
# SJÄLVSTÄNDIGA ARBETEN I MATEMATIK

MATEMATISKA INSTITUTIONEN, STOCKHOLMS UNIVERSITET


## Quasi-Newtonian Optimisation for Deep Neural Networks


av

**Peter Brohan**



2020 - No M8

# Quasi-Newtonian Optimisation for Deep Neural Networks

Peter Brohan

---

Självständigt arbete i matematik 30 högskolepoäng, avancerad nivå

Handledare: Yishao Zhou

2020

# Abstract

In his 2010 paper, '*Deep learning via hessian-free optimization*', Martens suggested techniques for the application of Quasi-Newtonian optimisation methods as a Neural Network learning algorithm. Here we examine the background of the paper, beginning with the structure and function of a neural network itself. We move on to consider a number of current popular alternative learning algorithms, considering their variety of approaches in approximating the optimisation problem in a tractable manner.

We then move on to look at Quasi-Newtonian methods in general, examining the Gauss-Newton, Levenberg-Marquardt and Truncated Newtonian approximations, each of which allows us make some approximation of the curvature of a given function, and to find approximate optimal solutions more practically than using the full Newton's Method.

We then consider the application of these methods to Neural Networks themselves, discuss further adaptations and run some small experiments to allow us some comparison of the Quasi-Newtonian approach to those methods in popular use today.

# Contents

# Abbreviations

| | |
|---|---|
| $\lceil \cdot \rceil$ | Ceiling Function |
| $\lfloor \cdot \rfloor$ | Floor Function |
| $\odot$ | Hadamard Product (componentwise matrix multiplication) |
| **CG** | Conjugate Gradient |
| **CNN** | Convolutional Neural Network |
| **FC** | Fully Connected layer of a Neural Network |
| **GD** | Gradient Descent |
| **SGD** | Stochastic Gradient Descent (although generally referring to Mini-Batch Gradient Descent) |

# Acknowledgements

I would like to acknowledge my supervisor, Yishao Zhou, for pointing me in the right direction and providing reassurance when I worried I was wandering off-course and my referee, Sven Raum, for his helpful comments and suggestions. Lastly I would like to thank the many people who have listened to me complain during the process, and all those who at least pretended to be interested while I talked endlessly about machine learning.

# 1. Introduction

## 1.1 A Brief Introduction to Deep Neural Networks

Deep neural networks, one of modern artificial intelligence research's largest breakthroughs, combine our mathematical understanding of function approximation and our biological understanding of the brain. They are the basis of breakthroughs in gameplay AI (for example Google's AlphaZero and AlphaStar), image recognition (Facebook's DeepFace project), speech recognition and natural language processing (Amazon's Alexa, Apple's Siri), and signal processing to name but a few examples. They have been shown to be flexible and powerful tools for a number of tasks previously difficult for artificial intelligences and their power and simplicity of operation has caused their widespread adoption throughout the space.

Neural networks are an increasing part of the landscape of machine learning. As with all machine learning methods, they attempt to build an increasingly accurate mathematical model of some function only through interactions with the inputs and, for supervised networks, the ideal outputs of the function. The models themselves are loosely based on the human brain, a series of interconnected nodes which each receive some input, minimally process it and then pass on some output, the idea being that the composition of these small calculations will result in some greater meaning.

Most simply, a neural network assumes that the target behaviour can be well modelled by a Continuous Piecewise Linear function [1], i.e. that there is some large collection of linear functions, each of which operates over only a small part of the domain, which returns results sufficiently similar to the target behaviour. This is not all that poor an assumption. Take the domain of 50x50px black and white images for example, and define the 'distance' between two images as the number of pixels which are different. If we are attempting to classify which images contain cats, it is likely that if one particular picture contains a cat, the images 'close by' will also contain cats to a high degree of certainty. Similarly if a particular image does not contain a cat, it is very unlikely that those around it will. Here there are likely to be generalisable classifiable areas. Naively we might think to take a random sampling of the images and classify all areas within a certain proximity of a classified image to be the same category (the k-nearest neighbours algorithm) however this quickly proves to be impossible. There are simply too many images, and the frequency of cats among them is far too small. Instead we hope that a neural network will be able to extrapolate some feature information from a given dataset, and use this to produce an appropriate model. More generally, neural networks attempt to give some sufficiently simple approximation of a complex function, with a small enough error for practical purposes.

### 1.1.1 Types of Neural Network

There are a huge number of types of neural network (a large number are listed at the Asimov Institute's "Neural Network Zoo"[1]), and even among these there are innumerable subtle differences between networks of the same category. Below I discuss two simpler networks, the fully-connected feedforward network and the convolutional network. These share the characteristic that each node passes information in the same direction, from the input towards the output, that the output of each node is determined entirely by the outputs of nodes closer to the input, and that given a single input, the output is uniquely defined. This makes the networks simpler to study, but they by no means make up all networks. Networks can be stateful, include recurrences to feed back information to earlier nodes or include random factors in their outputs. Each of these can have its own use in our attempt to model complex real-life functions.

We can also separate networks by their training into **supervised** networks, those which are trained using a set of examples to match a certain form of input to a certain output (e.g. classification networks or function approximators), and **unsupervised** networks, those which attempt to find some pattern in a data set without external input (e.g. component or cluster analysis).

## 1.2 Fully Connected Feedforward Neural Networks

### 1.2.1 Perceptrons

The simplest neural network is a single layer perceptron [2], which takes in a vector $x \in \mathbb{R}^n$ and outputs a classifier $y$.

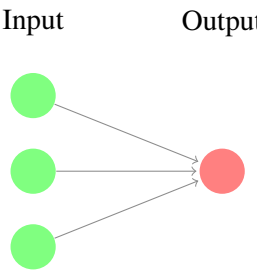


**Figure 1.1:** A single layer perceptron

We can interpret this as a weighted sum of the input values with some added bias term $b$. In vector notation we have that

$$y = x^T w + b \tag{1.1}$$

where $w$ is an $n \times 1$ matrix of weights. By using some learning algorithm to update the weights based on input data, a perceptron functions as a binary linear classifier, i.e. it is capable of classifying data into two sets if those sets are separated by a linear function using the classifier

$$z = \begin{cases} 0 & y > a \\ 1 & \text{otherwise} \end{cases} \tag{1.2}$$

[1] https://www.asimovinstitute.org/neural-network-zoo/

for some hyperperameter $a$.[1]

### 1.2.2 Multi-Layered Neural Networks

If our data does not admit linear classification, or we would like to gain some other information, we will need to add further complexity to the network. We can do this in two ways.

We can add further 'hidden' layers to the network (named as they are not directly accessible to a user of the network). This will hopefully allow us to better emphasise certain features of the data, e.g

$$y = x^T W_0 + b_0^T \tag{1.3}$$
$$z = yW_1 + b_1^T \tag{1.4}$$

where $W_0$ and $W_1$ are matrices of size $n \times m$ and $m \times l$ respectively and $b_0$ and $b_1$ vectors of length $m$ and $l$. However, being merely a product of vectors, this will also give only a linear function, and thus by simple matrix multiplication, it would be possible to build a network without this hidden layer which gained the same result. We must therefore also add an **activation function** to each layer in order to gain a non-linear output. The choice of activation function is a balance between adding sufficient (and appropriate) non-linearity in order that we gain sufficient complexity to appropriately model our target function, and allowing for efficient calculation. In general we choose a relatively simple, easily differentiable function, such as ReLU [3, p. 171], $ReLU(x) = \max(0,x)$, or the sigmoid function $S(x) = \frac{1}{1+e^{-x}}$, which depend only on hyperparameters of the network, and none of the internal weights. We can thus modify (1.3) so that our network consists of

$$y = a_{(0)}(x^T W_0 + b_0) \tag{1.5}$$
$$z = yW_1 + b_1 \tag{1.6}$$

for some activation function $a_{(0)}$.

Here every node of the network is connected to every node in the preceding and following layers, and every layer passes all of its information directly to the next layer, hence its name, the fully connected feedforward network. We can augment the network with further layers and nodes to increase the accuracy of the result, (although large increases to the number of nodes can lead to overfitting, and increases to the number of layers leads to overly complex computation, see [4]).

## 1.3 Convolutional Neural Networks

Convolutional neural networks (CNNs) allow us to tune networks to consider translation invariant features of the input, i.e. important aspects of the data which may appear at any part

[1] This is a form of classification by dimensional reduction. We attempt to find a hyperplane $x^T + b - a = 0$ which divides the space into the required pair of subsets. Unlike in Principl Component Analysis however, in which we identify the data's principal eigenvector, here we attempt to find a separator which corresponds to some other arbitrary property of the data.

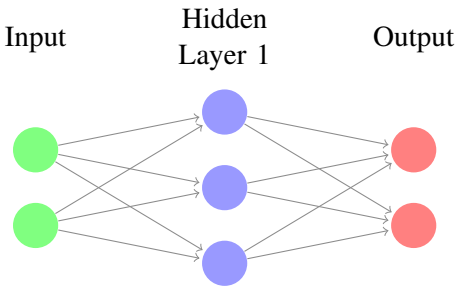


**Figure 1.2:** A fully connected feedforward network with one hidden layer

of the input, without the large number of neurons or layers that would be required in a fully connected layer. The archetypal CNN is made up of a number of convolutional layers (which can be further separated into padding layers, filter layers and pooling layers) together with some fully connected layers (as described above). CNNs are often used for image analysis, and so their components are frequently described as operating on a set of 2d 'layers' (confusingly, a different type of layer than those in the network)

### 1.3.1 Filters

To detect a particular feature in an input, we need only store the particular weights required for detecting that feature, rather than the weights for an entire layer. We therefore try to train only a relatively small number of neurons to detect some specific feature, and then 'step' these across the entire input in order to test for its existence at each location. This gives a single output for each set of inputs to the filter.

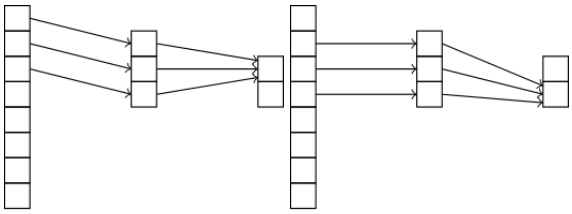

**Figure 1.3:** Two consecutive applications of a filter to a simple one dimensional input with stride 1

We can consider these filters as a set of matrices (or vectors in one dimension) $\mathcal{F}_\alpha$ which we apply to the input data. Each filter acts only on a subset of the input data the same size as the filter, and so if the filter $F$ is a vector of length $l$ we can calculate the single output of the filter later generated by the subset of the input data $x_*$:

$$y_* = \sum_{j=1}^{l} (F \odot x_*)_j \tag{1.7}$$

where $\odot$ is the Hadamard (componentwise) product. These subsets $x_*$ are generally chosen to be consecutive contiguous subsets (e.g. $y_1 = \sum(F \odot (x_{(1)}, x_{(2)}, x_{(3)}), y_2 = \sum(F \odot (x_{(2)}, x_{(3)}, x_{(4)}))\ldots$)

however neither of these are requirements. Consecutive subsets are often separated by some 'stride' $s$, (e.g. $x_{*_1} = (x_{(1)}, x_{(2)}, x_{(3)})$, $x_{*_2} = (x_{(3)}, x_{(4)}, x_{(5)})$) and members of the subsets can be separated by some constant (e.g. $x_* = (x_{(1)}, x_{(3)}, x_{(5)})$) (referred to as dilation).

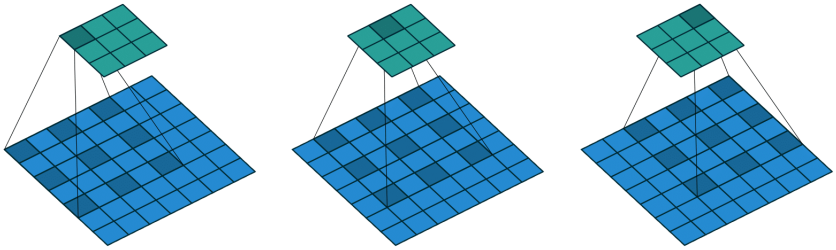

**Figure 1.4:** Three consecutive applications of a 2d filter with stride 1 and dilation 2 (images from [5])

In two dimensions the situation is similar as the sum of the Hadamard product of two matrices is functionally equivalent to the dot product of two vectors containing the same values. Following [6], we can define

$$vec(M) = \begin{bmatrix} M_{:,1} \\ \vdots \\ M_{:,b} \end{bmatrix} \in R^{ab \times 1}, \text{ where } M \in R^{a \times b} \tag{1.8}$$

$$mat(v)_{a \times b} = \begin{bmatrix} v_1 & & v_{(b-1)a+1} \\ \vdots & \dots & \vdots \\ v_a & & v_{ba} \end{bmatrix} \in R^{a \times b}, \text{ where } \overrightarrow{v} \in R^{ab \times 1} \tag{1.9}$$

We could thus consider a filter on a matrix $M$ as being a filter on $vec(M)$, with appropriate step and dilation to keep the sub-matrices as desired.

For inputs with more than a single layer (say $l$ layers), each filter consists of a tensor (a multi-dimensional array of matrices) of $\lambda$ matrices. We apply each filter to its corresponding layer in the input and sum the results to give an output, i.e. for a filter $\mathcal{F} = \left[F_1 \dots F_\lambda\right]$ and input $x_* = \left[x_{*_1} \dots x_{*_\lambda}\right]$

$$y_* = \sum_{i=1}^{\lambda} \sum_{j=1}^{l} (F_i \odot x_{*_i})_j \tag{1.10}$$

As with fully connected layers, after the convolution step the resulting matrix is generally added to a bias matrix and a component-wise activation function is applied (for the reasons given above).

### 1.3.2 Padding

Not all sizes of filter and stride lengths are appropriate for all inputs however. Take for example a two-dimensional input of size (3 x 3). A filter of size (2 x 2) with step of 2 will cover only

the upper left corner of the input; the first step will move the filter out of the range of the input. However, this may still be an appropriate filter and step size for the input at hand. To combat this issue, we can add padding to the edges of the input, increasing its size with neutral values so that the filter correctly steps across the domain.

A filter will only be able to completely cover an input (i.e. provide an output where every element of the input affects some part of the output) if either trivially the filter is the same size as the input (in which case this is merely a fully connected layer with a single output), or both the height and width of the input are divisible by the stride length. However, by extending the input, adding zeroes to the beginning and end of each column and row, we can create a larger matrix of which at least the submatrix containing our original input will be covered by the filter.

We can use padding to control the size of the output. With an input of size $n \times m$, using a filter of size $a \times b$ and stride $s$, the output will be of size $\left\lfloor \frac{n-(a-1)}{s} \right\rfloor \times \left\lfloor \frac{m-(b-1)}{s} \right\rfloor$. By adding padding of size $p$ we can increase this to $\left\lfloor \frac{n+p-(a-1)}{s} \right\rfloor \times \left\lfloor \frac{m+p-(b-1)}{s} \right\rfloor$.

We can also address the issue that the output is biased against values close to the edges of the input. With no padding, the first input contributes only to one output, whereas other values can contribute to up to $\left\lceil \frac{a}{s} \right\rceil \cdot \left\lceil \frac{b}{s} \right\rceil$ outputs. By increasing the padding around the input, we can increase the number of outputs taking some input from the edges. The smallest padding in which each input contributes to the same number of outputs (padding of size $max(a-1, b-1)$) is sometimes called full convolution[3, p.p. 344].

For a matrix $M$ of size $m \times n$, the padding operation can be defined recursively as

$$\mathrm{Pad}_{1,n,m}(M) = \begin{bmatrix} 0 & \cdots & 0 \\ \vdots & M & \vdots \\ 0 & \cdots & 0 \end{bmatrix}$$

$$\mathrm{Pad}_{t,n,m}(M) = \mathrm{Pad}_{1,n+t-1,m+t-1}(\mathrm{Pad}_{t-1,n,m}(M)) \tag{1.11}$$

For simplicity we can also define the padding operation as a matrix multiplication. For a matrix $M$ of size $m \times n$ we can define:

$$\mathrm{Pad}_{1,n,m}(M) = \begin{bmatrix} 0 & \cdots & 0 \\ & I_n & \\ 0 & \cdots & 0 \end{bmatrix} M \begin{bmatrix} 0 & & 0 \\ \vdots & I_m & \vdots \\ 0 & & 0 \end{bmatrix}$$

where $I_n$ is the $n \times n$ identity matrix. We can thus recursively define the padding operation as matrix multiplication in a similar manner to (1.11)

### 1.3.3 Pooling

In most cases, filters traverse the same inputs several times, and this is likely to cause redundancy in the output. A feature present in one area is likely also to be registered as present in a largely overlapping adjacent area. It is thus often practical to take a general overview of each small section of the output, registering only the presence of the feature, rather than its exact location (for further details see [3, §9.3]).

This is generally accomplished by either **max** or **average** pooling. Both function similarly to a filter[1], however max pooling returns only the maximal value of the input, while average pooling returns the mean of the input values. Here we set the stride equal to the size of the pool (usually $2 \times 2$), thus reducing the size of the output layers.

$$\begin{bmatrix} 39 & 33 & 41 & 33 \\ 46 & 26 & 32 & 22 \\ 2 & 30 & 14 & 2 \\ 35 & 4 & 28 & 8 \end{bmatrix} \longrightarrow \begin{bmatrix} 46 & 41 \\ 35 & 28 \end{bmatrix} \quad \begin{bmatrix} 39 & 33 & 41 & 33 \\ 46 & 26 & 32 & 22 \\ 2 & 30 & 14 & 2 \\ 35 & 4 & 28 & 8 \end{bmatrix} \longrightarrow \begin{bmatrix} 36 & 32 \\ 17.75 & 13 \end{bmatrix}$$

**Figure 1.5:** Max pooling and Average pooling of an input

A convolutional layer generally consists of a padding step ($P_i$), a filter step ($F_i$), an activation function ($a_i$) and a pooling step $\Phi_i$. We can view it as the composition of functions

$$Conv_i(x_i) = \Phi_i(a_i(F_i(P_i(x_i)))) \tag{1.12}$$

## 1.4 Optimisation Methods

### 1.4.1 The Optimisation Problem

As above, we can view any neural network as a composition of functions of the inputs $f(x)$. Once the network is trained, we can see this output as the prediction or categorisation of the input (depending on the purpose of the network), however in order to train the network instead we must consider two further aspects of the network, namely the loss function $\mathcal{L}(x,y)$ and the weights $W$.

