PES University, Bangalore

(Established under Karnataka Act No. 16 of 2013)

MAY 2020: IN SEMESTER ASSESSMENT (ISA) B.TECH. IV SEMESTER

UE18MA251- LINEAR ALGEBRA

MINI PROJECT REPORT

ON

Applications and Advantages of Linear Algebra in Neural Networks

Submitted by

1. Name Bhavin G Chennur
2. Name Monish S
3. Name Nishanth S Shastry

[redacted]

Table of Contents

Introduction

The idea of momentum acceleration which was introduced by Rumelhart et al in 1986 has been widely used in training neural networks. It is known that the feedforward neural network algorithms often provide an approximation to the trajectory in weight space computed by the gradient method. To speed up and stabilize the training iteration procedure of the algorithms, a momentum term is often added to the increment formula for the weights ([1][2]).

Our present investigation is on applying concepts of neural networks to linear algebra when linear algebra is used in neural networks, to investigate this it is extremely important for us to justify that linear algebra is used in neural networks, for this we divide our project into two major parts. One part justifies the use of linear algebra in neural networks and the second part applies concepts of neural networks to linear algebra to give better solutions.

Application of Linear Algebra in Neural Networks:

Let us be familiar with the terms…

1. **What is Linear Algebra?**

Linear algebra *is the branch of mathematics concerning* ***linear*** *equations and* ***linear*** *functions and their representations through matrices and vector spaces.*

2. **What is Neural Network?**

A ***neural network*** *is a series of algorithms that endeavor's to recognize underlying relationships in a set of data through a process that mimics the way the human brain operates. In this sense,* ***neural networks*** *refer to systems of neurons, either organic or artificial in nature.*

3. What is Deep Learning?

Deep learning *is a subset of* ***machine learning*** *in artificial intelligence (AI) that has networks capable of* ***learning*** *unsupervised from data that is unstructured or unlabelled. Also known as* ***deep*** *neural* ***learning*** *or* ***deep neural network****.*

The reason we bought in Deep Learning to this document is Neural Networks and Deep Learning are so interrelated. Below this sentence, we can understand what specifically differentiates the above two terms.

4. Is Neural Network and Deep Learning same?

Neural Networks *use neurons to transmit data in the form of input values and output values through connections,* ***Deep Learning*** *is associated with the transformation and extraction of feature which attempts to establish a relationship between stimuli and associated neural responses present in the brain.*

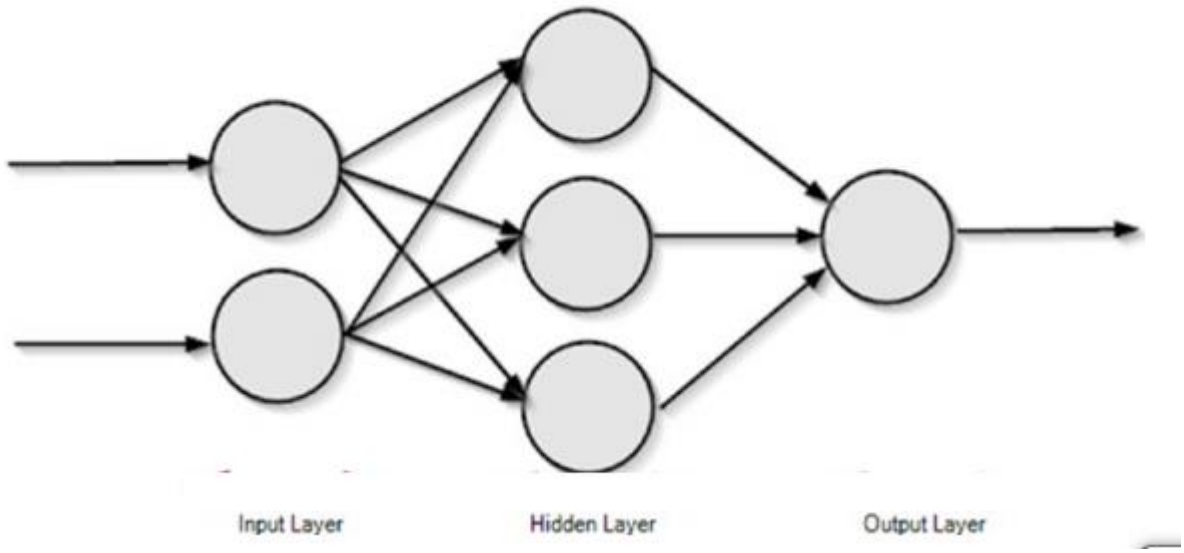
As we are clear with the terms we will be mainly using throughout this document, let us dive into how **Neural Networks** can be enhanced with the help of **Linear Algebra**.

LINEAR ALGEBRA USING NEURAL NETWORKS

If you start to learn deep learning, the first thing you will be exposed to is the feed forward neural network, which is the simplest and also highly useful network in deep learning. Under the hood, the feed forward neural network is just a composite function, that multiplies some matrices and vectors together.

The main highlight here is, vectors and matrices are the one of the ways to do these operations and this method is highly efficient.

To elaborate it, here's an example…



The image above shows a simple feed forward neural network that propagates information forwards.

The above image is a beautiful representation of the neural network, but how a computer can understand this. In a computer, the layers of the neural network are represented as vectors. Consider the input layer as X and the hidden layer as H. The output layer is not concerned for now. (The computation process of feed forward neural networks is not concerned here.)

So, it can be represented as a vectors and matrices as,

$$X = \begin{bmatrix} x1 \\ x2 \end{bmatrix} \quad vector\ (input\ layer)$$

$$W = \begin{bmatrix} w1 & w2 \\ w4 & w5 \\ x3 & w6 \end{bmatrix} \quad matrix\ (\ the\ weights\ for\ hidden\ layer\ 1)$$

$$the\ output\ is\ given\ by$$

$$\begin{bmatrix} h1 \\ h2 \\ h3 \end{bmatrix} = \begin{bmatrix} w1 & w2 \\ w4 & w5 \\ x3 & w6 \end{bmatrix} \cdot \begin{bmatrix} x1 \\ x2 \end{bmatrix} \quad (the\ product\ of\ vector\ and\ matrices)$$

$$\begin{bmatrix} h1 \\ h2 \\ h3 \end{bmatrix} = \begin{bmatrix} h1 \\ h2 \\ h3 \end{bmatrix} + bias$$

$$finally,$$

$$\begin{bmatrix} h1 \\ h2 \\ h3 \end{bmatrix} = f\left(\begin{bmatrix} h1 \\ h2 \\ h3 \end{bmatrix}\right) (activation\ step)$$

The above image shows the operations needed to compute the output for the first and the only hidden layer of the above neural network (the computation for the output layer is not shown). Let's break it down.

