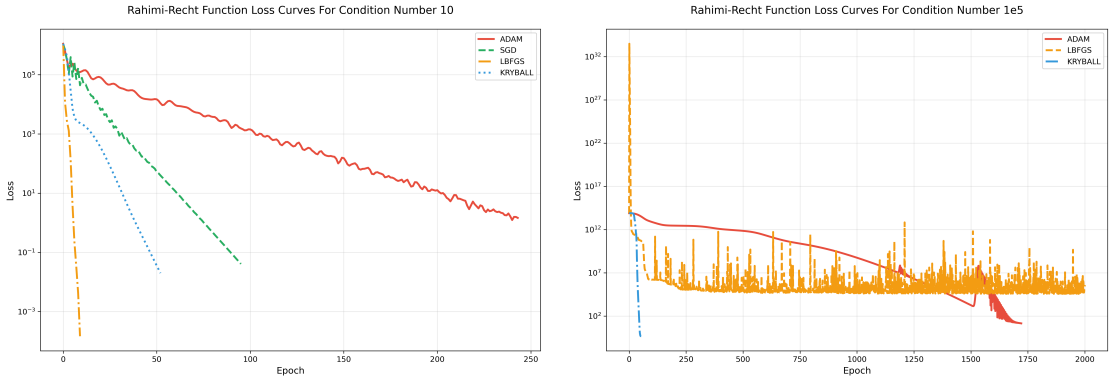


(c) The optimiser trajectories on the stochastic Rosenbrock function with $\lambda_1 = 0$ and $\lambda_2 = 3$.

Figure 5.1: The optimiser trajectories on the deterministic and stochastic Rosenbrock functions. KryBall and LBFGS are able to find the global minimum extremely quickly, and are not impacted by the ill-conditioning of the function. However, MSGD and Adam are impacted and converge much slower to the optimal solution.

formation. This allows our method to adapt the approximation quality and step sizes dynamically, enabling rapid convergence even in these highly ill-conditioned settings.

In summary, our method exhibits rapid convergence. It is competitive with the practical best possible convergence as illustrated by LBFGS. We also exhibit good stability and the ability to handle severe ill-conditioning, and are able to outperform state-of-the-art first-order methods on these problems.



(a) Loss curves for the Rahimi-Recht function with condition number $\kappa = 10$.

(b) Loss curves for the Rahimi-Recht function with condition number $\kappa = 1 \times 10^5$.

Figure 5.2: Loss curves for two instances of the Rahimi-Recht function. In Fig. 5.2(b), we have slight degree of ill-conditioning with $\kappa = 10$, and in Fig. 5.2(a), we have a very high degree of ill-conditioning with $\kappa = 1 \times 10^5$. We note that in Fig. 5.2(b), MSGD is not here as it diverged in every run very quickly.

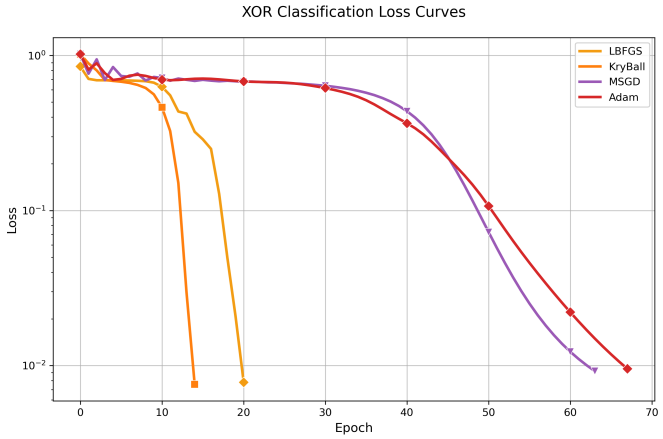


Figure 5.3: The loss curves for the XOR classification task. KryBall converges in 15 epochs, MSGD converges in 63 epochs and Adam converges to 67 epochs. LBFGS restricted to one evaluation per iteration converges in 20 epochs.

5.3.2 Task 2: XOR Classification

We now move onto our second task, XOR classification. Here, we evaluate the performance of our optimisers based on how quickly the converge to zero loss. We note that LBFGS is limited to one evaluation per epoch to make it fair for all optimisers. This is because for many deep learning tasks, we no longer have a best case pratical scenario or upper bound and need to ensure a fair comparison.

Fig. 5.3 shows the loss curves. We see that KryBall and LBFGS converge the quickest,

Table 5.2: Classification results for MNIST1D MLP, CIFAR10 CNN, and CIFAR10 ResNet-18 given Adam, MSGD and KryBall optimisers.

	Method	Final Training Loss ↓	Peak Test Accuracy ↑	Final Test Accuracy ↑
MNIST1D MLP	Adam	0.144	0.668	0.668
	MSGD	0.326	0.669	0.644
	KryBall	**0.001**	**0.685**	**0.685**
CIFAR10 CNN	Adam	**0.099**	0.782	**0.777**
	MSGD	0.111	0.777	0.771
	KryBall	0.135	**0.783**	0.762
CIFAR10 ResNet-18	Adam	**7.5e-5**	**0.843**	**0.842**
	MSGD	1e-4	0.832	0.829
	KryBall	0.002	0.832	0.822

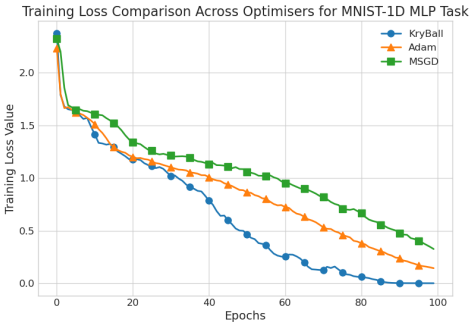
in less than 25 epochs. MSGD and Adam are also able to converge, but take longer. This shows that KryBall is able to generalise very quickly, and that the second-order approximation is helpful in identifying the optimal solution. As such, this shows in the simplest setting that our approximation is useful in tasks where there are non-linearities involved. As we transition into harder tasks, we hypothesise that our quadratic model will be able to generalise and add more insight into the optimisation landscape.

