Neural Network and its optimization via Hessian-free Newton's method

Aman Dhiman

A dissertation submitted for the partial fulfilment of BS-MS dual degree in Science

Department of Mathematics

Indian Institute of Science Education and Research

Mohali, India

Certificate of Examination

This is to certify that the dissertation titled **Neural Network and its optimization via Hessian-free Newton's method** submitted by **Mr. Aman Dhiman** (Reg. No. MS14079) for the partial fulfilment of BS-MS dual degree programme of the Institute, has been examined by the thesis committee duly appointed by the Institute. The committee finds the work done by the candidate satisfactory and recommends that the report be accepted.

Dr. Kapil Hari Paranjape	Dr. Lingaraj Sahu	Dr. Neeraja Sahasrabuddhe
(Committee member)	(Committee member)	(Committee member)

Dr. Shane D'mello	Dr. Anuj Sharma, PU
(Supervisor)	(Co-Supervisor)

Dated: April 25, 2019

Declaration

The work in this dissertation has been carried out by me under the guidance of **Dr. Anuj Sharma, Department of Computer Science, PU** at the **Indian Institute of Science Education and Research, Mohali**.
This work has not been submitted in part or in full for a degree, a diploma or a fellowship to any other university or institute. Whenever contribution of others are involved, every effort is made to indicate this clearly, with due acknowledgement of collaborative research and discussion. This thesis is a bonafide record of original work done by me and all sources listed within have been detailed in the bibliography.

(Candidate)
Aman Dhiman

Dated: April 25, 2019

In my capacity as the (supervisor)/(internal supervisor) of the candidate's project work, I certify that the above statements by the candidate are true to the best of my knowledge.

(Co-Supervisor)
Dr. Anuj Sharma,
Department of Computer Science,
Panjab University.

(Supervisor)
Dr. Shane D'mello,
Department of Mathematics,
IISER, Mohali.

Acknowledgement

Foremost, I would like to express my sincere gratitude to my advisor **Dr. Anuj Sharma, Department of Computer Science, PU**, for giving me an opportunity to work on a project of my interest and, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all time of research and provided me with the most valuable notes for my thesis. I would also like to thank **Dr. Shane D'mello, Department of Mathematics, IISER Mohali**, for providing motivation and support constantly throughout my thesis research. His constructive doubts regarding my topic encouraged me to explore the area more and understand things thoroughly, which has helped me alot to become confident in my research.

Contents

List of Figures

Abstract

Neural network has become a core part of machine learning in recent years and conventional neural network although successful, gets bottle-necked due to slow technology and lack of optimization potential in the common methods used to train the neural networks.
Some common methods used include, SGD(stochastic gradient descent), ADAM and ADAGRAD etc. These all algorithms are based on Gradient Descent, a first order root-finding algorithm. These are great for small scale neural network computation, but at industry level, where millions of data-points are produced, the neural networks required for the learning from the data-points either require large number of nodes or lots of layers, so it lacks the performance. The problem with Gradient Descent is that it becomes immensely slow as layers or nodes in the neural network increases, as well as the lack of optimal-direction finding potential on pathological curves makes it even harder to train networks using Gradient descent based algorithms.
2nd order optimization method, Newton's method, has been known to converge to the root faster than Gradient Descent and since it is a 2nd order algorithm, it has the curvature data, so we can modify the Newton's method to compensate for the problems that Gradient Descent faces. The thesis research deals with one such modification, which in optimization terminology is called Hessian-free approach. Further in this document, we will suggest ways to modify the Newton's method. The modified Hessian-free Newton's method will deal with the problem of optimization of cost function of the neural network efficiently.

Chapter 1

Introduction

1.1 Perceptron

Perceptrons were first developed by Frank Rosenblatt, inspired by work of Warren McCulloch and Walter Pitts [Nie]. To understand how modern neural networks work, it is necessary to understand 'Perceptron', which is the bone of the neural network model.

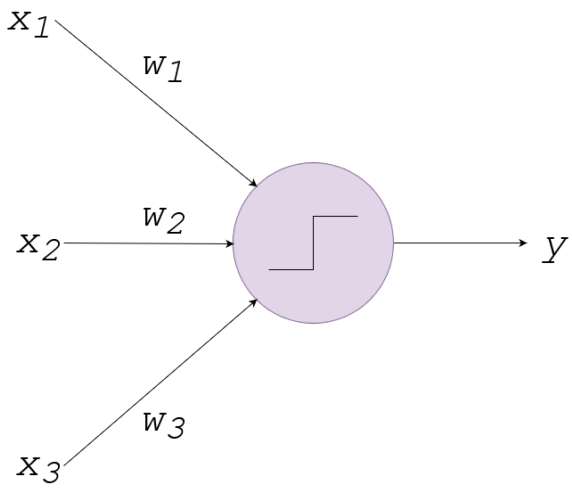


Figure 1.1: Perceptron

Perceptron is a simple model of biological neuron in an artificial network, that takes one or more binary inputs and produces a binary output.

Every decision we take is consequence of certain factors which selectively influence it. To accommodate that influence factor in the model, Rosenblatt introduced 'weights' associated with each input of the perceptron. For example, if input x_1 is influencing the output

more than input x_2, then $w_1 > w_2$, where w_1 and w_2 are respective weights associated with the inputs x_1 and x_2.

1.2 Decision making model

Warren McCulloch and Walter Pitts created a computational decision making model for neural networks, which they called *threshold logic*.

Let x_i be the i^{th} input and w_i be the corresponding weight. Then the weighted sum of all inputs is given by $\sum w_i x_i$, and the output, y, of model is,

$$y = \begin{cases} 0 & \text{if} \quad \sum w_i x_i \leq \text{threshold} \\ 1 & \text{if} \quad \sum w_i x_i > \text{threshold}, \end{cases} \tag{1.1}$$

where threshold is a fixed constant. This model provides basic structure to the decision making of the perceptron, and the output can be changed by varying the weights of the inputs.

