Contents

List of Figures

List of Tables

Part I

Mathematics

Chapter 1

Linear Algebra

1.1 With R

Reader is assumed to have installed R in the machine and have some Google knowledge on R. Aja captures the essence of Linear Algebra through R language, peculiar syntax of R is explained wherever needed. We knew you are more active and eager in trying out following code and we being lazy skipped the outputs!

1.1.1 Vector

To create a vector in R, there are several approaches. Lets create a vector called **"vec"**:

```
# We can use the concatenate function to make an
   ordered list of numbers, e.g.,
vec <- c( 1.9, -2.8, 5.6, 0, 3.4, -4.2 )
vec
# We can use the colon operator to make a sequence of
   integers, e.g.,
vec <- -5:5
vec
vec <- 5:-5
vec
# We can use the sequence function. We have to give a
    starting value, and ending value, and an increment
```

```
     value, e.g.,
vec <- seq(-3, 3, by=0.1)
vec
# We can use the replicate function to a get a vector
    whose elements are all the same, e.g.,
vec <- rep(1,10)
vec
#We can concatenate two or more vectors to make a
    larger vector, e.g.,
vec <- c( rep(1,3), 2:5, c(6,7,8))
vec
# We can take one of the columns from a data table and
     make it a vector, e.g.,
vec <- Data$ACT
```

Some vector functions available:

```
# Vector misc functions
# If we want to know how many elements are in the
    vector, we use the length function, e.g.,
length(vec)
# If we want to reference an element in the vector, we
     use square brackets, e.g.,
vec[5]
# If we want to reference a consecutive subset of the
    vector, we use the colon operator within the square
     brackets, e.g.,
vec[5:10]
```

1.1.2 Matrix

To create a matrix in R, we may use the matrix function. We need to provide a vector containing the elements of the matrix, and specify either the number of rows or the number of columns of the matrix. This number should divide evenly into the length of the vector, or we will get a warning.

```
# For example, to make a 4 x 5 matrix named mat
   consisting of the integers 1 through 20, we can do
   this:
mat <- array(1:20, dim=c(4,5))
mat

# For example, to make a 2 x 3 matrix named mat
   consisting of the integers 1 through 6, we can do
   this:
mat <- matrix( 1:6, nrow=2 )
#or this
mat <- matrix ( 1:6, ncol=3 )
#we get
mat
#Note that R places the row numbers on the left and
   the column numbers on top.  Also note that R filled
    in the matrix column-by-column.  If we prefer to
   fill in the matrix row-by-row, we must activate the
    byrow setting, e.g.,
mat <- matrix( 1:6, ncol=3, byrow=TRUE )
# In place of the vector 1:6  you would place any
   vector containing the desired elements of the
   matrix.

#Extracting a Row of a Matrix
mat[1,]
#Extracting a Column of a Matrix
mat[,1]

#Extracting several Rows and/or columns
mat <- matrix( 1:25, ncol=5, byrow=TRUE )
mat
mat[1:3,2:4]

#You can combine several matrices with the same number
    of
#columns by joining them as rows/cols, using the rbind
   ()/cbind()
```

```
#command
matA <- matrix(c (1 ,3 ,3 ,9 ,6 ,5) ,2 ,3)
matB <- matrix(c (9 ,8 ,8 ,2 ,9 ,0) ,2 ,3)
matA
matB
rbind(matA,matB)
rbind(matB,matA)
cbind(matA,matB)
cbind(matB,matA)

# Matrix Opeartions
# Transpose
matA
t( matA )

# Addition
matA <- matrix ( rep(2,6) , nrow=2, ncol=3, byrow=FALSE
    )
matB <- matrix ( rep(1,6) , nrow=2, ncol=3, byrow=FALSE
    )
matA
matB
matA + matB
matA - matB
```

1.1.2.1 Inverse of a Matrix

For a square matrix A, the inverse is written A^-1. When A is multiplied by A$^-$1 the result is the identity matrix I. Non-square matrices do not have inverses.

Note: Not all square matrices have inverses. A square matrix which has an inverse is called **invertible** or **nonsingular**, and a square matrix without an inverse is called **noninvertible** or **singular**.

$AA^{-1} = A^{-1}A = I$

Example :

For matrix $A_{2,2} = \begin{bmatrix} 4 & 3 \\ 3 & 2 \end{bmatrix}$, its inverse is $A^{-1} = \begin{bmatrix} -2 & 3 \\ 3 & -4 \end{bmatrix}$ since

$$AA^{-1} = \begin{bmatrix} 4 & 3 \\ 3 & 2 \end{bmatrix} \begin{bmatrix} -2 & 3 \\ 3 & -4 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \text{ and}$$
$$A^{-1}A = \begin{bmatrix} -2 & 3 \\ 3 & -4 \end{bmatrix} \begin{bmatrix} 4 & 3 \\ 3 & 2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Here are three ways to find the inverse of a matrix:

1.1.2.1.1 Shortcut for 2x2 matrices

For $A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$, the inverse can be found using this formula:
$A^{-1} = \frac{1}{detA} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix} = \frac{1}{ad-bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$

1.1.2.1.2 Augmented matrix method

Before we find the inverse, we see how to do **Gauss-Jordan elimination**. Use Gauss-Jordan elimination to transform [A — I] into [I — A-1].

Example : The following steps result in $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}^{-1}$

Finding the A^{-1}

$$\left[\begin{array}{cc|cc} 1 & 2 & 1 & 0 \\ 3 & 4 & 0 & 1 \end{array}\right] = \left[\begin{array}{cc|cc} 1 & 2 & 1 & 0 \\ 0 & -2 & -3 & 1 \end{array}\right] = \left[\begin{array}{cc|cc} 1 & 2 & 1 & 0 \\ 0 & 1 & 3/2 & -1/2 \end{array}\right]$$
$$= \left[\begin{array}{cc|cc} 1 & 0 & -2 & 1 \\ 0 & 1 & 3/2 & -1/2 \end{array}\right]$$

So we see that $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}^{-1} = \left[\begin{array}{cc|cc} 1 & 0 & -2 & 1 \\ 0 & 1 & 3/2 & -1/2 \end{array}\right]$

1.1.2.1.2.1 Gauss-Jordan elimination The system of equations
$x + y + z = 3$
$2x + 3y + 7z = 0$
$x + 3y - 2z = 17$
$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 3 \\ 2 & 3 & 7 & 0 \\ 1 & 3 & -2 & 17 \end{array}\right]$
Row operations can be used to express the matrix in reduced row-echelon form.
Which has following properties

- Each row contains only zerosuntil the first nonzero element, which must be 1.
- As the rows are followed from top to bottom, the first nonzero number occurs further to the right than in the previous row.
- The entries above and below the first 1 in each row must all be 0.
- Matrix in the reduced row-echelon form : $\begin{bmatrix} 1 & 0 & -2 & 0 & 6 \\ 0 & 1 & 7 & 0 & 1 \\ 0 & 0 & 0 & 1 & 5 \end{bmatrix}$

Solving the system of equations

$$\begin{aligned}\left[\begin{array}{ccc|c} 1 & 1 & 1 & 3 \\ 2 & 3 & 7 & 0 \\ 1 & 3 & -2 & 17 \end{array}\right] &= \left[\begin{array}{ccc|c} 1 & 1 & 1 & 3 \\ 0 & 1 & 5 & -6 \\ 0 & 2 & -3 & 14 \end{array}\right] \\ &= \left[\begin{array}{ccc|c} 1 & 0 & -4 & 9 \\ 0 & 1 & 5 & -6 \\ 0 & 0 & -13 & 26 \end{array}\right] \\ &= \left[\begin{array}{ccc|c} 1 & 0 & -4 & 9 \\ 0 & 1 & 5 & -6 \\ 0 & 0 & 1 & -2 \end{array}\right] \\ &= \left[\begin{array}{ccc|c} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 4 \\ 0 & 0 & 1 & -2 \end{array}\right]\end{aligned}$$

The augmented matrix now says that x=1, y=4, and z=-2.

1.1.2.1.3 Adjoint method

$A^{-1} = \frac{1}{det A}(adjoint of A)$
or
$A^{-1} = \frac{1}{det A}(cofactor of A)^T$

Example:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 1 & 0 & 6 \end{bmatrix}$$

Cofactor matrix is

$$\begin{bmatrix} A_{1,1} & -A_{1,2} & A_{1,3} \\ -A_{2,1} & A_{2,2} & -A_{2,3} \\ A_{3,1} & -A_{3,2} & A_{3,3} \end{bmatrix}$$

where $A_{1,1} = \begin{bmatrix} 4 & 5 \\ 0 & 6 \end{bmatrix} = 24$ $A_{1,2} = \begin{bmatrix} 0 & 5 \\ 1 & 6 \end{bmatrix} = 5....$ $A_{3,3} = \begin{bmatrix} 1 & 2 \\ 0 & 4 \end{bmatrix} = 4$

The cofactor matrix for A is $\begin{bmatrix} 24 & 5 & -4 \\ -12 & 3 & 2 \\ -2 & -5 & 4 \end{bmatrix}$, so the adjoint is $\begin{bmatrix} 24 & -12 & -2 \\ 5 & 3 & -5 \\ -4 & 2 & 4 \end{bmatrix}$.

Since det A = 22, we get $A^{-1} = \frac{1}{22}\begin{bmatrix} 24 & -12 & -2 \\ 5 & 3 & -5 \\ -4 & 2 & 4 \end{bmatrix} = \begin{bmatrix} 12/11 & -6/11 & -1/11 \\ 5/22 & 3/22 & -5/22 \\ -2/11 & 1/11 & 2/11 \end{bmatrix}$

```
# Multiplication
#Scalar
3 * matA
# To multiply two matrices with compatible dimensions
   (i.e., the number of columns of the first matrix
   equals the number of rows of the second matrix), we
    use the matrix multiplication operator %*%.  For
   example,
matA <- matrix( rep(1,6), nrow=2, ncol=3 )
matB <- matrix( rep(1,6), nrow=3, ncol=2 )
matA %*% matB
#If we just use the multiplication operator *,  R will
    multiply the corresponding elements of the two
   matrices, provided they have the same dimensions.
   But this is not the way to multiply matrices.

#Likewise, to multiply two vectors to get their scalar
    (inner) product, we use the same operator, e.g.,
#Technically, we should use the transpose of a.  But R
    will transpose a for us rather than giving us an
   error message.
a %*% b

#To create the identity matrix for a desired dimension
   , we use the diagonal function, e.g.,
I <- diag( 5 ) #5 x 5
I

#To find the determinant of a square matrix N, use the
    determinant function, e.g.,
A <- matrix( c(1,2,3,0,4,5,1,0,6), nrow=3, byrow=TRUE)
det( A )

# To obtain the inverse N-1 of an invertible square
   matrix N, we use the solve function, e.g.,
# If the matrix is singular (not invertible), or
   almost singular, we get an error message.
solve( A )

# Lets test matrix inversion
#AA^-1 = I
A <- matrix (c(1,3,3,9,6,5,9,1,8), ncol=3)
solve(A)
A %*% solve(A)
#See the difference with following code
zapsmall(A %*% solve(A))
```

1.1.2.2 Solving Systems of Linear Equation

Let the system of equations be
$3x_1 + 2x_2 - 1x_3 = 1$
$2x_1 - 2x_2 + 4x_3 = -2$
$-x_1 + 0.5x_2 - x_3 = 0$

$$\begin{bmatrix} 3 & 2 & -1 \\ 2 & -2 & +4 \\ -1 & 0.5 & -1 \end{bmatrix} * \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ -2 \\ 0 \end{bmatrix}$$

$A * x = b$
$A^{-1} * b = x$

```
A <- array(c(3,2,-1,2,-2,0.5,-1,4,-1), dim=c(3,3))
b <- c(1,-2,0)
solve(A,b)

A %*% solve(A,b)
#A * x = b
zapsmall(A %*% solve(A,b))
# A^-1 * b = x
solve(A) %*% b
```

Part II

Machine Learning

Chapter 1

Machine Learning Algorithms

1.1 The Big List

- Decision tree Induction
- Nearest Neighbour

Chapter 2

Machine Learning Terms

2.1 Regression

2.2 Over Fitting

2.3 Generalization

The process of generalization can be regarded as a search through an enormous, but finite, search space.

2.4 Greedy Search

Chapter 3

Bayesian Statistics

Bayesian statistics is a collection of tools that is used in a special form of statistical inference which applies in the analysis of experimental data in many practical situations in science and engineering. Bayes rule is one of the most important rules in probability theory.

Chapter 4

Learning Rules

4.1 Delta Rule

In machine learning, the delta rule is a gradient descent learning rule for updating the weights of the inputs to artificial neurons in single-layer neural network. It is a special case of the more general backpropagation algorithm. For a neuron j, with activation function $g(x)$, the delta rule for $j's$ ith weight w_{ji}, is given by

$$\Delta w_{ji} = \eta(t_i, y_i)g'(h_j)x_i$$

where η is a small constant called learning rate
$g(x)$ is the neuron's activation function
t_j is the target output
h_j is the weighted sum of the neuron's inputs
y_j is the actual output
x_i is the i th input.