We note that while this task is simple and may seem trivial, it is a good baseline and a bridge to more complex tasks in the next section. We now move onto these tasks in the next section.

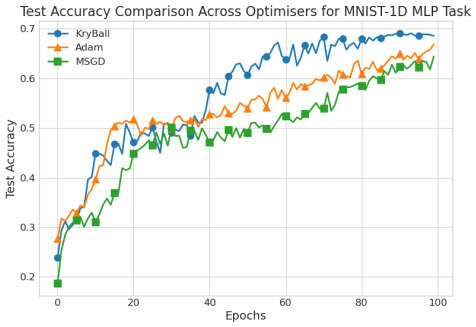
5.3.3 Task 3: Image Classification

We now move into the primary focus of our results, evaluating our optimiser on standard learning tasks that are the benchmark for optimisers. We evaluate our optimisers on three image classification tasks: MNIST-1D, CIFAR-10 CNN, and CIFAR-10 ResNet-18, which we described in Sec. 5.2.4. From here onwards, we do not use LBFGS since it becomes too expensive. Thus, our optimiser suite is now Adam, MSGD and KryBall. We present a summary of our results in Table 5.2. Our loss curves and accuracies are provided in Fig. 5.4.

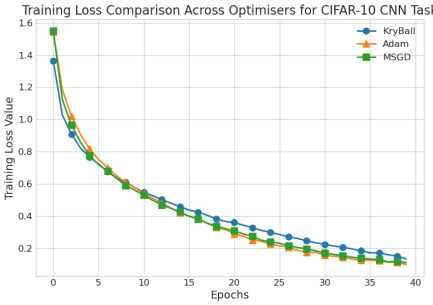
MNIST-1D MLP: KryBall demonstrates a clear advantage for the MNIST-1D MLP task. It is able to achieve the highest peak accuracy and the lowest final training loss. Fig. 5.4(a) and Fig. 5.4(b) show that KryBall converges quickly and makes a large improvement over Adam and MSGD, especially in 40 to 80 epoch range. We note that while a peak accuracy of 0.685 may not sound high, this contends with the best result on this specific model for MNIST-1D (Greydanus and Kobak, 2020). As such, we outperform Adam and MSGD in all metrics.



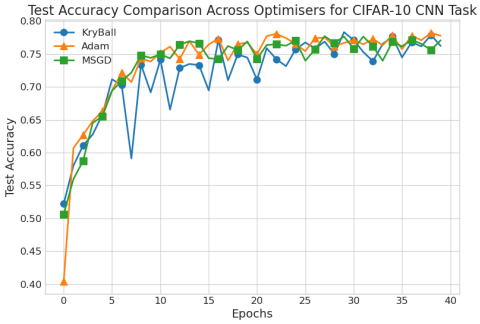
(a) Training Loss on MNIST-1D MLP



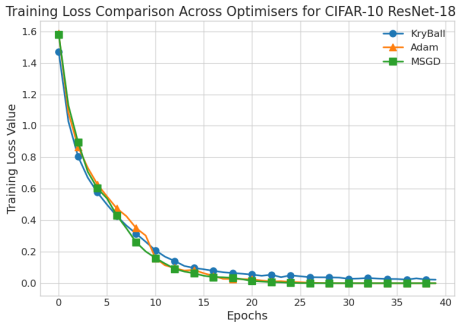
(b) Training Accuracy on MNIST-1D MLP



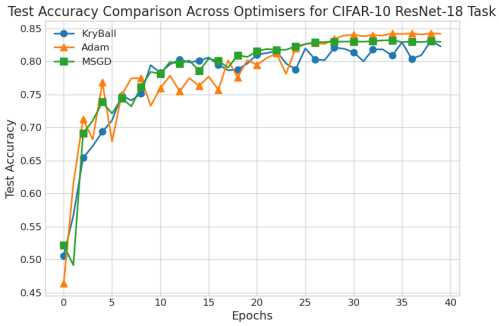
(c) Training Loss on CIFAR-10 CNN



(d) Training Accuracy on CIFAR-10 CNN



(e) Training Loss on CIFAR-10 ResNet-18



(f) Training Accuracy on CIFAR-10 ResNet-18

Figure 5.4: Loss curves and test accuracy comparisons among our image classification tasks consisting of MNIST-1D MLP, CIFAR-10 CNN and CIFAR-10 ResNet-18. Each task is evaluated with KryBall, Adam and MSGD.

CIFAR-10 CNN: Here, we see a more nuanced performance. Adam achieves the lowest final training loss, alongside the best final test accuracy. However, KryBall slightly outperforms Adam and records a higher peak test accuracy. The training loss curves in

Fig. 5.4(c) show Adam and MSGD reaching a slightly lower loss plateau than KryBall. In terms of test accuracy curves in Fig. 5.4(d), all three optimisers are competitive, with KryBall showing a strong peak but Adam and MSGD achieve slightly better peak test accuracy. A key observation here is that in Fig. 5.4(d), KryBall originally has a lower training loss, but then is surpassed by Adam and MSGD as training continue. During epochs 10 to 25, KryBall is also quite variant before converging in terms of test accuracy. This is unlike Adam and MSGD, who are consistent all the way.

CIFAR-10 ResNet-18: We see a similar comparison here. Adam has the best final training loss, and best peak test and final test accuracy. However, KryBall is generally competitive with Adam and MSGD. It achieves the same best test accuracy as MSGD, and only a slightly lower final test accuracy. Furthermore, Fig. 5.4(e) shows the same trend as in Fig. 5.4(c), where KryBall is initially competitive and achieves a lower training loss, but then is surpassed by Adam and MSGD as training continues. In Fig. 5.4(f), we see that KryBall is more stable than Adam and MSGD in the earlier epochs, but less stable in the later epochs and thus its performance suffers.