The above image shows the operations needed to compute the output for the first and the only hidden layer of the above neural network (the computation for the output layer is not shown). Let's break it down.

Every single column of the network are vectors. Vectors are dynamic arrays that are a collection of data (or features). In the current neural network, the vector '**x**' holds the input. It is not mandatory to represent inputs as vectors but if you do so, they become increasingly **convenient to perform operations in parallel**.

Deep learning and in specific, neural networks are computationally expensive, so they require this nice trick to make them compute faster.

It's called vectorization. They make computations extremely faster. This is one of the main reasons *why GPUs are required for deep learning*, as they are specialized in vectorized operations like matrix multiplication.

(You can refer the **blue** highlighted link above, to be short we can say Deep learning -Neural Network requires advanced hardware so that things like vectorization can be done easier.)

The hidden layer H's output is calculated by performing **H = f(*W*.x + b).**

Here W is called as the Weight matrix, b is called bias and f is the activation function.(To understand Feed Forward Network, look here.)

Let's breakdown the equation,

the first component is ***W***. ***x*** ; this is a **matrix-vector product,** because W is a matrix and ***x*** is a vector.

Before getting into multiplying these, let's get some idea about the notations: *usually vectors are denoted by small bold italic letters(like **x**) and matrices are denoted by capital bold italic letters(like **X**).If the letter is capital and bold but not italic then it is a tensor(like* **X***).*

In a computer science perspective:

Scalar: A single number. Vector : A list of values.(rank 1 tensor)

Matrix: A two dimensional list of values. (rank 2 tensor)

Tensor: A multi dimensional matrix with rank n.

Getting deeper into the above problem:

consider the W. x product :

$$\begin{bmatrix} h1 \\ h2 \\ h3 \end{bmatrix} = \begin{bmatrix} w1 & w2 \\ w3 & w4 \\ w5 & w6 \end{bmatrix} \cdot \begin{bmatrix} x1 \\ x2 \end{bmatrix}$$

this is done by taking each row of the first matrix and doing elemnt wise multiplication with each column of the second matrix.
thus,

$$\begin{aligned} h1 &= w1.x1 + w2.x2 \\ h2 &= w3.x1 + w3.x2 \\ h3 &= w5.x1 + w6.x2 \end{aligned}$$

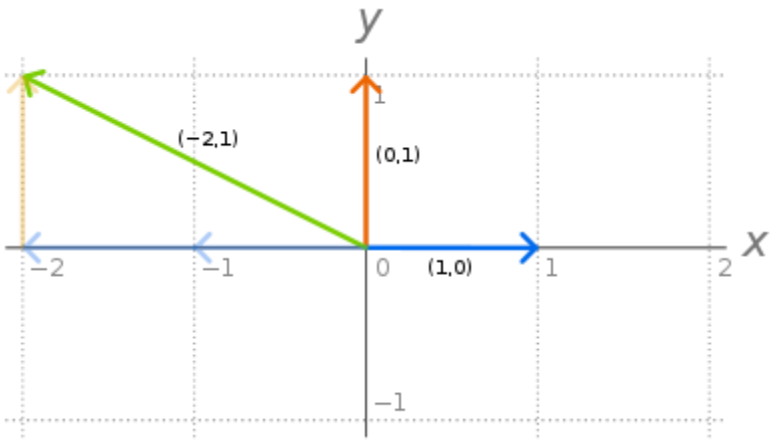
another example,

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{bmatrix}$$

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \cdot \begin{bmatrix} j & k & l \\ m & n & o \\ p & q & r \end{bmatrix} = \begin{bmatrix} aj+bm+cp & ak+bn+cq & al+bk+cl \\ dj+em+fp & dk+en+fq & dl+eo+fr \\ gj+hm+ip & gk+hn+iq & gl+ho+ir \end{bmatrix}$$

In a mathematical perspective:

Vector: A vector is a quantity that has both magnitude and direction. It is an entity that exists in space, it's existence is denoted by $\boldsymbol{x \in \mathbb{R}^2}$*if it is a 2 dimensional vector that exists in real space.(Each element denotes a coordinate along a different axis.)*



All vectors in 2D space can be obtained by linear combination of the two vectors called **basis vectors**.(denoted by i and j)(In general, a vector in N dimensions can be represented by N

basis vectors.)They are **unit normal vectors because their magnitude is one and they are perpendicular to each other**. One of these two vectors can't be represented by the other vector. So they are called as **linearly independent vectors**. (If any vector can't be obtained by a linear combination of a set of vectors, then the vector is linearly independent from that set). All the set of points in the 2D space that can be obtained by linear combination of these two vectors are said to be the **span** of these vectors. If a vector is represented by a linear combination(addition, multiplication) of set of other vectors, then it is **linearly dependent** on that set of vectors.(there is no use in adding this new vector to the existing set.)

Matrix:

A matrix is a 2D array of numbers. They represent **transformations**. Each column of a 2 * 2 matrix denotes each of the 2 basis vectors after the 2D space is applied with that transformation. Their space representation is $W \in \mathbb{R}^{3*2}$ ***having 3 rows and 2 columns.***

A matrix vector product is called transformation of that vector, while a matrix matrix product is called as composition of transformations.

There is only one matrix which does not does any transformation to the vector. It is the identity matrix(***I***). ***The columns of I represent the basis vectors.***

Determinant of the matrix ***A,*** denoted by det(***A***)the the scaling factor of the linear transformation described by the matrix.

Why is a mathematical perspective important to deep learning researchers? Because they help us understand the basic design concepts of fundamental objects.

Python Coding Involving Matrices:

Numpy is a library for python programming language.

```
In [1]: import numpy as np

In [4]: x = np.array([1,2])
        w = np.random.randn(3,2)
        b = 1

        h = w.dot(x) + b

In [12]: h.shape
Out[12]: (3,)

In [24]: h=h.reshape((3,1))

In [25]: h
Out[25]: array([[-0.56692122],
                [-1.46837044],
                [-2.21104227]])

In [18]: h.shape
Out[18]: (3, 1)

In [21]: h = np.transpose(h)

In [22]: h
Out[22]: array([[-0.56692122, -1.46837044, -2.21104227]])
```

Here, np.array creates a numpy array.

np.random is a package that contains methods for random number generation.
the dot method is to compute product between matrix.

We can change the shape of the numpy array and also check it.