We can consider the eventual output of the network to be $\mathcal{L}(y, f(x,W))$ (the loss given input $x$, ideal output $y$ and weights $W$). Our aim is, given a function $f$ (a network), a loss function $\mathcal{L}$, domain $X$ with cardinality |X|, and test set $S \subset X$ , find $W^*$ such that $\frac{1}{|X|}\sum_{x,y\in X}\mathcal{L}(y, f(x,W^*))$ is minimised, (i.e. find the ideal set of network weights).

Finding a true minimal point presents a difficult challenge, and we are likely to have to settle for a solution in which the loss function is almost minimised. We must make a choice about what our definition of 'almost' minimised means. Here we opt to have that the average loss is minimised, but we could instead require that the maximal loss is minimised for example. Such choices often depend on the purpose of the network in question.

Outside the theoretical realm, we are also presented with a further difficulty. The data on which the network will be trained and that on which it will operate are most often not the same. We thus need to take into account that while there may exist a solution which performs equally well for both the training and production data, it is likely that a solution found using only the training data will perform worse on any other data. In fact a naïve solution (although not generally optimal in terms of space or results) is to memorise all of the solutions for the test set and

[1] indeed average pooling is an example of a simple filter with stride equal to its width and a single repeated weight dependent only on the dimensions of the filter

randomly guess any other input. We therefore generally aim to find not a completely optimal solution, but one that is almost optimal during training, but also generalises well.

### 1.4.2 Backpropogation

Backpropogation is the application of the chain rule to the neural network function, and can be used to train supervised networks. For each piece of learning data provided to the network $x$, we also have an ideal output $\hat{y}$. Ideally we would like to be able to find some change in weights $\delta\theta$ that will decrease the loss function. However the neural network function is large and complex, and so it is likely impractical to calculate the gradient of the whole function in one pass.

However, the chain rule tells us that if each of the functions composed to make the network $N$ is differentiable [1], we can instead merely differentiate each of these individual functions in turn and use the result of the previous layer in the next calculation (for further details see §2.2). We begin by differentiating the final layer, and continuing back through the function, hence propagating backwards.

### 1.4.3 Genetic Evolution

It may be impractical or even impossible to make such calculations on a function however. Possibly it is simply too large for the architecture at hand, or has some complexity that makes it impossible to effectively analyse. Here we can still make some progress, although we must sacrifice some efficiency and we are less able to provide concrete predictions of the convergence rate of the algorithm.-

Instead we initialise a set of networks with random weights. After testing each of them against some measure (often although not always some loss function) we select those which have performed best and discard the remainder. We then add some random perturbation to the weights of the remaining networks and test them against the most successful options from the previous round. We continue until we have found a sufficiently effective network.

The randomness inherent in evolution means that it can be difficult to predict how long it will take to come to a solution, and the method is extremely reliant on having a beneficial initial condition (although this is also true for backpropogation). However, in situations involving extremely complex functions, or those with particularly pathological objective functions, for example those which are non-differentiable at a large number of points, or those for which we are able only to see the output of the loss function, with no information on its inner workings, this can remain a good or possibly the only option.

[1] The most common activation function, ReLU is not in fact differentiable, having a discontinuity at zero. This is generally solved by setting the gradient at 0 to be 0, arguing that this provides a sufficient approximation to a correct function gradient, although any subgradient could be used

# 2. Stochastic Gradient Descent

## 2.1 What is Stochastic Gradient Descent?

In solving the optimisation problem in §1.4.1, we do not have an insight into the problem over the whole domain (i.e. we cannot say anything very general about the relationship between the weights of the network and the outcome of the loss function). Instead we take the information that we do have: the value of the loss function $\mathcal{L}(y, f(x, \theta))$ for a particular set of weights $\theta$, and the gradient at that point, and attempt to move across the domain of possible network weights in such a way that the value of the loss function decreases. This approach from [7] generally describes gradient descent:

**Algorithm 1:** Gradient Descent

t = **learning rate**;
**while** *stopping criteria not met* **do**
    $\Delta w = -\nabla \text{loss}(w)$;
    $w := w + t\Delta w$
**end**

However, here we are presented with a number of problems. Firstly $t$ is a hyperparameter. We want to ensure that $t$ is small enough that the function is able to sufficiently converge, but large enough that convergence occurs at a practical rate. This depends on the function at hand, and also the nature of the convexities (a largely flat function with steep convexities will require a different approach from one with large, shallow convexities). While we can take analytical approaches to determining a value of $t$ for each step of the algorithm (see [7, §9.2]), this is generally not applied in machine learning and instead we generally opt to choose a conservative learning rate. Choosing the ideal learning rate for a particular problem presents a challenge. A frequent approach is merely to adopt some systematic method of testing rates on a short run or smaller problem and choosing that with the best result. Some alternative methods to avoid this difficulty are discussed in §2.3.

Considering that our loss function is large and complicated, it is extremely unlikely that the local minimum point we are approaching is the global minimum. While it would be impractical to require that we find this, ideally we would like to find a minimum that is *close* to global minimum (for some definition of close). We thus take an approach which sometimes moves us away from the nearest local maximum, in the hope that this moves us closer to some lower point.

Aside from the analytical problems, we are also presented with practical ones. It is likely that it will be impractical to calculate $-\nabla \text{loss}(x)$. The loss function is some function of the loss functions of all the training data for the network, which can be tens of thousands or millions of items of data (particularly for networks which take in streaming data from some sensor for

example). Instead we apply the algorithm to some subset of the data, making the assumption that the distribution of data in the selection is similar to that in the entire population[1] This will allow us to make some progress towards a local minimum which we hope is shared between this data and the set as a whole.

In **Stochastic Gradient Descent** we select a single training value $x^\dagger$ at random and calculate the gradient only for the loss function of that value, i.e. at each loop we select a new $x^\dagger$ and update $W := W - t\nabla \text{loss}_{x^\dagger}(W)$. Often rather than selecting the value entirely at random we take some more representative approach: in smaller data sets the input set can be randomly shuffled and then looped through, ensuring that each point is represented in the optimisation. In larger or streamed data sets, random new data points can be sampled and used to update the network weights. This method allows for faster calculation, however it also often results in noisier convergence, as there is no guarantee that consecutively chosen data points will give updates with similar directions.

**Mini-Batch Gradient Descent** provides some of the advantages of Stochastic Gradient Descent, while avoiding the difficulty in using the entire data set for each calculation (called **Batch Gradient Descent**). Here we select only a small subset of the training values with which to calculate the weight updates. This provides a number of advantages: the calculation is significantly more tractable, and we are able to choose batch sizes which are appropriate to the available memory. It is also likely that in a large data set, there are many pieces of similar data, and we reduce the likelihood that we are using computation time to complete a large number of similar loss calculations. Averaging the loss before optimising means that we are able to move more smoothly, and that we are more likely to gain a result that optimises more of the domain data. PyTorch and Keras both implement Mini-Batch Gradient Descent as their default SGD method.

## 2.2 Backpropogation for SGD

### 2.2.1 In Feedforward Networks

We can view a feedforward network as the composition of a set of functions $a_l$, the activation function for each layer $l$, $g_l$, the functions $g_l(x) := W_l x + b_l$ and $\mathcal{L}$, the loss function, such that

$$Loss(x) = \mathcal{L}(y, a_l(g_l(a_{l-1}(g_{l-1}(\dots a_1(g_1(x))\dots))))) \tag{2.1}$$

Notate the output of layer $l$ as $\sigma_l$, i.e.

$$\sigma_l = a_l(g_l(a_{l-1}(g_{l-1}(\dots a_1(g_1(x))\dots))))$$

[1] This assumption can cause particular problems for ordered data, e.g. testing sequential thermometer readings to determine an anomaly. Here readings come in large blocks of very similar readings, however readings separated by a long time can (but will not necessarily) differ by a great deal. Here we could cause the network to learn only about one single aspect of the data (say, readings during the daytime), and slowly shift the network to then recognise only night-time readings, and to mark daytime readings as errors, if the network is not provided with sufficiently random samples from the entire population set. This is often solved by retaining some older samples as a stock and adding samples from the stock into each mini batch when optimising.

and define $\sigma_0 = x$. In practice we would operate on tensors, however without loss of information, we can flatten tensors to vectors and so defining the operation solely for vectors is sufficient [3, §6.5.2].

We keep our input and ideal output $x$ and $y$ fixed, and instead redefine the loss function in terms of $W$, the matrix of all network weights. It would be difficult and inefficient to calculate this directly, however we can make use of the chain rule to calculate $\nabla_{W_l} Loss$ for each layer $l$.

We can write

$$\nabla_{W_l} Loss = \nabla_{a_l} Loss \odot \frac{da_l}{dg_l} \frac{\partial g_l}{\partial W_l} \tag{2.2}$$

On the assumption that we know $\mathcal{L}'_y(x)$ ($\mathcal{L}'(y,x)$ in terms of $x$), we have simply that

$$\nabla_{a_l} Loss = \mathcal{L}'_y(\sigma_l) = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial \sigma_{l(1)}} \\ \vdots \\ \frac{\partial \mathcal{L}}{\partial \sigma_{l(n)}} \end{bmatrix} =: \delta_{\mathcal{L}}$$

Having calculated $\sigma_l$ passing forwards through the network, we can simply save the value and use it here. We have that

$$\frac{da_l}{dg_l} = a'_l(g_l(\sigma_{l-1})) \tag{2.3}$$

Again, as we must calculate $g_l(\sigma_{l-1})$ in a forward pass through the network, we can re-use the value here. Finally for this layer, from above:

$$\frac{\partial g_l}{\partial W_l} = \frac{\partial (W_l \sigma_{l-1} + b_l)}{\partial W_l} = \sigma_{l-1}^T \tag{2.4}$$

We thus have that

$$\begin{aligned} \nabla_{W_l} Loss &= \nabla_{a_l} Loss \odot a'_l(g_l(\sigma_{l-1})) \cdot \sigma_{l-1}^T \\ &= \delta_{\mathcal{L}} \odot a'_l(g_l(\sigma_{l-1})) \cdot \sigma_{l-1}^T \end{aligned} \tag{2.5}$$

For simplicity we can define

$$\delta_l := \delta_{\mathcal{L}} \odot a'_l(g_l(\sigma_{l-1})) \tag{2.6}$$

We can then extend the ideas in (2.5) to calculate the partials gradient for the remaining weights. In general we can shorten (2.1) to say that

$$Loss(x) = \mathcal{L}_y(a_l(g_l(\dots a_r(g_r(\sigma_{r-1}))\dots))), \quad r < l-1 \tag{2.7}$$

We can then follow (2.2) to see that

$$\nabla_{W_r} Loss = \nabla_{a_l} Loss \odot \frac{da_l}{dg_l} \frac{dg_l}{da_{l-1}} \odot \frac{da_{l-1}}{dg_{l-1}} \dots \frac{dg_{r+1}}{da_r} \odot \frac{da_r}{dg_r} \frac{\partial g_r(\sigma_{r-1})}{\partial W_r}$$

as

$$\frac{dg_l}{da_{l-1}} = \frac{d(W_l a_{l-1} + b_l)}{a_{l-1}} = W_l^T$$

we can combine this with the definition in (2.3) and (2.4) to see that

$$\begin{aligned}\nabla_{W_r} Loss &= \nabla_{a_l} Loss \odot a_l'(g_l(\sigma_{l-1})) \cdot W_l^T \odot a_{l-1}'(g_{l-1}(\sigma_{l-2})) \odot \ldots \cdot W_{r+1}^T \odot a_r'(_r(\sigma_{r-1})) \cdot \sigma_{r-1} \\ &= \delta_l \cdot W_l^T \odot a_{l-1}'(g_{l-1}(\sigma_{l-2})) \odot \ldots \cdot W_{r+1}^T \odot a_r'(_r(\sigma_{r-1})) \cdot \sigma_{r-1}^T\end{aligned}$$

But we can see that clearly

$$\nabla_{W_r} Loss = \nabla_{W_{r+1}} Loss \cdot \frac{1}{\sigma_r} \cdot W_{r+1}^T \odot a_r'(g_r(\sigma_{r-1})) \cdot \sigma_{r-1}^T$$

and so, rather than computing the gradient of all weights together, we can instead calculate the partial derivative of the weights of each later in turn, and recursively calculate the next derivative by defining for layer $i$

$$\delta_{i-1} = \delta_i W_i^T \odot a_{i-1}'(g_{i-1}(\sigma_{i-2})) \tag{2.8}$$

and thus

$$\nabla_{W_i} Loss = \delta_i \sigma_{i-1}^T \tag{2.9}$$

Following the same procedure we can calculate the gradients for $\mathbf{b}$ in the same way, however as

$$\frac{d(Wx+b)}{db} = 1$$

we simply have that $\nabla_{b_i} Loss = \delta_i$.[1]

We thus need only a single pass forwards and an single pass backwards through the network to complete one loop of **Algorithm 1**.

### 2.2.2 In Convolutional Networks

We can apply the same approach to convolutional networks. Although the inner workings of each layer are more complex than those in a simple feedforward network, if we are able to define each layer as a composition of differentiable functions, then we can apply the same ideas of backpropogation.

We would also like to compute $\nabla_{W_i} Loss(x)$ for each layer $i$. Define $f_i$ as the function making up layer $i$. By (2.2.1), we need only calculate $\nabla_{W_l} f_l$ for each layer, and we can then use the chain rule to calculate the required multiplier.

For each $i$, $f_i(x)$ is, as above a combination of up to four functions, $P_i(x)$, a padding function $F_i(x)$, the function applying the filter, $a_i(x)$, an activation function, and $\Phi_i(x)$, a pooling function, such that $f_l(x) = \Phi_i(a_i(F_i(P_i(x))))$. We can thus write

$$\frac{\partial f_i(x)}{\partial W_i} = \frac{\partial \Phi_i(a_i(F_i(P_i(x))))}{\partial a_i(F_i(P_i(x)))} \frac{\partial a_i(F_i(P_i(x)))}{\partial F_i(P_i(x))} \frac{\partial F_i(P_i(x))}{\partial W_i}$$

[1] This is sometimes calculated together with the gradient for $W$ by appending the biases to the weights matrix. We can achieve the same result by appending 1s appropriately to the input and multiplying by the extended weight matrix. This requires fewer loops but more memory and more setup. As ever, it's a tradeoff as to which is appropriate for a particular scenario.

We have that $\Phi_i(x)$ is $Pool_i \cdot vec(x)$ for some 0,1 matrix $Pool_i$.[1] Thus

$$\frac{\partial \Phi_i(a_i(F_i(P_i(x))))}{\partial a_i(F_i(P_i(x)))} = Pool_i$$

As above,

$$\frac{\partial a_i(F_i(P_i(x)))}{\partial F_i(P_i(x))} = a_i'(F_i(P_i(x)))$$

Having calculated $F_i(P_i(x))$ on our forward pass of the network (and assuming we have chosen $A$ to be a scalar differentiable function), we can thus calculate this simply enough.

Finally

$$\frac{\partial F_i(P_i(x))}{\partial W_i} = \frac{\partial W_i(vec(P_i(x))) + b_i}{\partial W_i} = vec(P_i(x))$$

Thus

$$\frac{\partial f_i(x)}{\partial W_i} = Pool_i \cdot a_i'(F_i(P_i(x))) \cdot vec(P_i(x))$$

We can thus find a similar recursive formula to that in (2.9) with

$$\delta_i = Pool_i \cdot a_i'(F_i(P_i(x)) \quad i \in \mathbb{N}^+$$
$$\delta_0 = \frac{\partial Loss(\sigma_l)}{\partial \sigma_l}$$

Thus

$$\frac{\partial Loss(x)}{\partial W_{l-i}} = \left(\prod_{j=0}^{i} \delta_j\right) \cdot vec(P_i(\sigma_{(l-i)-1}))$$

## 2.3 Similar methods and adaptations

As well as just using SGD to improve the parameters in the network, there are other improvements one can use (PyTorch offers momentum[2] and Nestyov momentum). Adaptations of SGD include AdaGrad and ADAM, which use the same general approach but with small changes to improve convergence time or computational complexity.

### 2.3.1 Momentum

SGD throws away any previous information we might have found useful at the end of each iteration. It may be the case that our randomly selected inputs provide downward trajectories towards different local minima, (possibly indicating that neither is an appropriately general local minima), or that several of them generate a similar direction. It is likely that in the former case

[1] As much of the input to $Pool_i$ is discarded, it is possible to speed up the gradient calculation by tracking which values will later be discarded and skipping all further backpropogation steps involving them.

[2] sometimes also referred to as "heavy ball"

we would want to take only cautious steps in these directions, while in the latter this would strengthen our conviction and encourage us to travel further in the indicated direction. In order to achieve this we can continue to move in the previously indicated direction during the next iteration, i.e. rather than merely subtracting the calculated gradient $-\nabla loss(w)$ as in **Algorithm 1** in each step, instead we store a velocity vector $v$, which we update each iteration. As described in [3, p.p.294]:

**Algorithm 2:** Stochastic Gradient Descent with Momentum

t = **learning rate**;
v = 0 **initial momentum**;
$\beta$ = **momentum parameter**;
**while** *stopping criteria not met* **do**
  Sample a minibatch of $m$ examples from the training set $\{x^{(1)},\ldots,x^{(m)}\}$ with corresponding targets $y^{(i)}$ ;
  Compute gradient estimate $\mathbf{g} \leftarrow \frac{1}{m}\nabla_W \sum_i Loss(f(x^{(i)};W),y^{(i)}).$ ;
  Compute velocity update $v \leftarrow \beta v - tg.$ ;
  Apply update; $W \leftarrow W + v$
**end**

If we write $\mathbf{g}_i$ as the gradient estimate for minibatch $i$ and $\mathbf{v}_0$ as the initial velocity then we can view the velocity at iteration $i$ as the weighted sum of previous gradients

$$\mathbf{v}_i = t(g_i - \beta g_{i-1} - \beta^2 g_{i-2} + \ldots \beta^{i-1} g_1 + \beta^i v_0) \tag{2.10}$$

We are thus able to retain some of the previously calculated information about the topology of the space in our future decisions.

