UNIVERSITY OF
CAMBRIDGE
Department of Computer
Science and Technology

Rethinking Proximal Optimisation for Deep Learning

Charlie Tan

King's College

June 2023

Submitted in partial fulfillment of the requirements for the
Master of Philosophy in Advanced Computer Science

Total page count: 63

Main chapters (excluding front-matter, references and appendix): 51 pages (pp 7–57)

Main chapters word count: 14994

Methodology used to generate that word count:

```
texcount -inc -total -sum -brief main.tex
```

Declaration

I, Charlie Tan of King's College, being a candidate for the Master of Philosophy in Advanced Computer Science, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed: Charlie Tan

Date: 13th June 2023

Abstract

Deep learning presents a challenging non-convex optimisation problem, most commonly solved using variations of gradient descent. Gradient descent can be considered the simplest example of proximal optimisation, in which only first-order information is employed. Exploiting second-order information within the proximal optimisation framework gives rise to algorithms such as Newton-Raphson and natural gradients. Such algorithms are computationally intractable for modern neural network architectures, leading to a number of approximations thereof. In this work, we propose a novel approach for proximal optimisation within deep learning, closely related to the reinforcement learning algorithm proximal policy optimisation. The algorithm is evaluated on a range of image classification experiments, demonstrating its viability for optimising neural networks; the results however indicate a lower rate of convergence than a stochastic gradient descent baseline.

Acknowledgements

With great thanks to Dr Ferenc Huszár, without whose mentorship this work would not have been possible. With further thanks to Dr Challenger Mishra, and Theodore Long.

Contents

Chapter 1

Introduction

1.1 Motivation

Deep learning is a central technique within the modern machine learning toolkit, having been applied with great success to an extensive variety of tasks and data modalities [1–3]. Training modern neural networks requires high-performance computing resources, with the largest models requiring thousands of dedicated processing units for month-long training runs [4]. There is evidently a massive cost, both financial and in terms of energy, to neural network training. Given the pervasive applications of deep learning, the acceleration of optimisation is of widespread interest. More efficient optimisation algorithms could reduce the computational cost of deep learning training, reducing energy consumption or enabling more widespread experimentation and deployment.

1.2 Context

Stochastic gradient descent (SGD) is the canonical gradient-based optimisation algorithm and is widely successful within deep learning. As a first-order method, it is both time and memory efficient, requiring only the gradient vector to execute an optimisation step. SGD steps can be considered to solve a simple proximal objective, in which we minimise the objective function within a trust region surrounding our current parameter values, using only first-order approximations [5]. Introducing second-order approximations of the proximal objective terms leads to second-order methods such as Newton-Raphson, generalised Gauss-Newton, and natural gradients. It is hypothesised that the introduction of second-order information could greatly reduce the number of optimisation steps required, leading to a reduction in overall time complexity [6, 7].

Despite the proposed advantages, exact second-order optimisation is impractical for deep learning, necessitating the solution of a linear system of size d where d is the number of network parameters. Solving a linear system is an $\mathcal{O}(n^3)$ operation, making this computa-

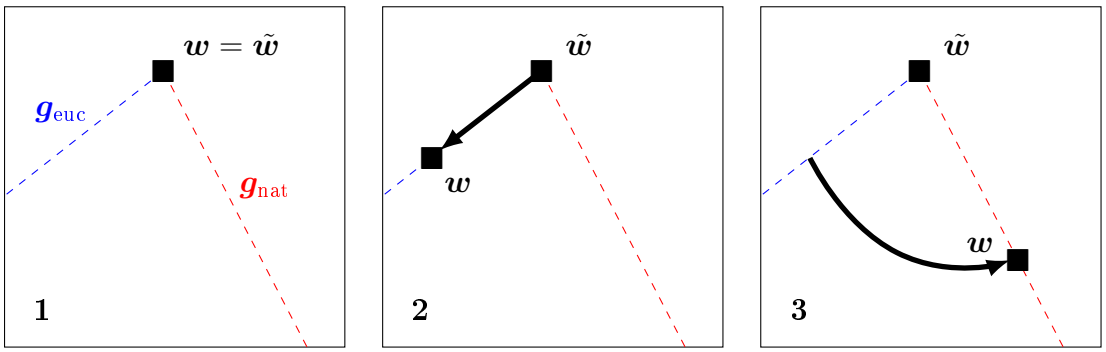

Figure 1.1: **Deadweight inner loop approximates the natural gradient using only first-order gradient steps.** Deadweight maintains two parameter vectors; the active model $\boldsymbol{w}$ and the reference model $\tilde{\boldsymbol{w}}$. Seeking to optimise an objective function $R(\boldsymbol{w})$, deadweight constructs and optimises a proximal objective $R_{\text{prox}}(\boldsymbol{w}, \tilde{\boldsymbol{w}})$. Tile 1: At initialisation, the active and reference model are in the same position $\boldsymbol{w} = \tilde{\boldsymbol{w}}$. Tile 2: After updating the active model $\boldsymbol{w}$ by a single gradient step we have necessarily followed the Euclidean gradient of R. This is due to R_{prox} being equal to R where $\boldsymbol{w} = \tilde{\boldsymbol{w}}$. Tile 3: After several further gradient steps on the proximal objective, the active model has converged onto the natural gradient trajectory. We now have an approximation to the natural gradient given by $\boldsymbol{w} - \tilde{\boldsymbol{w}}$.

tionally intractable for training modern neural networks, where the parameter counts readily exceed millions. There are therefore many proposed approximations of second-order methods. Notable examples include Adam [8], Hessian-Free [6] and Kronecker-Factored Approximate Curvature [7].

Generalisation presents an additional complication to neural network optimisation; unlike numerical optimisation, in machine learning, we seek strong performance on unseen samples. Generalisation can be promoted with explicit regularisation, such as weight decay or dropout [9]. However, optimisation algorithms can be inherently biased towards generalising solutions, known as implicit regularisation. Understanding the implicit regularisation of SGD [10, 11], and that of algorithms such as Adam [12] and natural gradient descent [13], is a highly active area of research. Interestingly, it is proposed that second-order methods may lack the strong bias towards generalisation of SGD [14, 15].

1.3 Contributions

In this work, we propose a novel proximal optimisation algorithm for neural networks, denoted 'deadweight' optimisation. Similar to other second-order methods, deadweight can be considered to repeatedly solve a trust-region problem. However, where existing methods leverage second-order Taylor expansions, deadweight differs by exploiting a non-zero first-order proximity term produced from two concurrently maintained parameter vectors, in a similar manner to proximal policy optimisation from reinforcement learning [16]. We hypothesise that by exploiting second-order information, deadweight will enable the optimisation of neural networks in fewer gradient steps than gradient descent. Achieving

faster convergence would imply significant practical relevance of this work. Nonetheless, demonstrating the viability of deadweight is still of theoretical interest; a novel approximation to second-order methods may have interesting inductive biases for further study. In figure 3.4, we present an illustration of the different stages of a deadweight inner loop.

The contributions of this work are as follows:

- Proposing novel proximal optimisation algorithm for neural networks; 'deadweight' optimisation.
- Demonstrating the success of deadweight on the Rosenbrock function as a toy example, including qualitative evidence of the algorithm approximating natural gradients.
- Conducting rigorous evaluation using fully-connected and convolutional architectures on the MNIST [17] and CIFAR-10 [18] datasets, demonstrating its viability for neural network optimisation.
- Comparing results with stochastic gradient descent and Hessian-Free as baselines, indicating an inferior rate of convergence to stochastic gradient descent but superior to Hessian-Free.
- Analysing the per-iteration time complexity of deadweight relative to SGD and Hessian-Free [6], indicating comparable time complexity to SGD.

Chapter 2

Background

In this chapter, we will introduce the proximal optimisation framework and its connection to trust-region optimisation. We will further discuss how algorithms such as gradient descent, Newton-Raphson, and natural gradient descent can be considered approximations of an idealised proximal optimisation method. This forms the necessary background for our proposed algorithm, deadweight optimisation.

2.1 Composite Optimisation

At its most abstract, this project is concerned with the gradient-based optimisation of composite functions of the form $h = g \circ f : \mathcal{W} \to \mathbb{R}$. We assume $g : \mathcal{Y} \to \mathbb{R}$ is convex with respect to $\boldsymbol{y} \in \mathcal{Y}$, however $f : \mathcal{W} \to \mathcal{Y}$ can be an arbitrary differentiable non-linear map. We seek to find the inputs $\hat{\boldsymbol{w}} \in \mathcal{W}$ that minimise h. The significance of h being composite is that we can define notions of proximity in both the input space $\mathcal{W}$ and an intermediate space $\mathcal{Y}$. These two notions of distance give rise to several interesting optimisation algorithms, within the framework of proximal optimisation [19]. Supervised learning with neural networks is the motivation of this project, in this context f is our model function and g is a loss function defined on the model outputs.

$$\hat{\boldsymbol{w}} = \operatorname*{argmin}_{\boldsymbol{w}} h(\boldsymbol{w}) \tag{2.1}$$

2.1.1 Global and Local Minima

If h is quadratic with respect to $\boldsymbol{w}$ there may be a closed-form solution for $\hat{\boldsymbol{w}}$. If instead h is only convex, there will be a single local minimum that also globally minimises the function; global optimisation reduces to simply finding this minimum. However non-convex functions can have multiple local minima, which may not themselves globally minimise the function. These local minima render the global optimisation of non-convex functions significantly harder than the convex case; assuming a non-convex h solving equation 2.1

is NP-Hard [20]. Within deep learning, we generally employ heuristic algorithms such as stochastic gradient descent (SGD), greatly decreasing the computational complexity, without the guarantee of finding the global minima.

2.1.2 Gradients and Jacobians

The simplest gradient-based algorithms use first-order derivatives to optimise $\boldsymbol{w}$. The gradient of g with respect to $\boldsymbol{y} = f(\boldsymbol{w})$ is defined as in equation 2.2. Assuming f is not invertible this alone cannot be used to optimise $\boldsymbol{w}$; for this we require $\nabla_{\boldsymbol{w}} h(\boldsymbol{w})$. The compositional nature of h requires that we use the chain rule; this necessitates the Jacobian of f, denoted as $\boldsymbol{J}_f$, and defined elementwise in equation 2.3. The gradient is a special case of the Jacobian where the function is scalar-valued, both represent a first-order approximation to the variation of a function with respect to its inputs. With the Jacobian defined we simply left multiply the gradient of g with $\boldsymbol{J}_f^T$ to get the gradient with respect to the inputs $\boldsymbol{w}$, as in equation 2.4.

$$\nabla_{\boldsymbol{y}} g(\boldsymbol{y}) = \begin{bmatrix} \frac{\partial g}{\partial y_i} & \cdots & \frac{\partial g}{\partial y_k} \end{bmatrix}^T \tag{2.2}$$

$$(\boldsymbol{J}_f)_{ij} = \frac{\partial f_i}{\partial x_j} \tag{2.3}$$

$$\nabla_{\boldsymbol{w}} h(\boldsymbol{w}) = \boldsymbol{J}_f^T \nabla_{\boldsymbol{y}} g(f(\boldsymbol{w})) \tag{2.4}$$

2.1.3 Hessians and pullback Hessians

Some optimisation algorithms require second-order derivatives. The Hessian $\boldsymbol{H}_h$ of a scalar-valued function h is defined elementwise in equation 2.5. The Hessian describes the curvature of the function in the region surrounding the evaluation point. Considering the gradient as a first-order approximation of h, the curvature describes whether the function curves upwards (positive curvature) or downwards (negative curvature) relative to the first-order approximation. This curvature information can be exploited by optimisation algorithms, enabling more accurate approximations to h at greater distances from the evaluation point [21].

$$(\boldsymbol{H}_h)_{ij} = \frac{\partial^2 h}{\partial x_i x_j} \tag{2.5}$$

In some cases, it will not be possible to compute the Hessian directly, or the true Hessian may lack a necessary property such as positive definiteness, as is required for approximate solution with the conjugate gradient method [6]. For this reason, we follow Grosse [5] in defining the 'pullback' of a Hessian. Here we use second-order information from g but

only a first-order approximation of f. This can be thought of as the Hessian of h where f is replaced with its first-order Taylor approximation. This takes the form of equation 2.6. Due to the convexity of g, $\boldsymbol{G}_g$ is guaranteed to be positive-definite [22]. We will later see that the generalised Gauss-Newton method and natural gradients both rely on such 'pullback' Hessians [19].

$$\boldsymbol{G}_h = \boldsymbol{J}_f^T \boldsymbol{H}_g \boldsymbol{J}_f \tag{2.6}$$

2.2 Metric Tensors

Here we briefly introduce metric tensors, a concept from differential geometry describing distances and angles. Defining a metric on the intermediate $\mathcal{Y}$-space enables us to measure dissimilarity between optimisation iterates in this space. Whilst having a metric theoretically enables us to exactly compute distances, for all but the simplest problems the resulting integral will be intractable. However, the metrics also appear as the second-order Taylor approximations of proximity terms [5]. This gives a second-order approximation of the distance and forms the basis of natural gradient methods.

2.2.1 Euclidean Metric

The Euclidean metric is simply the identity matrix. It is the simplest metric, enabling the trivial computation of exact distances and angles. Since exact distances can be handled, the metric itself will not appear often in calculations. However, the squared Euclidean distance between points in the intermediate $\mathcal{Y}$-space will be used extensively when dealing with the Rosenbrock function in Chapter 3.

$$\boldsymbol{F} = \boldsymbol{I} \tag{2.7}$$

2.2.2 Fisher Information

Information geometry is the study of statistical models as manifolds of probability distributions [23]. A fundamental concept of information geometry is the Fisher information metric. Let $p(\boldsymbol{y}; \boldsymbol{\theta})$ be a probability distribution parameterised by $\boldsymbol{\theta}$. The Fisher scores are the gradients of the negative log-likelihood of this distribution $-\nabla_{\boldsymbol{\theta}} \log p(\boldsymbol{y}; \boldsymbol{\theta})$. The Fisher information is then the covariance of the Fisher scores, as in equation 2.8. Under mild assumptions, this is equivalent to the expected Hessian of the negative log-likelihood, where the expectation is taken with respect to the parametric distribution itself $p(\boldsymbol{y}; \boldsymbol{\theta})$ [5].

$$\boldsymbol{F}_{\boldsymbol{\theta}} = \mathbb{E}_{p(\boldsymbol{y};\boldsymbol{\theta})}[\nabla_{\boldsymbol{\theta}} \log p(\boldsymbol{y}; \boldsymbol{\theta}) \nabla_{\boldsymbol{\theta}} \log p(\boldsymbol{y}; \boldsymbol{\theta})^T] \tag{2.8}$$

We additionally comment on the empirical Fisher information; this takes the same form as equation 2.8, however the expectation is taken with respect to the empirical distribution of a dataset, and not the distribution of the model itself [24]. Despite their similar forms, these matrices have fundamentally different properties for optimisation [25].

2.3 Optimisation in Supervised Deep Learning

Here we introduce supervised deep learning and its relation with the general composite optimisation framework introduced in section 2.1.

2.3.1 Approximating Distributions

In this work, we consider a standard supervised learning framework. We have a data space $\mathcal{X} \times \mathcal{Y}$, consisting of features $\boldsymbol{x} \in \mathcal{X}$ and targets $\boldsymbol{y} \in \mathcal{Y}$. For classification problems, the targets represent discrete class labels, whereas for regression they will be continuous values. There is a joint distribution over features and targets $p(\boldsymbol{x}, \boldsymbol{y})$. Since the targets are conditionally dependent on the features, we may decompose this joint distribution into the feature distribution and the conditional distribution of targets given features $p(\boldsymbol{x}, \boldsymbol{y}) = p(\boldsymbol{x})p(\boldsymbol{y}|\boldsymbol{x})$. We are concerned with learning a function or *model* $f : \mathcal{X} \times \mathcal{W} \to \mathcal{Y}$, with parameters $\boldsymbol{w} \in \mathcal{W}$, to approximate $p(\boldsymbol{y}|\boldsymbol{x})$.

In order to evaluate the quality of a parameter vector $\boldsymbol{w} \in \mathcal{W}$ we employ a loss function $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$. For features $\boldsymbol{x}$, the loss function measures the similarity between $p(\boldsymbol{y}|\boldsymbol{x})$ and $p(\boldsymbol{y}|\boldsymbol{x}; \boldsymbol{\theta})$. We define the expectation of the loss, with respect to the true joint distribution of the data as the population risk R, as in equation 2.9. Approximating the distribution $p(\boldsymbol{y}|\boldsymbol{x})$ amounts to identifying the parameter vector $\hat{\boldsymbol{w}}$ that minimises R. This is a form of maximum likelihood estimation; we are seeking the parameters that maximise the likelihood of our model producing the correct target, for a given feature.

$$R(\boldsymbol{w}) = \mathbb{E}_{p(\boldsymbol{x},\boldsymbol{y})} \left[\ell(\boldsymbol{y}, f(\boldsymbol{x}; \boldsymbol{w})) \right] \tag{2.9}$$

$$\hat{\boldsymbol{w}} = \operatorname*{argmin}_{\boldsymbol{w}} R(\boldsymbol{w}) \tag{2.10}$$

2.3.2 Finite Samples and Generalisation

In supervised learning, the true distribution $p(\boldsymbol{x}, \boldsymbol{y})$ is unknown; we only have access to a finite dataset of N samples $\{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^N$, drawn independently and identically $(\boldsymbol{x}_i, \boldsymbol{y}_i) \sim p(\boldsymbol{x}, \boldsymbol{y})$. We refer to this as the training dataset. Using this finite dataset we define the *sample risk* and *empirical risk* as in equations 2.11 and 2.12. The empirical risk can be considered the expectation of the sample risk under the empirical distribution of the training dataset. The challenge of supervised learning is to learn a model using only this

empirical distribution, such that the model *generalises* to unseen samples drawn from the population distribution.

$$R^{(i)}(\boldsymbol{w}) = \ell(\boldsymbol{y}^{(i)}, f(\boldsymbol{x}^{(i)}; \boldsymbol{w})) \tag{2.11}$$

$$\hat{R}(\boldsymbol{w}) = \frac{1}{N} \sum_{i=1}^{N} R^{(i)}(\boldsymbol{w}) \tag{2.12}$$

The objective of generalisation is a fundamental difference between machine learning and classical optimisation [26]. Generalisation is formally defined as the difference of the population risk (equation 2.9) and the empirical risk (equation 2.12). However since we do not have access to the true data generating distribution, we use a test dataset of a further M samples drawn from $p(\boldsymbol{x}, \boldsymbol{y})$, such that no sample is present in both the training and test data. Performance on the unseen samples of the test dataset is used as a proxy for the population risk.