Here you can see that, the product of W.x is a vector and it is added with b, which is a scalar. This automatically expands b as transpose ([1,1]). ***This implicit copying of b to several locations is called as broadcasting.***

Second part: Advantages of applying the concepts of Neural Networks to Linear Algebra:

When a linear system Ax = b is not solvable we can take an indirect method to solve the system (anyhow, indirect approach works even when the system is solvable), the indirect method would give an approximate solution to the system with a small error. Stationary iterative methods are indirect methods to solve a linear system of equations, it gives a sequence of values before it arrives at an approximate solution to the linear system, if the sequence is divergent the method fails to give a solution, otherwise if the sequence is convergent in nature we get an approximate solution. Introducing a momentum factor to stationary iterative methods accelerates the convergence and if some stationary iterations are divergent adding a momentum term can make the sequence convergent.

Review of Literature

Although it is well known that such a momentum term greatly improved the speed of learning, there have been few rigorous studies of its convergence property. Ning Qian [3] studies its mechanisms, it shows that in the limit of continuous time, the momentum parameter is analogous to the mass of Newtonian particles that move through a viscous medium in a conservative force field, and the behavior of the system near a local minimum is equivalent to a set of coupled and damped harmonic oscillators. The momentum term improves the speed of convergence by bringing some eigen components of the system closer to critical damping. Phansalkar and Sastry [4] give a stability analysis for the BP (back-propagation) algorithm with momentum. They show that the stable points of the BP algorithm with momentum are local minima of the least squares error, and other equilibrium points are unstable. The convergence of the BP algorithm with momentum is also considered by Bhaya [5] and Torii [6]. They require the gradient of the error function to be a linear function of the weight, and in [5] it is proved that the BP algorithm with momentum is equivalent to the conjugate gradient method in a certain sense. In [7], [8], [9], [10], some convergence results are given for two-layer feedforward and multi-layer neural networks with momentum, respectively. These results are of global nature in that they are valid for any arbitrarily given initial values of the weights.

In this paper, we will discuss some other applications for momentum algorithms in numerical algebra, that is, we will use the momentum term to accelerate the stationary iterations for solving linear systems.

Report on present investigation

Stationary Iterative Methods with Momentum Acceleration

In this section we use the momentum term to accelerate the stationary iterations for solving linear systems.

Consider the iterative solution of a linear system of equations

$Ax = b$ – (1)

where $A = R^{m \times m}$ is nonsingular, and $x,b \in R^m$.

To solve we split $A = M - N$,

Here M is an inverible matrix.

There are various ways of choosing M and N, in this project we give solutions that work for any M and N

$\Rightarrow (M - N)x = b$

$\Rightarrow Mx - Nx = b$

$\Rightarrow Mx = Nx + b$

$\Rightarrow x = (M)^{-1}Nx + (M)^{-1}b$ – (2)

Let $T = (M)^{-1}N$

$\Rightarrow T = (M)^{-1}(M - A)$

$\Rightarrow T = I - M^{-1}A$ – (3)

Using (3) in (2) we get :

$\Rightarrow x = Tx + (M)^{-1}b$

As we are assigning a value to x using it's previous value we can write it as :

$\Rightarrow x^{(k+1)} = Tx^{(k)} + (M)^{-1}b \quad , k = 0,1,\ldots \quad -(4)$

here T is called an iterative matrix.

the iteration (4) converges if and only if the spectral radius $\rho(T)$ of the iteration matrix T satisfies $\rho(T)<1$.

- An added fact,

An iterative method is linear if there exists a matrix

$T \in R^{n\times n}$ such that

$e^{k+1} = Te^k$ for all k>=0,

Here, if x* is the actual value and x^k is the approximate value, $e^k = x^k - x^*$

- An iterative method with a given iteration matrix T is called convergent if the following holds

  1) $\rho(T) < 1$
  2) the limit of the sequence $\{T^k\}$ is zero as $k \rightarrow \infty$,

this sequence $\{T^k\}$ is all the values of x the iterative method gives.

- It is known that the magnitude of $\rho(T)$ decides the convergence rate for $\{T^k\}$,

  the smaller $\rho(T)$ is, the bigger is the asymptotic rate of convergence (i.e. , $\ln \rho(T)$) .

So for iteration (4),

if the spectral radius of its iteration matrix T is very close to 1, then T^k tends to zero very slowly.

This ends the discussion for stationary iterative methods.

We now make stationary iterative methods stronger with the introduction of momentum factor to this process,

Consider a momentum factor $\mu \in R$, we know add a momentum term

$\mu(x^k - x^{(k-1)})$ to accelerate (4), that is, consider the following iteration:

$$x^{(1)} = Tx^{(0)} + (M)^{-1}b$$

$$x^{(k+1)} = Tx^{(k)} + (M)^{-1}b + \mu(x^k - x^{(k-1)}) \quad , k = 1,2, \ldots \quad – (5)$$

We can represent (5) into another stationary iteration and look for the momentum factor μ which minimizes the spectral radius $\rho(T)$ of its iteration matrix T.

Let

$$A' = \begin{pmatrix} A - \mu M & \mu M \\ -I & I \end{pmatrix}$$

$$M' = \begin{pmatrix} M & 0 \\ 0 & I \end{pmatrix}$$

then the linear system $Ax = b$ is equivalent to the following linear system:

$$A'z = \begin{pmatrix} A - \mu M & \mu M \\ -I & I \end{pmatrix}\begin{pmatrix} x \\ x \end{pmatrix} = \begin{pmatrix} b \\ 0 \end{pmatrix} \quad – (6)$$

Notice that

$$A' = \begin{pmatrix} I & \mu M \\ 0 & I \end{pmatrix}\begin{pmatrix} A & 0 \\ 0 & I \end{pmatrix}\begin{pmatrix} A & 0 \\ -I & I \end{pmatrix}$$

We know that A is non-singular, all the three individual matrices in the product above are non-singular therefore A' is also non-singular.

To solve (6), we consider a stationary iterative method based on the splitting

$A' = M' - (M' - A')$

and the associated iteration matrix is

$H = I - (M)^{-1}A$ (this formula was derived in eqn (3))

After simplification we get,

$$H = \begin{pmatrix} \mu I + T & -\mu I \\ I & 0 \end{pmatrix} \quad -(7)$$

Eqn. (4),

$$x^{(k+1)} = Tx^{(k)} + (M)^{-1}b \quad , k = 0,1,\ldots$$

was the solution before we introduced the momentum factor.

We need to develop a similar solution for the second iteration we considered that includes the momentum factor.

Let

$$z^k = \begin{pmatrix} x^{(k)} \\ x^{(k-1)} \end{pmatrix}, k = 1,2,\ldots$$

z^k is a replacement for $x^{(k+1)}$, it contains the next term in the sequence.

