AMERICAN UNIVERSITY OF BEIRUT

SECOND ORDER TRUST REGION OPTIMIZATION METHODS FOR TRAINING NEURAL NETWORKS: BEYOND INEXACT NEWTON

by
KYLE LOMER

A thesis
submitted in partial fulfillment of the requirements
for the degree of Master of Science
to the Department of Computer Science
of the Faculty of Arts and Sciences
at the American University of Beirut

Beirut, Lebanon
September 2020

AMERICAN UNIVERSITY OF BEIRUT

SECOND ORDER TRUST REGION OPTIMIZATION METHODS FOR TRAINING NEURAL NETWORKS: BEYOND INEXACT NEWTON

by

KYLE LOMER

Approved by:

Dr. George Turkiyyah, Professor Computer Science	Advisor George Turkiyyah
Dr. Shady Elbassuoni, Assistant Professor Computer Science	Member of Committee
Dr. Izzat El Hajj, Assistant Professor Computer Science	Member of Committee

Date of thesis defense: September 3, 2020

AMERICAN UNIVERSITY OF BEIRUT

THESIS, DISSERTATION, PROJECT RELEASE FORM

Student Name: Lomer Kyle Julian
Last First Middle

⊗ Master's Thesis ○ Master's Project ○ Doctoral Dissertation

☒ I authorize the American University of Beirut to: (a) reproduce hard or electronic copies of my thesis, dissertation, or project; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes.

☐ I authorize the American University of Beirut, to: (a) reproduce hard or electronic copies of it; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes after: **One ___ year from the date of submission of my thesis, dissertation or project.**
Two ___ years from the date of submission of my thesis , dissertation or project.
Three ___ years from the date of submission of my thesis , dissertation or project.

K. Lomer 09/09/2020
Signature Date

This form is signed when submitting the thesis, dissertation, or project to the University Libraries

Acknowledgements

With the greatest of thanks to my advisor Professor George Turkiyyah for his help and his wisdom. The guidance and encouragement he provided were the stick and carrot that made this thesis possible. Thanks also to the committee members, Professor Shady Elbassuoni and Professor Izzat El Hajj, for everything I learnt in their engaging lectures and for sharing their expertise.

On a personal note, thanks to my partner, Palika. For pretending to be interested in saddle points, putting up with me during lockdown and making sure I went outside after it. Thanks to my family for their love and support in this chapter of my life, even from thousands of miles away.

Lastly a huge thanks to the Lebanese people. Lebanon has experienced so many ups and downs in my two years here but I only experienced warmth, compassion and companionship, both inside of AUB and out. My heart goes out to you all.

An Abstract of the Thesis of

Kyle Lomer for Master of Science
Major: Computer Science

Title: Second Order Trust Region Optimization Methods for Training Neural Networks: Beyond Inexact Newton

Second order optimization methods have always been less widely used for training neural networks than first order methods such as Stochastic Gradient Descent. This is mainly due to the complexity and high costs in terms of both processor and memory resources of second order methods. In recent years more work has been done to adapt these methods to make them more suitable for training neural networks. In this paper we demonstrate how trust region methods can be used to improve the convergence and cost-effectiveness of second order optimization. This is achieved by only using cheap first order information when it is an appropriate approximation for the expensive second order information, based on the relative size of the trust region. We also present techniques to automatically tune the hyperparameters these methods introduce; including a novel approach to adaptive regularization. These methods are demonstrated on autoencoders and image classifiers in comparison to first order methods.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this thesis we discuss the application of second order methods to train feed forward neural networks, in particular convolutional neural networks for autoencoders and image classification. We focus on the Inexact Newton method used in combination with a Trust Region for globalization.

1.1 Thesis Structure

We begin here in Chapter 1 defining the problem and analysing the challenges for an optimization method training neural networks. In Chapter 2, we look at related work, beginning with first order methods then reviewing a range second order methods including Inexact Newton. We compare their strengths and weaknesses. In Chapter 3 we describe the components and techniques that make up our methods before presenting the full algorithms in Chapter 4 and analysing their costs. Chapter 5 shows the results from our experiments. We justify the choices we made in building the methods and demonstrate that hyper parameter tuning is unnecessary. Then we show the performance of the methods compared

to first order methods for a range of models. In Chapter 6 we conclude and discuss extensions of the work in this paper.

1.2 Goals and Contributions

Many papers have demonstrated the potential of Inexact Newton for training neural networks, but there are still no established second order methods used by the machine learning community. We aim to build upon the previous work in the fields of machine learning and optimization to create a sophisticated method with two specific goals:

1. To reduce the per iteration computational cost of the Inexact Newton method. The cheaper and faster the method runs, the better.

2. To reduce the amount of hyper parameter tuning required. Hyper parameters are often tuned through grid or random search and this requires many runs. A method which requires little or no tuning at all runs quicker overall than one which only runs fast with well tuned hyper parameters.

This thesis makes the following contributions towards achieving these goals:

1. A novel approach to adaptive regularization for Trust Region methods.

2. New trust region methods which reduce the computation cost per iteration by using first order information when the trust region is small and a combination of first and second order information when the trust region is large.

3. A detailed survey and analysis of techniques to speed up the Inexact Newton method for neural networks.

4. An experimental comparison of these methods to first order methods on several standard neural network models.

5. A software package to share these optimization techniques with the machine learning community.

1.3 Problem Definition

1.3.1 Neural Networks

In theory [1] the problem of training a neural network is posed as:

$$\min_{w} \mathbb{E}_{(x,y)\backsim\nu}[L(y, \phi(w, x))] \tag{1.1}$$

For trainable weights $w \in \mathbb{R}^p$.

$\phi : \mathbb{R}^p \times \mathbb{R}^n \to \mathbb{R}^d$ the model evaluation.

$L : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ the loss function.

This is the expected loss of the model with respect to x, y, data values distributed to the probability distribution ν. Practically, ν is unknown and instead we have N (possibly noisy) data points from the distribution. Thus, we must make do with solving an approximate problem based on our available data points; $x^i \in \mathbb{R}^n$ the N training inputs and $y^i \in \mathbb{R}^d$ the N training outputs.

Thus the problem becomes:

$$\min_{w} f(w) \equiv \min_{w} \frac{1}{N} \sum_{i=1}^{N} L(y^i, \phi(w, x^i)) \tag{1.2}$$

A first order optimization method uses the first derivative, the gradient $\nabla f(w)$,

to update w and in turn reduce $f(w)$. A second order method uses the second derivative, the Hessian $\nabla^2 f(w)$ (or some approximation of $\nabla^2 f(w)$), to update w and in turn reduce $f(w)$.

1.3.2 Feed Forward Networks

A feed forward network is a subclass of neural networks which can be expressed as a finite sequence of l layers where the i^{th} layer is a function ϕ_i. ϕ can thus be written as

$$\phi(w, x) = \phi_l(w_l, ...\phi_3(w_3, \phi_2(w_2, \phi_1(w_1, x)))...) \tag{1.3}$$

Each layer has its own unique weights w_i which are some of the parameters that make up w. The function ϕ_i computes linear weighted sums based on the weight values w_i and the output values of the previous layer. It then applies a non-linear activation function (such as sigmoid, tanh or ReLu) to each of the output values. In this paper we consider two types of feed forward networks, autoencoders and image classifiers, both of which use convolutional layers. The weights for these layers represent the kernel of a filter which is applied to the input by the convolution operation.

1.4 Challenges in Minimization

There are many factors to consider when deciding which optimization method to use to train a Neural Network. A practical approach requires us to consider the trade-off between these methods. We consider the training time to reach a desired level of loss (or accuracy) to be the most import factor. The training time itself is determined by the convergence rate and the cost per iteration of the method.

Typically second order methods have a higher rate of convergence than first order methods, however these steps have a greater computational cost and thus take more time to perform each iteration. One way that training time can be reduced is through parallelization. For large models, training is usually done on GPUs. This can provide huge reductions in training time due to the massive parallel processing power they offer. First order methods use lots of cheap consecutive steps so have less potential for parallelization than second order methods.

Another serious challenge is ensuring the desired loss value is actually achievable. The loss functions for deep neural networks tend to be ill-conditioned non-convex functions. This can cause problems for optimizers in several ways. Ill-conditioning is particularly problematic for first order methods as this means the local gradient direction has little effect on the overall loss value. Ill-conditioning is less of a problem for second order methods as the Hessian contains curvature information, however it can still be a problem if the Hessian is close to singular. Non-convexity also introduces saddle points which attract some optimization methods, stopping the methods from minimizing the loss function beyond this point. Non-convexity also removes the guarantee of a positive semi-definite Hessian which is a requirement for some second order methods to converge.

First order methods such as Stochastic Gradient Descent (SGD) are notoriously sensitive to the learning rate parameter. If the learning rate is set too high they may never converge, whilst if the learning rate is set too low, convergence is very slow. This typically means that for a given model, many runs are required to tune the optimizer to reach the desired level of accuracy. More complex optimization techniques introduce more parameters that can be tuned. This means that more training attempts are required, particularly if using random or grid search to tune the hyperparameters. Ideally we want methods which don't re-

quire any tuning of hyperparameters. This could be achieved in a number of ways: automatic hyperparameter tuning being part of the method; the method performing consistently well for a given set of default values; or the method performing consistently well regardless of the values of the hyperparameters.

A final practical consideration is the memory usage of the method. Due to the large number of variables that are used in modern neural networks, these methods can be very memory intensive. This is particularly a problem if the training takes place on a GPU, where memory resources are more scarce. A naive second order implementation would require storing the Hessian which increases the memory requirements from $O(n)$ to $O(n^2)$ and would be prohibitive.

Chapter 2

Related Work

Ever since back-propagation was introduced [2], first order optimization methods have dominated as the technique to train neural network. Over the years there have been many advances to improve the convergence of these methods including the addition of momentum in methods such as Nesterov accelerated Gradient [3]; adaptive steps used in RMSProp [4] and AdaGrad [5]; and more sophisticated combinations of these in methods such as Adam [6]. Regardless, all first order methods work on the same premise: lots of very cheap small steps. Second order methods are less widely used in machine learning. They were first considered in theory some time ago [7], made feasible by the adjoint method of Hessian vector products [8]. Since then, interest in second order methods has continued to grow with a number of novel additions in recent years, which aim to reduce the cost of these methods as much as possible whilst keeping them robust.

The simplest feasible second order method is the Hessian Free Inexact Newton method. This method is the basis of this paper. It makes training with a large number of parameters (and a large Hessian) feasible. An excellent overview

of this method was put forward in 2010 [9], using the Gauss Newton approximation. Another detailed review of the Hessian Free method and its variants is the more recent 2018 paper [10]. Several other papers have looked at tweaks on the method such as changing the solver or the globalization technique [11, 12]. Interest in Inexact Newton has also increased as convergence has been proved when subsampling the Hessian in comparison to the batch size, reducing the cost of the operation [11, 13, 14, 15]. Over the last 10 years there has been a massive increase in availability and power of GPUs as well the software tools which make them easy to use. This is particularly helpful for second order methods, as the expensive Hessian-vector product needs to be performed over sizeable batches for stability even when using subsampling. Whilst these methods tend to perform well on small networks, tests on larger networks have mixed results [16].

There are also a number of second order methods that use techniques different to Inexact Newton. The K-FAC method [17] uses Kronecker-Factored Approximate Curvature, an approximation of the Fisher Information Matrix to produce a low cost and highly parallelizible second order method which has led to impressive results such as *Training ResNet-50 on ImageNet in 35 Epochs* [18]. L-BFGS (Limited Memory BFGS) is another method which has been proposed. This uses curvature information from previous iterations in a limited number of directions and performs BFGS updates. Variants on this include progressive batching [19] and an adaptive version [20] inspired by Adamax and AdaGrad. Similarly, Lissa creates a low order approximation of the inverse of the H [21]. The Curveball method [22] builds on the work of [23] to present a Gauss-Newton method with momentum while automatically tuning the hyperparameters. This method trains modern deep Neural Networks with millions of parameters, with a wall clock time

comparable to Adam. Each of these methods offer an interesting approach which could be expanded on, and we hope that some aspects of the methods we introduce could also be applicable to these.

Chapter 3

Methods

Second order optimization methods derive from Newton's method. At each iteration they use the minimizer of the local quadratic approximation of a function as the search direction p [24]. Let $g = \nabla f(w)$ and $H = \nabla^2 f(w)$ then for a given w:

$$\min_p f(w) + g^T p + \frac{1}{2} p^T H p \tag{3.1}$$

Assuming the Hessian H is symmetric semi-positive definite, this is solved by taking the derivative with respect to p and equating to 0. Thus $p = -H^{-1}g$. This gives us an iterative optimization algorithm:

Algorithm 1 Newton's Method

1: **while** $f(w) > \epsilon$ **do**
2: $p \leftarrow -H^{-1}g$
3: Compute step size α
4: $w \leftarrow w + \alpha p$
5: **end while**

A common method for computing α is Backtracking Line Search. This takes an initial step size ($\alpha = 1$) and decreases it by a constant factor $\beta \in (0, 1)$ until $w + \alpha p$ sufficiently reduces f.

Algorithm 2 Backtracking Line Search

1: $\alpha \leftarrow 1$
2: **while** $f(w + \alpha p) > f(w) + c\alpha \nabla f(w)^T p$ **do**
3: $\quad \alpha \leftarrow \beta\alpha$
4: **end while**

The Wolfe condition for sufficient descent is $f(w+\alpha p) > f(w)+c\alpha\nabla f(w)^T p$ for some $c \in (0, 1)$ which implies convergence when met[24]. The cost of backtracking line search is the forward pass through the network at each step to calculate $f(w+\alpha p)$ which can become costly if the search direction is only a local minimization direction. More than this, for a non-convex loss function if there is negative curvature the line search loop may never terminate. This is once of the reasons we use the trust region method as a globalization technique, detailed in section 3.6.

There are still several issues with Newton's Method, the first being that it requires the computation of H. This is very expense to store as it uses $O(n^2)$ memory. Secondly, it requires the solution of $Hp = -g$. This is typically computed through the LU factorization of H which is expensive to compute, $O(n^3)$. Lastly for a complex non-convex function $f(w) = L(y, \phi(w, x))$, H is not guaranteed to be semi-positive definite so the method may not even converge at all. For these reasons we suggest the following adaptations to reduce the computational cost and improve the stability of the method.