It holds that
$h_j = \sum x_i w_{ji}$
$y_i = g(h_j)$

4.1.1 Derivation

The delta rule is derived by attempting to minimize the error in the output of the neural network through gradient descent. The error for a neural network with j, outputs can be measured as

$$E = \frac{1}{2}\sum_j (t_j - y_j)^2$$

In this case, we wish to move through "weight space" of the neuron (the space of all possible values of all of the neuron's weights) in proportion to the gradient of the error function with respect to each weight. In order to do that, we calculate the partial derivative of the error with respect to each weight. For the i ,th weight, this derivative can be written as

$$\frac{\partial E}{\partial w_{ji}}$$

Because we are only concerning ourselves with the j th neuron, we can substitute the error formula above while omitting the summation:

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial \left(\frac{1}{2}\left(t_j - y_j\right)^2\right)}{\partial w_{ji}}$$

Next we use the chain rule to split this into two derivatives

$$= \frac{\partial \left(\frac{1}{2}\left(t_j - y_j\right)^2\right)}{\partial y_j}\frac{\partial y_j}{\partial w_{ji}}$$

To find the left derivative, we simply apply the general power rule

$$\begin{aligned}
&= -\left(t_j - y_j\right)\frac{\partial y_j}{\partial w_{ji}} \\
&= -\left(t_j - y_j\right)\frac{\partial (g(h_j))}{\partial h_j}\frac{\partial h_j}{\partial w_{ji}} \\
&= -\left(t_j - y_j\right) g'(h_j)\frac{\partial h_j}{\partial w_{ji}} \\
&= -\left(t_j - y_j\right) g'(h_j)\frac{\partial \left(\sum_k x_i w_{ji}\right)}{\partial w_{ji}}
\end{aligned}$$

$$\frac{\partial x_i w_{ji}}{\partial w_{ji}} = x_i$$

$$\frac{\partial E}{\partial w_{ji}} = -\left(t_j - y_j\right) g'(h_j) x_i$$

As noted above, gradient descent tells us that our change for each weight should be proportional to the gradient. Choosing a proportionality constant η and eliminating the minus sign to enable us to move the weight in the negative direction of the gradient to minimize error, we arrive at our target equation:

$$\Delta w_{ji} = \alpha(t_j - y_j)g'(h_j)x_i \,.$$

Part III

Parallel Computing

Part IV

Big Data

Chapter 1

What the crap is this?

We will come back to this later. Bcoz we don't know what to write?

1.1 YYY??

1.1.1 Compare: one of the first mistaken attitudes!

A typical transaction system lives inside the range of some gigabytes data, in random access. It runs inside an application server, acceses and takes control over a relational database allocated in a different server, transporting data in and out of it. Interacts with online clients, keeping a reduced size of data in transport, with a shared and limited bandwith, the operations are most of all continuous reading over small sets of data, combined with some maintenance & CRUD operations. Bigger data processing is done in batch, but the architecture is the same.

1.1.2 But a different scenario ?

What happens if we need to process 1 petabyte/week ?, let's say a much lesser size, 1 terabyte/day, less, 100 gigabytes/day Under the traditional schema, it would be like moving an elephant over a string thousands of times, the time-windows would not allow to work with realtime information, but always trying to keep up and losing the race.

1.1.3 Want problems ?

The network would saturate, batch processes would take days to finish information of hours, the harddisk latency access it will transform into an incremental overhead, having a painful impact on overall costs. The traditional approach would mantain the architecture, but changing hardware, sophisticated requirements, each time more expensive, bigger, but over a limited growth architecture. How many times will you change your system's architecture to non-linear scalar solutions to keep up with the race of growing data, ?

1.1.4 The solution is in the focus !

Problem 1: Size of data
Systems handling public data, can have huge processing flows with hundreds of terabytes, public websites have had increased their data up to petabytes !
Strategy 1: Distribute and paralellize the processing
If chewing 200 Tb of data takes 11 days in 1 computer, let divide the whole process into as many machines needed to finish the task in minutes or hours ! If we'll have lot of computers, so be they cheap and easily mantained !, into a simple add & replace node architecture.

Problem 2: Continuous failure and cluster growth
In a big cluster, lot of machines fail everyday, besides the cluster size cannot be fixed and sometimes cannot be planned, it should grow easily at the speed of data.
Strategy 2: High fault tolerance and high scalability
Any design must have an excelent fault-tolerance mechanism, a very flexible maintenance, and its service availability should keep up naturally to its cluster size. So we need a distributed storage, atomic, transactional, with perfect linear scalability.

Problem 3: Exponential overhead in data transport
From data source to its processing location, bad harddisk latency and saturated networks will end up exploting
Strategy 3: Move application logic to data storage It must allow to handle data and processes separately and in a transparent way, applications should take care of their own business application code, and the platform

should automatically deploy it into the datastorage, execute and handle their results,

So... Distribute and Paralellize processing High fault-tolerance mechanism, High scalability Take application processes to data storage

One Solution: Hadoop Distributed Filesystem + Hadoop MapReduce

Anyway, Hadoop is not bullet-proof, actually, it's a very specific solution for some big-data scenarios, with needs of realtime processing, and technology choices for random access to data. (See HBase, Hive)

Hadoop is not designed to replace RDBMS, but instead, has been proven to handle -in a much performant way- huge amounts of data, whereas traditional enterprise database clusters, wouldn't work not even close at the same overall costs !

Chapter 2

Architecture

2.1 FileSystem

- Scales linearly over a set of low-budget comodity machines, doubled the amount of machines = reduced the processing time to half
- Tolerates faults at different levels, from network, switches, disks, nodes, readdressing data traffic to other nodes, accomplishing a replica factor
- Flexible scalability, for maintenance tasks, you only have to dis/connect computers into the rack
- Lets you allocate any kind of files and formats, although it has better performance on files bigger than 128mb
- The results of MapReduce jobs are stored in the filesystem
- There's a unique namespace, with an automatic replication schema administered by the master, it's not possible to impact on a certain node in the cluster, whether to allocate files or execute jobs
- The master itself balances the workload over execution plans and status reports from the slave nodes
- Has a mechanism of continuous replication per file, rack-aware, to extend data reliability and data availability warranties
- Has an automatic file checksum with inmediate correction
- Works in a master-slave schema where slaves share nothing between them, they only respond to master requests
- The master coordinates all types of transactions, read/write, replication, restore, manages the tx log and the filesystem namespace
- The slaves only take charge on low level operations over data, read, write,

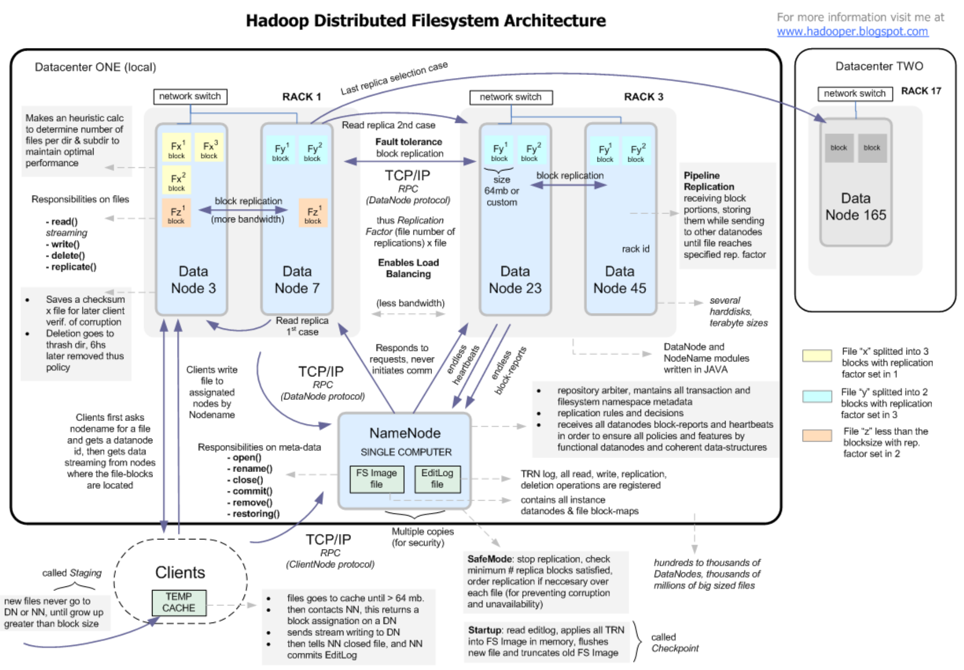

Hadoop Distributed Filesystem Architecture
For more information visit me at www.hadooper.blogspot.com
Datacenter ONE (local)
network switch
RACK 1
Last replica selection case
network switch
RACK 3
Datacenter TWO
network switch
RACK 17
Makes an heuristic calc to determine number of files per dir & subdir to maintain optimal performance
Responsibilities on files
- read() streaming
- write()
- delete()
- replicate()
Saves a checksum x file for later client verif. of corruption
Deletion goes to thrash dir, 6hs later removed thus policy
Data Node 3
Data Node 7
block replication (more bandwidth)
Read replica 2nd case
Fault tolerance block replication
TCP/IP RPC (DataNode protocol)
thus Replication Factor (file number of replications) x file
Enables Load Balancing
(less bandwidth)
Read replica 1st case
size 64mb or custom
block replication
rack id
Data Node 23
Data Node 45
Pipeline Replication receiving block portions, storing them while sending to other datanodes until file reaches specified rep. factor
several harddisks, terabyte sizes
block
Data Node 165
DataNode and NodeName modules written in JAVA
Responds to requests, never initiates comm
endless heartbeats
endless block-reports
Clients write file to assignated nodes by Nodename
TCP/IP RPC (DataNode protocol)
Clients first asks nodename for a file and gets a datanode id, then gets data streaming from nodes where the file-blocks are located
NameNode SINGLE COMPUTER
FS Image file
EditLog file
Responsibilities on meta-data
- open()
- rename()
- close()
- commit()
- remove()
- restoring()
repository arbiter, mantains all transaction and filesystem namespace metadata
replication rules and decisions
receives all datanodes block-reports and heartbeats in order to ensure all policies and features by functional datanodes and coherent data-structures
TRN log, all read, write, replication, deletion operations are registered
contains all instance datanodes & file block-maps
File "x" splitted into 3 blocks with replication factor set in 1
File "y" splitted into 2 blocks with replication factor set in 3
File "z" less than the blocksize with rep. factor set in 2
TCP/IP RPC (ClientNode protocol)
Multiple copies (for security)
called Staging
new files never go to DN or NN, until grow up greater than block size
Clients
TEMP CACHE
files goes to cache until > 64 mb.
then contacts NN, this returns a block assignation on a DN
sends stream writing to DN
then tells NN closed file, and NN commits EditLog
SafeMode: stop replication, check minimum # replica blocks satisfied, order replication if neccesary over each file (for preventing corruption and unavailability)
Startup: read editlog, applies all TRN into FS Image in memory, flushes new file and truncates old FS Image
called Checkpoint
hundreds to thousands of DataNodes, thousands of millions of big sized files

Figure 2.1: The picture shows a more practical brief over HDFS responsibilities & execution flows

deletion, transport to-from client

2.2 Processing Model

Main characteristics of Map Reduce

- It's a processing model about dividing and distributing information, in two chained phases: map first, then reduce
- Both phases have as input and output, a key-value pair list,
- The schema allows to define method parameters and own logic for both phases, as well as their own partitioning system and intermediate storage between phases.
- The transactions are handled by a JobTracker daemon, that runs the initial data partitioning and the intermediate data combination, by posting tasks of type Map and type Reduce over the TaskTracker daemons (1-n x computer) of the nodes involved in the cluster, according the data being

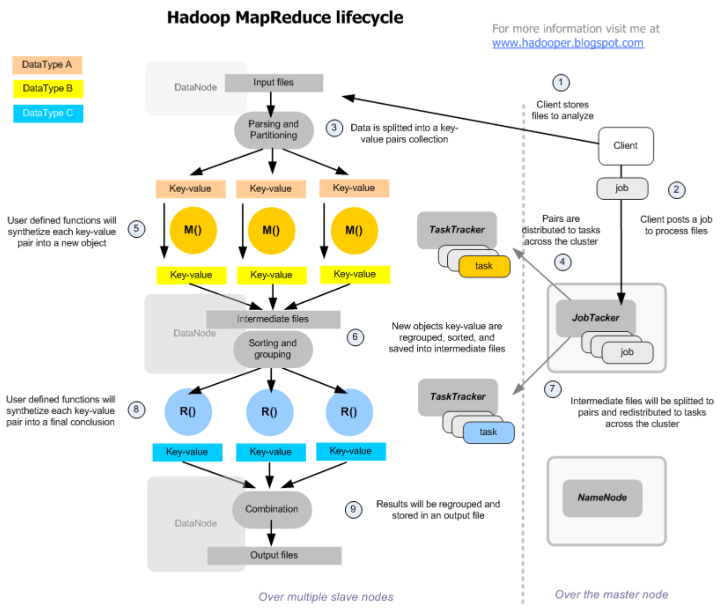
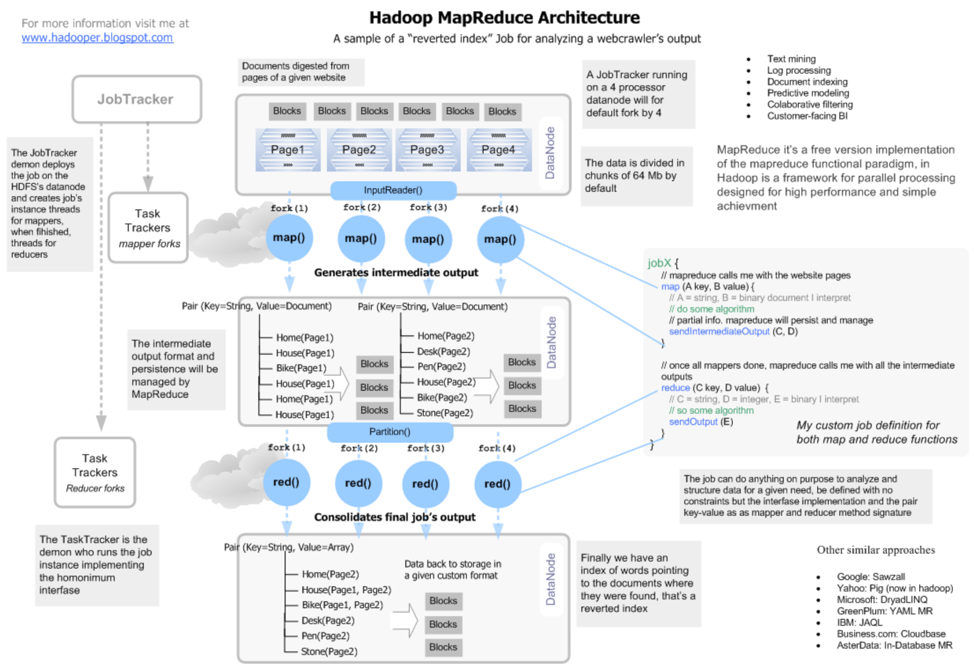