The model can be simplified mathematically by using $w \cdot x$ instead of $\sum w_i x_i$ and taking 'threshold' term to the L.H.S. of the inequality. Let $-threshold = b$, where b is called the bias. Then the output, y, of the model is given by:

$$output = \begin{cases} 0 & \text{if} \quad w \cdot x + b \leq 0 \\ 1 & \text{if} \quad w \cdot x + b > 0. \end{cases} \tag{1.2}$$

We can implement the Decision making model in the perceptron and produce outputs. A perceptron works well only when the prediction we want is linearly separable. Adding more perceptron to the layer of single perceptron will allow the separation in different linear directions allowing more abstract separation than a single perceptron.

1.3 Multilayer Perceptron model

In many references or books, for historical reasons, MLP(multilayer perceptron model) is used to describe modern neural network, but we would be using MLP for the network of perceptron, which is less confusing.

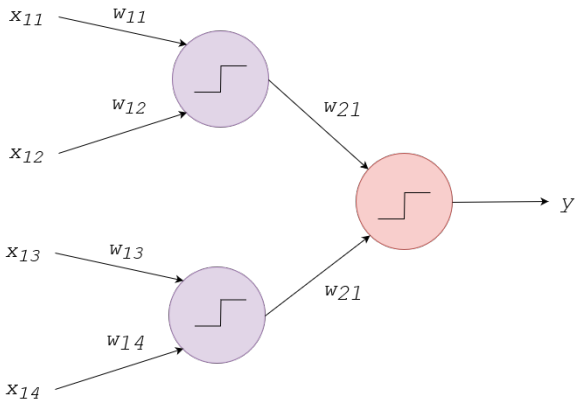


Figure 1.2: Multilayer perceptron network: Every circle represents a perceptron, every inward arrow represents input, outward arrow represents output.

In the figure 1.2, each hidden-layer is taking 'decision' at a more complex or abstract level than the previous perceptron layer. This way the MLP can engage in more sophisticated decision making.

1.4 Perceptron as NAND gate

I am including this topic, as it is a very interesting to know that Perceptron model is consistent with the fundamental gates and Boolean mathematics, that is used to do binary computation.

Fundamental logic gates include, AND & OR. Now to show that perceptron can mimic the fundamental gates, lets design a perceptron that can mimic NAND gate. NAND gate is universal gate and can be used to build every possible Boolean logic. Consider a perceptron with two input x_1 and x_2, with $w_1 = w_1 = -4$ and $b = 7$. This perceptron look like this:

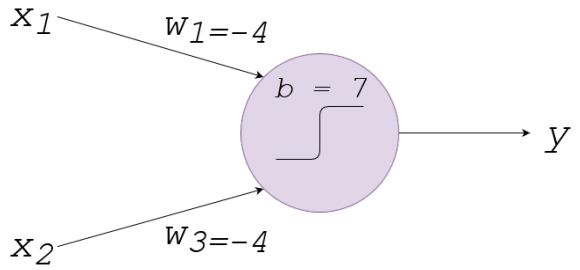


Figure 1.3: An example of perceptron mimicking NAND gate

Table 1.1: Logic table

w_1	w_2	evaluate	output
1	1	$-4(1)-4(1)+7$	0
1	0	$-4(0)-4(1)+7$	1
0	1	$-4(1)-4(0)+7$	1
0	0	$-4(0)-4(0)+7$	1

This means that a perceptron can easily evaluate any logical function.

Chapter 2

Understanding Neural Network

Given certain amount of data, the neural network undergoes learning process and trains itself for the data. Learning algorithm follows the idea, that small change in weights should make small change in the output. Only then the learning can be controlled and made efficient over many iterations.

Perceptron is not very good with small changes. Small change in the perceptron leads to large change in output. It even flips the output of the perceptron sometimes, which is not appropriate for the leaning algorithm.

2.1 Sigmoid Neuron

Consider $z = w \cdot x + b$, then z can be enhanced using a sigmoid function to overcome the problem of learning in a perceptron.

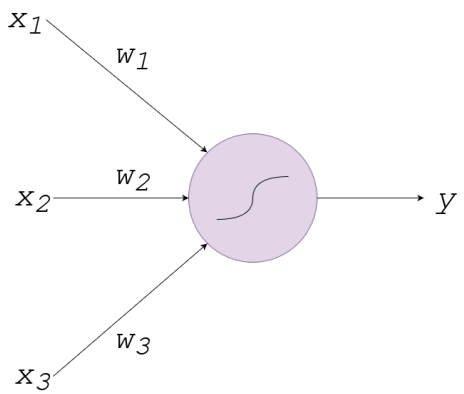


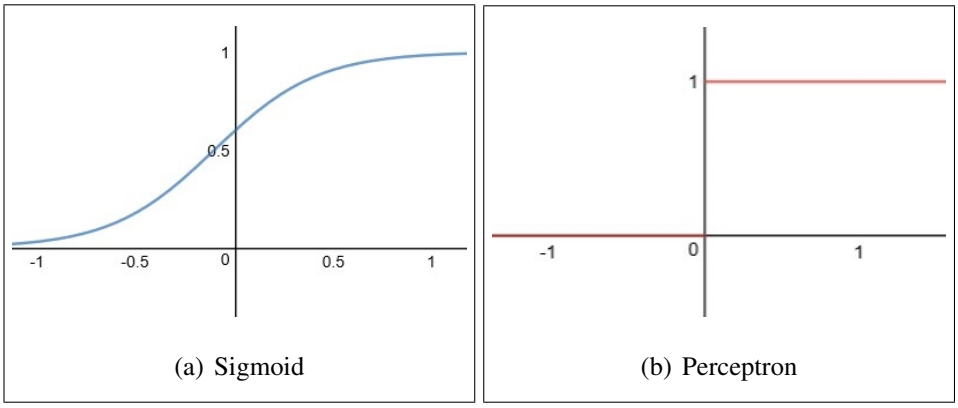
Figure 2.1: Sigmoid Neuron

The output of sigmoid neuron is given by:

$$\sigma_{sigmoid}(z) = \frac{1}{1+e^{-z}}.$$

The sigmoid neuron acts same as perceptron at $z \gg 0$ and $z \ll 0$.