Because (5) after adding the the momentum term has 3 values of x, see

$$x^{(k+1)} = Tx^{(k)} + (M)^{-1}b + \mu(x^k - x^{(k-1)}) \quad , k = 1,2, \ldots$$

z^k has 2 values of x and we can accommodate the third value in a solution that is similar to:
eqn (4), $x^{(k+1)} = Tx^{(k)} + (M)^{-1}b \quad , k = 0,1,\ldots$

Before we get to the solution for the iteration with the momentum term,

let f = $\begin{pmatrix} M^{(-1)}b \\ 0 \end{pmatrix}$, because we need a matrix representation and we don't have multiple values to represented in f we introduce a zero in the second row.

Finally, the solution for the stationary iteration with the momentum term (as in eqn (5)) is,

$x^{(1)} = Tx^{(0)} + (M)^{-1}b,$

$z^{(1)} = (x^{(1)\wedge T}, x^{(0)\wedge T})^T$

$z^{(k+1)} = Hz^{(k)} + f, \quad k = 1,2, \ldots$ – (8)

Here the initial conditions are given for both x and z.

Conditions on the spectral radius:

We discuss conditions for $\rho(T) < 1$, consider the following results:

Lemma 3.1

Both the roots of the quadratic equation $x^2 - dx + c = 0$ are less than 1 in modulus if and only if $| d - d'c | < 1 - |c|^2$

where d' is the conjugate number of d.

Lemma 3.2

1) If μ = 0, then the two sets of eigenvalues of T and H have the relation:

$\sigma(H) = \sigma(T) \cup \{0\}$

Here the set of eigenvalues of H and T are denoted by σ(H) and σ(T) respectively.

2) If μ! = 0, then for every eigenvalue α of T, there exist two eigenvalues λ of H such that

$\lambda^2 - (\mu + \alpha)\lambda + \mu = 0$ – (9)

Conversely for any eigenvalue λ of H, there exists an eigenvalue α of T which satisfies (9).

Proof

(1) is evident, we now give the proof for (2). Let (α, q) be an eigen pair of T (an eigen pair is a pair of an eigenvalue, and its associated eigenvector), if λ satisfies (9), we now prove it is an eigenvalue of H. Since μ! = 0, from (9) we know λ! = 0 (remember, x = (-b +/- sqrt(b^2 – 4ac)) / 2a). Let p = λq, then

$$H\begin{pmatrix} p \\ q \end{pmatrix} = \begin{pmatrix} \mu I + T & -\mu I \\ I & 0 \end{pmatrix}\begin{pmatrix} \lambda q \\ q \end{pmatrix}$$

$$= \begin{pmatrix} \mu\lambda q + \alpha\lambda q - \mu q \\ \lambda q \end{pmatrix}$$ (we have taken replaced T by its eigenvalue α)

$$= \begin{pmatrix} \lambda^2 q \\ \lambda q \end{pmatrix}$$ (Because $\lambda^2 = (\mu + \alpha)\lambda - \mu$ from (9))

$$= \lambda\begin{pmatrix} p \\ q \end{pmatrix}$$

which means λ is an eigenvalue of H.

Conversely, for any eigen pair of H such that

$$H\begin{pmatrix} p \\ q \end{pmatrix} = \lambda\begin{pmatrix} p \\ q \end{pmatrix}$$

which means p = λq and μp + Tp - μq = λp.

Since μ! = 0, it is easy to know p! = 0, otherwise q = 0, which would contradict the fact that $(p^T, q^T)^T$ is an eigenvector, therefore p! = 0. With p! = 0 it holds that λ! = 0.

(Remember, eqn (9) was $\lambda^2 - (\mu + \alpha)\lambda + \mu = 0$)

Take $\alpha = (\lambda^2 - \mu\lambda + \mu) / \lambda$, then we can easily obtain

$Tp = \alpha p$, which means that α is an eigenvalue of T.

From the proof of **Lemma 3.2** we know that H is non-singular if and only if $\mu \mathrel{!}= 0$.

Theorem 3.3

$\rho(H) < 1$ if and only if for each eigenvalue $\alpha=\xi+i\eta$ of T, $\xi, \eta \in R$.

The point (ξ, η) lies in the interior of the ellipse

$((x + \mu)^2 / (1 + \mu)^2) + (y^2 / (1 - \mu)^2) = 1, \quad -1 < \mu < 1.$

Proof

By **Lemma 3.1** and **Lemma 3.2**, and take $d = \mu + \alpha$ and $c = \mu$, then

(remember that **spectral radius ρ(H)** of a square matrix or a bounded linear operator is the largest absolute value of its eigenvalues)

$\rho(H) < 1 \iff |d - d'c| < 1 - |c|^2$

$\iff |\mu + \xi + i\eta - \mu (\mu + \xi - i\eta)| < 1 - \mu^2$

after further simplification,

$\rho(H) < 1 \iff ((\xi + \mu)^2 / (1 + \mu)^2) + (\eta^2 / (1 - \mu)^2) < 1, -1 < \mu < 1.$

Corollary 3.4

Suppose $\sigma(T) \subseteq (-3,1)$ (the set of eigenvalues) and let α_1 be the smallest eigenvalue of T, then

$\rho(H) < 1$ if and only if $\mu \in ((-1 - \alpha_1) / 2, 1)$.

Notice that it holds $((-1 - \alpha_1) / 2, 1) \subseteq (-1,1)$ since $-3 < \alpha_1 < 1$.

Now we give the optimal momentum factor μ^* which minimizes $\rho(H)$ by the following **Theorem 3.6**. It is sophisticated when T has some complex eigenvalues, so in this paper and for this problem we assume that all the eigenvalues of T are real, and the interval which includes all the eigenvalues of T can be $(-3,1)$.

Theorem 3.6

Let $-3 < \alpha_1 \leq \alpha_2 \leq \ldots \leq \alpha_m < 1$ be all the eigenvalues of T, then the optimal momentum factor μ^* which minimizes $\rho(H)$ is

$\mu^* = \max \{(1 - \text{sqrt}(1 - \alpha_m))^2, (1 - \text{sqrt}(1 - \alpha_1))^2\} \in [0,1)$ – (10)

and

$\min \rho(H) = \text{sqrt}(\mu^*)$ – (11)

Results and Discussions:

From **Theorem 3.3** or **Corollary 3.4** we see that adding a momentum term can accelerate the convergence for stationary iterations. Especially for some stationary iterations which are divergent, after added with a momentum term then they can become convergent.

In **Corollary 3.4** H is the iteration matrix used and it is obtained after the addition of the momentum factor μ.