2.3.3 Supervised Deep Learning and Composite Optimisation

Supervised learning can be cast within the framework of composite optimisation, with the empirical risk as our objective function h. Here ℓ takes the place of g, and for convenience we treat our features $\boldsymbol{x} \in \mathcal{X}$ as constants. The empirical risk can then be regarded as the composition of the model and the loss function $R = \ell \circ f : \mathcal{W} \to \mathbb{R}$. In section 2.1 we assumed that g was convex, a useful property for optimisation. Mean squared error and cross-entropy are both convex in this sense, satisfying this assumption.

Where f is a neural network, to obtain the parameter vectors $\boldsymbol{w}$ we must flatten and concatenate the parameter matrices describing each of the network layers. Obtaining the gradient with respect to this parameter vector requires the backpropagation algorithm [27], a technicality we can safely ignore in this work. There are also technical details for handling the Jacobian matrices, these shall similarly be ignored without complication.

2.3.4 Bias Towards Generalisation

Neural networks are often employed in the *overparameterised regime*, wherein the model has sufficient capacity to memorise its training data [10]. Classical learning theory indicates that within this regime models should overfit their training data and generalise poorly [28]. The widespread empirical success of deep learning therefore defies description by classic learning theory. The bias of neural networks towards generalising solutions is typically attributed to the implicit regularisation of optimisation methods such as SGD [29]. There has also been significant interest into the geometry of solutions within parameter space, particularly notions of 'flat minima' [30–33]. Moreover, the mapping between

parameters and function itself is considered to be biased towards simple functions [14, 34].

Stochastic gradients are an important consideration within deep learning; evaluating the mean gradient over the entire training dataset would require often excessive memory for modern architectures and datasets. Even in cases where this deterministic setting is feasible, using a mini-batch of data can provide a good approximation to the true gradient with significantly lower computational cost. Lastly, and most importantly the stochasticity of mini-batch gradients introduces additional implicit regularisation [29, 35]. Another subtlety arises in second-order optimisation; along with the gradient, matrices such as Hessians, Jacobians or the Fisher Information are stochastically estimated from a mini-batch of data.

2.3.5 Momentum and Normalisation

There are a number of techniques that can be employed to improve the convergence of deep learning training or enhance the generalisation performance. Two prominent examples are momentum and normalisation. Momentum updates parameters using an exponential moving average of the gradient, creating a 'heavy-ball' like behaviour. Momentum is understood to address many of the shortcomings of first-order optimisation [36]. Several normalisation schemes exist, including batch normalisation [37], layer normalisation [38] and group normalisation [39]. All normalise the activations to have zero mean and unit variance but differ in how the activations are grouped for normalisation. Normalisation can be viewed as a form of adaptive reparameterisation, helping to improve the conditioning of the learning problem, as well as enabling stable training at higher learning rates [24].

2.4 Desiderata for Neural Network Optimisers

Evidently, there are a number of considerations when designing or selecting an optimisation algorithm for deep learning. We formalise these considerations into 4 desiderata:

1. Computationally tractable for network architectures with millions or billions of parameters, both in terms of time and memory complexity.

2. Rapid convergence, either requiring few total steps to converge or taking many highly efficient steps.

3. Bias towards generalisation, achieving comparable or superior generalisation performance to stochastic gradient descent.

4. Architecture agnostic; able to operate on a vector of parameters irrespective of how these parameters are employed within the architecture.

Note that we explicitly exclude a requirement to leverage second-order information, view-

ing this to be an approach, as opposed to a desirable property.

The first requires little justification; we must be able to execute the algorithm using modern architectures on current hardware. Algorithms could fail to meet this by requiring excess computation per iteration, the storage of excess information, or both. Likewise, the importance of the second is readily apparent; accelerating convergence is one of the primary motivations for research into neural network optimisation [40]. An algorithm that fails number three would be of no practical relevance; ideally the generalisation performance would exceed that of SGD, however achieving comparable performance in fewer iterations is another valid objective. The importance of number four is slightly more subtle. However, being able to change architecture components and configurations without modifying the optimisation configuration is a highly desirable property. Naturally, a trade-off between these desiderata can be expected, and the relative importance of these may be highly contextual.

2.5 Proximal Optimisation

We now proceed to proximal optimisation, an optimisation framework that penalises large steps between iterates [5].

2.5.1 Trust-Regions and Proximal Optimisation

Iterative optimisers repeatedly optimise local approximations to the objective function. These local approximations typically consist of first and second-order Taylor expansions, enabling highly efficient approximations of function values in the neighbourhood of our current parameter values $\boldsymbol{w}^{(t)}$. However, this approximation will be increasingly less accurate further away from $\boldsymbol{w}^{(t)}$, as higher-order effects dominate. We therefore define a *trust-region* surrounding $\boldsymbol{w}^{(t)}$ in which we trust our approximation of R to be accurate, and optimise in this region [5]. This trust region is defined using a radius r and some distance function ρ, as in equation 2.13. Here $R_{\approx}$ denotes a non-specified approximation to R.

$$\boldsymbol{w}^{(t+1)} = \underset{\boldsymbol{w}:\rho(\boldsymbol{w},\boldsymbol{w}^{(t)})<r}{\operatorname{argmin}} R_{\approx}(\boldsymbol{w}) \tag{2.13}$$

A Taylor expansion of order one will always result in a convex local approximation. Under certain assumptions, convexity will also hold for a Taylor expansion of order two. It immediately follows that if our local approximation is convex, the solution to equation 2.13 will be found on the boundary where $\rho(\boldsymbol{w}, \boldsymbol{w}^{(t)}) = r$. By the method of Lagrangian multipliers, any convex trust-region problem can be equivalently represented as a *proximal* method [5, 13]. Here the constrained optimisation problem of equation 2.13 is expressed as an unconstrained problem called a *Lagrangian*, where the constraint is imposed through a

proximity term $\rho(\boldsymbol{w}, \boldsymbol{w}^{(t)})$, as in equation 2.14. The Lagrangian is solved when all partial derivatives, in this case $\nabla_{\boldsymbol{w}} L$ and $\frac{\partial L}{\partial \lambda}$ are equal to 0.

$$L(\boldsymbol{w}, \lambda) = R_{\approx}(\boldsymbol{w}) + \lambda\rho(\boldsymbol{w}, \boldsymbol{w}^{(t)}) - \lambda r \tag{2.14}$$

Often we are content setting the λ directly and knowing this corresponds to some trust-region radius r, without explicitly knowing the size of this region. In some cases such as gradient descent and natural gradients the step direction is in fact independent of the radius [13]. This leads us to the notion of proximal optimisation, with update equation given by equation 2.15

$$\boldsymbol{w}^{(t+1)} = \operatorname{argmin} \left[R_{\approx}(\boldsymbol{w}) + \lambda\rho(\boldsymbol{w}, \boldsymbol{w}^{(t)}) \right] \tag{2.15}$$

2.5.2 Proximal Optimisation of Composite Objectives

We will now address how proximal optimisation may exploit the compositional structure of objective functions. As mentioned in section 2.1, an interesting aspect of composite optimisation is that we have both the weight space and intermediate space in which we can penalise movement. We follow Grosse [19] in defining an idealised proximal optimisation method, where $\rho : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$ is some dissimilarity function operating on elements of the $\mathcal{Y}$-space. The term $\lambda_{\mathcal{W}}||\boldsymbol{w} - \boldsymbol{w}^{(t)}||^2$ controls weight space proximity, by penalising large changes in weights between steps. The term $\lambda_{\mathcal{Y}}\rho(f(\boldsymbol{w}), f(\boldsymbol{w}^{(t)}))$ controls prediction space proximity by penalising large changes in predictions between steps.

$$\begin{aligned} \boldsymbol{w}^{(t+1)} &= \operatorname*{argmin}_{\boldsymbol{w}} \left[R(\boldsymbol{w}) + \lambda_{\mathcal{W}}||\boldsymbol{w} - \boldsymbol{w}^{(t)}||^2 + \lambda_{\mathcal{Y}}\rho(f(\boldsymbol{w}), f(\boldsymbol{w}^{(t)})) \right] \\ &= \operatorname*{argmin}_{\boldsymbol{w}} \left[R_{\text{prox}}(\boldsymbol{w}, \boldsymbol{w}^{(t+1)}) \right] \end{aligned} \tag{2.16}$$

Evidently, R_{prox} will be no easier to optimise than the original objective R. However, as shown by Grosse [19] a number of optimisation algorithms can be derived as approximations of this idealised update rule. The general derivation is consistent between algorithms:

1. Appropriately set $\lambda_{\mathcal{W}}$ and/or $\lambda_{\mathcal{Y}}$ to enforce weight and/or prediction space proximity.
2. Approximate $R(\boldsymbol{w})$ and $\rho(f(\boldsymbol{w}), f(\boldsymbol{w}^{(t)}))$ with Taylor expansions of at most order two.
3. Derive an update rule by solving the resulting convex local approximation.

We note that all methods using second-order Taylor approximations fail the first desiderata as defined in section 2.4; they are intractable in deep learning. The Hessian is a $d \times d$ matrix which for even modest modern neural network architectures will be impossible to compute and store, and intractable to invert. Therefore despite already including approximate terms, all of the following algorithms, gradient descent aside, are to be considered ideal algorithms to be further approximated.

2.5.3 Iterative Optimisation is Proximal Optimisation

We will now discuss how several first and second-order iterative optimisation algorithms can be cast as approximations of the idealised proximal objective stated in equation 2.16 [19]. In table 2.1 we present a summary of the derivations.

Gradient Descent

Gradient descent is the simplest gradient-based optimiser. It is often introduced without the connection to proximal optimisation; we simply take a step in the direction of the negative gradient at our current parameter values, scaled by a learning rate. However, the update rule implicitly encodes a locality constraint in the parameter space. Referring to our idealised proximal objective equation 2.16, we simply take a first order Taylor expansion of R and set $\lambda_{\mathcal{Y}} = 0$ [5]. This leads to the following update equation, where we see that the reciprocal of our weight space penalty scale takes the place of the learning rate.

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \lambda_{\mathcal{W}}^{-1} \nabla R(\boldsymbol{w}^{(t)}) \tag{2.17}$$

Newton-Raphson

Newton-Raphson is the canonical second-order optimisation algorithm. Each iteration constructs and solves a second-order Taylor expansion of R in the region of the current parameter values $\boldsymbol{w}^{(t)}$, leading to the update defined by equation 2.18. This is expressible in the framework of our idealised proximal objective 2.16, where $\lambda_{\mathcal{Y}} = 0$ and the value of $\lambda_{\mathcal{W}} \geq 1$ defines the *damping* applied. Here $\boldsymbol{H}_R = \nabla^2 R(\boldsymbol{w})|_{\boldsymbol{w}=\boldsymbol{w}^{(t)}}$ is the Hessian of the risk term evaluated at the current position.

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - (\boldsymbol{H}_R + \lambda_{\mathcal{W}} \boldsymbol{I})^{-1} \nabla R(\boldsymbol{w}^{(t)}) \tag{2.18}$$

There are two main motivations for Newton-Raphson over gradient descent. Firstly, the second-order Taylor approximation of the objective term R implies our local approximation to be more accurate at large distances from our current iterate $\boldsymbol{w}^{(t)}$ [21]. This means we can take larger steps, corresponding to a larger trust region, before training becomes unstable. Secondly, for a quadratic objective, the convergence of gradient descent

is inversely proportional to the condition number of the quadratic. Newton-Raphson can be considered to precondition the optimisation problem, reducing the condition number and improving convergence rate [19]. Near minima, non-quadratic functions can be well approximated locally with a quadratic.

Whilst Newton-Raphson defines and solves a local quadratic approximation, this quadratic may be indefinite due to the presence of negative curvature, which can be present in deep learning where the objective function is non-convex [6]. This means Newton-Raphson is only guaranteed to find a stationary point, of which minima and saddle points are valid solutions. For this reason, Newton-Raphson is unsuitable for use in cases where our objective function is non-convex and we may have saddle points [5].

Generalised Gauss-Newton

As discussed previously, the Hessian of the risk function may be indefinite, presenting an obstacle for approximate inversion. We may exploit the convexity of the loss function to produce the generalised Gauss-Newton Hessian $\boldsymbol{G}_R$ [5]. This generalises the Gauss-Newton method, defined only for squared error to an arbitrary convex loss function.

$$\boldsymbol{G}_R = \boldsymbol{J}_f^T \boldsymbol{H}_\ell \boldsymbol{J}_f$$

Replacing $\boldsymbol{H}_R$ in equation 2.18 with $\boldsymbol{G}_R$, leads to the generalised Gauss-Newton method. This can be thought of as performing Newton-Raphson on a modified version of the risk function where f is linearised using a first-order Taylor approximation [5]. Referring back to section 2.1, the generalised Gauss-Newton matrix is the pullback Hessian of the risk function.

Natural Gradient Descent

In the case where $\lambda_{\mathcal{Y}} > 0$ we are interested in penalising the step size in the intermediate $\mathcal{Y}$-space. Natural gradients use a first-order approximation to R and a second-order approximation to ρ, with $\lambda_{\mathcal{W}} \geq 0$ again defining the damping applied. This gives the following update rule, where $\boldsymbol{H}_\rho = \nabla^2 \rho(\boldsymbol{w}, \boldsymbol{w}^{(t)})|_{\boldsymbol{w}=\boldsymbol{w}^{(t)}}$ is the Hessian of the proximity term evaluated at the current position.

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - (\lambda_{\mathcal{Y}} \boldsymbol{H}_\rho + \lambda_{\mathcal{W}} \boldsymbol{I})^{-1} \nabla R(\boldsymbol{w}^{(t)}) \tag{2.19}$$

Natural gradients were originally defined by Amari [13] using the KL-divergence as ρ, leading the Hessian $\boldsymbol{H}_\rho$ to be the Fisher information on the space of distributions. However, we can define natural gradients using different $\mathcal{Y}$-space metrics and dissimilarity functions [5]. There is typically a natural choice of metric in the space, connected to our choice of loss function. In this work, we will consider the squared Euclidean distance and

	Gradient Descent	Newton-Raphson	Natural Gradient
$\lambda_{\mathcal{W}}$	> 0	damping	damping
$\lambda_{\mathcal{Y}}$	$= 0$	$= 0$	> 0
$R(\boldsymbol{w})$	First-order	Second-order	First-order
$\rho(f(\boldsymbol{w}), f(\boldsymbol{w}^{(t)}))$	N/A	N/A	Second-order

Table 2.1: **Derivation of common optimisation algorithms from the idealised proximal objective.** The idealised proximal objective is stated in equation 2.16. All of gradient descent, Newton-Raphson and natural gradient descent can be derived as approximations of this update rule [5]. To achieve this we set $\lambda_{\mathcal{W}}$ and $\lambda_{\mathcal{Y}}$ as in the table, defining penalties for the weight ($\mathcal{W}$) and intermediate ($\mathcal{Y}$) spaces respectively. We furthermore approximate the terms $R(\boldsymbol{w})$ and $\rho(f(\boldsymbol{w}), f(\boldsymbol{w}^{(t)}))$ with Taylor expansions of the stated order. We emphasise that natural gradients are not specific to any dissimilarity function ρ, rather different choices of ρ correspond to different metrics on the $\mathcal{Y}$-space.

KL-divergence. The KL-divergence can be thought of as analogous to a squared distance defined for probability distributions.

Natural gradient descent is an approximation of doing gradient descent directly in the $\mathcal{Y}$-space, which is something we cannot meaningfully do for a non-invertible f. Formally, the natural gradient is invariant to reparameterisation up to first-order [5, 14]. In contrast, gradient descent is parameterisation dependent; the optimisation trajectory and solution found depends on the specific parameterisation of the model [5]. Secondly, defining distance in the intermediate $\mathcal{Y}$-space, natural gradients are the steepest direction in which we can update our parameters to minimise the loss [13].

Chapter 3

Deadweight Optimisation

In Chapter 2 we saw how the compositional nature of supervised learning objectives can be exploited by proximal optimisation. We further discussed that several iterative methods, including gradient descent, Newton-Raphson and natural gradient descent, can be cast as approximations of an idealised proximal update rule. Here we proceed by developing a novel optimisation method, deadweight optimisation, to perform proximal optimisation using only first-order gradient descent steps, for arbitrary distance functions. To guide discussion, the Rosenbrock function is employed as a motivating example throughout this chapter. We additionally discuss shortcomings of our first proposed method and introduce a series of improvements and optimisations to our base algorithm.

3.1 A Toy Problem: Rosenbrock Function

The Rosenbrock function is a test function first defined by Rosenbrock [41], and stated in equation 3.1.

$$h(\boldsymbol{w}) = h(w_1, w_2) = (1 - w_1)^2 + 100(w_2 - w_1^2)^2 \tag{3.1}$$

The Rosenbrock function can be decomposed into the framework of composite optimisation [5], making it a useful toy example for proximal optimisation. In equations 3.2 and 3.3 the Rosenbrock function is decomposed into a non-linear map f, and a convex loss function ℓ, namely the squared Euclidean distance.

$$\boldsymbol{y} = f(\boldsymbol{w}) = f(w_1, w_2) = (1 - w_1, \sqrt{100}) \tag{3.2}$$

$$z = \ell(\boldsymbol{y}) = \ell(y_1, y_2) = y_1^2 + y_2^2 \tag{3.3}$$

The Rosenbrock is an established test function for optimisation, owing to the challenging

problem it presents. The function is non-convex with a single minimum located in an almost-flat valley. The minimum is found at $w_1 = w_2 = 1$. Entering the valley is trivial, however, the gradient along the bottom towards the minimum is negligible making convergence very slow. Another useful property of the Rosenbrock as a test function is that its non-linear map f is invertible. Hence, we can exactly compute the $\mathcal{Y}$-space gradient which is our idealised natural gradient path.

Since our 'loss' function is squared Euclidean distance, the squared Euclidean distance is a natural choice of dissimilarity function for points in the $\mathcal{Y}$-space. We can regard there as being a Euclidean metric on this space, where the 'loss' is the squared distance to the origin. Another interpretation is that the squared difference is the Bregman divergence generated by the squared distance. Measuring proximity using the loss function's generated Bregman divergence leads to the interesting result that the pullback Hessian of the proximity term is a scalar multiple of the pullback Hessian of the loss [5]. This implies that the generalised Gauss-Newton and natural gradient methods are equal up to scale factor.

The optimisation trajectories for $\mathcal{W}$-space gradient descent, $\mathcal{Y}$-space gradient descent, and natural gradient descent are given in figure 3.1. We observe that $\mathcal{W}$-space gradient descent immediately enters the valley, and then converges along the valley floor; in $\mathcal{Y}$-space this path has a distinct turn in its trajectory when we reach the bottom of the valley. Since ℓ is quadratic the $\mathcal{Y}$-gradient path is exactly a straight line from the initialisation to the origin; in $\mathcal{W}$-space this is a curved path that avoids the valley until the minimum. We observe the natural gradient to be imperceptible from the $\mathcal{Y}$-space gradient. Intuitively, natural gradients follow this path by penalising travel towards the valley floor due to the high positive curvature present in that direction.