However, this introduces a new source of potential error, namely that while each individual gradient update will direct the solution towards some form of local minima, this weighted sum may not. To correct for this we can instead make use of **Nesteyov Momentum**, in which we apply this weighted sum of previous terms and *then* calculate and apply the gradient descent step.

**Algorithm 3:** Stochastic Gradient Descent with Nesteyov Momentum

t = **learning rate**;
v = 0 **initial momentum**;
$\beta$ = **momentum parameter**;
**while** *stopping criteria not met* **do**
  Sample a minibatch of $m$ examples from the training set $\{x^{(1)},\ldots,x^{(m)}\}$ with corresponding targets $y^{(i)}$ ;
  Compute lookahead weights $W^* \leftarrow W + \beta v$ ;
  Compute gradient estimate $g \leftarrow \frac{1}{m}\nabla_{W^*} \sum_i Loss(f(x^{(i)};W^*),y^{(i)}).$ ;
  Compute velocity update $v \leftarrow \beta v - tg.$ ;
  Apply update; $W \leftarrow W + v$
**end**

These approaches serve to remove some of the zig-zagging effect characteristic of SGD

even on simple datasets, as in **Figure 2.1**. Both rely on some important assumptions, namely

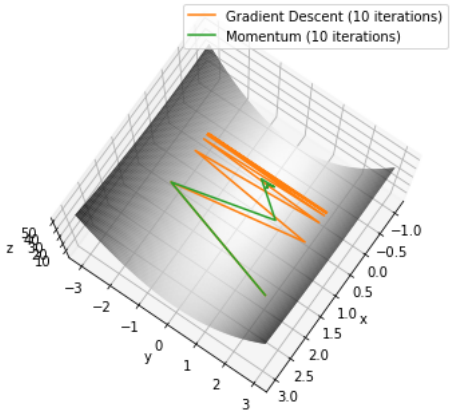


**Figure 2.1:** Gradient descent vs gradient descent with 0.3 momentum, both with learning rate 0.2 optimising $f(x,y) = x^2 + 5y^2$

that all data is equally trustworthy, that it is unlikely that a single data point will be re-sampled sufficiently frequently to greatly affect the result, and the topology of the loss function is not perverse in such a manner that this is likely to reinforce a false result, however these are safe assumptions to make for many data sets, and often we can pre-process the data in some way to make this a safe approach.

A particular difficulty however is the introduction of a new hyperparameter $\beta$. Without previous knowledge of the domain, we are required to tune this hyperparameter in order to achieve a sufficient improvement in the result (too high a weighting and the algorithm will be insufficiently sensitive to new data, too low and we are unable to counteract the slow convergance of standard SGD).

### 2.3.2 AdaGrad

We can instead adaptively use the previous gradients in order to control the direction of movement. Specifically, we would like to move more in flatter directions on the domain, and take more caution (and thus move more slowly) across steeper parts. We can take a simple measure of this magnitude by calculating the componentwise square of the gradient $\mathbf{g} \odot \mathbf{g}$, and can encourage greater movement in those directions with smaller components by using $\left(\frac{t}{\sqrt{\mathbf{g} \odot \mathbf{g}}}\right)_{i,j}$ as the learning rate for weight $w_{i,j}$.[1]

More generally, we would like for this effect to take into account more of our understanding of the domain than just the current point, and to be strong early in the learning process while decreasing over time as we (ideally) grow closer to a minimum. Both of these effects can be achieved by summing all previous squared gradients and multiplying by the reciprocal of the

[1] One could view this as similar to the approximation of curvature used in Newton's method in §3

square root of this sum $\frac{t}{\sqrt{\sum_\alpha \mathbf{g}_\alpha \odot \mathbf{g}_\alpha}}$. In order to avoid division by zero, we add a small constant $\varepsilon$ to the denominator.

**Algorithm 4:** AdaGrad

t = **learning rate**;
$\varepsilon$ = **division constant**;
**while** *stopping criteria not met* **do**
  Sample a minibatch of $m$ examples from the training set $\{x^{(1)},\ldots,x^{(m)}\}$ with corresponding targets $y^{(i)}$ ;
  Compute gradient estimate $g \leftarrow \frac{1}{m}\nabla_W \sum_i Loss(f(x^{(i)};W),y^{(i)}).$ ;
  Update gradient sum $r \leftarrow r + g \odot g$ ;
  Compute update $v \leftarrow -\frac{t}{\varepsilon+\sqrt{r}} \odot g.$ ;
  Apply update; $W \leftarrow W + v$
**end**

AdaGrad allows us to move quickly over initially flat areas, however it is sensitive to the initial conditions. The gradients calculated early in the iteration are given large weight in the direction moved across the domain, and so rather than the more general search performed by SGD, AdaGrad is significantly influenced by its initial point and, running on the assumption that our large initial movements will move us towards a steeper downwards slope, the learning rate rapidly decreases. If this assumption is incorrect convergence can be slow.

### 2.3.3 RMSProp

In order to combat these difficulties, we can instead weight the previous gradients in our sum as in (2.10) to decay the effect of earlier positions. This both reduces the large effect of the initial position and prevents the decrease in learning rate outside convex bowls[1].

**Algorithm 5:** RMSProp

t = **learning rate**;
$\varepsilon$ = **division constant**;
$\rho$= **decay rate**;
**while** *stopping criteria not met* **do**
  Sample a minibatch of $m$ examples from the training set $\{x^{(1)},\ldots,x^{(m)}\}$ with corresponding targets $y^{(i)}$ ;
  Compute gradient estimate $g \leftarrow \frac{1}{m}\nabla_W \sum_i Loss(f(x^{(i)};W),y^{(i)}).$ ;
  Update gradient sum $r \leftarrow \rho r + (1-\rho) g \odot g$ ;
  Compute update $v \leftarrow -\frac{t}{\varepsilon+\sqrt{r}} \odot g.$ ;
  Apply update; $W \leftarrow W + v$
**end**

[1] If the desired minimum is not in a convex bowl however, as in the Rosenbrock function for example (see figure 5.1), the learning rate can remain unhelpfully large.

### 2.3.4 ADAM

ADAM attempts to solve the issue of a static learning rate by combining the previous approaches. Rather than merely relying on a a single exponentially weighted average, the algorithm estimates the first and second moments of the gradient ($m$ and $v$), using weighted averages of the gradient and its square at each iteration, corrects these against initialisation bias and computes the update $\delta W = -\frac{\tilde{m}}{\varepsilon+\tilde{v}}$.[1] This bias-correction prevents the gradients from being too heavily influenced by their zero-initialisation[8, §3]

**Algorithm 6:** ADAM[8]

t = **learning rate**;
$\rho_1, \rho_2$= **decay rates**;
$\varepsilon$ = **division constant**;
$m_0 \leftarrow 0$;
$v_0 \leftarrow 0$;
$t \leftarrow 0$;
**while** *stopping criteria not met* **do**
  $t \leftarrow t+1$ Sample a minibatch of $m$ examples from the training set $\{x^{(1)},\ldots,x^{(m)}\}$ with corresponding targets $y^{(i)}$ ;
  Compute gradient estimate $g_1 \leftarrow \frac{1}{m}\nabla_W \sum_i Loss(f(x^{(i)};W),y^{(i)})$. ;
  Update first moment estimate $m_t \leftarrow \rho_1 \cdot m_{t-1} + (1-\rho_1)\cdot g_t$ ;
  Update second moment estimate $v_t \leftarrow \rho_2 \cdot v_{t-1} + (1-\rho_2)\cdot g_t \odot g_t$ ;
  Correct first moment estimate for bias $\tilde{m}_t \leftarrow \frac{m_t}{1-\rho_1^t}$;
  Correct second moment estimate for bias $\tilde{v}_t \leftarrow \frac{v_t}{1-\rho_2^t}$;
  Compute update $\Delta W \leftarrow -t \cdot \frac{\tilde{m}_t}{\varepsilon+\sqrt{\tilde{v}_t}}$. ;
  Apply update; $W \leftarrow W + \Delta W$
**end**

### 2.3.5 Criticisms

There are some criticisms, specifically Wilson et al.[9] argue that while adaptive methods might alleviate the need for parameter tuning, they instead converge to 'simpler' solutions less likely to generalise correctly to a test set. In their experiments they note that while adaptive measures tend to train well initially, they tend to converge to a less optimal solution, whereas purely momentum based methods converged to solutions which displayed improved outcomes on the test set. While this remains a source of debate, adaptive methods remain popular and continue to be offered and widely used in both PyTorch and Keras.

[1] [3] notes that this "use of momentum in combination with rescaling does not have a clear theoretical motivation".

# 3. Conjugate Gradient

## 3.1 The Conjugate Gradient method

Minimising a function in a large domain can (as we have seen) be an extremely complex problem. For this chapter we will assume that we have found suitable quadratic approximation of the function we would like to minimise, $f(x) = x^T Ax - b^T c + c$, with $A$ a symmetric positive definite matrix[1] and instead attempt to minimise this. The pricipal behind the Conjugate Gradient method is that we are able to select some lower dimension sub-domain and attempt to minimise the function our approximation over just this. We can then expand the subdomain by adding some conjugate vector to the basis and attempt to further minimise the function by traversing only along this vector. We can therefore move towards an estimate of a minimum without any backtracking, which should allow for faster convergence.

This requires some background explanation. Ideally we would like to find a path to the minimum point where the path between each guess is orthogonal to the one before. For example to solve

$$f(x) = x^T \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} x - x^T \begin{bmatrix} 0 \\ 8 \end{bmatrix} + 16$$

we might begin with a guess at the point $x_0 = (0,0)$ and some direction to minimise along, say $d_0 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$. If $x^*$ is the minimal point, we thus want to find some $\alpha$ such that

$$(x^* - (x_0 + \alpha d_0))^T \cdot d_0 = 0$$

i.e. such that once we have moved to the point $x_0 + \alpha d_0$, *any* further travel towards $x^*$ is orthogonal to $d_0$. But by rearranging, we see that

$$\alpha = -\frac{(x^* - x_0)^T d_0}{d_0^T d_0}$$

in other words if we are able to solve this, we must also be able to find the vector $x^* - x_0$, and there would be no need to complete any further iterations. As we do not have this information, we must take another tack.

We can't find vectors of the form $x^* - x_i$, but we do have that if our function is of the form $f(x) = \frac{1}{2}x^T Ax - b^T x + c$ and $A$ is symmetric positive definite, we must have that $f$ is minimised

[1] A matrix such that for all vectors $v$, $v^T Av > 0$.

by the solution to $g(x) = Ax - b$.[1][10, Appendix C1] i.e. we have that $Ax^* = b$. We therefore *can* find the *residual*, $Ax^* - Ax_i = b - Ax_i$ for any point $x_i$. We could instead use *this* to aid us in finding the solution.

Rather than solving the problem in $\mathbb{R}^n$, we can instead attempt to solve the problem in the eigenspace $\mathcal{S}_A \subset \mathbb{R}^n$, the $A$-invariant subspace[2] with basis vectors being the eigenvectors of $A$. We can easily define a linear map $\phi : \mathcal{R}^n \to \mathcal{S}_A, x \mapsto Ax$. Rather than choosing directions which are orthogonal in $\mathbb{R}^n$, we can instead choose directions which have orthogonal images under $\phi$. We refer to these directions as *conjugate*. More generally, a pair of vectors $u$, $v$ are $A$-conjugate if $u^T Av = 0$. As $A$ is positive definite, this is equivalent to our previous definition.

After selecting a direction $d_i$, we would therefore like to find $\underset{\alpha_i}{\operatorname{argmin}} f(x_i + \alpha_i d_i)$.

Following the example from [10] we can expand to get

$$\begin{aligned}
\frac{d}{d\alpha_i} f(x_i + \alpha_i d_i) &= 0 \\
(b - Ax_{i+1})^T d_i &= 0 \\
(b - A(x_i + \alpha_i d_i))^T d_i &= 0 \\
(b - Ax_i)^T d_i &= \alpha_i (Ad_i)^T d_i \\
r_i^T d_i &= \alpha_i d_i^T A\dot{i} \\
\alpha_i &= \frac{r_i^T d_i}{d_i^T A d_i}
\end{aligned}$$

We can also note that

$$\begin{aligned}
r_{i+1} &= b - Ax_{i+1} \\
&= b - A(x_i + \alpha_i d_i) \\
&= r_i - \alpha_i A d_i
\end{aligned}$$

and so we need perform only the single matrix-vector multiplication $Ad_i$ each round to calculate both $x_{i+1}$ and $r_{i+1}$.

Thus, having chosen a direction, we can find find an appropriate step length such that any further steps should be $A$-conjugate to $d_i$. However, the problem still remains that we must choose an appropriate sequence of conjugate directions. Here we can turn to the theory of Krylov Subspaces.

---

[1] We have that $\nabla f(x) = (\frac{\partial f}{x_{(1)}}, \ldots \frac{\partial f}{x_{(n)}}) = \frac{1}{2}A^T x + \frac{1}{2}Ax - b$. The gradient must thus be 0 when $Ax = b$ as $A$ is symmetric. For any arbitrary vector $p$, and optimal point $x^*$ we have that
$f(x^* + p) = \frac{1}{2}(x^* + p)^T A(x^* + p) - b^T(x^* + p) + c$
$= \frac{1}{2}\left((x^*)^T Ax^* + p^T Ap\right) + b^T p - b^T p - b^T x^* + c$
$= f(x^*) + \frac{1}{2}p^T Ap$ As $A$ is positive definite, $x^*$ must be a minimum.

[2] A subspace such that $\forall x \in \mathcal{S}_A, Ax \in \mathcal{S}_A$

### 3.1.1 Krylov subspaces

We know that the basis vectors of $\mathcal{S}_A$ are the eigenvectors of $A$, however it is likely that $A$ is far too large and complex a matrix for us to be able to calculate these. Instead we would like to create some subspace that approximates $\mathcal{S}_A$ without all of its complexity, and without having access to $\mathcal{S}_A$ or $A$ directly.

If the vector $b$ is not an eigenvector of $A$, then, by definition $Ab$ is an $m$-vector not co-linear with $b$. We can thus build up an $A$-invariant subspace

$$\mathcal{K}_n(b,A) = \text{span}\{b, Ab, A^2b, \ldots, A^{n-1}b\}$$

the order $n$ Krylov subspace of $A$ and $b$. If $b \in \mathcal{S}_A$ then $\mathcal{K}_n \subseteq \mathcal{S}_A$, and if $A^t b$ is not an eigenvector of $A$ for too small a value $t < n-1$ then $\mathcal{K}_n$ is a good approximation of $\mathcal{S}_A$. Importantly basis elements of $\mathcal{K}_n$ are also basis elements of $\mathcal{S}_A$, and so if we can find an $A$-conjugate basis for $\mathcal{K}_n$, we can use these as directions in our optimisation.

We do this using a modification of the Gram-Schmidt process. We first note that given the construction of $\alpha$ above, the residuals $r_i$ must be orthogonal to each other. The set $\{r_0, \ldots r_{n-1}\}$ is also the basis of $\mathcal{K}_n$. As $r_0 = b$, this is trivially true for $n = 0$. By the construction of $r_i$, we can see that $r_1 \in \text{span}\{r_0, Ar_0\} = \mathcal{K}_2$. By induction if $r_{i-1} \in \mathcal{K}_i$, we have that $r_i \in \mathcal{K}_i \cup A\mathcal{K}_i = \mathcal{K}_{i+1}$. As the residuals are mutually orthogonal, they must form an orthogonal basis to $K_{i+1}$. As a side note, we can consider the $n$th iteration of CG the best solution in $\mathcal{K}_n$, as we solve for the best solution in each orthogonal basis vector in each step.

Given some direction $d_i$, and the next basis vector $r_{i+1}$, we would like to find an $A$-conjugate direction $d_{i+1}$, i.e. such that $d_i A d_{i+1} = 0$. Again, using Gram-Schmidt, we subtract some linear combination of the previous directions in order to achieve the conjugacy

$$\begin{aligned}
d_{i+1} &= r_{i+1} - \sum_{n=0}^{i} \beta_{i+1,n} d_n \\
d_{i+1} i^T A d_j &= r_{i+1} A d_j + \sum_{n=1}^{i} \beta_{i+1,n} d_n A d_j \\
0 &= r_{i+1} A d_j + \sum_{n=1}^{i} \beta_{i+1,n} d_n A d_j \\
r_{i+1} A d_j &= -\beta_{i+1,j} d_j A d_j \\
\beta_{i+1,j} &= -\frac{r_{i+1} A d_j}{d_j A d_j}
\end{aligned} \tag{3.1}$$

However, we can make use of the identity

$$r_{i+1}^T r_{j+1} = r_{i+1}^T r_j - \alpha_j r_{i+1}^T A d_j$$

to note that

$$r_{i+1}^T A d_j = \frac{1}{\alpha_j} \left( r_{i+1}^T r_j - r_{i+1}^T r_{j+1} \right)$$

As all residues are mutually orthogonal, and we have that $j < i+1$, we can see that if $i \neq j$, $r_{i+1}^T A d_j = 0$, however if $i = j$,

$$r_{i+1}^T A d_i = \frac{1}{\alpha_i} r_{i+1}^T r_{i+1}$$

Substituting back into (3.1) we have that

$$\beta_{i+1,i} = \frac{1}{\alpha_i} \frac{r_{i+1}^T r_{i+1}}{d_i^T A d_i}$$
$$\alpha_i = \frac{r_i^T d_i}{d_i^T A d_i}$$

As $d_i = r_i + \beta d_{i-1}$, and $r_i$ is orthogonal to all $r_k$, $k \neq i$, we can rewrite $\alpha_i$ as

$$\alpha_i = \frac{r_i^T r_i}{d_i^T A d_i}$$

and thus we have that

$$\beta_{i-1,i} = \frac{r_{i+1}^T r_{i+1}}{r_i^T r_i}$$

and thus that the direction

$$d_{i+1} = r_{i+1} - \frac{r_{i+1}^T r_{i+1}}{r_i^T r_i} d_i$$

is $A$-conjugate to all $i$ previous directions, and is thus an acceptable search direction.

If $A$ is an $n \times n$ matrix, if we repeat the process $n$ times, we will have optimised over $n$ orthogonal directions in the at most $n$-dimensional $\mathcal{S}_A$, and thus the process must converge. (In general, if we require convergence only to a close approximation, this is likely to be significantly faster).