Let's look at an example that covers the entire concept:

Let the coefficient matrix A of the linear system $Ax = b$ be

$$A = \begin{pmatrix} 8 & 0 & -4 \\ 0 & 4 & 0 \\ -7 & 0 & 5 \end{pmatrix}, \text{ and let } M = \begin{pmatrix} 4 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 4 \end{pmatrix}$$

then the 'iteration matrix T of the iteration' (4) is

$$T = I - M^{-1}A = \begin{pmatrix} -1 & 0 & 1 \\ 0 & 0 & 0 \\ 7/4 & 0 & -1/4 \end{pmatrix}$$

After solving,

$\sigma(T) = \{-2, 0, ¾\}$ so $\rho(T) = 2 > 1$.

If we add the momentum term to iteration (4) with momentum factor $\mu = 0.5359$ then for iteration (8),

$\rho(H) = 0.7321$

Python Implementation:

Importing the necessary Python libraries,

```
In [1]: #Python Libraries
        import numpy as np
        import scipy.linalg as la
        from numpy.linalg import *
```

Taking the inputs of matrices, A and b so as to satisfy the eqn, Ax = b

```
In [2]: print("\nStationary Iterative Methods with Momentum to accelerate the stationary iterations for solving linear systems\n")

        print("Enter the details for Matrix 'A': - ")
        print("")
        R = int(input("Enter the number of rows for Matrix 'A':"))
        C = int(input("Enter the number of columns for Matrix 'A':"))
        print("Enter the entries in a single line (separated by space)\nrow wise for Matrix 'A': ")
        entries = list(map(float, input().split()))

        A = np.array(entries).reshape(R, C)
        print("")

        print("\nA: -", A,"\n")

        print("Enter the details for Matrix 'b': - ")
        print("")
        print("Enter the entries in a single line (separated by space)\nrow wise for Matrix 'b': ")
        entries = list(map(float, input().split()))
        b = np.array(entries).reshape(R)
        print("b: -", b,"\n")
```

After taking the input from the user,

```
Stationary Iterative Methods with Momentum to accelerate the stationary iterations for solving linear systems

Enter the details for Matrix 'A': -

Enter the number of rows for Matrix 'A':3
Enter the number of columns for Matrix 'A':3
Enter the entries in a single line (separated by space)
row wise for Matrix 'A':
8 0 -4 0 4 0 -7 0 5


A: - [[ 8.  0. -4.]
 [ 0.  4.  0.]
 [-7.  0.  5.]]

Enter the details for Matrix 'b': -

Enter the entries in a single line (separated by space)
row wise for Matrix 'b':
1 2 3
b: - [1. 2. 3.]
```

Python code to display the other necessary matrices which will be used for calculation,

```
In [3]: I = np.identity(C)
        print("")
        print("I", I,"\n")

        print("The Matrix 'D': - ")

        D = np.diag(A)
        D = D*I
        print("D: -", D)

        Dinv = np.linalg.inv(D)
        print("")
        print("Dinv: -", Dinv)

        T = I - np.dot(Dinv, A)
        print("")
        print("T: -", T)

        print("\n From Theorem 3.3, we understand that spectral radius ρ(H) of a square matrix or a bounded linear operator")
        print("is the largest absolute value of its eigenvalues\n")

        eigvals, eigvecs = la.eig(T)
        eigvals = eigvals.real
        print("")
        print("Eigen Values of 'T': -", eigvals,"\n")
        print("")
        print("Eigen Vectors of 'T': -", eigvecs,"\n")
```

Output for the above Python code,

```
I [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]

The Matrix 'D': -
D: - [[8. 0. 0.]
 [0. 4. 0.]
 [0. 0. 5.]]

Dinv: - [[0.125 0.    0.   ]
 [0.    0.25  0.   ]
 [0.    0.    0.2  ]]

T: - [[0.  0.  0.5]
 [0.  0.  0. ]
 [1.4 0.  0. ]]

 From Theorem 3.3, we understand that spectral radius ρ(H) of a square matrix or a bounded linear operator
is the largest absolute value of its eigenvalues


Eigen Values of 'T': - [ 0.83666003 -0.83666003  0.        ]


Eigen Vectors of 'T': - [[ 0.51298918 -0.51298918  0.        ]
 [ 0.          0.          1.        ]
 [ 0.85839508  0.85839508  0.        ]]
```

Python code to calculate the matrix H stated in eqn 7,

```
mu = float(input("Enter the 'mu' value to calculate Matrix 'H': - "))
print("mu = ",mu,"\n")

H = np.array([[(np.dot(mu, I))+T, -(np.dot(mu, I))], [I, 0]])
print("H: -", H)

print("\nFrom Theorem 3.6,")
print("Let -3 < α1 < α2 < …  < αm < 1 be all the eigenvalues of T, then the optimal")
print("momentum factor μ* which minimizes ρ(H) is,")
print("μ* = max {(1 - sqrt (1 - αm))2, (1 - sqrt (1 - α1))2} ∈ [0,1)      - (10)")
print("and")
print("min ρ(H) = sqrt(μ*)                                                - (11)")

alpha_max = max(eigvals)
alpha_min = min(eigvals)
print("")
print("alpha_max = ", alpha_max)
print("\nalpha_min = ", alpha_min)

mu_star = max(np.power((1 - np.sqrt(1 - alpha_max)),2), np.power((1 - np.sqrt(1 - alpha_min)),2))
print("")
print("mu_star = ", mu_star)

min_rho_H = np.sqrt(mu_star)
print("")


print("min_rho_H = ", min_rho_H)
```

Output for the above python code,

```
Enter the 'mu' value to calculate Matrix 'H': - 0.5359
mu =  0.5359

H: - [[array([[0.5359, 0.    , 0.5   ],
       [0.    , 0.5359, 0.    ],
       [1.4   , 0.    , 0.5359]])
  array([[-0.5359, -0.    , -0.    ],
       [-0.    , -0.5359, -0.    ],
       [-0.    , -0.    , -0.5359]])]
 [array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]]) 0]]

From Theorem 3.6,
Let -3 < α1 < α2 < …  < αm < 1 be all the eigenvalues of T, then the optimal
momentum factor μ* which minimizes ρ(H) is,
μ* = max {(1 - sqrt (1 - αm))2, (1 - sqrt (1 - α1))2} ∈ [0,1)      - (10)
and
min ρ(H) = sqrt(μ*)                                                - (11)

alpha_max =  0.8366600265340756

alpha_min =  -0.8366600265340756

mu_star =  0.35503316670026464

min_rho_H =  0.5958465966171701
```