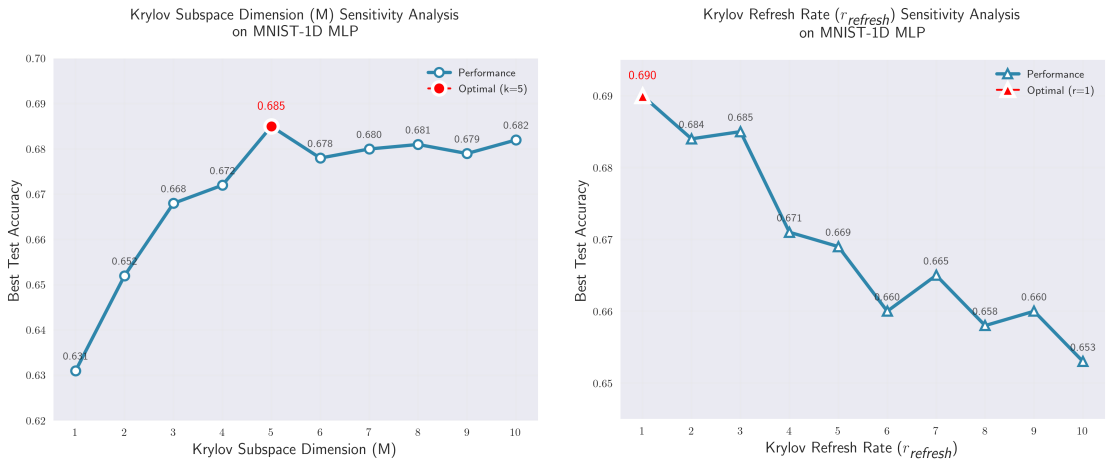
To recap, KryBall outperforms the state-of-the art Adam and MSGD on the MNIST-1D MLP task, but does not outperform them on the CIFAR-10 tasks. More interestingly, we see that KryBall achieves lower training loss initially in all tasks early on, but then is surpassed by Adam and MSGD as training continues for the CIFAR-10 tasks. Near the end of training, KryBall is less stable than Adam and MSGD and is more variant. We discuss this further inSec. 5.4.

5.3.4 Sensitivity Analysis

We now perform a sensitivity analysis of our method on the MNIST-1D MLP task. We choose this since it is our best performing task. Our sensitive analysis is on the Krylov subspace dimension M and the krylov refresh rate $r_{refresh}$. This is done by varying M and $r_{refresh}$ from their hyperparameter ranges and keeping all other hyperparameters constant. We present the results in Fig. 5.5.

Krylov Subspace Size (M): We see that the Krylov subspace size M is sensitive to the performance of KryBall. For low M, performance suffers since we are not able to approximate the Hessian well. As M increases, we obtain good performance. However, as we continue to increase M past a certain point (5), our performance plateaus. This is interesting, as usually a higher M results in a better approximation. We suspect that for $M = 5$ onwards, the dominant curvature information has already been captured. The new basis vectors that are generated are less relevant and do not add useful information, since we have already captured most of the curvature information previously. We present some further analysis about our approximation in Sec. 5.4.

Krylov Refresh Rate ($r_{refresh}$): The Krylov refresh rate is best when $r_{refresh} = 1$. This is because when $r_{refresh} = 1$, we compute the Krylov subspace at every iteration and are always using up to date information. As $r_{refresh}$ increases, we instead fall back to our approximation not at the current point in the optimisation landscape. This



(a) The sensitivity analysis of the Krylov subspace dimension M on MNIST-1D MLP.

(b) The sensitivity analysis of the krylov refresh rate $r_{refresh}$. on MNIST-1D MLP.

Figure 5.5: Sensitivity analysis of KryBall on the MNIST-1D MLP task. We vary the Krylov dimension M, the maximum trust region radius Δ_{max}, and the Krylov refresh rate $r_{refresh}$. The parameters are varied according to their hyperparameter ranges we defined in Sec. 5.3.4. The best performing point is marked in red.

is fine if we our information is not too outdated, as is the case for $r_{refresh} = 2$ and $r_{refresh} = 3$. However, as $r_{refresh} \geq 4$ our information is outdated and thus our quadratic model approximation is no longer accurate. This results in KryBall performing worse. In practice, we choose $r_{refresh} = 3$ as our default value.