3.2 A First Attempt

Recall the idealised proximal update rule, restated in equation 3.4. As discussed in section 2.5, several second-order optimisation algorithms including Newton-Raphson, generalised Gauss-Newton and natural gradients can be seen as approximations of this idealised update rule [19]. These algorithms are obtained, for appropriate constants $\lambda_{\mathcal{W}}$ and $\lambda_{\mathcal{Y}}$, by approximating the terms $R(\boldsymbol{w})$ and $\rho(f(\boldsymbol{w}), (\boldsymbol{w}^{(t)}))$ using first and second-order Taylor expansions.

$$\boldsymbol{w}^{(t+1)} = \underset{\boldsymbol{w}}{\operatorname{argmin}} \left[R(\boldsymbol{w}) + \lambda_{\mathcal{W}} ||\boldsymbol{w} - \boldsymbol{w}^{(t)}||^2 + \lambda_{\mathcal{Y}} \rho(f(\boldsymbol{w}), f(\boldsymbol{w}^{(t)})) \right] \tag{3.4}$$

3.2.1 Motivating Observation

Up to first-order approximation the $\mathcal{Y}$-space proximity term in equation 3.4 will be necessarily zero. Penalising inter-iteration distance in this space therefore requires a second-

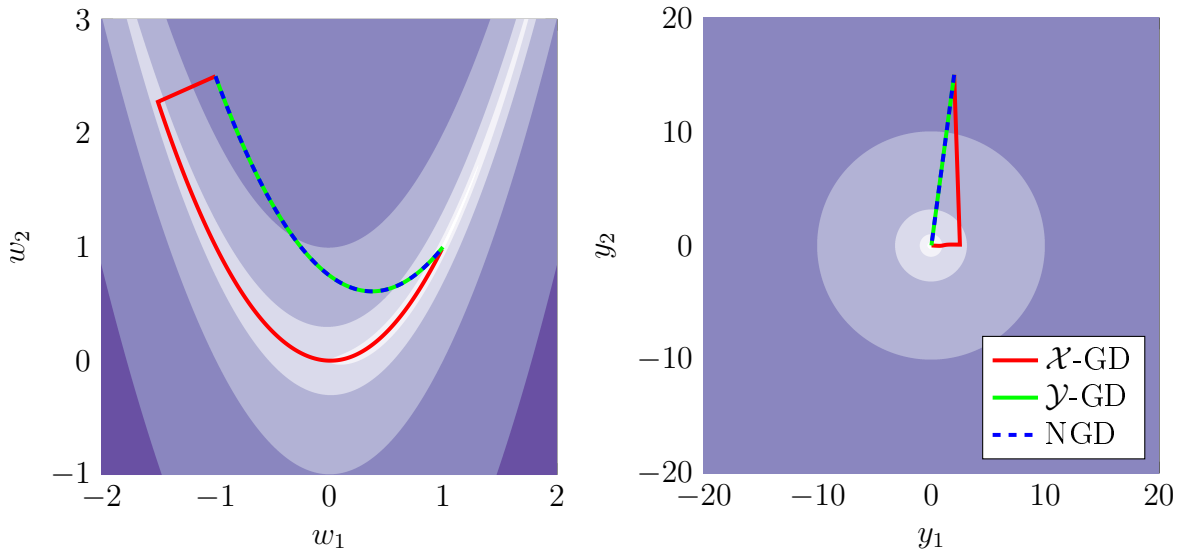

Figure 3.1: **Natural gradient descent follows the $\mathcal{Y}$-space gradient trajectory**. The composition of a non-linear map f with a convex function g makes the Rosenbrock function a useful toy example for deep learning. Left: the full non-convex Rosenbrock function $g \circ f$, with minimum at $\boldsymbol{w} = [1, 1]$. Right: the squared distance 'loss' function g, a quadratic with minimum at the origin. $\mathcal{W}$-space gradients take steps in the steepest direction in the $\mathcal{W}$-space, immediately entering the valley and then converging slowly along the valley floor. $\mathcal{Y}$-space gradients travel in a straight line from the initialisation to the minimum in $\mathcal{Y}$-space; in the $\mathcal{W}$-space this is a curved path that avoids the valley until the minimum is reached. The natural gradients follow the $\mathcal{Y}$-space gradient trajectory.

order approximation of this term, as in natural gradient descent. This project began with the observation that non-zero proximity term gradients can alternatively be achieved by maintaining a reference weight vector $\tilde{\boldsymbol{w}} \neq \boldsymbol{w}^{(t)}$, in place of the previous iterate $\boldsymbol{w}^{(t)}$. We hypothesised that introducing this proximity penalty would introduce some of the inductive bias of natural gradients, and potentially accelerate optimisation, without explicitly utilising any costly second-order information. We define the updated proximal objective function R_{prox} in equation 3.5. In natural gradient descent $\lambda_{\mathcal{W}}$ corresponds to damping, for convenience we shall assume this to be zero for now.

$$R_{\text{prox}}(\boldsymbol{w}, \tilde{\boldsymbol{w}}) = R(\boldsymbol{w}) + \lambda_{\mathcal{Y}} \rho(f(\boldsymbol{w}), f(\tilde{\boldsymbol{w}})) \tag{3.5}$$

3.2.2 Solving Proximal Objective with Coordinate Descent

Coordinate descent is a method for optimising multivariate functions, wherein each variate is optimised one by one. We observe that equation 3.5 is function of two vector-valued inputs $\boldsymbol{w}$ and $\tilde{\boldsymbol{w}}$. We began by proposing to optimise this function using vector coordinate descent over these two inputs. It was hoped that this would produce a method with some properties of natural gradient descent, using only first-order gradient steps. However, this algorithm has a number of shortcomings towards this aim, which are useful for highlighting

how the methodology developed and contrasting with the final algorithm.

This algorithm, 'proximal coordinate descent', is presented in algorithm 1. The algorithm begins by initialising $\boldsymbol{w}$ and $\tilde{\boldsymbol{w}}$ to the same point in parameter space $\boldsymbol{w}^{(0)}$. We then enter an optimisation loop, starting with a gradient step on $\boldsymbol{w}$. Since $\boldsymbol{w} = \tilde{\boldsymbol{w}}$ and therefore $\nabla_{\boldsymbol{w}} R_{\text{prox}}(\boldsymbol{w}, \tilde{\boldsymbol{w}}) = \nabla_{\boldsymbol{w}} R(\boldsymbol{w})$ this is a normal gradient step on the risk function R. This is followed by a gradient step on $\tilde{\boldsymbol{w}}$; since the risk function does not depend on this input we take a step in the negative gradient of $\nabla_{\tilde{\boldsymbol{w}}} \lambda_{\mathcal{Y}} \rho(f(\boldsymbol{w}), f(\tilde{\boldsymbol{w}}))$. When making this step it is essential that $\tilde{\boldsymbol{w}}$ does not 'catch-up' with $\boldsymbol{w}$, such that when we next step $\boldsymbol{w}$ in the following loop iteration we have a non-zero proximity gradient $\boldsymbol{g}_{\boldsymbol{w}} = \nabla_{\boldsymbol{w}} J(\boldsymbol{w}) + \nabla_{\boldsymbol{w}} \lambda_{\mathcal{Y}} \rho(f(\boldsymbol{w}), f(\tilde{\boldsymbol{w}}))$. The algorithm then proceeds to alternate between $\boldsymbol{w}$ and $\tilde{\boldsymbol{w}}$ until a maximum number of iterations or some convergence measure has been reached.

Algorithm 1 Proximal Coordinate Descent

Input: $\eta =$ active learn rate, $\alpha =$ reference learn rate
1: initialise $\boldsymbol{w} = \tilde{\boldsymbol{w}} = \boldsymbol{w}^{(0)}$
2: **for** $s \leftarrow 0$ to S **do**
3: $\quad \boldsymbol{g}_{\boldsymbol{w}} \leftarrow \nabla_{\boldsymbol{w}} R_{\text{prox}}(\boldsymbol{w}, \tilde{\boldsymbol{w}})$
4: $\quad \boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \boldsymbol{g}_{\boldsymbol{w}}$
5: $\quad \boldsymbol{g}_{\tilde{\boldsymbol{w}}} \leftarrow \nabla_{\tilde{\boldsymbol{w}}} R_{\text{prox}}(\boldsymbol{w}, \tilde{\boldsymbol{w}})$
6: $\quad \tilde{\boldsymbol{w}} \leftarrow \tilde{\boldsymbol{w}} - \alpha \boldsymbol{g}_{\tilde{\boldsymbol{w}}}$
7: **end for**

This proximity term contribution to the gradient $\boldsymbol{g}_{\boldsymbol{w}}$ was proposed to introduce a penalty on large steps in the $\mathcal{Y}$-space, and therefore some qualities of natural gradients. The shortcoming of this algorithm to achieve this can be summarised into two components:

1. Taking a single gradient step on $\boldsymbol{w}$ is not sufficient to converge to optimality for the current $\tilde{\boldsymbol{w}}$.

2. Assuming that $\boldsymbol{w}$ is optimal for the current $\tilde{\boldsymbol{w}}$, the gradient of the proximity term $\nabla_{\tilde{\boldsymbol{w}}} \rho(f(\boldsymbol{w}), f(\tilde{\boldsymbol{w}}))$ will not approximate the natural gradient.

3.2.3 Optimality of Active Model

Recall from section 2.5 that we can consider proximal optimisation as solving a trust-region problem for some radius implicitly defined by our choice $\lambda_{\mathcal{Y}}$ [5]. We may define this optimality as the gradient $\nabla_{\boldsymbol{w}} R_{\text{prox}}$ converging to zero. To understand why achieving an optimal value of $\boldsymbol{w}$ is desirable, recall that natural gradients can be considered a second-order approximation to this trust region problem. Our approach instead iteratively employs first-order gradients to solve this trust-region problem. However, prior to convergence to optimality, we have no guarantees on the position of $\boldsymbol{w}$, nor the direction of $\nabla_{\tilde{\boldsymbol{w}}} ||f(\boldsymbol{w}) - f(\tilde{\boldsymbol{w}})||^2$, which we are hoping to approximate the natural gradient.

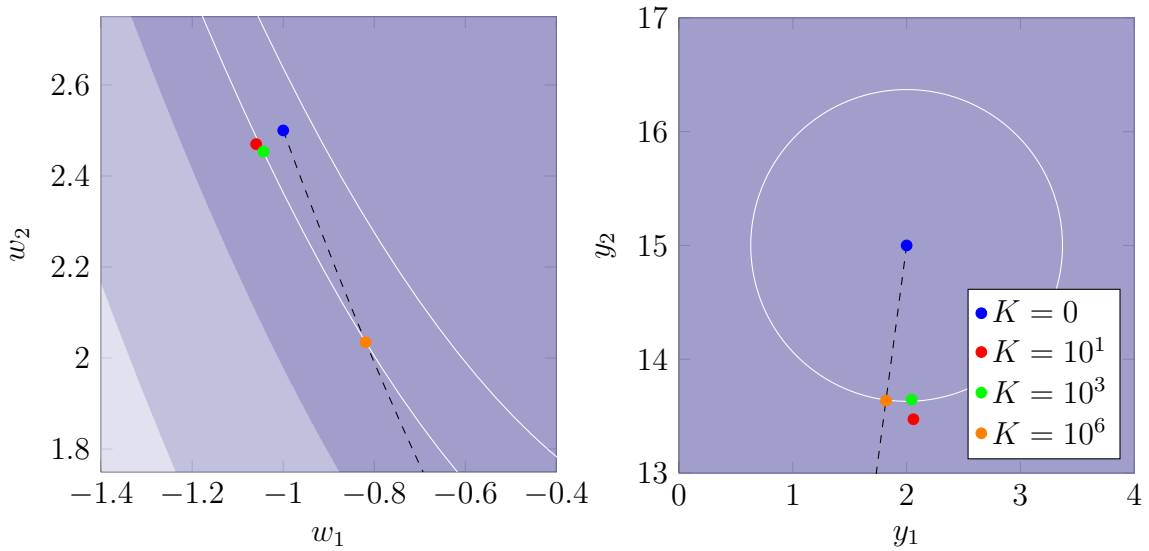

Figure 3.2: **A single inner loop step is not sufficient to converge to optimality**. Deadweight inner loop position after K steps on Rosenbrock function. Black dashed line is $\mathcal{Y}$-space gradient, the idealised natural gradient path. White solid line is a level set of the proximal objective for the current $\tilde{\boldsymbol{w}}$. Even after 1,000 steps the inner loop has not converged to the optimal position on the level set. After 100,000 iterations it has approximately converged to the intersection of the natural gradient with the level set.

That $\boldsymbol{w}$ does not achieve optimality in the original algorithm is easily demonstrated for the first step following initialisation, where as discussed previously we simply take a normal gradient step on $R(\boldsymbol{w})$. Assuming the natural gradient is not a scalar multiple of the Euclidean ($\mathcal{W}$) gradient, this step cannot optimally minimise the proximal risk for the current reference model $\tilde{\boldsymbol{w}} = \boldsymbol{w}^{(0)}$. Figure 3.2 displays the position of $\boldsymbol{w}$ after 10, 1,000 and 100,000 inner loop steps on the Rosenbrock function; evidently a significant number are required to converge to the optimal point, itself on the natural gradient path. Taking a sub-optimal step on $\boldsymbol{w}$, leads to a sub-optimal step on $\tilde{\boldsymbol{w}}$ when we step to minimise $\rho(f(\boldsymbol{w}), f(\tilde{\boldsymbol{w}}))$. This algorithm was found to approximately follow the path of gradient descent, hence the influence of the proximity term is therefore negligible.

3.2.4 Reference Model Objective

The original algorithm is a form of coordinate descent; we alternatively optimise $\boldsymbol{w}$ and $\tilde{\boldsymbol{w}}$. Crucially both are optimised to minimise the same objective function R_{prox}. We will now demonstrate how this leads the algorithm to fail to approximate natural gradient descent, even assuming an optimal $\boldsymbol{w}$. We will prove this by counterexample on the Rosenbrock function.

We have the squared distance as our loss function $\ell(f(\boldsymbol{w})) = ||f(\boldsymbol{w})||^2$, and the squared difference in $\mathcal{Y}$-space as our proximity term $\rho(f(\boldsymbol{w}), f(\tilde{\boldsymbol{w}})) = ||f(\tilde{\boldsymbol{w}}) - f(\boldsymbol{w})||^2$. Our idealised natural gradient trajectory is to follow the $\mathcal{Y}$-space gradient. For squared distance the $\mathcal{Y}$-space gradient is a straight line from our initialisation $f(\boldsymbol{w}^{(0)})$ to the origin $\mathbf{0}$. After

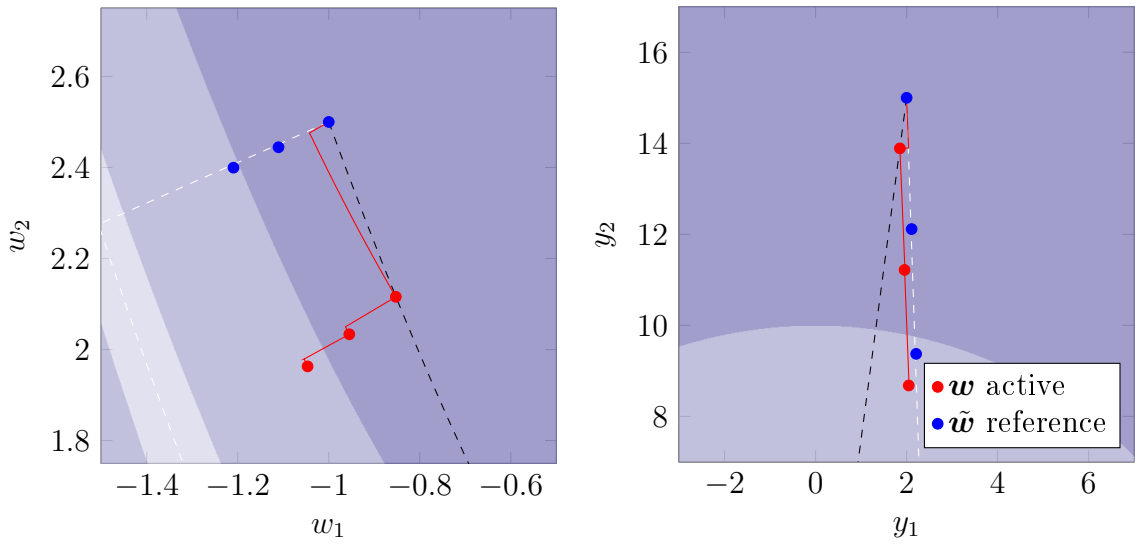

Figure 3.3: **Minimising incorrect objective with reference model results in approximately following the gradient descent path.** 3× deadweight reference $\tilde{\boldsymbol{w}}$ and corresponding optimal active model $\boldsymbol{w} = \boldsymbol{w}^*$. Blue dots are reference models, and red are optimal active models. Red line is the active model inner loop trajectory. For the first reference model, we observe that the optimal active model converges onto the natural gradient path (black dashed). However subsequent outer steps indicate the reference models to instead approximately follow the $\mathcal{W}$-space gradient path (white dashed) due to the incorrect minimisation objective for $\tilde{\boldsymbol{w}}$.

the inner loop optimisation, the active model $\boldsymbol{w} = \boldsymbol{w}^*$ is optimal for the reference model $\tilde{\boldsymbol{w}} = \boldsymbol{w}^{(0)}$ which remains at the initialisation. Since we are solving the Lagrangian for a given $\lambda_{\mathcal{Y}}$, corresponding to some trust region, this optimal $f(\boldsymbol{w}^*)$ will be approximately located on the $\mathcal{Y}$-space linear interpolation between our initialisation $f(\boldsymbol{w}^{(0)})$ and the origin $\mathbf{0}$; a point on our idealised update path. This implies that $f(\boldsymbol{w}^*) = \beta f(\boldsymbol{w}^{(0)})$, for some scalar $0 < \beta < 1$. The gradient $\nabla_{\tilde{\boldsymbol{w}}} R_{\text{prox}}(\boldsymbol{w}, \tilde{\boldsymbol{w}})$ evaluated at $\{\boldsymbol{w}, \tilde{\boldsymbol{w}}\} = \{\boldsymbol{w}^*, \boldsymbol{w}^{(0)}\}$ can therefore be derived as follows:

$$\begin{aligned}
\nabla_{\tilde{\boldsymbol{w}}} R_{\text{prox}}(\boldsymbol{w}, \tilde{\boldsymbol{w}})|_{\{\boldsymbol{w}^*, \boldsymbol{w}^{(0)}\}} &= \lambda_{\mathcal{Y}} \nabla_{\tilde{\boldsymbol{w}}} ||f(\boldsymbol{w}^{(0)}) - f(\boldsymbol{w}^*)||^2 \\
&= 2\lambda_{\mathcal{Y}} \boldsymbol{J}_f^T (f(\boldsymbol{w}^{(0)}) - f(\boldsymbol{w}^*)) \\
&= 2\lambda_{\mathcal{Y}} \boldsymbol{J}_f^T (f(\boldsymbol{w}^{(0)}) - \beta f(\boldsymbol{w}^{(0)})) \\
&= 2\lambda_{\mathcal{Y}} \boldsymbol{J}_f^T ((1 - \beta) f(\boldsymbol{w}^{(0)}))
\end{aligned} \tag{3.6}$$