In short, we can formalise the CG algorithm as:

**Algorithm 7:** The Conjugate Gradient Method

$\mathbf{x}_0 = 0$ ;
$\mathbf{r}_0 = \mathbf{d}_0 = \mathbf{b}$;
**for** $k = 1$ ***to*** $N$ **do**
  $\alpha_{k-1} = \frac{\mathbf{r}_{k-1}^T \mathbf{r}_{k-1}}{\mathbf{d}_{k-1}^T A \mathbf{d}_{k-1}}$;
  $\mathbf{x}_k = \mathbf{x}_{k-1} + \alpha_{k-1} \mathbf{d}_{k-1}$;
  $\mathbf{r}_k = \mathbf{r}_{k-1} - \alpha_{k-1} A \mathbf{d}_{k-1}$;
  $\beta_k = \frac{\mathbf{r}_k^T \mathbf{r}_k}{\mathbf{r}_{k-1}^T \mathbf{r}_{k-1}}$;
  $\mathbf{d}_k = \mathbf{r}_k + \beta_k \mathbf{d}_{k-1}$;
**end**

# 4. Newton's Method

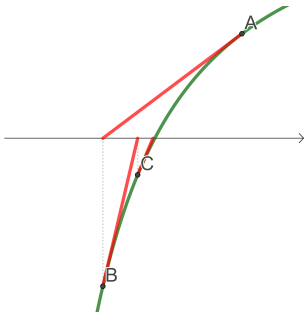


**Figure 4.1:** Using three iterations of Newton's Method to approximate the root of a function.

Newton's Method for functions is an iterative root-finding algorithm making use of the first derivative. Taking some function $f(\mathbf{x})$, we can approximate this around the point $\mathbf{x}_0$ as the linear function $f'(\mathbf{x}_0)(\mathbf{x}-\mathbf{x}_0)+f(\mathbf{x}_0)$. We can easily find the root of this approximation

$$\mathbf{x}_1 = \mathbf{x}_0 - \frac{f(\mathbf{x}_0)}{f'(\mathbf{x}_0)} \tag{4.1}$$

On the assumption that the function $f$, is continuously differentiable and that our initial guess is sufficiently close to a root, following this iteration provides us with a fast method to solve the problem.

While this is not natively a method to solve optimisation problems (as the minimum is very unlikely to be zero, and is extremely likely to occur at a position at which the function is ill-suited to the algorithm), if our function is *twice* differentiable, we can instead attempt apply this method to the gradient in an attempt to find a stationary point. However, the behaviour of the iterations far from the roots can be complex, and so we add some further requirements in order to ensure that the function will converge correctly.

## 4.1 Newton's Method for Optimisation

### 4.1.1 Convex Functions

Our search for a good solution (see §1.4.1) assumes that a good local minimum lies at a stationary point (or an almost stationary point). We can thus attempt to find a root of the gradient

function, and iterate towards a point at which the gradient is zero [1] using the iteration

$$\mathbf{x}_{n+1} = \mathbf{x}_n - (\nabla^2 f(\mathbf{x}_n))^{-1} \nabla f(\mathbf{x}_n). \tag{4.2}$$

Here $\nabla^2$ is the Hessian matrix

$$\nabla^2 f(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2 f}{\partial \mathbf{x}_1^2} & \frac{\partial^2 f}{\partial \mathbf{x}_1 \partial \mathbf{x}_2} & \cdots & \frac{\partial^2 f}{\partial \mathbf{x}_1 \partial \mathbf{x}_l} \\ \frac{\partial^2 f}{\partial \mathbf{x}_2 \partial \mathbf{x}_1} & \frac{\partial^2 f}{\partial \mathbf{x}_2^2} & \cdots & \frac{\partial^2 f}{\partial \mathbf{x}_2 \partial \mathbf{x}_l} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial \mathbf{x}_l \partial \mathbf{x}_1} & \frac{\partial^2 f}{\partial \mathbf{x}_l \partial \mathbf{x}_2} & \cdots & \frac{\partial^2 f}{\partial \mathbf{x}_l^2} \end{bmatrix}$$

However, here we note that $\nabla^2 f(x)$ must be an invertable matrix. We can also revisit our concern from above that not all initial points will converge to a solution. We can solve both of these problems together by limiting ourselves to strictly convex functions (we will return to the idea of non-convex functions in §4.2.1-2). We define a function to be strictly convex on a set $\mathcal{S}$ if for all $x, y$ in $\mathcal{S}$, for any $t$ between 0 and 1 (inclusive), $f(x + t(x - y) < t \cdot f(x) + (1 - t) f(y)$, i.e. that a line drawn between $f(x)$ and $f(y)$ always lies above the function $f$. Here, all Newton iterations must converge to a minimum (see §4.3), and the Hessian must be positive definite and thus non-singular.

In order to use Newton's method to find such a minimum we can thus use the algorithm

**Algorithm 8:** Newton's Method[7, §9.5.2]

$x$ = **starting point**;
$\varepsilon$ = **tolerance**;
**while do**
  Compute the Newton step and decrement $\Delta x_{nt} \leftarrow -\nabla^2 f(x)^{-1} \nabla f(x)$
  $\lambda^2 \leftarrow \nabla f(x)^T \nabla^2 f(x)^{-1} \nabla f(x)$ ;
  Stopping criterion. **quit** if $\frac{\lambda^2}{2} \leq \varepsilon$ ;
  Line search. Choose step size $t$ by line search ;
  Update $x \leftarrow x + t \Delta x_{nt}$
**end**

It is helpful to guard against moving too far in areas of low curvature. As we multiply by the inverse curvature each step, if it is small we are likely to step a very long way. It is preferable to prevent this for two reasons. Firstly, our underlying assumption that the behaviour of the function close to the initial point is similar to that *at* the point will not be fulfilled if we move far from it, and we will thus be even less successful with our predictions. Secondly, as discussed above, we are likely to be operating on some convex approximation of the true function in order to make use of the Newton Method, and our approximation is likely to be significantly less

[1] This is equivalent to taking the second order Taylor series around $x_0$: $f(x) \approx f(\mathbf{x}_0) + \nabla f(\mathbf{x}_0)^T (\mathbf{x} - \mathbf{x}_0) + \frac{1}{2} (\mathbf{x} - \mathbf{x}_0)^T \nabla^2 f(x) (\mathbf{x} - \mathbf{x}_0)$ and finding the minimal point of this approximation at each iteration.

accurate further away from the initial point. Instead we perform some extra checks to determine an appropriate step length[1].

While this is a robust method and converges quadratically (see §4.3), in this form it is still not appropriate to use as an optimisation method. Our assumption that the Hessian is positive definite is certainly not necessarily the case, and in fact we may move towards a saddle point or a local maximum if we do not add something to the algorithm.

The algorithm also requires that we calculate the inverse Hessian for each step. The Hessian is an $n \times n$ matrix (where $n$ is length of the input vector $x$), of second derivatives. Even the Hessian matrix itself would require a great deal of both memory and computation time to compute. As neural networks have thousands of parameters, this is not a practical computation, let alone finding the inverse for each iteration. Instead we must make some modifications in order to be practical.

## 4.2 Adaptations to Newton's Method

If we are trying to minimise the least squares loss, we can make use of the simplicity and predictability of the function to make some better approximations in out methods.

For least squares loss, we say that for any point $x$ we can calculate the residual

$$r(x) = f(x) - f(\hat{x}) \tag{4.3}$$

where $f(\hat{x})$ is the optimal point (note that it is not necessarily the case that $\hat{x}$ is the the domain of $x$). As in [11, p.p. 246], for our loss function $f$ we take:

$$f(x) = \frac{1}{2}\sum_{j=1}^{n} r_j^2(x) \tag{4.4}$$

i.e. the sum of squares of each component of the residual. Differentiating we get that

$$\nabla f(x) = \sum_{j=1}^{n} r_j(x)\nabla r_j(x) = \begin{bmatrix} \nabla r_1(x)^T \\ \nabla r_2(x)^T \\ \vdots \\ \nabla r_n(x)^T \end{bmatrix}^T r(x) = J(x)^T r(x) \tag{4.5}$$

$$\begin{aligned} \nabla^2 f(x) &= \sum_{j=1}^{n} \nabla r_j(x)\nabla r_j(x)^T + \sum_{j=1}^{n} r_j(x)\nabla^2 r_n(x) \\ &= J(x)^T J(x) + \sum_{j=1}^{n} r_j(x)\nabla^2 r_j(x) \end{aligned} \tag{4.6}$$

where

$$J(x) = \begin{bmatrix} \dfrac{\partial f(x)}{\partial f(x_{(1)})} & \cdots & \dfrac{\partial f(x)}{f(x_{(n)})} \end{bmatrix}$$

[1] As we will use the Levenberg-Marquardt method from §4.2.2 to dampen the function, we omit a discussion of line search methods, but a thorough overview can be found in [7, §9.2]

is the Jacobian of $f(x_{(1)},\ldots,x_{(n)})$. From §2.2 we can see that for the loss functions of feed-forward and convolutional neural networks, the Jacobian is relatively simple to calculate and indeed can be effectively and quickly calculated by automatic differentiation packages.

However, the term $\sum_{j=1}^{n} r_j(x)\nabla^2 r_j(x)$ requires calculation of the Hessian, and, as discussed in the previous section, calculating this in full is in general not a practical proposition.

### 4.2.1 Gauss-Newton

For any initial point $x_i$, we can take the Taylor expansion

$$f(x_{i+1}) = f(x_i) + J_i(x_{i+1} - x_i) + HOT$$

If we drop the higher order terms we can say that

$$f(x_{i+1}) - f(\hat{x}) = f(x_i) - f(\hat{x}) + J_i(x_{i+1} - x_i)$$

By (4.3) we have that

$$r(x_{i+1}) = J_i(x_{i+1} - x_i) + r(x_i)$$

We would thus like to find a direction $d = x_{i+1} - x_i$ which minimises $r(x_{i+1})$, in other words to find the best-fit solution to

$$J_i d = -r(x_i) \tag{4.7}$$

From (4.2), in the standard Newton Method, we were looking for a direction $d_i$ such that

$$\nabla^2 f(x_i) d_i = -\nabla f(x_i) \tag{4.8}$$

But, by left-multiplying (4.7) by $J_i^T$ and substituting (4.5) into (4.8) we can see that a solution to (4.7) will give an estimate of the solution to (4.8) under the approximation $\nabla^2 f(x_i) \approx J_i^T J_i$, i.e.

$$\nabla^2 f(x) \approx G(x) = \nabla^2 f(x) - \sum_{j=1}^{m} r_j(x)\nabla^2 r_j(x) \tag{4.9}$$

Methods making use of the approximation in (4.9) are generally known as Gauss Newton methods. There are several advantages to this approach. Firstly we already possess good tools to solve this form of problem for large matrices (namely the Conjugate Gradient method outlined in §3.1). As $J_i^T J$ is certainly a symmetric positive semidefinite matrix, the domain on which we run the CG algorithm will not have negative curvature and we can have confidence in its results. Indeed, if $\nabla f(x_i) \neq 0$ we have that [11, (10.25)]

$$d_i^T \nabla f(x_i) = d_i^T J_i^T r(x_i) = -d_i^T J_i^T J_i d_i = -||J_i d_i||^2 \leq 0$$

in other words, if the Jacobian is of full rank, solving (4.7) will give us a descent direction if we are not already at a local minimum. If $J_i d_i = 0$, and $J_i$ is of full rank, we have that $\nabla f(\mathbf{x}_i) = 0$ and that this is thus an optimal point.

We can view this, as noted by [12], as a second order model of the objective function with the negative curvature removed.

How good this convergence is depends on the solution at $x^*$, the optimal point *inside* the domain of $x$. If $r(x^*) = 0$, i.e. $f(x^*) = f(\hat{x})$ then Kelley [13, 2.4.2] notes that the Gauss Newton method will converge quadratically. We can see a similar requirement that $H(x^*) = 0$ and $J^T J(x^*) = 0$ to gain quadratic convergence from (4.19). The larger $r(x^*)$, the larger $\sum_{j=1}^{m} r_j(x)\nabla^2 r_j(x)$, and so the worse both our approximation and the convergence will be.

However, we would ideally like to be able to use the approximation even when the loss function is not least squares. Schraudolph [14] defines the 'Extended Gauss Newton Approximation' in a similar way as above, noting that

$$f(W) = \frac{1}{|S|} \sum_{(x,y)\in S} L(y, f(x,W))$$

$$\nabla h(W) = \frac{1}{|S|} \sum_{(x,y)\in S} J^T \nabla_z L(y,z)$$

$$H(W) = \frac{1}{|S|} \sum_{(x,y)\in S} J^T H_L J + HOT$$

Where $H_L$ is the hessian of the loss function $\mathcal{L}$ at $f(x,W)$ and $|S|$ is the cardinality of the batch. We thus have that Gauss-Newton is in fact a special case of the more general algorithm, where $H_L = I$. In general these hessians are not complex, and the extra work created by the generalisation is small. This is discussed at greater length by Martens and Sutskever in [15].

### 4.2.2 Levenberg-Marquardt

Rather than using our approximation to find a single descent direction and attempting to minimise the loss function in that direction, we can instead take a different tactic. Already, to find a direction we have made an approximation and performed some analysis, however we could take this one step further and assume that for some region around our initial guess $x_i$, our approximation is reasonably close to the true loss function. If this is the case, then minimising the approximation over the given region should give us some improvement, and this improvement should bring us close to the optimal point of the region. Such methods are known as "trust region" methods and they can offer improvements in situations in which standard Gauss Newton does not give good results.

If we take this trust region to be a disc of radius $\Delta$ around the initial point, from (4.7) we would like to find the optimal solution to

$$\min_d ||J_i d + r_i||^2 : \quad ||d|| < \Delta \tag{4.10}$$

for each iteration. Clearly if the direction found by the Gauss-Newton algorithm is inside the region, this will be the solution to (4.10) otherwise the solution must be some point $d$ such that $||d|| = \Delta$.

We show this following the example from [11] relying on the lemma

**Lemma 4.2.1.** *[11, Lemma 4.7] Define*

$$m(d) = g^T d + \frac{1}{2} d^T B d$$

*where B is a symmetric matrix. Then*

*i) m attains a minimum if and only if B is positive semidefinite and g is in the range of B*

*If B is positive semidefinite, every d such that $Bd = -g$ is a minimum of m.*

*ii) m has a unique minimiser if and only if B is positive definite.*

*Proof.* i) $\Leftarrow$: Take some $d$ such that $Bd = -g$. Then for all vectors $v \in \mathbb{R}^n$:

$$\begin{aligned} m(d+v) &= g^T(d+v) + \frac{1}{2}(d+v)^T B(d+v) \\ &= g^T d + g^T v + \frac{1}{2}d^T Bd + \frac{1}{2}d^T Bv + \frac{1}{2}v^T Bd + \frac{1}{2}v^T Bv \\ &= g^T d + \frac{1}{2}d^T Bd + g^T v + \frac{1}{2}(Bd)^T v - \frac{1}{2}v^t g + \frac{1}{2}v^T bv \\ &= g^T d + \frac{1}{2}d^T Bd + g^T v - \frac{1}{2}g^T v - \frac{1}{2}v^t g + \frac{1}{2}v^t Bw \\ &= m(d) + \frac{1}{2}v^T Bv \end{aligned}$$

But as $B$ is positive semidefinite, we have that

$$m(d+v) \geq m(d) \quad \forall v \in \mathbb{R}^n$$

$\Rightarrow$: Take $d$ a minimiser of $m$. Then we have that

$$\nabla m(d) = g + Bd = 0$$

We thus have that $g$ is in the range of $B$. Taking the second derivative:

$$\nabla^2 m(d) = B$$

As $d$, is a minimum point of $m$, $\nabla^2 m(d)$ must be positive semidefinite. Thus. $B$ is positive semidefinite as required.

ii) $\Leftarrow$: From i) we have that $m(d+v) = m(d) + \frac{1}{2}v^T bv$. If $B$ is positive definite, and $v \neq 0$, then $\frac{1}{2}v^T Bv > 0$. Thus $\forall v \in \mathbb{R}^n, v \neq 0$ we have that

$$m(d+v) > m(d)$$

and thus $d$ is a unique minimiser.

$\Rightarrow$: Say $d$ is a unique minimiser of $B$. By definition $B$ is positive semidefinite. If $B$ is not positive definite then there exists some $v \in \mathbb{R}^n, v \neq 0$ such that $Bv = 0$. However then we have that $m(d+v) = m(d)$, and thus the solution is not unique.

□

We can then apply this to our situation. We have a quadratic function $||J_i d + r_i||^2 = ||r_i||^2 + 2J_i d + d^t J_i^T J_i d$ which we would like to minimise with respect to the constraint $||d|| < \Delta$

**Theorem 4.2.2.** *[11, Theorem 4.1] $d^*$ is a global solution to the problem*

$$\min_{d\in\mathbb{R}^n} m(d) = f(x) + g^T d + \frac{1}{2} d^T B d \quad \textit{such that } ||d|| \leq \Delta \tag{4.11}$$

*if and only if $d^*$ is a feasible solution and there exists some real value $\lambda \geq 0$ such that*

*i)* $(B+\lambda I)d^* = -g$

*ii)* $\lambda(\Delta - ||d^*||) = 0$

*iii)* $(B+\lambda I)$ *is positive semidefinite*

*Proof.* $\Leftarrow$: Assume that there exists some $\lambda$ such that i), ii) and iii) are satisfied. Then we can argue, making use of **Lemma 4.2.1**, that $d^*$ is the global minimum of

$$\begin{aligned} g^T d + \frac{1}{2} d^T (B+\lambda I) d &= g^T d + \frac{1}{2} b^T B d + \frac{\lambda}{2} d^T I d \\ &= m(d) + \frac{\lambda}{2} d^T d \end{aligned}$$

As $d^*$ is a minimum we have that

$$m(d) + \frac{\lambda}{2} d^T d \geq m(d^*) + \frac{\lambda}{2} d^{*T} d$$

$$m(d) \geq m(d^*) + \frac{\lambda}{2}(d^{*T} d^* - d^T d) \tag{4.12}$$

By condition ii):

$$\lambda(\Delta^2 - ||d||^2) = 0 \quad \text{i.e.} \quad \lambda(\Delta^2 - d^{*T} d^*) = 0$$

Substituting this in we get that

$$m(d) \geq m(d^*) + \frac{\lambda}{2}(\Delta^2 - d^T d)$$

However as $d \in \mathbb{R}^n$, $d^T d$ is positive, and thus if $d^T d < \Delta^2$, i.e. $||d|| < \Delta$ (the condition from (4.11)), we have that

$$m(d) \geq m(d^*)$$

Thus $d^*$ is a global minimiser of (4.11).