For $z \gg 0$, $e^{-z} \approx 0$, then $\sigma(z) = 1$. For $z \ll 0$, $e^{-z} \approx \infty$, then $\sigma(z) = 0$. So, for very large positive and negative values of z, the sigmoid neuron behaves as perceptron.



(a) Sigmoid (b) Perceptron

Figure 2.2: Output comparison between sigmoid and perceptron neuron

In case of perceptron, a small change in weights will yield either 0 or 1 as output. Sigmoid neuron, the smoothness of the sigmoid curve allows to make small change in weights and biases, causing small change in the output.

The change in output with respect to the change in weights and biases is a linear function of Δw and Δb.

$$\triangle\sigma(z) \approx \sum_j \frac{\partial\sigma}{\partial w_j} \triangle w_j + \frac{\partial\sigma}{\partial b} \triangle b,$$

where sum is over all the inputs of the neuron.

2.2 Neural Network

Sigmoid neuron is one of many such possible neurons. The choice of neuron to build a layer in neural network depends on the problem. Consider a neural network as in figure 2.3, the *hidden* layer of the network constitutes its main processing unit.This is where the decisions are processed. Choosing the number of *hidden* layers for neural network is totally problem based. In some cases, having only one *hidden* layer yields proper outputs whereas some neural network has to incorporate large amounts of *hidden* layers, and such neural networks are called Deep neural networks.

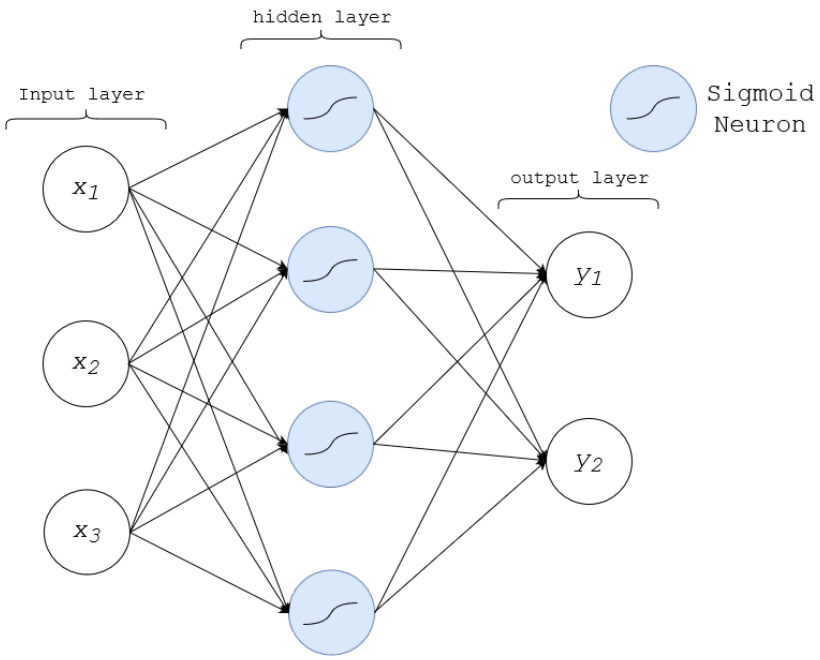


Figure 2.3: A neural network

2.3 Training neural network

Learning: Learning is acquiring knowledge or skill, or modify an existing behaviour, to achieve desired results. This can be done by two ways:

1. By rewarding a *good behaviour*.

2. By punishing a *bad behavior*.

A neural network, during training, performs learning algorithm and modifies the initial weights. This is done by punishing the network for every wrong guess by reducing weights of the connections that lead to that prediction. [1]By convention the other weights of the connections leading to correct prediction will get rewarded. An untrained network means the weights and biases of the network are normally distributed and doesn't have the required configuration of weights and biases to produce correct prediction.
We will first look at the learning rule that we will follow further in the research.

Error correction learning rule: Let error-function for the neural network (θ) be $E(\theta_i) = \frac{(f_{act} - f(\theta_i))^2}{2}$, where $f(\theta_i)$ is the ith instance of predicted output and f_{act} is the actual output.

[1]If particularly the connection with wrong predictions are getting punished, then other weights which are unchanged are automatically getting rewarded with respect to the later.

The θ in $(i+1)^{th}$ instance is adjusted using:

$$\theta_{i+1} = \theta_i + \triangle E(\theta_i) X_i,$$

where $\triangle E(\theta_i) = -2\eta(f(\theta)_{act} - f(\theta)_{pred})$, η is the learning rate of the neural network and X_i is the input vector.

$$\boxed{\theta_{i+1} = \theta_i - 2\eta(f_{act} - f(\theta)_{pred})X_i}.$$

Weights are adjusted only when neurons respond in error i.e,

$$(f_{act} - f(\theta)_{pred}) > 0 \quad or \quad (f_{act} - f(\theta)_{pred}) < 0$$

As soon as there is some amount of error in the prediction, the neural network will update the weights towards the direction of reducing error. In the above definition, the weights are updated using Gradient Decent. This update cycle goes on until either the error has become 0, or the error has converged to value near 0.