As discussed previously, at initialisation $\boldsymbol{w} = \tilde{\boldsymbol{w}} = \boldsymbol{w}^{(0)}$, and therefore the proximal gradient reduces to the objective gradient $\nabla_{\boldsymbol{w}} R_{\text{prox}}(\boldsymbol{w}, \tilde{\boldsymbol{w}})|_{\{\boldsymbol{w}^{(0)}, \boldsymbol{w}^{(0)}\}} = \nabla_{\boldsymbol{w}} R(\boldsymbol{w})|_{\boldsymbol{w}^{(0)}}$. For ℓ as the squared distance, this results in the gradient defined in equation 3.7.

$$\nabla_{\boldsymbol{w}} R_{\text{prox}}(\boldsymbol{w}, \tilde{\boldsymbol{w}})|_{\{\boldsymbol{w}^{(0)}, \boldsymbol{w}^{(0)}\}} = \nabla_{\boldsymbol{w}} R(\boldsymbol{w})|_{\boldsymbol{w}^{(0)}} = 2\boldsymbol{J}_f^T f(\boldsymbol{w}^{(0)}) \tag{3.7}$$

From equations 3.6 and 3.7, equation 3.8 immediately follows; in other words, after the inner loop completes, when we update $\tilde{\boldsymbol{w}}$ we simply step on a scaled version of the $\mathcal{W}$-space gradient of $R(\boldsymbol{w})$ at this point. Recalling that we intend to solve a sequence of trust-region optimisation problems; this has the effect of establishing the following trust region a 'normal' gradient descent step from the initialisation. As successive steps take place we begin approximately following $\mathcal{W}$-space gradient descent, where the only influence introduced by the proximal risk is a down-scaled gradient. In figure 3.3, we present three outer loop steps using this algorithm, observing the reference model iterates (blue dots) to follow approximately the $\mathcal{W}$-space gradient descent path (white, dashed). Preliminary tests indicated this algorithm to become unstable after reaching the valley floor on the Rosenbrock function.

$$\nabla_{\tilde{\boldsymbol{w}}} R_{\text{prox}}(\boldsymbol{w}, \tilde{\boldsymbol{w}})|_{\{\boldsymbol{w}^*, \boldsymbol{w}^{(0)}\}} = \lambda_{\mathcal{Y}}(1-\beta)\nabla_{\boldsymbol{w}} R(\boldsymbol{w})|_{\boldsymbol{w}^{(0)}} \tag{3.8}$$

3.3 Proposed Algorithm

We have established that proximal coordinate descent suffers from two limitations; a lack of convergence of the inner loop, and optimising the reference model using an incorrect objective. The first can be resolved by introducing an inner optimisation loop, in which we iteratively optimise $\boldsymbol{w}$ until it is optimal for the current $\tilde{\boldsymbol{w}}$. The second is solved by stepping $\tilde{\boldsymbol{w}}$ to minimise the squared Euclidean distance in the $\mathcal{W}$-space as opposed to the $\mathcal{Y}$-space. Intuitively, after solving the trust-region problem we have established the $\mathcal{W}$-space direction in which we should travel to minimise our objective within our given trust region.

Algorithm 2 Deadweight Optimisation

Input: $\eta =$ active learn rate, $\alpha =$ reference learn rate, $\epsilon =$ converge threshold

initialise $\boldsymbol{w} = \tilde{\boldsymbol{w}} = \tilde{\boldsymbol{w}}^{(0)}$
for $s \leftarrow 0$ to S **do**
  while $||\boldsymbol{g}_{\boldsymbol{w}}|| > \epsilon$ **do**
    $\boldsymbol{g}_{\boldsymbol{w}} \leftarrow \nabla_{\boldsymbol{w}} R_{\text{prox}}(\boldsymbol{w}, \tilde{\boldsymbol{w}})$
    $\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \boldsymbol{g}_{\boldsymbol{w}}$
  end while
  $\boldsymbol{g}_{\tilde{\boldsymbol{w}}} \leftarrow \boldsymbol{w} - \tilde{\boldsymbol{w}}$
  $\tilde{\boldsymbol{w}} \leftarrow \tilde{\boldsymbol{w}} - \alpha \boldsymbol{g}_{\tilde{\boldsymbol{w}}}$
end for

Addressing these limitations of proximal coordinate descent leads to the 'base' deadweight algorithm, presented in algorithm 2. Compared to algorithm 1, we observe that the following changes:

1. We employ an inner loop to optimise $\boldsymbol{w}$ until it converges to optimality for the current $\tilde{\boldsymbol{w}}$, as measured by the gradient norm reaching a defined threshold $||\boldsymbol{g}_{\boldsymbol{w}}|| < \epsilon$.

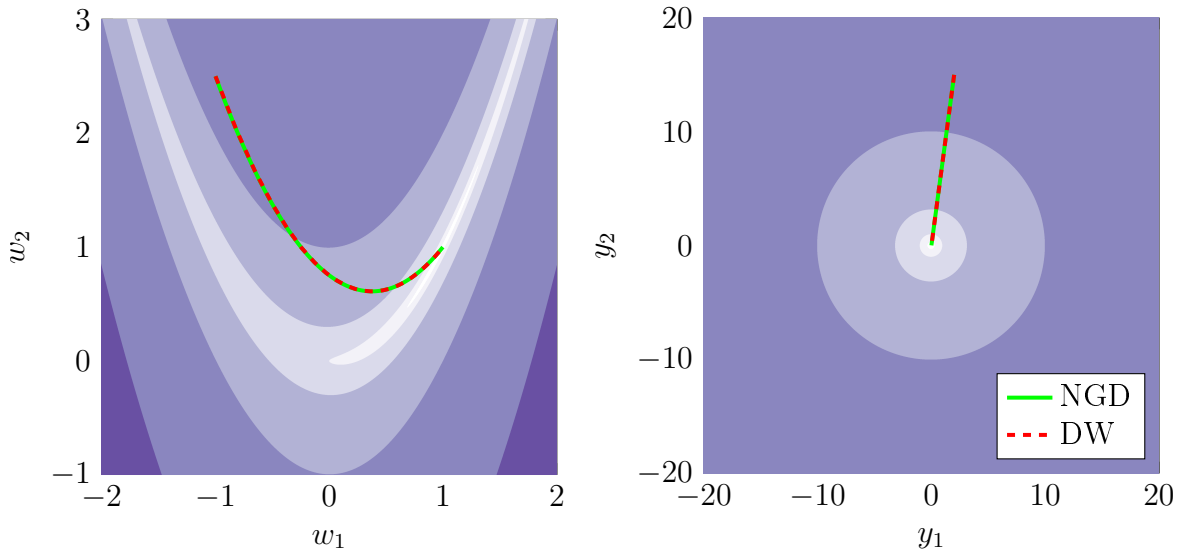

Figure 3.4: **Deadweight follows the natural gradient trajectory**. The natural gradient trajectory is a straight line from initialisation to the origin, appearing as a curved path in the $\mathcal{X}$-space. Deadweight closely follows this path, providing qualitative evidence the algorithm is operating as intended.

2. Once $\boldsymbol{w}$ has reached optimality, we step $\tilde{\boldsymbol{w}}$ to minimise the squared $\mathcal{W}$-space distance $||\boldsymbol{w}-\tilde{\boldsymbol{w}}||^2$. Since this is defined directly on the $\mathcal{X}$-space this has the added advantage of not requiring a costly backpropagation step.

We note that the second adjustment implies that we are no longer doing coordinate descent; both $\boldsymbol{w}$ and $\tilde{\boldsymbol{w}}$ now have distinct objective functions. In figure 3.4 we provide qualitative evidence of deadweight closely approximating the natural gradient path on the Rosenbrock function.

3.3.1 Hyperparameters

In algorithm 2, we see that deadweight has three hyperparameters; η is the active model learning rate, α is the reference model learning rate, and ϵ is the convergence criterion. In figure 3.5 we see that increasing the leniency of the convergence criterion leads to deviation from the natural gradient trajectory, but brings an overall reduction in inner steps required to converge. These results suggest there may be a trade-off between the quality of the natural gradient approximation, and the computational cost per outer step. Furthermore, in figure 3.6 we observe that increasing the reference model learning rate has negligible influence on the outer loop trajectory, whilst significantly decreasing the number of inner loop steps to convergence.

3.3.2 Variants and Optimisations

We propose the following optimisations to be made to the base deadweight defined in algorithm 2. All three optimisation are designed to accelerate the convergence of the inner

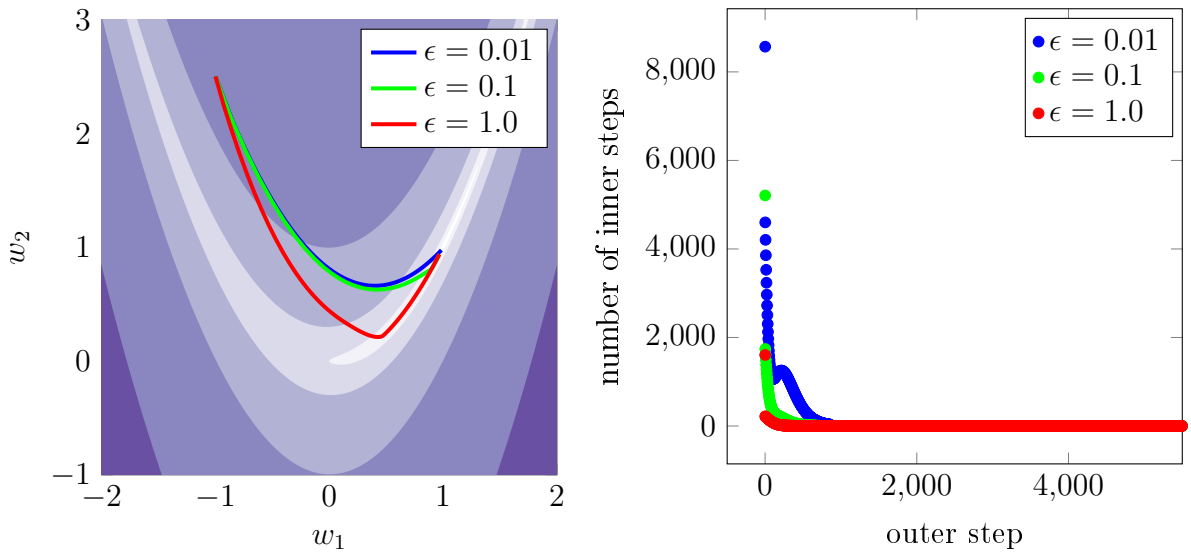

Figure 3.5: **Increasing the leniency of the inner loop convergence criterion leads to a worse approximation of the natural gradient, but a reduced number of inner steps overall.** The strictest convergence criterion $||\boldsymbol{g_w}|| < \epsilon = 0.01$ requires 634,370 inner steps to reach the minimum. Increasing ϵ to 0.1 introduces a minor deviation to the trajectory, but reduces the inner steps to 166,709. Increasing even further to 1.0, leads to a major deviation in the trajectory, although even further reduces the inner steps to 75,476. In this case the majority of the inner steps are used following the gradient descent path along the bottom of the valley.

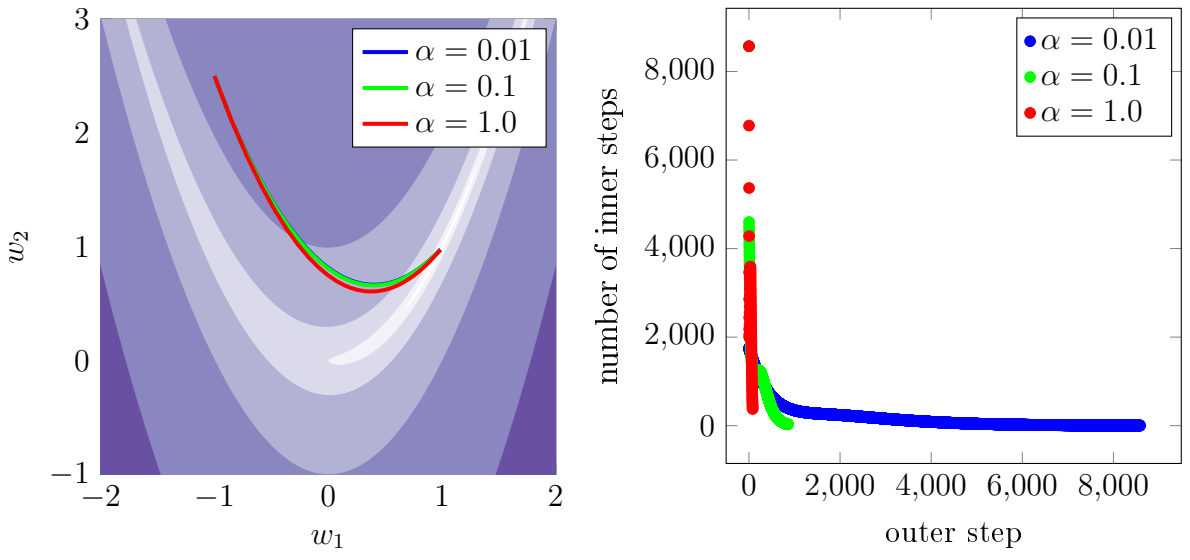

Figure 3.6: **Increasing the reference model learning rate significantly reduces the number of inner loop steps, with minor influence on the outer loop trajectory.** Setting α to 0.01, 0.1 and 1.0 leads to 1,507,956, 634,370, and 187,800 total number of inner steps respectively. Whilst using a high α leads to more inner steps being taken per outer step, overall the decrease in outer steps outweighs this advantage; it is more efficient to take larger (and more expensive) but fewer outer steps.

loop, which was identified as the most important factor in convergence rate, particularly while maintaining an accurate approximation of natural gradients.

1. Inner loop momentum.
2. Free-step
3. Line search along $\boldsymbol{g}_{\tilde{\boldsymbol{w}}}$ for the 'free-step' magnitude.

Momentum

As discussed in section 2.5, momentum is an effective means of improving the convergence of gradient descent. Since our inner loop is simply a gradient descent loop on a modified objective, it is natural to investigate its efficacy in this case. One potentially important subtlety of applying momentum to this inner loop objective is whether the momentum should persist between inner loops, or be reinitialised to zero at the start of each inner loop. Intuitively, this may be advantageous if the trajectory is reasonably consistent between inner loops. We organise momentum options as a new hyperparameter '`momentum`' with three possible values (`none`, `reinitialise`, `persistent`). In figure 3.7 we present results for each of these options. Evidently, using momentum approximately halves the number of inner loop steps. There is no advantage in this case to using persistent momentum. Momentum requires a marginal increase in per-iteration time complexity, at the cost of maintaining an input vector $\boldsymbol{w}$ in memory.

Free Step

Another natural optimisation to be made is to take a 'free step' on the active model $\boldsymbol{w}$ whilst we update the reference model $\tilde{\boldsymbol{w}}$ during an outer step. More explicitly, whilst updating $\tilde{\boldsymbol{w}} \leftarrow \tilde{\boldsymbol{w}} - \alpha \boldsymbol{g}_{\tilde{\boldsymbol{w}}}$, we also update $\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha \boldsymbol{g}_{\tilde{\boldsymbol{w}}}$. The following inner loop then proceeds with the active model initialised to this new point. Intuitively, if this new initialisation point is closer to the optimal point, fewer inner loop steps may be required. Introducing a line search is a further optimisation that can be made; instead of using a constant α, and hence using the same step size as the reference model, we can line perform a line search to minimise the proximal objective R_{prox}. In this work we elect for a simple golden section search [42]. We organise these options as another new hyperparameter '`freestep`' with three possible values (`none`, `basic`, `linesearch`). In figure 3.8, we present results for these options. We see that either form of free-step reduces the number of inner steps by approximately a factor of four, however line search in this case is inferior. Free-step itself requires a negligible increase in the time complexity of an outer step, however using golden section search requires several function evaluations which increases the complexity significantly. However, these function evaluations do not require gradient information nor backpropagation, so it is plausible that reducing the inner loop steps using line search could reduce the overall wall-clock time.

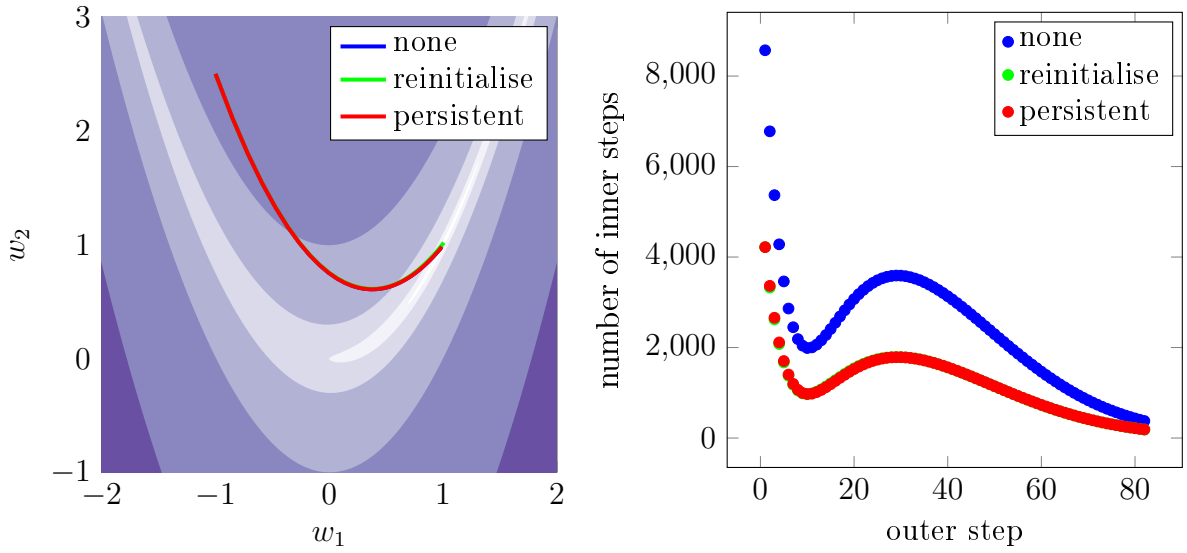

Figure 3.7: **Employing momentum greatly reduces the number of inner loop iterations, with negligible influence on the outer loop trajectory.** Without momentum (`momentum = none`) requires 187,800 inner steps to converge to the minimum, this is reduced to 93,268 by employing momentum (`momentum = reinitialise`). Using a persistent momentum instance between inner loops (`momentum = persistent`) leads to a minor increase of 23 in the overall inner loop steps when compared with reinitialised momentum. Here all three $\mathcal{W}$-space plots are imperceptible from each other, the inner step plots of reinitialise and persistent are also highly similar.