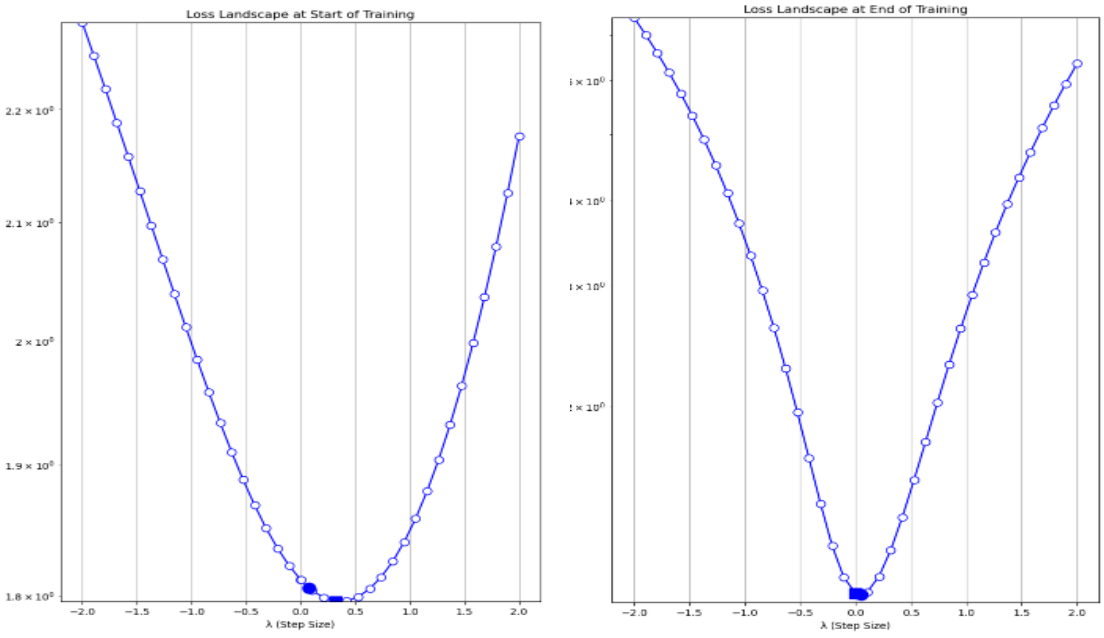
5.4 Discussion and Further Analysis

In this section, we present critical questions about the results we have observed and the choices we have made in evaluating our algorithm. We then answer these questions with hypotheses, experiments and further analysis.

Why do we see late epoch instability but good early epoch performance?

We saw in Sec. 5.3.3 that KryBall achieves good early epoch performance, but late epoch instability. We suspect this has to do with our quadratic approximation. In the early epochs, our quadratic model is accurate and a good approximation of the landscape. However, as training progresses, the landscape changes and we suspect that a quadratic model is less accurate.

Ma et al. (2022) state that the loss landscape of neural networks is multiscale, and that the quadratic model is only a local approximation of the landscape. Specifically, they find that in a neighbourhood of the local minima, the loss mixes a continuum of scales



(a) The loss landscape at the start of training (first epoch).

(b) The loss landscape near the end of training (last epoch).

Figure 5.6: The loss landscape of the MNIST-1D MLP model from the view of SGD.

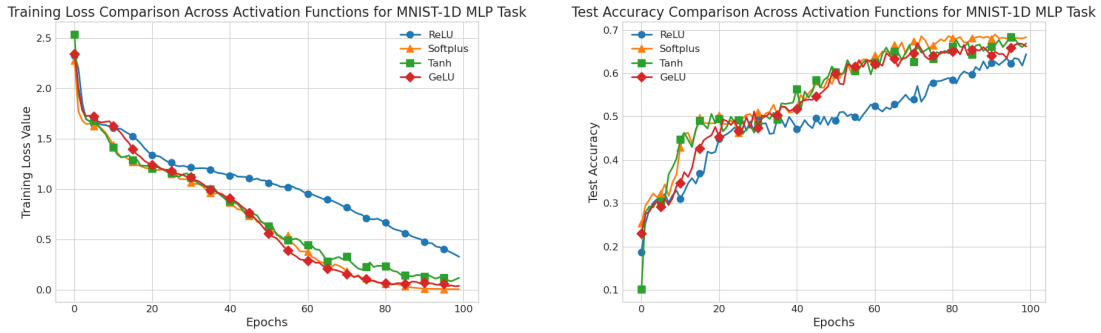
and instead grows subquadratically. As dimensionality increases, they state that the loss landscape instead shows several separate scales. This means that our quadratic model is not accurate in the presence of local minima. Given that the loss landscape is multiscale, and more importantly subquadratic near the local minima, this means that a first-order approximation may be better. In Fig. 5.4(d) and Fig. 5.4(f), it is plausible that the variance we see near the end of training is due to this.

To illustrate this, we perform a small experiment. We take the MNIST-1D MLP model with SGD and at distinct points in the training process, we sample the loss landscape. This is done through an exhaustive line search over $\lambda \in [-2, 2]$. We then plot what the loss landscape is at the start and end of training. This empirically shows a slice of the loss landscape from the perspective of SGD.

At the start of training, we see that the landscape is shaped exactly like a quadratic bowl in Fig. 5.6(a). However, near the end of training, that the landscape is approximately quadratic instead as seen in Fig. 5.6(b). It no longer is shaped exactly like a quadratic, but instead is stretched inwards. This agrees with Ma et al. (2022), and it is plausible that this illustrates the change in the loss landscape as training progresses.

What is the effect of the activation function on our optimiser?

All classification tasks in Sec. 5.3.2 and Sec. 5.3.3 are evaluated with the Softplus activation function. Now, we examine the effect of the activation function on our optimiser.



(a) Loss curves of the MNIST-1D MLP task with different activation functions using KryBall.

(b) Test accuracy of the MNIST-1D MLP task with different activation functions using KryBall.

Figure 5.7: The loss curves and test accuracy of the MNIST-1D MLP task evaluated with KryBall and ReLU, Softplus, Tanh and GeLU activation functions.

We evaluate the MNIST-1D MLP task with the following different activation functions using KryBall: ReLU, Softplus, Tanh and GeLU.

In Fig. 5.7, there is a stark distinction between the smooth activations and the non-smooth activation function ReLU. When using smooth activation functions, we reach close to zero training loss and our test accuracy is much better. However, the loss is much higher and our accuracy is lower when using ReLU. We have a good reason for this, which is that the quadratic approximation suffers in the presence of non-smooth activation functions. ReLU introduces non-differentiability at zero. This means that our quadratic model and trust-region framework, whose underlying assumptions are that we operate in smooth regions, is violated. Our method relise on meaningful HVPs for the approximation, but these are not well-defined at the transition points of ReLU units. This leads to poor curvature approximations. At this point, if we try to approximate the Hessian and the surrounding region using a quadratic model, we ultimately will not be able to. As such, it is better to use first-order methods that do not have these assumptions when using non-smooth activation functions.

Are our directions meaningful and distinct?

