Introduction to the Theories Behind Second-order Optimisation Methods

and the Detailed Analysis of the BFGS Algorithm

Yue Xiao - 45307157

STAT3007 Semester 1, 2021

I give consent for this to be used as a teaching resource.

Contents

1 Introduction

Training a neural network is the process to tweak the weight parameters to minimise the value of the loss function in an iterative manner.

Finding the optimal weight parameters is not a trivial task and can consume vast amount of computational resources in large scale architectures. Therefore, investigating optimisation techniques that can reduce the required training iterations, while providing sufficient performance accuracy becomes an important focus in deep learning.

There are two main categories of optimisation algorithms: first-order and second-order numerical methods. Although first-order techniques, those which only use gradient information (e.g. gradient descent) are the most commonly used class in practice, second-order techniques advances to the higher level by constructing the next training iteration using additional information provided by the Hessian [8].

In this tutorial paper, we will cover the concept and the theories behind general second-order numerical methods and specifically, the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm.

2 Descent Methods

Before going further, let's review a general class of iterative methods, called the descent methods which governs the process of learning. Given an initial point $x_0 \in \mathbb{R}^n$, the main idea of descent methods is to find the next point $x_1 \in \mathbb{R}^n$ such that the loss function is closer to its local minimum. This process is iterated until reaching the local minimum. Formally,

$$x_{k+1} = x_k + \alpha_k p_k \qquad\qquad k = 0, 1, ... \tag{1}$$

where

1. $p_k \in \mathbb{R}^n$ is a descent direction at x_k
2. $\alpha_k > 0$ is an appropriate step-size at k^{th} iteration, along the direction p_k

Different algorithms of descent methods consist of finding different p_k and α_k such that it converges to a local minimum in finite steps. It is the unconstrained optimisation framework that are used in tweaking the weight parameters in neural network training. In the context of deep learning, the x_k variables above are corresponding to the weight parameters.

There are restrictions in choosing p_k and α_k, because not every direction or step size guarantee the convergence to a minimum. Next, we are going to talk about these restrictions or conditions that guarantee the sense of "descend".

2.1 Descent Direction

A direction $p \in \mathbb{R}^n$ is a descent direction of the loss function $f(x)$ at x if it decreases the function value for some step size $\alpha > 0$. This means that if we look at the loss along the direction p starting from x, then there would be a point that the loss is less than $f(x)$.

In other words, a direction p is a descent direction for loss function $f(x)$ at x if, for some $\alpha_0 > 0$, we have

$$f(x + \alpha p) < f(x) \qquad\qquad \forall \alpha \in (0, \alpha_0] \tag{2}$$

2.1.1 Sufficient condition for descent direction

We can also view the descent direction p as the direction such that the rate of change in f is negative. In other words, its derivative f' along p is negative. Therefore, we have the following **sufficient condition** for descent direction: if

$$f'(x; p) =< \nabla f(x), p >< 0 \tag{3}$$

then p is a descent direction for f at x.

More generally, we have the following statement:

Theorem 1 *$\forall S \succ 0$ (i.e. S is a positive definite matrix), $p = -S\nabla f(x)$ is a descent direction*

Proof. Suppose $S \in \mathbb{R}^{n\times n} \succ 0$, then $< v, Sv >> 0 \;\; \forall$ non-zero vector $v \in \mathbb{R}^n$.
Therefore, pick $v = \nabla f(x)$, then we have $< \nabla f(x), S\nabla f(x) >> 0$

$$\implies < \nabla f(x), -S\nabla f(x) >< 0$$

Set $p = -S\nabla f(x)$, then p is a descent direction satisfying $< \nabla f(x), p >< 0$ □

As a result, we can express the iterative formula (1) as the following:

$$x_{k+1} = x_k - \alpha_k S_k \nabla f(x) \qquad\qquad k = 0, 1, ... \tag{4}$$

Equation (4) tells us that many descent directions actually use the information provided by the gradient ∇f. In fact, they involve in finding a factor S_k conditioned on the gradient to make the "descending" much more efficient.

Exercise 1 *Show that the descent direction in Gradient Descent satisfies condition* (3).

2.2 Step size

In addition to finding the descent direction p, we also want to determine how far we should travel along p. As a terminology in deep learning, this is usually referred as the **learning rate**. if α_k is dependent on k, then it is referred as the **adaptive learning rate**. For this tutorial, we only consider the adaptive case because the constant case is a special case of adaptive step sizing.

Formally, finding the ideal step-size requires to solve:

$$\alpha_k = \arg\min_{\alpha>0} f(x_k + \alpha p_k) \tag{5}$$

However, apart from a few special cases, solving (5) **exactly** generally is considered to be impossible. As a result, we often resort to solving it **inexactly**, which is what we are going to talk about next.

2.2.1 Inexact Line Search

Solving (5) requires substantial amount of computational resources. Even if possible, devoting such amount to finding an α which precisely minimises f in one particular direction is not desirable. These amount of resources could instead, be employed to finding a better search direction, hence improving the efficiency overall [5]. In other words, the goal of deploying inexact line search is to find a step size α such that it provides a **reasonable amount** of reduction in the loss function f. To guarantee this, we need to impose a few conditions to justify what the "reasonable reduction" in the loss function truly means.