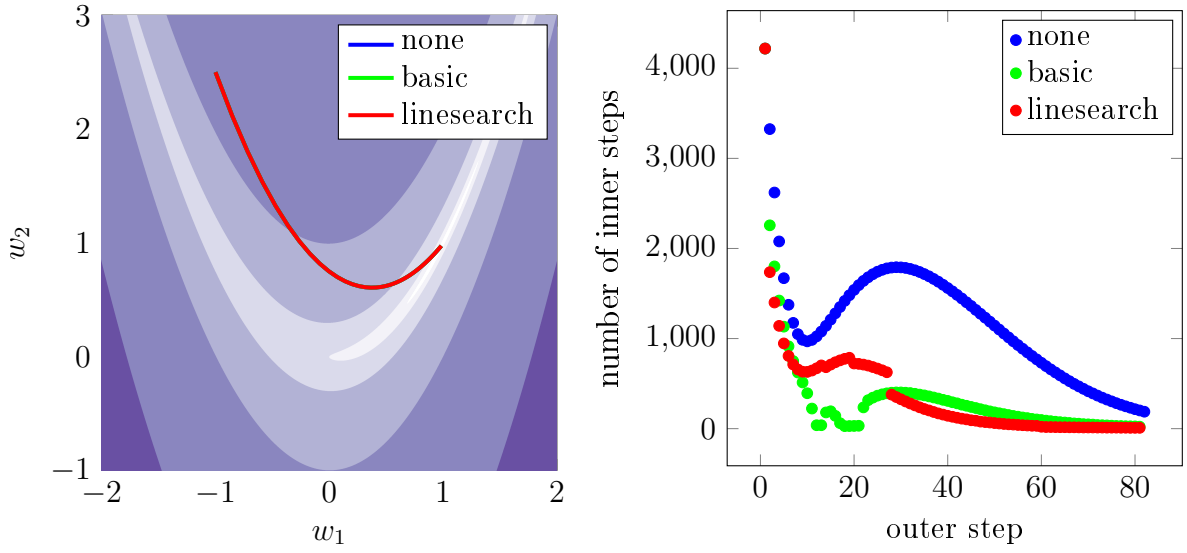

Figure 3.8: **Employing a 'free step' on the active model notably decreases the number of inner loop steps required to converge to the minimum.** A free step means whilst updating the reference model $\tilde{\boldsymbol{w}} \leftarrow \tilde{\boldsymbol{w}} - \alpha \boldsymbol{g}_{\tilde{\boldsymbol{w}}}$, we also update the active model $\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha \boldsymbol{g}_{\tilde{\boldsymbol{w}}}$. Without a free step (`freestep = none`) convergence requires 93,268 inner loop steps. Using the basic free step (`freestep = basic`) reduces this to 22,306, with line search (`freestep = linesearch`) is slightly worse with 25,412. Here all three $\mathcal{W}$-space plots are again imperceptible from each other.

3.4 Deadweight and Proximal Policy Optimisation

After resolving the limitations of the original proposal in section 3.2, deadweight optimisation bears similarities to proximal policy optimisation (PPO), an optimisation method from reinforcement learning [16]. Further details on PPO are provided in section 5.4. Here we emphasise the distinctions between these two methods:

- The specific construction of the surrogate objective discussed in section 5.4 is considered part of the PPO algorithm. Since deadweight is defined for supervised learning, we do not need to construct a surrogate objective.
- PPO iterates on the same 'batch' of samples (trajectories) for an entire inner loop [19], in contrast deadweight cycles through batches as in SGD. This increases the data throughput of deadweight, ideal for supervised learning where the dataset is a fixed collection of samples.
- Since the deadweight inner loop iterates over different batches from the dataset we are not concerned about an inner loop overfitting a specific batch of samples. We may therefore optimise the inner loop to convergence, and assume the solution to be approximately representative of the empirical data distribution.
- PPO takes a fixed number of K inner loop iterations [16]. This has many advantages, including simplicity and avoiding the possibility of becoming stuck inside an inner loop. However, always taking a fixed number of iterations may have the following disadvantages: (i) where the inner loop is easy to solve, we may be taking unnecessary inner loop steps (ii) in some cases the inner loop may require a large number of iterations to solve, and terminating at a fixed number will lead to a sub-optimal outer step. By using some measure of convergence on the inner loop, potentially with a 'back-up' maximum number of iterations, deadweight is able to mitigate the issue of wasting iterations, and only exit the inner loop unconverged if we reach an iteration number deemed excessive.
- Since deadweight optimises the inner loop to convergence, using several batches from the data distribution, we are able to make optimisations such as `freestep`. We believe such optimisations would not work in PPO due to the small sample size used in the inner loop, and the lack of convergence of the inner loop.

Chapter 4

Evaluation

In Chapter 3 we defined the proposed deadweight optimisation, a novel method for proximal optimisation in supervised learning. We now proceed to evaluate deadweight on two image classification datasets; MNIST [17] and CIFAR-10 [18]. Image classification is a commonly used task for the evaluation of optimisation algorithms [7, 8, 43], and for empirical studies of generalisation [11, 44, 45].

4.1 Procedure

In all experiments we train with cross-entropy loss, without weight decay. Weight decay is a form of explicit regularisation, and was omitted to contrast the implicit generalisation performance of deadweight with that of the baselines. Where mini-batches are employed we use a batch size of 1024. We implement all evaluation procedure using PyTorch [46].

We employ stochastic gradient descent (SGD), and Hessian-Free [6] as baselines. Stochastic gradient descent is ubiquitous within deep learning, with an implicit bias towards generalisation [12, 14, 35]. It therefore represents a strong baseline against which to compare deadweight and Hessian-Free. Hessian-Free is a second-order optimisation algorithm [6] that approximates the generalised Gauss-Newton method. Hessian-Free is not commonly employed within deep learning; nonetheless, it represents an established baseline for second-order optimisation [19]. For further details on Hessian-Free, refer to section 5.2. Since the inner loop of deadweight is simply SGD on a modified objective, we may directly compare the number of gradient steps between these algorithms. Hessian-Free is based on the conjugate gradient method, wherein one step is approximately equal in complexity to a backpropagation step [21]. For this reason, we may also relate the complexity of Hessian-Free to that of SGD.

4.1.1 Datasets

We use both the MNIST [17] and CIFAR-10 datasets [18]. The MNIST dataset contains 70,000 samples of handwritten digits. In modern machine learning, MNIST can only be considered a toy example; simply put, it is too simple of a task from which to draw meaningful conclusions. However, it is highly suitable as a first example of deep learning; it being simple and easy to solve enables us to evaluate optimisation algorithms in the full-batch deterministic setting, removing complications introduced by stochastic gradients. We employ MNIST without data augmentation, and reserve 5,000 of the training samples as a validation dataset for hyperparameter optimisation.

In contrast, CIFAR-10 is a more difficult task, containing 60,000 images corresponding to 10 classes of physical object. Whilst not at the scale of ImageNet [47], CIFAR-10 is suitably challenging to draw conclusions from its results, leading to its use in many empirical studies of generalisation [33, 44, 45]. Following such studies, we employ random pad and crop, random rotations, and random flip augmentations. We once again reserve 5,000 of the training samples as a validation dataset.

4.1.2 Architectures

We evaluate deadweight using three architectures; a multi-layer perceptron (MLP), LeNet [48], and a larger CNN denoted as Simple-CNN [44]. MLPs are the simplest, and most general, neural network architecture. The MLP architecture employed has two hidden layers of sizes $[300, 100]$ and uses ReLU activation. This MLP is suitable for use on the MNIST dataset, achieving in excess of 98% test accuracy with SGD. However, it is not suitable for CIFAR-10, achieving approximately 50% test accuracy; MLP results on CIFAR-10 were not explored further for this reason.

LeNet is an early example of a convolutional neural network (CNN), proposed by Lecun et al. [48]. CNNs are ubiquitous within computer vision due to the equivariance of the convolution operation to circular translations. In practice this implies that a feature can be detected irrespective of its spatial position, a frequently useful geometric prior for image data [49]. LeNet was selected as a basic first CNN and achieves greater than 99% test accuracy on MNIST and approximately 78% test accuracy of CIFAR-10 with SGD. Results on both datasets were therefore explored and presented.

Simple-CNN is a five-layer convolutional neural network [44]. Simple-CNN was originally defined to include batch normalisation [37]. As discussed in section 2.3, normalisation is known to address many of the same limitations as second-order optimisation, hence was omitted from the architecture. Simple-CNN achieves 87% test accuracy on CIFAR-10 with SGD; whilst not state-of-the-art this is closer to modern standards than the 78% of LeNet. This higher performance is primarily a function of the size of the networks; LeNet contains only 83,000 parameters in contrast to Simple-CNN's 1,500,000.

4.1.3 Hyperparameter Optimisation

Achieving the highest performance for an optimisation task requires hyperparameter optimisation. Hyperparameters are typically optimised using Bayesian optimisation or grid search. The principal advantage of Bayesian optimisation over grid searching is that the optimal hyperparameters may be achieved in fewer evaluations, reducing the computational cost [50]. Additionally, Bayesian optimisation is not restricted to a finite grid of values, permitting more precise identification of optimal hyperparameters. However, since deadweight is not yet well understood, grid searching provides an understanding of how the algorithm behaves over a wide range of configurations; for this reason we elect to use grid search in this work.

There are several ways in which we could define the hyperparameter optimisation objective. The simplest would be the number of iterations until convergence to a given value of training loss; exactly the quantity we are optimising, albeit through stochastic estimation. However, as discussed in section 2.1, machine learning differs from classical optimisation by the objective of generalisation. We therefore define the grid search criterion as the number of steps until a threshold of validation accuracy is achieved; this value is determined from a simple SGD run. In many cases Hessian-Free does not converge to the target value within the budget of 10,000 conjugate gradient steps; in these cases the optimal hyperparameter values are identified as those that maximise the validation accuracy.

When using multiple seeds, with SGD it is trivial to take the mean of the metrics, since evaluation is performed at regular intervals. However, with both Hessian-Free and deadweight, it is more strongly motivated to trigger evaluation when performing an outer step, as opposed to after a fixed number of inner iterations. This presents a minor obstacle for averaging over seeds since the outer steps will occur after different numbers of inner steps. To overcome this, we linearly interpolate between the evaluation steps giving an approximate evaluation value for every inner step, before taking the mean of these values. For all presented results the mean of five seeds is used, with $\pm\sigma$ shaded.

The grid search configurations for each of the evaluated algorithms follow. The full grid searches were performed for all experiments except Simple-CNN CIFAR-10, where the computational cost was deemed prohibitively high. For this experiment a reduced grid search was defined, informed by the optimal configurations on the simpler tasks.

Stochastic Gradient Descent

We grid search the learning rate over the values $\{0.01, 0.04, 0.1, 0.4\}$, and momentum over the values $\{0.0, 0.3, 0.6, 0.9\}$. In addition to momentum, learning rate annealing is another technique for improving convergence [51]. We used cosine annealing, and tune the annealing period length over the values of $\{1000, 2000, 5000, 10000\}$. Larger values lead to annealing being performed more slowly, training is terminated after the annealing

period. Whilst using higher learning rates with momentum, it is desirable to 'warm-up' at the start of training; we use a fixed 5% of total iterations for warm-up [51]. The grid search used to optimise SGD is summarised in table 4.1. We use 5 seeds for this grid search.

Hessian-Free

We used an existing implementation of HF [52]. The main hyperparameters were identified to be the learning rate and associated line search option, as well as the damping initialisation. Where a line search is performed the learning rate is used as an initial value. We grid search over the following learning rate values $\{0.001, 0.01, 0.1, 1.0\}$, and the Boolean flag `linesearch`. For the damping, we grid search over the same $\{0.001, 0.01, 0.1, 1.0\}$. We note that the damping performed within Hessian-Free is adaptive through a heuristic, hence the damping value is simply used as an initialisation for this heuristic [6]. The grid search used to optimise Hessian-Free is summarised in table 4.2. We use 5 seeds for this grid search.

Deadweight

Deadweight introduces a significant number of hyperparameters. For this reason, the grid search was split into two components. In the first, we optimise the hyperparameters broadly defining the inner loop problem to be solved. Once the optimal values for these hyperparameters have been identified, we perform a second grid search in which we optimise the hyperparameters controlling optimisations to the inner loop.

Since the deadweight inner loop is simply SGD on a modified objective, we have values for learning rate and momentum. Learning rate was searched over the values $\{0.01, 0.04, 0.1\}$, whereas momentum was fixed at 0.9. We then have hyperparameters $\lambda_{\mathcal{Y}}$ and $\lambda_{\mathcal{X}}$, corresponding to the proximity weighting and damping respectively. We grid search $\lambda_{\mathcal{Y}}$ over the values $\{0.0, 0.1, 1.0, 10.0\}$ and $\lambda_{\mathcal{X}}$ over $\{0.0, 0.01, 0.1, 1.0\}$. We lastly have the convergence criterion, achieving which triggers an outer step, which is tuned over $\{0.01, 0.04, 0.1, 0.4\}$. Informed by preliminary experiments, and qualitative results on the Rosenbrock function we fix the reference model learning rate $\alpha = 1.0$. For this first grid search, `freestep` was set to `basic` and `momentum` to `reinitialise`. The decision to use `freestep` = `basic` was based on preliminary experiments indicating this to improve convergence, we will later see this to not be uniformly the case. Training deadweight with `momentum` = `none` lead to very poor convergence, hence the use of `momentum` = `reinitialise`. Due to the large size of this search grid (192 evaluations per seed), we use only three seeds.

Upon identifying the optimal configuration in the previous grid search, we commence a second grid search. This grid search is over the algorithm optimisations, namely; `freestep` (with options `none`, `basic` and `linesearch`) and momentum (with options `reinitialise` and `persistent`). This leads to a grid of six elements, which we search using five seeds.

Hyperparameter	Grid Values
Learning Rate	0.01, 0.04, 0.1, 0.4
Momentum	0.0, 0.3, 0.6, 0.9
Annealing Iterations	1000, 2000, 5000, 10000

Table 4.1: **Stochastic gradient descent grid search.** The Simple-CNN experiments used a fixed learning rate of 0.1 and momentum of 0.9 and grid searched only over annealing iterations.

Hyperparameter	Grid Values
Learning Rate	0.001, 0.01, 0.1, 1.0*
Line Search	True, False
Damping	0.001, 0.01, 0.1*, 1.0*

Table 4.2: **Hessian-Free grid search.** Values with * were omitted for the Simple-CNN experiment. We observe that this corresponds with the highest learning rate and damping values.

Hyperparameter	Grid Values
Learning Rate	0.01, 0.04, 0.1
$\lambda_{\mathcal{Y}}$	0.0*, 0.1*, 1.0, 10.0
$\lambda_{\mathcal{X}}$	0.0*, 0.01, 0.1*, 1.0*
ϵ	0.01*, 0.04, 0.1, 0.4

Table 4.3: **Deadweight main grid search.** Values with * were omitted for the Simple-CNN experiment. We observe this to eliminate the lower $\lambda_{\mathcal{Y}}$ values corresponding to large trust regions. We also observe the lowest convergence criterion was eliminated, due to it being too strict and difficult to achieve when using stochastic gradients.

4.2 Results

Five experimental configurations were conducted in total; MLP MNIST Full-batch (FB), MLP MNIST, LeNet MNIST, LeNet CIFAR-10 and Simple-CNN CIFAR-10. For all MNIST experiments the convergence threshold used for hyperparameter optimisation was 98% validation accuracy. For LeNet CIFAR-10 and Simple-CNN CIFAR-10, we used 75% and 85% validation accuracy respectively. The results for these experiments are presented in figures 4.2 through 4.5. The optimal hyperparameters for each experiment and algorithm are presented in tables 4.4 to 4.6.

4.2.1 MNIST

The simplest experiment is MLP MNIST full-batch. This experiment mostly serves as a 'sanity check' that deadweight is a viable optimisation algorithm for deep learning without the complication of stochastic gradients. We observe the results in figure 4.1 to support this claim. In this experiment gradient descent converges fastest, followed by deadweight and lastly Hessian-Free. For deadweight we note a large increase in test loss after the training accuracy converges; however, this does not correspond to a significant decrease in test accuracy, with all algorithms achieving comparable performance on this metric.

The next experiment MLP MNIST mini-batch introduces stochastic gradients to the optimisation problem. Whilst still a toy example, this is closer to modern deep learning practice where dataset sizes preclude the use of full-batch training. The results in figure 4.2 indicate deadweight to be functional when using mini-batches of data. It does however highlight a potential pathology in the stochastic case; all measured metrics plateau sub-optimally relative to SGD. For this experiment Hessian-Free does not meet the target of 98% validation accuracy, hence its hyperparameters were identified as those that maximised the validation accuracy. The slow convergence of Hessian-Free is more pronounced in this experiment than in the full-batch case.

LeNet MNIST proceeds by replacing the MLP in the previous experiment with LeNet. This experiment remains simple and serves to demonstrate deadweight on a convolutional architecture. In figure 4.4 we observe deadweight to converge at a comparable rate to SGD for 1,000 gradient steps. However, beyond 1,000 gradient steps the performance of deadweight decreases on all measured metrics. In this case, Hessian-Free was able to reach the target 98% accuracy, however convergence is notably slow in comparison to SGD and deadweight.

	MNIST			CIFAR-10	
	MLP (FB)	MLP	LeNet	LeNet	S-CNN
Learning Rate	0.4	0.4	0.1	0.1	0.1
Momentum	0.9	0.6	0.9	0.9	0.9
Annealing Iterations	2000	1000	5000	5000	5000

Table 4.4: **Stochastic gradient descent optimal hyperparameters.** The objective of minimising iterations to the target validation accuracy favours high learning rates with momentum. The convolutional architectures require longer training periods than the MLP experiments.