Song et al. (2025) show that gradients exhibit spurious alignment with the dominant eigenspace of the Hessian during training. Specifically, when the gradient appears to align with the top eigenspace, projecting it onto this subspace actually yields poor training progress Song et al. (2025). This finding suggests that our construction of distinct search directions is likely meaningful. Our method explicitly constructs directions from different sources. The SFN step, where we apply the abs function to the eigenvalues, alters the dominant subspace by ensuring positive definiteness. We hypothesis that this breaks the spurious alignment observed in standard training, making our dominant eigenspace directions genuinely useful rather than being correlated with

the gradient. The strong empirical performance of KryBall on ill-conditioned problems, where eigenspace structure is critical. This supports the hypothesis that our multi-directional search strategy captures meaningful aspects of the optimisation landscape that individual directions cannot provide alone.

How can we tell if our computed SFN step approximates the true SFN step?

The crux of the Krylov subspace approach is to approximate the Hessian with a low-dimensional projected Hessian that lets us compute the SFN step. We now quantify how to analytically measure this approximation quality through the reconstruction error

$$||r|| = ||z^* - \hat{z}||, \tag{5.5}$$

where z^* is the true SFN direction and $\hat{z}$ is our Krylov approximation:

$$\hat{z} = \sum_{k=0}^{m-1} \alpha_k H^k \hat{g}. \tag{5.6}$$

Using the eigendecomposition $H = V\Lambda V^T$ and letting $c = V^T\hat{g}$, we can express both directions in terms of the eigenvalues. The true SFN direction becomes

$$z^* = -V|\Lambda|^{-1}c, \tag{5.7}$$

while our Krylov approximation can be written as

$$\hat{z} = V\left(\sum_{k=0}^{m-1} \alpha_k \Lambda^k\right) c. \tag{5.8}$$

We can then minimise the reconstruction error by formulating it as a linear system.

$$r = z^* - \hat{z} = V(-|\Lambda|^{-1} - \sum_{k=0}^{m-1} \alpha_k \Lambda^k)c \tag{5.9}$$

By setting the middle term to zero, we get,

$$\Rightarrow -|\Lambda|^{-1} = \sum_{k=0}^{m-1} \alpha_k \Lambda^k \tag{5.10}$$

$$\Rightarrow -|\lambda_i|^{-1} = \sum_{k=0}^{m-1} \alpha_k \lambda_i^k \quad \text{for each eigenvalue } \lambda_i \tag{5.11}$$

$$\Rightarrow M\alpha = -1_r \quad \text{where } M_{i,k} = |\lambda_i|\lambda_i^k. \tag{5.12}$$

Solving this linear system for α and computing $||r||$ tells us how well our basis represents the ideal $|\Lambda|^{-1}$ operator. A small reconstruction error indicates that our limited Krylov subspace adequately spans the directions needed for an accurate SFN step. A large reconstruction error indicates that at the current point, our computed SFN step is not a good approximation of the true SFN step. Thus, we can use this to monitor the quality of our SFN step.

5.5 Limitations and Improvements

While KryBall demonstrates promising performance across several tasks, our evaluation reveals important limitations. We now discuss these limitations and how to address them.

5.5.1 Late-Epoch Instability

As demonstrated in our CIFAR-10 experiments, KryBall exhibits increased variance and instability in later training epochs. This occurs when the quadratic approximation becomes less accurate near local minima, where the loss landscape exhibits multiscale, subquadratic behaviour (Ma et al., 2022). We also saw empirical evidence of this when we sampled the optimiser trajectory in Fig. 5.6. One way to address this is to switch off the quadratic model and only use first-order information as we approach the local minima. This would likely require a heuristic where after a certain point in training, we switch off the quadratic model and only use either SGD or Adam. One such heuristic could be to use the gradient norm to determine when we've entered a region where the quadratic approximation breaks down. We can then switch to first-order methods when g falls below a threshold τ. We can also consider the decay rate of the gradient norm to determine when we should switch. We discuss these further in Sec. 6.2.1.

5.5.2 Dependence on Smooth Activation Functions

A fundamental limitation of KryBall is that it requires smooth activation functions to operate effectively. Our analysis in Fig. 5.7 shows that non-smooth activation functions such as ReLU significantly degrade performance, as points of non-differentiability violate the underlying assumptions of the quadratic model. To address this, we require a way to approximate these points without using the quadratic model. Approximation by finite differences is one such method. This has been done by Moosavi-Dezfooli et al. (2019) to approximate the Hessian cheaply, and has shown to be effective in CIFAR-10 classification tasks.

5.5.3 Hyperparameter Sensitivity

Our sensitivity analysis reveals that KryBall is reasonably dependent on the Krylov dimension M and refresh rate r_{refresh}. We note that while our hyperparameters are not as sensitive as Adam and MSGD, they still require careful tuning, where suboptimal values can lead to poor performance as we saw in Fig. 5.5. One way to address this could be to use adaptive schemes that automatically adjust M and r_{refresh}. This has been done by Henriques et al. (2019), who automatically rescale hyperparameters by using an objective change heuristic.

5.5.4 Trust-Region Step Rejection

Our trust-region framework can reject optimisation steps when the quadratic model poorly predicts actual function behavior. While this provides stability, frequent step rejections can slow convergence compared to first-order methods that always accept their steps. This is particularly the case when the Hessian approximation degrades or is in highly non-quadratic regions. A good way to address this is to simply use the gradient or momentum as a fallback. Moreover, we can instead consider the hybrid approach where if our step is rejected, we proceed with a first-order optimiser such as Adam or MSGD.

5.5.5 Computational Overhead

KryBall requires computing HVPs and constructing Krylov subspaces, introducing additional computational cost compared to first-order methods. While HVPs can be computed efficiently, the overall cost per iteration remains higher than first-order methods. The key limitation here is that the main computational cost comes from the construction of the Krylov subspace. As our Krylov dimension M is fixed throughout training, this means we concur a cost of $O(M \times N)$ per every $r_{refresh}$ iterations. This overhead could be mitigated by adaptively reducing the Krylov dimension M in regions where high-quality approximations are not critical, or by increasing the refresh rate $r_{refresh}$ when the landscape is relatively stable.