2.2.1.1 Armijo condition

One of the conditions that should be naturally imposed is to ensure that the step size is not excessively large. As mentioned in equation (2), if we found a descent direction p, then we can also find an interval $(0, \alpha_0]$ such that the loss is decreased. This implies that we want to have a step-size small enough such that it lies in the interval. Although the requirement for α imposed by Equation (2) ensures some sort of decrease, relying solely on it does not guarantee convergence to the minimiser. This is further illustrated in Figure 1, where the iterates failed to converge to the minimum of $f(x)$ if α is chosen solely based on Equation (2). Hence, a condition

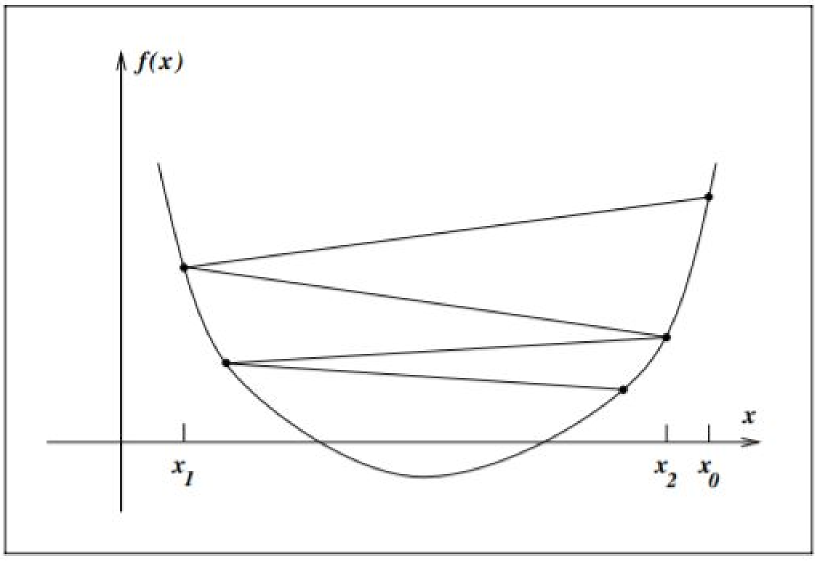


Figure 1: Insufficient decrease in f. Source: [4]

even stronger than equation (2) is imposed to achieve **sufficient decrease** in loss, that is, the **Armijo condition**: if

$$f(x + \alpha p) \leq f(x) + \alpha c_1 \nabla^T f(x) p \tag{6}$$

for some search control parameter $c_1 \in (0, 1)$ and descent direction p, then α provides sufficient reduction in the loss function.

The goal of the above Equation (6) is imposing a stricter criterion for reduction sufficiency than Equation (2). Furthermore, the rationale behind the Armijo condition is to find an α that provides $f(x_k + \alpha p)$ to be lower than the first order Taylor Approximation of $f(x_k + \alpha p)$ (RHS of Equation (6)). It is to ensure that the loss function **decreases faster than its step-wise linear interpolation**. This idea is illustrated on Figure 2.

In practice, control parameter c_1 is usually picked as 10^{-4} [4].

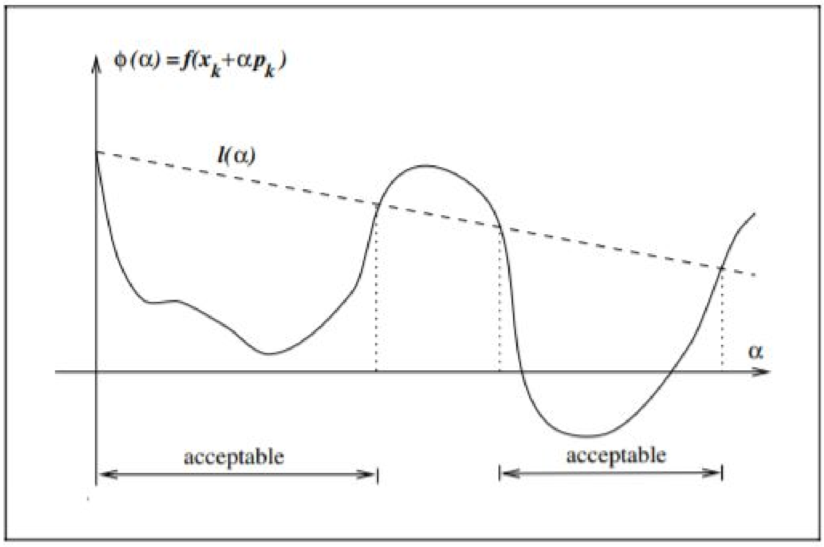


Figure 2: Armijo condition acceptance for α. $l(\alpha)$ denotes the RHS of Equation (6). Source [4]

2.2.1.2 the Curvature Condition

Although Armijo condition effectively provides sufficient decrease by providing a stricter upper-bound on $f(x + \alpha p)$, it is not enough to ensure reasonable progress. As illustrated in Figure 2, arbitrary small α values satisfy the condition. Therefore, there might be cases that the step sizes are so small such that the convergence to the minimiser is incredibly slow.

Hence, to place additional restrictions to rule out unacceptably short step, we introduce the **curvature condition**, which requires α to satisfy:

$$\nabla f(x_k + \alpha_k p_k)^T p_k \geq c_2 \nabla f(x_k)^T p_k \tag{7}$$

where $c_2 \in (c_1, 1)$ is the control parameter.
The curvature condition stated in Equation (7) places further restrictions on how the gradient behaves step-wise.

Let $\phi(\alpha) = f(x_k + \alpha_k p_k)$, then the RHS of (7) is the initial gradient of f along the descent direction p_k at x_k (i.e. $\phi'(0)$) and the LHS would be the gradient after traveling along that direction for α_k steps (i.e. $\phi'(\alpha_k)$). The curvature condition essentially imposes a condition that if the step-wise slope $\phi'(\alpha_k)$ is more negative than the initial slope $\phi'(0)$, then we have an indication that we can reduce the loss f significantly by moving further along the chosen direction. In contrast, if $\phi'(\alpha_k)$ is positive or even just slightly negative, then it is a sign that there might not be much decrease in this direction, so it makes sense to terminate the line search. This idea is illustrated in Figure 3.

The typical values of c_2 are 0.1 when p_k is obtained through a non-linear conjugate gradient method, and 0.9 if it is obtained through a Newton's or quasi-Newton method.