Hadoop MapReduce lifecycle
For more information visit me at
www.hadooper.blogspot.com
DataType A
DataType B
DataType C
DataNode
Input files
1
Client stores files to analyze
Client
job
2
Client posts a job to process files
Parsing and Partitioning
3
Data is splitted into a key-value pairs collection
Key-value
Key-value
Key-value
User defined functions will synthetize each key-value pair into a new object
5
M()
M()
M()
TaskTracker
task
Pairs are distributed to tasks across the cluster
4
Key-value
Key-value
Key-value
JobTacker
job
DataNode
Intermediate files
Sorting and grouping
6
New objects key-value are regrouped, sorted, and saved into intermediate files
User defined functions will synthetize each key-value pair into a final conclusion
8
R()
R()
R()
TaskTracker
task
7
Intermediate files will be splitted to pairs and redistributed to tasks across the cluster
Key-value
Key-value
Key-value
NameNode
DataNode
Combination
9
Results will be regrouped and stored in an output file
Output files
Over multiple slave nodes
Over the master node

Figure 2.2: The picture shows how these methods will interact in phases.

processed

- The Reduce phase only starts when finished the Map, cause after the Map the resulting keys are combined, to distribute a sorted list of key-value pairs between the Reducers, that can be matched at the end of them.
- The process is transactional, those map or reduce tasks not executed, (for data availability issues) will be reattempted a number of times, and then redistributed to other nodes.
- First the files are partitioned in parts that will be distributed to process across the cluster nodes
- Each part is parsed in pairs of Key(sorteable object) - Value(object), that will be the input parameters for the tasks implementing the Map function
- These user defined tasks (map), will read the value object, do something with it, and then build a new key-value list that will be stored by the framework, in intermediate files.
- Once all the map tasks are finished, it means that the whole data to process was completely read, and reordered into this mapreduce model of key-value paris.
- These intermediate key-value results are combined, resulting a new paris of key-value that will be the input for the next reduce tasks


For more information visit me at
www.hadooper.blogspot.com
Hadoop MapReduce Architecture
A sample of a "reverted index" Job for analyzing a webcrawler's output
JobTracker
The JobTracker demon deploys the job on the HDFS's datanode and creates job's instance threads for mappers, when fihished, threads for reducers
Task Trackers
mapper forks
Documents digested from pages of a given website
Blocks
Page1
Page2
Page3
Page4
DataNode
InputReader()
fork (1)
fork (2)
fork (3)
fork (4)
map()
Generates intermediate output
A JobTracker running on a 4 processor datanode will for default fork by 4
The data is divided in chunks of 64 Mb by default
Text mining
Log processing
Document indexing
Predictive modeling
Colaborative filtering
Customer-facing BI
MapReduce it's a free version implementation of the mapreduce functional paradigm, in Hadoop is a framework for parallel processing designed for high performance and simple achievment
Pair (Key=String, Value=Document)
The intermediate output format and persistence will be managed by MapReduce
Home(Page1)
House(Page1)
Bike(Page1)
Home(Page2)
Desk(Page2)
Pen(Page2)
House(Page2)
Bike(Page2)
Stone(Page2)
Partition()
Task Trackers
Reducer forks
red()
Consolidates final job's output
The TaskTracker is the demon who runs the job instance implementing the homonimum interfase
Pair (Key=String, Value=Array)
House(Page1, Page2)
Bike(Page1, Page2)
Data back to storage in a given custom format
Finally we have an index of words pointing to the documents where they were found, that's a reverted index
My custom job definition for both map and reduce functions
The job can do anything on purpose to analyze and structure data for a given need, be defined with no constraints but the interfase implementation and the pair key-value as as mapper and reducer method signature
Other similar approaches
Google: Sawzall
Yahoo: Pig (now in hadoop)
Microsoft: DryadLINQ
GreenPlum: YAML MR
IBM: JAQL
Business.com: Cloudbase
AsterData: In-Database MR

```
jobX {
  // mapreduce calls me with the website pages
  map (A key, B value) {
    // A = string, B = binary document I interpret
    // do some algorithm
    // partial info. mapreduce will persist and manage
    sendIntermediateOutput (C, D)
  }

  // once all mappers done, mapreduce calls me with all the intermediate
  outputs
  reduce (C key, D value)  {
    // C = string, D = integer, E = binary I interpret
    // so some algorithm
    sendOutput (E)
  }
}
```

SELECT

Figure 2.3: Let's see a sample job with a reverted-index function, for analyzing the webcrawler's output files (just for instance)

- These user defined tasks (reduce), will read the value object, do something with it, and then produce the 3rd and last list of key-value pairs, that the framework will combine, and regroup into a final result.

MapReduce something, is about iterate a huge record collection, extract something good, mix and regroup intermediate results, that's all, it may look more complex than what it is.

Part V

Neural Network

Chapter 1

What is Hopfield?

1.1 Introduction

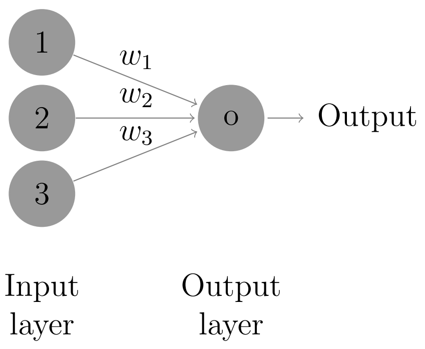


Figure 1.1: An artificial neuron as used in a Hopfield network

Hopfield networks are constructed from artificial neurons Figure 1.1. These artificial neurons have N inputs. With each input i there is a weight w_i associated. They also have an output. The state of the output is maintained, until the neuron is updated. Updating the neuron entails the following operations:

- The value of each input, x_i is determined and the weighted sum of all inputs, $\sum_{i=1}^{n} w_i x_i$ is calculated.
- The output state of the neuron is set to $+1$ if the weighted input sum is larger or equal to 0. It is set to -1 if the weighted input sum is smaller than 0.

- A neuron retains its output state until it is updated again.

  Written as formula:

$$o = \begin{cases} 1 : & \sum_i w_i x_i >= 0 \\ -1 : & \sum_i w_i x_i < 0 \end{cases}$$

A Hopfield network is a network of N such artificial neurons, which are fully connected. The connection weight from neuron j to neuron i is given by a number w_{ij}. The collection of all such numbers is represented by the weight matrix W, whose components are w_{ij}.

Now given the weight matrix and the updating rule for neurons the dynamics of the network is defined if we tell in which order we update the neurons. There are two ways of updating them:

- **Asynchronous**: one picks one neuron, calculates the weighted input sum and updates immediately. This can be done in a fixed order, or neurons can be picked at random, which is called asynchronous random updating.
- **Synchronous**: the weighted input sums of all neurons are calculated without updating the neurons. Then all neurons are set to their new value, according to the value of their weighted input sum. The lecture slides contain an explicit example of synchronous updating.

1.2 Use of the Hopfield network

The way in which the Hopfield network is used is as follows. A pattern is entered in the network by setting all nodes to a specific value, or by setting only part of the nodes. The network is then subject to a number of iterations using asynchronous or synchronous updating. This is stopped after a while. The network neurons are then read out to see which pattern is in the network. The idea behind the Hopfield network is that patterns are stored in the weight matrix. The input must contain part of these patterns. The dynamics of the network then retrieve the patterns stored in the weight matrix. This is called **Content Addressable Memory (CAM)**. The network can also be used for auto-association. The patterns that are stored in the network are divided in two parts: **cue** and **association**. By entering the cue into the network, the entire pattern, which is stored in the weight matrix, is retrieved. In this way the network restores the association that belongs to a given cue. The stage is now almost set for the Hopfield network,

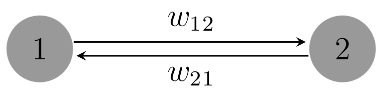


Figure 1.3: There are just two options for the weight matrix: $w_{i,j}$ = 1 or $w_{i,j}$ = -1. If the weight is 1, there are two stable states under synchronous updating {+1, +1}, or {-1, -1}. For a weight of -1, the stable states will be {-1, +1} or {+1, -1} depending on initial conditions. Under synchrononous updating the states are oscillatory.

we must only decide how we determine the weight matrix. We will do that in the next section, but in general we always impose two conditions on the weight matrix:
- Symmetry : $w_{ij} = w_{ij}$
- No self connections: $w_{ii} = 0$

It turns out that we can guarantee that the network converges under asynchronous updating when we use these conditions

1.3 Training the network

1.3.1 A Simple Example

Consider the two nodes in Fig 1.3 Depending on which values they contain initially they will reach end states +1, +1 or -1,-1 under asynchronous updating.
This simple example illustrates two important aspects of the Hopfield network: the steady final state is determined by the value of the weight and the network is sign-blind; it depends on the initial conditions which final state will be reached, {+1,+1} or {-1, -1} Figure 1.3.

1.4 Setting the weight matrix

1.4.1 A single pattern

Consider two neurons. If the weight between them is positive, then they will tend to drive each other in the same direction: this is clear from the two-neuron network in the example. It is also true in larger networks: suppose we have neuron i connected to neuron j with a weight of +1, then the contribution of neuron i to the weighted input sum of neuron j is **positive** if $x_i = 1$ and negative if $x = -1$. In other words neuron, i tries to drive neuron j to the same value as it has currently. If the connection weights between them is negative, however, neuron i will try to drive neuron j to the opposite value! This inspires the choice of the Hebb rule: given a network of N nodes and faced with a pattern $x = (x_1, ..., x_N)$ that we want to store in the network, we chose the values of the weight matrix as follows:

$$w_{i,j} = x_i x_j$$

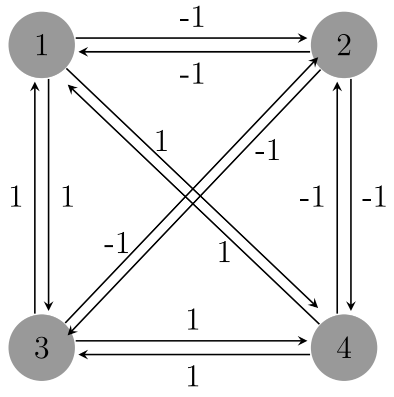


Figure 1.4: Single Layer Hopfield network

To see how this works out in a network of four nodes, consider Figure 1.3. Note that weights are optimal for this example in the sense that if the pattern (1, -1, 1, 1) is present in the network, each input node receives the maximal or minimal (in the case of neuron 2) input some possible! We can actually prove this: suppose the pattern $\vec{x}$, which has been used to train the network, is present in the network and we update neuron i, what will happen? The weighted input sum of a node is often denoted by $h_i = \sum_j w_{i,j} x_j$, which is called the local field. If w_{ij} was chosen to store this very same pattern, we can calculate h_i:

$$h_i = \sum_j w_{i,j} x_i x_j = \sum_{j=1}^{3} = x_i x_j x_j = x_i + x_i + x_i = 3x_i$$

This is true for all nodes i. So all other three nodes (self connections are forbidden!) in the network give a summed contribution of $3x_i$. This means that x_i will never change value, the pattern is stable!

$$h_i = \sum_j w_{i,j} v_j = -\sum_j w_{i,j} x_j = -\sum_j x_i x_j x_j = -\sum_{j=1}^{3} x_i = 3x_i = 3v_i$$

Again, this pattern is maximally stable: the Hopfield network is sign-blind.

1.4.2 Weight Determination

How to determine weight matrix for multiple patters? You can go about this in the following way.

1.4.2.1 Binary to Bipolar Mapping

By replacing each 0 in a binary string with a -1, you get the corresponding bipolar string.

$$f(x) = 2x - 1$$

For inverse mapping, which turns a bipolar string into a binary string, you use the following function:

$$f(x) = (x + 1)/2$$

1.4.2.2 Patterns Contribution to Weight

We work with the bipolar versions of the input patterns. We take each pattern to be recalled, one at a time, and determine its contribution to the weight matrix of the network. The contribution of each pattern is itself a matrix. The size of such a matrix is the same as the weight matrix of the network. Then add these contributions, in the way matrices are added, and you end up with the weight matrix for the network, which is also referred to as the correlation matrix. Let us find the contribution of the pattern $A = (1, 0, 1, 0)$: First, we notice that the binary to bipolar mapping of $A = (1, 0, 1, 0)$ gives the vector $(1, 1, 1, 1)$. Then we take the transpose, and multiply, the way matrices are multiplied, and we see the following:

$$\begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix} * \begin{bmatrix} 1 & -1 & 1 & -1 \end{bmatrix} = \begin{bmatrix} 1 & -1 & 1 & -1 \\ -1 & 1 & -1 & 1 \\ 1 & -1 & 1 & -1 \\ -1 & 1 & -1 & 1 \end{bmatrix}$$

Now subtract 1 from each element in the main diagonal (that runs from top left to bottom right). This operation gives the same result as subtracting the identity matrix from the given matrix, obtaining 0s in the main diagonal. The resulting matrix, which is given next, is the contribution of the pattern $(1, 0, 1, 0)$ to the weight matrix.

$$W_1 = \begin{bmatrix} 0 & -1 & 1 & -1 \\ -1 & 0 & -1 & 1 \\ 1 & -1 & 0 & -1 \\ -1 & 1 & -1 & 0 \end{bmatrix}$$

Similarly, we can calculate the contribution from the pattern
$B = (0, 1, 0, 1)$ its bipolar version $B = (-1, 1, -1, 1)$ by verifying that pattern B's contribution is the same matrix as pattern A's contribution.

$$\begin{bmatrix} -1 \\ 1 \\ -1 \\ 1 \end{bmatrix} * \begin{bmatrix} -1 & 1 & -1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & -1 & 1 & -1 \\ -1 & 1 & -1 & 1 \\ 1 & -1 & 1 & -1 \\ -1 & 1 & -1 & 1 \end{bmatrix}$$

$$W_2 = \begin{bmatrix} 0 & -1 & 1 & -1 \\ -1 & 0 & -1 & 1 \\ 1 & -1 & 0 & -1 \\ -1 & 1 & -1 & 0 \end{bmatrix}$$

$$W_1 + W_2 = \begin{bmatrix} 0 & -2 & 2 & -2 \\ -2 & 0 & -2 & 2 \\ 1 & -2 & 0 & -2 \\ -2 & 1 & -2 & 0 \end{bmatrix}$$

We can now optionally apply an arbitrary scalar multiplier to all the entries of the matrix if you wish. This is shown in the 2.0.1.