3.1 Inexact Newton

The most computationally expensive part of Newton's Method is solving $Hp = -g$. To reduce the cost we can instead approximately solve the equation to get the value of p within some tolerance η.

To find a suitable value of p we use a Kylov-Subspace linear solver. There

Algorithm 3 Inexact Newton

1: **while** $f(w) > \epsilon$ **do**
2: find p s.t. $\|Hp + g\| \leq \eta\|g\|$
3: Compute step size α
4: $w \leftarrow w + \alpha p$
5: **end while**

are several possible of these including the Conjugate Gradient method (CG), the Minimal Residual method (MINRES) and the Generalized Minimal Residual method (GMRES). A comparison of these [11] found that there was not a clear preferred method between these when training neural networks. We chose CG as it is the most widely used and referenced for Inexact Newton in numerical optimization literature.

The forcing term η is the tolerance to which the CG method solves the linear system. It is possible to set η to a fixed value. But we chose not to do this for two reasons, because it introduces another hyperparameter to the method and because intuitively the benefit of being close to the exact value of p will vary depending on our progress. To begin with we expect to make sufficient progress using a direction closer to the first derivative, but as we become closer to the minimum additional curvature information will be useful. The Eisenstat-Walker (EW) method (choice 2 from [25]) for choosing the forcing term does exactly that, taking the following value at the kth iteration:

$$\eta_k = \gamma \left(\frac{\|\nabla f(w_k)\|}{\|\nabla f(w_{k-1})\|} \right)^{\mu} \tag{3.2}$$

We take the default values from the paper $\eta_0 = 0.1, \gamma = 1, \mu = \frac{1+\sqrt{5}}{2}$ and use the default maximum threshold $\eta_{\max} = 0.9$ but avoid using the safeguard ($\eta_k = \max\{\eta_k, \gamma\eta_{k-1}^{\mu}\}$ whenever $\gamma\eta_{k-1}^{\mu} > 0.1$) as we found it gave much larger tolerance values. The safeguard made us stop prematurely, falling on the wrong side of the

trade-off between computational cost and minimization progress, and we didn't run into the problem of shrinking tolerances it was meant to stop. Similar to [9], we found that in later stages the more CG runs are needed to reach useful curvature information. For this reason we also tuned the maximum number of CG iterations to be adaptive by $\max\{10, 0.1k\}$ where k is the current iteration of the full minimization routine.

3.2 Conjugate Gradient Method

We chose CG as the solver for step 2 of Algorithm 3. This method solves a linear system $Ax = b$ to a desired tolerance η, by interpreting the system as the minimization problem:

$$\min_x \frac{1}{2}x^T Ax - b^T x \tag{3.3}$$

This has a solution when the gradient $Ax - b = 0$. In our case the linear system is $Hp = -g$. The method works by iteratively reducing the residual $r_k = Ax_k - b$ in orthogonal directions p_k from the Krylov Subspace $\mathcal{K}_k(A, r_0) = \text{span}\{r_0, Ar_0, A^2r_0, \ldots, A^kr_0\}$. If we meet a direction of negative curvature we terminate early and handle it as explained in section 3.7.

3.3 Hessian Free Vector Product

The most attractive thing about the CG-Inexact Newton method is that H is never explicitly required, we just need to evaluate the action of H on a vector. If the CG algorithm is implemented carefully it only needs to evaluate Hp_k once per iteration. This gives a huge computational advantage over traditional Newton particularly when H is very large or sparse. The method to evaluate the

Algorithm 4 Conjugate Gradient

$r_0 \leftarrow Ax_0 - b; \quad p_0 \leftarrow -r_0; \quad k \leftarrow 0$
while $\frac{\|r_k\|}{\|b\|} > \eta$ **do**
    if $p_k^T A p_k < 0$ **then**
        Terminate early
    end if
    $\alpha_k \leftarrow \frac{r_k^T r_k}{p_k^T A p_k}$
    $x_{k+1} \leftarrow x_k + \alpha_k p_k$
    $r_{k+1} \leftarrow r_k + \alpha_k A p_k$
    $\beta_{k+1} \leftarrow \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$
    $p_{k+1} \leftarrow -r_{k+1} + \beta_{k+1} p_k$
    $k \leftarrow k + 1$
end while

the Hessian vector product is explained in [8]. In a similar way to Back Propagation [2] performing a forward and backwards sweep of the Neural Network to evaluate the gradient, the Hessian vector product must perform two forward and backward sweeps, to compute $Hv = \nabla(\nabla f(w)^T v)$. It is worth mentioning that a similar technique exists for calculating the action of the Gauss Newton matrix on a vector [23]. Analysis shows this method to be cheaper than that of the Hessian, although there is not currently a consensus as to which is preferred, whether the saving in computation outweighs the reduction in curvature information. Some argue that Gauss-Newton is better due to the increased stability and guaranteed positive definiteness [9] but others say it is more prone to sticking at saddle points and makes less progress in the same time [26, 12]. We view this as beyond the scope of this paper, leaving it open to future work. Whilst using a Hessian Free method reduces the cost when compared to traditional Newton, the Hessian vector product is the most expensive operation in our method so it is still important to minimize the number of evaluations.

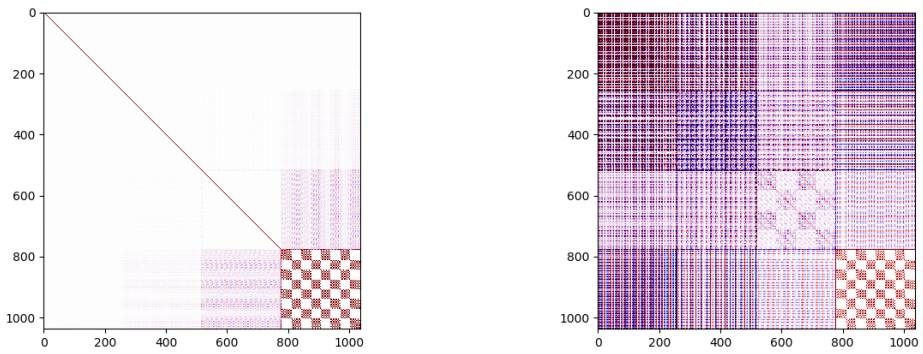

Figure 3.1: The Hessian for an MNIST Autoencoder after 0 iterations (left) and 200 iterations (right). Positive and negative values are represented by red and blue respectively.

3.4 Stochastic Hessian

One of the greatest improvements to gradient descent is the addition of mini-batching used in SGD [27]. Mini-batching provides both a reduction in per iteration cost and adds randomization which helps to avoid saddle points and improving generalization of the model. It is natural to seek an equivalent way to reduce the cost of the Hessian vector evaluations. Recent advances have proved convergence of Inexact Newton with a subsampled Hessian, and with both a subsampled Hessian and gradient [11, 13, 14, 15]. More than this, the subsample size of the Hessian can be much smaller relative to that of the gradient, further reducing the cost of evaluating the Hessian vector product. We use N_X to refer to the mini-batch size at each iteration and N_S to refer to the Hessian subsample size.

3.5 Preconditioning

Since the Hessian-vector product is the most expensive operation in the CG-Inexact Newton method, we can significantly speed up the method by reducing the number of times this operation is called. The convergence rate of CG is dependent on the clustering of the eigenvalues of H. If many of the eigenvalues are close in value to each other, the method converges much faster. Preconditioning takes advantage of this by transforming our linear system to one with a more favourable eigenvalue distribution, solving this system to the desired tolerance then transforming the solution back to the original linear system. This transformation is based on some non-singular matrix C with $\hat{x} = Cx$. When CG with preconditioning is implemented, we actually make use of a different matrix, $M = C^T C$.

Algorithm 5 Preconditioned Conjugate Gradient

$r_0 \leftarrow Ax_0 - b; \quad y_0 \leftarrow M^{-1}r_0; \quad p_0 \leftarrow -y_0; \quad k \leftarrow 0$
while $\frac{\|r_k\|}{\|b\|} > \eta$ **do**
  if $p_k^T A p_k < 0$ **then** ▷ Stop on Negative Curvature
    Return x_k
  end if
  $\alpha_k \leftarrow \frac{r_k^T y_k}{p_k^T A p_k}$
  $x_{k+1} \leftarrow x_k + \alpha_k p_k$
  $r_{k+1} \leftarrow r_k + \alpha_k A p_k$
  $y_{k+1} \leftarrow M^{-1} r_{k+1}$
  $\beta_{k+1} \leftarrow \frac{r_{k+1}^T y_{k+1}}{r_k^T y_k}$
  $p_{k+1} \leftarrow -y_{k+1} + \beta_{k+1} p_k$
  $k \leftarrow k + 1$
end while

This is a great idea in theory but raises the question, what is a good preconditioner for the Hessian of a neural network? We want a matrix which is cheap to compute but is a good approximation for the Hessian. A few options to consider include a regularized Fisher information matrix [9, 28], an approximation of the

Gauss Newton [29] and a Low rank L-BFGS matrix [30]. We chose the regularized Fisher information preconditioner as it requires the least computation to generate, assuming we already have computed the gradient $\nabla f(w)$.

$$M = (diag(\nabla f(w) \odot \nabla f(w)) + \lambda I)^{\alpha} \tag{3.4}$$

Where a typical value of α would be 0.75.

3.6 Trust Region Globalization

The biggest challenge we face when training neural networks is ensuring convergence. The loss functions are non convex with many plateaus and saddle points [1] which can attract second order methods or lead to poor second order approximations. Trust regions [31] offer a more robust way to traverse the troublesome loss function landscape than just taking fixed steps in the Newton direction or using backtracking line search. As we will also see, they give us a way to reduce the computation at each step and avoid the expensive computation of the CG method when there is little to gain from it.

The fundamental idea behind a trust region (TR) is to approximately solve the minimization problem for a fixed subregion, most commonly a ball around the current position. The size of the ball changes at each iteration depending on how well the TR subproblem approximates the objective function. This is computed by the metric ρ. If the approximation is poor we reduce the size of the TR so it will be easier to reduce the loss function at the next iteration. If on the other hand the approximation is good, we expand the TR to allow us to make even more progress at the next iteration.

Given a TR of radius Δ, the minimization problem we want to solve at each iteration is:

$$\min_p m(p) = g^T p + \frac{1}{2} p^T B p \qquad \text{s.t. } \|p\| \leq \Delta \tag{3.5}$$

For $g = \nabla f(w)$ and B a matrix such as the Hessian, possibly with regularization. This problem, known as the TR subproblem can be viewed as a bounded quadratic approximation of $f(w+p) - f(w)$. There are many methods that can be used to approximately solve the subproblem which we will discuss here.

Algorithm 6 Trust Region

while $f(w) > \epsilon$ **do**
  Find p by solving TR subproblem
  $\rho \leftarrow \frac{f(w)-f(w+p)}{-m(p)}$
  if $\rho < 0.25$ **then** ▷ Reduce radius and don't update
    $\Delta \leftarrow 0.25\|p\|$
  else
    if $\rho > 0.75$ and $\|p\| = \Delta$ **then** ▷ Increase radius
      $\Delta \leftarrow \min\{2\Delta, \Delta_{\max}\}$
    end if
    $w \leftarrow w + p$ ▷ Update
  end if
end while

3.6.1 Inexact Newton Step

The simplest solution to the subproblem is to inexactly compute the Newton step p^N. This is an approximation of the minimizer of $m(p)$ so if it is within the TR we can use it, if not we can scale the step by $\frac{\Delta}{\|p^N\|}$ to give the point at which it intersects the TR barrier. This simple approach doesn't always work in practice. When p^N is far from the TR and there is high curvature using only the Newton direction can give poor results, often worse than backtracking line search.

3.6.2 Dogleg Method

When the TR is very small, the negative gradient is a better minimization direction than the Newton direction but as the TR increases in size the linear approximation is no longer reliable and we should use the second order approximation. When the TR is between these two cases we would like to interpolate the two directions, this is provided by the dogleg method. We use the Cauchy Point $p^C = -\frac{g^T g}{g^T H g} g$, which is the unconstrained minimizer along the steepest descent direction. Then we find the minimum value in the TR along two line segments, first from our current position to the Cauchy Point, then from the Cauchy Point to the quadratic minimizer. A 2D example of this is shown in fig. 3.2. Formally, this is the problem of finding τ to give the optimum $p(\tau)$ defined as:

$$p(\tau) = \begin{cases} \tau p^C & 0 \leq \tau \leq 1 \\ (2-\tau)p^C + (\tau - 1)p^N & 1 \leq \tau \leq 2 \end{cases}$$

The optimum value of τ can be found by solving a quadratic equation. There are a couple of tricks we can use with this method to decrease the computational cost when using the dogleg method with Inexact Newton [32]. If the Cauchy Point is outside of the TR, we do not need to compute p^N at all. Before computing p^N we should also check if p^C is within the tolerance for the Inexact Newton condition $\|Hp^C + g\| \leq \eta\|g\|$. Note that this method does not work well if the matrix B is not positive-semidefinite. We use a regularization term to try to enforce this but to improve robustness in the case that both p^C and p^N lie in the TR, we should set p to be whichever of them reduces the objective function the most.