There are basically four main methods of stationary iterative methods,

i) Jacobi Iterative Method

ii) Gauss-Seidel Iterative Method

iii) Successive Overrelaxation (SOR)

iv) Symmetric Successive Overrelaxation (SSOR)

We will be implementing Jacobi method and Gauss-Seidel Method with the momentum factor (mu_star) discussed in the previous sections.

i) Jacobi Iterative Method

Algorithm for Jacobi Iterative method,

Choose an initial guess $x^{(0)}$ to the solution x.
for $k = 1, 2, \ldots$
  for $i = 1, 2, \ldots, n$
    $\bar{x}_i = 0$
    for $j = 1, 2, \ldots, i-1, i+1, \ldots, n$
      $\bar{x}_i = \bar{x}_i + a_{i,j} x_j^{(k-1)}$
    end
    $\bar{x}_i = (b_i - \bar{x}_i)/a_{i,i}$
  end
  $x^{(k)} = \bar{x}$
  check convergence; continue if necessary
end

Python code which shows the Jacobi function which will solve the system of equations given to it.

In [4]:
```
print("\nSolving the System by Jacobi Iterative Method Method\n")

def jacobi(A, b, x0, tol, maxiter=200):
    """
    Performs Jacobi iterations to solve the line system of
    equations, Ax=b, starting from an initial guess, ``x0``.

    Terminates when the change in x is less than ``tol``, or
    if ``maxiter`` [default=200] iterations have been exceeded.

    Returns 3 variables:
        1.  x, the estimated solution
        2.  rel_diff, the relative difference between last 2
            iterations for x
        3.  k, the number of iterations used.  If k=maxiter,
            then the required tolerance was not met.
    """
    n = A.shape[0]
    x = x0.copy()
    x_prev = x0.copy()
    k = 0
    rel_diff = tol * 2

    while (rel_diff > tol) and (k < maxiter):

        for i in range(0, n):
            subs = 0.0
            for j in range(0, n):
                if i != j:
                    subs += A[i,j] * x_prev[j]

            x[i] = (b[i] - subs ) / A[i,i]
            x[i] = x[i] + (mu_star*(x[i] - x[i-1]))
        k += 1

        rel_diff = np.linalg.norm(x - x_prev) / np.linalg.norm(x)
        #print(x, rel_diff)
        print("_____________________________________________________________________________________________")
        print(k, "             ", x, "           ", rel_diff)
        x_prev = x.copy()
    print("_____________________________________________________________________________________________\n")
    return x, rel_diff, k
```

```
# Main code starts here
# ---------------------
#x0 = np.zeros(C);
print("Enter the details for Matrix 'x0' (guess) : - ")
print("")
#R = int(input("Enter the number of rows for Matrix 'x0' (guess) : -"))
print("Enter the entries in a single line (separated by space)\nrow wise for Matrix 'x0' (guess) : - ")
entries = list(map(float, input().split()))
x0 = np.array(entries).reshape(R)
tol = 1E-9
maxiter = 200
print("_____________________________________________________________________________________________\n")
print("Iterations      Solution of X                                    Real Difference")
x, rel_diff, k = jacobi(A, b, x0, tol, maxiter)
if k == maxiter:
    print(('WARNING: the Jacobi iterations did not '
           'converge within the required tolerance.'))
print(('The solution is %s; within a tolerance of %g, '
       'using %d iterations.' % (x, rel_diff, k)))
print('Solution error = norm(Ax-b) = %g' % \
           norm(np.dot(A,x)-b))
print('Condition number of A = %0.5f' % cond(A))
print('Solution from built-in functions = %s' % solve(A, b))
jacobi_error = (np.dot(A,x)-b)
jacobi_k = k
jacobi_soln = x
```

Output for the above python code,

```
Solving the System by Jacobi Iterative Method Method

Enter the details for Matrix 'x0' (guess) : -

Enter the entries in a single line (separated by space)
row wise for Matrix 'x0' (guess) : -
0 0 0
```

Iterations	Solution of X	Real Difference
1	[0.16937915 0.61738137 0.59382904]	1.0
2	[0.36087916 0.54939251 0.93928744]	0.3495989317594821
3	[0.47228377 0.50984018 1.31661425]	0.2655950871599116
4	[0.59396541 0.46663916 1.54329176]	0.15182785001398397
5	[0.66706515 0.44068633 1.7833416]	0.1290821227688151
6	[0.74447724 0.41320247 1.93177287]	0.08035928373650257
7	[0.79234386 0.39620823 2.08466072]	0.07112528705301084

(This goes on until 88th iteration)

81	[1.01843782 0.31593738 2.63287549]	4.542297970627722e-09
82	[1.01843783 0.31593738 2.63287549]	3.6338005112859077e-09
83	[1.01843783 0.31593738 2.6328755]	2.9265314127287296e-09
84	[1.01843783 0.31593737 2.63287551]	2.341994079098492e-09
85	[1.01843784 0.31593737 2.63287551]	1.885550699049915e-09
86	[1.01843784 0.31593737 2.63287552]	1.5093968183107016e-09
87	[1.01843784 0.31593737 2.63287552]	1.2148703762036463e-09
88	[1.01843784 0.31593737 2.63287552]	9.72780165999455e-10

```
The solution is [1.01843784 0.31593737 2.63287552]; within a tolerance of 9.7278e-10, using 88 iterations.
Solution error = norm(Ax-b) = 4.60507
Condition number of A = 12.75493
Solution from built-in functions = [1.41666667 0.5        2.58333333]
```

ii) Gauss-Seidel Iterative Method

Algorithm for Gauss-Seidel Iterative method,

Choose an initial guess $x^{(0)}$ to the solution x.
for $k = 1, 2, \ldots$
  for $i = 1, 2, \ldots, n$
    $\sigma = 0$
    for $j = 1, 2, \ldots, i-1$
      $\sigma = \sigma + a_{i,j} x_j^{(k)}$
    end
    for $j = i+1, \ldots, n$
      $\sigma = \sigma + a_{i,j} x_j^{(k-1)}$
    end
    $x_i^{(k)} = (b_i - \sigma)/a_{i,i}$
  end
  check convergence; continue if necessary
end

Python code which shows the Gauss-Seidel function which will solve the system of equations given to it.