Chapter 3

Optimization methods

Optimization in learning process of neural network makes sure that the neural network performs better than the previous iteration.

Cost function of neural network is given by:

$$f(\theta) = \frac{1}{l}\sum \xi(predicted, target) \rightarrow \text{Cost function}$$

where, $\xi = ||target - predicted||^2$ and $predicted$ is a function of θ.

Learning process involves finding minimum of $f(\theta)$.

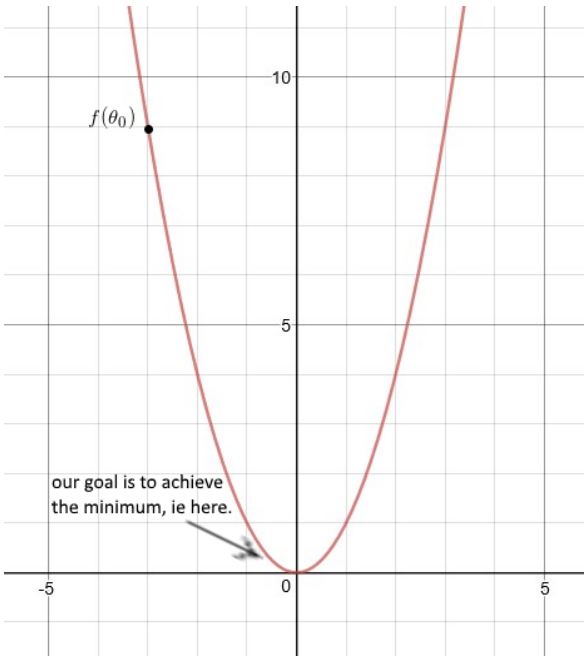


Figure 3.1: A very simple cost function.

As seen earlier, weights are updated using Gradient decent, which is a line search method, that updates the function towards its minimum.

3.1 Gradient Decent

Line search methods: Line search methods is a family of iterative optimization methods, where the iterations are given by:

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

The idea is to search for a direction such that $x_{k+1} < x_k$.
One such method is Gradient decent method, where iterations are given by,

$$x_{k+1} = x_k - \eta \nabla f(x_k),$$

here η is the learning rate.

1. if η is very small, then the optimization takes very long to converge to the minimum and is very heavy on computation.

2. if η is very high, then there is high chance of overshooting away from the minimum.

So, appropriate learning rate has to be decided for best results. Gradient Descent is computationally expensive for large neural networks, because convergence to minimum becomes slower with increase in network layers and also all the gradients are stored until the whole training data is completed and then the weights are updated. Imagine a deep neural network with 100 layers, for $n * n$ weight matrix it is doing the calculation n^2 times, which is computationally heavy as well as very storage heavy. So, commonly SGD (stochastic gradient descent) is used.
Stochastic Gradient Descent SGD is the application of Gradient descent over random batch sample of training data. This way the computation is reduced to that small batch and next batch is not used until learning from previous batch of training data is updated in the network.

3.2 Newton's method

In optimization , it is well known that Gradient descent is unsuitable for minimizing pathological curves since it is first order optimization method.
But a 2^{nd} order root finding method is more efficient in such cases as they use the curvature to find the direction of minima in the curve. One such method is Newton's method.

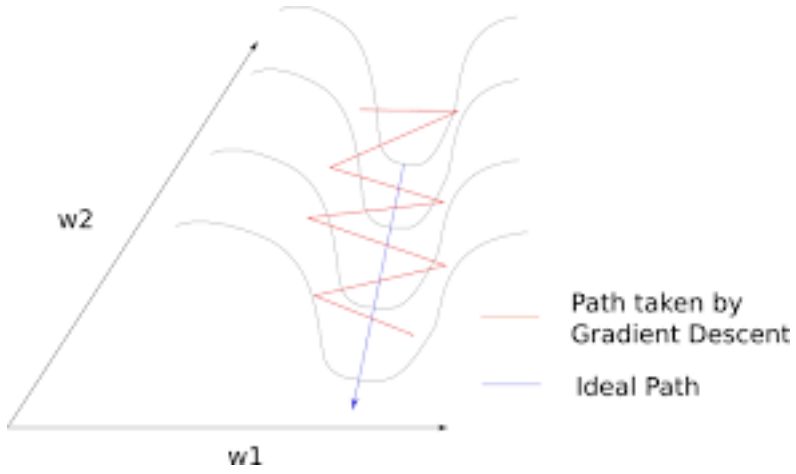


Figure 3.2: Gradient descent direction on a pathological curve[Pat]

Definition. *Newton's method: Given* $f : \mathbb{R} \to \mathbb{R}$ *be a differentiable function, let* $x_i \in \mathbb{R}$ *then* x_{i+1} *is given by :*

$$x_{i+1} = x_i - \frac{f(x_i)}{f`(x_i)}.$$

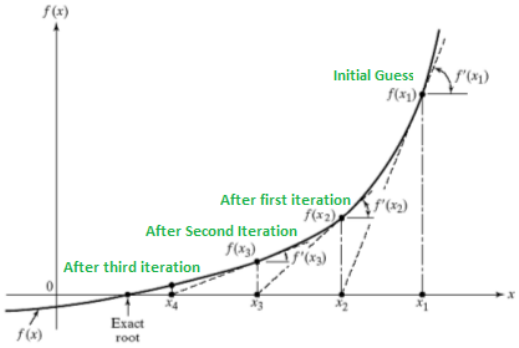


Figure 3.3: Iterations in Newton's method[met]

2**nd order Newton's method**: Any differentiable function f can be locally approximated around θ by:

for small p,

$$f(\theta + p) = g_\theta(p) \approx f(\theta) + p\nabla f(\theta) + \frac{1}{2}p^T \nabla^2 f(\theta) p,$$

here $\nabla^2 f(\theta) = H \to$ Hessian of $f(\theta)$ and p̂ is the direction of the minima.