1.4.2.3 Autoassociative Network

The Hopfield network just shown has the feature that the network associates an input pattern with itself in recall. This makes the network an **autoassociative** network. The patterns used for determining the proper weight matrix are also the ones that are **autoassociatively** recalled. These patterns are called the exemplars. A pattern other than an exemplar may or may not be recalled by the network. Of course, when you present the pattern 0000, it is stable, even though it is not an exemplar pattern.

1.4.2.4 Orthogonal Bit Patterns

You may be wondering how many patterns the network with four nodes is able to recall. Let us first consider how many different bit patterns are orthogonal to a given bit pattern. This question really refers to bit patterns in which at least one bit is equal to 1. A little reflection tells us that if two bit patterns are to be orthogonal, they cannot both have 1's in the same position, since the dot product would need to be 0. In other words, a bitwise logical AND operation of the two bit patterns has to result in a 0. This suggests the following. If a pattern P has k, less than 4, bit positions with 0 (and so $4 - k$ bit positions with 1), and if pattern Q is to be orthogonal to P, then Q can have 0 or 1 in those k positions, but it must have only 0 in the rest $4 - k$ positions. Since there are two choices for each of the k positions, there are 2^k possible patterns orthogonal to P. This number 2^k of patterns includes the pattern with all zeroes. So there really are $2^k - 1$ non-zero patterns orthogonal to P. Some of these $2^k - 1$ patterns are not orthogonal to each other. As an example, P can be the pattern (0 1 0 0), which has $k = 3$ positions with 0. There are $2^3 - 1 = 7$ nonzero patterns orthogonal to (0 1 0 0). Among these are patterns (1 0 1 0) and (1 0 0 1), which are not orthogonal to each other, since their dot product is 1 and not 0.

1.4.2.5 Network Nodes and Input Patterns

Since our network has four neurons in it, it also has four nodes in the directed graph that represents the network. These are laterally connected because connections are established from node to node. They are lateral because the nodes are all in the same layer. We started with the patterns A = (1, 0, 1, 0) and B = (0, 1, 0, 1) as the exemplars. If we take any other nonzero pattern that is orthogonal to A, it will have a 1 in a position where B also has a 1. So the new pattern will not be orthogonal to B. Therefore, the orthogonal set of patterns that contains A and B can have only those two as its elements. If you remove B from the set, you can get (at most) two others to join A to form an orthogonal set. They are the patterns (0, 1, 0, 0) and (0, 0, 0, 1). If you follow the procedure described earlier to get the correlation matrix, you will get the following weight matrix:

$$W = \begin{bmatrix} 0 & -1 & 3 & -1 \\ -1 & 0 & -1 & -1 \\ 3 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 \end{bmatrix}$$

With this matrix, pattern A is recalled, but the zero pattern (0, 0, 0, 0) is obtained for the two patterns (0, 1, 0, 0) and (0, 0, 0, 1). Once the zero pattern is obtained, its own recall will be stable.

1.5 Energy

We can define an energy for each node:

$$E_i = -\frac{1}{2} h_i x_i$$

Note that the energy is positive if the sign of h_i and x_i is different! An update of node i will result in a sign change in this case because the local field h_i has a different sign. The updating will change the energy form a positive number to a negative number. This corresponds to the intuitive idea that stable states have a low energy. We can define an energy for the entire network:

$$E(\vec{x}) = \sum_i E_i = -\sum_{i,j} \frac{1}{2} h_i x_i = -\frac{1}{2} \sum_{i,j} w_{i,j} x_i x_j$$

Again, it is clear that for the pattern that has been used to train the weight matrix the energy is minimal. In this case:

$$E = -\frac{1}{2} \sum_{i,j} w_{i,j} x_i x_j = -\frac{1}{2} \sum_{i,j} x_i x_j x_j = -\frac{1}{2} \sum_{i,j} 1 = -\frac{1}{2} (N-1)^2$$

1.6 Stability of a single pattern

So far we have looked at the situation where the same patterns was in the network that was used to train the network. Now let us assume that another pattern is in the network. It is pattern $\vec{y}$ which is the same as pattern $\vec{x}$

except for three nodes. The network, as usual, has N nodes. No we are updating a node i of the network. What is the local field? It is given by:

$$h_i = \sum_j w_{i,j} y_j = \sum_j x_i x_j y_j$$

We can split this sum in 3 nodes, which have an opposite value of $\vec{x}$ and $N3$ nodes which have the same value:

$$h_i = \sum_{j=1}^{N-3} x_i x_j x_j + \sum_{j=1}^{3} x_i x_j - x_j = (N-3)x_i - 3x_i = (N-6)x_i$$

For $N > 6$ all the local fields point the direction of x_i, the pattern that was used to define the weight matrix!. So updating will result in no change (if the value of node i is equal to x_i) or an update (if the value of node i is equal to $-x_i$). So, the pattern that is stored into the network is stable: up to half of the nodes can be inverted and the network will still reproduce $\vec{x}$.

Chapter 2

C++ Code

Code : aja/example/ann/hopfield_network.cpp

2.0.1 Example A Simple Hopfield Network

A simple single layer Hopfield network is created with four neurons. We place, in this layer, four neurons, each connected to the rest, as shown in Figure 2.1. Some of the connections have a positive weight, and the rest have a negative weight. The network will be presented with two input patterns, one at a time, and it is supposed to recall them. The inputs would be binary patterns having in each component a 0 or 1. If two patterns of equal length are given and are treated as vectors, their dot product is obtained by first multiplying corresponding components together and then adding these products. Two vectors are said to be orthogonal, if their dot product is 0. The mathematics involved in computations done for neural networks include matrix multiplication, transpose of a matrix, and transpose of a vector. The inputs (which are stable, stored patterns) to be given should be orthogonal to one another.

The two patterns we want the network to recall are $A = (1, 0, 1, 0)$ and $B = (0, 1, 0, 1)$, which you can verify to be orthogonal. Recall that two vectors A and B are orthogonal if their dot product is equal to zero. This is true in this case since

$A_1B_1 + A_2B_2 + A_3B_3 + A_4B_4 = (1 \times 0 + 0 \times 1 + 1 \times 0 + 0 \times 1) = 0$

The following matrix W gives the weights on the connections in the network.

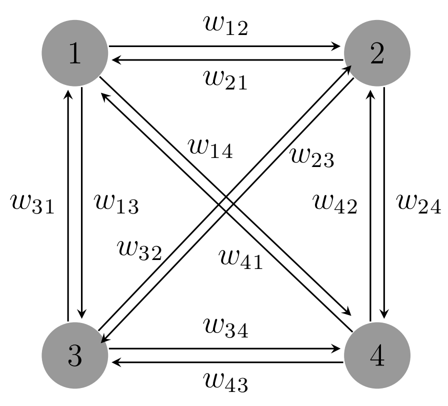


Figure 2.1: Single Layer Hopfield network

$$W = \begin{bmatrix} 0 & -3 & 3 & -3 \\ -3 & 0 & -3 & 3 \\ 3 & -3 & 0 & -3 \\ -3 & 3 & -3 & 0 \end{bmatrix}$$

We need a threshold function also, and we define it as follows. The threshold value θ is 0.

$$f(t) = \begin{cases} 1 & \text{if } t \geq \theta \\ 0 & \text{if } t < \theta \end{cases}$$

We have four neurons in the only layer in this network. We need to compute the activation of each neuron as the weighted sum of its inputs. The activation at the first node is the dot product of the input vector and the first column of the weight matrix (0 -3 3 -3). We get the activation at the other nodes similarly. The output of a neuron is then calculated by evaluating the threshold function at the activation of the neuron. So if we present the input vector A, the dot product works out to 3 and f(3) = 1. Similarly, we get the dot products of the second, third, and fourth nodes to be -6, 3, and -6, respectively. The corresponding outputs therefore are 0, 1, and 0. This means that the output of the network is the vector (1, 0, 1, 0), same as the input pattern. The network has recalled the pattern as presented, or we can say that pattern A is stable, since the output is equal to the input. When B is presented, the dot product obtained at the first node is -6 and the output is 0. The outputs for the rest of the nodes taken together with the output of the first node gives (0, 1, 0, 1), which means that the network has stable recall for B also.

So far we have presented easy cases to the networkvectors that the Hopfield network was specifically designed (through the choice of the weight matrix) to recall. What will the network give as output if we present a pattern different from both A and B? Let C = (0, 1, 0, 0) be presented to the network. The activations would be -3, 0, -3, 3, making the outputs 0, 1, 0, 1, which means that B achieves stable recall. This is quite interesting. Suppose we did intend to input B and we made a slight error and ended up presenting C, instead. The network did what we wanted and recalled B. But why not A? To answer this, let us ask is C closer to A or B? How do we compare? We use the distance formula for two four-dimensional points. If (a, b, c, d) and (e, f, g, h) are two four-dimensional points, the distance between them is:

$$\sqrt{(ae)^2 + (bf)^2 + (cg)^2 + (dh)^2}$$

The distance between A and C is $\sqrt{3}$, whereas the distance between B and C is just 1. So since B is closer in this sense, B was recalled rather than A. You may verify that if we do the same exercise with D = (0, 0, 1, 0), we will see that the network recalls A, which is closer than B to D.

2.1 Asynchronous Update

The Hopfield network is a recurrent network. This means that outputs from the network are fed back as inputs. This is not apparent from Figure2.2.

The Hopfield network always stabilizes to a fixed point. There is a very important detail regarding the Hopfield network to achieve this stability. In the examples thus far, we have not had a problem getting a stable output from the network, so we have not presented this detail of network operation. This detail is the need to update the network asynchronously. This means that changes do not occur simultaneously to outputs that are fed back as inputs, but rather occur for one vector component at a time.

The true operation of the Hopfield network follows the procedure below for input vector **Invec** and output vector **Outvec**:

- Apply an input, Invec, to the network, and initialize Outvec = Invec
- Start with i = 1
- Calculate Value i = DotProduct (Invec i, Column i of Weight matrix)4.

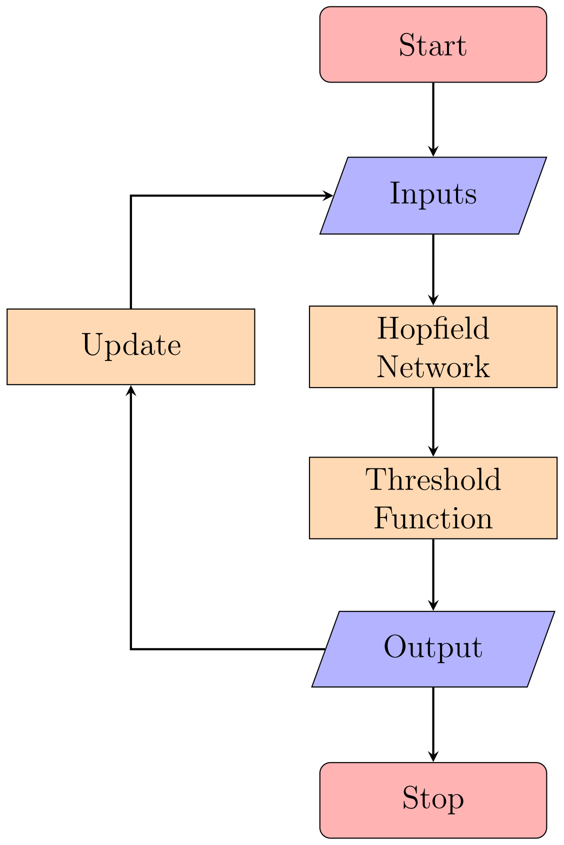


Figure 2.2: Asynchonous Update Flow

- Calculate Outvec i = f(Value i) where f is the threshold function discussed previously
- Update the input to the network with component Outvec i
- Increment i, and repeat steps 3, 4, 5, and 6 until Invec = Outvec(note that when i reaches its maximum value, it is then next reset to 1 for the cycle to continue)

Now lets see how to apply this procedure. Building on the last example, we now input E = (1, 0, 0, 1), which is at an equal distance from A and B. Without applying the asynchronous procedure above, but instead using the shortcut procedure weve been using so far, you would get an output F = (0, 1, 1, 0). This vector, F, as subsequent input would result in E as the output. This is incorrect since the network oscillates between two states. We have updated the entire input vector synchronously. Now lets

Table 2.1: Example of Asynchronous Update for the Hopfield Network

Step	i	Invec	Column of weight vectors	Value	Outvec	notes
0		1001			1001	initialization : set Outvec = Invec =Input pattern
1	1	1001	0 -3 3 -3	-3	0001	column 1 of Outvec changed to 0
2	2	0001	-3 0 -3 3	3	0101	column 2 of Outvec changed to 1
3	3	0101	3 -3 0 -3	-6	0101	column 3 of Outvec stays as 0
4	4	0101	-3 3 -3 0	3	0101	column 4 of Outvec stays as 1
5	1	0101	0 -3 3 -3	-6	0101	column 1 stable as 0
6	2	0101	-3 0 -3 3	3	0101	column 2 stable as 1
7	3	0101	3 -3 0 -3	-6	0101	column 3 stable as 0
8	4	0101	-3 3 -3 0	3	0101	column 4 stable as 1; stable recalled pattern = 0101

apply asynchronous update. For input E, (1,0,0,1) we arrive at the following results detailed for each update step, in Table 2.1

2.2 Classes in C++ Implementation

In our C++ implementation of this network, there are the following classes: a **network class**, and a **neuron class**. In our implementation, we create the network with four neurons, and these four neurons are all connected to one another. A neuron is not self-connected, though. That is, there is no edge in the directed graph representing the network, where the edge is from one node to itself. But for simplicity, we could pretend that such a connection exists carrying a weight of 0, so that the weight matrix has 0s in its principal diagonal.

The functions that determine the neuron activations and the network output are declared public. Therefore they are visible and accessible without restriction. The activations of the neurons are calculated with functions defined in theneuron class. When there are more than one layer in a neural network, the outputs of neurons in one layer become the inputs for neurons in the next layer. In order to facilitate passing the outputs from one layer as inputs to another layer, our C++ implementations compute the neuron outputs in the network class. For this reason the threshold function is made a member of the network class. We do this for the Hopfield network as well. To see if the network has achieved correct recall, you make comparisons between the presented pattern and the network output, component by component.