In this section, we presented the evaluations of our optimiser, KryBall, on a range of tasks. We evaluated KryBall's performance on ill-conditioned problems, simple binary classification, and more complex image classification tasks. We discussed KryBall's performance in comparison to the state-of-the-art optimisers Adam and MSGD. We then followed with a sensitivity analysis, a discussion about key questions and our hypotheses, and finally the limitations of our method. We now move on to conclude this thesis.

Chapter 6

Conclusion

In this chapter, we summarise the main contributions of this thesis, discuss its implications and suggest future work directions.

6.1 Summary

In this thesis, we introduced KryBall, a novel second-order optimisation algorithm designed to address fundamental limitations in current optimisation methods for deep learning. We make a number of contributions that span both the practical and theoretical aspects of optimisation.

For our theoretical contributions, we present the KryBall algorithm that combines the Saddle-Free-Newton method with a Krylov-subspace approach and integrates this within a trust-region framework. We connect the mathematical foundations of Krylov methods to practical optimisation performance, and address the issue of saddle point proliferation in high-dimensional spaces. We provide systematic analysis of loss landscape behaviour, problem conditioning, the effect of activation functions, and identify cases where our method excels and fails (such as when non-smooth activations are used). We also provide analytical tools for assessing the effectiveness of Krylov subspace approximations and demonstrate how to quantify the approximation error in our approach.

For the practical contributions, we present an implementation of Krylov subspace based Saddle-Free Newton methods integrated with a trust-region approach. We present the first N-dimensional subspace optimiser fully integrated with PyTorch, alongside an extensible testing suite. We also present our evaluations on ill-conditioned functions and classification tasks, where we demonstrate that KryBall is competitive with state-of-the-art methods, achieves rapid convergence on specific tasks, and is particularly well-suited to ill-conditioned problems.

6.2 Future Work

While KryBall demonstrates promising performance across several optimisation tasks, our evaluation reveals several avenues for improvement and extension. The limitations identified in Sec. 5.5, particularly around late-epoch instability, scalability, and dependence on smooth activation functions, point to promising research directions that can enhance our method. In this section, we outline key areas where future research could build upon our work. We discuss hybrid approaches, the need for theoretical rigour, improved model evaluations, support for non-smooth optimisation and computational efficiency.

6.2.1 Hybrid Approaches

Our analysis in Sec. 5.5.1 revealed that KryBall exhibits late-epoch instability when the loss landscape transitions from quadratic to subquadratic behavior near convergence. A promising direction is developing hybrid optimisation schemes that dynamically switch between second-order and first-order methods based on local landscape. There are many approaches to do this. For example, we could run KryBall for the first k iterations until we are no longer confident that our quadratic approximation is accurate, and then switch to MSGD or Adam. We could also run KryBall every k iterations, interweaving the updates with MSGD or Adam through training. The second approach has already seen wide use. The Sophia optimiser, which we covered in Chapter 3, is an example of this approach, as it weaves its second order update with signSGD (Liu et al., 2023).

6.2.2 Theoretical Rigour

While we empirically evaluate KryBall and can adhere to the rapid convergence it exhibits in tasks such as ill-conditioned function optimisation and XOR classification, we currently lack strong theoretical guarantees. Future research should focus on establishing formal convergence rates and characterising the conditions in which approximation methods hold. For example, this involves developing theoretical frameworks that connect the reconstruction error analysis to actual convergence properties. We can further extend this by establishing error bounds on the approximation methods we use. This includes formalising theoretical upper and lower bounds, as well as introducing an expectation on the error introduced by the approximation methods.

6.2.3 Improved Model Evaluations

Our evaluation focused primarily on small-scale problems, with ResNet-18 on CIFAR-10 representing the largest model tested. Future work should evaluate KryBall's scalability to contemporary large-scale applications such as transformer models, large convolutional networks, and other modern architectures. We also recommend expanding our evaluation suite to include more diverse tasks rather than just image classification. Examples of such tasks include natural language processing, such as language modelling on the Penn

Treebank dataset (Marcus et al., 1993), which involves predicting the next word in a sentence. Other tasks include 2D Neural Radiance Fields (Mildenhall et al., 2021). This involves predicting the radiance of a scene from a single image, and activation maximisation, which involves optimising an input image to maximise the activation of a specific neuron in a neural network (Olah et al., 2017).

6.2.4 Non-Smooth Optimisation

Our analysis demonstrated that KryBall's performance degrades significantly with non-smooth activation functions like ReLU due to violations of the quadratic model's smoothness assumptions. Future research should explore specialised variants that handle non-differentiable points more effectively. This could be done by using finite difference approximations, as was discussed in Sec. 3.3 or developing smoothed approximations of non-smooth activations, Another direction involves investigating how recent advances in non-smooth optimiation theory could be incorporated into Krylov-based methods to maintain second-order benefits while handling piecewise linear functions.

6.2.5 Computational Efficiency

We discussed the limitation of having computational overhead in Sec. 5.5.5. As such, future work could focus on exploring ways to reduce the computation overhead associated with second-order methods. First, developing more efficient Arnoldi iteration implementations that exploit sparsity patterns or low-rank structure could reduce the cost of subspace construction. Secondly, we can investigate adaptive schemes that dynamically adjust hyperparameters depending on the current point in the optimisation landscape. This ensures that parameters like the Krylov dimension M and the Krylov refresh rate $r_{refresh}$ are only updated when necessary. Thirdly, we explore research into creating parallelisable and distributed computing algorithms for common matrix operations, such as matrix inversion and eigendecomposition. These directions could significantly improve practical performance, and in combination with researching theoretical guarantees as stated in Sec. 6.2.2, we could ensure that second-order methods are more widely adaptable.