Assuming that H is symmetric positive-definite (SPD),the optimization of the approximation can be done by solving,

$$\nabla^2 H p = -\nabla f(\theta).$$

Now using Newton's method is impractical on a deep learning setup due to following challenges:

- Due to quadratic relation between hessian and the no. of parameters, which means increasing layers will increase the required computation for the hessian by the order of 2.
- Also Hessian can be indefinite, in which case our f may not have a minimum.
- For large values of p, we cannot trust the above approximation.

But we have a way around to all the challenges above

Chapter 4

Hessian-free Newton's method

2nd order optimization method are theoritically more efficient than any 1st order method as 2nd order methods can utilise the curvature information to decide a better descent path for the function.

As described in the paper [Mar10], if the curvature is low in some direction d, means that the gradient of the function changes slowly along d. This small change in gradient means that the minima of the curvature is far in the curve and hence choosing a direction p on the curve, which travels far along the d will be best even though the reduction in the function is very small. Similarly, if curvature of the function is small, then choosing a direction p which travels small distance along d will be more efficient.

Our optimization method must take care of the above phenomena otherwise it could lead to some unwanted results.

- A *bouncing behaviour* can be observed.
- Low curvature will be explored very slowly. If the only direction of descent are the ones with low curvature then that would lead to very impractically slow exploration and even appear like the function has reached a local minima.

Newton's method updates the function by,

$$x_{n+1} = x_n - \frac{\nabla f(x_n) d^T}{dH(f(x_n)) d^T}, \tag{4.1}$$

here d is the descent direction vector. So basically Newton's method does the computation of change in function ($\nabla f(x_n) d^T$) with respect to the curvature ($dH(f(x_n)) d^T$) by definition.

As introduced, Newton's method is impractical to be used for machine learning. We can make it practical by modifying our Newton's method to overcome the challenges described before.

4.1 Hessian-free Optimization

Hessian-free is the basis of the 2nd order optimization approach that we are going to study. Now standard Newton's method uses equation 4.1 to optimize $g_x(p)$ by computing the Hessian matrix B and then solving,

$$Bp = -\nabla f(x).$$

Computing B is going to be expensive for deep networks, instead, Hessian-free optimizes $g_x(p)$ by utilizing two simple techniques.
First technique that Hessian-free uses is called finite-difference method.

Definition. *Finite-difference: Let B is $N \times N$ matrix and p is N-dimensional vector, then matrix-vector product Bp can be computed using finite-difference method,*

$$Bp = \lim_{\varepsilon \to 0} \frac{\nabla f(x+\varepsilon p) - \nabla f(x)}{\varepsilon}.$$

This method can compute the matrix-vector product at the cost of a single extra gradient computation .

4.1.1 Conjugate Gradient algorithm

Second technique is a very effective algorithm for optimizing quadratic functions such as $g_x(p)$.

Definition. *Let A is $N \times N$ matrix be symmetric and positive definite. We say that a vector p is A-conjugate if,*

$$p^T Ap = 0.$$

Lemma. *If A is positive definite matrix and the set of nonzero vectors $p_0, p_1, p_2, p_3, \cdots, p_k$ are pairwise A-conjugate, then these vectors are linearly independent.*

Proof. Suppose $p_1, p_2, \cdots, p_k$ not all zero, are not linearly independent. Assuming coefficient $\rho_0 \neq 0$,

$$\rho_0 p_0 + \rho_1 p_1 + \cdots + \rho_k p_k = 0,$$

multiplying both side by $p_0^T A$, we get,

$$\rho_0 p_0^T A p_0 + \rho_1 p_0^T A p_1 + \cdots + \rho_k p_0^T A p_k = 0,$$

where all but the first term vanishes because of *A-conjugacy*.
$\implies \rho_0 p_0^T A p_0 = 0.$
On the other hand, since $p_0 \neq 0$ and $A > 0$, we must have $p_0^T A p_0 > 0 \implies \rho_0 p_0^T A p_0 \neq 0$.
Hence, contradiction. □

Since set of p's are linearly independent, they span the whole space $\mathbb{R}^n$. We can express the difference between the exact solution x^* and x_0 (our initial guess) as a linear combination of the conjugate vectors, in turn expressing the exact solution as the sum of the linear combination and x_0:

$$x^* = x_0 + \rho_0 p_0 + \rho_1 p_1 + \cdots + \rho_{n-1} p_{n-1}. \tag{4.2}$$

Till now we are using the assumption that the set of A-conjugate directions exist. Practically we have to create such vectors and there are many ways to do so.
The most storage and computation effective method to create A-conjugate set is Conjugate direction method. It turns out that using the conjugacy property each new p_k can be computed using p_{k-1} given by:

$$p_k = -r_k + \beta_k p_{k-1}$$

Here r_k is the gradient(nd hence the method is called Conjugate gradient method) and β_k is given by:

$$\beta_k = \frac{r_k^T A p_{k-1}}{p_{k-1}^T A p_{k-1}} = \frac{r_k^T r_k}{r_{k-1}^T r_{k-1}}.$$