2.3 A New Weight Matrix to Recall More Patterns

Lets continue to discuss this example. Suppose we are interested in having the patterns $E = (1, 0, 0, 1)$ and $F = (0, 1, 1, 0)$ also recalled correctly, in addition to the patterns $\vec{A}$ and $\vec{B}$. In this case we would need to train the network and come up with a learning algorithm, which we will discuss in more detail later in the book. We come up with the matrix W_1, which follows...

$$W_1 = \begin{bmatrix} 0 & -5 & 4 & 4 \\ -5 & 0 & 4 & 4 \\ 4 & 4 & 0 & -5 \\ 4 & 4 & -5 & 0 \end{bmatrix}$$

Try to use this modification of the weight matrix in the source program, and then compile and run the program to see that the network successfully recalls all four patterns A, B, E, and F.

Chapter 3

Backpropagation

3.1 Feedforward Backpropagation Network

The feedforward backpropagation network is a **very popular model** in neural networks. It **does not have feedback connections**, but errors are backpropagated during training. **Least mean squared error** is used. Many applications can be formulated for using a feedforward backpropagation network, and the methodology has been a model for most multilayer neural networks. Errors in the output determine measures of hidden layer output errors, which are used as a basis for adjustment of connection weights between the input and hidden layers. Adjusting the two sets of weights between the pairs of layers and recalculating the outputs is an iterative process that is carried on until the errors fall below a tolerance level. Learning rate parameters scale the adjustments to weights. A momentum parameter can also be used in scaling the adjustments from a previous iteration and adding to the adjustments in the current iteration.

3.1.1 Mapping

The feedforward backpropagation network maps the input vectors to output vectors. Pairs of input and output vectors are chosen to train the network first. Once training is completed, the weights are set and the network can be used to find outputs for new inputs. The **dimension of the input vector determines** the **number of neurons in the input layer**, and the **number of neurons in the output layer** is determined by the **di-**

mension of the outputs. If there are k neurons in the input layer and m neurons in the output layer, then this network can make a mapping from k-dimensional space to an m-dimensional space. Of course, what that mapping is depends on what pair of patterns or vectors are used as exemplars to train the network, which determine the network weights. Once trained, the network gives you the image of a new input vector under this mapping. Knowing what mapping you want the feedforward backpropagation network to be trained for implies the dimensions of the input space and the output space, so that you can determine the numbers of neurons to have in the input and output layers.

3.1.2 Layers

The architecture of a feedforward backpropagation network is shown in Figure 3.1. While there can be many hidden layers, we will illustrate this network with only one hidden layer. Also, the number of neurons in the input layer and that in the output layer are determined by the dimensions of the input and output patterns, respectively. It is not easy to determine how many neurons are needed for the hidden layer. In order to avoid cluttering the figure, we will show the layout in Figure 3.1 with four input neurons, three neurons in the hidden layer, and one output neuron(s), with a few representative connections. The network has three fields of neurons: one for input neurons, one for hidden processing elements, and one for the output neurons. As already stated, connections are for feed forward activity. There are connections from every neuron in field A to every one in field B, and, in turn, from every neuron in field B to every neuron in field C. Thus, there are two sets of weights, those figuring in the activations of hidden layer neurons, and those that help determine the output neuron activations. In training, all of these weights are adjusted by considering what can be called a cost function in terms of the error in the computed output pattern and the desired output pattern.

3.2 Training

The feedforward backpropagation network undergoes supervised training, with a finite number of pattern pairs consisting of an input pattern and

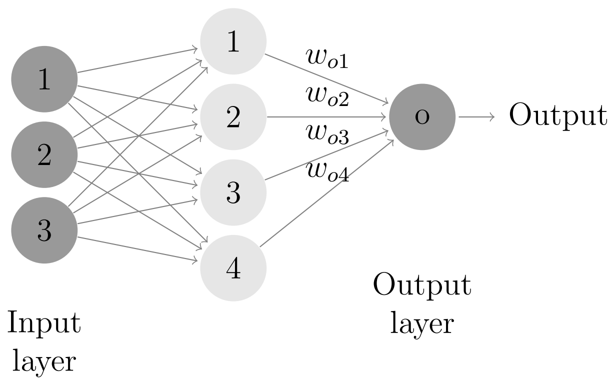


Figure 3.1: An simple Backpropagation Network

a desired or target output pattern. An input pattern is presented at the input layer. The neurons here pass the pattern activations to the next layer neurons, which are in a hidden layer. The outputs of the hidden layer neurons are obtained by using perhaps a **bias**, and also a threshold function with the activations determined by the weights and the inputs. These hidden layer outputs become inputs to the output neurons, which process the inputs using an optional bias and a threshold function. The final output of the network is determined by the activations from the output layer. The computed pattern and the input pattern are compared, a function of this error for each component of the pattern is determined, and adjustment to weights of connections between the **hidden layer** and **the output layer** is computed. A similar computation, still based on the error in the output, is made for the connection weights between the **input** and **hidden layers**. The procedure is repeated with each pattern pair assigned for training the network.Each pass through all the training patterns is called a **cycle or an epoch**. The process is then repeated as many cycles as needed until the error is within a prescribed tolerance.

3.3 Illustration

3.3.1 Adjustment of Weights of Connections from a Neuron in the Hidden Layer

We will be as specific as is needed to make the computations clear. First recall that the activation of a neuron in a layer other than the input layer is the sum of products of its inputs and the weights corresponding to the connections that bring in those inputs. Let us discuss the jth neuron in the hidden layer. Let us be specific and say $j = 2$. Suppose that the input pattern is $(1.1, 2.4, 3.2, 5.1, 3.9)$ and the target output pattern is $(0.52, 0.25, 0.75, 0.97)$. Let the weights be given for the second hidden layer neuron by the vector $(-0.33, 0.07, -0.45, 0.13, 0.37)$. The activation will be the quantity:

$$(-0.33*1.1)+(0.07*2.4)+(-0.45*3.2)+(0.13*5.1)+(0.37*3.9) = 0.471$$

Now add to this an optional bias of, say, 0.679, to give 1.15. If we use the sigmoid function given by:

$$\frac{1}{1+\exp^{-x}}$$

with x = 1.15, we get the output of this hidden layer neuron as 0.7595.

We need the computed output pattern also. Let us say it turns out to be *actual* $= (0.61, 0.41, 0.57, 0.53)$, while the desired pattern is *desired* $= (0.52, 0.25, 0.75, 0.97)$. Obviously, there is a discrepancy between what is desired and what is computed. The component-wise differences are given in the vector, $desired - actual = (-0.09, -0.16, 0.18, 0.44)$.

We use these to form another vector where each component is a product of the error component, corresponding computed pattern component, and the complement of the latter with respect to 1. For example, for the first component, error is -0.09, computed pattern component is 0.61, and its complement is 0.39. Multiplying these together $(0.61 * 0.39 * -0.09)$, we get -0.02. Calculating the other components similarly, we get the vector $(-0.02, -0.04, 0.04, 0.11)$. i.e error $= actual \times (1 - actual) \times (d - a)$

The $desired - actual$ vector, which is the error vector multiplied by the actual output vector, gives you a value of error reflected back at the output of the hidden layer. This is scaled by a value of (1-output vector), which is the first derivative of the output activation function for numerical stability). You will see the formulas for this process later in this chapter. The backpropagation of errors needs to be carried further. We need now the weights on the connections between the second neuron in the hidden layer that we are concentrating on, and the different output neurons. Let us say these weights are given by the vector $(0.85, 0.62, 0.10, 0.21)$. The error of the second neuron in the hidden layer is now calculated as below, using its output.

$error = 0.7595 * (1 - 0.7595) * ((0.85 * -0.02) + (0.62 * -0.04) + (-0.10 * 0.04) + (0.21 * 0.11)) = -0.0041$.

Again, here we multiply the error ($e.g., -0.02$) from the output of the current layer, by the output value (0.7595) and the value $(1 - 0.7595)$. We use the weights on the connections between neurons to work backwards through the network. Next, we need the learning rate parameter for this layer; let us set it as 0.2. We multiply this by the output of the second neuron in the hidden layer, to get 0.1519. Each of the components of the vector $(-0.02, -0.04, 0.04, 0.11)$ is multiplied now by 0.1519, which our latest computation gave. The result is a vector that gives the adjustments to the weights on the connections that go from the second neuron in the hidden layer to the output neurons. These values are given in the vector $(-0.003, -0.006, 0.006, 0.017)$. After these adjustments are added, the weights to be used in the next cycle on the connections between the second neuron in the hidden layer and the output neurons become those in the vector $(0.847, 0.614, -0.094, 0.227)$.

3.3.2 Adjustment of Weights of Connections from a Neuron in the Input Layer

Let us look at how adjustments are calculated for the weights on connections going from the ith neuron in the input layer to neurons in the hidden layer. Let us take specifically i = 3, for illustration. Much of the information we need is already obtained in the previous discussion for the second hidden layer neuron. We have the errors in the computed output as the vector (0.09, 0.16, 0.18, 0.44), and we obtained the error for the second neuron in the hidden layer as 0.0041, which was not used above. Just as the error in the output is propagated back to assign errors for the neurons in the hidden layer, those errors can be propagated to the input layer neurons. To determine the adjustments for the weights on connections between the input and hidden layers, we need the errors determined for the outputs of hidden layer neurons, a learning rate parameter, and the activations of the input neurons, which are just the input values for the input layer. Let us take the learning rate parameter to be 0.15. Then the weight adjustments for the connections from the third input neuron to the hidden layer neurons are obtained by multiplying the particular hidden layer neurons output error by the learning rate parameter and by the input component from the input neuron. The adjustment for the weight on the connection from the third input neuron to the second hidden layer neuron is 0.15 * 3.2 * 0.0041, which works out to 0.002. If the weight on this connection is, say, 0.45, then adding the adjustment of -0.002, we get the modified weight of 0.452, to be used in the next iteration of the network operation. Similar calculations are made to modify all other weights as well.

3.3.3 Adjustments to Threshold Values or Biases

The bias or the threshold value we added to the activation, before applying the threshold function to get the output of a neuron, will also be adjusted based on the error being propagated back. The needed values for this are in the previous discussion. The adjustment for the threshold value of a neuron in the output layer is obtained by multiplying the calculated error (not just the difference) in the output at the output neuron and the learning rate parameter used in the adjustment calculation for weights at this layer. In our previous example, we have the learning rate parameter as 0.2, and the error vector as (0.02, 0.04, 0.04, 0.11), so the adjustments to the threshold values of the four output neurons are given by the vector $(0.004, 0.008, 0.008, 0.022)$. These adjustments are added to the current levels of threshold values at the output neurons. The adjustment to the threshold value of a neuron in the hidden layer is obtained similarly by multiplying the learning rate with the computed error in the output of the hidden layer neuron. Therefore, for the second neuron in the hidden layer, the adjustment to its threshold value is calculated as $0.15 * 0.0041$, which is 0.0006. Add this to the current threshold value of 0.679 to get 0.6784, which is to be used for this neuron in the next training pattern for the neural network.

3.4 Mathematical Derivations

3.4.1 Nomenclature

The nomenclature we use in the training algorithm for the backpropagation net is as follows:
$\boldsymbol{x}$ Input training vector: $X = (x_1, ..., x_i, ..., x_n)$.
$\boldsymbol{t}$ Output target vector: $t = (t_1, ..., t_k, ..., t_m)$.
$\boldsymbol{\delta_k}$ Portion of error correction weight adjustment for w_{jk} that is due to an error at output unit Y_k; also, the information at the unit Y_k that is propagated back to the hidden units that feed into one unit Y_k.
$\boldsymbol{\delta_i}$ Portion of error correction weight adjustment for V_{ij} that is due to the backpropagation of error information from the output layer to the hidden unit Z_j.
$\boldsymbol{\eta}$ Learning rate
$\boldsymbol{X_i}$ Input unit i: For an input unit, the input signal and output signal are the same, namely, x_i.
$\boldsymbol{v_{Oj}}$ Bias on the hidden unit j.
$\boldsymbol{Z_j}$ Hidden unit j:
The net input to Z_j is denoted $z\_in_i = v_{oj} + \sum_i x_i v_{ij}$.
The output signal(activation) of Z_j is denoted by $z_j = f(z\_in_i)$.
$\boldsymbol{w_{ok}}$ Bias on output unit k.
$\boldsymbol{Y_k}$ Output unit k:
The net input to Y_k is denoted by $y\_in_k = w_{ok} + \sum_j z_j w_{jk}$
The output signal(activation) of Y_k is denoted by $y_k = f(y\_in_k)$.

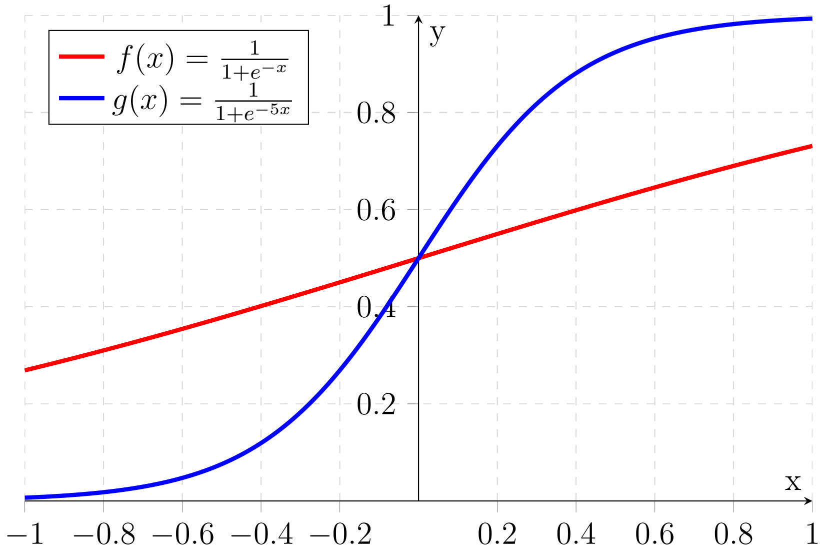
3.4.2 Activation Function

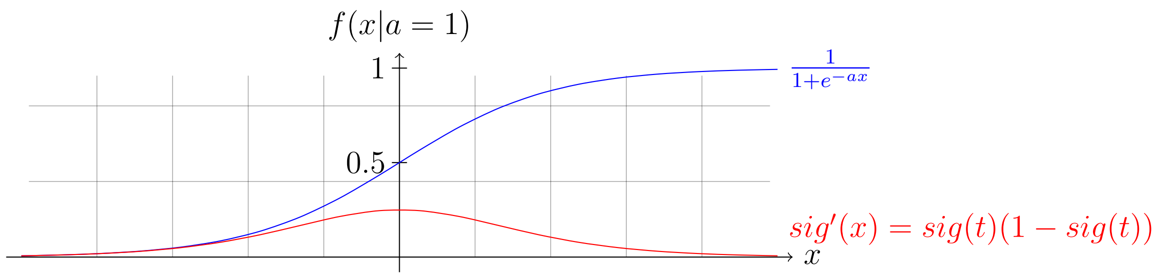
An activation function for back propagation should have several important characteristics:
1. It should be continuous, differentiable, and monotonically decreasing.