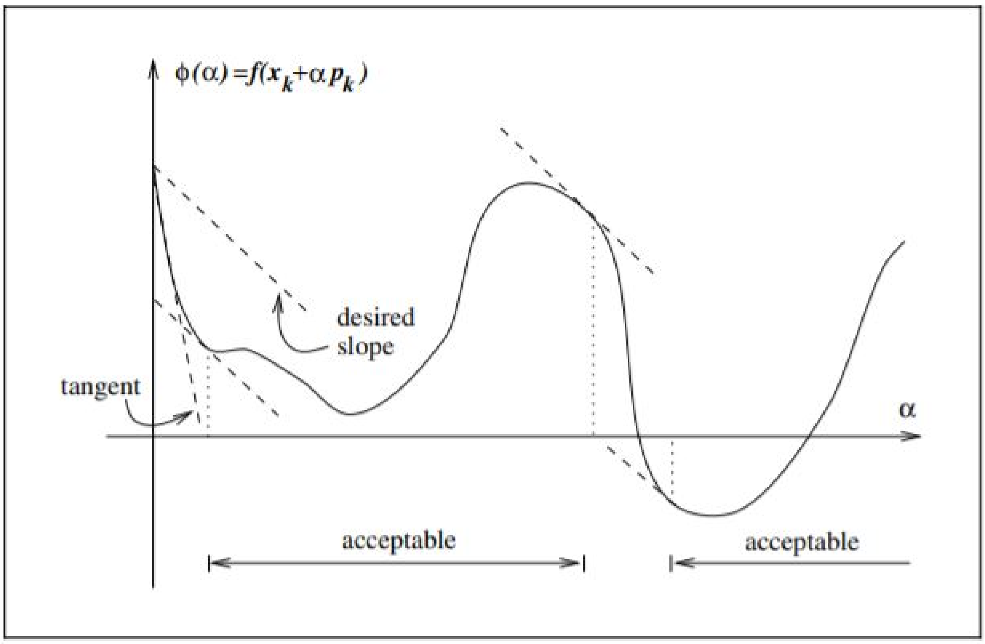


Figure 3: the curvature condition acceptance for α_k. Source: [4]

2.2.1.3 Wolfe Condition

As mentioned previously, the Armoji condition stated in Equation (6) provides sufficient decrease, however it does not stop insufficient progress by picking arbitrary small step sizes. On the other hand, the curvature condition state in Equation (7) provides that the line search terminates when the step size is far enough where the loss no longer decreases significantly, but it does not restrict a α_k for which that the f is higher than the staring point is chosen. Hence, it is clear to see that using the aforementioned conditions separately might result in insufficient descent iterations.

That being said, using them collectively yields a really effectively search strategy, known the as the **Wolfe Conditions** which requires α_k to satisfy **both** the **Armijo Condition** and the **Curvature Condition**. Figure 4 illustrates this effect.

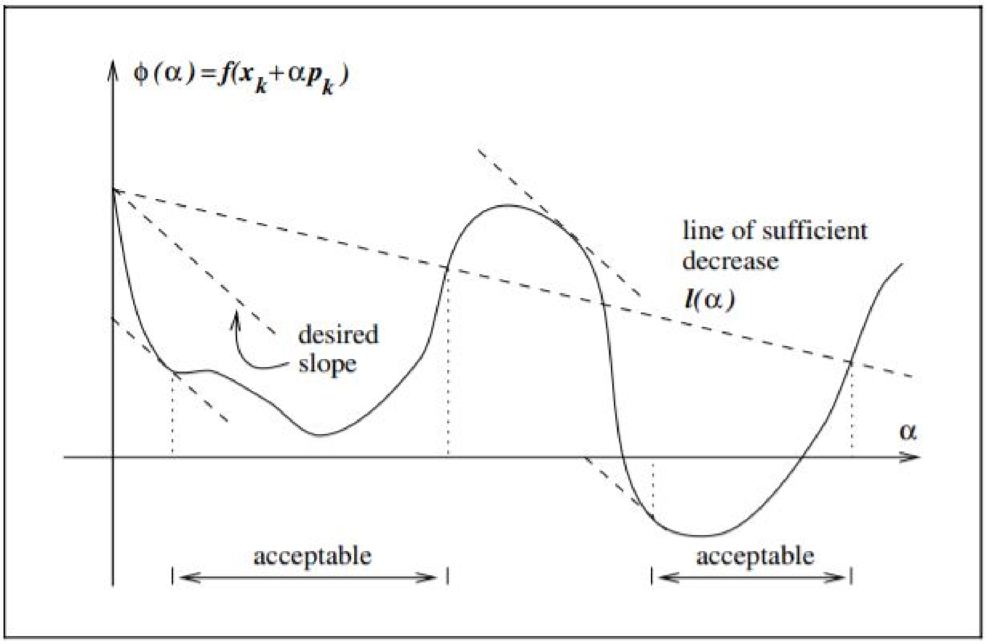


Figure 4: Wolfe Condition acceptance for α_k. Source: [4]

As shown in Figure 4, a step size α_k can still satisfy the Wolfe Conditions even if it is far away from a minimiser of $\phi(\alpha)$. Therefore, we can modify the curvature condition such that it no longer allows the gradient $\phi(\alpha_k)$ to be too positive, hence restricting in only choosing α_k that lies in at least a broad neighbourhood of a local minimiser of $\phi(\alpha)$. This is called the **Strong Wolfe Conditions**, which requires α_k to satisfy

$$f(x_k + \alpha_k p_k) \leq f(x_k) + \alpha c_1 \nabla f(x_k)^T p_k \tag{8}$$

$$|\nabla f(x_k + \alpha_k p_k)^T p_k| \leq c_2 |\nabla f(x_k)^T p_k| \tag{9}$$

for $0 < c_1 < c_2 < 1$.

Finally, we are going to prove the existence of step sizes that satisfy the (Strong) Wolfe Conditions, for every smooth function f that has a minimum:

Theorem 2 *Let p_k be a descent direction at x_k, and assume that f is bounded below along the ray $\{x_k + ap_k | a > 0\}$. Then if $0 < c_1 < c_2 < 1$, there exist intervals of step sizes satisfying the*

Wolfe conditions and the strong Wolfe conditions.