4.1.2 Damping

Overshooting or divergence from the actual minimum is a flaw in our Hessian-free approach. The curvature matrix in Hessian-free method is not any approximation rather it is the exact curvature data of the objective function. So it allows for the identification of a

descent direction which has very low curvature. When such a direction is identified, and also the direction has reasonably large reduction then Conjugate Gradient method will tend to move far in the direction causing it to overshoot from the approximate minimum.
To overcome damping, damping parameter λ can be accommodated with the hessian matrix as :

$$H + \lambda I = -\nabla f(x).$$

λ will control how 'conservative' the approximation is, corresponding to each direction d.
We can adjust λ using Levenberg-Marquardt hueristic technique:
Define,

$$\rho_k = \frac{f(\theta^k + \alpha_k d^k) - f(\theta^k)}{\alpha_k \nabla f(\theta^k)^T d^k + \frac{1}{2}(\alpha_k)^2 (d^k)^T G^k d_k} \tag{4.3}$$

as the ratio between the actual function reduction and the predicted function.
Based on ρ_k, λ_{k+1} can be computed by,

$$\lambda_{k+1} = \begin{cases} \lambda_k \times drop & \rho_k > 0.75, \\ \lambda_k & 0.25 \leq \rho_k \leq 0.75, \\ \lambda_k \times boost & otherwise, \end{cases} \tag{4.4}$$

where *drop* and *boost* are given constants. [Giba] So if the predicted reduction is close to true reduction, then the choice of direction closer to the Newton direction is considered by reducing λ_k, else if the predicted reduction is far from the true reduction, then λ_k is adjusted to iterate in a direction away from Newton direction.

4.1.3 Gauss-Newton matrix

[Giba] Conjugate gradient method assumes that Hessian is positive semidefinite. But this assumption is not always true and Hessians can be non positive definite. In order to use conjugate gradient method on our quadratic objective function, we need to approximate the Hessian such that the approximation will be positive semidefinite.

Gauss-Newton matrix: Let us denote Gauss-Newton matrix as G. It is an approximation of Hessian H such that that G is always positive semi-definite. It is helpful in implementation of Conjugate Gradient method, because now we dont have to assume the convexity of H.

Given the objective function $f(x) = \sigma(g(x))$ where σ is the activation function, Hessian H is given by

$$H_{ij} = \nabla^2 f(x) = \frac{\partial}{\partial x_j}\frac{\partial f(X)}{\partial x_i} = \frac{\partial}{\partial x_j}\frac{\partial \sigma(g(x))}{\partial x_i}.$$

After applying chain rule,

$$H_{ij} = \sum_{k=0}^{n} \frac{\partial}{\partial x_j}\left(\frac{\partial \sigma}{\partial g_k(x)}(f(x))\frac{\partial g_k(x)}{\partial x_i}\right),$$

then applying product rule:

$$H_i j = \sum_{k=0}^{n} \frac{\partial}{\partial x_j}\left(\frac{\partial \sigma}{\partial g_k(x)}(g(x))\right)\frac{\partial g_k(x)}{\partial x_i} + \sum_{k=0}^{n}\frac{\partial \sigma}{\partial g_k(x)}(g(x))\frac{\partial g_k(x)}{\partial x_i x_j},$$

similar calculation in individual summands gives us

$$\vdots$$

$$H_{ij} = \sum_{k=0}^{n}\sum_{l=0}^{n}\frac{\partial \sigma^2}{\partial g_l(x) g_k(x)}(g(x))\frac{\partial g_l(x)}{\partial x_j}\frac{\partial g_k(x)}{\partial x_i} + \sum_{k=0}^{n}\frac{\partial \sigma}{\partial g_k(x)}(g(x))\frac{\partial g_k(x)^2}{\partial x_i \partial x_j}.$$

Now Gauss-Newton method forms a positive semi-definite approximation of hessian by neglecting the second term. This new matrix is :

$$G = J_f^T H_\sigma J_f$$

Now that we have defined an approximation of H i.e G, we dont have to compute and store all G's, instead we can compute the Gv (matrix-vector) multiplication.

$$Gv = J_f^T H_\sigma J_f v$$

This will be done in two steps,

1. compute the vector Jv.
2. compute the matrix $v^{`} = H_\sigma Jv$.
3. compute the vector $J^t v^{`}$.

Let's compute $v^{`}$ using $\mathscr{R}\{\cdot\}$ method given by:

$$Jv = \mathscr{R}_v f(x) = \frac{\partial}{\partial \varepsilon} f(x+\varepsilon v)|_{\varepsilon=0} = \lim_{\varepsilon \to 0}\frac{f(x+\varepsilon v) - f(x)}{\varepsilon}.$$

where $f : \mathbb{R}^n \to \mathbb{R}^m$ is a vector valued function.

Now to compute $v^{`} = H_\sigma Jv$, we will use Backward $\mathscr{R}\{\cdot\}$ algorithm.

With that computation done, we need to figure out a way to compute $J^T v^{'}$. For that we can make use of the backpropagation method which converts gradient computation as:

$$(\nabla E)_i = f(x)\frac{\partial f(x)}{\partial x_i},$$

where E is the error function and $f(x)$ is the output vector of last layer.

$$(J^T f(x))_i = \left(\begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_2}{\partial x_1} & \cdots & \frac{\partial f_n}{\partial x_1} \\ \frac{\partial f_1}{\partial x_2} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_2} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial f_1}{\partial x_m} & \frac{\partial f_2}{\partial x_m} & \cdots & \frac{\partial f_n}{\partial x_m} \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{bmatrix} \right)_i = \sum_{k=0}^{n} f_k \frac{\partial f_k}{\partial x_i}.$$

Now if we insert vector v in place of the output vectors, it simply multiplies by J^T.