2. Furthermore, for computational efficiency, it is desirable that its derivative be easy to compute. For the most commonly used activation functions, the value of the derivative (at particular value of the independent variable) can be expressed in terms of the value of the function (at that value of the independent variable). Usually the function is expected to *saturate*, i.e., approach finite maximum and minimum values asymptotically.

3.4.2.1 Sigmoid Function

One of the most typical activation functions is the binary sigmoid function, which has range (0,1) and is defined as
$f_1(x) = \frac{1}{1+\exp(-x)}$,
with
$f_1'(x) = f_1(x)[1 - f_1(x)]$





3.4.3 Derivation

- Uses Delta learning rule

$$\begin{aligned} \Delta w &= \eta r x \\ r x &= \Delta E \\ \Delta w &= -\eta \Delta E \end{aligned}$$

- It follows gradient descent method

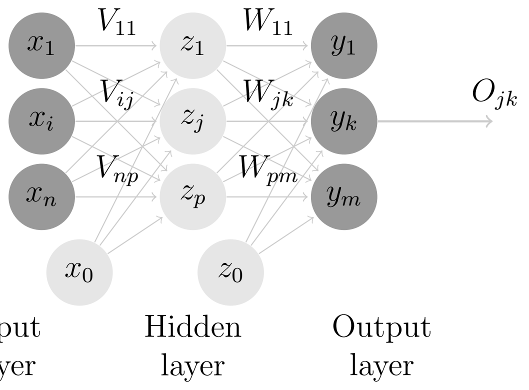


Figure 3.2: An Multi Layer Backpropagation Network

- If bias is not included in the network the activation fuction

$$f(net) = \begin{cases} 1 & Net > \theta \\ -1 & Net < \theta \end{cases}$$

$$Net = \sum w^T x$$

- If bias is included we assume $\theta = 0$

$$f(net) = \begin{cases} 1 & Net > 0 \\ -1 & Net < 0 \end{cases}$$

3.5 Weight Updation in Output Layer

$$W_{jk}(t+1) = W_{jk}(t) + \Delta W_{jk}$$
$$\Delta W_{jk} = \eta r x$$
In delta rule
$$r = (d_i - O_i) f'(Net)$$
$$\Delta W_{jk} = \eta (t_k - O_{jk}) f'(O_{jk}) O_{jk}$$

$$\Delta W_{jk} = \eta (t_k - O_{jk}) O_{jk} (1 - O_{jk}) O_{ij} \tag{3.1}$$

3.5.1 Proof:

$$E = t_k - O_{j,k}$$
$$By Least Mean Square$$
$$E = \frac{1}{2}(t_k - O_{jk})^2$$
$$O_{jk} = f(Net_{jk})$$

$$Net_{jk} = \sum W_{jk} \times Ojk \tag{3.2}$$

According to delta learning rule

$$\Delta W = -\eta \Delta E$$
$$\Delta W_{jk} = -\eta \frac{\partial \Delta E}{\partial W_{j,k}}$$

$$\frac{\partial \Delta E}{\partial W_{j,k}} = \frac{\partial \Delta E}{\partial Net_{j,k}} \times \frac{\partial Net_{j,k}}{\partial W_{j,k}} \tag{3.3}$$

$$\begin{aligned}
\mathit{\Delta} W_{jk} &= -\eta \frac{\partial \mathit{\Delta} E}{\partial Net_{j,k}} \times \frac{\partial Net_{j,k}}{\partial W_{j,k}} \\
&= -\frac{\partial \mathit{\Delta} E}{\partial Net_{j,k}} \times \frac{\partial Net_{j,k}}{\partial W_{j,k}}
\end{aligned}$$

By considering $\eta = -1$

$$\begin{aligned}
&= \delta_{jk} \times \frac{\partial Net_{jk}}{\partial W_{jk}} \\
\frac{\partial Net_{jk}}{\partial W_{jk}} &= \frac{\partial \sum W_{jk} O_{ij}}{\partial W_{jk}} \\
&= O_{ij} \\
\mathit{\Delta} W_{jk} &= \delta_{jk} O_{ij} \\
\delta_{jk} &= -\frac{\partial E}{\partial Net_{jk}}
\end{aligned}$$

By chain rule

$$\begin{aligned}
\delta_{jk} &= -\frac{\partial E}{\partial O_{jk}} \times \frac{\partial O_{jk}}{\partial Net_{jk}} \\
\frac{\partial E}{\partial O_{jk}} &= \frac{\partial \frac{1}{2}(t_k - O_{jk})^2}{\partial O_{jk}} \\
&= -t_k + O_{jk} \\
\frac{\partial O_{jk}}{\partial Net_{jk}} &= \frac{\partial F(Net_{jk})}{\partial Net_{jk}} \\
&= F'(Net_{jk}) \\
&= O_{jk}(1 - O_{jk}) \\
\delta_{jk} &= (t_k - O_{jk}) O_{jk} (1 - Ojk) \\
\mathit{\Delta} &= (t_k - O_{jk}) O_{jk} (1 - O_{jk}) O_{ij}
\end{aligned}$$

3.5.2 Weight Updation in the Hidden Layer

$$\mathit{\Delta} V_{ij} = \eta\ \delta_{ij}\ x$$

$$\begin{aligned}
\delta_{ij} &= -\frac{\partial \mathit{\Delta} E}{\partial Net_{ij}} \\
&= -\frac{\partial \mathit{\Delta} E}{\partial O_{ij}} \times \frac{O_{ij}}{\partial Net_{ij}} \\
&= -\frac{\partial \mathit{\Delta} E}{\partial Net_{jk}} \times \frac{\partial Net_{jk}}{\partial O_{ij}} \times \frac{\partial}{\partial Net_{ij}} f(Net_{ij}) \\
&= -\delta_{jk} \times \frac{\partial}{\partial O_{ij}} \sum W_{jk} O_{ij} \times f'(Net_{ij}) \\
&= -\delta_{jk} \times \sum W_{jk} \times O_{ij}(1 - O_{ij})
\end{aligned}$$

$$\mathit{\Delta} V_{ij} = \eta\ \delta_{jk}\ \sum W_{jk}\ O_{ij}(1 - O_{ij})\ x$$

3.6 Algorithm

1. Initialize the weights
2. Choose proper activation function
3. For each training input vector do the following steps until $\mathit{\Delta} w = 0$
   a) Calculate the net value of hidden layer using inputs and weights
      $Net_{i,j} = \sum V_{i,j} x_i$
   b) Apply activation function and find output of hidden layer or input of output layer
      $O_{i,j} = f(Net_{i,j})$
   c) Calculate the Net value of the output layer
      $Net_{i,j} = \sum W_{i,j} O_{i,j}$
   d) Apply activation function and find output of output layer
      $O_{j,k} = f(Net_{j,k})$
   e) Calculate the error $E = t_k - O_{j,k}$, using LMS error principle
   f) Calculate the portion of the error ∂_{jk} which has to be back propagated to the hidden layer
      $\partial_{jk} = -\frac{\partial E}{\partial Net_{jk}}$
      $\quad = (t_k - O_{jk}) O_{jk} (1 - O_{jk})$
   g) Adjust the weights $\mathit{\Delta} W_{jk} = \eta \partial_{jk} O_{ij}$

h) Calculate the portion of the error ∂_{ij} that has to be back propagated to input layers
$\partial_{ij} = \partial_{jk} \sum(W_{jk}) O_{ij}(1 - O_{ij})$

i) Adjust the weights

j) Check for weight convergence

Chapter 4

Reference Links

4.1 Backpropagation:

http://en.wikipedia.org/wiki/Delta_rule
http://en.wikipedia.org/wiki/Backpropagation
https://www4.rgu.ac.uk/files/chapter3%20-%20bp.pdf
http://docs.opencv.org/modules/ml/doc/neural_networks.html

4.2 Maths

http://en.wikipedia.org/wiki/Partial_derivative
http://en.wikipedia.org/wiki/Gradient_descent
http://en.wikipedia.org/wiki/Chain_rule
http://en.wikipedia.org/wiki/Power_rule

Chapter 5

Instrumentation, Measurement, and Industrial Applications

Instrumentation and measurement play a relevant role in any industrial applications. Without sensors, transducers, converters, acquisition channels, signal processing, image processing, no measurement system and procedure will exist and, in turn, no industry will actually exist. They are in fact the irreplaceable foundation of any monitoring and automatic control system as well as for any diagnosis and quality assurance.

A number of results concerning the use of neural techniques are known in different applications, encompassing intelligent sensors and acquisition systems, system models, signal processing, image processing, automatic control systems, and diagnosis.

5.1 Fundamentals

The concept of measurement has been deep-rooted in the human culture since the origin of civilization, as it has always represented the basis of the experimental knowledge, the quantitative assessment of goods in commercial transactions, the assertion of a right, and so on. After Galileo Galilei put experimentation at the base of the modern science and showed that it is the only possible starting point for the validation of any scientific theory, the measurement activity has become more and more important. More than one century ago, William Thomson, Lord Kelvin, reinforced this concept by stating: **"I often say that when you can measure what you are speaking about, and can express it in numbers, you know something about it; but when you cannot express it in numbers your knowledge about it is of meager and unsatisfactory kind; it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of science, whatever the matter may be. So, therefore, if science is measurement, then without metrology there can be no science"**.

Under this modem vision of science, the measurement of a physical quantity is generally defined as the quantitative comparison of this same quantity with another one, which is homogeneous with the measured one, and is considered as the measurement unit. In order to perform this quantitative comparison, five agents are needed,

- **The measurand**: it is the quantity to be measured, and it often represents a property of a physical object and is described by a suitable mathematical model.
- **The standard**: it is the physical realization of the measurement unit.
- **The instrument**: it is the physical device that performs the comparison.
- **The method**: the comparison between the measurand and the standard is performed by exploiting some physical phenomena (thermal dilatation, mechanical force between electric charges, and so on); according to the considered phenomenon, different methods can be implemented.
- **The operator**: he supervises the whole measurement process, operates the measurement devices and reads the instrument.

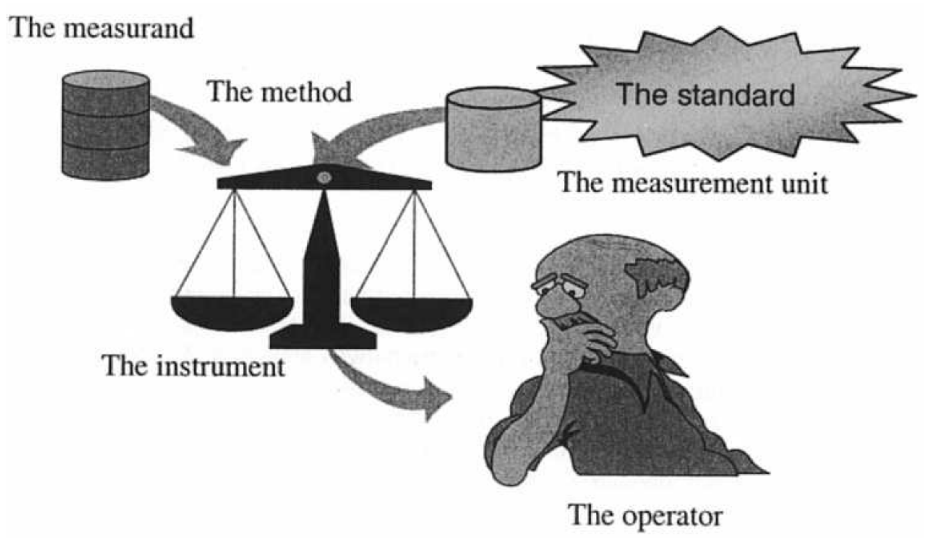


Figure 5.1: Pictorial representation of Measurement

Chapter 6

Common Terms to Start with Neural Network

6.1 Neural Network

A neural network is a massively parallel distributed processor made up of simple processing units, which has a natural propensity for storing experimental knowledge and making it available for use. It resembles the brain in two aspects:

1. Knowledge is acquired by the network from its environment through a learning process.
2. Inter-neuron connections strengths, known as synaptic weights, are used to store the acquired knowledge.

6.2 Stability for a Neural Network

Stability refers to such convergence that facilitates an end to the iterative process. For example, if any two consecutive cycles result in the same output for the network, then there may be no need to do more iterations. In this case, convergence has occurred, and the network has stabilized in its operation. If weights are being modified after each cycle, then convergence of weights would constitute stability for the network. In some situations, it takes many more iterations than you desire, to have output in two consecutive cycles to be the same. Then a tolerance level on the convergence criterion can be used. With a tolerance level, you accomplish early but satisfactory termination of the operation of the network.

6.3 Plasticity for a Neural Network

Suppose a network is trained to learn some patterns, and in this process the weights are adjusted according to an algorithm. After learning these patterns and encountering a new pattern, the network may modify the weights in order to learn the new pattern. But what if the new weight structure is not responsive to the new pattern? Then the network does not possess plasticitythe ability to deal satisfactorily with new short-term memory (STM) while retaining long-term memory (LTM). Attempts to endow a network with plasticity may have some adverse effects on the stability of your network.

Plasticity permits the developing nervous system to adapt to its surrounding environment.