6.3 Concluding Thoughts

Optimisation in deep learning is a challenging problem, but it is fundamental for advancing the state-of-the-art. Optimisation has led to the development of powerful models, techniques and novel innovations. The work presented in this thesis is a step towards more efficient optimisation. We present KryBall, our second-order optimisation algorithm that combines the advantages of first and second-order methods by using Krylov subspaces, the Saddle-Free-Newton, and a trust-region framework. We thus take a step towards the goal of efficiently navigating the optimisation landscape, and improving the performance of integral machine learning models and tasks in today's world.

Bibliography

Alspector, J.; Meir, R.; Yuhas, B.; Jayakumar, A.; and Lippe, D., 1992. A parallel gradient descent method for learning in analog vlsi neural networks. *NIPS*, 5 (1992). [Cited on page 36.]

Belsley, D. A.; Kuh, E.; and Welsch, R. E., 2005. *Regression diagnostics: Identifying influential data and sources of collinearity*. John Wiley & Sons. [Cited on page 15.]

Boyd, S. P. and Vandenberghe, L., 2004. *Convex optimization*. Cambridge university press. [Cited on pages 12, 13, 15, 23, 24, and 31.]

Chen, X.; Liang, C.; Huang, D.; Real, E.; Wang, K.; Pham, H.; Dong, X.; Luong, T.; Hsieh, C.-J.; Lu, Y.; et al., 2024. Symbolic discovery of optimization algorithms. *NIPS*, 36 (2024). [Cited on pages 47 and 48.]

Choromanska, A.; Henaff, M.; Mathieu, M.; Arous, G. B.; and LeCun, Y., 2015. The loss surfaces of multilayer networks. In *Artificial intelligence and statistics*, 192–204. PMLR. [Cited on pages 18 and 19.]

Chowdhery, A.; Narang, S.; Devlin, J.; Bosma, M.; Mishra, G.; Roberts, A.; Barham, P.; Chung, H. W.; Sutton, C.; Gehrmann, S.; et al., 2023. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24, 240 (2023), 1–113. [Cited on page 1.]

Conn, A. R.; Scheinberg, K.; and Vicente, L. N., 2009. *Introduction to derivative-free optimization*. SIAM. [Cited on page 47.]

Dagréou, M.; Ablin, P.; Vaiter, S.; and Moreau, T., 2024. How to compute hessian-vector products? In *ICLR Blogposts 2024*. https://iclr-blogposts.github.io/2024/blog/bench-hvp/. Https://iclr-blogposts.github.io/2024/blog/bench-hvp/. [Cited on pages 26, 27, and 28.]

Dauphin, Y. N.; Pascanu, R.; Gulcehre, C.; Cho, K.; Ganguli, S.; and Bengio, Y., 2014. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. *NIPS*, 27 (2014). [Cited on pages 2, 3, 18, 19, 24, 25, and 49.]

Deisenroth, M. P.; Faisal, A. A.; and Ong, C. S., 2020. *Mathematics for machine learning*. Cambridge University Press. [Cited on pages 8, 9, 10, 12, 16, 20, 23, and 24.]

Dozat, T., 2016. Incorporating nesterov momentum into adam. (2016). [Cited on pages 37 and 40.]

Duchi, J.; Hazan, E.; and Singer, Y., 2011. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12, 7 (2011). [Cited on pages 37 and 38.]

Geoff, H., 2012. Rmsprop, coursera: Neural networks for machine learning. [Cited on page 38.]

Goodfellow, I.; Bengio, Y.; and Courville, A., 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org. [Cited on pages 8, 9, 10, 16, 17, 19, 20, 21, and 24.]

Greydanus, S. and Kobak, D., 2020. Scaling down deep learning with mnist-1d. *arXiv preprint arXiv:2011.14439*, (2020). [Cited on pages 62 and 68.]

Gutknecht, M. H., 2007. A brief introduction to krylov space methods for solving linear systems. In *Frontiers of Computational Science: Proceedings of the International Symposium on Frontiers of Computational Science 2005*, 53–62. Springer. [Cited on pages 27, 28, 29, and 30.]

He, K.; Zhang, X.; Ren, S.; and Sun, J., 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 770–778. [Cited on page 62.]

Henriques, J. F.; Ehrhardt, S.; Albanie, S.; and Vedaldi, A., 2019. Small steps and giant leaps: Minimal newton solvers for deep learning. In *ICCV*, 4763–4772. [Cited on pages 27, 37, 45, 46, 60, and 75.]

Hornik, K.; Stinchcombe, M.; and White, H., 1989. Multilayer feedforward networks are universal approximators. *Neural networks*, 2, 5 (1989), 359–366. [Cited on page 17.]

Johnson, R. and Zhang, T., 2013. Accelerating stochastic gradient descent using predictive variance reduction. *NIPS*, 26 (2013). [Cited on page 36.]

Jumper, J.; Evans, R.; Pritzel, A.; Green, T.; Figurnov, M.; Ronneberger, O.; Tunyasuvunakool, K.; Bates, R.; Žídek, A.; Potapenko, A.; et al., 2021. Highly accurate protein structure prediction with alphafold. *nature*, 596, 7873 (2021), 583–589. [Cited on page 1.]

Kingma, D. P., 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, (2014). [Cited on pages 1, 21, 38, 39, and 40.]

Krizhevsky, A.; Hinton, G.; et al., 2009. Learning multiple layers of features from tiny images. (2009). [Cited on page 62.]

Krizhevsky, A.; Sutskever, I.; and Hinton, G. E., 2017. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60, 6 (2017), 84–90. [Cited on page 1.]

Li, L.; Jamieson, K.; DeSalvo, G.; Rostamizadeh, A.; and Talwalkar, A., 2018. Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research*, 18, 185 (2018), 1–52. [Cited on page 63.]

Liu, H.; Li, Z.; Hall, D.; Liang, P.; and Ma, T., 2023. Sophia: A scalable stochastic second-order optimizer for language model pre-training. *arXiv preprint arXiv:2305.14342*, (2023). [Cited on pages 25, 44, 45, and 78.]