Proof. Let $\phi(\alpha) = f(x_k + \alpha p_k)$, then it is bounded below $\forall \alpha > 0$.
Since $0 < c_1 < 1$, then $l(\alpha) := f(x_k) + \alpha c_1 \nabla f(x_k)^T p_k$ is unbounded below.
Also since $\phi(0) = l(0)$,then this implies that $\phi(\alpha)$ and $l(\alpha)$ must intersect at least once for some α. Let α' be the first intersecting point, that is,

$$f(x_k + \alpha' p_k) = f(x_k) + \alpha' c_1 \nabla f(x_k)^T p_k \tag{10}$$

Since $l(\alpha)$ is a linear decreasing function over α, then the Armoji condition holds for all step sizes less than α'.
Now we check for the curvature condition:
By the mean value theorem, there exists $\alpha'' \in (0, \alpha')$ such that

$$f(x_k + \alpha' p_k) - f(x_k) = \alpha' \nabla f(x_k + \alpha'' p_k)^T p_k \tag{11}$$

From (10), we have $f(x_k + \alpha' p_k) - f(x_k) = \alpha' c_1 \nabla f(x_k)^T p_k$, combining with Equation (11), we have

$$\begin{aligned} \alpha' \nabla f(x_k + \alpha'' p_k)^T p_k &= \alpha' c_1 \nabla f(x_k)^T p_k \\ \implies \nabla f(x_k + \alpha'' p_k)^T p_k &= c_1 \nabla f(x_k)^T p_k \\ &> c_2 \nabla f(x_k)^T p_k \end{aligned} \tag{12}$$

Since $c_1 < c_2$ and $\nabla f(x_k)^T p_k =< f(x_k), p_k >< 0$.
Therefore α'' satisfies the Wolfe conditions and the inequality holds strictly for both Armoji condition and the curvature condition. Hence, by the smoothness of f, there is an interval around α'' for which the Wolfe conditions hold.
Moreover, since LHS of (12) is negative, then

$$|\nabla f(x_k + \alpha'' p_k)^T p_k| \leq c_2 |\nabla f(x_k)^T p_k| \tag{13}$$

hence, the Strong Wolfe conditions also hold in the same interval. □

The Wolfe Conditions can be used in most line search methods. In particular, they are important in the implementation of quasi-Newton methods, which will be discussed later in this tutorial paper. Next, we will propose a numerical algorithm for implementing the inexact line search using the Wolfe Conditions.

2.2.2 Line Search Algorithm: Backtracking

The inexact line search algorithm is also iterative. Moreover, the most used algorithm involves in backtracking - starts with a large α_0, then iteratively shrinking it until the Wolfe Conditions are met.

In pracise, α_0 is usually chosen as 1.

The details are stated below:

Algorithm 1: Backtracking (Strong) Wolfe Line Search

```
input  : initial step size $\alpha_0$,boolean indicating Strong condition $is\_strong$, descent
         direction $p$, starting point $x$, objective function $f$ , control parameters $c_1$ and $c_2$
$\alpha = \alpha_0$;
$\phi\_0 = f(x)$;
$g\_0 = \nabla f(x)^T p$;
while True do
    $\phi\_\alpha = f(x + \alpha p)$;
    $g\_\alpha = \nabla f(x + \alpha p)^T p$;
    armoji_rhs = $\phi\_0 + \alpha c_1 g\_0$;
    curvature_rhs = $c_2 g\_0$;
    if $is\_strong$ then
        if $\phi\_\alpha \leq$ armoji_rhs and $abs(g\_\alpha) \leq abs$(curvature_rhs) then
            break;
        end
    else
        if $\phi\_\alpha \leq$ armoji_rhs and $g\_\alpha >$ curvature_rhs then
            break;
        end
    end
    $\alpha = 2\alpha$;
end
output : step size $\alpha$ that satisfies (Strong) Wolfe Conditions
```

2.3 Summary

- descent methods choose a direction and its associated step size to tweak the parameters such that it provides sufficient decrease in the loss
- guarantee convergence if satisfying certain conditions
- as stated in Equation (4), many descent methods involve in placing a positive definite matrix conditioner S_k on the gradient
- If conditioner $S = I$ is the identity matrix, then the method is gradient descent.

In the next session, we are investigating the connection between second-order optimisation methods and the tuning of the conditioner S_k.

3 Newton's Method

Newton's method is an old yet powerful technique in solving various problems. Although it was first proposed as a root-finding method for polynomials, its usage was transferred into finding local optimum of functions by finding the roots of the gradient function. It is the first second-order optimisation technique proposed for training [8], and has since underlies other complex second-order techniques.

Firstly, we show the derivation of Newton's method. Consider the loss function $f(x)$ at point x, its second-order Taylor expansion from another point x_0 is:

$$\begin{aligned} f(x) &= f(x_0) + \nabla f(x_0)^T(x - x_0) + \frac{1}{2}(x - x_0)^T H(x_0)(x - x_0) \\ \implies \nabla f(x) &= \nabla f(x_0) + H(x_0)(x - x_0) \end{aligned} \tag{14}$$

Setting the gradient $\nabla f(x)$ to 0, then

$$\begin{aligned} 0 &= \nabla f(x_0) + H(x_0)(x - x_0) \\ \implies x &= x_0 - H^{-1}(x_0)\nabla f(x_0) \end{aligned} \tag{15}$$

The equation above approximates the local minimiser x using gradient and Hessian information in point x_0. Extending this idea to an iterative manner, we then have the famous Newton's method. Formally, for a current iterate x_k, the next iterate approximated by Newton's method is

$$x_{k+1} = x_k - \alpha_k H^{-1}(x_k)\nabla f(x_k) \tag{16}$$

where $H(x_k)$ is the hessian of the loss function f at iterate x_k and $\alpha_k \in [0, 1]$ is the step size via line search. This is equivalent to setting the conditioner $S_k = -\alpha_k H^{-1}(x_k)$ in Equation (4).