6.4 Short-Term Memory and Long-Term Memory

We alluded to short-term memory (STM) and long-term memory (LTM) in the previous paragraph. STM is basically the information that is currently and perhaps temporarily being processed. It is manifested in the patterns that the network encounters. LTM, on the other hand, is information that is already stored and is not being currently processed. In a neural network, STM isusually characterized by patterns and LTM is characterized by the connections weights. The weights determine how an input is processed in the network to yield output. During the cycles of operation of a network, the weights may change. After convergence, they represent LTM, as the weight levels achieved are stable.

6.5 Generalization

It refers to the ability of neural networks to produce reasonable outputs for inputs not encountered during the training.

6.6 Learning Algorithm

The procedure used to perform the learning process is called a learning algorithm, the function of which is to modify the synaptic weights of the network in an orderly fashion to attain the design objective.

6.7 Layers in a Neural Network

A neural network has its neurons divided into subgroups, or fields, and elements in each subgroup are placed in a row, or a column, in the diagram depicting the network. Each subgroup is then referred to as a layer of neurons in the network.A great many models of neural networks have two layers, quite a few have one layer, and some have three or more layers. A number of additional, so-called hidden layers are possible in some networks, such as the Feed-forward backpropagation network. When the network has a single layer, the input signals are received at that layer, processing is done by its neurons,and output is generated at that layer. When more than one layer is present, the first field is for the neurons that supply the input signals for the neurons in the next layer. A layer is also referred to as a field. Then the different layers can be designated as field A, field B, and so on, or shortly, F_A , F_B .

6.8 Neural Processing

How do you recognize a face in a crowd? How does an economist predict the direction of interest rates? Faced with problems like these, the human brain uses a web of interconnected processing elements called neurons to process information. Each neuron is autonomous and independent; it does its work asynchronously, that is, without any synchronization to other events taking place. The two problems posed, namely recognizing a face and forecasting interest rates, have two important characteristics that distinguish them from other problems: First, the problems are complex, that is, you cant devise a simple step-by-step algorithm or precise formula to give you an answer; and second, the data provided to resolve the problems is equally complex and may be noisy or incomplete. You could have forgotten your glasses when youre trying to recognize that face. The economist may have

at his or her disposal thousands of pieces of data that may or may not be relevant to his or her forecast on the economy and on interest rates.
The vast processing power inherent in biological neural structures has inspired the study of the structure itself for hints on organizing human-made computing structures. Artificial neural networks, the subject of this book, covers the way to organize synthetic neurons to solve the same kind of difficult, complex problems in a similar manner as we think the human brain may. This chapter will give you a sampling of the terms and nomenclature used to talk about neural networks. These terms will be covered in more depth in the chapters to follow.

Chapter 7

Types of Networks

The models that we overview in this chapter are the

1. Perceptron
2. Hopfield
3. Adaline
4. Feed-Forward Backpropagation
5. Bidirectional Associative Memory
6. Brain-State-in-a-Box
7. Neocognitron
8. Fuzzy Associative Memory
9. ART1
10. ART2

7.1 Based on Learning

A network can be subject to supervised or unsupervised learning. The learning would be supervised if external criteria are used and matched by the network output, and if not, the learning is unsupervised. This is one broad way to divide different neural network approaches. Unsupervised approaches are also termed self-organizing. There is more interaction between neurons, typically with feedback and intralayer connections between neurons promoting self-organization.
Supervised networks are a little more straightforward to conceptualize than unsupervised networks. You apply the inputs to the supervised network

along with an expected response, much like the Pavlovian conditioned stimulus and response regimen. You mold the network with stimulus-response pairs. Astock market forecaster may present economic data (the stimulus) along with metrics of stock market performance (the response) to the neural network to the present and attempt to predict the future once training is complete.

You provide unsupervised networks with only stimulus. You may, for example, want an unsupervised network to correctly classify parts from a conveyor belt into part numbers, providing an image of each part to do the classification (the stimulus). The unsupervised network in this case would act like a look-up memory that is indexed by its contents, or a Content-Addressable-Memory (CAM).

Chapter 8

External Exploration

8.1 Organizations

Below of some of the organizations which are having major impact on Neural Network research.

- IEEE Neural Network Council,
- The INNS - International Neural Network Society
- The ENNS - European Neural Network Society (the most re-known and largest international scientific/technological non-profit associations concerned with neural networks).
- AIIA - Italian Association for Artificial Intelligence, SIREN - Italian Association for Neural Networks,
- UNIMI-DTI - University of Milan: Department of Information Technologies.

Chapter 9

What is Perceptron?

In machine learning, the perceptron is an algorithm for supervised classification of an input into one of several possible non-binary outputs. It is a type of linear classifier, i.e. a classification algorithm that makes its predictions based on a linear predictor function combining a set of weights with the feature vector. The algorithm allows for online learning, in that it processes elements in the training set one at a time.

The perceptron algorithm dates back to the late 1950s; its first implementation, in custom hardware, was one of the first artificial neural networks to be produced.

9.1 A Bit of History

The perceptron algorithm was invented in 1957 at the Cornell Aeronautical Laboratory by Frank Rosenblatt, funded by the United States Office of Naval Research.

More on the web!

9.2 Definition

In the modern sense, the perceptron is an algorithm for learning a **binary classifier**: a function that maps its input x (a real-valued vector) to an output value f(x) (a single binary value):

$$f(x) = \begin{cases} 1 : & w.x + b > 0 \\ 0 : & otherwise \end{cases}$$

where...

- **w** is a vector of real-valued weights,
- $\mathbf{w} \cdot \mathbf{x}$ is the dot product (which here computes a weighted sum), and **b** is the 'bias',
- a constant term that does not depend on any input value.

If b is negative, then the weighted combination of inputs must produce a positive value greater than $|b|$ in order to push the classifier neuron over the 0 threshold. Spatially, the bias alters the position (though not the orientation) of the decision boundary. The perceptron learning algorithm does not terminate if the learning set is not linearly separable. If the vectors are not linearly separable learning will never reach a point where all vectors are classified properly. The most famous example of the perceptron's inability to solve problems with linearly nonseparable vectors is the Boolean exclusive-or problem.

In the context of neural networks, a perceptron is an **artificial neuron** using the **Heaviside step function** as the activation function. The perceptron algorithm is also termed the **single-layer perceptron**, to distinguish it from a multilayer perceptron, which is a misnomer for a more complicated neural network. As a linear classifier, the single-layer perceptron is the **simplest feedforward neural network**.

9.3 Learning Algorithm

Below is an example of a learning algorithm for a (single-layer) perceptron. For multilayer perceptrons, where a hidden layer exists, more sophisticated algorithms such as **backpropagation** must be used. Alternatively, methods such as the **delta rule** can be used if the function is **non-linear** and **differentiable**, although the one below will work as well.

When multiple perceptrons are combined in an artificial neural network, each output neuron operates independently of all the others; thus, learning each output can be considered in isolation.

9.3.0.1 Definition

We first define some variables:

- $y = f(\vec{\mathbf{z}})$ denotes the output from the perceptron for an input vector $\vec{\mathbf{z}}$.
- **b** is the bias term, which in the example below we take to be 0.
- $D = \{\vec{\mathbf{X}}_1, d_1),\ldots,(\vec{\mathbf{X}}_s, d_s)\}$ is the training set of s samples, where:
  - $\vec{\mathbf{X}}_j$ is the n-dimensional input vector.
  - $\mathbf{d_j}$ is the desired output value of the perceptron for that input.

We show the values of the features as follows:

- $x_{j,i}$ is the value of the ith feature of the jth training input vector.
- $x_{j,0} = 1$.

To represent the weights:

- w_i is the ith value in the weight vector, to be multiplied by the value of the ith input feature.
- Because $x_{j,0} = 1$, the w_0 is effectively a learned bias that we use instead of the bias constant b.

To show the time-dependence of $\mathbf{w}$, we use:

- $w_i(t)$ is the weight i at time t.
- α is the learning rate, where $0 < \alpha \leq 1$.
  Too high a learning rate makes the perceptron periodically oscillate around the solution unless additional steps are taken.

9.3.1 Steps

1. Initialise the weights and the threshold. Weights may be initialised to 0 or to a small random value. In the example below, we use 0.
2. For each example j in our training set D , perform the following steps over the input $\mathbf{x}_j$ and desired output d_j :
   a) Calculate the actual output:
   $\mathbf{y_j(t) = f[w(t) \cdot x_j] = f[w_0(t) + w_1(t)x_{j,1} + w_2(t)x_{j,2} + \cdots + w_n(t)x_{j,n}]}$
   b) Update the weights:
   $\mathbf{w_i(t+1) = w_i(t) + ff(d_j - y_j(t))x_{j,i}}$ for all feature $\mathbf{0 \leq i \leq n}$.
3. For offline learning, the step 2 may be repeated until the iteration error $\mathbf{\frac{1}{s}\sum_{j=1}^{s} |d_j - y_j(t)|}$ is less than a user-specified error threshold **fl**, or a predetermined number of iterations have been completed.

The algorithm updates the weights after steps $2a$ and $2b$. These weights are immediately applied to a pair in the training set, and subsequently updated, rather than waiting until all pairs in the training set have undergone these steps.

9.4 Convergence

The perceptron is a linear classifier, therefore it will never get to the state with all the input vectors classified correctly if the training set $\mathbf{D}$ is not linearly separable, i.e. if the positive examples can not be separated from the negative examples by a hyperplane.

9.5 Pocket Algorithm

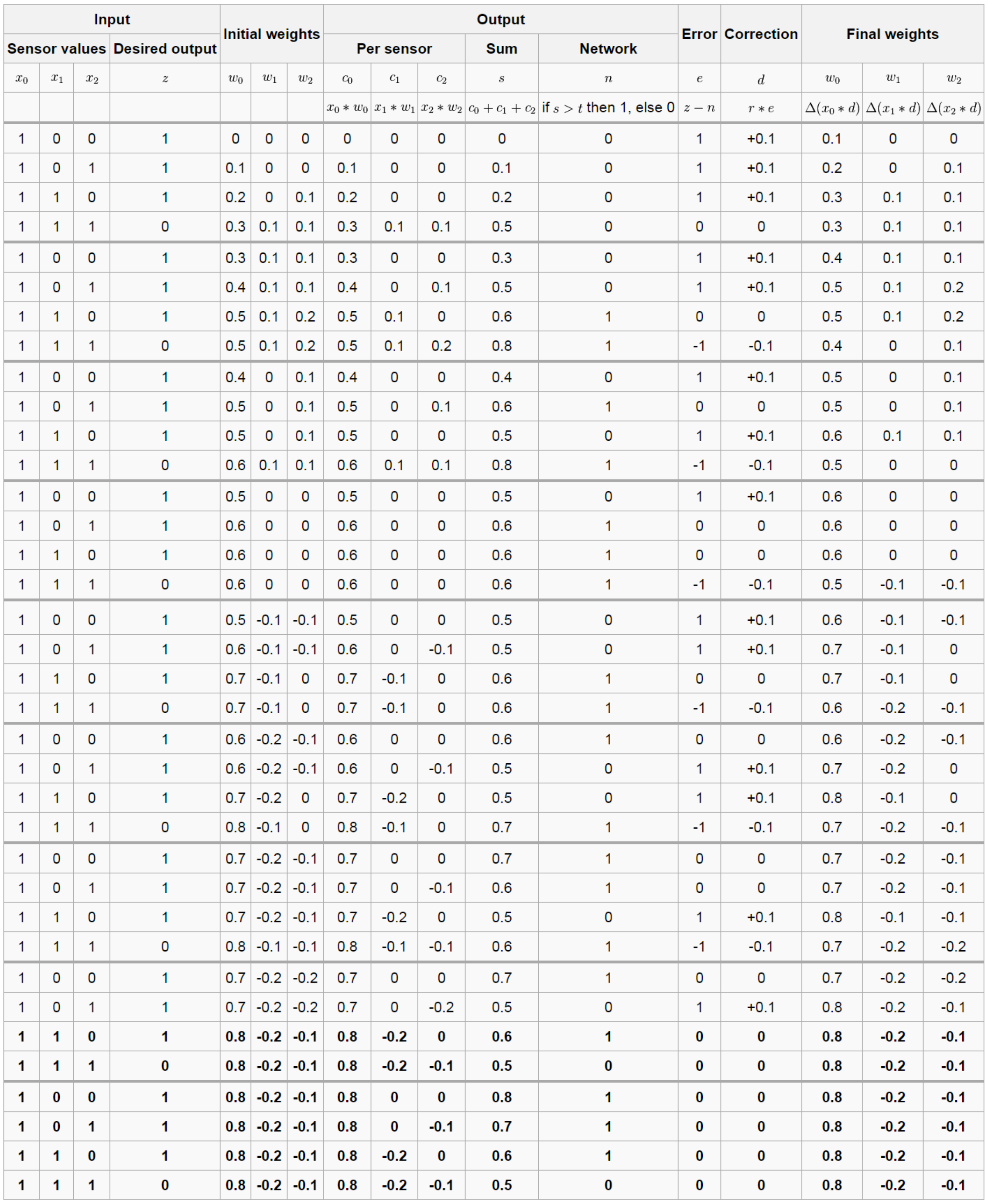
The pocket algorithm with ratchet (Gallant, 1990) solves the stability problem of perceptron learning by keeping the best solution seen so far "in its pocket". The pocket algorithm then returns the solution in the pocket, rather than the last solution. It can be used also for non-separable data sets, where the aim is to find a perceptron with a small number of misclassifications.

9.6 Example

A perceptron learns to perform a binary NAND function on inputs x_1 and x_2 .

- Inputs: x_0 , x_1 , x_2 , with input x_0 held constant at 1.
- Threshold (θ): 0.5
- Bias (b): 0
- Learning rate (r): 0.1
- Training set, consisting of four samples: $\{((1,0,0),1),((1,0,1),1),((1,1,0),1),((1,1,1),0)\}$

In the following, the final weights of one iteration become the initial weights of the next. Each cycle over all the samples in the training set is demarcated with heavy lines.