Algorithm 7 Dogleg Minimization

Compute p^C
if $\|p^C\| \geq \Delta$ **then**
    $p \leftarrow \frac{\Delta}{\|p^C\|} p^C$
else if $\|Hp^C + g\| \leq \eta \|g\|$ **then**
    $p \leftarrow p^C$
else
    Compute p^N with CG method
    if $\|p^N\| \leq \Delta$ **then**
        if $f(w + p^N) < f(w + p^C)$ **then**
            $p \leftarrow p^N$
        else
            $p \leftarrow p^C$
        end if
    else
        Compute τ
        $p \leftarrow p(\tau)$
    end if
end if
Update w and TR with p

3.6.3 2D Subspace Minimization

Finding the optimum position on the dogleg path is relatively cheap but for a small increase in computational cost we can do much better. 2D subspace minimization searches for the best solution to the TR subspace problem on the hyperplane span$\{p^C, p^N\}$. This hyperplane includes all of the points along the dogleg path and many more. A 3D example of this is shown in fig. 3.3. The 2D TR subspace problem is

$$\min_{\alpha,\beta} m(\alpha, \beta) = g^T P \begin{pmatrix} \alpha \\ \beta \end{pmatrix} + \frac{1}{2} \begin{pmatrix} \alpha, \beta \end{pmatrix} P^T H P \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \qquad \text{s.t. } \|p\| \leq \Delta \tag{3.6}$$

For $P = [p^C p^N]$ we then set $p = \alpha p^C + \beta p^N$. The main additional cost of this method is computing $P^T H P$, which requires 3 Hessian-vector products. Fortu-

nately we already have computed $g^T H g$ when generating p^C so can reuse this value. All that remains to do then is solve a quartic equation to find α and β [33]

Algorithm 8 2D Subspace Minimization

1: Compute p^C
2: Compute p^N
3: Solve 2D Subspace subproblem to get α, β
4: $p = \alpha p^C + \beta p^N$
5: Update w and TR with p

3.6.4 Low-cost 2D Subspace Minimization

We can reduce the cost of the 2D Subspace method using the same trick as with the dogleg method. When the TR is small there is no need to perform the costly operations generating p^N, instead only use the gradient direction.

Algorithm 9 Low Cost 2D Subspace Minimization

1: Compute p^C
2: **if** $\|p^C\| \geq \Delta$ **then**
3: $\quad p \leftarrow \frac{\Delta}{\|p^C\|} p^C$
4: **else if** $\|Hp^C + g\| \leq \eta \|g\|$ **then**
5: $\quad p \leftarrow p^C$
6: **else**
7: $\quad$ Compute p^N with CG method
8: $\quad$ Solve 2D Subspace subproblem to get α, β
9: $\quad p = \alpha p^C + \beta p^N$
10: **end if**
11: Update w and TR with p

3.7 Regularization

Regularization is the addition of a term to the problem:

$$\min_w f(w) \equiv \min_w \frac{1}{N} \sum_{i=1}^{N} L(y^i, \phi(w, x^i)) + \frac{1}{2} \lambda \|w\|^2 \tag{3.7}$$

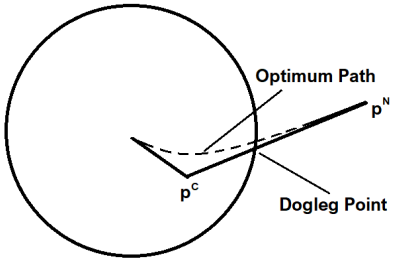

Figure 3.2: The optimum path versus the dogleg path, with the dogleg point on the TR boundary

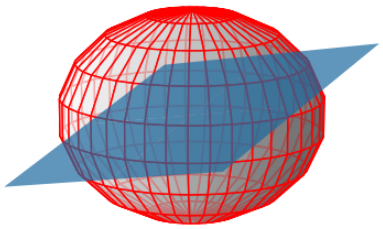

Figure 3.3: 2D subspace in a 3D TR

Regularization serves two purposes, to ensure a positive-semidefinite Hessian and to limit the step size. By adding a regularization parameter $\frac{1}{2}\lambda\|w\|^2$ the Hessian increases by λI. This increases the value of the eigenvalues by λ. If the unregularized Hessian has eigenvalues $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n$ and the regularization parameter λ is chosen such that $\lambda > \min\{0, \lambda_n\}$ then the regularized Hessian, H, is guaranteed to be positive-semidefinite. This has the effect of transforming the landscape of the non-convex loss function to a function which is easier to minimize.

We also use regularization to limit the size of the weights to stop them exploding to extreme values. There are times when the second order approximation is not particularly accurate and the newton direction is very large. Including the size of the variable w as a term in the objective function stops the weights from getting too large. This reason for regularizing is less of a motivation for us as it is also achieved by the TR. In fact, this style of damping via regularization can be viewed as a soft TR, as shown by the Levenberg-Marquardt method commonly used with the Gauss-Newton, hence is less relevant to our method.

This leads us to the question, how to figure out the size of λ to use? We don't

want to introduce another hyperparameter so need to tune it automatically. For our purposes, the size needs to be relative to the largest negative eigenvalue. The Lanczos method [34] is a Krylov Subspace method which estimates the largest and smallest eigenvalues of a matrix A. Similar to CG, Lanczos creates a series of orthogonal vectors from a Krylov subspace. These are used for a tridiagonal approximation T_k of the factorization of A with diagonal values α_k and off diagonal values β_k. The T_k can be easily diagonalized to give an approximation of the most dominant eigenvalues. After k iterations the method generates k approximate eigenvalues, $\lambda'_1 \geq \cdots \geq \lambda'_k$ with $\lambda'_1 \leq \lambda_{\max}$ and $\lambda'_k \geq \lambda_{\min}$, respectively the largest and smallest eigenvalues of A. Equality is only reached after n iterations, however, experimentally we found that the method reaches a close approximation within 10 iterations. We can then multiply by a factor slightly greater than 1 to ensure positive semi-definiteness. Again, similar to CG, we can implement this in a Hessian-free fashion. We do not explicitly need to compute the matrix A (H in our case), only the product with a vector. The Lanczos method is still relatively costly though, so we opt not to run it at every iteration; instead it runs once to initialize the regularization before we begin the minimization loop. In the case that Lanczos returns a positive value for the smallest eigenvalue we still suggest using a small regularization parameter for its damping properties and improved stability.

As we move closer to the optimum point the amount of regularization required actually reduces and we notice the effect of the damping phenomenon. Our steps become smaller and fall within the TR instead of beyond it. To deal with this we must reduce the amount of damping. Using the TR as an indicator we multiply the regularization by a constant factor $0 < \zeta < 1$ whenever we update the weights with a step which falls within the TR.

Algorithm 10 Lanczos Algorithm

$q_0 \leftarrow 0; \beta_0 \leftarrow 0$
Initialize q_1 to a random unit vector
for $i \leftarrow 1$ to k **do**
    $z \leftarrow Aq_i$
    $\alpha_i \leftarrow z_i^T z$
    $z \leftarrow z - \alpha_i q_i - \beta_{i-1} q_{i-1}$
    Orthogonalize z with respect to previous q_i
    $\beta_i \leftarrow \|z\|$
    $q_{i+1} \leftarrow z/\beta_i$
end for
Create $k \times k$ tridiagonal matrix T_k from α_k and β_k values
Compute eigenvalues of T_k

Algorithm 11 TR Minimization with automatic regularization

Run Lanczos for 10 iterations to approximate $\lambda_{\min}$
$\lambda \leftarrow \max(-1.2\lambda_{\min}, \lambda')$
while $f(w) > \epsilon$ **do**
    Find p by solving TR subproblem
    Compute ρ and if large enough update w
    if w was updated and $\|p\| < \Delta$ **then**
        $\lambda \leftarrow \zeta\lambda$
    end if
end while

Algorithm 11 for automatically tuning λ actually introduces two new parameters: λ', the value to set λ to if the initial Hessian is positive-definite and ζ, the reduction factor. We argue that their value is not too important as long as it is reasonable. λ' just needs to be positive. If it is too large it may reduce the rate of convergence at the start but the reduction factor will take care of this. We use 0.1 as a default. For the reduction factor, as long as ζ is less than 1 we will reduce λ sufficiently. Section 5.3.2 gives the details of how we determined the values for these parameters as well the 10 iterations and factor of 1.2.

Reducing the regularization may lead to directions of negative curvature, but these can be used to our advantage and indeed give us better results [26, 35, 36].

Some methods, such as Saddle-Free Newton [37] and others [38, 11], actively seek out directions of negative curvature to use. This can be expensive in practice, so we take a more relaxed approach. When we find a direction of negative curvature we stop the CG iterations early and make use of it in solving the TR subproblem. This is easily done for the 2D subspace method, but more care must be taken with the dogleg method [32], hence the additional checks.

3.8 Momentum

Momentum has become one of the key techniques for improving the convergence of first order methods. Stochastic Gradient Descent with momentum, Nesterov accelerated gradient and Adam all use their own version of momentum to give iterative updates that use more than just the current local gradient information. The idea is that by combining the step direction from the previous iteration with the steepest descent direction at the current iteration, a direction of persistent descent can be generated which is better at consistently minimizing the loss function. It also helps counter the noise that comes from using a stochastic method and avoids sticking at saddle points. Momentum has been used far less with second order methods but it is worth considering how it could be incorporated. The general equation for the kth momentum step is:

$$w_{k+1} = w_k + \alpha_k p_k + \beta_k (w_k - w_{k-1}) \tag{3.8}$$

The simplest approach is to pick α_k first using whatever TR method is preferred and using a fixed value β for all β_k. This is essentially stochastic gradient descent using a second order step direction instead of the gradient. One clear issue with

this method is that it introduces the additional hyper parameter β which could require tuning. A more robust technique is to apply 2D subspace minimization to the two directions p_k and $m_k = w_k - w_{k-1}$. This is the suggested technique used by Curveball [22] which comes at the cost of 3 additional Hessian vector products to calculate the 2×2 matrix involved. Whilst this cost can be reduced through some clever reuse of previous values, some additional network sweeps will always be required which make it less favourable.

A middle ground between the two is an adaptive method which requires no tuning but equally isn't computationally expensive [39]. The step is calculated as follows:

$$y_k = w_k + p_k \tag{3.9}$$

$$\beta_k = \min\{\beta, \|\nabla f(y_k)\|, \|p_k\|\} \tag{3.10}$$

$$v_k = y_k + \beta_k(y_k - y_{k-1}) \tag{3.11}$$

$$w_{k+1} = \underset{w \in \{y_k, v_k\}}{\operatorname{argmin}} f(w) \tag{3.12}$$

The additional computation here is the calculation of $\nabla f(y_k)$ and the evaluation of $f(y_k)$ and $f(v_k)$ and due to the comparison we make at the end we ensure the step taken is never worse than when we don't use momentum. We make a comparison between these in the results section.

One final technique to consider is CG warm start. This is a very different style of momentum which is only applicable to Hessian Free methods. Here we initialize the conjugate gradient routine with the step direction used at the previous iteration. The idea of initializing to this value is that if the step direction is similar to the previous iteration the number of inner CG iterations will be vastly reduced. In practice this isn't always the case, particularly when using

a stochastic Hessian and a poor initialization could lead to more CG iterations rather than less.

A comparison of these techniques was performed in section 5.3.3 and based on the results we decided to only use CG warm start as a momentum technique.

Chapter 4

Costs of Proposed Methods

4.1 Proposed Algorithms

We combine the techniques described in chapter 3 to define three Hessian-free, TR methods: *2D subspace*, *low-cost 2D subspace* and *dogleg* as shown in algorithms 12 and 13.

Algorithm 12 2D Subspace Method

1: Run Lanczos for 10 iterations to approximate $\lambda_{\min}$
2: $\lambda \leftarrow \max(-1.2\lambda_{\min}, 0.1)$
3: **while** $f(w) > \epsilon$ **do**
4: Compute EW tolerance η
5: Compute p^N with CG to EW tolerance
6: Find p by solving 2D subspace subproblem with $\nabla f(w)$ and p^N
7: Compute ρ and update w, Δ and λ accordingly
8: **end while**

4.2 Parallelization and Iteration Cost

By running the code on a large enough GPU we are able to completely parallelize the most expensive operations, that is sweeps of the network. Assuming we have

Algorithm 13 Low-cost 2D Subspace and Dogleg Methods

Run Lanczos for 10 iterations to approximate $\lambda_{\min}$
$\lambda \leftarrow \max(-1.2\lambda_{\min}, 0.1)$
while $f(w) > \epsilon$ **do**
    Compute EW tolerance η
    Compute p^C
    if $\|p^C\| \geq \Delta$ **then**
        $p \leftarrow \frac{\Delta}{\|p^C\|} p^C$
    else if $\|Hp^C + g\| \leq \eta\|g\|$ **then**
        $p \leftarrow p^C$
    else
        Compute p^N with CG to EW tolerance
        if $\|p^N\| \leq \Delta$ **then**
            if $f(w + p^N) < f(w + p^C)$ **then**
                $p \leftarrow p^N$
            else
                $p \leftarrow p^C$
            end if
        else
            Find p by solving 2D subspace or dogleg subproblem with p^C and p^N
        end if
        Compute ρ and update w, Δ and λ accordingly
    end if
end while

enough GPU resources to handle the entire mini-batch in parallel we can assess the cost of a single iteration of each method by the number of sweeps the method requires, not taking mini-batch size or subsampling into consideration. We count the number of sweeps as follows:

- Evaluation of the model - 1 sweep
- Gradient - 2 sweeps
- Hessian vector product - 4 sweeps for the first call and 2 sweeps for all successive calls in the same iteration (assuming reuse of inner sweeps)

We can now count the number of sweeps that each of our methods use per it-

eration. Note that these counts assume some caching is used so the gradient evaluation is done once and the value is reused when needed.

4.2.1 Lanczos Regularization

If we perform 10 iterations of Lanczos at the start of the routine, first we evaluate the Hessian inner loop (2 sweeps), then each iteration evaluates a Hessian-vector product (2 sweeps).

$$\textbf{Total sweeps} = 22 \tag{4.1}$$

4.2.2 2D Subspace Method

For this method we have 4 parts: initialization of gradient and Hessian inner loop (4 sweeps), computation of p^N (2(#CG iters + 1) sweeps), 2D subspace minimization (4 sweeps) and finally perform the TR update (4 sweeps).

$$\textbf{Total sweeps} = 2(\#\text{CG iters} + 1) + 12 \tag{4.2}$$

4.2.3 Low-cost 2D Subspace Method

For this method we have a couple of different branches depending on whether we use the low-cost or the full computation: first in all cases we have the initialization of gradient and Hessian inner loop (4 loops) and computation of p^C (2 sweeps). If p^C is beyond the boundary or p^C satisfies the EW condition (0 sweeps) else computation of p^N (2(#CG iters + 1) sweeps) and 2D subspace minimization (4 sweeps). Finally in both cases perform the TR update (4 sweeps).

$$\textbf{Total sweeps} = 10 \textbf{ or } 2(\#\text{CG iters} + 1) + 14 \tag{4.3}$$

4.2.4 Dogleg Method

Similar to the previous method, there are different branches depending on whether we use the low-cost or the full computation: first in all cases we have the initialization of gradient and Hessian inner loop (4 sweeps) and computation of p^C (2 sweeps). If p^C is beyond the boundary or p^C satisfies the EW condition (0 sweeps) else computation of p^N (2(#CG iters + 1) sweeps). If p^N is beyond the TR we compute the dogleg point (0 sweeps) else we compare the loss of p^C and p^N (2 sweeps). Finally in all cases perform the TR update (4 sweeps).

$$\textbf{Total sweeps} = 10 \textbf{ or } 2(\#\text{CG iters} + 1) + 10 \textbf{ or } 2(\#\text{CG iters} + 1) + 12 \quad (4.4)$$

Chapter 5

Results

We now analyse the experimental outcomes of our methods. We experimentally justify the choices made in constructing the method and demonstrate the hyper parameter robustness. Then we make comparisons of our method against first order methods for a range of models.