Newton's method is a very efficient algorithm since it incorporates more information about the curvature of the loss function and requires significantly fewer iterations to convergence to a local minimum compared to first-order techniques. In fact, using certain forms of loss functions (such as the quadratic loss, MSE etc), Newton's method is able to reach the minimum in a single iteration! [8].

However, its theoretical efficiency comes with a trade-off in high computational complexity. The need to directly compute and store the Hessian Matrix for each iteration proved to be impractical. Especially in the context of neural network training, where there can be thousands of parameters. Secondly, since the Hessian matrix H might not be positive definite, then the direction $p_k = -H^{-1}(x_k)\nabla f(x_k)$ may not always be a descent direction. This means that the theoretical results discussed in Section 2 no longer apply and thus, no guarantee convergence.

To improve the stability and ensures efficient computations, there are plethora of second-order optimisation methods that aim to approximate the Hessian matrix rather than computing it directly. One of these approximate classes is called the quasi-Newton methods, which is what we are discussing next.

Exercise 2 *Consider the quadratic function*

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

where $x \in \mathbb{R}^n$, constant vector $b \in \mathbb{R}^n$, constant $c \in \mathbb{R}$ and positive definite matrix $A \in \mathbb{R}^{n\times n}$. Given an initial guess x_0, show that Newton's method converges to the local minimum x^* in one iteration. Explain why x^* is also a global minimum.

4 Quasi-Newton Methods

Quasi-Newton methods, as its name suggests, are built on top of the vanilla Newton's method yet they are not exactly doing what Newton's method does. It is like Newton's method because it tries to retrieve additional curvature information provided by the Hessian to improve convergence; It is not like Newton's method because it does not actually use the Hessian matrix. In fact, quasi-Newton methods are generally used when the Hessian matrix is unavailable.

Instead, quasi-Newton methods aim to find an approximation matrix that resembles the Hessian efficiently. Formally, quasi-newton methods look for a descent direction in the form:

$$p_k = -B_k^{-1} \nabla f(x_k) \tag{17}$$

where B_k is an approximation matrix for the Hessian.

To successfully capture the features of the Hessian through an approximation matrix B_k, we want the approximation to somewhat resemble the Hessian. That is, we impose that B_k must be **symmetric**. Moreover, as we mentioned in the previous session, the Hessian itself is ill-conditioned when it is not positive definite, which does not guarantee convergence. Therefore, we impose that B_k is also **positive definite**.

The goal of generating matrix B_k is not necessarily to generate a sequence $\{B_k\}$ such that it converges to the Hessian of the minimiser $H(x^*)$ as this potentially slows down the convergence when approaching the minimiser x^*. In fact, it suffices that B_K becomes increasingly accurate approximate to $H(x^*)$ **along the search direction**. In other words, we have the following convergence property:

Theorem 3 (necessary and sufficient condition of quasi-newton methods)

$$\lim_{k \to \infty} \frac{||(B_k - H(x^*))p_k||}{||p_k||} = 0 \tag{18}$$

*holds if and only if the convergence is **superlinear** (in-between quadratic and linear convergence)*

5 BFGS Algorithm

The most popular quasi-Newton algorithm is the Broyden–Fletcher–Goldfarb–Shanno algorithm (BFGS), named after its discovers. It is considered to be the most effective algorithm of all quasi-Newton methods [7]. It determines the descent direction by gradually improving an approximation matrix B_k as in Equation (18) using only the gradient information. More specifically, B_k is generated via the secant method, which is considered as a finite-difference approximation of Newton's method.

First, let's introduce the idea of the secant method. Consider Newton's iteration for optimisation:

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)} \tag{19}$$

Approximating the second derivative $f''(x_k)$ using first order backward finite difference, we obtain

$$f''(x_k) = \frac{f'(x_k) - f'(x_{k-1})}{x_k - x_{k-1}} \tag{20}$$

Combining Equation (19) and (20), we have the secant iteration:

$$x_{k+1} = x_k - f'(x_k)\frac{x_k - x_{k-1}}{f'(x_k) - f'(x_{k-1})} \tag{21}$$

Extending the secant approximation in Equation (20) to the multivariate case, we obtain the **secant equation** for descending:

$$\begin{aligned} x_{k+1} - x_k &= B_{k+1}^{-1}(\nabla f(x_{k+1}) - \nabla f(x_k)) \\ \implies B_{k+1}(x_{k+1} - x_k) &= \nabla f(x_{k+1}) - \nabla f(x_k) \\ B_{k+1}s_k &= y_k \end{aligned} \tag{22}$$

where $s_k = x_{k+1} - x_k$ is the change of displacement and $y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$ is the change in gradient.
We then enforce B_{k+1} to be a symmetric positive definite matrix by requiring the next iterate to satisfy the following curvature condition

$$s_k^T y_k > 0 \tag{23}$$

Exercise 3 *prove the above inequality. i.e. if B_{k+1} is symmetric positive definite, then* (23) *is satisfied.*

Coupling Condition (23) with (Strong) Wolfe line search conditions (Equation (8) and (9)), we can always find a next iterate x_{k+1} that satisfies the curvature condition (23) even if the loss function f is not convex [7].

Exercise 4 *Show that the condition* (23) *is satisfied if the Strong Wolfe conditions (Equation* (8) *and* (9)*) are satisfied.*

Although satisfying the curvature condition (23) allows us to find a solution B_{k+1} for the secant solution, it admits an infinite number of solution candidates, since there are $\frac{n(n+1)}{2}$ degrees of freedom in a symmetric positive definite matrix which exceed the n conditions imposed by the secant equation (22). Therefore, we need to impose additional constraints or regularisation to determine an unique B_{k+1}. That is, we want B_{k+1} to be as close to the current approximation B_k as possible by solving the following constrained optimisation problem:

$$B_{k+1} = \min_B ||B - B_k|| \tag{24a}$$

$$\text{subject to } B = B^T, Bs_k = y_k \tag{24b}$$

Note that we explicit left out the definition of norm chosen above because the optimality of Problem (24a) is independent of the choice of matrix norm. Indeed, **different matrix norms give rise to a different quasi-Newton method**. Next, we introduce a choice of norm that leads to the formulation of BFGS.