Input				Initial weights			Output					Error	Correction	Final weights		
Sensor values			Desired output				Per sensor			Sum	Network					
x_0	x_1	x_2	z	w_0	w_1	w_2	c_0	c_1	c_2	s	n	e	d	w_0	w_1	w_2
							$x_0 * w_0$	$x_1 * w_1$	$x_2 * w_2$	$c_0 + c_1 + c_2$	if $s > t$ then 1, else 0	$z - n$	$r * e$	$\Delta(x_0 * d)$	$\Delta(x_1 * d)$	$\Delta(x_2 * d)$
1	0	0	1	0	0	0	0	0	0	0	0	1	+0.1	0.1	0	0
1	0	1	1	0.1	0	0	0.1	0	0	0.1	0	1	+0.1	0.2	0	0.1
1	1	0	1	0.2	0	0.1	0.2	0	0	0.2	0	1	+0.1	0.3	0.1	0.1
1	1	1	0	0.3	0.1	0.1	0.3	0.1	0.1	0.5	0	0	0	0.3	0.1	0.1
1	0	0	1	0.3	0.1	0.1	0.3	0	0	0.3	0	1	+0.1	0.4	0.1	0.1
1	0	1	1	0.4	0.1	0.1	0.4	0	0.1	0.5	0	1	+0.1	0.5	0.1	0.2
1	1	0	1	0.5	0.1	0.2	0.5	0.1	0	0.6	1	0	0	0.5	0.1	0.2
1	1	1	0	0.5	0.1	0.2	0.5	0.1	0.2	0.8	1	-1	-0.1	0.4	0	0.1
1	0	0	1	0.4	0	0.1	0.4	0	0	0.4	0	1	+0.1	0.5	0	0.1
1	0	1	1	0.5	0	0.1	0.5	0	0.1	0.6	1	0	0	0.5	0	0.1
1	1	0	1	0.5	0	0.1	0.5	0	0	0.5	0	1	+0.1	0.6	0.1	0.1
1	1	1	0	0.6	0.1	0.1	0.6	0.1	0.1	0.8	1	-1	-0.1	0.5	0	0
1	0	0	1	0.5	0	0	0.5	0	0	0.5	0	1	+0.1	0.6	0	0
1	0	1	1	0.6	0	0	0.6	0	0	0.6	1	0	0	0.6	0	0
1	1	0	1	0.6	0	0	0.6	0	0	0.6	1	0	0	0.6	0	0
1	1	1	0	0.6	0	0	0.6	0	0	0.6	1	-1	-0.1	0.5	-0.1	-0.1
1	0	0	1	0.5	-0.1	-0.1	0.5	0	0	0.5	0	1	+0.1	0.6	-0.1	-0.1
1	0	1	1	0.6	-0.1	-0.1	0.6	0	-0.1	0.5	0	1	+0.1	0.7	-0.1	0
1	1	0	1	0.7	-0.1	0	0.7	-0.1	0	0.6	1	0	0	0.7	-0.1	0
1	1	1	0	0.7	-0.1	0	0.7	-0.1	0	0.6	1	-1	-0.1	0.6	-0.2	-0.1
1	0	0	1	0.6	-0.2	-0.1	0.6	0	0	0.6	1	0	0	0.6	-0.2	-0.1
1	0	1	1	0.6	-0.2	-0.1	0.6	0	-0.1	0.5	0	1	+0.1	0.7	-0.2	0
1	1	0	1	0.7	-0.2	0	0.7	-0.2	0	0.5	0	1	+0.1	0.8	-0.1	0
1	1	1	0	0.8	-0.1	0	0.8	-0.1	0	0.7	1	-1	-0.1	0.7	-0.2	-0.1
1	0	0	1	0.7	-0.2	-0.1	0.7	0	0	0.7	1	0	0	0.7	-0.2	-0.1
1	0	1	1	0.7	-0.2	-0.1	0.7	0	-0.1	0.6	1	0	0	0.7	-0.2	-0.1
1	1	0	1	0.7	-0.2	-0.1	0.7	-0.2	0	0.5	0	1	+0.1	0.8	-0.1	-0.1
1	1	1	0	0.8	-0.1	-0.1	0.8	-0.1	-0.1	0.6	1	-1	-0.1	0.7	-0.2	-0.2
1	0	0	1	0.7	-0.2	-0.2	0.7	0	0	0.7	1	0	0	0.7	-0.2	-0.2
1	0	1	1	0.7	-0.2	-0.2	0.7	0	-0.2	0.5	0	1	+0.1	0.8	-0.2	-0.1
1	**1**	**0**	**1**	**0.8**	**-0.2**	**-0.1**	**0.8**	**-0.2**	**0**	**0.6**	**1**	**0**	**0**	**0.8**	**-0.2**	**-0.1**
1	**1**	**1**	**0**	**0.8**	**-0.2**	**-0.1**	**0.8**	**-0.2**	**-0.1**	**0.5**	**0**	**0**	**0**	**0.8**	**-0.2**	**-0.1**
1	**0**	**0**	**1**	**0.8**	**-0.2**	**-0.1**	**0.8**	**0**	**0**	**0.8**	**1**	**0**	**0**	**0.8**	**-0.2**	**-0.1**
1	**0**	**1**	**1**	**0.8**	**-0.2**	**-0.1**	**0.8**	**0**	**-0.1**	**0.7**	**1**	**0**	**0**	**0.8**	**-0.2**	**-0.1**
1	**1**	**0**	**1**	**0.8**	**-0.2**	**-0.1**	**0.8**	**-0.2**	**0**	**0.6**	**1**	**0**	**0**	**0.8**	**-0.2**	**-0.1**
1	**1**	**1**	**0**	**0.8**	**-0.2**	**-0.1**	**0.8**	**-0.2**	**-0.1**	**0.5**	**0**	**0**	**0**	**0.8**	**-0.2**	**-0.1**

Figure 9.1: Example of weight updates for the Perceptron Network

9.7 Example : *aja/example/ann/perceptron_network.cpp*

In our C++ implementation of this network, we have the following classes: we have separate classes for input neurons and output neurons. The **input_neuron** class is for the input neurons. This class has weight and activation as data members. The **output_neuron** class is similar and is for the output neuron. It is declared as a friend class in the **input_neuron** class. The output neuron class has also a data member called output. There is a network class, which is a friend class in the **output_neuron** class. An instance of the network class is created with four input neurons. These four neurons are all connected with one output neuron.

9.7.0.1 input_neuron class

The member functions of the input_neuron class are:
• A default constructor,
• A second constructor that takes a real number as an argument, and
• A function that calculates the output of the input neuron.

The constructor taking one argument uses that argument to set the value of the weight on the connection between the input neuron and the output neuron. The functions that determine the neuron activations and the network output are declared public. The activations of the neurons are calculated with functions defined in the neuron classes. A threshold value is used by a member function of the output neuron to determine if the neurons activation is large enough for it to fire, giving an output of 1.

9.7.0.2 Implementation of Functions

The network is designed to have four neurons in the input layer. Each of them is an object of class input_neuron, and these are member classes in the class network. There is one explicitly defined output neuron of the class out_neuron. The network constructor also invokes the neuron constructor for each input layer neuron in the network by providing it with the initial weight for its connection to the neuron in the output layer. The constructor for the output neuron is also invoked by the network constructor, at the same time initializing the output and activation data members of the output neuron each to zero. To make sure there is access to needed information and functions, the output neuron is declared a friend class in the class input_neuron. The network is declared as a friend class in the class out_neuron.

9.7.0.3 Input/Output

There are two data files used in this program. One is for setting up the weights, and the other for setting up the input vectors. On the command line, you enter the program name followed by the weight file name and the input file name. You can find a file called weight.dat, which contains the following data:
2.0 3.0 3.0 2.0
3.0 0.0 6.0 2.0
These are two weight vectors.
You can find a file called input.dat with the two data vectors below:
1.95 0.27 0.69 1.25
0.30 1.05 0.75 0.19

9.7.1 Execution

Listing 9.1: perceptron_netwotk.cpp output

```
This program is for a perceptron network with
input layer of 4 neurons, each connected to the
output neuron.

This example takes Real number as Input Signals
Please enter the number of weights/vectors : 2
This is vector # 1
Please enter a threshold value for
output neuron, eg 7.0 : 7

Input neuron 1 value is : 1.95 weight is : 2
and its activation is : 3.9
Input neuron 2 value is : 0.27 weight is : 3
```

```
and its activation is : 0.81
Input neuron 3 value is : 0.69 weight is : 3
and its activation is : 2.07
Input neuron 4 value is : 1.25 weight is : 2
and its activation is : 2.5

Output neuron activation is : 9.28

The output neuron activation exceeds the−
threshold value of 7
Output value is 1

This is vector # 2
Please enter a threshold value for
output neuron, eg 7.0 : 7

Input neuron 1 value is : 0.3 weight is : 3
and its activation is : 0.9
Input neuron 2 value is : 1.05 weight is : 0
and its activation is : 0
Input neuron 3 value is : 0.75 weight is : 6
and its activation is : 4.5
Input neuron 4 value is : 0.19 weight is : 2
and its activation is : 0.38

Output neuron activation is : 5.78

The output neuron activation is smaller than the−
threshold value of 7
Output value is 0
```

Chapter 10

XOR Function

The ability of a Perceptron in evaluating functions was brought into question when Minsky and Papert proved that a simple function like XOR (the logical function exclusive or) could not be correctly evaluated by a Perceptron. The XOR logical function, f(A,B), is as follows:

Minsky and Papert showed that it is impossible to come up with the proper set of weights for the neurons in the single layer of a simple Perceptron to evaluate the XOR function. The reason for this is that such a Perceptron, one with a single layer of neurons, requires the function to be evaluated, to be linearly separable by means of the function values. The concept of **linear separability** is explained next. But let us show you first why the simple perceptron fails to compute this function.

Since there are two arguments for the XOR function, there would be two neurons in the input layer, and since the functions value is one number, there would be one output neuron. Therefore, you need two weights w 1 and w 2 ,and a threshold value . Let us now look at the conditions to be satisfied by the ws and the so that the outputs corresponding to given

Table 10.1: XOR Table

A	B	$f(A,b) = XOR(A,B)$
0	0	0
0	1	1
1	0	1
1	1	0

Table 10.2: Conditions on Weights

Input	Activation	Output	Needed Condition
0, 0	0	0	0 ¡
0, 1	w_1	1	w_1 ¿
1, 0	w_2	1	w_2 ¡
1, 1	$w_1 + w_2$	0	$w_1 + w_2$ ¡

inputs would be as for the XOR function.First the output should be 0 if inputs are 0 and 0. The activation works out as 0. To get an output of 0, you need 0 ¡ . This is your first condition. 10.2 shows this and two other conditions you need, and why.

From the first three conditions, you can deduce that the sum of the two weights has to be greater than , which has to be positive itself. Line 4 is inconsistent with lines 1, 2, and 3, since line 4 requires the sum of the two weights to be less than, This affirms the contention that it is not possible to compute the XOR function with a simple perceptron.

Geometrically, the reason for this failure is that the inputs (0, 1) and (1, 0) with which you want output 1, are situated diagonally opposite each other, when plotted as points in the plane, as shown below in a diagram of the output (1=T,0=F):

F	T
T	F

You cant separate the Ts and the Fs with a straight line. This means that you cannot draw a line in the plane in such a way that neither (1, 1) -¿F nor (0,0)-¿F is on the same side of the line as (0, 1) -¿T and (1, 0)-¿ T.

10.1 Linear Separability

What linearly separable means is, that a type of a linear barrier or a separator **a line in the plane**, or **a plane in the three-dimensional space**, or **a hyperplane in higher dimensions** should exist, so that the set of inputs that give rise to one value for the function all lie on one side of this barrier, while on the other side lie the inputs that do not yield that value for the function. A hyperplane is a surface in a higher dimension, but with

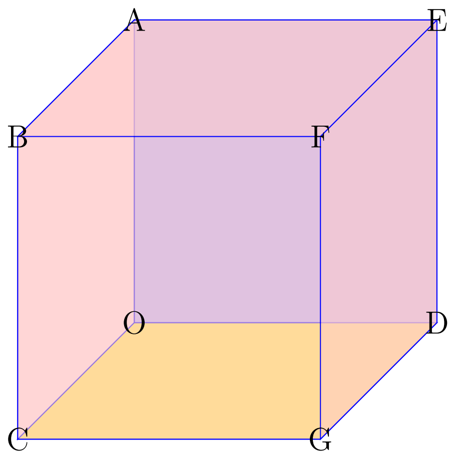


Figure 10.1: Separating plane

a linear equation defining it much the same way a line in the plane and a plane in the three-dimensional space are defined.

To make the concept a little bit clearer, consider a problem that is similar but, let us emphasize, not the same as the XOR problem.

Imagine a cube of 1-unit length for each of its edges and lying in the positive octant in a xyz-rectangular coordinate system with one corner at the origin. The other corners or vertices are at points with coordinates (0, 0, 1), (0, 1, 0),(0, 1, 1), (1, 0, 0), (1, 0, 1), (1, 1, 0), and (1, 1, 1). Call the origin O, and the seven points listed as C, A, B, D, G, E, and F, respectively. Then any two faces opposite to each other are linearly separable because you can define theseparating plane as the plane halfway between these two faces and also parallel to these two faces. For example, consider the faces defined by the set of points O, A, B, and C and by the set of points D, E, F, and G. They are parallel and 1 unit apart, as you can see in Figure 5.1. The separating plane for these two faces can be seen to be one of many possible planesany plane in between them and parallel to them. One example, for simplicity, is the plane that passes through the points (1/2, 0, 0), (1/2, 0, 1), (1/2, 1, 0), and (1/2, 1, 1). Of course, you need only specify three of those four points because a plane is uniquely determined by three points that are not all on the same line. So if the first set of points corresponds to a value of say, +1 for the function, and the second set to a value of 1, then a single-layer Perceptron can determine, through some training algorithm, the correct weights for the connections, even if you start with the weights being initially all 0

Consider the set of points O, A, F, and G. This set of points cannot be linearly separated from the other vertices of the cube. In this case, it would be impossible for the single-layer Perceptron to determine the proper weights for the neurons in evaluating the type of function we have been discussing.

Chapter 11

References

Without following links Ctrl + c and Ctrl + v would have not happened!

http://en.wikipedia.org/wiki/Perceptron

Chapter 12

Pros and Cons of Artificial Neural Network

12.1 Benefits