$\Rightarrow$: Assume $d^*$ is a global solution to (4.11). We can divide the possibilities up. Firstly, say $||d^*|| < \Delta$. Then trivially, i), ii) and iii) hold for $\lambda = 0$.

Otherwise we have that $||d^*|| = \Delta$. Then $\Delta - ||d^*|| = 0$, and so either $\lambda = 0$ or $\Delta - ||d^*|| = 0$, fulfilling condition ii ).

Here we can apply the KKT conditions (see [7, §5.5.3]) to say that there exists a Lagrangian Dual function

$$\mathscr{L}(d,\lambda) = m(d) + \frac{\lambda}{2}(d^T d - \lambda^2)$$

with a stationary point at $d^*$. Differentiating, we have

$$\nabla_d \mathscr{L}(d,\lambda) = g^T + Bd + \lambda d^T$$

As $\mathcal{L}(d,\lambda)$ is stationary at $d^*$, we have that

$$g^T + (B+\lambda I)d^* = 0$$

in other words that

$$(B-\lambda I)d^* = -g^T \tag{4.13}$$

proving i). As $d^*$ is a global solution, we can make use of the inequality (4.12) again. Expanding this we we get

$$g^T d + \frac{1}{2}dBd^T \geq g^T d^* + \frac{1}{2}d^*Bd^*T + \frac{\lambda}{2}(d^{*T}d^* - d^T d)$$

Substituting in (4.13)

$$-((B+\lambda I)d^*)^T d + \frac{1}{2}dBd^T \geq -((B+\lambda I)d^* =^T d^* + \frac{1}{2}d^*Bd^{*T} + \frac{\lambda}{2}(d^{*T}d^* - d^T d)$$

Rearranging we get that

$$\frac{1}{2}\left(d^{*T}Bd^* - d^*TBd - d^TBd^* + d^TBd + \lambda(d^{*T}d - d^{*T}d - d^Td^* + d^Td)\right) \geq 0$$

Collecting terms we find

$$\frac{1}{2}(d-d^*)^T(B+\lambda I)(d-d^*) \geq 0$$

and thus $B+\lambda I$ is positive semidefinite, condition iii). □

We can take this to mean that we can solve (4.11) by finding some $\lambda$ and $d^*$ such that **4.2.2 i),ii),iii)** are all satisfied. This is clearly an impractical proposition, as we are unable to efficiently solve the quadratic optimization problem for the whole domain, and reducing the domain does not provide us with any further tools. However, we are able to use this formulation to find an estimate of the optimal solution. We can see that an increase in $\lambda$ corresponds to a decrease in $\delta$, i.e. the higher the damping paramater, the smaller the trust radius. By choosing some $\lambda$ sufficiently large such that $B+\lambda I$ is positive semidefinite and solving **4.2.2 i)** for $d^*$, we have that there exists some $\Delta_* = ||d^*||$ such that $d^*$ is the global solution to (4.11) where $||d|| \leq \Delta_*$. The parameter $\lambda$ is sometimes referred to as a "damping parameter" as we can use it in this manner to restrict the solution to a related trust-region $||d^*|| \leq \Delta_*$.

Clearly it is not practical to use a single parameter $\lambda$ for each iteration. The trust region (or appropriate value of $\Delta$) for the quadratic approximation at a point varies depending on the local curvature, and so the related damping parameter $\lambda_\Delta$ should also vary. As noted in [16], there do exist methods to calculate such a parameter, however as they must be re-calculated for each point of the iteration, instead it is more practical to utilise a simpler approach.

For each iteration, we calculate the ratio of the true improvement, $(f(x_x) - f(x_k + d_k))$ and the improvement predicted by the model $(m_k(0) - m_k(d_k))$

$$\rho_k = \frac{f(x_k) - f(x_k + d_k)}{m_k(0) - m_k(d_k)} \tag{4.14}$$

If this ratio is close to 1, we can consider the model to be relatively accurate in the surrounding area, thus decreasing the damping parameter $\lambda_k$ (and increasing the corresponding trust radius $\Delta_k$).

Consequently, we use **Algorithm 9**.

**Algorithm 9:** Levenberg-Marquardt algorithm

```
x = starting point;
λ = damping parameter;
while not converged do
    B = ∇²f(x) + λI;
    g = ∇f(x);
    d = argmin_d Bd + g;
    ρ = (f(x) − f(x+d)) / (m_x(0) − m_x(d));
    if ρ < 1/4 then
        λ = 3/2 λ
    else
        if ρ > 3/4 then
            λ = 2/3 λ
        end
    end
    x = x+d
end
```

### 4.2.3 Truncated Newton

Presented by Dembo and Steihaug in [17], the Truncated Newton method takes the approach that while it may not be convenient to exactly solve (4.8) for each step, such accuracy is perhaps not required. Instead we can use the CG method from Chapter 3 to solve (4.8) and iterate until some required level of accuracy is attained (hence 'truncated' as we do not complete the CG iteration). We can then make some step (the length of which can be governed by any of a number of conditions) in this direction and continue the Newton iteration.

They extend **Algorithm 7** (which we recall attempts to find the solution to $Ax = b$) to **Algorithm 10**.

We thus have two possibilities for returning a result. If the algorithm encounters a direction of negative curvature, we have, from §3 that the CG algorithm is not appropriate. We thus exit, returning either the latest approximation to the solution or, if this is the first step, the steepest descent direction. Rather than merely exiting when the curvature is negative, in practice we must

**Algorithm 10:** Truncated Conjugate Gradient Method

$\mathbf{x}_0 = 0$ ;
$\mathbf{r}_0 = \mathbf{d}_0 = \mathbf{b}$;
$\delta_0 = r_0^T r_0$ ;
**for** $k = 1$ ***to*** $N$ **do**
  **if** $d_{k-1}^T A d_{k-1} \leq \varepsilon(\delta_{k-1})$ **then**
    exit: $x = \begin{cases} d_0 \text{ if } k = 0 \\ x_{k-1} \text{ otherwise} \end{cases}$
  **else**
    continue
  **end**
  $\alpha_{k-1} = \frac{r_{k-1}^T r_{k-1}}{d_{k-1}^T A d_{k-1}}$;
  $x_k = x_{k-1} + \alpha_{k-1} d_{k-1}$;
  $r_k = r_{k-1} - \alpha_{k-1} A d_{k-1}$;
  **if** $\frac{||r_k||}{||b||} \leq \eta$ **then**
    exit: $x = x_k$
  **else**
    continue
  **end**
  $\beta_k = \frac{r_k^T r_k}{r_{k-1}^T r_{k-1}}$;
  $d_k = r_k + \beta_k d_{k-1}$;
  $\delta_k = r_k^T r_k + \beta_k^2 \delta_k$
**end**

require the curvature to be "sufficiently positive"[17], to avoid any potential rounding errors, and so we also exit similarly if $\frac{d^T A d}{r^T r} = \frac{d^T A d}{d^T d} \leq \varepsilon$ for some small $\varepsilon$.[1]

Otherwise we calculate the *relative residual* $\frac{||r_k||}{||b||}$ and terminate if this is smaller than some termination condition. This has the advantage of being scale invariant and computationally simple to calculate using values already required for the operation of the algorithm.

Setting aside the choice of $\varepsilon$, which by necessity is a function of the floating point precision limit of the machine, we must select an appropriate sequence of terminations conditions $\eta$ such that our outer Newtonian iteration correctly converges to a local minimizer in an appropriate time.

Dembo and Steighaus suggest the sequence $\eta_k = \min\{\frac{1}{k}, ||g(x_k)||^t\}$ for some $0 < t \leq 1$, which they show converges with order $1 + t$ [17, Theorem 2.3]. Nash [18] however notes that this is therefore no longer scale invariant and Eisenstat and Walker [19] propose some scale invariant options.

[1] It is possible that this misses some solutions which would be calculable with exact methods, however **a)** Such methods are likely too slow for practical use and **b)** We have already sacrificed some accuracy in favour of calculating a better approximate solution in more cases.

Martens[16] however notes that as CG optimises the quadratic

$$\phi(x) = \frac{1}{2}x^T Ax - b^T x$$

and suggests that it is therefore inappropriate to condition the termination on (4.8). He instead proposes a lookback condition, namely that the method be terminated at iteration $k$ if

$$k < i \quad \text{and} \quad \phi(x_k) < 0 \quad \text{and} \quad \frac{\phi(x_k) - \phi(x_{k-i})}{\phi(x_k)} < i \cdot \varepsilon$$

where $i$ is some varying lookback variable and $\varepsilon$ is some small positive value.

We can thus solve Newton's Method with **Algorithm 11**:

**Algorithm 11:** Truncated Newton Method

$x =$ starting point ;
**while** *True* **do**
  Use **Algorithm 10** to solve $H_n d_n = -\nabla f(x_n)$ ;
  Quit if some **stopping condition** reached ;
  Choose some step size $t$;
  Update $x \leftarrow x + t d_n$
**end**

## 4.3 Convergence of Newtonian Methods

### 4.3.1 The Traditional Method

Following [11], we assume that $f$, the function to be minimised, is twice differentiable, is continuous within some open neighbourhood of $x^*$ the optimal solution, that $\nabla f(x^*) = 0$ and also that $\nabla^2 f$ is positive semi-definite and Lipschitz continuous with Lipschitz constant $M$.[1]

From (4.2) we have that

$$\begin{aligned} x_{k+1} - x^* &= x_k - (\nabla^2 f(x_k))^{-1} \nabla f(x_k) - x^* \\ &= (\nabla^2 f(x_k))^{-1} (\nabla^2 f(x_k)(x_k - x^*) - (\nabla f(x_k) - \nabla f(x^*)) \end{aligned} \tag{4.15}$$

By Rolle's theorem, we have that

$$\nabla f(x+p) = \nabla f(x) + \int_0^1 \nabla^2 f(x+tp)^T p \, dt \tag{4.16}$$

and so substituting in $p = x_k - x^*$ we get that

$$\nabla f(x_k) - \nabla f(x^*) = \int_0^1 \nabla^2 f(x_k + t(x_k - x^*))^T (x_k - x^*) \, dt$$

[1] i.e. that $|f(y) - f(x)| \leq M|y - x|$ for all $x$ and $y$ in some open neighbourhood of $x^*$.

We can use this to find the norm of the second multiplicand of (4.15):

$$\begin{aligned}
&||\nabla^2 f(x_k)(x_k - x^*) - (\nabla f(x_k) - \nabla f(x^*))|| = \\
=&||\nabla^2 f(x_K)(x_k - x^*) - \int_0^1 \nabla^2 f(x_k + t(x_k - x^*))^T (x_k - x^*)\, dt|| \\
=&|| \int_0^1 \nabla^2 f(x_k)(x_k - x^*) - \nabla^2 f(x_k + t(x_k - x^*))^T (x_k - x^*)\, dt|| \\
=&|| \int_0^1 (\nabla^2 f(x_k) - \nabla^2 f(x_k + t(x_k - x^*))^T (x_k - x^*)\, dt||
\end{aligned}$$

By the triangle inequality

$$\begin{aligned}
\leq& \int_0^1 ||\nabla^2 f(x_k) - \nabla^2 f(x_k + t(x_k - x^*))||\, ||x_k - x^*||\, dt \\
=&||x_k - x^*|| \int_0^1 ||\nabla^2 f(x_k) - \nabla^2 f(x_k + t(x_k - x^*))||\, dt
\end{aligned}$$

By the Lipschitz condition

$$\begin{aligned}
\leq&||x_k - x^*|| \int_0^1 M||x_k - t(x_k - x^*) - x_k||\, dt \\
=&||x_k - x^*|| \int_0^1 tM||x_k - x^*||\, dt \\
=&\frac{1}{2}M||x_k - x^*||^2
\end{aligned}$$

We therefore have that

$$||x_{k-1} - x^*|| \leq \frac{1}{2}M||(\nabla^2 f(x_k))^{-1}||\, ||x_k - x^*||^2$$

From our condition, we have that there is some radius $r > 0$ where

$$||(\nabla^2 f(x_k))^{-1}|| \leq 2||(\nabla^2 f(x^*))^{-1}|| \quad \forall x_k \text{ such that } ||x_k - x^*|| \leq r$$

Thus inside this region

$$||x_{k+1} - x^*|| \leq M||(\nabla^2 f(x^*))^{-1}||\, ||x_k - x^*||^2$$

Define $M^* = M||(\nabla^2 f(x^*))^{-1}||$ (clearly a constant) and so

$$\frac{||x_{k+1} - x^*||}{||x_k - x^*||^2} \leq M^* \tag{4.17}$$

For all $x_0$ such that the above conditions hold and $||x_0 - x^*|| < \frac{1}{2M^*}$, we have that

$$\begin{aligned}
\frac{||x_1 - x^*||}{||x_0 - x^*||^2} &\geq 4M^{*2}||x_1 - x^*|| \\
&\text{thus} \\
4M^{*2}||x_1 - x^*|| &\leq M^* \\
||x_1 - x^*|| &\leq \frac{1}{4M^*}
\end{aligned}$$

Following this argument inductively, it is clear to see that within this neighbourhood the Newton iterates, $\{x_i\}$ converge quadratically to $x^*$.

### 4.3.2 Gauss-Newton

We can use a similar approach to that in §4.3.1. We have that

$$\begin{aligned} x_{k+1} - x^* &= (J^T J)^{-1}(x_k)\nabla f(x_k) - x^* \\ &= (J^T J)^{-1}(x_k)((J^T J)(x_k)\cdot(x_k - x^*) + \nabla f(x^*) - \nabla f(x_k)) \end{aligned} \tag{4.18}$$

Making use of (4.6) and (4.16) we can expand out $\nabla f(x_k) - \nabla f(x^*)$ to get that

$$\begin{aligned} &\nabla f(x_k) - \nabla f(x^*) = \\ &= \int_0^1 J^T J(x^* + t(x_k - x^*))\cdot(x_k - x^*)dt + \int_0^1 \sum_{j=1}^{m} r_j(x^* + t(x_k - x^*))\nabla^2 r_j(x^* + t(x_k - x^*))\cdot(x_k - x^*)dt \end{aligned}$$

For simplicity we define

$$H(x) = \sum_{j=1}^{m} r_j(x)\nabla^2 r_j(x)$$

We can thus write the norm of the second multiplicand of (4.18) as

$$\begin{aligned} &||(J^T J)(x_k)\cdot(x_k - x^*) - (\nabla f(x^*) - \nabla f(x_k)))|| = \\ &= ||(J^T J)(x_k)\cdot(x_k - x^*) - \int_0^1 J^T J(x^* + t(x_k - x^*))\cdot(x_k - x^*)dt + \int_0^1 H(x^* + t(x_k - x^*))dt|| \\ &\leq ||x_k - x^*||\cdot||\int_0^1 (J^T J(x_k) - J^T J(x^* - t)x_k - x^*))dt + \int_0^1 H(x^* + t(x_k - x^*))dt|| \end{aligned}$$

Assuming $J^T J(x)$ is Lipschitz with co-efficient $M$, we have that

$$||x_k - x^*||\cdot||\int_0^1 J^T J(x_k) - J^T J(x^* - t(x_k - x^*))dt|| \leq \frac{1}{2}M||x - x^*||^2$$

We therefore have that

$$\nabla f(x_k) - \nabla f(x^*) \leq \frac{1}{2}M||x_k - x^*||^2 + ||x_k - x^*||\cdot||\int_0^1 H(x^* + t(x_k - x^*))dt||$$

substituting back into (4.18) we get:

$$||x_k - x^*|| \leq \frac{1}{2}M(J^T J(x_k))^{-1}||x_k - x^*||^2 + ||(J^T J(x_k))^{-1}\int_0^1 H(x^* + t(x_k - x^*))dt||||x_k - x^*||$$

Which is approximately equal to

$$||x_k - x^*|| \leq M^*||x_k - x^*||^2 + ||(J^T J(x^*))^{-1}H(x^*)||||x_k - x^*|| \tag{4.19}$$

We can therefore see that if $||(J^T J(x^*))^{-1}H(x^*)||$ is small (and our assumption in using Gauss-Newton is generally that $||H(x^*)||$ *is* small) we have superlinear and almost quadratic convergence, and that if $||H(x^*)|| = 0$, we indeed have quadratic convergence.

### 4.3.3 Levenberg-Marquardt

If, for every iteration, each step taken is within the trust region, as stated in §4.2.2 each step will merely be that taken by the Gauss-Newton algorithm. Thus, if the algorithm converges at all, it must eventually converge with the superlinear convergance of the Gauss-Newton method. We therefore argue that if Levenberg-Marquardt converges at all, there must be some $n \in \mathbb{N}$ such that iterates $x_i, i \geq n$ converge superlinearly.

Analysis of L-M methods can be challenging, as they can vary greatly in their convergence depending on the method used to solve (4.10). Rather than finding a general solution to all of these, we find a simple solution, and use the bound given by this to argue that all better solutions must have convergence at least as rapid.