5.1 DFP Formula

A easy solution to the minimisation problem (24) can be obtained by choosing the weighted Frobenius norm:

$$||A||_W = ||W^{\frac{1}{2}} A W^{\frac{1}{2}}||_F \tag{25}$$

where $||C||_F = \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij}^2$ is the standard Frobenius norm. The weight matrix W can be any symmetric positive definite matrix that satisfies $Wy_k = s_k$, similar to the condition imposed in (22).

If we set the W to be $W = G_k^{-1}$, where G_k is the the average Hessian:

$$G_k = [\int_0^1 H(x_k + \eta \alpha_k p_k) d\eta] \tag{26}$$

then the secant equation is satisfied using Taylor's theorem:

$$y_k = G_k \alpha_k p_k = G_k s_k \tag{27}$$

Using the weighted Frobenius norm coupling with the average Hessian as the weight matrix, the unique minimiser of Problem (24) is called the **DFP updating formula**:

$$B_{k+1} = (I - \rho_k y_k s_k^T) B_k (I - \rho_k y_k s_k^T) + \rho_k y_k y_k^T \tag{28}$$

where $\rho_k = \frac{1}{y_k^T s_k}$

5.2 BFGS formula

The DFP update formula in (28) was superseded by the BFGS updated formula, which achieved more efficiency by making a simple change in the derivation that led to Equation (28).

Instead of imposing conditions on the matrix B_{k+1} and solve for the secant equation (22) which we recall below:

$$B_{k+1} s_k = y_k$$

we impose the similar conditions on the inverse of B_{k+1}, denoted by $O_{k+1} = B_{k+1}^{-1}$ and solve the following modified secant equation

$$O_{k+1} y_k = s_k \tag{29}$$

Therefore, by formulating this as a similar constrained minimisation problem, written as

$$\min_O ||O - O_k|| \tag{30a}$$

$$\text{subject to } O = O^T, Oy_k = s_k \tag{30b}$$

Choosing the weighted Frobenius form with the same weight matrix as in the DFP formula, we have the **BFGS updating formula**

$$O_{k+1} = (I - \rho_k y_k s_k^T) O_k (I - \rho_k y_k s_k^T) + \rho_k s_k s_k^T \tag{31}$$

with the same $\rho_k = \frac{1}{y_k^T s_k}$

The reason why such a simple modification lies in the computational costs in solving the linear systems for quasi-Newton update:

$$p_k = -B_k^{-1} \nabla f(x_k)$$

Although obtaining B_k is efficient, solving for p_k amounts to computing the inverse of a matrix. There are different ways to compute the inverse, most directly by Gaussian Elimination,forward-backward LU decomposition or the Coopersmith-Winograd algorithm. However, even the fastest algorithm for inverses requires moderate computational complexity. Hence, it is well known in the numerical algebra community that it is basically **forbidden to invert matrices when possible** and one should **avoid inverting matrices at all costs**.

Therefore, if we approximate for $B_k^{-1} = O_k$ from the start, we are left with solving the following linear systems

$$p_k = -O_k \nabla f(x_k)$$

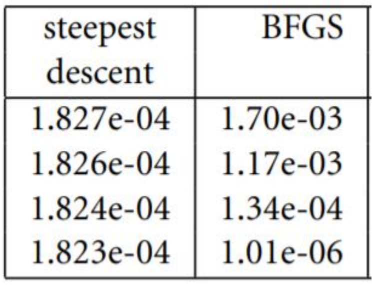
which excludes the need for matrix inversion.

Unfortunately, there is no set formula to choose the initial approximation matrix O_0 in practice. The most straightforward selection is to set it to the identity matrix i.e. $O_0 = I$. Now that we have all the ingredients available, we have finally state the BFGS algorithm:

Algorithm 2: BFGS Algorithm

```
input  : starting point $x_0$, error tolerance $\epsilon > 0$, initial Hessian approximation $O_0$
$k = 0$;
while $||\nabla f(x_k)|| > \epsilon$ do
    $p_k = -O_k \nabla f(x_k)$ (search direction);
    Compute $\alpha_k$ from Algorithm 1;
    $x_{k+1} = x_k + \alpha_k p_k$;
    $s_k = x_{k+1} - x_k = \alpha_k p_k$;
    $y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$;
    Compute $O_{k+1}$ by BFGS update formula in Equation (31);
    $k = k + 1$
end
output : the minimiser $x_k$
```

Provided in [7], the values of $||x_k - x^*||$ for the last four iterations are shown in the table below. From the starting point $x_0 = (-1.2, 1)$ on the Rosenbrock's function, the steepest descent



steepest descent	BFGS
1.827e-04	1.70e-03
1.826e-04	1.17e-03
1.824e-04	1.34e-04
1.823e-04	1.01e-06

required 5264 iterations, whereas BFGS was able to converged in 34 iterations.

5.3 Limited-Memory BFGS

Although BFGS provides an efficient approximation to the Hessian matrix, it has high memory storage requirement. In fact, building O_k over iterations quickly makes the approximation dense, which increases the storage requirement during computation heavily. Therefore, instead of storing fully dense matrices, we seek to represent them implicitly with a few representative vectors.

To achieve this, Limited-Memory BFGS (L-BFGS) uses only the curvature information in the most recent iterations to construct the next inverse Hessian approximation. Earlier iterations are less likely to to have relevant information for the behaviour of the Hessian at the current iteration, hence we can discard them in order to save storage. Recall the following

$$x_{k+1} = x_k + \alpha_k p_k \tag{32}$$

$$p_k = -O_k \nabla f(x_k) \tag{33}$$

$$O_{k+1} = V_k^T O_k V_k + \rho_k s_k s_k^T \tag{34}$$

where

1. $\rho = \frac{1}{y_k^T s_k}$
2. $V_k = I - \rho_k y_k s_k^T$
3. $s_k = x_{k+1} - x_k$
4. $y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$

Instead of storing O_k directly, we aim to represent them through m number of the vector pairs $\{s_i, y_i\}$. Therefore, our storage requirement is reduced to storing m vector pairs, rather than a whole matrix. This implies that when we compute the product $O_k \nabla f(x_k)$ - which is essentially a vector, it can be obtained purely through performing a sequence of vector summations and inner products involving $\{s_i, y_i\}$ and $\nabla f(x_k)$ [3]. There are at most m vector pairs that are stored, therefore, we only include m most recent vector pairs. When a new pair $\{s_{k+1}, y_{k+1}\}$ is computed, we discard the oldest pair in the sequence. i.e. $\{s_{k-m+1}, y_{k-m+1}\}$.