	MNIST			CIFAR-10	
	MLP (FB)	MLP	LeNet	LeNet	S-CNN
Learning Rate	1.0	0.01	0.1	0.1	0.1
Line search	True	False	True	True	True
Damping	0.1	0.001	0.001	0.01	0.001

Table 4.5: **Hessian-Free optimal hyperparameters.** The learning rates selected are generally higher values, and line search is useful in all but MLP MNIST (mini-batch).

	MNIST			CIFAR-10	
	MLP (FB)	MLP	LeNet	LeNet	S-CNN
Learning Rate (η)	0.1	0.1	0.01	0.01	0.1
$\mathcal{Y}$-space penalty ($\lambda_{\mathcal{Y}}$)	1.0	10.0	10.0	1.0	1.0
$\mathcal{W}$-space penalty ($\lambda_{\mathcal{W}}$)	0.01	0.01	0.01	0.01	0.01
Converge Criterion (ϵ)	0.04	0.4	0.4	0.04	0.1
Free Step	Search	None	None	None	None
Momentum	Persist	Persist	Persist	Persist	Reinit
Max Inner				100	100

Table 4.6: **Deadweight optimal hyperparameters.** The $\lambda_{\mathcal{Y}}$ values are consistently in the higher range of those available, in contrast the $\lambda_{\mathcal{W}}$ values are in their lower range. Line searching is only optimal for the full-batch MNIST experiment, whereas persistent momentum is preferred in all experiments except full-batch MNIST. The top and middle sections relate to first and second grid searches. Max inner was not grid searched and introduced only for the CIFAR-10 experiments.

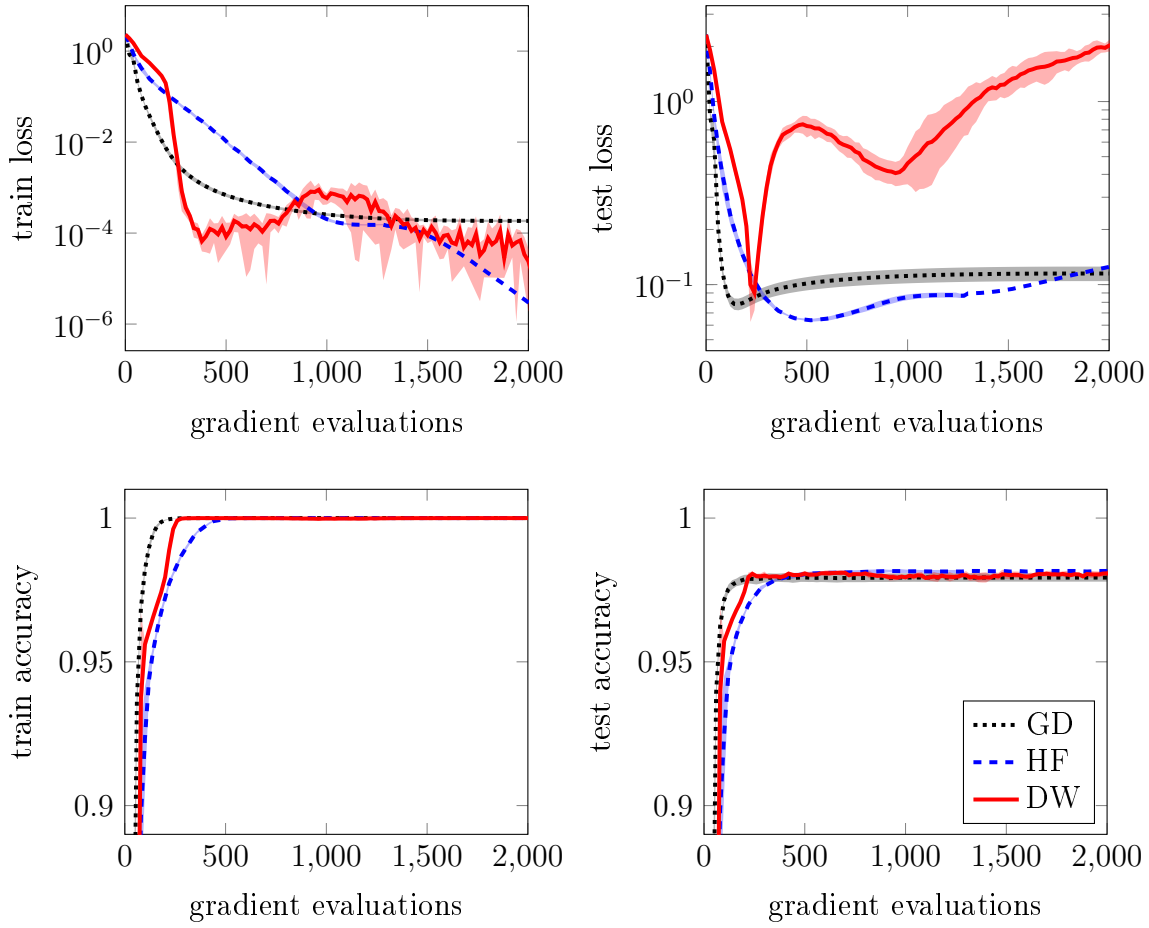

Figure 4.1: **Full-batch ($B = 55000$) MNIST training with MLP. Deadweight is a viable optimisation algorithm for deep learning**. Deadweight is successful at minimising the training loss, converging to lower training loss than gradient descent in fewer iterations. However, as measured by loss, the generalisation performance overfits strongly, achieving a comparable value to gradient descent before increasing rapidly. In terms of accuracy, both training and test performance converge faster for deadweight than Hessian-Free but slower than gradient descent. We further note that the large increase in test loss does not correspond to a large decrease in test accuracy.

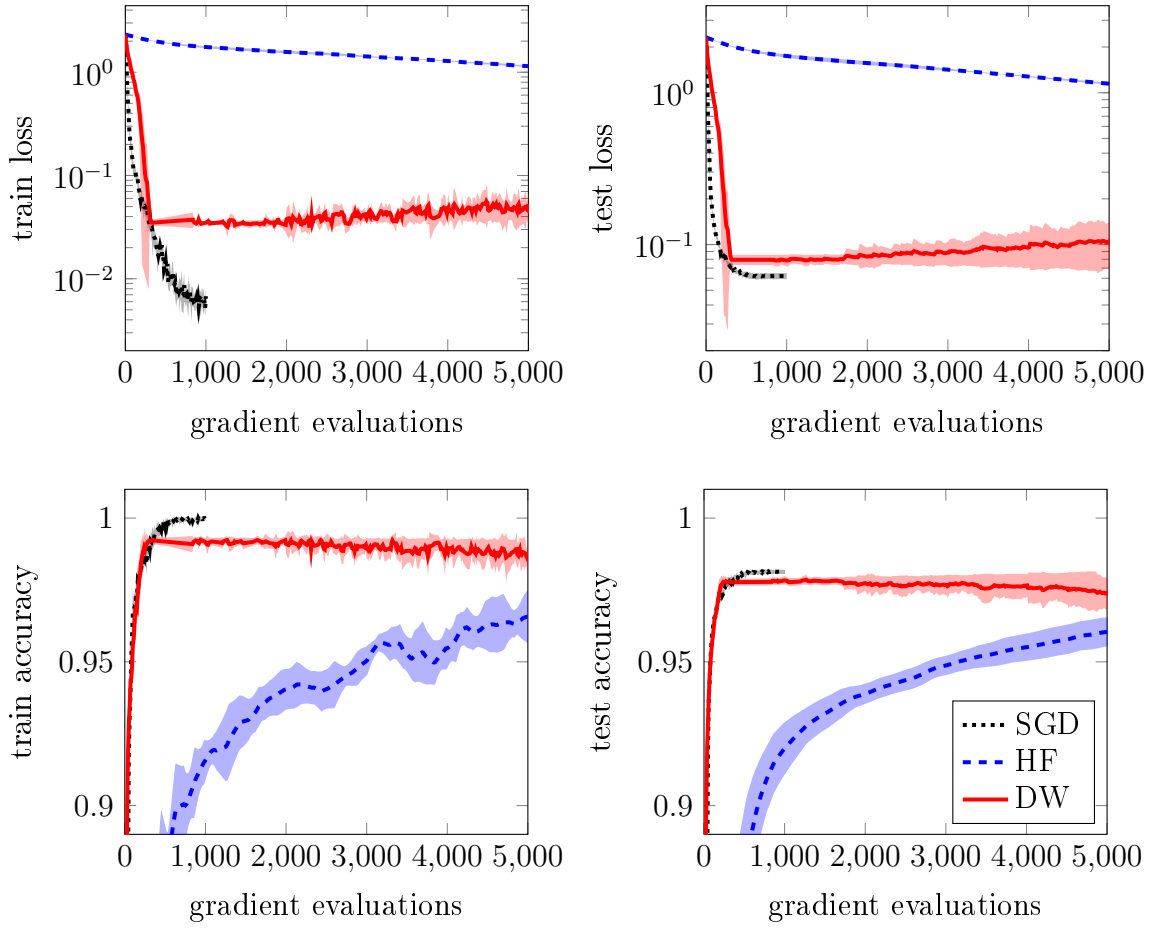

Figure 4.2: **Mini-batch ($B = 1024$) MNIST training with MLP. For stochastic gradients deadweight converges quickly but plateaus sub-optimally in all measured metrics**. Deadweight starts by converging at a comparable rate to SGD, but plateaus before reaching 100% training accuracy and at an inferior test accuracy. Hessian-Free converges much more slowly and does not achieve comparable performance to either SGD nor deadweight within 5,000 iterations. The SGD plots terminate at 1,000 iterations due to this being the number of annealing iterations identified by hyperparameter optimisation.

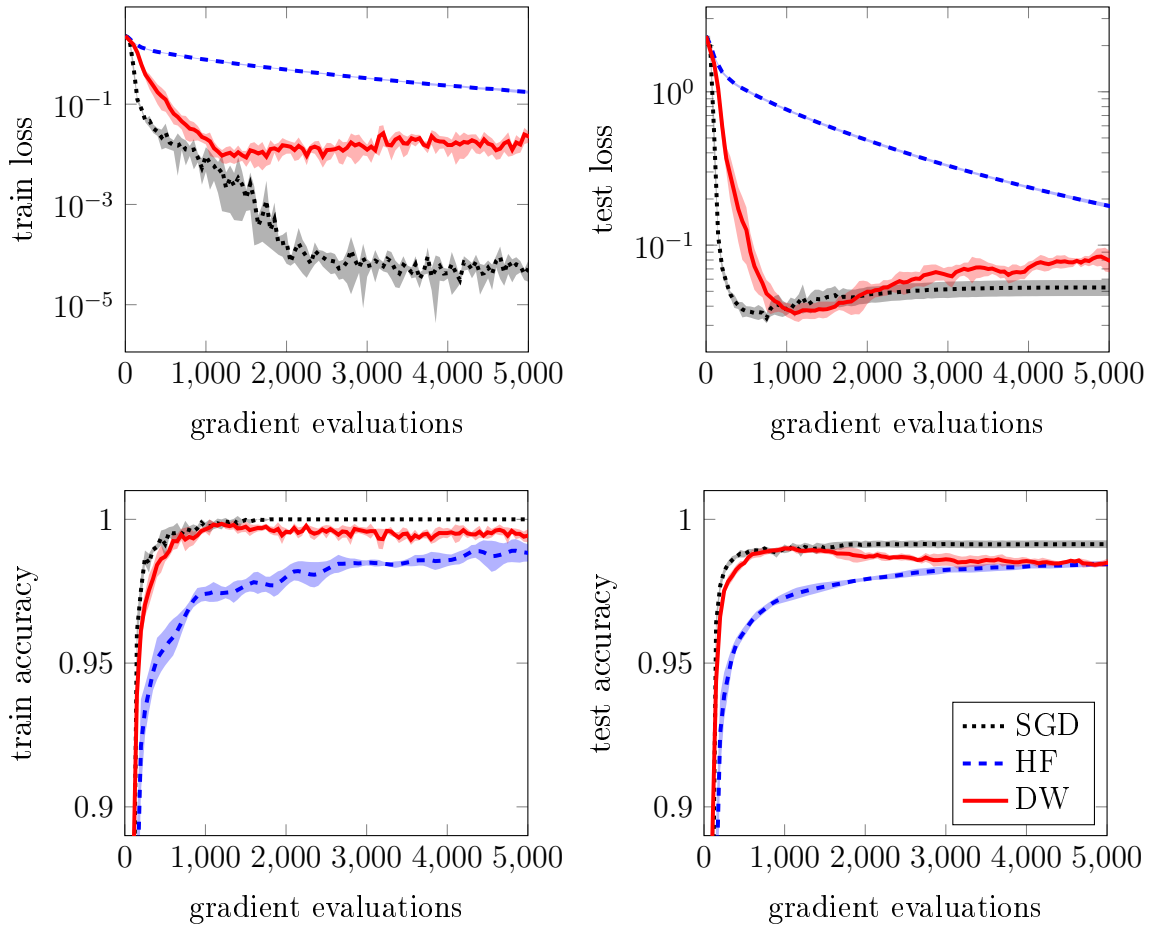

Figure 4.3: **Mini batch ($B = 1024$) MNIST training with LeNet. The performance of deadweight is comparable to SGD before a prolonged deterioration occurs.** After 1,000 gradient steps, deadweight and SGD have comparable performance on all metrics. Beyond 1,000 gradient steps, SGD continues to improve slightly whereas the performance of deadweight decreases on all measured metrics. Once again Hessian-Free (HF) converges slowly, only approaching comparable performance after 5000 evaluations.

4.2.2 CIFAR-10

In figure 4.4 we present the results for LeNet CIFAR-10. In these results we see a novel pathology of deadweight; the inner loop being unable to converge sufficiently to trigger an outer step. There were no outer steps after gradient step 50, appearing as a straight line from this point until the end of training when evaluation is triggered irrespective of inner loop convergence. A solution to this issue was to introduce a maximum number of inner loop steps, this is denoted as deadweight max-inner (DW MI) and leads to good convergence on all metrics. We however note that the convergence rate is approximately half of SGD. One interesting observation from figure 4.4, is the deadweight and deadweight max-inner both achieve comparable performance after 10,000 iterations, despite 'plain' deadweight not taking any outer steps after the 50th inner step.

The final experiment applies the Simple-CNN (S-CNN) to CIFAR-10. In figure 4.5 we once again observe deadweight being unable to trigger an outer step, and therefore additionally evaluate deadweight max-inner. We then observe a comparable relationship between SGD and deadweight max-inner as for LeNet CIFAR-10. However, both 'plain' deadweight and Hessian-Free converge very poorly in this experiment, unlike on LeNet where deadweight and deadweight max-inner eventually achieved comparable performance. We see significant variance in the Hessian-Free plots.

4.2.3 Ablation and Time Complexity

In tables 4.7 and 4.8 we present an ablation study of the algorithm optimisations for deadweight. We present the ablation study results for LeNet on both MNIST and CIFAR-10; these results are from the second deadweight grid search. In table 4.9 we additionally present results for the wall clock time per inner step on Simple-CNN CIFAR-10.

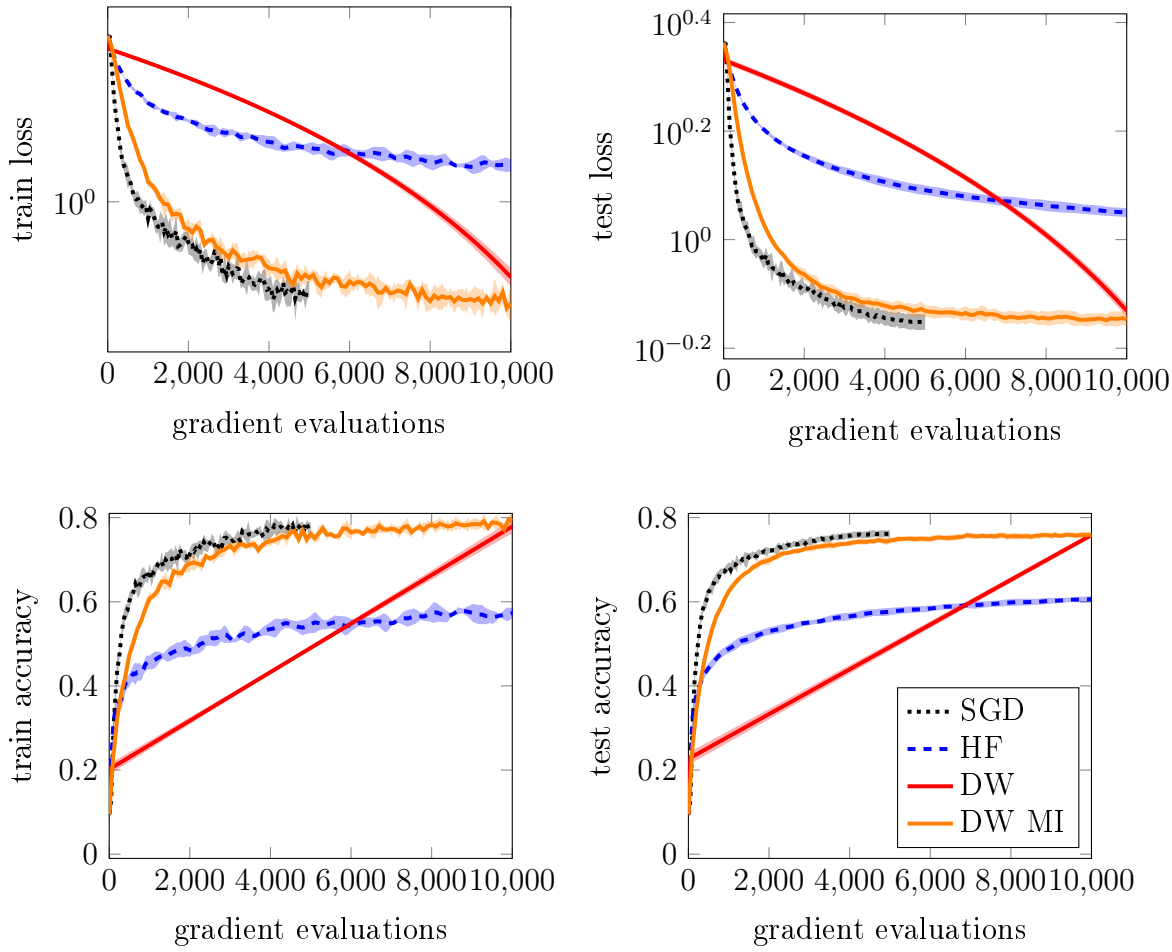

Figure 4.4: **Mini-batch ($B = 1024$) CIFAR-10 training with LeNet. Introducing a maximum number of inner loop steps improves the convergence of deadweight**. Without a maximum number of inner loop steps, deadweight initially converges well, but after (mean) outer step 37 the convergence criterion is never met again, hence there are no evaluation points until the end of training. Introducing a maximum of 100 inner loop steps prevents the algorithm from becoming stuck within the inner loop, denoted as deadweight max-inner (DW MI). For all measured metrics deadweight max-inner achieves comparable performance to SGD but requires approximately twice the gradient steps to achieve this. In contrast, Hessian-Free converges slowly and does not achieve comparable performance. The SGD plots terminate at 5,000 iterations due to this being the number of annealing iterations identified by hyperparameter optimisation.