Unless explicitly stated, all results we display are the average of five runs of the test, using a different random initialization for each run. We seed the runs to keep the random initializations and batch shuffling the same for each method. We ran the tests on a virtual machine with a single 32GB NVidia V100 GPU and 8 AMD EPYC 7551p vCPUs.

We record the loss and accuracy against three different measurements: the number of iterations, number of network sweeps and time as each gives a different insight. Plotting against the number of iterations gives a clear idea of the rate of convergence of the method. This scale is particularly useful when analysing different versions of our methods and the parameters. Network sweeps gives a fairer

Name	Model	Data Set	Input Size	Parameters
AEM	Autoencoder	MNIST	(28,28,1)	1037
AEC	Autoencoder	CIFAR-10	(32,32,3)	11299
L5M	LeNet5 Classifier	MNIST	(32,32,1)	61706
R18C	ResNet18 Classifier	CIFAR-10	(32,32,3)	11190730

Table 5.1: Details of each model used in testing

theoretical comparison between different methods, counting the number of expensive operations that require full sweeps of the network. For SGD and Adam, the two first order methods we consider, this is exactly two sweeps per iteration. This is also an implementation agnostic metric so gives a better idea of the theoretical performance without having to worry about implementation details. Lastly, plotting against time gives a realistic indication of the performance.

5.1 Models

We test our methods on 4 different models, 2 autoencoders and two image classifiers. The datasets used were the handwritten digits dataset MNIST[40] (60000 training and 10000 testing data points) and collection of 10 different image classes CIFAR-10[41] (50000 training and 10000 testing data points) as provided by Tensorflow. We used similar autoencoder models to [11]. For the MNIST dataset we use [4,4,4,1] filters at each level, with respective sizes [8,4,4,8], for CIFAR-10 we use [4,4,4,8,4,4,3] filters at each level with sizes [16, 8, 8, 4, 4, 8, 8, 16]. The step size is 2 in both cases. We used well established image classification models LeNet5 [42] and the much larger ResNet18[43]. Due to the input dimensions required by LeNet5 we pad the MNIST dataset from (28,28,1) to (32,32,1) for this model. The input size and number of parameters of each model are shown in table 5.1

5.2 Implementation

One of the contributions of this paper is the software package containing implementations of the methods and a test suite to reproduce our results. The code is implemented in Tensorflow v2.1 which we picked for its flexibility, ease of use on GPU and TPU and the large community. It also fully compatible with the Keras module which makes testing possible on a large range of Neural Networks. We chose to use eager execution for our testing, that is a real time execution of the operations as opposed to a graph execution which fully defines the computation graph before executing the operations. Although the graph execution is marginally faster, Eager execution is the standard implementation method as of Tensorflow v2.0 and it makes debugging and analysis of the methods easier. For fairness, we implemented the first order methods ourselves as a composition of tensorflow operations rather than using the predefined Tensorflow optimizers which have their own well optimized GPU kernels.

5.3 Method Justification and Hyperparameter Values

Our methods use many additional techniques when compared to the original Inexact Newton method. Whilst this increases the complexity of our methods, each of these additions helps us address our two goals for the method, reducing the computational cost or reducing the need to tune hyper parameters. Here we look at the experimental results with the method to justify their incorporation and show how they work. Unless explicitly stated, here we use the 2D-Subspace method with default parameters and an average of 5 runs.

5.3.1 CG Tolerance

Th CG tolerance determines how many CG iterations to perform. We can either use a fixed tolerance or the EW method to set the value of η. Whilst the EW method introduces several hyperparameters $(\gamma, \mu, \eta_{\max}, \eta_0)$ we find that their default values (1, $\frac{1+\sqrt{5}}{2}$, 0.9, 0.1 respectively) are sufficient. Figure 5.1 shows that for AEM we see that the EW method performs much better, reaching a lower loss value and running much faster. This is because it mostly uses a small number of iterations for CG at each stage. The fixed methods always use the maximum number of iterations, except towards the end when the maximum number of iterations has increased so much that the tolerance of 0.1 is achievable. Figure 5.2 shows that for L5M we see slightly different behaviour. Initially the fixed methods reduce the loss much faster than EW, however the smallest tolerances overfit the function and only the 0.1 tolerance stays low. The EW method closes the gap between them eventually.

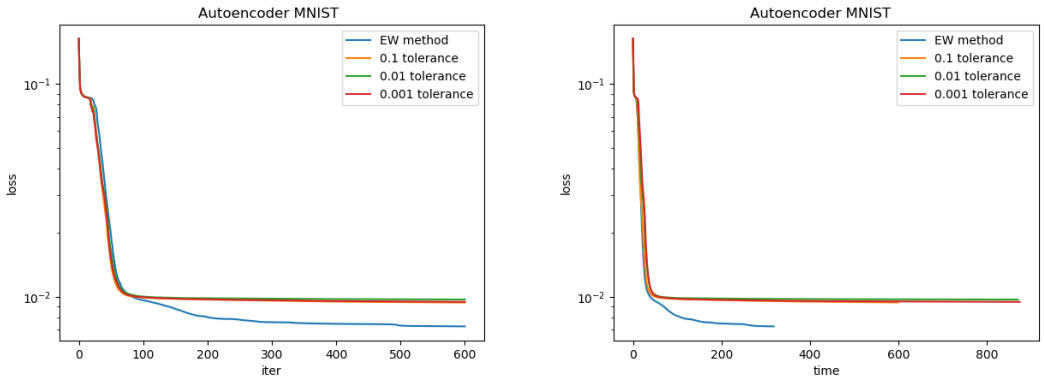

Figure 5.1: AEM for different CG tolerance routines with respect to iterations and time

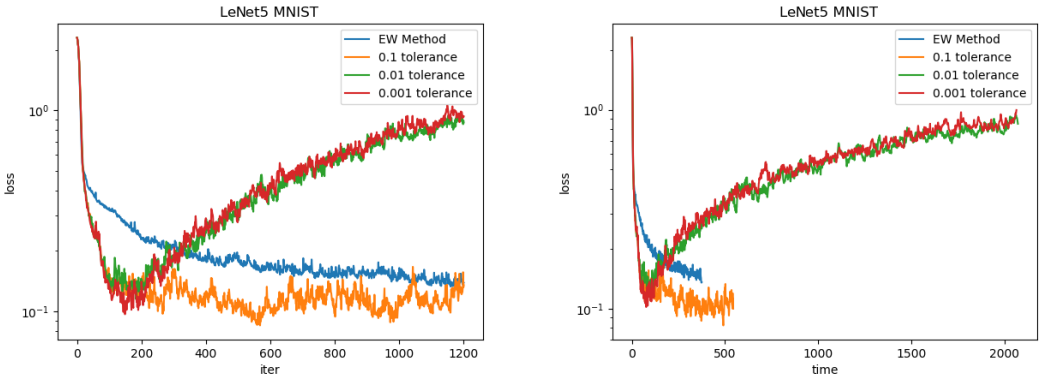

Figure 5.2: L5M for different CG tolerance routines with respect to iterations and time

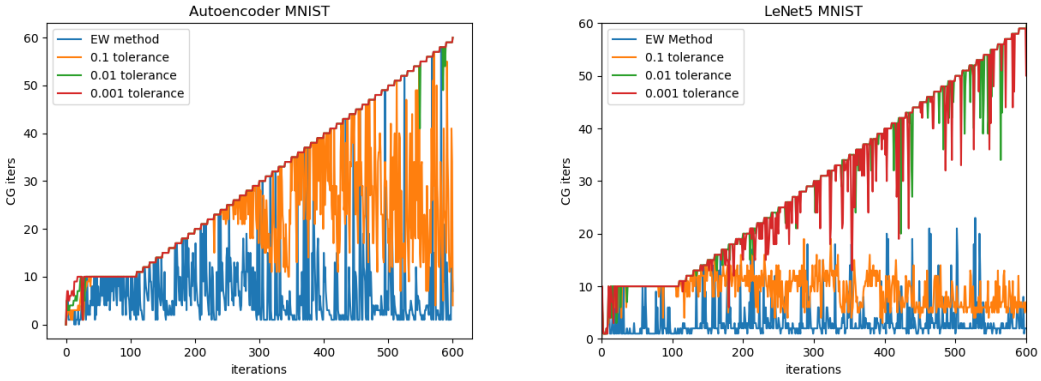

Figure 5.3: The number of iterations of CG performed at each iteration of the minimization routine

5.3.2 Regularization

Our regularization parameter λ is tuned in two ways: we initialize it with Lanczos, then we reduce it by a constant factor each time our step is within the TR. There are four different parameters introduced by this method: the number of Lanczos iterations; the factor to multiply the initial value by, a minimum initialization regularization and the factor by which to reduce the regularization. Analysis of the convergence of Lanczos on our models, as shown in fig. 5.4, demonstrates that we reach a relative error of 0.1 within 10 iterations. The Lanczos method ap-

proaches the min eigenvalue from above, so if $\lambda_{\min}$ is negative our approximation $\lambda'_{\min}$ is $\lambda_{\min} < \lambda'_{\min} < 0.9\lambda_{\min}$. Thus $\frac{\lambda'_{\min}}{0.9} < \lambda_{\min}$. So by performing 10 iterations of Lanczos and multiplying by 1.2 (which is slightly larger than $\frac{1}{0.9}$ to be safe) we guarantee the regularization will make our Hessian positive definite.

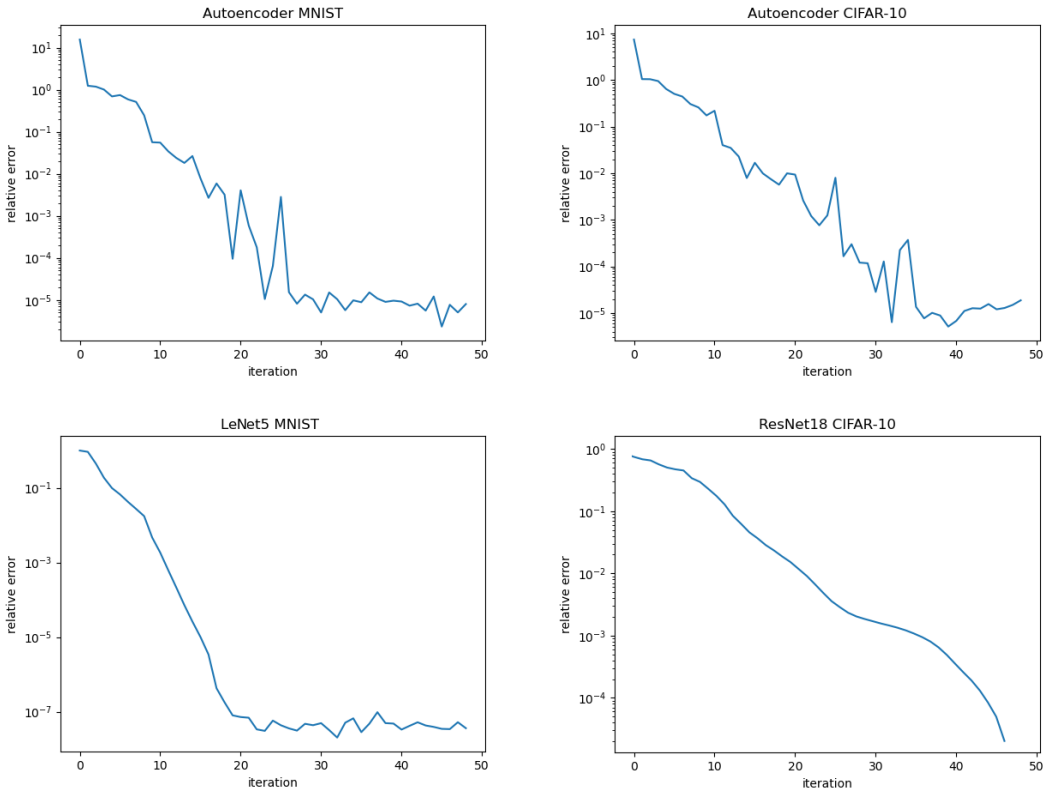

Figure 5.4: Convergence of Lanczos for each model

As previously mentioned, the minimum value for regularization initialization is fairly arbitrary. Even if a very large value is chosen the reduction factor takes care of it. We found 0.1 was a suitable value. Lastly for the reduction factor, fig. 5.5 shows the size of the minimum eigenvalue of the Hessian does indeed reduce as we reduced the loss function. We tried a few different values for the regularization reduction factor, but as seen in fig. 5.6 the results were relatively similar for 0.99, 0.9 and 0.5. For this reason we chose 0.9 as our reduction factor.