Now we seek to derive an expression for O_k in terms of only the vector pairs. Consider the standard BFGS update rule

$$O_k = V_k^T O_{k-1} V_k + \rho_{k-1} s_{k-1} s_{k-1}^T \tag{35}$$

By recursively substituting the same update rule expression of O_i for past iterations $i = k - m, k-m+1, ..., k-1$, we get

$$\begin{aligned} O_k &= V_{k-1}^T [V_{k-2}^T O_{k-2} V_{k-2} + \rho_{k-2} s_{k-2} s_{k-2}^T] V_{k-1} + \rho_{k-1} s_{k-1} s_{k-1}^T \\ &= ... \\ &= [V_{k-1}^T ... V_{k-m}^T] O_{k-m} [V_{k-m} ... V_{k-1}] \\ &+ \rho_{k-m} [V_{k-1}^T ... V_{k-m}^T] s_{k-m} s_{k-m}^T [V_{k-m} ... V_{k-1}] \\ &+ ... \\ &+ \rho_{k-1} s_{k-1} s_{k-1}^T \end{aligned} \tag{36}$$

Unlike the standard form, O_{k-m} is not available as we no longer store the matrix directly. Hence we replace it by some initial approximation matrix O_k^0. Hence

$$\begin{aligned} O_k &= [V_{k-1}^T ... V_{k-m}^T] O_k^0 [V_{k-m} ... V_{k-1}] \\ &+ \rho_{k-m} [V_{k-1}^T ... V_{k-m}^T] s_{k-m} s_{k-m}^T [V_{k-m} ... V_{k-1}] \\ &+ ... \\ &+ \rho_{k-1} s_{k-1} s_{k-1}^T \end{aligned} \tag{37}$$

From the vector operation based expression of O_k can we derive a recursive vector operation based procedure to compute the product $O_k \nabla f(x_k)$. This is known as the **two-loop recursion**. Its implementation details is outlined below:

Algorithm 3: two-loop recursion

```
input  : $m$ recent vector pairs $\{s_i, y_i\}$, initial approximation matrix $O_k^0$
$q = \nabla f(x_k)$;
for $i = k-1, k-2, ..., k-m$ do
    $a_i = \rho_i s_i^T q$;
    $q = q - a_i y_i$;
end
$r = O_k^0 q$;
for $i = k-m, k-m+1, ..., k-1$ do
    $\beta = \rho_i y_i^T r$;
    $r = r + s_i(a_i - \beta)$;
end
output : $r$ = $O_k \nabla f(x_k)$
```

In practice, the initial approximation O_k^0 is set to be $O_k^0 = \gamma_k I$, where I is the identity matrix and

$$\gamma_k = \frac{s_{k-1}^T y_{k-1}}{y_{k-1}^T y_{k-1}} \tag{38}$$

is the scaling factor that estimate the size of the Hessian matrix along the most recent descent direction p_{k-1}.

Combine everything together, we obtain the formal algorithm for L-BFGS:

Algorithm 4: L-BFGS Algorithm

```
input  : starting point $x_0$, error tolerance $\epsilon > 0$, storage capacity $m$
$k = 0$;
while $||\nabla f(x_k)|| > \epsilon$ do
    Compute $O_k^0$ as in Equation (38);
    Compute $p_k = -O_k \nabla f(x_k)$ from Algorithm 3 (search direction);
    Compute $\alpha_k$ from Algorithm 1 (step size);
    $x_{k+1} = x_k + \alpha_k p_k$;
    $s_k = x_{k+1} - x_k = \alpha_k p_k$;
    $y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$;
    if $k > m$ then
        Discard $(s_{k-m}, y_{k-m})$ from the vector pair sequence $\{s_i, y_i\}$;
    end
    Save $(s_k, y_k)$ $k = k + 1$
end
output : the minimiser $x_k$
```

6 Limitation of (L-)BFGS

- **High Computational Complexity** (L-)BFGS has to significantly more vector/matrix operations in order to choose a descent direction and step size.

- **High memory complexity** BFGS suffers from the need to store the Hessian matrix. In large-scale neural networking architectures, where there are thousands of parameters to tweak, these requirement becomes fatal. Even with L-BFGS, the vector pairs can pose a burden on memory storages.

- **Not applicable to minibatching** Due to the nature of (L-)BFGS, it is very sensitive to changes in the input data. Therefore, batch normalisation becomes extremely difficult in (L-)BFGS. In other words, it only works in full-batching training, hence it has limited applicability for large-scale problems.

7 Interesting and relevant resources

There are plenty of second-order methods out there apart from quasi-Newton methods! Here are some resources that might cover more about second-order optimisation methods in general.

- For a really broad coverage of numerical optimisation in general, which what this tutorial is heavily inspired on. Refer to the book [6]

- For a recent review of various second-order methods. Refer to [8]

- A modern version of L-BFGS, which avoids dot product operations in the two-loop recursion step is proposed in [1].