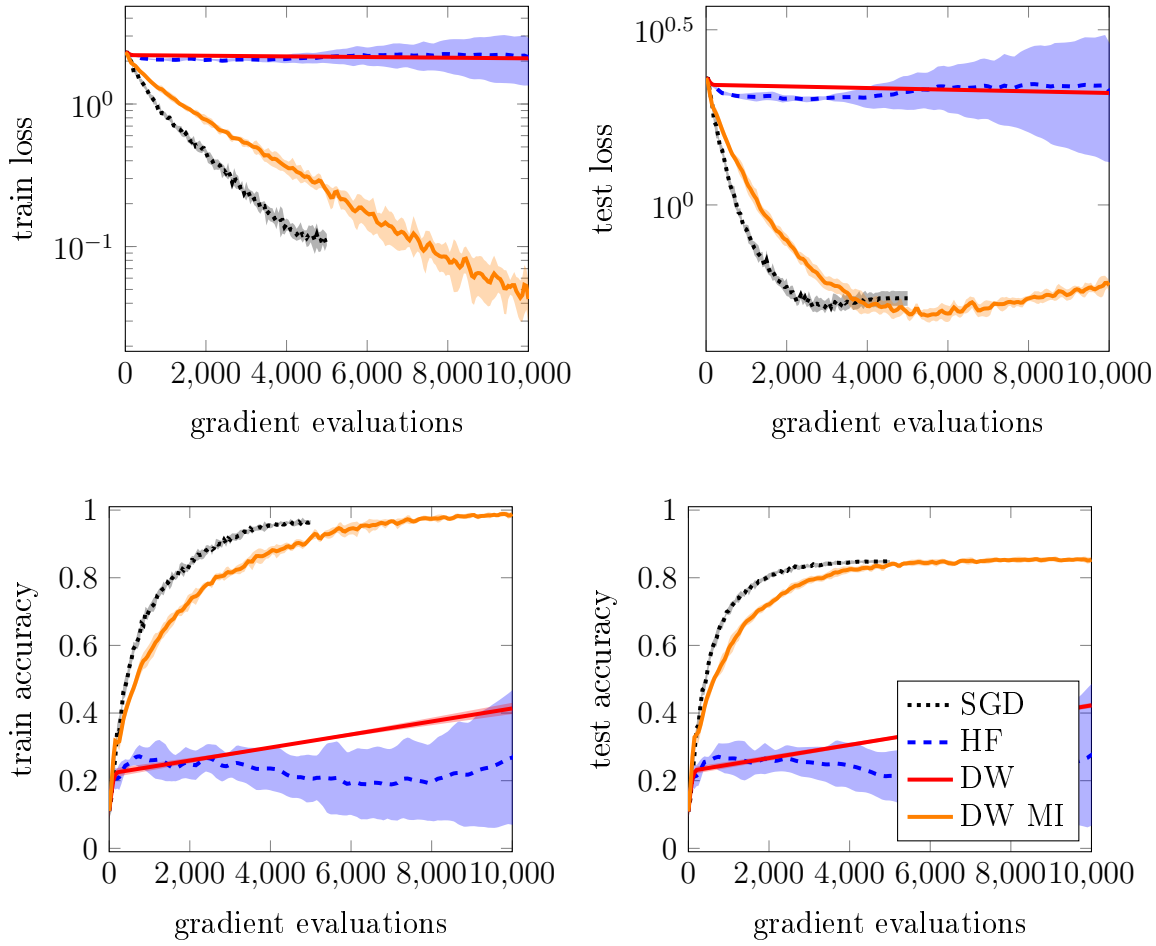

Figure 4.5: **Mini-batch ($B = 1024$) CIFAR-10 training with Simple CNN. Deadweight max-inner greatly exceeds the performance of 'plain' deadweight.** For training metrics, deadweight max-inner slightly exceeds the performance of SGD but requires a longer training period to achieve this. For test metrics the performance is comparable, with convergence occurring slightly earlier than training metrics. Hessian-Free and 'plain' deadweight converge very poorly. The SGD plots terminate at 5,000 iterations due to this being the number of annealing iterations identified by hyperparameter optimisation.

Momentum	Reinitialised			Persistent		
Free-Step	None	Basic	Search	None	Basic	Search
It 98% TA	509 ± 99	452 ± 57	480 ± 94	289 ± 65	287 ± 57	293 ± 57
It 98% EA	558 ± 127	487 ± 60	506 ± 86	355 ± 48	340 ± 38	342 ± 35
Peak TA	99.9 ± 0.05	100.0 ± 0.04	99.9 ± 0.06	100.0 ± 0.00	100.0 ± 0.04	100.0 ± 0.00
Peak EA	99.1 ± 0.07	98.9 ± 0.13	99.0 ± 0.16	99.0 ± 0.08	99.0 ± 0.08	99.0 ± 0.14

Table 4.7: **Ablation results on LeNet MNIST.** Peak accuracy values are consistent across different optimisations, however, iterations until convergence vary widely. Introducing persistent momentum decreases the iterations for convergence significantly for all free-step options. Where persistent momentum is employed, the free-step options have marginal influence. However, where momentum is reinitialised, taking no free step is slightly inferior to the other options. Using line search provides no noticeable benefit for either momentum type. It = (inner) iterations (for), TA = test accuracy, EA = evaluation / test accuracy.

Momentum	Reinitialised			Persistent		
Free-Step	None	Basic	Search	None	Basic	Search
It 75% TA	3420 ± 504	3360 ± 459	3460 ± 528	3800 ± 494	3640 ± 500	3760 ± 393
It 75% EA	6060 ± 1353	6080 ± 1320	6100 ± 1723	5820 ± 1486	5840 ± 1891	5800 ± 1493
Peak TA	80.3 ± 0.85	80.4 ± 0.55	80.6 ± 0.85	80.9 ± 1.16	80.7 ± 0.68	80.4 ± 0.46
Peak EA	76.2 ± 0.43	76.2 ± 0.49	76.2 ± 0.53	76.3 ± 0.60	76.3 ± 0.65	76.1 ± 0.62

Table 4.8: **Ablation results on LeNet CIFAR-10 with max inner** $= 100$**.** Once again peak accuracy values are consistent across different optimisations, however in this case the variation in iterations for convergence is less significant. Persistent momentum causes a slight increase in iterations for convergence on training accuracy, but a slight decrease in iterations for test accuracy. However, the high standard deviation for the iterations until test accuracy must be noted. The influence of the free-step options is negligible for both momentum types. It = (inner) iterations (for), TA = train accuracy, EA = evaluation / test accuracy.

	SGD	Hessian-Free	Deadweight
Time / Inner Step (ms)	51.1 ± 0.3	113.1 ± 0.6	58.0 ± 0.0
Number of Outer Steps	N/A	43 ± 0	111 ± 2
Time / Outer Step (s)	N/A	26.70 ± 0.14	5.25 ± 0.03

Table 4.9: **Time complexity results for Simple CNN on CIFAR-10.** Deadweight introduces only a 13.5% increase in wall time per inner step, relative to SGD. In contrast, each Hessian-Free inner step is 108% more time complex than SGD.

4.2.4 Summary

We summarise the experimental results as follows:

1. Deadweight is a viable algorithm for training fully-connected and convolutional neural networks on MNIST.
2. Introducing a maximum number of inner loop steps enables comparable performance to SGD on CIFAR-10, at a lower rate of convergence than SGD.
3. The mini-batch MNIST deadweight experiments experienced a plateau/deterioration in performance after initially strong convergence.
4. In no case did a deadweight variant achieve higher testing accuracy in fewer iterations than SGD.
5. Hessian-Free performed poorly on all experiments except MLP MNIST (full-batch), where it remained the slowest to converge.

4.3 Discussion

We arrange our discussion around key observations from the results.

No advantage to second-order optimisation: In no cases did Hessian-Free or deadweight outperform SGD in convergence rate. In some cases, deadweight achieves marginally higher peak performance, but this was not the grid-search objective and hence a different SGD configuration may achieve even higher peak metrics. For deadweight it is apparent that the algorithm in the evaluated form is inferior to SGD. The poor convergence of Hessian-Free is notable; only on full-batch MNIST did Hessian-Free achieve comparable peak performance to SGD. We cannot eliminate the possibility of implementation errors in the public implementation employed. One possible explanation for the rapid convergence of SGD is the use of momentum, and modern initialisation schemes, which are thought to reduce the benefit gained from second-order optimisation [36].

Large increase in test loss on MLP MNIST full-batch: The relatively low variance suggests this phenomenon to be repeated across seeds, a further investigation in which the individual seeds were plotted indicated this to be the case. The large increase in test loss without a corresponding decrease in test accuracy suggests that where the model is making inaccurate predictions, these are being made with high confidence. However, this phenomenon was not observed in later experiments, leading us to hypothesise it to be a result of using full-batch gradients.

Plateau/deterioration for mini-batch MNIST: The low variance of the plateau suggests that all seeded runs experienced a similar phenomenon. Once again this was confirmed by plotting the seeds separately. It is possible other hyperparameter configurations did not suffer from this effect, and by defining the convergence criterion as 98%

validation accuracy we have selected a sub-optimal run. Since the training loss and accuracy are decreasing along with the test metrics, this would not be considered overfitting but some kind of instability. Ideally, once the active model has reached a solution to the objective function, the outer steps will simply lead the deadweight model to catch up. However, these results indicate this is not happening, perhaps due to the distance between the models being too large. Further work could address this behaviour using an increasing schedule of lambdas, to progressively decrease the size of the trust region in which we optimise, reducing the distance between the models.

Optimal deadweight hyperparameters: We see in table 4.6 that the MLP MNIST and LeNet MNIST have higher $\lambda_{\mathcal{Y}}$ and ϵ than the MLP MNIST (full-batch) experiment. Intuitively this makes sense, since the variance of the stochastic gradients leads to higher gradient norm values, making it harder to reach a given convergence criterion ϵ. Having a larger $\lambda_{\mathcal{Y}}$ makes our trust region smaller, placing more emphasis on the proximity term and likely leading the inner loop solution to be closer to the reference model. Having a larger ϵ simply makes the convergence criterion less strict, triggering an outer step at a larger gradient norm value. The combination of these factors leads to the inner loop criterion being met more easily, taking more outer steps and being less likely to become stuck in an inner loop. This pattern of higher $\lambda_{\mathcal{Y}}$ and ϵ is broken by the CIFAR-10 experiments, however, this can be understood from the use of a maximum number of inner steps, because of which it is not possible for the algorithm to become stuck within an inner loop, therefore placing less emphasis on making the inner loop strictly 'solvable'.

Ablation / algorithm optimisations: In tables 4.7 and 4.8 we see the effect of introducing the different algorithm optimisations, discussed in section 3.3.2. On both MNIST and CIFAR-10 these optimisations have negligible influence on the peak train and test accuracies achieved. In contrast, the inner loop iterations until convergence do vary with these optimisations. On MNIST a strong case can be made that persistent momentum is beneficial for improving rate of convergence. However, on CIFAR-10 the relative reduction in mean iterations is much smaller, and with a greater standard deviation. This is likely a result of the CIFAR-10 results using 100 max inner iterations, making the length of the inner loops less dependent on these optimisations hence leading to a less sensitive convergence rate. The only context in which the different free step types have a notable influence on the convergence rates is MNIST with reinitialised momentum, in all other settings these have negligible impact. We do however note that `freestep` = none was selected as superior in all but the Simple CNN experiment, as seen in table 4.6.

Time complexity: In table 4.9, we see that the time complexity per inner step is comparable between SGD and deadweight but much higher for Hessian-Free. The similar time complexities of SGD and deadweight are expected; the inner loop of deadweight performs SGD on a modified objective. The minor increase can be attributed to handling the outer loop operations, such as stepping the deadweight model. The large increase in

time complexity per inner step of Hessian-Free is unexpected, with Grosse [19] stating that a conjugate gradient step has approximately the complexity of a backpropagation step. Possible explanations for this increase include; the use of line searching, the adaptive damping heuristic or the use of conjugate gradient backtracking [6]. Another likely factor is that the standard implementation of backpropagation in PyTorch is highly optimised, whereas the conjugate gradient implementation of [52] may be less performant. With respect to the convergence results, these time complexity results imply that deadweight is slightly slower to converge than previously understood, and Hessian-Free is significantly slower.

Chapter 5

Related work

In this chapter, we discuss existing neural network optimisation algorithms. Adaptive and approximate second-order methods are generally proposed to accelerate convergence, with comparable generalisation error to stochastic gradient descent (SGD) deemed acceptable. In contrast, sharpness-aware methods are designed specifically to improve generalisation. We further briefly introduce two policy optimisation methods from reinforcement learning, of which proximal policy optimisation bears similarity to the proposed deadweight.

5.1 Adaptive Optimisation

Adaptive optimisers scale the gradient using a running average of the element-wise gradient magnitudes; RMSProp [53] and Adam [8] are two prominent examples of such methods. The algorithms are similar, with Adam essentially adding momentum to RMSProp. Both maintain an estimate of the second moments of the parameter-wise derivatives [53]. This estimate is an exponential moving average of the squared magnitude of the gradient elements. For RMSProp, the update vector is obtained by scaling the gradient vector by the inverse square root of the second moments. Adam differs by additionally maintaining an estimate of the first moment of the derivative magnitudes [8]. The update vector in this case is given by dividing the first-order estimate element-wise by the square root second-order estimate.

The element-wise scaling employed by RMSProp and Adam results in the updates to all parameters being approximately equal to a constant [24]. Updates to parameters with small gradient magnitudes will be amplified, enabling these parameters to converge more rapidly. In contrast, parameters with large gradient magnitudes will take smaller steps, potentially reducing instability. It additionally implies the updates to be invariant to elementwise scaling of the gradients [8]. An alternative perspective is that the second-moment estimates used in RMSProp and Adam are diagonal estimations of the *empirical* Fisher information [24]. However, the empirical Fisher information differs in optimisation

properties to the true Fisher information [25], hence this is not a diagonal approximation to natural gradients.

Notably, the bias of Adam towards generalisation has been questioned [12], it is proposed that Adam achieves more rapid convergence at the cost of inferior generalisation performance. A more general study of the effect of preconditioning on generalisation was conducted by Amari et al. [15]. Nonetheless, it remains the second most popular choice of optimiser, after SGD. It is particularly popular for specific applications, such as the training of large language models [40]. Connecting Adam to our desiderata, defined in section 2.4; Adam can be considered to meet all desiderata except bias towards generalisation, which is both application-specific and disputed. A recently proposed algorithm named Sophia [40] has been proposed to achieve a two-fold speed-up over Adam.

5.2 Approximate Second-Order Optimisation

As discussed in section 2.5, second-order methods such as Newton-Raphson and natural gradients are intractable within deep learning. There are therefore a number of proposed approximations. Whilst adaptive optimisation can be considered to approximate second-order methods, we make a distinction here with non-diagonal approximations. We will focus on two approximations; Hessian-Free [6] and Kronecker-Factored Approximate Curvature (K-FAC) [7].

Truncated-Newton methods, also known as Hessian-Free optimisation, are a family of approximations to Newton-Raphson [54]. Here the conjugate gradient method is employed to approximate the solution to the (local) quadratic objective without computing or inverting the Hessian [6]. Exactly solving linear systems using conjugate gradients is comparable in complexity to other methods [19]. However conjugate gradients can be terminated early, resulting in an efficient but approximate solution [6]. Hessian-Free was first applied to deep learning by Martens [6]. The conjugate gradient method requires a positive semi-definite Hessian matrix, an assumption that does not hold for neural networks [55]. Martens [6] instead use the Gauss-Newton Hessian, which is guaranteed to be positive semi-definite, and discussed further in section 2.1. An adaptive heuristic is used to tune the damping, wherein damping is increased or decreased according to an estimate of the Taylor expansion accuracy. Whilst Martens [6] used Hessian-Free to approximate the generalised Gauss-Newton method, natural gradients can also be approximated with this method [15].

Hessian-Free never gained popularity within deep learning. Sutskever et al. [36] note that momentum combined with improved initialisation schemes enable SGD to handle many of the issues, such as curvature, that Hessian-Free was proposed to overcome. Furthermore Grosse [19] comment that when Hessian-Free was first applied to deep learning in 2010, architectures presented highly poorly conditioned optimisation objectives. Im-

proved modern architectures, initialisation schemes, and momentum therefore limit the utility of Hessian-Free. Whilst a single conjugate gradient step has comparable complexity to a gradient descent step, the conjugate gradient method requires the use of the same batch repeatedly until the inner loop has terminated [6]. This implies Hessian-Free to have lower data throughput than SGD or Adam, a potentially large disadvantage [19].

Kronecker-Factored Approximate Curvature fits a parametric approximation to the Fisher information [56]. Central to the method is the Kronecker product, enabling matrix-matrix products to be decomposed into matrix-vector products [19]. The Fisher information is approximated using a block diagonal matrix, with each block being a Kronecker factorised matrix representing a layer of the neural network. Inverting a block diagonal matrix is equivalent to inverting the individual blocks, thereby reducing computational complexity significantly. Strong performance was demonstrated on ImageNet by Ba et al. [57], where K-FAC was demonstrated to accelerate training two-fold versus the inclusion of batch normalisation in the network. Notably, K-FAC is not architecture agnostic, it requires derivations for different types of neural network layers such as convolutional [7] and recurrent [58].

5.3 Sharpness-Aware Optimisation

Whilst the previous algorithms were proposed to accelerate convergence, those in this section are designed to instead improve generalisation. As discussed in section 2.3, there has been significant work studying the geometry of SGD solutions in relation to generalisation [30–33, 59]. Connections between notions of 'flatness' and strong generalisation performance date back to [30], where it was proposed that flat minima had lower minimum descriptive lengths, and thus implemented simpler functions.