Chapter 5

Simulation

5.1 Feed-forward Neural network

[Wan+18] Construction of Feed-forward network is as follows:

Let n_m denote the no. of nodes in mth layer such that n_0 is the no. of features and n_L is the no. of classesuch thathw weight matrix W^m and the bias vector b^m at the mth layer are

$$W^m = \begin{bmatrix} w^m_{11} & w^m_{12} & \cdots & w^m_{1n_m} \\ w^m_{21} & w^m_{22} & \cdots & w^m_{2n_m} \\ \vdots & \vdots & \vdots & \vdots \\ w^m_{n_{m-1}1} & w^m_{n_{m-1}2} & \cdots & w^m_{n_{m-1}n_m} \end{bmatrix} \quad \text{and} \quad b^m = \begin{bmatrix} b^m_1 \\ b^m_2 \\ \vdots \\ b^m_{n_m} \end{bmatrix}$$

We will consider the input layer as layer 0, i.e

$$s^{0,i} = z^{0,i} = x^i$$

, where $s^{m,i}$ and $z^{m,i}$ are the respective input vector and output vector for the ith instance of the mth layer and x^i is the ith feature vector in the dataset.

The calculation between $(m-1)$th and mth layer is given by:

$$s^{m,i} = (W^m)^T z^{m-1,i} + b^m, m = 1, \cdots, L, i = 1, \cdots, l$$

$$z^{m,i}_j = \sigma(s^{m,i}_j), j = 1, \cdots, n_m, m = 1, \cdots, L, i = 1, \cdots, l \tag{5.1}$$

here $\sigma(\cdot)$ is the activation function.

We will modify weight matrix for the construction of weight vector of the whole neural network.

Concatenate the weight matrix of mth layer,

$$w^m = [w^m_{11} \cdots w^m_{n_{m-1}1} w^m_{12} \cdots w^m_{n_{m-1}2} \cdots w^m_{1n_m} \cdots w^m_{n_{m-1}n_m}]^T$$

The weight matrix of the whole neural network is given by:

$$\theta = \begin{bmatrix} w^1 \\ b^1 \\ \vdots \\ w^L \\ b^L \end{bmatrix} \tag{5.2}$$

This completes our neural network with L layers and n parameters given by:

$$n = \sum_{m=1}^{L} (n_{m-1} \times n_m + n_m)$$

5.2 Algorithms

Algorithm 1: Trust region method (Levenberg-Marquardt) [Bö]

Initialization:

initial approximation x_0,

maximum step length $\bar{\lambda}$,

initial trust region approximation $\lambda_0 \in (0, \bar{\lambda})$,

acceptance constant $\phi \in [0, 1/4]$;

For $k = 0, 1, \cdots$ **until** x_k **is optimal:**

- Solve,

$$\min_p m_k(p) = f_k + p^T \nabla f_k + 1/2 p^T H_k p,$$

  such that $||p|| \leq \lambda_k$, approximated for a trial step p_k.

- Calculate $\rho_k = \frac{f(x_k) - f(x_k + p_k}{m_k(0) - m_k(p_k)}$

- Update the current point,

$$x_{k+1} = \begin{cases} x_k + p_k & \text{if } p_k > \phi \\ x_k & otherwise \end{cases}$$

- Update the trust region,

$$\lambda_{k+1} = \begin{cases} 1/4\lambda_k & \text{if } \rho_k < 1/4, \\ \min(2\lambda_k, \bar{\lambda}) & \text{if } \rho_k < 1/4 \text{ and } \|\rho_k\| = \lambda_k \\ \lambda_k & otherwise. \end{cases}$$

Algorithm 2: Conjugate Gradient algorithm [Gibb]

$i = 0$, $x_i = x_0$ be our initial point;

- **Find best step size:** compute α to minimize the function $f(x_i + \alpha d_i)$: $\alpha = -\frac{d_i^T (Ax_i + b)}{d_i^T A d_i}$.

- **Update the guess:** $x_{i+1} = x_i + \alpha d_i$.

- **Iterate:** Repeat until all n directions are explored.

Algorithm 3: Hessian-free optimization [Cha]

For every epoch:

- $g_n \leftarrow \nabla f(\theta_n)$
- Compute λ using *Algorithm 1*.
- Define, $B_n = G_n v + \lambda v$,G_n as approximation of Hessian.
- $CG - Minimize(B_n, g_n)$ using *Algorithm 2*.
- Update θ by :

$$\theta_{n+1} = \theta_n + p_n$$

5.3 Running Hessian-free Newton's method

Xor Dataset : An XOR gate is a digital logic gate with two or more inputs and one output that performs exclusive disjunction.

Table 5.1: Logic table

input A	input B	output
1	1	0
1	0	1
0	1	1
0	0	0

For the example run, we are generating a random dataset of 2 columns, inputs and outputs respectively of 50000 rows. Input consists of two columns, input A and input B, similar to the table above.

From this test run, we want to test our hypothesis, that Hessian free Newtons method is faster than Gradient descent based algorithm in general. We will use Stochastic Gradient Descent(SGD) method for the comparison.

SGD : η = 0.001, Hessian-Free

The updates in Hessian-free are not same as SGD visually. Before completing one update, Hessian-free undergoes conjugate gradient iterations which is not visible in the plot 5.1. The