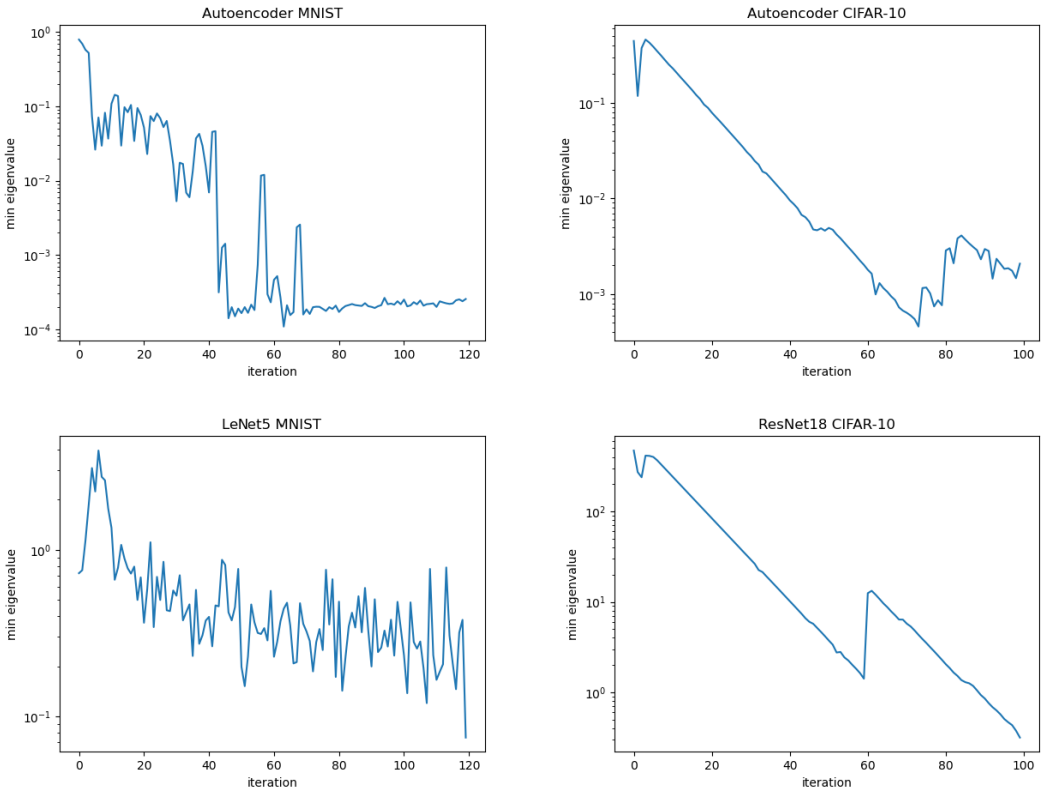

Figure 5.5: Magnitude of the smallest eigenvalue found after 10 iterations of Lanczos

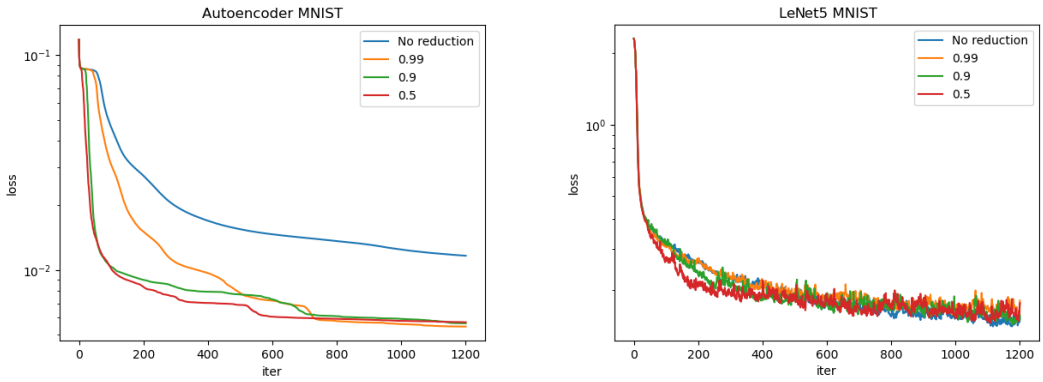

Figure 5.6: Loss per iteration for different regularization reduction parameters

5.3.3 Momentum

There are two types of momentum we discussed. First we looked at possible momentum schemes where the step is a linear combination of the previous and the new step directions. Figure 5.7 shows that these methods have little effect on the convergence of our functions; hence our decision to exclude them from our final method. The second technique we tried was CG warm start, using the previous step direction to initialise CG. Figure 5.8 shows CG warm start gave substantially better performance for the autoencoder. Whilst it was a slower start per iteration than without for the image classifier, the time saved due to few CG iterations makes it a preferable technique.

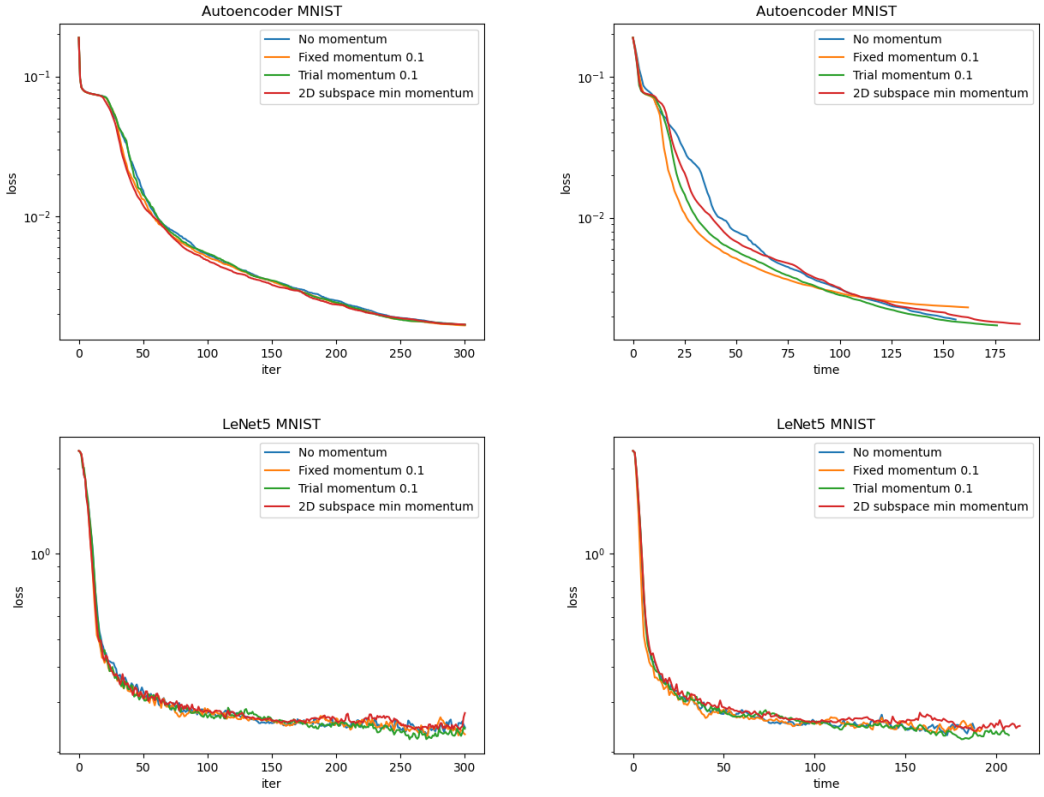

Figure 5.7: Loss by iteration and time for AEM and L5M with different momentum types

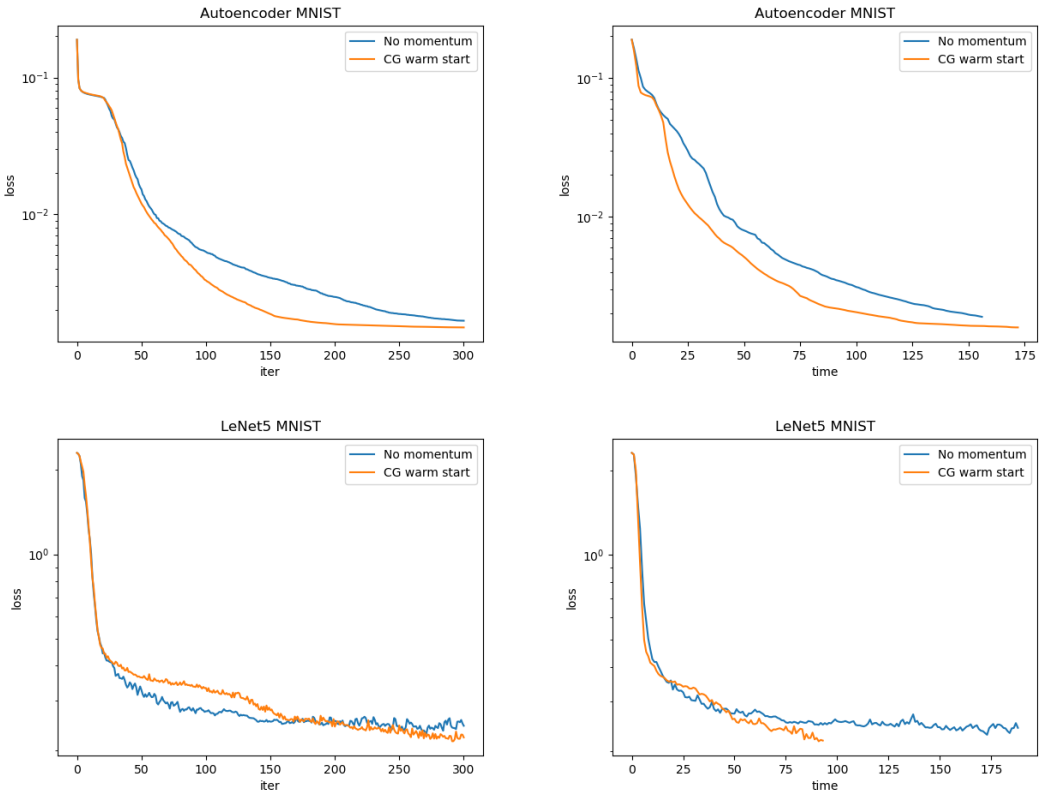

Figure 5.8: Loss by iteration and time for AEM and L5M with CG warm start

5.3.4 Minibatching and Subsampling Robustness

Two important hyperparameters to consider are N_X the minibatch size at each iteration and N_S the number of subsamples from the minibatch used for the Hessian. We performed two experiments to improve our understanding of the effect of these parameters on convergence and memory consumption both for RAM and GPU memory. We measured the maximum RAM usage of the methods with the unix command `/usr/bin/time`. Currently the Tensorflow GPU profiler does not include the exact GPU memory allocations and by default it reserves all available memory. We configured Tensorflow to only allocate the necessary GPU memory and measured the size of this allocation with `nvidia-smi` though this

should be considered an approximate upper limit on GPU memory.

First we looked at the effect of changing N_X without subsampling (so $N_S = N_X$). Table 5.2 and table 5.3show the RAM and GPU memory respectively. Minibatch size had little effect on the convergence rate or the run time, however the memory cost increases both on and off the GPU. Given the the greater increase in GPU memory compared to RAM as batch size increases (and the relative prevalence of available RAM over GPU memory) the GPU memory seems to be the limiting factor in mini-batch size. Comparing the GPU memory usage to SGD we see that for larger minibatch sizes the memory usage is lower, which demonstrates the additional memory costs of second order methods. Figure 5.9 shows that the batch size doesn't drastically effect the convergence, but a larger batch size is preferable if possible.

Next we looked at the effect of using sub-sample size $N_S < N_X$. For convergence we tried at a fixed minibatch size of $N_X = 1000$ with $N_S = 10, 100, 1000$. In fig. 5.10 we see similar convergence rates for $N_S = 1000$ and $N_S = 100$ but poorer convergence for $N_S = 10$. To compare the memory usage we used the sub-sample size $N_S = 100$ with batch sizes $N_X = 1000, 10000$. Table 5.2 and creftable:GPU-minibatch show sub-sampling gave memory consumption much closer to SGD than Inexact Newton without sub-sampling.

Generally, we view these hyper-parameters as being hardware limited. There seems to be no negative consequence of making the batch size and sub-sample size as large as the GPU memory allows, however for larger models and input data sizes there may be some tuning required to find optimal performance based on a trade-off between the two and the memory available but we view it as beyond the scope of this work. As we have ample GPU memory available and see little effect on batch size for our test cases we will use a batch size of 1000 without

sub-sampling ($N_X = N_S = 1000$).