Motivated by such connections, sharpness-aware minimisation aims to optimise for flat minima, achieving state-of-the-art generalisation results on a variety of benchmark tasks [43]. The intuition is simple; we are seeking a region of uniformly low loss, hence we are concerned with the maximum loss value in the neighbourhood of our parameter values. This is formalised into a minimax problem in which we minimise the maximum loss within a radius of our parameters. Approximating this with a first-order Taylor expansion leads us to the objective in equation 5.1. Here κ is a constant scaling the norm of the gradient term, defining the size of the neighbourhood. Using two gradient evaluations we are able to optimise this objective function, increasing complexity but enhancing the bias towards generalisation.

$$R^{\text{SAM}}(\boldsymbol{w}) = R(\boldsymbol{w} + \kappa \nabla_{\boldsymbol{w}} R(\boldsymbol{w})) \tag{5.1}$$

Following the success of SAM, a number of follow-up works have been proposed. Adaptive

SAM introduced Adam-like adaptive scaling to the neighbourhood definition [60]. However, Dinh et al. [61] showed that reparameterising the weights of the network enables minima to be made arbitrarily sharp or flat without changing the function implemented. Even without reparameterisation, the connection between notions of flatness and generalisation was shown to be tenuous. This led to the proposal of Fisher-SAM, which instead defines an ellipsoidal neighbourhood within the distribution space using an approximation of the Fisher information [62].

5.4 Policy Optimisation

In reinforcement learning, a policy $\pi : \mathcal{S} \times \mathcal{W} \rightarrow \mathcal{A}$ is a function, parameterised by $\boldsymbol{w} \in \mathcal{W}$, that takes a state $s \in \mathcal{S}$ as input, and outputs an action $a \in \mathcal{A}$ [63]. A sequence of state-action pairs is referred to as a *trajectory* $\tau = (s_1, a_1, ...s_T, a_T) \in \mathcal{T}$, on which a *reward* is defined $R : \mathcal{T} \rightarrow \mathbb{R}$. We seek to optimise our policy to maximising the expected reward of our policy, as stated in equation 5.2. For all but the simplest problems, the expectation in equation 5.2 is intractable; we instead sample trajectories from our policy and optimise this stochastic approximation.

$$J(\boldsymbol{w}) = \mathbb{E}_{\tau \sim \pi_{\boldsymbol{w}}} R(\tau) \tag{5.2}$$

Within reinforcement learning producing new trajectory samples can be highly costly. For this reason, modern policy gradient methods use surrogate objectives in place of equation 5.2 [64]. These surrogate objectives enable the reuse of trajectories for successive optimisation steps [65], increasing the sample efficiency. However, these surrogate objectives become increasingly inaccurate further from the parameters $\tilde{\boldsymbol{w}}$ used to generate the trajectories, leading to high gradient variance. One solution is to solve a constrained optimisation problem, before regenerating the trajectories with a new $\tilde{\boldsymbol{w}}$. As discussed in section 2.5, natural gradients can be considered to locally solve a constrained optimisation problem, with a trust region defined using the Fisher information. Employing natural gradients within reinforcement learning is known as natural policy gradients [66]. As in supervised learning, natural policy gradients are computationally intractable for non-trivial problems.

Trust-Region Policy Optimisation (TRPO) [67] and Proximal Policy Optimisation (PPO) [16] are two approximations to natural policy gradients. Similar to Hessian-Free, TRPO employs the conjugate gradient method to approximately solve the constrained surrogate objective. After the step direction has been established, a line search is performed to ensure a KL-divergence constraint is met between the policy iterates [65]. Instead of using the conjugate gradient to approximately solve a linear system, PPO uses SGD or Adam to directly optimise the proximal objective. PPO uses a heuristic to manage the size of the trust region, achieving a similar effect to the line search within TRPO.

TRPO and PPO are commonly used within 'deep' reinforcement learning, wherein the policy π is a neural network. So why are these algorithms not used for supervised learning? Abstracting away the surrogate objective from TRPO effectively recovers Hessian-Free. On the other hand, surrogate objective aside, we are not aware of a PPO-like algorithm being applied to supervised learning. Without further modification, this would imply taking several gradient steps on the same batch of data. As is the case with Hessian-Free (and subsequently TRPO), this limits the data throughput, undesirable for supervised learning [19]. In contrast, in reinforcement learning the high cost of obtaining samples outweighs the lower data throughput in favour of sample efficiency.

Chapter 6

Conclusions and Future Work

6.1 Conclusion

This work aimed to propose a novel optimisation algorithm for neural networks. More specifically, we sought to propose a first-order method that shared some inductive bias with natural gradient descent. Existing algorithms generally suffer from being intractable [13], factorise second-order information [56], or require the efficient implementation of specific operations such as conjugate gradient [6]. The proposed algorithm is theoretically motivated and addresses each of these limitations. As the algorithm developed, similarities became apparent with proximal policy optimisation, a method from reinforcement learning. These similarities are discussed further in section 3.4. The qualitative results in Chapter 3, demonstrate deadweight as able to closely approximate natural gradient descent on a simple test function.

Deadweight was comprehensively evaluated using three architectures and two image classification datasets. The results demonstrated that deadweight was a viable method for optimising neural networks, but suffered from a convergence rate slower than stochastic gradient descent (SGD), as well as various pathologies during convergence. For the CIFAR-10 experiments it was necessary to introduce a maximum number of inner loop steps to enable consistent convergence. In section 2.4, we introduce four desiderata for neural network optimisation algorithms; computational tractability, rapid convergence, bias towards generalisation and being architecture agnostic. With respect to the results of this work, we believe all desiderata are achieved by deadweight aside from rapid convergence, which we may be able to improve further in future work.

6.2 Limitations

Whilst we believe deadweight optimisation is theoretically well-motivated, and the evaluation procedure to be sufficiently comprehensive to draw meaningful conclusions, there

remain limitations of this work.

Evaluation procedure: Notably, evaluation was only performed on image classification, the results would be strengthened by employing alternative data modalities. It would additionally be preferable to present results for SGD without momentum, with the work of Sutskever et al. [36] demonstrating momentum to overcome many limitations of first-order optimisation. It is plausible that deadweight may converge more rapidly than momentum-free SGD, and this would provide an interesting point of discussion.

Hyperparameter Optimisation Another area of potential limitation was the definition of the deadweight grid searches. Ideally, the entire search space would have been grid searched. However, we used a 'split' grid search. It is quite possible that only evaluating the best configuration from the first grid search in the second introduced some bias towards certain optimisations. Interestingly, the optimisation used for the first grid search was not found to be the highest performing in the second. Optimising both deadweight and the baselines with Bayesian optimisation may increase their respective performances, and provide further insight.

Qualitative Results: A final notable limitation of this work is the lack of qualitative results for deadweight on deep learning tasks. Whilst the qualitative properties were discussed on the test function, this did not translate easily to high-dimensional neural network parameter spaces. Since it is not possible to plot the neural network parameter spaces meaningfully, it would be necessary to study other properties, such as the convergence of the logits themselves [14]. Another approach would be to implement natural gradient descent using Hessian-Free and study isolated inner optimisation loops, permitting comparison of the step directions and magnitudes.

6.3 Future Work

Evaluating deadweight on a wider variety of tasks and datasets is an obvious direction for further work. However, we do not feel this to be the most pressing direction in which to continue this project. We would be eager to compare deadweight to a momentum-free SGD baseline, a baseline that deadweight may already surpass in convergence rate. Nonetheless, we believe the performance of deadweight could be improved, and this to be the highest priority for future work. There are three directions we feel are worth exploring further:

Alternative means of defining inner loop convergence. The simplest method for handling inner loop convergence is to simply set a fixed number of inner iterations, as done by PPO [16]. However as discussed in section 3.4, this may lead to sub-optimal usage of an inner step budget. For deadweight we defined the inner loop convergence using the gradient norm $||\boldsymbol{g_w}||$. We could alternatively use the value of the proximal objective R_{prox} itself, defining an outer step to take place after P iterations without improvement.

Smaller reference model learning rates α on deep learning tasks. Whilst, this was investigated on the Rosenbrock function in section 3.3.1, this was not continued onto the deep learning tasks. Intuitively, taking smaller steps on the deadweight model implies that the corresponding minimum is closer to the current active model position. In other words, the active model is in a better initialisation for solving the updated proximal objective. Whilst this may lead to a significant increase in outer steps required, it should also increase the stability of the algorithm, and prevent some of the pathologies observed in section 4.2.

Alternative inner loop optimiser: Using Adam to optimise the inner loop is known to improve the convergence of PPO [16]. It is therefore natural to investigate the effect of this on deadweight.

One final direction of future work, unrelated to improving the convergence rate of deadweight, is a qualitative study of the algorithm's inductive bias on deep learning tasks. This may prove to be the most important application of deadweight, as an efficient method for proximal optimisation and the approximation of natural gradients for empirical study. The work of Kerekes et al. [14] includes a study of the logit convergence rate of both gradient descent and natural gradient descent, the behaviour of deadweight on this task could validate the similarities to natural gradients on a deep learning task. There are furthermore several works analysing the geometry of SGD solutions [31–33], further work could explore the bias of deadweight towards such solutions. Additional interesting phenomenon that could be explored include double descent [44], grokking [68, 69] and bad minima [11]; the interplay of second-order method with which remains an interesting and mostly unexplored area.

Bibliography

[1] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[2] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A. A. Kohl, Andrew J. Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W. Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. Highly accurate protein structure prediction with AlphaFold. *Nature*, 596(7873):583–589, August 2021. ISSN 1476-4687. doi: 10.1038/s41586-021-03819-2.

[3] Alexander Kirillov, Eric Mintun, Nikhila Ravi, Hanzi Mao, Chloe Rolland, Laura Gustafson, Tete Xiao, Spencer Whitehead, Alexander C. Berg, Wan-Yen Lo, Piotr Dollár, and Ross Girshick. Segment Anything, April 2023.

[4] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal,

Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. PaLM: Scaling Language Modeling with Pathways, October 2022.

[5] Roger Grosse. Chapter 3: Metrics. In *Neural Network Training Dymanics*. 2021.

[6] James Martens. Deep learning via Hessian-free optimization. In *International Conference on Machine Learning*, June 2010.

[7] Roger Grosse and James Martens. A Kronecker-factored approximate Fisher matrix for convolution layers. In *Proceedings of The 33rd International Conference on Machine Learning*, pages 573–582. PMLR, June 2016.

[8] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization, January 2017.

[9] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[10] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization, February 2017.

[11] Shengchao Liu, Dimitris Papailiopoulos, and Dimitris Achlioptas. Bad Global Minima Exist and SGD Can Reach Them, February 2021.

[12] Nitish Shirish Keskar and Richard Socher. Improving Generalization Performance by Switching from Adam to SGD, December 2017.

[13] Shun-ichi Amari. Natural Gradient Works Efficiently in Learning. *Neural Computation*, 10(2):251–276, February 1998. ISSN 0899-7667. doi: 10.1162/089976698300017746.

[14] Anna Kerekes, Anna Mészáros, and Ferenc Huszár. Depth Without the Magic: Inductive Bias of Natural Gradient Descent, November 2021.

[15] Shun-ichi Amari, Jimmy Ba, Roger Grosse, Xuechen Li, Atsushi Nitanda, Taiji Suzuki, Denny Wu, and Ji Xu. When Does Preconditioning Help or Hurt Generalization?, December 2020.

[16] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms, August 2017.

[17] Yann LeCun. The MNIST database of handwritten digits. 1998.

[18] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[19] Roger Grosse. Chapter 4 Second-Order Optimization. In *Neural Network Training Dymanics*. 2021.

[20] Katta G. Murty and Santosh N. Kabadi. Some NP-complete problems in quadratic and nonlinear programming. *Mathematical Programming*, 39(2):117–129, June 1987. ISSN 1436-4646. doi: 10.1007/BF02592948.

[21] Roger Grosse. Chapter 2: Taylor Approximations. In *Neural Network Training Dymanics*. 2021.

[22] Nicol N. Schraudolph. Fast Curvature Matrix-Vector Products for Second-Order Gradient Descent. *Neural Computation*, 14(7):1723–1738, July 2002. ISSN 0899-7667. doi: 10.1162/08997660260028683.

[23] Shun-ichi Amari. *Information Geometry and Its Applications*. Springer, February 2016. ISBN 978-4-431-55978-8.

[24] Roger Grosse. Chapter 5: Normalisation. In *Neural Network Training Dymanics*. 2021.

[25] Frederik Kunstner, Philipp Hennig, and Lukas Balles. Limitations of the empirical Fisher approximation for natural gradient descent. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[26] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[27] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, October 1986. ISSN 1476-4687. doi: 10.1038/323533a0.

[28] Devansh Arpit, Stanisław Jastrzębski, Nicolas Ballas, David Krueger, Emmanuel Bengio, Maxinder S. Kanwal, Tegan Maharaj, Asja Fischer, Aaron Courville, Yoshua Bengio, and Simon Lacoste-Julien. A Closer Look at Memorization in Deep Networks, July 2017.

[29] Samuel L. Smith, Benoit Dherin, David G. T. Barrett, and Soham De. On the Origin of Implicit Regularization in Stochastic Gradient Descent, January 2021.

[30] Sepp Hochreiter and Jürgen Schmidhuber. Flat Minima. *Neural Computation*, 9(1): 1–42, January 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.1.1.

[31] Timur Garipov, Pavel Izmailov, Dmitrii Podoprikhin, Dmitry Vetrov, and Andrew Gordon Wilson. Loss Surfaces, Mode Connectivity, and Fast Ensembling of DNNs, October 2018.

[32] Haowei He, Gao Huang, and Yang Yuan. Asymmetric Valleys: Beyond Sharp and

Flat Local Minima. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[33] Samuel K. Ainsworth, Jonathan Hayase, and Siddhartha Srinivasa. Git Re-Basin: Merging Models modulo Permutation Symmetries, December 2022.

[34] Guillermo Valle-Pérez, Chico Q. Camargo, and Ard A. Louis. Deep learning generalizes because the parameter-function map is biased towards simple functions, April 2019.

[35] Jonas Geiping, Micah Goldblum, Phillip E. Pope, Michael Moeller, and Tom Goldstein. Stochastic Training is Not Necessary for Generalization, April 2022.

[36] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on Machine Learning*, pages 1139–1147. PMLR, May 2013.

[37] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *Proceedings of the 32nd International Conference on Machine Learning*, pages 448–456. PMLR, June 2015.

[38] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer Normalization, July 2016.

[39] Yuxin Wu and Kaiming He. Group Normalization. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 3–19, 2018.

[40] Hong Liu, Zhiyuan Li, David Hall, Percy Liang, and Tengyu Ma. Sophia: A Scalable Stochastic Second-order Optimizer for Language Model Pre-training, May 2023.

[41] H. H. Rosenbrock. An Automatic Method for Finding the Greatest or Least Value of a Function. *The Computer Journal*, 3(3):175–184, January 1960. ISSN 0010-4620. doi: 10.1093/comjnl/3.3.175.

[42] Jack Kiefer. Sequential minimax search for a maximum. *Proceedings of the American mathematical society*, 4(3):502–506, 1953.

[43] Pierre Foret, Ariel Kleiner, Hossein Mobahi, and Behnam Neyshabur. Sharpness-Aware Minimization for Efficiently Improving Generalization, April 2021.

[44] Preetum Nakkiran, Gal Kaplun, Yamini Bansal, Tristan Yang, Boaz Barak, and Ilya Sutskever. Deep Double Descent: Where Bigger Models and More Data Hurt, December 2019.

[45] Jonathan Frankle and Michael Carbin. The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks, March 2019.

[46] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban

Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[47] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, May 2017. ISSN 0001-0782. doi: 10.1145/3065386.

[48] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998. ISSN 1558-2256. doi: 10.1109/5.726791.

[49] Michael M. Bronstein, Joan Bruna, Taco Cohen, and Petar Veličković. Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges, May 2021.

[50] Matthias Feurer and Frank Hutter. Hyperparameter optimization. *Automated machine learning: Methods, systems, challenges*, pages 3–33, 2019.

[51] Leslie N. Smith and Nicholay Topin. Super-convergence: Very fast training of neural networks using large learning rates. In *Artificial Intelligence and Machine Learning for Multi-Domain Operations Applications*, volume 11006, pages 369–386. SPIE, May 2019. doi: 10.1117/12.2520589.

[52] ltatzel. PyTorchHessianFree, 2022.

[53] Geoffrey Hinton. Neural Networks for Machine Learning: Lecture 6a Overview of mini-batch gradient descent, 2014.

[54] Ron S. Dembo and Trond Steihaug. Truncated-Newton algorithms for large-scale unconstrained optimization. *Mathematical Programming*, 26(2):190–212, June 1983. ISSN 1436-4646. doi: 10.1007/BF02592055.

[55] Levent Sagun, Leon Bottou, and Yann LeCun. Eigenvalues of the Hessian in Deep Learning: Singularity and Beyond, November 2016.

[56] James Martens and Roger Grosse. Optimizing Neural Networks with Kronecker-factored Approximate Curvature, June 2020.

[57] Jimmy Ba, Roger Grosse, and James Martens. Distributed Second-Order Optimization using Kronecker-Factored Approximations. In *International Conference on Learning Representations*, July 2022.

[58] James Martens, Jimmy Ba, and Matt Johnson. Kronecker-factored Curvature Approximations for Recurrent Neural Networks. In *International Conference on Learning Representations*, February 2022.

[59] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima, February 2017.

[60] Jungmin Kwon, Jeongseop Kim, Hyunseo Park, and In Kwon Choi. ASAM: Adaptive Sharpness-Aware Minimization for Scale-Invariant Learning of Deep Neural Networks. In *Proceedings of the 38th International Conference on Machine Learning*, pages 5905–5914. PMLR, July 2021.

[61] Laurent Dinh, Razvan Pascanu, Samy Bengio, and Yoshua Bengio. Sharp Minima Can Generalize For Deep Nets, May 2017.

[62] Minyoung Kim, Da Li, Shell X. Hu, and Timothy Hospedales. Fisher SAM: Information Geometry and Sharpness Aware Minimisation. In *Proceedings of the 39th International Conference on Machine Learning*, pages 11148–11161. PMLR, June 2022.

[63] Wouter van Heeswijk. Policy Gradients In Reinforcement Learning Explained. https://towardsdatascience.com/policy-gradients-in-reinforcement-learning-explained-ecec7df94245, April 2023.

[64] Mingfei Sun, Benjamin Ellis, Anuj Mahajan, Sam Devlin, Katja Hofmann, and Shimon Whiteson. Trust-Region-Free Policy Optimization for Stochastic Policies, February 2023.

[65] Ted Moskovitz, Michael Arbel, Ferenc Huszar, and Arthur Gretton. Efficient Wasserstein Natural Gradients for Reinforcement Learning, March 2021.

[66] Sham Kakade. A natural policy gradient. In *Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic*, NIPS'01, pages 1531–1538, Cambridge, MA, USA, January 2001. MIT Press.

[67] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust Region Policy Optimization, April 2017.

[68] Alethea Power, Yuri Burda, Harri Edwards, Igor Babuschkin, and Vedant Misra. Grokking: Generalization Beyond Overfitting on Small Algorithmic Datasets, January 2022.

[69] Ziming Liu, Eric J. Michaud, and Max Tegmark. Omnigrok: Grokking Beyond Algorithmic Data, October 2022.