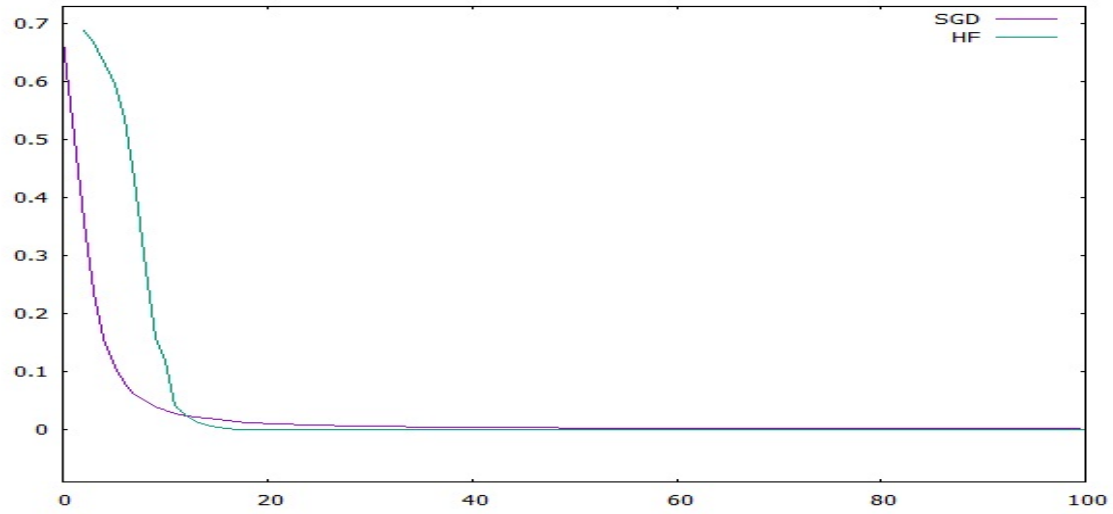


Figure 5.1: Cost Vs Instance comparison between Hessian-free and SGD method

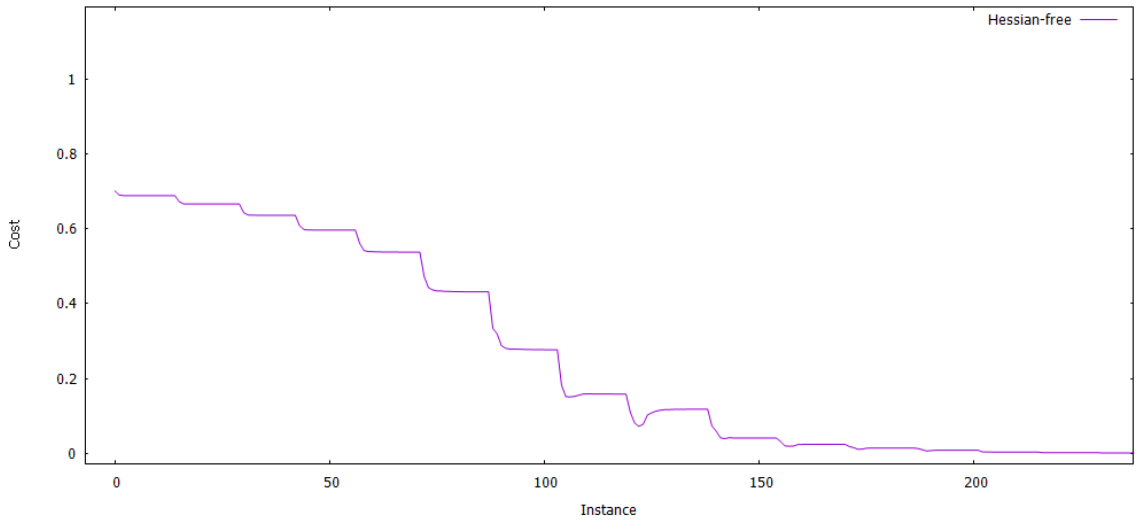


Figure 5.2: Cg iteration incorporated Hessian-free Cost vs Instance

conjugate gradient updates are averaged over an update and then used for the plot.Plot 5.2 show the actual updates including the conjugate gradient updates in Hessian-free method. The grooves in the plot 5.2, are the iterations due to conjugate gradient updates in the Hessian-free method. Due to these small iterations, the performance of Hessian-free has improved over the updates as compared to SGD.

Conclusion

The simulation over XOR dataset is a test run and the plots do not prove that Hessian-free is faster than gradient descent based method.
However, plot 5.1 brings strength to the hypothesis that Hessian-free is faster than gradient descent based methods, as it can be seen that after 10th update instance Hessian-free passes SGD in cost reduction.

Bibliography

[Ahm14] Amir Ali Ahmadi. *Lec10p1*. 2014.
URL: http://www.princeton.edu/~amirali/Public/Teaching/ORF363_COS323/F14/ORF363_COS323_F14_Lec10.pdf.

[Bö] Niclas Börlin. *Trust region and Levenberg-Marquardt method*.
URL: https://www8.cs.umu.se/kurser/5DA001/HT13/lectures/C4.pdf.

[Cha] Carlos Eduardo Cancino Chacon. *Hessian Free optimization*.
URL: www.ofai.at/~maarten.grachten/downloads/lrn2cre8/dlmp-workshop/slides/carlos/Hessian_Free.pdf.

[Giba] Andrew Gibiansky. *Gauss newton matrix*.
URL: http://andrew.gibiansky.com/blog/machine-learning/gauss-newton-matrix/.

[Gibb] Andrew Gibiansky. *Hessian free Optimization*.
URL: http://andrew.gibiansky.com/blog/machine-learning/hessian-free-optimization/.

[Mar10] James Martens. "Deep learning via Hessian-free optimization." In: *ICML*. Vol. 27. 2010, pp. 735–742.

[met] Newton-Raphson method.
URL: https://www.geeksforgeeks.org/program-for-newton-raphson-method/.

[Mor78] Jorge J Moré. "The Levenberg-Marquardt algorithm: implementation and theory". In: *Numerical analysis*. Springer, 1978, pp. 105–116.

[Nie] Michael Nielsen. *Neural Networks and Deep learning*. URL: http://neuralnetworksanddeeplearning.com/chap1.html.

[Pat] *Intro to optimization in deep learning: Momentum, RMSprop and Adam*. 2018. URL: https://blog.paperspace.com/intro-to-optimization-momentum-rmsprop-adam/.

[Ray14] Nirmal Rayan. "Performance Optimization of Levenberg-Marquardt Algorithm with Parallelization". In: May 2014.

[Wan+18] Chien-Chih Wang et al. "Distributed newton methods for deep neural networks". In: *Neural computation* 30.6 (2018), pp. 1673–1724.