In [6]:
```
print("Solving the System by Gauss-seidel Iterative Method Method")

def GS(A, b, x0, tol, maxiter=200):
    """
    Performs Gauss-seidel iterations to solve the line system of
    equations, Ax=b, starting from an initial guess, ``x0``.

    Terminates when the change in x is less than ``tol``, or
    if ``maxiter`` [default=200] iterations have been exceeded.

    Returns 3 variables:
        1.  x, the estimated solution
        2.  rel_diff, the relative difference between last 2
            iterations for x
        3.  k, the number of iterations used.  If k=maxiter,
            then the required tolerance was not met.
    """
    n = A.shape[0]
    x = x0.copy()
    x_prev = x0.copy()
    k = 0
    rel_diff = tol * 2

    while (rel_diff > tol) and (k < maxiter):

        for i in range(0, n):
            d = b[i]
            for j in range(0, n):
                if i != j:
                    d -= A[i][j] * x[j]

            x[i] = d / A[i][i]
            x[i] = x[i] + (mu_star*(x[i] - x[i-1]))
        k += 1

        rel_diff = np.linalg.norm(x - x_prev) / np.linalg.norm(x)
        #print(x, rel_diff)
        print("_____")
        print(k, "             ", x, "           ", rel_diff)
        x_prev = x.copy()
    print("_____\n")
    return x, rel_diff, k
```

```
# Main code starts here
# ---------------------
#x0 = np.zeros(C);
print("Enter the details for Matrix 'x0' (guess) : - ")
print("")
#R = int(input("Enter the number of rows for Matrix 'x0' (guess) : -"))
print("Enter the entries in a single line (separated by space)\nrow wise for Matrix 'x0' (guess) : - ")
entries = list(map(float, input().split()))
x0 = np.array(entries).reshape(R)
tol = 1E-9
maxiter = 200
print("_____________________________________________________________________________________________________\n")
print("Iterations      Solution of X                                    Real Difference")
x, rel_diff, k = GS(A, b, x0, tol, maxiter)
if k == maxiter:
    print(('WARNING: the Gauss-Seidel iterations did not '
           'converge within the required tolerance.'))
print(('The solution is %s; within a tolerance of %g, '
       'using %d iterations.' % (x, rel_diff, k)))
print('Solution error = norm(Ax-b) = %g' % \
          norm(np.dot(A,x)-b))
print('Condition number of A = %0.5f' % cond(A))
print('Solution from built-in functions = %s' % solve(A, b))
GS_error = (np.dot(A,x)-b)
GS_k = k
GS_soln = x
```

Output for the python code,

```
Solving the System by Gauss-seidel Iterative Method Method
Enter the details for Matrix 'x0' (guess) : -

Enter the entries in a single line (separated by space)
row wise for Matrix 'x0' (guess) : -
0 0 0
_____________________________________________________________________________________________________

Iterations      Solution of X                                    Real Difference
_____________________________________________________________________________________________________
1               [0.16937915 0.61738137 0.91514914]               1.0
_____________________________________________________________________________________________________
2               [0.46449957 0.51260383 1.51220579]               0.40543484499200144
_____________________________________________________________________________________________________
3               [0.65704043 0.44424544 1.90173425]               0.21347298640276935
_____________________________________________________________________________________________________
4               [0.7826569  0.39964742 2.1558683 ]               0.12326434653127465
_____________________________________________________________________________________________________
5               [0.86461092 0.37055103 2.32166906]               0.07474035992435445
_____________________________________________________________________________________________________
6               [0.91807892 0.35156812 2.42983989]               0.046660321934147426
_____________________________________________________________________________________________________
7               [0.95296222 0.33918339 2.50041212]               0.02954512738697224
_____________________________________________________________________________________________________
```

(This goes on until 48th iteration)

```
_____________________________________________________________________________________________________
41              [1.01843781 0.31593738 2.63287547]               1.3861251399229814e-08
_____________________________________________________________________________________________________
42              [1.01843782 0.31593738 2.63287549]               9.043282140061298e-09
_____________________________________________________________________________________________________
43              [1.01843783 0.31593738 2.63287551]               5.899969217424825e-09
_____________________________________________________________________________________________________
44              [1.01843783 0.31593737 2.63287552]               3.8492258525886786e-09
_____________________________________________________________________________________________________
45              [1.01843784 0.31593737 2.63287552]               2.511291184777623e-09
_____________________________________________________________________________________________________
46              [1.01843784 0.31593737 2.63287553]               1.6384031341697808e-09
_____________________________________________________________________________________________________
47              [1.01843784 0.31593737 2.63287553]               1.0689181460440756e-09
_____________________________________________________________________________________________________
48              [1.01843784 0.31593737 2.63287553]               6.973774686557008e-10
_____________________________________________________________________________________________________

The solution is [1.01843784 0.31593737 2.63287553]; within a tolerance of 6.97377e-10, using 48 iterations.
Solution error = norm(Ax-b) = 4.60507
Condition number of A = 12.75493
Solution from built-in functions = [1.41666667 0.5        2.58333333]
```

The below code is to calculate the errors in the solutions using the Jacobi and Gauss-Seidal methods with regard to the original solution which can be calculated using the inbuilt functions in python library.

```
In [7]: print("Jacobi method Solution error: -", jacobi_error)
        print("Gauss method Soulution error: -", GS_error)
```

```
Jacobi method Solution error: - [-3.38399938 -0.73625051  3.03531274]
Gauss method Soulution error: - [-3.38399939 -0.73625051  3.03531276]
```

```
In [13]: print("inbuilt function soln", np.linalg.solve(A,b))
         print("Jacobi method soln", jacobi_soln)
         print("Gauss-Seidel soln", GS_soln)
```

```
inbuilt function soln [1.41666667 0.5        2.58333333]
Jacobi method soln [1.01843784 0.31593737 2.63287552]
Gauss-Seidel soln [1.01843784 0.31593737 2.63287553]
```

(As it is seen clearly that there is very little difference between the stationary iterative methods but they are have difference with inbuilt function)

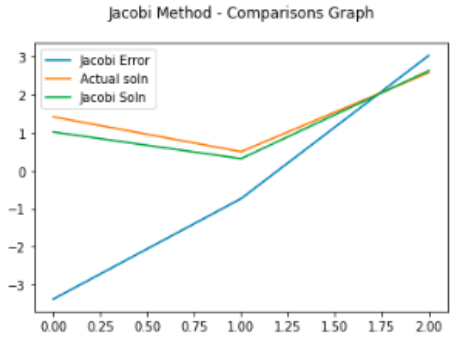
Now we will visualize the solutions from Jacobi and Gauss-Seidal methods and the inbuilt solution. For this to happen we will import the below library,

```
In [8]: import matplotlib.pyplot as plt
```

The below graphs illustrate the comparison graphs of the Stationary methods with the inbuilt function soln's.