	Inexact Newton				SGD			
	Batch				Batch			
Model	**100**	**1000**	**10000**	**Full**	**100**	**1000**	**10000**	**Full**
AEM	3.10	3.09	3.67	4.37	3.10	3.10	3.56	3.99
AEC	4.73	4.73	6.32	7.84	4.73	4.74	6.12	7.24
L5M	2.89	2.89	3.22	3.83	2.89	2.89	3.21	3.83
R18C	3.85	3.94	5.15	NA	3.79	3.79	4.50	5.59

Table 5.2: Memory usage (Gb) for first and second order methods. NA is used when we couldn't perform this run

	Inexact Newton				SGD			
	Batch				Batch			
Model	**100**	**1000**	**10000**	**Full**	**100**	**1000**	**10000**	**Full**
AEM	3.98	3.98	6.98	20.98	3.98	3.98	3.98	8.98
AEC	5.35	5.35	11.35	30.93	5.35	5.35	5.35	11.35
L5M	4.98	4.98	6.98	18.98	4.98	4.98	6.98	18.98
R18C	11.35	11.35	30.93	OOM	11.35	11.35	11.35	30.93

Table 5.3: GPU memory usage (Gb) for first and second order methods by batch size. OOM is used when we exceeded the available GPU memory

	100 subsamples		No subsampling	
	Batch		Batch	
Model	**1000**	**10000**	**1000**	**10000**
AEM	3.09	3.61	3.09	3.67
AEC	4.73	6.13	4.73	6.32
L5M	2.89	3.23	2.89	3.22
R18C	3.94	4.54	3.94	5.16

Table 5.4: Memory usage (Gb) with and without subsamplingsubsampling

	100 subsamples		No subsampling	
	Batch		Batch	
Model	**1000**	**10000**	**1000**	**10000**
AEM	3.98	3.98	3.98	6.98
AEC	5.35	5.35	5.35	11.35
L5M	4.98	6.98	4.98	6.98
R18C	11.35	11.35	11.35	30.93

Table 5.5: GPU memory usage (Gb) with and without subsampling

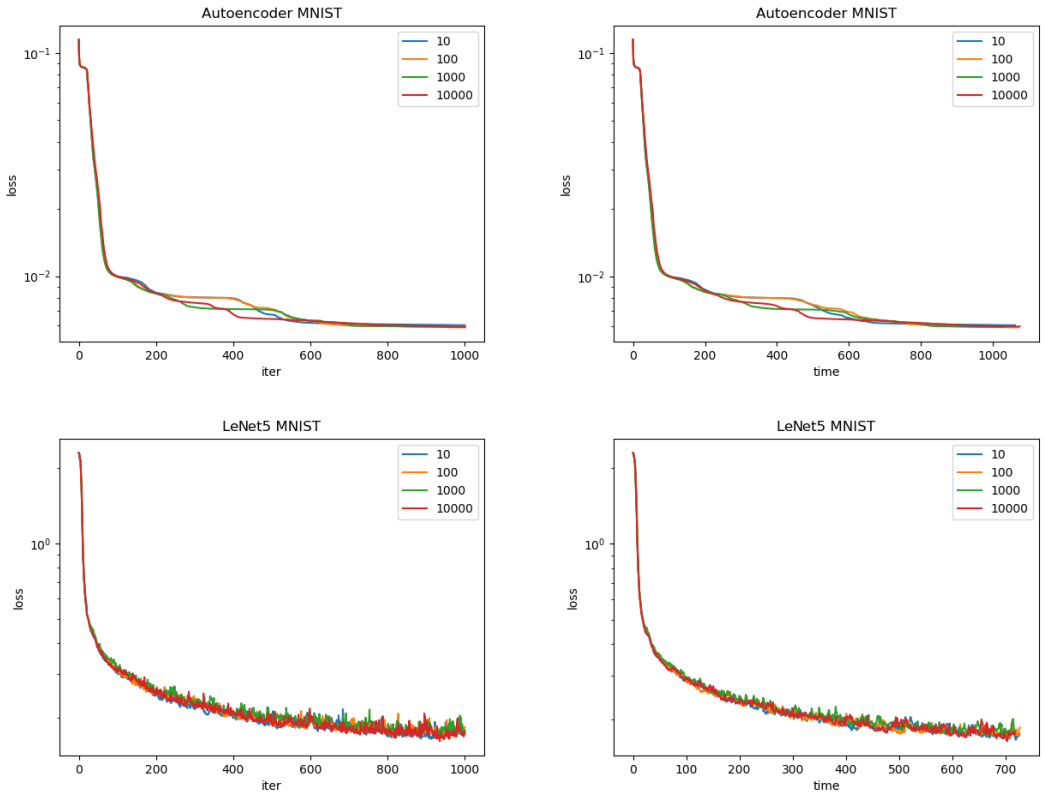

Figure 5.9: Loss by iteration and time for AEM and L5M with different batch sizes

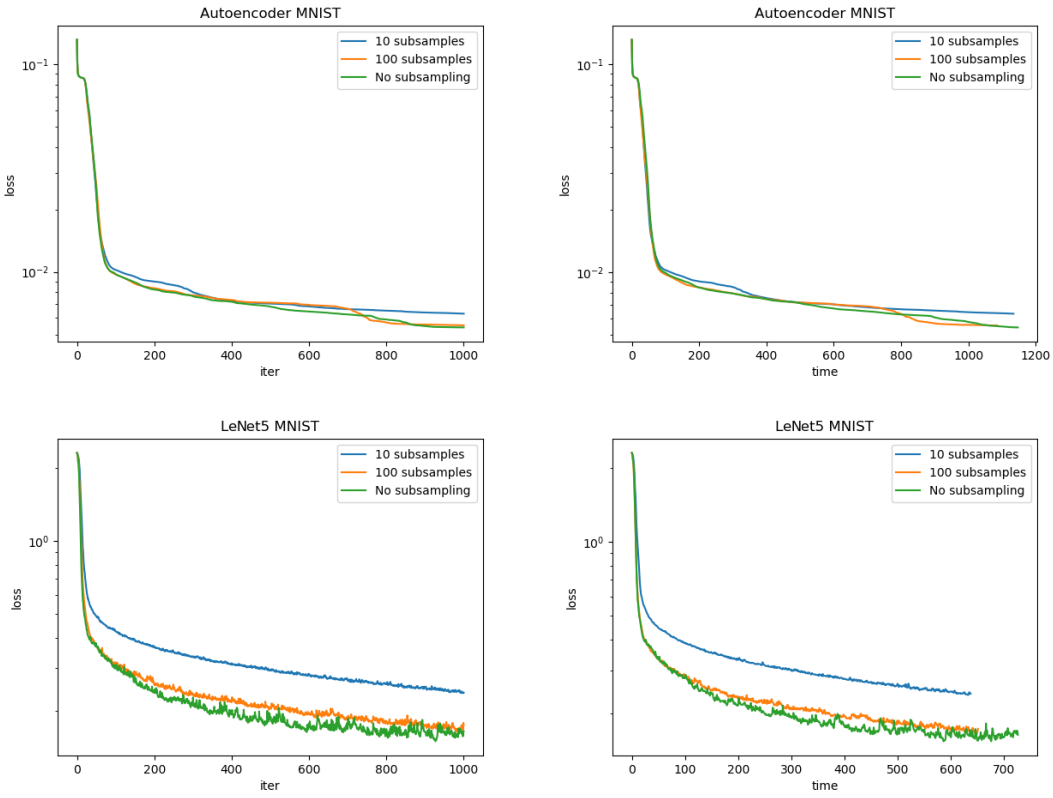

Figure 5.10: Loss by iteration and time for AEM and L5M with different subsample sizes

5.3.5 Trust Region, Step Size Robustness

For first order methods, the learning rate must be tuned for each model, an example of this is in fig. 5.11. On the other hand, our methods choose a step size based on the TR which grows and shrinks automatically. The parameters we use for growing and shrinking the TR are defaults which are widely used in the optimization community. We find that a maximum TR size is also not needed as the TR size doesn't blow up, in all of our tests it stayed within reasonable bounds. One parameter to consider is the TR radius initialization. This has far less effect than the learning rate does for first order methods, but in general we found that smaller initializations gave slightly better results, as seen in fig. 5.12.

In general it seems best to start with a small TR (0.001) and letting it naturally grow as needed.

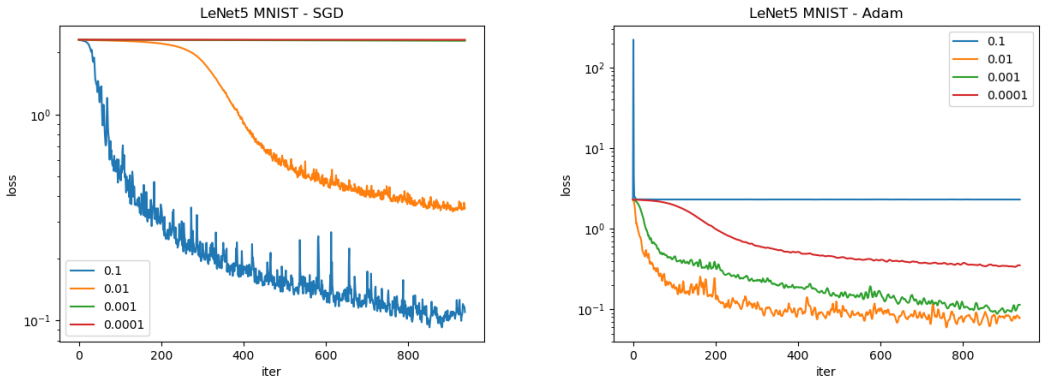

Figure 5.11: Performance of first order methods for different step sizes. This shows the sensitivity of first order methods to step size, something mitigated by trust regions

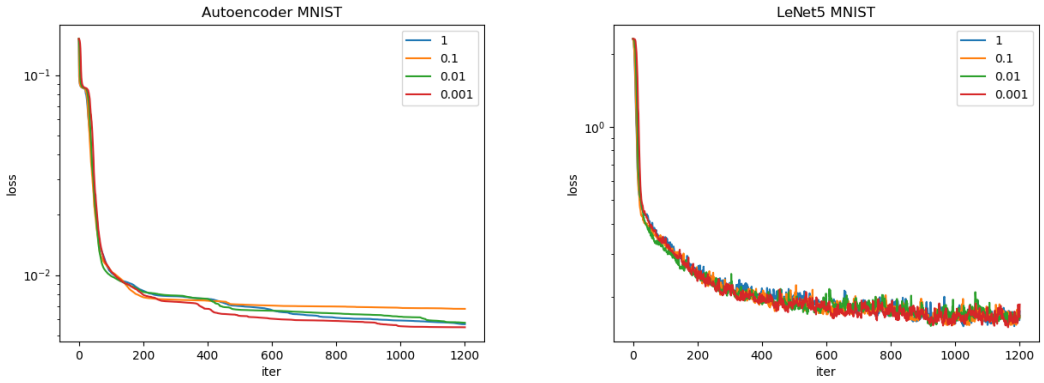

Figure 5.12: The performance of the 2D subspace TR methods for different TR initialization sizes

5.4 Method Comparison

We tested our methods on the four models mentioned above, using the default parameters and configurations for the three proposed TR methods *2D-subspace*, *lowcost 2D-subspace* and *dogleg*, as justified in the previous section. For compar-

ison we also tried the TR *step* method, which uses the Inexact Newton direction at each step, scaled by the TR. We compared these to the SGD and Adam first order methods which were hyperparameter tuned via a grid search for learning rates (0.1, 0.01, 0.001 and 0.0001) and batch sizes (32, 64, 128 and 512). The tuned values are shown in table 5.6. We ran each model for 40000 sweeps of the network which we found was enough to compare the convergence behaviours of the models.

	SGD		**Adam**	
Model	**LR**	**Batch**	**LR**	**Batch**
AEM	0.1	32	0.01	128
AEC	0.01	32	0.01	64
L5M	0.1	512	0.001	512
R18C	0.01	512	0.0001	512

Table 5.6: The tuned hyperparameter values of learning rate (LR) and batch size for first order methods SGD and Adam

5.4.1 Autoencoders

For the MNIST data set the results in table 5.7 show Adam performed the best, closely followed by our methods, out of which the Dogleg method was slightly better. The TR Step method performed similarly to our methods but became unstable after 200 iterations (as seen in fig. 5.14) suggesting it is less robust to the changes in regularization. Figure 5.15 shows that SGD progressed at a much slower rate and was still decreasing when we stopped, making it unclear whether or not it would reduce the loss as much as the other methods. Comparing by iteration we see that the second order methods converge faster. This is very promising as if we are able to reduce the cost per iteration (through a better preconditioner) we could make the method more competitive when compared by sweeps or time.

Optimizer	Min TrL	Min TL	End TL	Iters	Time (s)
TR Step	0.0079	0.0078	0.017	783	799
TR Dogleg	0.0055	0.0054	0.0054	722	807
TR 2D Subspace	0.0067	0.0067	0.0067	731	808
TR LC 2D Subspace	0.0065	0.0065	0.0066	730	811
SGD	0.021	0.024	0.024	20000	434
Adam	0.0048	0.0051	0.0052	20000	630

Table 5.7: Train Loss (TrL) and Test Loss (TL) for AEM

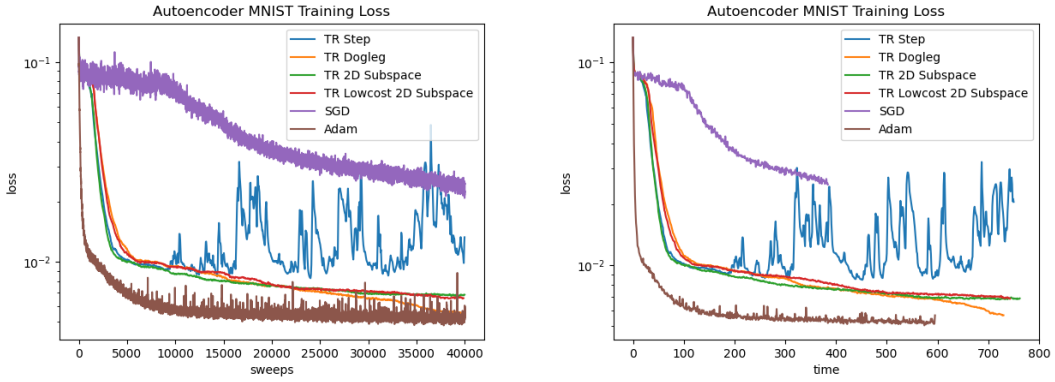

Figure 5.13: AEM training loss by sweeps and time

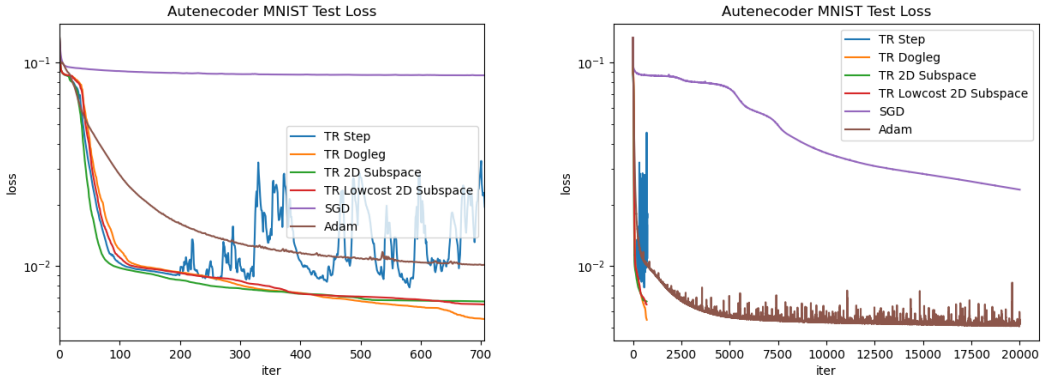

Figure 5.14: Test loss by iteration, up to the minimum (left) and maximum (right) number of iterations completed in 40000 sweeps

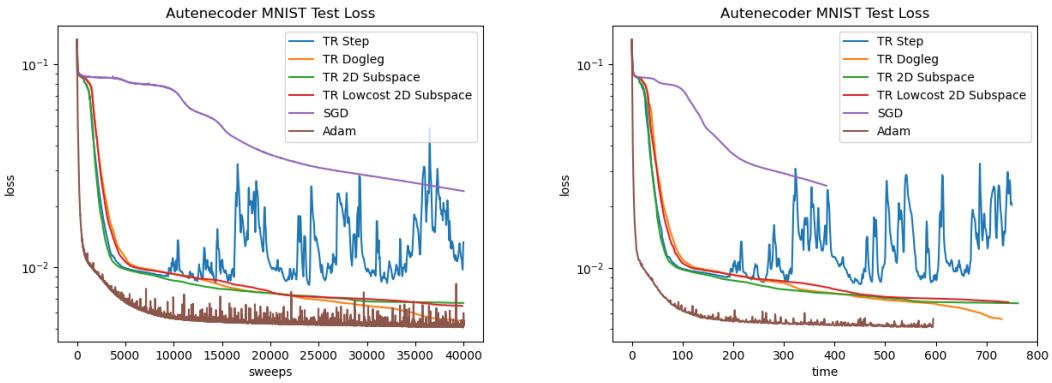

Figure 5.15: AEM test loss by sweeps and time

For the CIFAR-10 data set, table 5.8 shows that Adam performed the best, followed by all of the TR methods which performed more or less identically, although the step method was slightly less smooth. Figure 5.17 shows the TR methods reached a plateau after just 25 iterations or 200 sweeps, suggesting there is a saddle point they were unable to escape. The Adam method also was stuck around this loss of 0.062 for around 100 iterations but managed to escape it eventually. The SGD method was once again the slowest although it gradually reduced the gap with the TR methods, as seen in fig. 5.18. The plots of the first order methods have much more noise than that of the second order methods, we expect this is partly due to the smaller batch sizes giving a higher variance in the loss function at each iteration, but it also demonstrates the stability of our methods.