To give this approximation, we simply find the optimal solution along the path of steepest descent (as in **Algorithm 1**), which is inside the trust region i.e.

$$d_k^{est} = \underset{d_k}{\operatorname{argmin}} f(x_k) + g_k^T d_k, \quad ||d_k|| \leq \Delta_k$$

We can quickly solve this by travelling as far as possible in the direction $-g_k$, giving

$$d_k^{est} = -\frac{g_k}{||g_k||}\Delta_k$$

However, this would imply that the optimal point lies on the boundary of the trust region, which may not be the case. We thus in fact wish to find the value of $\tau \in (0,1]$ such that $d_k^C := \tau \cdot d_k^{est}$ minimises (4.11). Assuming that $B_k$ is positive definite[1] we have that $\tau$ is the solution to the positive quadratic optimisation problem

$$\tau = \underset{\zeta}{\operatorname{argmin}}\, m(-\zeta \frac{g_k}{||g_k||}\Delta_k) = -\zeta \frac{||g_k||^2}{||g_k||}\Delta_k + \frac{\zeta^2}{2}\frac{\Delta_k^2}{||g_k||^2} g_k^T B_k g_k$$

By quick differentiation we have that

$$||g_k||\Delta_k = \tau \left(\frac{\Delta_k}{||g_k||}\right)^2 g_k^T B_k g_k$$
$$\tau = \frac{||g_k||^3}{\Delta_k g_k^T B_k g_k}$$

However, we require that $\tau$ be in range, and so

$$\tau = \min\left(\frac{||g_k||^3}{\Delta_k g_k^T B_k g_k}, 1\right) \text{ i.e.}$$
$$d_k^C = -\left[\min\left(\frac{||g_k||^3}{\Delta_k g_k^T B_k g_k}, 1\right)\right] \cdot \frac{g_k}{||g_k||}\Delta_k$$

[1] We are able to assume this as we plan to use the positive definite hessian approximation from §4.2.1 in our calculation. It is also possible to show this for all Hessian matrices, see [11, Chapter 4].

The point $x_k + d_k^C$, known as the *Cauchy Point* [11, §4.2] certainly represents a decreasing step from $x_k$, although other methods are likely to present significantly improved results (specifically the CG method from §3, given that it begins with this direction as a first iterate). We can thus use this result to bound the convergence.

**Lemma 4.3.1.** *[11, Lemma 4.3] Given m as defined in (4.11), B a positive definite matrix, $\Delta > 0$, let $x + d^C$ be the Cauchy point, then*

$$m(d^C) - m(0) \leq -\frac{1}{2}||g|| \min\left(\Delta, \frac{||g||}{||B||}\right) \tag{4.20}$$

*Proof.* We prove this by taking two cases.

**Case 1.**

$$\frac{||g||^3}{\Delta g^T B g} \leq 1$$

Here we have that

$$d^C = -\frac{||g||^3}{\Delta g^T B g} \cdot \frac{g}{||g||}\Delta = -\frac{||g||^2 g}{g^T B g}$$

Substituting into (4.11), we have that

$$\begin{aligned} m(d^C) - m(0) &= -\frac{||g||^2||g||^2}{g^T B g} + \frac{1}{2}\left(\frac{||g||^2}{g^T B g}\right)^2 g^T B g \\ &= -\frac{||g||^4}{g^T B g} + \frac{1}{2}\frac{||g||^4}{g^T B g} \\ &= -\frac{1}{2}\frac{||g||^4}{g^T B g} \\ &\leq -\frac{1}{2}\frac{||g||^4}{||g||^2||B||} \\ &= -\frac{1}{2}\frac{||g||^2}{||B||} \end{aligned}$$

**Case 2.**

$$\frac{||g||^3}{\Delta g^T B g} > 1 \tag{4.21}$$

Thus

$$d^C = -\frac{g}{||g||}\Delta$$

We thus have

$$\begin{aligned} m(d^C) - m(0) &= -g^T \frac{g}{||g||}\Delta + \frac{1}{2}\left(\frac{\Delta}{||g||}\right)^2 g^T B g \\ &= \Delta||g|| + \frac{1}{2}\frac{\Delta^2}{||g||^2} g^T B g \end{aligned}$$

By (4.21) and the positive-definiteness of $B$ we have that

$$g^T Bg < \frac{||g||^3}{\Delta}$$

and so

$$\begin{aligned} m(d^C) - m(0) &\leq -\Delta||g|| + \frac{1}{2}\frac{\Delta^2}{||g||^2}\frac{||g||^3}{\Delta} \\ &= -\Delta||g|| + \frac{1}{2}\Delta||g|| \\ &= -\frac{1}{2}||g||\Delta \end{aligned}$$

Thus, as $\Delta$ and $\frac{||g||}{||B||}$ are both positive real values, we have that

$$m(d^C) - m(0) \leq -\frac{1}{2}||g|| \min\left(\Delta, \frac{||g||}{||B||}\right)$$

as required. □

We thus have a limit on the minimum possible decrease of the quadratic approximation $m$ in a round of Levenberg-Marquardt.

Following the examples of Nocedal[11] and Sorensen[20]:

**Theorem 4.3.2.** *Given $x_k$, the iterates of **Algorithm 9**, if $g_k$ is Lipschitz differentiable with constant $\gamma$ for any given k, $||B_k|| \leq \beta$, B positive definite, $\Delta, \beta, \gamma \in \mathbb{R}^+$, $f$ bounded below on the set $S = \{x | f(x) \leq f(x_0)\}$, then*

$$\liminf_{k\to\infty} ||g_k|| = 0$$

*Proof.* Assume that the theorem is false, i.e that the exists some $k \in \mathbb{N}$ such that for all $k \geq K$:

$$||g_k|| \geq \varepsilon$$

for some $\varepsilon \in \mathbb{R}^+$.

We will attempt to construct some $\overline{\Delta}$ such that we can be certain that if a step of **Algorithm 9** causes the trust radius to decrease, we can be certain that the radius was already at least $\overline{\Delta}$, i.e.

$$\Delta_{k+1} < \Delta_k \Rightarrow \Delta_k \geq \overline{\Delta}$$

If this is the case, we must have that for all $k$ where this is true:

$$\Delta_k \geq \min\left(\Delta_K, \eta\left(\frac{3}{2}\right)\overline{\Delta}\right) \tag{4.22}$$

where $\eta(x)$ is the corresponding multiplier for the change in trust radius if the dampening parameter is multiplied by $x$[1]. If $\Delta_k$ is greater than $\overline{\Delta}$, then it must be that $\Delta_{k+1} > \Delta_k$. Thus the

[1] recall that if $x > 1$, $\eta(x) < 1$

smallest possible radius may only occur on iterations directly after the radius has been reduced. The smallest radius from which we can reduce is $\overline{\Delta}$, and thus the smallest possible radius is $\eta(\frac{3}{2})\Delta$. However it is possible that we begin with some smaller radius, and so we include this for such a case.

Then, say there exists some index $K_1$ such that for all $k \geq K_1$ $\rho_k < \frac{1}{4}$. Then, by **Algorithm 9**, $\Delta_{k+1} < \Delta_k$, for all $k > K_1$, and so $\lim_{k\to\infty} \Delta_k = 0$. But this contradicts (4.22).

Thus, there must exist some infinite subsequence $\mathcal{K} \in \mathbb{N}^+$ such that $\rho_k \geq \frac{1}{4}$ for all $k \in \mathcal{K}$. By the definition of $\rho$ in (4.14) and **Lemmma 4.3.1**:

$$f_k - f_{k+1} \geq \frac{1}{8}\varepsilon \min\left(\Delta_k, \frac{\varepsilon}{\beta}\right)$$

However, $\varepsilon$ and $\beta$ are constants, $f_k$ is a Cauchy sequence and $f$ is bounded below. Thus

$$\lim_{k\to\infty} \Delta_k = 0$$

which again contradicts (4.22). Thus if $\overline{\Delta}$ exists, by contradiction there must exist no such $\varepsilon$. Thus

$$\liminf_{k\to\infty} ||g_k|| = 0 \tag{4.23}$$

□

**Lemma 4.3.3.** *The requirements for $\overline{\Delta}$ in **Theorem 4.3.2** are fulfilled by*

$$\overline{\Delta} = \min\left(\frac{1}{4}\frac{\varepsilon}{\frac{\beta}{2}+\gamma}, R_0\right)$$

*where $R_0$ is the radius of the largest open set containing $x_0$ on which $g$ is Lipschitz differentiable.*

*Proof.* By (4.14):

$$\begin{aligned} |\rho_k - 1| &= \left|\frac{f(x_k) - f(x_k + d_k) + m_k(p) - m_k(0)}{m_k(0) - m_k(d_k)}\right| \\ &= \left|\frac{m_k(d_k) - f(x_k + d_k)}{m_k(0) - m_k(d_k)}\right| \end{aligned}$$

By (4.16), we have that:

$$m_k(d_k) - f(x_k + d_k) = \frac{1}{2}d_k^T B d_k - d_k^T \int_0^1 B(x_k + t d_k)^T d_k dt$$

If $||B_k|| \leq \beta$ and $\nabla f$ has Lipschitz constant $\gamma$, we this have that

$$|m_k(d_k) - f(x_k + d_k)| \leq \frac{1}{2}\beta||d_k||^2 + \gamma||d_k||^2$$

Using the result from **Lemma 4.3.1**, we thus have that

$$|\rho_k - 1| \leq \frac{\left(\frac{\beta}{2} + \gamma\right)||d_k||^2}{\frac{1}{2}||g_k|| \min\left(\Delta, \frac{||g_k||}{||B_k||}\right)}$$
$$\leq \frac{(\frac{\beta}{2} + \gamma)\Delta_k^2}{\frac{1}{2}\varepsilon \min\left(\Delta_k, \frac{\varepsilon}{\beta}\right)}$$

By inspection $\overline{\Delta} < \frac{\varepsilon}{\beta}$, thus, if $\Delta < \overline{\Delta}$, transitively, $\Delta < \frac{\varepsilon}{\beta}$. We thus have that

$$|\rho_k - 1| \leq \frac{\left(\frac{\beta}{2} + \gamma\right)\Delta_k^2}{\frac{1}{2}\varepsilon\Delta_k} = \frac{\left(\frac{\beta}{2} + \gamma\right)\Delta_k}{\frac{1}{2}\varepsilon}$$
$$\leq \frac{\left(\frac{\beta}{2} + \gamma\right)\overline{\Delta}}{\frac{1}{2}\varepsilon}$$
$$\leq \frac{1}{4} \cdot \frac{2 \cdot \left(\frac{\beta}{2} + \gamma\right)}{\varepsilon} \cdot \frac{\varepsilon}{\left(\frac{\beta}{2} + \gamma\right)} = \frac{1}{2}$$

Thus $\frac{1}{2} \leq \rho_k \leq \frac{3}{2}$, i.e. $\rho > \frac{1}{4}$. Therefore by **Algorithm 9**, $\lambda_{k+1} \leq \lambda_k$, i.e. $\Delta_{k+1} \geq \Delta_k$ if $\Delta_k < \overline{\Delta}$. □

We therefore have that by (4.23), the gradient exhibits at least linear convergence. By the previous arguments, we have that sequence will likely exhibit faster convergence than this, giving superlinear and possibly almost quadratic convergence.

### 4.3.4 Truncated Newton

In [21], Dembo et al. give a series of results for the convergence of general 'Inexact Newton Methods' (that is, any method which finds an approximation of the solution to $Hd = -g$ each iteration rather than solving exactly for $d$). Specifically they show that

**Theorem 4.3.4.** *[17, Theorem 2.3] Let $x_k \to x^*$ where $H(x^*)$ is positive definite, and assume that $H$ is Lipschitz continuous at $x^*$. Then*

1. *$x_k \to x^*$ superlinearly if and only if $\frac{||r_k||}{||g(x_k)||} \to 0$ as $k \to \infty$*
2. *$x_k \to x^*$ with order $(1+t)$ if and only if*

$$\limsup_{k \to \infty} \frac{||r_k||}{||g(x_k)||^{1+t}} < \infty$$

The proof itself is rather long and technical, and so we omit it here, however we note a corollary that for some sufficiently small positive real value $\varepsilon$ and large integer $i$, if

$$\varepsilon \leq \min\left(\frac{||g_k||}{i}, ||g_k||\right)^2 \tag{4.24}$$

for all $k > i$, then the method will converge superlinearly.

# 5. Truncated Newton For Deep Learning

## 5.1 Advantages of Truncated Newton

In [16], Martens brings together the techniques in Chapter 4, and discusses applying them to deep learning. He discusses the (then) current state of deep learning techniques, and notes that while gradient descent methods are overwhelmingly the most popular choice as optimisation function, it is well understood that they are not always effective. He argues that some objective functions "exhibit pathological curvature making them nearly impossible for curvature-blind methods like gradient-descent to successfully navigate". A well known example (often used to test potential optimiser functions) is the Rosenbrock function (**Figure 5.1**), which shows a significant improvement when optimising using Newton's method over gradient descent.[1]

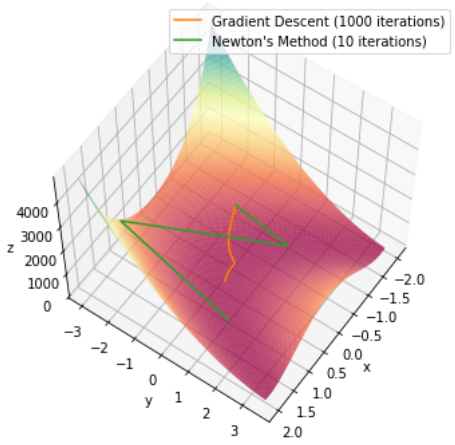


**Figure 5.1:** The Rosenbrock function $f(x,y) = 100(y-x^2)^2 + (x-1)^2$ exhibits pathological curvature across the valley. While there exists a global minimum at $(1,1)$, having entered the valley, traversing around the shallow area proves difficult. We can see that Newton's Method is able to navigate by moving further in areas of low curvature, whereas Gradient Descent moves extremely slowly through this valley.

This is not such an unreasonable example. As Martens notes, it is frequently the case that in training, the inner layers of a deep network can be difficult to optimise. The outer layers have a large effect on the resultant error function, and so if their gradient becomes relatively

[1] It should be noted that not all problems are improved by considering the curvature. For example, the Rastrigian function $f(x,y) = 30 + x^2 + y^2 - 10(\cos(2\pi x) - \cos(2\pi y))$ is difficult to optimise because it has a large number of local minima, i.e. is highly non-convex. This is not solved merely by considering the curvature.

significantly larger or smaller than the gradients of the inner layers, in order to make appropriate progress the learning rate must be changed, often in such a way as to invalidate the contribution of the inner layers to the eventual direction. Newton's method avoids this issue by being scaling invariant, and thus correctly considering the changes brought about by the inner layers, irrespective of their gradients' relative sizes.

### 5.1.1 Scaling Invariance

One of the particular advantages of the Newton Method over Gradient Descent is its scaling invariance, i.e. the fact that if the function it is optimising is scaled up or down, the method will still find an optimal solution using a scaled version of the same steps, and in the same time.

**Theorem 5.1.1.** *Newton's method is scaling invariant, i.e. for $f$ a twice-differentiable function $\mathbb{R}^n \to \mathbb{R}$ and $A$ an invertable $n \times n$ matrix, if $\phi(y) = f(Ay)$, $y_0 = A^{-1}x_0$, $x_0 \in^n$, and $x_k$ and $y_k$ are the kth iterates of **Algorithm 8** for functions, $f$ and $\phi$ respectively, then $y_k = Ax_k$.*

*Proof.* We will prove this by induction. By assumption the base case is already true, so we need only show the induction step.

From **Algorithm 8** we have that

$$y_{k+1} = y_k - \nabla^2\phi(y_k)^{-1}\nabla\phi(y_k) \tag{5.1}$$

$$\begin{aligned} \text{As } \phi(y) &= f(Ay) \\ \nabla\phi(y) &= A^T\nabla f(Ay) \\ \text{and } \nabla^2\phi(y) &= A^T\nabla^2 f(Ay)A \end{aligned}$$

Thus, substituting into (5.1) we have that

$$\begin{aligned} y_{k+1} &= y_k - \left(A^T\nabla^2 f(Ay_k)A\right)^{-1}A^T\nabla f(Ay_k) \\ &= y_k - A^{-1}(\nabla^2 f(Ay_k))^{-1}\nabla f(Ay_k) \\ \text{thus } Ay_{k+1} &= Ay_k - (\nabla^2 f(Ay_k))^{-1}\nabla f(Ay_k) \end{aligned}$$

By the induction hypothesis

$$\begin{aligned} Ay_{k+1} &= x_k - (\nabla^2 f(x_k))^{-1}\nabla f(x_k) \\ \text{i.e. } Ay_{k+1} &= x_{k+1} \end{aligned}$$

as required. □

This is not the case for simple gradient descent. From **Algorithm 1**, we have that

$$x_{k+1} = x - \nabla f(x)$$

Using the same notation as for **Theorem 5.1.1** we have that

$$y_{k+1} = y_k - A^T\nabla f(Ay)$$

we thus have that the update direction at each stage is not similarly scaled (i.e. it's scaled by $A^T$ rather than $A$) and so there is no direct relationship between the directions for the scaled functions.

From this we can infer that Newton's method does not need to concern itself with finding an ideal learning rate for a particular problem, a significant difficulty faced when using gradient descent methods. Finding an appropriate learning rate can be a time consuming process, and there is no guarantee that a learning rate which has performed well from a particular initial point will give a good convergence from another. The work done by the later algorithms in §2 is instead handled for us by the method itself, and less pre-conditioning of the data will be required, as we do not have to consider the particular size of features in our domain.

### 5.1.2 The Hessian Free Method

As discussed above, Newton's method itself is not appropriate for deep learning. The number of parameters means that storing, let alone calculating the inverse of the Hessian matrix is impractical. If we wish to get the benefits, we must instead use some simpler-to-calculate alternative. Truncated Newton (**Algorithm 11**) replaces the difficult act of calculating the exact inverse of the Hessian with using the conjugate gradient method to calculate an appropriate approximation. This has the particular advantages that *a)* We need only calculate the product $H_i d_k$ for a given round $i$ of the iteration rather than the significantly more memory-heavy inverse, and *b)* CG generally converges to a good approximation quickly, and so the number of these calculations we require is also likely to be small. While we could calculate these products using the finite differences method, we can instead solve this problem while gaining further computational advantages by making use of some of the other methods from §3.

### 5.1.3 Adaptations

Martens notes that the standard Truncated Newton implementations are not sufficient to train neural networks. Neural networks optimise over extremely high-dimensional and complex domains, and so some further improvements need to be added.

Firstly, we encounter the difficulty that the CG method assumes a positive definite Hessian matrix. It would be possible to use some other method if the Hessian was not positive definite, but this would require testing the Hessian each time, and the possibility of the frequent use of another, complex method. Instead we can take the Gauss-Newton approximation from §4.2.1, $G = J^T H_L J \approx H$. As noted above, this is definitely positive definite, and has good convergence close to a local optimum. (If we are not close to a local optimum **Algorithm 11** will restart and try for a better approximation).