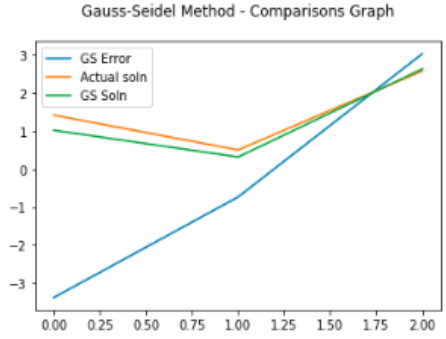
```
In [9]: plt.title("Jacobi Method - Comparisons Graph\n")
        plt.plot(jacobi_error, label='Jacobi Error')
        plt.plot(solve(A,b), label='Actual soln')
        plt.plot(jacobi_soln, label='Jacobi Soln')
        plt.legend()
```

```
Out[9]: <matplotlib.legend.Legend at 0x1f09c0a6688>
```

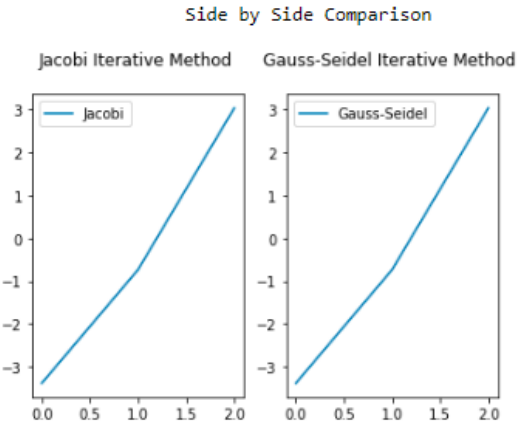



```
In [10]: plt.title("Gauss-Seidel Method - Comparisons Graph\n")
         plt.plot(GS_error, label='GS Error')
         plt.plot(solve(A,b), label='Actual soln')
         plt.plot(GS_soln, label='GS Soln')
         plt.legend()
```

```
Out[10]: <matplotlib.legend.Legend at 0x1f09c180e08>
```



The graphs below show the errors in solving with stationary methods with momentum to solve faster.




```
In [11]: print("\n\n\t\t  Side by Side Comparison")
         fig = plt.figure()
         ax1 = fig.add_subplot(1, 2, 1)
         ax2 = fig.add_subplot(1, 2, 2)
         ax1.plot(jacobi_error, label='Jacobi')
         ax2.plot(GS_error, label='Gauss-Seidel')
         ax1.set_title('Jacobi Iterative Method \n')
         ax1.legend()
         ax2.set_title('Gauss-Seidel Iterative Method \n')
         ax2.legend()
         plt.show()
```

Summary and Conclusions:

Linear algebra, probability and calculus are the 'languages' in which machine learning, Deep Learning, Neural Networks is formulated. Learning these topics will contribute to a deeper understanding of the underlying algorithmic mechanics and allow development of new algorithms.

When confined to smaller levels, everything is math behind deep learning. So it is essential to understand basic linear algebra before getting started with deep learning and programming it. With the basic knowledge of linear algebra, it easy to proceed with the stationary iterative methods to solve system of linear equations.

Linear algebra's application on neural networks, like how matrices, vectorization, tensors comes into the picture when we are dealing with deep learning. This actually shows that linear algebra is the core to develop algorithms for machine learning.

When a linear system Ax = b is not solvable we can take an indirect method to solve the system, the indirect method would give an approximate solution to the system with a small error. Stationary iterative methods are indirect methods to solve a linear system of equations, it gives a sequence of values before it arrives at an approximate solution to the linear system, if the sequence is divergent the method fails to give a solution, otherwise if the sequence is convergent in nature we get an approximate solution.

We saw that introducing a momentum factor to stationary iterative methods accelerates the convergence and if some stationary iterations are divergent adding a momentum term can make the sequence convergent.

With this knowledge of solving system of linear equations using stationary iterative methods along with the momentum factor to accelerate the solving of the system it helps to turn the formulas and steps to solve the system using an object oriented programming language – Python.

While programming in the Python language, when the formulas and theorems take the form of the code, we get to infer a few conclusions,

I. We will come across the term solution error, which states that there will be an error depending on the 'μ' value, the system of equations and the guess matrix which are all input by the user.

II. Each of the two methods (Jacobi and Guass-Seidel methods) we get to see that for the solution vector 'x' there will exist errors in the solution of the solution vectors with respect to Jacobi and Seidel methods respectively.

III. Once the momentum is added to the Jacobi and Guass-Seidel functions we get to see that the functions solve the system faster and with less solution errors. For the Jacobi Method the system of equations takes 108 iterations and for the Guass-Seidel Method the system of equations takes 56 iterations.

IV. The above iterations reduced to 88 and 48 iterations for Jacobi and Guass-Seidel methods respectively (*when the momentum is added*).

V. This give us a tentative result which is that the Guass-Seidel method is faster than the Jacobi method to solve the system of linear equations whether or not we include the momentum to accelerate the solving process.

Bibliography

1) D. E. Rumelhart and J. L. McClelland eds., Parallel Distributed Processing: Explorations in the Microstructure of Cognition, vol. 1, chapter 1, page number 4, Cambridge MA: MIT Press, Cambrige,1986.

2) J. J. H. Miller, On the location of zeros of certain classes of polynomials with applications to numerical analysis, edition 1, chapter 8, page nos. 397 to 406, J. Inst. Math. Appl., USA,1971.

3) N. Qian On the momentum term in gradient descent learning algorithms Neural Networks 12 (1999) 145-151.

4) V. V. Phansalkar and P. S. Sastry Analysis of the back-propagation algorithm with momentum IEEE Transactions on Neural Networks 5(3) (1994) 505-506.

5) A. Bhaya and E. Kaszkurewicz Steepest descent with momentum for quadratic functions is a version of the conjugate gradient method Neural Networks 17 (2004) 65-71.

6) M. Torii and M. T. Hagan Stability of steepest descent with momentum for quadratic functions IEEE Transactions on Neural Networks 13(3) (2002) 752-756.

7) W. Wu N. M. Zhang Z. X. Li L. Li and Y. Liu Convergence of gradient method with momentum for back-propagation neural networks Journal of Computational Mathematics 26(4) (2008) 613-623.

8) N. M. Zhang W. Wu and G. F. Zheng Convergence of gradient method with momentum for two-layer feedforward neural networks IEEE Transactions on Neural Networks 17(2) (2006) 522-525.

9) N. M. Zhang Deterministic convergence of an online gradient method with momentum ICIC 2006 Lecture Notes in Computer Science 4113 (2006) 94-105.

10) N. M. Zhang An online gradient method with momentum for two-layer feedforward neural networks Applied Mathematics and Computation 212 (2009) 488-498.