Optimizer	**Min TrL**	**Min TL**	**End TL**	**Iters**	**Time (s)**
TR Step	0.060	0.062	0.062	1000	1677
TR Dogleg	0.060	0.062	0.062	1046	1546
TR 2D Subspace	0.060	0.062	0.062	925	1688
TR LC 2D Subspace	0.060	0.062	0.062	1039	1528
SGD	0.052	0.065	0.065	20000	683
Adam	0.025	0.028	0.029	20000	1014

Table 5.8: Train Loss (TrL) and Test Loss (TL) for AEC

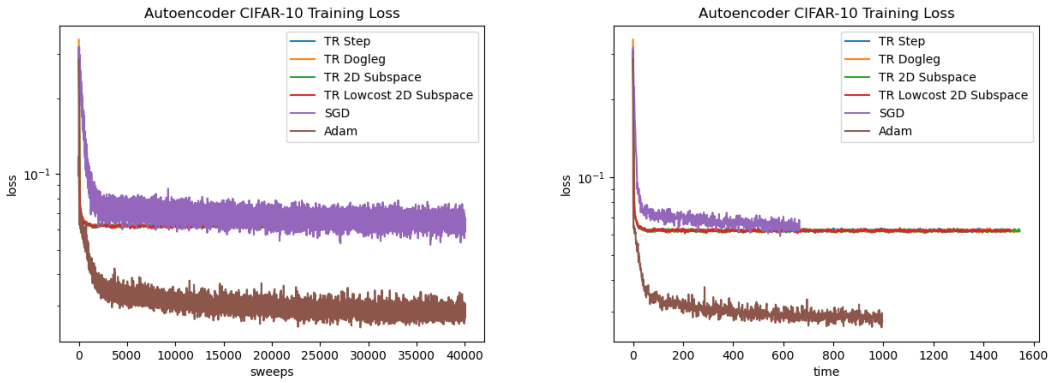

Figure 5.16: AEC training loss by sweeps and time

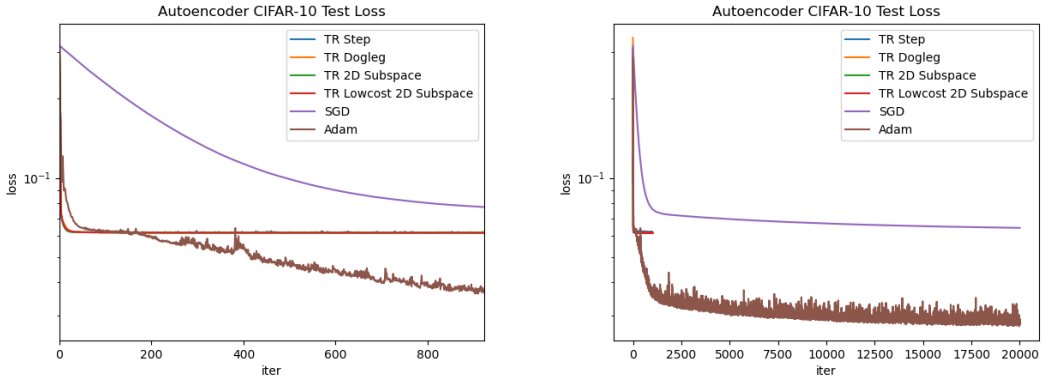

Figure 5.17: Test loss by iteration, up to the minimum (left) and maximum (right) number of iterations completed by an optimizer in 40000 sweeps

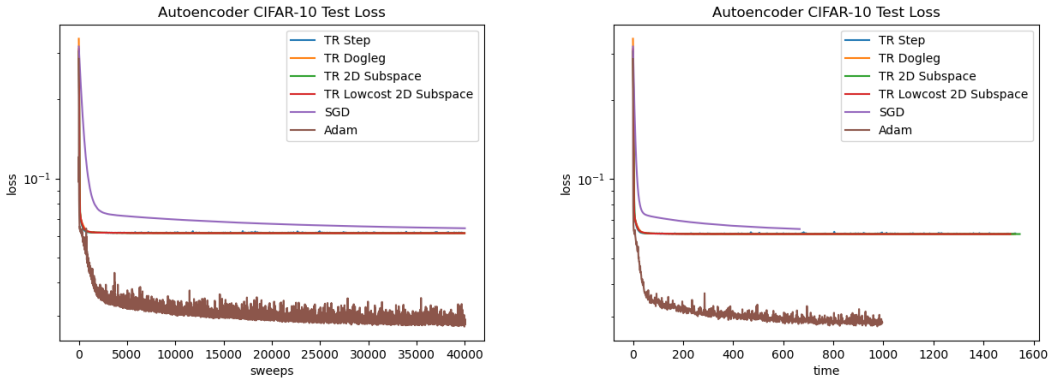

Figure 5.18: AEC test loss by sweeps and time

5.4.2 Image Classification

For the Lenet5 model, the results in table 5.9 show that the Adam optimizer performed the best, closely followed by SGD. For training loss fig. 5.19 shows that first order methods continually reduced the loss, but this led to slight overfitting of the training data as seen in fig. 5.21, something not exhibited by the second order methods. The TR Step method progressed well, almost to the level of the first order methods and didn't overfit, despite some instabilities. Figure 5.22 shows that the other TR methods were unable to reach more than 95% accuracy. On a per iteration basis, fig. 5.20 the second order methods converged faster initially but were quickly overtaken by the first order methods as they plateaued. The TR step method however matched Adam per iteration, although it was not as stable.

Optimizer	**Min TrL**	**Min TL**	**End TL**	**Max TA**	**Iters**	**Time (s)**
TR Step	0.023	0.051	0.058	0.98	1007	747
TR Dogleg	0.11	0.16	0.17	0.95	978	711
TR 2D Subspace	0.11	0.16	0.18	0.95	865	707
TR LC 2D Subspace	0.12	0.16	0.18	0.95	967	706
SGD	5.4×10^{-5}	0.042	0.064	0.99	20000	347
Adam	3.5×10^{-7}	0.031	0.064	0.99	20000	573

Table 5.9: Train Loss (TrL), Test Loss (TL) and Test Accuracy (TA) for L5M

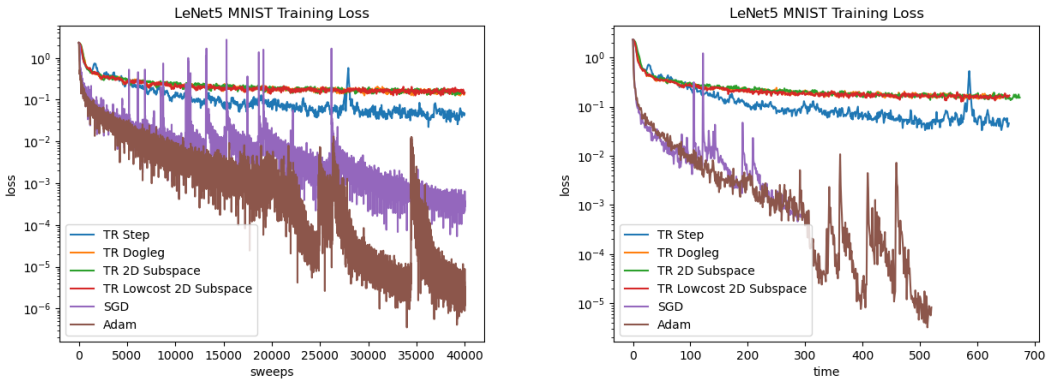

Figure 5.19: L5M training loss by sweeps and time

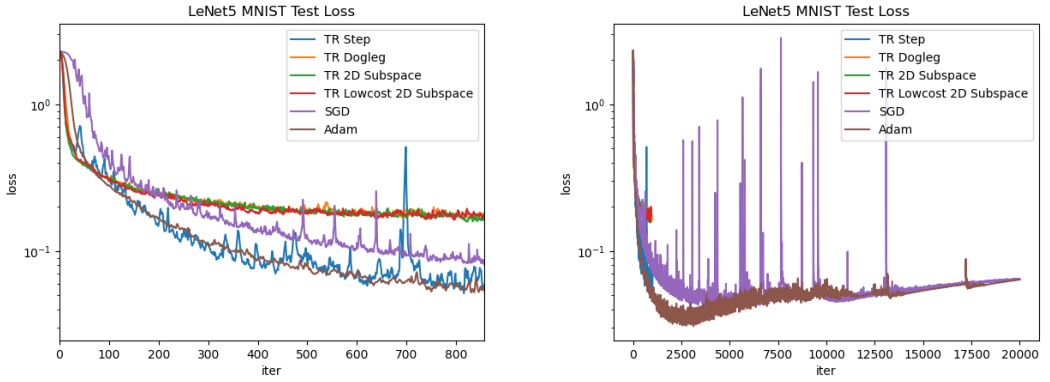

Figure 5.20: L5M test loss by iteration, up to the minimum (left) and maximum (right) number of iterations completed in 40000 sweeps

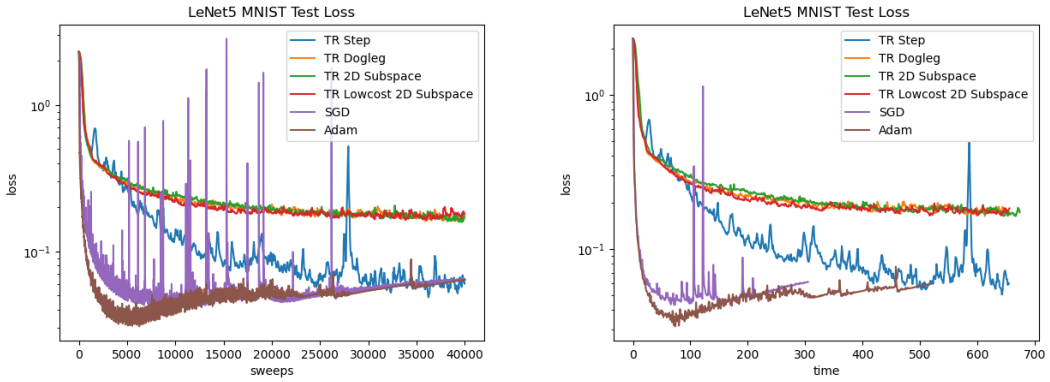

Figure 5.21: L5M test loss by sweeps and time

Optimizer	Min TrL	Min TL	End TL	Max TA	Iters	Time (s)
TR Step	1.53	1.57	1.99	0.46	2060	3425
TR Dogleg	1.42	1.47	1.48	0.48	2500	2354
TR 2D Subspace	1.37	1.42	1.42	0.50	1800	4527
TR LC 2D Subspace	1.41	1.45	1.45	0.49	2840	2786
SGD	2.7×10^{-3}	1.12	2.67	0.65	19698	2049
Adam	9.5×10^{-6}	1.17	6.32	0.61	19698	3887

Table 5.10: Train Loss (TrL), Test Loss (TL) and Test Accuracy (TA) for R18C

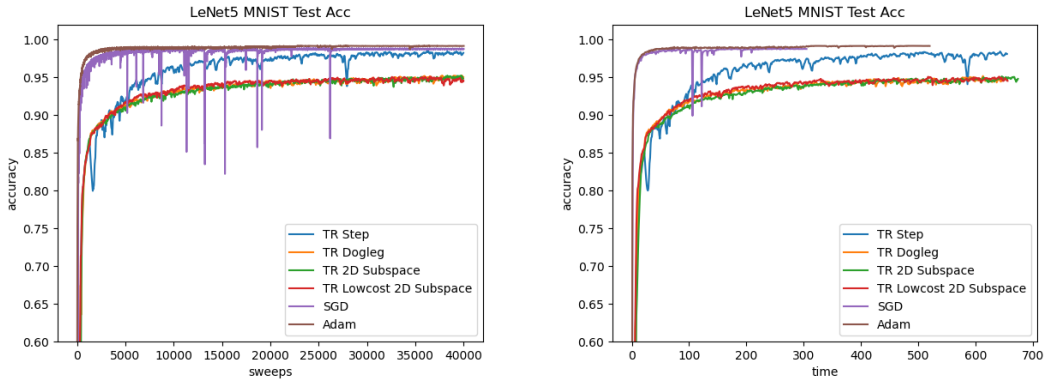

Figure 5.22: L5M test accuracy by sweeps and time

For the ResNet18 model, the results in table 5.10 show that the SGD optimizer performed the best, closely followed by Adam. But in fig. 5.25 we see that the first order methods strongly overfitted the training data, hence the low loss levels seen in fig. 5.25 are not useful. The proposed methods performed similarly, with TR 2D Subspace reaching slightly lower loss values than the others, however the TR Dogleg and Lowcost 2D Subspace methods were nearly twice as fast (seen in fig. 5.25 and fig. 5.26). The TR Step method meanwhile was not stable and did not perform as well.