8 Exercise Solutions

Solution 1. *Show that the descent direction in Gradient Descent satisfies condition* (3)*.*

Proof. the descent direction in Gradient descent is $p = -\nabla f(x)$, then consider

$$\begin{aligned} < \nabla f(x), p > &= < \nabla f(x), -\nabla f(x) > \\ &= -\nabla f(x)^T \nabla f(x) \\ &= -|\nabla f(x)|^2 \\ &\leq 0 \end{aligned}$$

□

Solution 2. *Consider the quadratic function*

$$f(x) = \frac{1}{2} x^T A x - b^T x + c$$

where $x \in \mathbb{R}^n$, constant vector $b \in \mathbb{R}^n$, constant $c \in \mathbb{R}$ and positive definite matrix $A \in \mathbb{R}^{n \times n}$. Given an initial guess x_0, show that Newton's method converges to the local minimum x^* in one iteration. Explain why x^* is also a global minimum.

Proof. The gradient $\nabla f(x) = Ax - b$. let x^* be a minimum, then we have

$$\begin{aligned} \nabla f(x^*) &= 0 \\ &= Ax^* - b \\ \implies x^* &= A^{-1} b \end{aligned}$$

Observe that the Hessian $H(x) = A$
Now we consider the Newton iteration with initial guess x_0, then the next iterate is computed by

$$\begin{aligned} x_1 &= x_0 - H(x_0)^{-1} \nabla f(x_0) \\ &= x_0 - A^{-1}(Ax_0 - b) \\ &= A^{-1} b \end{aligned}$$

Hence, we have

$$x_1 = x^*$$

Since f is a quadratic function, it is convex. This implies that there is only one stationary point. Hence x_1 is the global minimum □

Solution 3. *Show that if B_k is symmetric positive definite, then* (23) *is satisfied.*

Proof. Suppose B_{k+1} is symmetric positive definite, and we have $B_{k+1}s_k = y_k$. Consider multiplying the LHS by s_k^T,

$$\begin{aligned} s_k^T y_k &= s_k^T B_{k+1} s_k \\ &> 0 \end{aligned}$$

□

Solution 4. *Show that the condition* (23) *is satisfied if the Strong Wolfe conditions (Equation* (8) *and* (9)*) are satisfied.*

Proof. Suppose that we have a descent direction and step size such that

$$x_{k+1} = x_k + \alpha_k p_k$$

where α_k satisfies the Wolfe Condition. Namely,

$$\nabla f(x_k + \alpha_k p_k)^T p_k \geq c_2 \nabla f(x_k)^T p_k$$

Then let $s_k = x_{k+1} - x_k = \alpha_k p_k$, we have

$$\begin{aligned} &\nabla f(x_{k+1})^T p_k \geq c_2 \nabla f(x_k)^T p_k \\ \implies &\nabla f(x_{k+1})^T \alpha_k p_k \geq c_2 \nabla f(x_k)^T \alpha_k p_k \\ \implies &\nabla f(x_{k+1})^T s_k \geq c_2 \nabla f(x_k)^T s_k \end{aligned}$$

Rearrange the inequality, we have

$$(\nabla f(x_{k+1}) - \nabla f(x_k))^T s_k \geq (c_2 - 1) \nabla f(x_k)^T s_k$$

Let $y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$, then we have

$$\begin{aligned} y_k^T s_k &= s_k^T y_k \\ &\geq (c_2 - 1) \nabla f(x_k)^T s_k \\ &= (c_2 - 1) \alpha_k \nabla f(x_k)^T p_k \end{aligned}$$

Since p_k is a descent direction such that $\nabla f(x_k)^T p_k =< \nabla f(x_k), p_k >< 0$ and $c_2 < 1$ so that $(c_2 - 1) < 0$, then we have

$$\begin{aligned} s_k^T y_k &\geq (c_2 - 1) \alpha_k \nabla f(x_k)^T p_k \\ &> 0 \end{aligned}$$

□

References

[1] Weizhu Chen, Zhenghao Wang, and Jingren Zhou. “Large-scale L-BFGS using MapReduce”. In: *Advances in Neural Information Processing Systems*. Ed. by Z. Ghahramani et al. Vol. 27. Curran Associates, Inc., 2014. URL: `https://proceedings.neurips.cc/paper/2014/file/e49b8b4053df9505e1f48c3a701c0682-Paper.pdf`.

[2] “Fundamentals of Unconstrained Optimization”. In: *Numerical Optimization*. New York, NY: Springer New York, 2006, pp. 164–192. ISBN: 978-0-387-40065-5. DOI: `10.1007/978-0-387-40065-5_2`. URL: `https://doi.org/10.1007/978-0-387-40065-5_2`.

[3] “Large-Scale Unconstrained Optimization”. In: *Numerical Optimization*. New York, NY: Springer New York, 2006, pp. 164–192. ISBN: 978-0-387-40065-5. DOI: `10.1007/978-0-387-40065-5_7`. URL: `https://doi.org/10.1007/978-0-387-40065-5_7`.

[4] “Line Search Methods”. In: *Numerical Optimization*. New York, NY: Springer New York, 2006, pp. 30–65. ISBN: 978-0-387-40065-5. DOI: `10.1007/978-0-387-40065-5_3`. URL: `https://doi.org/10.1007/978-0-387-40065-5_3`.

[5] David G. Luenberger and Yinyu Ye. “Basic Descent Methods”. In: *Linear and Nonlinear Programming*. Cham: Springer International Publishing, 2016, pp. 213–262. ISBN: 978-3-319-18842-3. DOI: `10.1007/978-3-319-18842-3_8`. URL: `https://doi.org/10.1007/978-3-319-18842-3_8`.

[6] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. second. New York, NY, USA: Springer, 2006.

[7] “Quasi-Newton Methods”. In: *Numerical Optimization*. New York, NY: Springer New York, 2006, pp. 135–163. ISBN: 978-0-387-40065-5. DOI: `10.1007/978-0-387-40065-5_6`. URL: `https://doi.org/10.1007/978-0-387-40065-5_6`.

[8] Hong Hui Tan and King Hann Lim. “Review of second-order optimization techniques in artificial neural networks backpropagation”. In: *IOP Conference Series: Materials Science and Engineering* 495 (June 2019), p. 012003. DOI: `10.1088/1757-899x/495/1/012003`. URL: `https://doi.org/10.1088/1757-899x/495/1/012003`.