Secondly, it is important to consider that **Algorithm 11** takes a quadratic approximation to the error function and attempts to optimise this. The function itself however is generally very much not quadratic, and so it is quite possible that any step we take will take us well away from the area in which this is an appropriate approximation. Although the Newton method's long steps in areas of low curvature is generally helpful, the sheer unpredictability of the function means that we should not hold too great a hope in any estimation of the gradient at a particular point. Rather we can use the Levenberg-Marquardt approach from §4.2.2 to impose a trust

region on the solution. In combination with the above we thus instead approximate the hessian at each iteration with $H \approx \hat{G} = J^T H_L J + \lambda I$. As discussed above, Martens suggests that we use the simple update heuristic:

$$\lambda_i = \begin{cases} \frac{3}{2}\lambda_{i-1} \text{ if } \rho < \frac{1}{4} \\ \lambda_{i-1} \text{ if } \frac{1}{4} \leq \rho \leq \frac{3}{4} \\ \frac{2}{3}\lambda_{i-1} \text{ if } \rho > \frac{3}{4} \end{cases}$$

in order to avoid overcomplicating the computation.

Even with these adaptations, it is unlikely that we will be able to perform effective optimisations using a whole testing set at once. Instead it is preferable to compute the steps using batches. Here the recommendation is to use a single large batch to calculate the vectors $\hat{G}d$, but to use the entire test set to calculate the loss function and gradient (the latter two only being calculated once per iteration). The hope here is that the decreased number of iterations allows us the freedom to compute more cycles, and so to make use of the whole dataset.

We can also attempt to take some inspiration from **Algorithm 2**. It seems likely that if a descent direction was valid from our previous position, is it likely to still have some value after an iteration (especially as each full CG iteration does not utilise all of the test set). We can thus begin the next round of CG iterations with the previous descent direction. If we are correct in our assumption, we will require fewer CG iterations to find an acceptable convergence.

We can thus expand **Algorithm 11** to **Algorithm 12**, taking into account the Gauss-Newton approximation and the Levenberg-Marquardt dampening. This gives the advantage of a scaling invariant algorithm which is able to take into account the curvature information from the Newton Method while not being restricted by the difficulty of computing the entire inverse hessian.

**Algorithm 12:** NewtonCG Method

$x =$ starting point ;
$\lambda =$ damping parameter;
**while** *True* **do**
  $B_n = J_n^T J_n(x) + \lambda I$;
  Use **Algorithm 10** to solve $B_n d_n = -\nabla f(x_n)$ ;
  $\rho = \frac{f(x) - f(x+d)}{f(x) - m_x(x)}$ ;
  Update $\lambda$ according to heuristic;
  Quit if some **stopping condition** reached ;
  Choose some step size $t$;
  Update $x \leftarrow x + t d_n$
**end**

# 6. Experiments

## 6.1 Methodology

We used Quan and Lin's Python implementation of SimpleNN[1] in order to compare the optimisation methods. SimpleNN implements two convolutional neural networks, $CNN_4$ and $CNN_7$ in TensorFlow, and provides an implementation of Martens' Truncated Newton method for optimisation, along with those pre-packaged in TensorFlow. The architectures of $CNN_4$ and $CNN_7$ are shown in **Tables 6.1** and **6.2**.

| | Input | |
|---|---|---|
| Layer 1 | Convolutional | $32 \times 5 \times 5$ filters<br>ReLu Activation |
| | Pool | Max pool, $2 \times 2$ filters with step 2 |
| Layer 2 | Convolutional | $64 \times 3 \times 3$ filters<br>ReLu Activation |
| | Pool | Max pool, $2 \times 2$ filters with step 2 |
| Layer 3 | Convolutional | $64 \times 3 \times 3$filters<br>ReLu Activation |
| | Pool | Max pool, $2 \times 2$ filters with step 2 |
| Layer 4 | Fully Connected | Mean Squared Error Loss |

**Table 6.1:** Architecture of $CNN_4$

SimpleNN largely implements the Hessian-Free method described in [16], (**Algorithm 12**), however choosing to truncate the conjugate gradient iteration on the simpler heuristic of $||r_i|| < \varepsilon$ or $i > i_{\max}$ ($i$ the iteration number) for some fixed $\varepsilon$. Here we can make use of the inequality (4.24) to to argue that this is largely sufficient to guarantee convergence. For any choice of $\varepsilon$, this condition will eventually be violated, and this presents a trade-off between the accuracy of the end result and the speed of each iteration, which we accept.

Following the example from [9], we initially ran short tests (25 epochs[2]) for each of the gradient descent methods on a logarithmic grid of learning rates to gain an estimate of the optimal rate for each data set. We initially tested rates close to the default rate for each set in TensorFlow, testing

$$\{0.025, 0.05, 0.1, 0.25, 0.5\} \times 10^n$$

[1] https://github.com/cjlin1/simpleNN

[2] After one epoch the network has seen each item of the test set at least once.

| | Input | |
|---|---|---|
| Layer 1 | Convolutional | $32 \times 5 \times 5$ filters<br>ReLu Activation |
| Layer 2 | Convolutional | $32 \times 3 \times 3$ filters<br>ReLu Activation |
| | Pool | Max pool, $2 \times 2$ filters with step 2 |
| Layer 3 | Convolutional | $64 \times 3 \times 3$ filters<br>ReLu Activation |
| Layer 4 | Convolutional | $64 \times 3 \times 3$ filters<br>ReLu Activation |
| | Pool | Max pool, $2 \times 2$ filters with step 2 |
| Layer 5 | Convolutional | $64 \times 3 \times 3$filters<br>ReLu Activation |
| Layer 6 | Convolutional | $128 \times 3 \times 3$filters<br>ReLu Activation |
| | Pool | Max pool, $2 \times 2$ filters with step 2 |
| Layer 7 | Fully Connected | Mean Squared Error Loss |

**Table 6.2:** Architecture of $CNN_7$

where $n$ causes the default rate to be in range, and selecting the rate giving the smallest loss after the test run. If the most successful learning rate was at the edge of the search space, we ran further tests in that direction in order to ensure we had selected the correct rate. While this is not a perfect selection process (in particular, methods easily affected by initial conditions may give false reports if they are run on a poor seed), the testing of multiple rates helps to identify a pattern and gives a good chance of success. The default rates along with the rates chosen can be found in Appendix A.1. Both SGD with Momentum and Nesteyov methods were run with a momentum parameter of 0.9.

Each gradient optimiser was then run for 250 epochs on each network on each data set in order to allow each run to come close to a solution. If, on examining the results, it appeared that any run had yet to either diverge or settle close to a critical point, we would have resumed training the model. After each batch we logged the test accuracy, the accuracy on the training set, and the time taken to process the batch.

The Quasi-Newtonian optimiser was run for 150 iterations, using a random selection of 1000 training items each iteration to run the CG method. Similarly to the gradient optimisers, we logged the test accuracy, training accuracy and time taken for each iteration.

All implementations were run on an NVIDIA Geforce GTX 1060 with 6GB of RAM.

### 6.1.1 Data Sets

The data sets were chosen to show the capabilities of the optimisers, providing similar tasks, but with varying degrees of complexity, without being so complex that it was impractical to run repeated experiments. The MNIST dataset [22] is frequently used as a baseline to show the

functionality of Neural Networks. It consists of 70 000 centred, greyscale images of handwritten digits, 60 000 training images and 10 000 test. Each test item is a $28 \times 28$ pixel image, along with the correct corresponding digit. The images themselves contain few features, only a single layer, and no distracting information beyond the number themselves.

The CIFAR10 [23] data set consists of a collection of 60 000 $32 \times 32$ colour images, divided between 10 categories. This data is much more complex than MNIST, as the categories themselves are more abstract (e.g. dog, cat, ship), of varying distinctiveness (small images of dogs can be very difficult to distinguish from small images of cats), and the variance within the category can also vary greatly, (an image of a ship may be from above, directly from the front, or at an oblique angle from the side for example, thwarting any attempt to define a category merely by simple shape recognition for example). While it is still possible to gain good results on the MNIST dataset by using PCA or nearest-neighbour analysis, the results for these techniques on CIFAR10 tend to be poor.

## 6.2 Results

Each network was trained using each optimiser five times using a random initialisation each time. We have graphed the mean results for each training method as a solid line, with a shaded area representing plus or minus one standard deviation. For each data set and network, we present the training and testing accuracy for the gradient descent methods, as well as the Quasi-Newtonian method. As they use different methods of loading the data, their iterations are not directly comparable, however from a practical standpoint it is useful to compare their speed of convergence. Therefore we finally present all five runs of both the Quasi-Newtonian method alongside the most successful gradient descent method, tracking their changes in accuracy on the test data set against time.

### 6.2.1 MNIST

As noted above, MNIST is not a complex data set, we would expect that if both our network and optimiser function correctly, that they will be able to correctly identify the images without difficulty, and with a good general solution. Here we have truncated the results in figures 6.1, 6.2 and 6.3. While we trained the networks for significantly longer than displayed, they converged quickly, and the accuracy did not change significantly outside the graphed area. **Figure 6.3** displays the same range of data as figures **6.1** and **6.2** (5 epochs of ADAM and 50 iterations of NewtonCG).

#### $CNN_4$

We can see that, as expected, all of the optimisers perform similarly well. They are each able to consistently find a good solution which generalises well to the test set. The good generalisation should be expected. While for all good data sets, the training data is similar to the test set, this is particularly true for MNIST as each image has so few features. We can see that the NewtonCG method also performs around as well as ADAM, finding as good a solution in a slightly faster time.

*$CNN_7$*

Here we see similar results to the previous test. The gradient descent algorithms take slightly longer to attain maximal accuracy, likely due to the increased complexity, however the majority of the graphs are the same. One run of the SGD optimiser failed to converge until Epoch 35 (the full graph with convergence is shown in the appendix in **Figure A.1**), which explains the poor convergence and large standard deviation in **Figure 6.4**. One run also failed to converge at all and was excluded from the results. It is likely that this occurred because the learning parameter is well tuned to some parts of the domain, but not to others (i.e. that different part of the domain have different scaling). Thus for some initial states, convergence occurs quickly, however for others SGD slowly travels around the domain until it arrives at a location which is correctly scaled and concave. The adaptive methods on the other hand are able to mitigate this difficulty somewhat.

The NewtonCG optimiser again performs well. It still optimises quickly and to a good and generalisable result, however we can see from **Figure 6.6**, which also plots 5 Epochs of ADAM and 50 iterations of NewtonCG, that it is significantly slower. While it still generally converges faster, it takes three to four times as long as on $CNN_4$ to complete the same number of iterations (between 1199 and 1769 seconds compared to 150 - 430 seconds for $CNN_4$), whereas ADAM increases from taking 370-387 seconds to taking 714-730 seconds to complete 5 epochs. The large range of times can be attributed to the number of CG iterations each run completed. Especially when the network was close to being converged, the optimiser would be required to run a large number of CG iterations to find a small decrease in loss. For networks for which we can be less certain of the expected accuracy, this increase in time, and the potential variance could present a difficulty. While this could be alleviated by decreasing the maximum possible number of CG iterations, this would in turn significantly increase the number of Newton Iterations required for convergence.

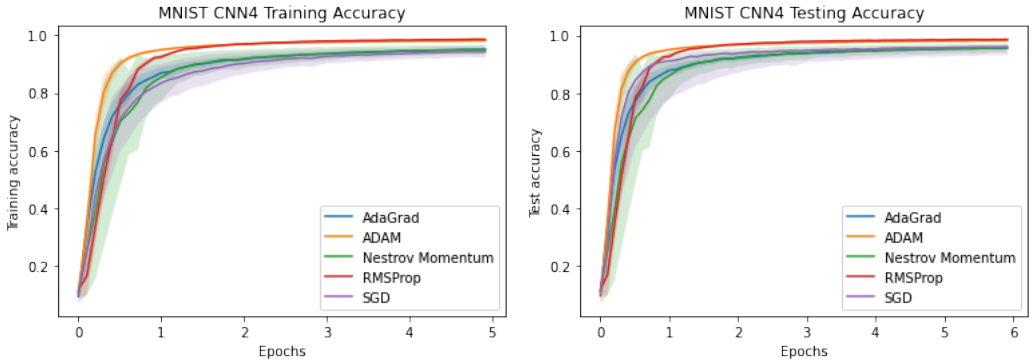


**Figure 6.1:** Results for training the $CNN_4$ network using gradient descent methods

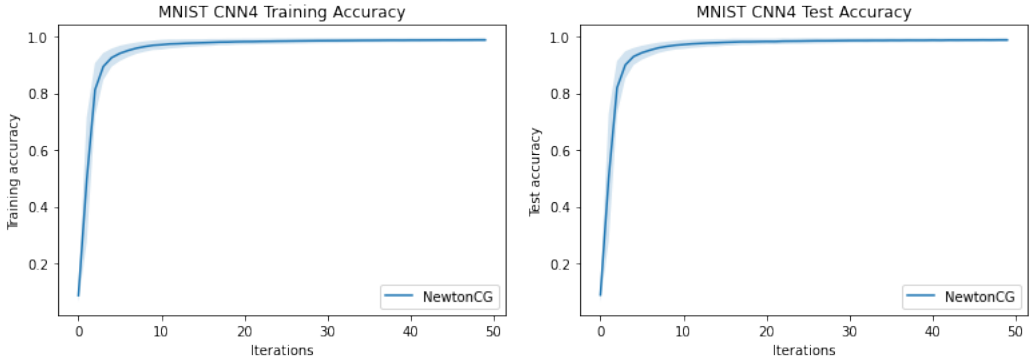


**Figure 6.2:** Results for training the $CNN_4$ network using the NewtonCG method

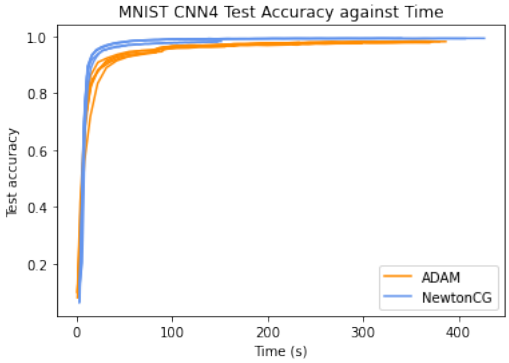


**Figure 6.3:** Comparison of the most successful gradient decent method (ADAM) against the NewtonCG method over time on $CNN_4$

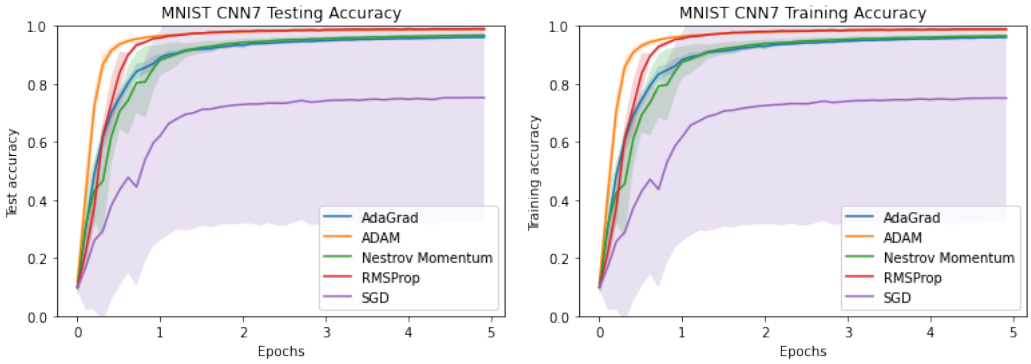


**Figure 6.4:** Results for training the $CNN_7$ network using gradient descent methods

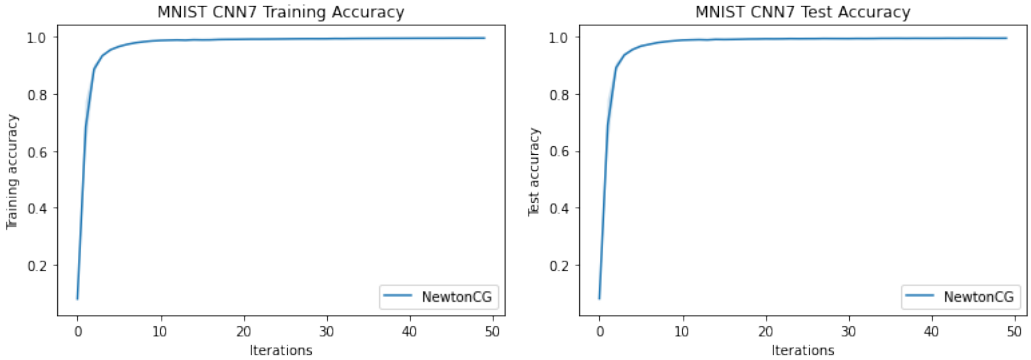


**Figure 6.5:** Results for training the $CNN_7$ network using the NewtonCG method

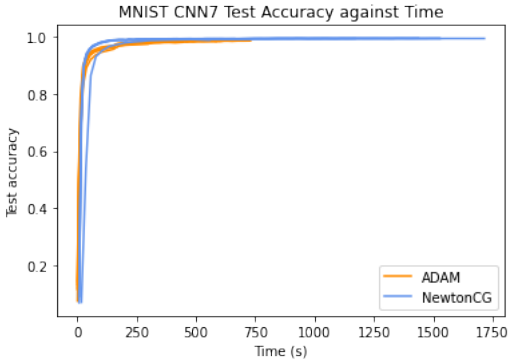


**Figure 6.6:** Comparison of the most successful gradient decent method (ADAM) against the NewtonCG method over time on $CNN_7$

### 6.2.2 CIFAR10

*$CNN_4$*

It is first important to note that each of the gradient descent training graphs shows fifty times as many epochs as those for MNIST. As predicted, CIFAR10 is a more complex problem, and requires significantly more training in order for the network to settle on successful weights. We can see that for all training methods there is an initial period of fast convergence followed by a long tail of slow convergence. Again, ADAM and RMSProp significantly outperform the three other gradient descent methods, and all adaptive methods give more consistent results. The results on the test set are significantly worse than those on the training set, although this difference is not particularly marked for any particular optimiser. It is to be expected that, given the variety of images present in each category, that any result will be less generalisable than those for MNIST.