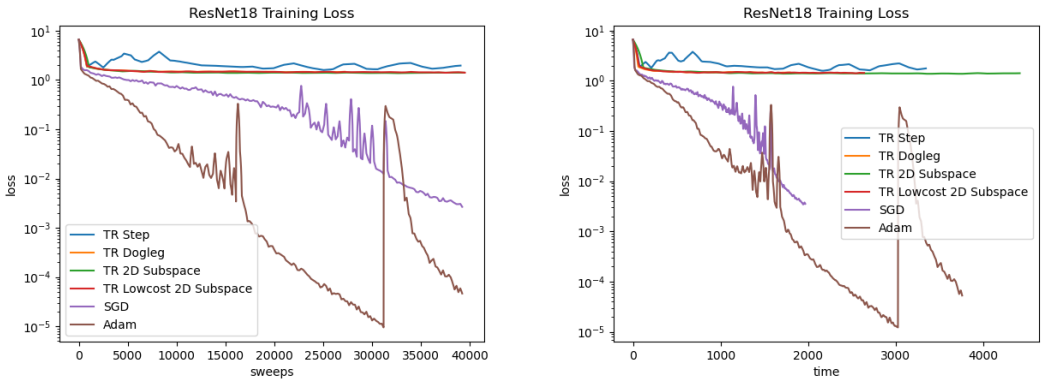

Figure 5.23: R18C training loss by sweeps and time

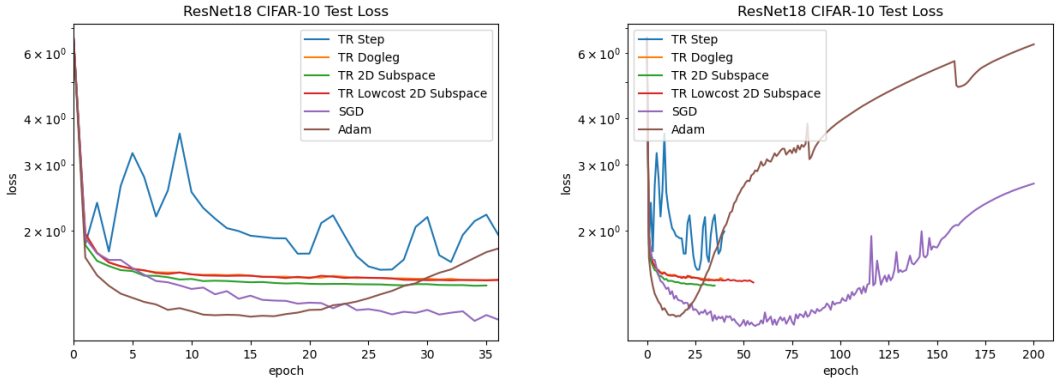

Figure 5.24: R18C test loss by epoch, up to the minimum (left) and maximum (right) number of epochs completed in 40000 sweeps

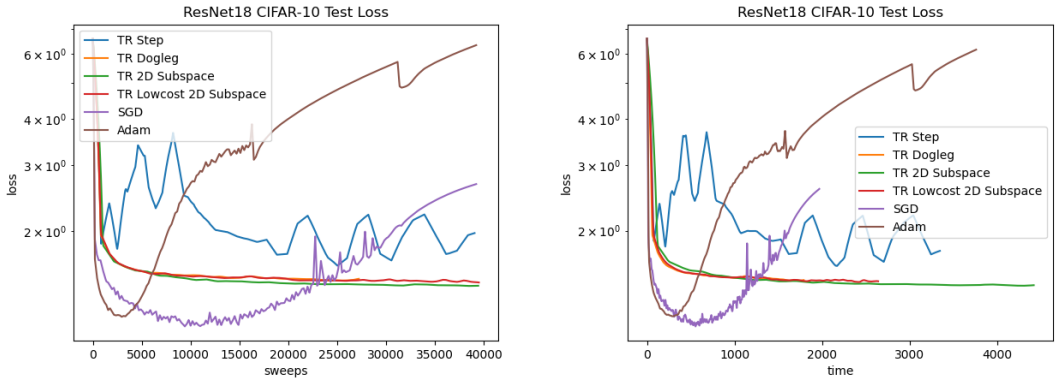

Figure 5.25: R18C test loss by sweeps and time

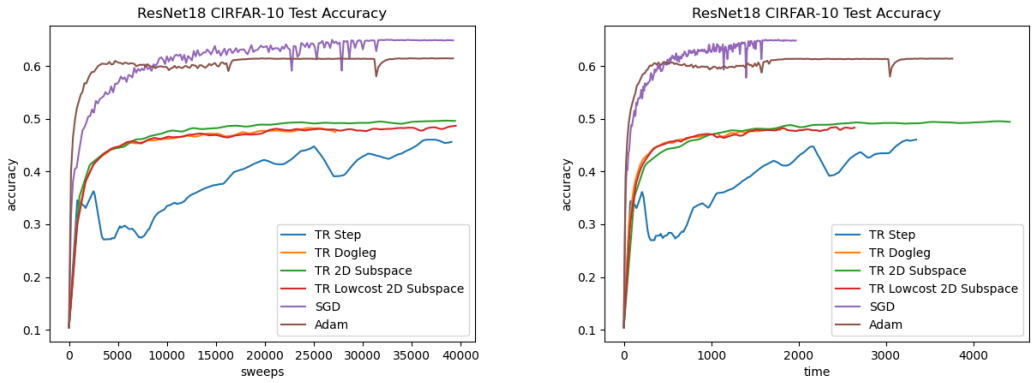

Figure 5.26: R18C test accuracy by sweeps and time

Chapter 6

Conclusion

6.1 Summary

In this thesis we proposed three second order optimization methods for training neural networks based on using the Inexact Newton Method with a Trust Region. We have analysed the effects of several additions to this method which either reduce the cost of the method, improve the convergence or automatically tune the hyperparameters the method introduces. We then compared our methods against two of the most commonly used first order methods on autoencoder and image classifiers. The performance was in some cases better than SGD and in some cases comparable to Adam despite no hyperparameter tuning being applied to our methods. This demonstrates that our methods produce robust results in a feasible time without the need for hyperparameter tuning. The three methods themselves perform similarly in terms of convergence on a per iteration basis. However we begin to see a difference in runtime as the number of model variables increase, showing the benefit of the low-cost implementation. In almost all tests they outperformed the basic TR Step method in both convergence and stability

and they demonstrated no overfitting.

6.2 Future Work

Whilst we tried to be as thorough as possible in this thesis, there were several important topics that we felt were beyond the scope of this work. We would like to expand the testing to other types of models such as larger convolutional neural networks and recurrent neural networks. It would also be interesting to compare our results to other second order methods. We would like to investigate the effect of different preconditioners on the number of CG iterations and convergence rate. Whilst we have performed some eigenvalue analysis to increase our understanding of the loss function landscape, we believe a detailed study of the form of the Hessian could further this and guide preconditioner design for different models. We would also be interested in comparing other trust region methods, in particular the Steighaug-Toint method, which solves the subproblem iteratively in a comparable way to CG. We would like to also try implementing our methods for sparse networks. Lastly, we would like to see the performance difference between implementing them directly in CUDA as opposed to using the higher level TensorFlow operations.

Appendix A

Abbreviations

AEC	Autoencoder with CIFAR-10 dataset
AEM	Autoencoder with MNIST dataset
CG	Conjugate Gradient
vCPU	Virtual Central Processing Unit
EW	Eisenstat-Walker
GPU	Graphical Processing Unit
L5M	LeNet5 with MNIST dataset
LC	Low-cost
R18C	ResNet18 with CIFAR-10 dataset
RAM	Random Access Memory
SGD	Stochastic Gradient Descent
TR	Trust Region

Bibliography

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. The MIT Press, 2016.

[2] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Learning Representations by Back-Propagating Errors*, p. 696–699. Cambridge, MA, USA: MIT Press, 1988.

[3] Y. Nesterov, "A method for solving the convex programming problem with convergence rate $o(\frac{1}{k^2})$," 1983.

[4] T. Tieleman and G. Hinton, "Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude." COURSERA: Neural Networks for Machine Learning, 2012.

[5] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *The Journal of Machine Learning*, vol. 12, pp. 2121–2159, 2011.

[6] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *International Conference on Learning Representations*, 12 2014.

[7] S. Becker and Y. Lecun, "Improving the convergence of back-propagation learning with second-order methods," 01 1989.

[8] B. Pearlmutter, "Fast exact multiplication by the hessian," *Neural Computation*, vol. 6, 02 1994.

[9] J. Martens, "Deep learning via hessian-free optimization," in *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, (Madison, WI, USA), p. 735–742, Omnipress, 2010.

[10] L. Bottou, F. E. Curtis, and J. Nocedal, "Optimization methods for large-scale machine learning," *ArXiv*, vol. abs/1606.04838, 2018.

[11] T. O'Leary-Roseberry, N. Alger, and O. Ghattas, "Inexact newton methods for stochastic nonconvex optimization with applications to neural network training," 2019.

[12] P. Xu, F. Roosta-Khorasani, and M. W. Mahoney, "Second-order optimization for non-convex machine learning: An empirical study," 2017.

[13] R. Bollapragada, R. Byrd, and J. Nocedal, "Exact and inexact subsampled newton methods for optimization," 2016.

[14] F. Roosta-Khorasani and M. W. Mahoney, "Sub-sampled newton methods i: Globally convergent algorithms," 2016.

[15] F. Roosta-Khorasani and M. W. Mahoney, "Sub-sampled newton methods ii: Local convergence rates," 2016.

[16] P. Chen and C.-J. Hsieh, "A comparison of second-order methods for deep convolutional neural networks," 2018.

[17] J. Martens and R. Grosse, "Optimizing neural networks with kronecker-factored approximate curvature," 2015.

[18] K. Osawa, Y. Tsuji, Y. Ueno, A. Naruse, R. Yokota, and S. Matsuoka, "Large-scale distributed second-order optimization using kronecker-factored approximate curvature for deep convolutional neural networks," 2018.

[19] R. Bollapragada, D. Mudigere, J. Nocedal, H.-J. Shi, and P. Tang, "A progressive batching l-bfgs method for machine learning," 02 2018.

[20] N. S. Keskar and A. S. Berahas, "adaqn: An adaptive quasi-newton algorithm for training rnns," 2015.

[21] N. Agarwal, B. Bullins, and E. Hazan, "Second-order stochastic optimization for machine learning in linear time," *J. Mach. Learn. Res.*, vol. 18, p. 4148–4187, Jan. 2017.

[22] J. F. Henriques, S. Ehrhardt, S. Albanie, and A. Vedaldi, "Small steps and giant leaps: Minimal newton solvers for deep learning," 2018.

[23] N. Schraudolph, "Fast curvature matrix-vector products for second-order gradient descent," *Neural computation*, vol. 14, pp. 1723–38, 08 2002.

[24] J. Nocedal and S. J. Wright, *Numerical Optimization*. New York, NY, USA: Springer, second ed., 2006.

[25] S. C. Eisenstat and H. F. Walker, "Choosing the forcing terms in an inexact newton method," *SIAM Journal on Scientific Computing*, vol. 17, no. 1, pp. 16–32, 1996.

[26] E. Mizutani and S. E. Dreyfus, "Second-order stagewise backpropagation for hessian-matrix analyses and investigation of negative curvature," *Neural Networks*, vol. 21, no. 2-3, pp. 193–203, 2008.

[27] H. Robbins and S. Monro, "A Stochastic Approximation Method," *Annals of Mathematical Statistics*, vol. 22, pp. 400–407, Sep 1951.

[28] J. Duchi, "Lecture 09 - Chapter 8: Fisher Information." Stanford: Lecture Notes for Statistics 311 / Electrical Engineering 377, 2014.

[29] O. Chapelle and D. Erhan, "Improved preconditioner for hessian free optimization," in *In NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, 2011.

[30] T. N. Sainath, L. Horesh, B. Kingsbury, A. Y. Aravkin, and B. Ramabhadran, "Accelerating hessian-free optimization for deep neural networks by implicit preconditioning and sampling," 2013.

[31] A. R. Conn, N. I. M. Gould, and P. L. Toint, *Trust Region Methods*. Society for Industrial and Applied Mathematics, 2000.

[32] R. P. Pawlowski, J. P. Simonis, H. F. Walker, and J. N. Shadid, "Inexact newton dogleg methods," *SIAM Journal on Numerical Analysis*, vol. 46, no. 4, pp. 2112–2132, 2008.

[33] N. Mayorov, "2D Subspace Trust-Region Method," 2015.

[34] J. W. Demmel, *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, 1997.

[35] X. He, D. Mudigere, M. Smelyanskiy, and M. Takác, "Large scale distributed hessian-free optimization for deep neural network," *CoRR*, vol. abs/1606.00511, 2016.

[36] A. Olivares, J. Moguerza, and F. Prieto, "Nonconvex optimization using negative curvature within a modified linesearch," *European Journal of Operational Research*, vol. 189, pp. 706–722, 09 2008.

[37] Y. N. Dauphin, R. Pascanu, Ç. Gülçehre, K. Cho, S. Ganguli, and Y. Bengio, "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization," *CoRR*, vol. abs/1406.2572, 2014.

[38] P. Xu, F. Roosta, and M. W. Mahoney, "Newton-type methods for non-convex optimization under inexact hessian information," 2017.

[39] Z. Wang, Y. Zhou, Y. Liang, and G. Lan, "Cubic regularization with momentum for nonconvex optimization," 2018.

[40] Y. LeCun, C. Cortes, and C. Burges, "Mnist handwritten digit database," *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, vol. 2, 2010.

[41] A. Krizhevsky, "Learning multiple layers of features from tiny images," tech. rep., 2009.

[42] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[43] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015.