Loshchilov, I., 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, (2017). [Cited on pages 40 and 41.]

Ma, C.; Kunin, D.; Wu, L.; and Ying, L., 2022. Beyond the quadratic approximation: the multiscale structure of neural network loss landscapes. *arXiv preprint arXiv:2204.11326*, (2022). [Cited on pages 71, 72, and 75.]

Mao, A.; Mohri, M.; and Zhong, Y., 2023. Cross-entropy loss functions: Theoretical analysis and applications. In *International conference on Machine learning*, 23803–23828. PMLR. [Cited on pages 61 and 63.]

Marcus, M.; Santorini, B.; and Marcinkiewicz, M. A., 1993. Building a large annotated corpus of english: The penn treebank. *Computational linguistics*, 19, 2 (1993), 313–330. [Cited on page 79.]

Martens, J. et al., 2010. Deep learning via hessian-free optimization. In *ICML*, vol. 27, 735–742. [Cited on pages 27 and 45.]

Mildenhall, B.; Srinivasan, P. P.; Tancik, M.; Barron, J. T.; Ramamoorthi, R.; and Ng, R., 2021. Nerf: Representing scenes as neural radiance fields for view synthesis. *Communications of the ACM*, 65, 1 (2021), 99–106. [Cited on page 79.]

Moosavi-Dezfooli, S.-M.; Fawzi, A.; Uesato, J.; and Frossard, P., 2019. Robustness via curvature regularization, and vice versa. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 9078–9086. [Cited on page 75.]

Nemirovski, A.; Juditsky, A.; Lan, G.; and Shapiro, A., 2009. Robust stochastic approximation approach to stochastic programming. *SIAM Journal on optimization*, 19, 4 (2009), 1574–1609. [Cited on page 36.]

Nesterov, Y., 1983. A method for unconstrained convex minimization problem with the rate of convergence o (1/k2). In *Dokl. Akad. Nauk. SSSR*, vol. 269, 543. [Cited on pages 37 and 40.]

Nocedal, J. and Wright, S. J., 2006. *Numerical Optimization*. Springer, New York, NY, USA, 2e edn. [Cited on pages 8, 9, 10, 11, 12, 13, 15, 20, 21, 23, 24, 25, 27, 28, 29, 30, 31, 33, 36, 43, and 45.]

Olah, C.; Mordvintsev, A.; and Schubert, L., 2017. Feature visualization. *Distill*, (2017). doi:10.23915/distill.00007. Https://distill.pub/2017/feature-visualization. [Cited on page 79.]

Paszke, A.; Gross, S.; Chintala, S.; Chanan, G.; Yang, E.; DeVito, Z.; Lin, Z.; Desmaison, A.; Antiga, L.; and Lerer, A., 2017. Automatic differentiation in pytorch. (2017). [Cited on pages 16, 17, 20, 24, 27, 28, 57, and 58.]

Pearlmutter, B. A., 1994. Fast exact multiplication by the hessian. *Neural computation*, 6, 1 (1994), 147–160. [Cited on pages 26 and 27.]

Polyak, B. T., 1964. Some methods of speeding up the convergence of iteration methods. *Ussr computational mathematics and mathematical physics*, 4, 5 (1964), 1–17. [Cited on page 37.]

Rahimi, A. and Recht, B., 2017. The kitchen sink. https://archives.argmin.net/2017/12/05/kitchen-sinks/. Accessed: [Date you accessed it]. [Cited on pages 21 and 60.]

Reddi, S. J.; Kale, S.; and Kumar, S., 2019. On the convergence of adam and beyond. *arXiv preprint arXiv:1904.09237*, (2019). [Cited on pages 39 and 41.]

Robbins, H. and Monro, S., 1951. A stochastic approximation method. *The annals of mathematical statistics*, (1951), 400–407. [Cited on pages 1, 21, and 36.]

Ruder, S., 2016. An overview of gradient descent optimization algorithms. (2016). [Cited on pages 35, 36, and 38.]

Song, M.; Ahn, K.; and Yun, C., 2025. Does sgd really happen in tiny subspaces? *ICML*, (2025). [Cited on page 73.]

Sun, S.; Cao, Z.; Zhu, H.; and Zhao, J., 2019. A survey of optimization methods from a machine learning perspective. *IEEE transactions on cybernetics*, 50, 8 (2019), 3668–3681. [Cited on pages 36, 42, and 43.]

Sutskever, I.; Martens, J.; Dahl, G.; and Hinton, G., 2013. On the importance of initialization and momentum in deep learning. In *ICML*, 1139–1147. PMLR. [Cited on page 37.]

Taniguchi, S.; Harada, K.; Minegishi, G.; Oshima, Y.; Jeong, S. C.; Nagahara, G.; Iiyama, T.; Suzuki, M.; Iwasawa, Y.; and Matsuo, Y., 2024. Adopt: Modified adam can converge with any β_2 with the optimal rate. *NIPS*, (2024). [Cited on pages 41 and 42.]

Vinyals, O. and Povey, D., 2012. Krylov subspace descent for deep learning. In *Artificial intelligence and statistics*, 1261–1268. PMLR. [Cited on pages 30 and 46.]

Wigner, E. P., 1958. On the distribution of the roots of certain symmetric matrices. *Annals of Mathematics*, 67, 2 (1958), 325–327. [Cited on page 18.]

Yao, Z.; Gholami, A.; Shen, S.; Mustafa, M.; Keutzer, K.; and Mahoney, M., 2021. Adahessian: An adaptive second order optimizer for machine learning. In *AAAI*, vol. 35, 10665–10673. [Cited on page 44.]

Yousefi, M. and Martínez, Á., 2023. Deep neural networks training by stochastic quasi-newton trust-region methods. *Algorithms*, 16, 10 (2023), 490. [Cited on page 43.]

Zeiler, M. D., 2012. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, (2012). [Cited on page 38.]