In **Figure 6.9** we compare the time and accuracy of 15 epochs of RMSProp against 150 iterations of NewtonCG. We can see that the Newton method converges significantly faster as well as significantly more consistently, not displaying the noise present in the RMSProp iterates.

*$CNN_7$*

Here we start to see the benefits of using a larger network. Every algorithm trains more slowly than it did on the smaller network, however it converges to a more accurate solution. The more complex network is able to identify a greater number of features, and its greater level of abstraction allows for to find greater similarities between the more disparate images.

It is interesting to note however that the training accuracy continues to increase well past the point at which the testing accuracy plateaus. This is certainly some overfitting on the training dataset, however interestingly it has not brought on a corresponding decrease in accuracy on the testing set.

We can also see on both this and the previous network that the non-adaptive methods, while having a greater range of results, appear to converge more consistently.

Again, NewtonCG converges faster in real time, although each run takes a less consistent length of time. ADAM eventually converges to a similar level of accuracy, however it takes over an hour longer, and the accuracy is much less stable.

Interestingly, these results do not match those from Wilson et al. [9] (noted in §2.3.5), and in fact show adaptive methods as having significantly better convergence and generalisability. It is possible that this is due to the design of the network (they ran their experiment on a similarly designed but deeper and wider network)[1], and it is possible that there is some threshold of network complexity or particular matching of network and data set which causes non-adaptive methods to perform better than adaptive (they chose to use previously tested well performing networks specific to each data set, rather than consider the functionality of generic networks). This is a difficult consideration. See §6.4 for further discussion.

[1] see http://torch.ch/blog/2015/07/30/cifar.html

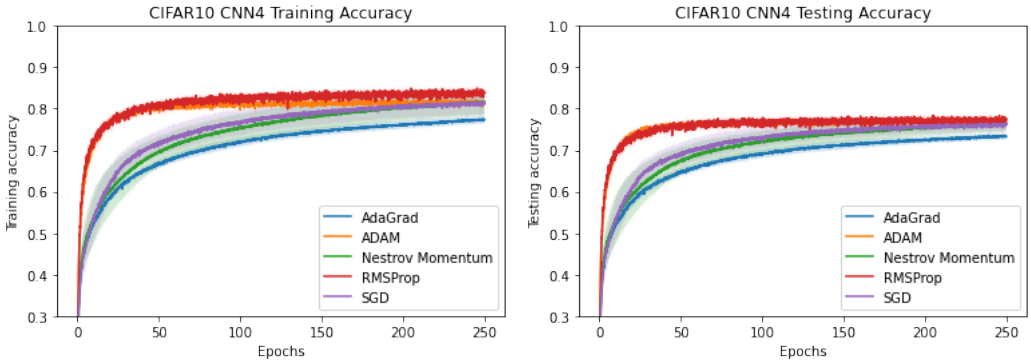


**Figure 6.7:** Results for training the $CNN_4$ network using gradient descent methods

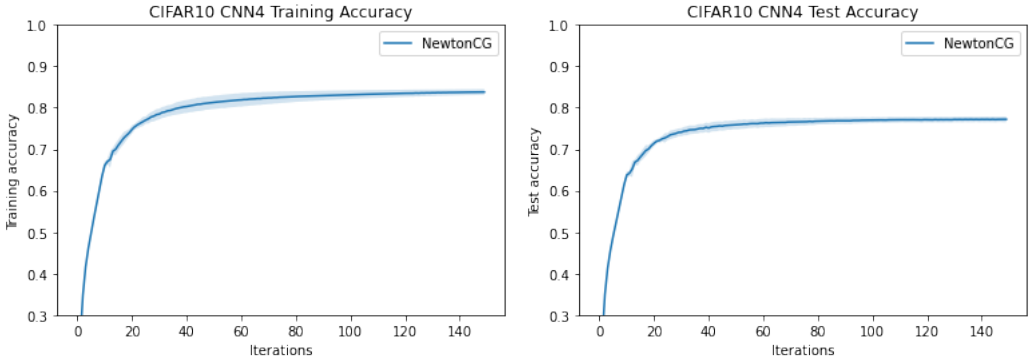


**Figure 6.8:** Results for training the $CNN_4$ network using the NewtonCG method

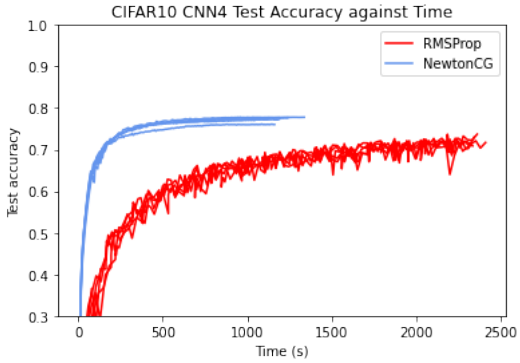


**Figure 6.9:** Comparison of the most successful gradient decent method (RMSProp) against the NewtonCG method over time on $CNN_4$

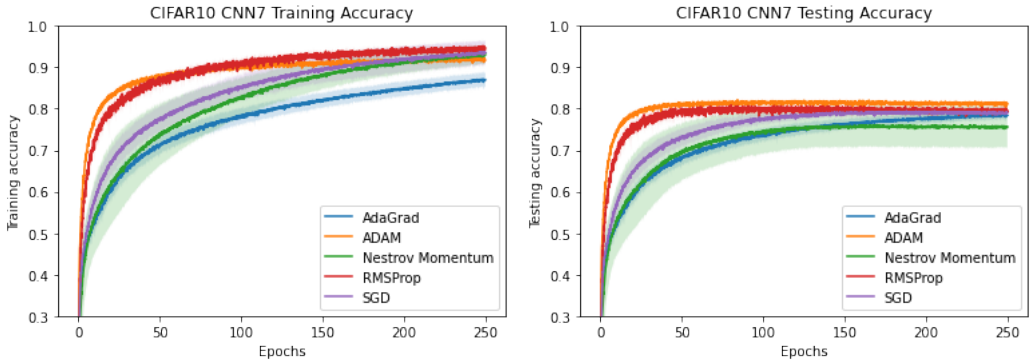


**Figure 6.10:** Results for training the $CNN_7$ network using gradient descent methods

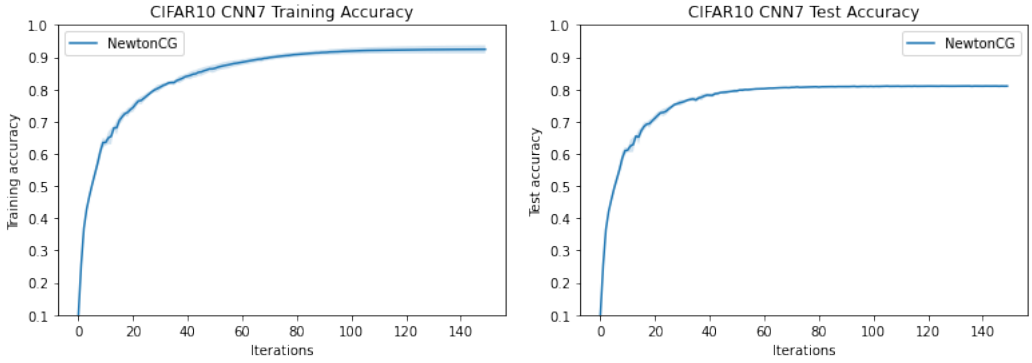


**Figure 6.11:** Results for training the $CNN_7$ network using the NewtonCG method

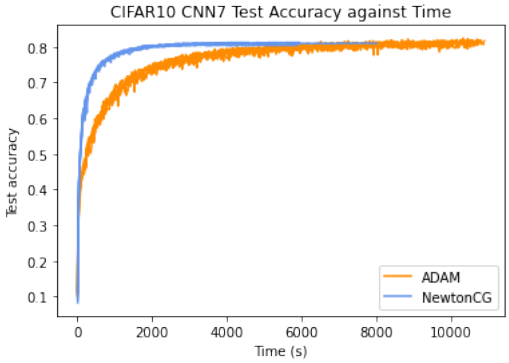


**Figure 6.12:** Comparison of the most successful gradient decent method (ADAM) against the NewtonCG method over time on $CNN_7$

## 6.3 Conclusion

The NewtonCG method represents a practical alternative to the standard Gradient Descent methods as an optimisation method for at least small-scale deep learning problems. It is able to take advantage of estimates of local curvature to make improved guesses at both appropriate step direction and length not available to even adaptive GD methods.

In particular, it is possible to gain good or even better results than for GD methods without the parameter tuning they require. As with batch gradient descent, memory limitations still require that we make some sacrifices in the quantity of training data we are able to include in each iteration, and we thus still require a hyperparamater to control this. This is likely to remain a requirement for any optimisation algorithm however, and will allow adaptation for an online approach[1]. More generally the method provides us with scaling invariance, which will allow us both this simplicity of setup, but also hints at the possibility to make use of learning from previous similar data sets help future optimisation. This is something which is particularly difficult for GD methods, due partly to their scaling invariance.

Our experiments show that, at least for small scale problems, the method is able to find an effective solution at least as quickly as the most effective GD methods, and that such a solution is likely to be both consistent and well generalisable. However, this can rely on each iteration having a lot of information about the dataset. Convergence will decrease as the the proportion of samples included in the CG iteration decreases.

A particular disadvantage is in the predictability of running time. Even among these small examples, although most training times were similar, there were several large outliers, generally caused by a large number of CG iterations being run. For use cases in which consistent training time is an important factor, possibly those where computing time is particularly expensive, this could present a large drawback.

NewtonCG is also not currently well implemented in standard machine learning libraries. Currently PyTorch does not natively implement Conjugate Gradient at all, while in Tensorflow it exists only as an experimental algorithm. The full algorithm is not available as a package for either library, and an efficient implementation would likely require additional C and/or CUDA packages in order to be efficient. It would be a significant undertaking to build an efficient general purpose optimiser, and this makes it a difficult choice for non-technical users, or those who do not wish to maintain their own optimiser code.

In general while being well understood as a general optimisation method, and while increasingly being considered in the machine learning field, the NewtonCG method is not as well understood as GD methods. It is likely that there are still further computational optimisations to be found, and we still do not well understand what optimisations we may be missing by excluding information in our approximations. As Martens notes in [15], ”optimisation theory has a long way to go before being able to predict the performance of a method like [NewtonCG]”.

[1] Online learning is any method in which rather than accessing the data as a whole, we access it in sequential chunks

## 6.4 Future Developments

We would have liked to have tested NewtonCG on larger data sets and a larger variety of network structures. We were limited by time, the networks take a long time to train and analyse, and so it was not feasible to test large networks or data sets in any repeatable manner. Generally other analyses have also focused on small convolutional networks, and in a field in which optimisation can be so unpredictable and contain so many confounding factors, it would be useful to investigate a series of larger, more complex and varied problems in order to better asses the optimiser.

Ideally we would have liked to have implemented a more general purpose optimiser for pyTorch and/or TensorFlow. SimpleNN is limited in its capabilities (another reason why we were unable to run tests on a wider range of data sets), and although we were able to modify it to fit our needs, it is not as flexible as the built-in optimisers. However, this proved to be more complex than we had anticipated. A general optimiser would allow for more flexible testing, as well as for the optimier to be more accessible to the public. Some attempts have been made at this in the past, e.g. [24], however they are generally not well maintained, or are still challenging to implement.

As noted for in results for [9], it is difficult to consider a problem merely in a single dimension. It would be advantageous to consider a wider variety of types of problem, as well as a wider variety of network designs and types. The science of constructing an appropriate network to a particular problem is still not well understood, and the particular effect of changing any specific network parameter, be it the depth, the activation functions or the optimisation function is still not generally known. It is possible (even likely) that this is too difficult a problem, and that we will find that it is not possible to give a good general overview of what is likely to be an effective network for a particular purpose ahead of time, but further analysis would be helpful.

Currently we rely on a general understanding of 'similar' data sets to decide on architecture and learning methods for networks, however it would be beneficial to be able to better codify this. A greater understanding of the metric spaces of large data sets may allow us to better perform this analysis. This is again likely to be a trade-off between the possibility of performing rigorous mathematical analysis and the time required to perform calculations however.

# A. Additional data

## A.1 Learning rates for each optimiser

| Optimiser | Default learning rate | MNIST | | CIFAR10 | |
|---|---|---|---|---|---|
| | | $CNN_4$ | $CNN_7$ | $CNN_4$ | $CNN_7$ |
| AdaGrad | 0.001 | 0.01 | 0.005 | 0.025 | 0.01 |
| ADAM | 0.001 | 0.0025 | 0.0005 | 0.0025 | 0.001 |
| Nesteyov | 0.01 | 0.005 | 0.0025 | 0.005 | 0.005 |
| RMSProp | 0.001 | 0.0005 | 0.0005 | 0.001 | 0.0005 |
| SGD | 0.01 | 0.005 | 0.005 | 0.01 | 0.005 |

**Table A.1:** Default and selected learning rates for experiments in §6

## A.2 Extended graphs

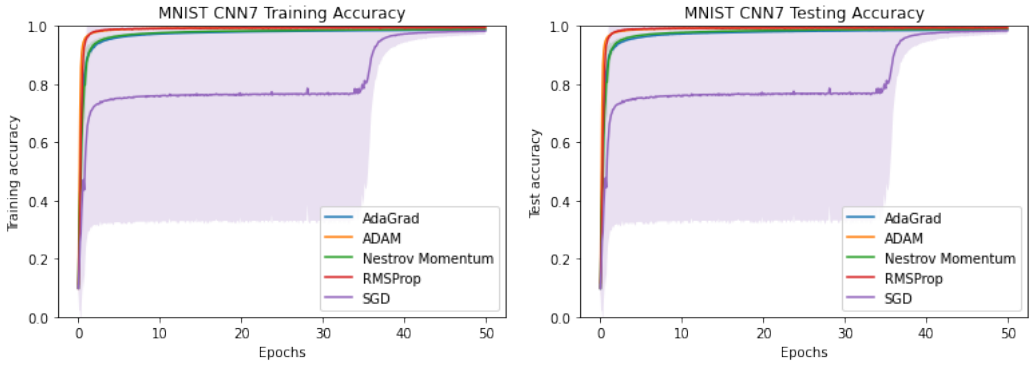


**Figure A.1:** Results for training the $CNN_7$ network using gradient descent methods for 50 epochs

# References


[1] G. STRANG. *Linear Algebra and Learning from Data*. Wellesley-Cambridge Press, 2019. 1

[2] MARVIN MINSKY AND SEYMOUR PAPERT. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge, MA, USA, 1969. 2

[3] IAN GOODFELLOW, YOSHUA BENGIO, AND AARON COURVILLE. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`. 3, 6, 11, 14, 17

[4] D STATHAKIS. **How many hidden layers and nodes?** *International Journal of Remote Sensing*, **30**(8):2133–2147, 2009. 3

[5] VINCENT DUMOULIN AND FRANCESCO VISIN. **A guide to convolution arithmetic for deep learning**. *ArXiv e-prints*, mar 2016. 5

[6] CHIEN-CHIH WANG, KENT LOONG TAN, AND CHIH-JEN LIN. **Newton Methods for Convolutional Neural Networks**, 2018. 5

[7] STEPHEN P BOYD. *Convex optimization*. Cambridge University Press, Cambridge, 2004. 9, 24, 25, 29

[8] DIEDERIK P KINGMA AND JIMMY BA. **Adam: A method for stochastic optimization**. *arXiv preprint arXiv:1412.6980*, 2014. 17

[9] ASHIA C WILSON, REBECCA ROELOFS, MITCHELL STERN, NATI SREBRO, AND BENJAMIN RECHT. **The marginal value of adaptive gradient methods in machine learning**. In *Advances in Neural Information Processing Systems*, pages 4148–4158, 2017. 17, 45, 51, 55

[10] JONATHAN RICHARD SHEWCHUK. **An Introduction to the Conjugate Gradient Method Without the Agonizing Pain**. August 1994. 20

[11] JORGE NOCEDAL AND STEPHEN WRIGHT. *Numerical optimization*. Springer Science & Business Media, 2006. 25, 26, 27, 29, 33, 36, 37, 38

[12] VYACHESLAV KUNGURTSEV AND TOMAS PEVNY. **Algorithms for solving optimization problems arising from deep neural net models: smooth problems**, 2018. 26

[13] CARL T KELLEY. *Iterative methods for optimization*. SIAM, 1999. 27

[14] NICOL N SCHRAUDOLPH. **Fast Curvature Matrix-Vector Products for Second-Order Gradient Descent**. *Neural Computation*, **14**(7):1723–1738, 2002. 27

[15] JAMES MARTENS AND ILYA SUTSKEVER. *Training Deep and Recurrent Networks with Hessian-Free Optimization*. 27, 54

[16] JAMES MARTENS. **Deep learning via hessian-free optimization.** 2010. 30, 33, 41, 45

[17] RON DEMBO AND TROND STEIHAUG. **Truncated-newton algorithms for large-scale unconstrained optimization**. *Mathematical Programming*, **26**(2):190–212, 1983. 31, 32, 40

[18] STEPHEN G NASH. **A survey of truncated-Newton methods**. *Journal of Computational and Applied Mathematics*, **124**(1-2):45–59, 2000. 32

[19] STANLEY C. EISENSTAT AND HOMER F. WALKER. **Choosing the Forcing Terms in an Inexact Newton Method**. *SIAM Journal on Scientific Computing*, **17**(1):16–32, 1996. 32

[20] D. C. SORENSEN. **Newton's Method with a Model Trust Region Modification**. *SIAM Journal on Numerical Analysis*, **19**(2):409–426, April 1982. 38

[21] RON S DEMBO, STANLEY C EISENSTAT, AND TROND STEIHAUG. **Inexact Newton Methods**. *SIAM journal on numerical analysis*, **19**(2):400–408, 1982. 40

[22] YANN LECUN AND CORINNA CORTES. **MNIST handwritten digit database**. 2010. 46

[23] ALEX KRIZHEVSKY. **Learning multiple layers of features from tiny images**. Technical report, 2009. 47

[24] SUDHIR B. KYLASA, FARBOD ROOSTA-KHORASANI, MICHAEL W. MAHONEY, AND ANANTH GRAMA. **GPU Accelerated Sub-Sampled Newton's Method**. *CoRR*, **abs/1802.09113**, 2018. 55