UNIT I

1

APPLIED MATH AND MACHINE LEARNING BASICS

Unit Structure

1.0 OBJECTIVES

This chapter will help to you understand the:

- Concepts of basic Mathematics useful in Machine learning and deep Learning
- Basic concepts related scalar, vectors, matrix and tensor
- Different types of matrix and its operations
- Decomposition of matrix

1.1 INTRODUCTION AND OVERVIEW OF APPLIED MATH AND MACHINE LEARNING BASICS

This section of the book tells some basic mathematical concepts which helps to understand the deep Learning. Deep learning is a subdomain of machine learning which is concerned with algorithms, mathematical functions and artificial neural network.

1.2 LINEAR ALGEBRA

Linear Algebra is one of the widely used branches of mathematics related with mathematical structures which is continuous rather than discrete mathematics. It includes operations like addition, scalar multiplication that helps to understand concepts like linear transformations, vector spaces, linear equations, matrices and determinants. A good knowledge of Linear Algebra is important to understand and working with many essential machine learning algorithms, especially algorithms related with deep learning.

1.2.1 Scalars, Vectors, Matrices and Tensors:

Let's start with some basic definitions:

Scalar	Vector	Matrix	Tensor
1	$\begin{bmatrix} 2 \\ 3 \end{bmatrix}$	$\begin{bmatrix} 1 & 2 \\ 4 & 5 \end{bmatrix}$	$\begin{bmatrix} [2\ 3] & [8\ 2] \\ [3\ 6] & [2\ 7] \end{bmatrix}$

(1.1)

Understanding of a scalar, a vector, a matrix, a tensor

Scalar: A scalar is a single number represent 0^{th} order tensor. Scalars are written in lowercase and italics. For Example: *n.* there is many different sets of numbers with interest in deep learning. The notation $x \in \mathbb{R}$ represents x is a scalar belonging to a real values numbers i.e. $\mathbb{R}$. $\mathbb{N}$ states the set of positive integers (1, 2, 3, …). $\mathbb{Z}$ states the integers, which is combination of positive, negative and zero values. Rational numbers are representing by notation $\mathbb{Q}$

Python code to explains arithmetic operations on Scalars:

```
Code: # In-Built Scalars
a = 15
b = 3
print(type(a))
print(type(b))
print(a + b)
print(a - b)
print(a * b)
print(a / b)
Output:
<class 'int'>
<class 'int'>
15
```

```
12
45
5
```

```
# Python code snippet checks if the given variable is scalar or not.
import numpy as np
# Is Scalar Function
def isscalar(num):
    if isinstance(num, generic):
        return True
    else:
        return False
print(np.isscalar(2.4))
print(np.isscalar([4.1]))
print(np.isscalar(False))
Output:
True
False
True
```

Vector: A vector is an ordered array of single numbers represent 1st order tensor. Vectors should be written in lowercase, bold, and italics. For Example: ***x.***

$$\boldsymbol{x} = [\boldsymbol{x}_1\, \boldsymbol{x}_2\, \boldsymbol{x}_3 \ldots \boldsymbol{x}_n] \quad \textbf{or} \quad \boldsymbol{x} = \begin{bmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \\ \vdots \\ \boldsymbol{x}_n \end{bmatrix} \tag{1.2}$$

Here, first element of $\boldsymbol{x}$ is $\boldsymbol{x}_1$, the next second element is $\boldsymbol{x}_2$ and so on element gets listed. To understand the necessary element of a vector index wise, the scalar element of a vector positioned i^{th} is written as ***x[i].*** Suppose S={1,2,4} then $\boldsymbol{x}_1$=1, $\boldsymbol{x}_2$=2, $\boldsymbol{x}_4$=4. The – sign to index is used to indicate the complement of a S, like for example $\boldsymbol{x}_{-1}$ is the vector consisting of all elements of $\boldsymbol{x}$ except $\boldsymbol{x}_1$, and $\boldsymbol{x}_{-s}$ is the vector consist of all elements of $\boldsymbol{x}$ except $\boldsymbol{x}_1$, $\boldsymbol{x}_2$ and $\boldsymbol{x}_4$. Vectors are pieces of objects known as vector spaces, which can be considered as collection of all possible vectors of a particular dimension.

```
#Python code demonstrating Vectors
import numpy as np
# Declaring Vectors
x = [2, 4, 6]
y = [3, 5, 1]
print(type(x))
# Vector addition using Numpy
```

```
z = np.add(x, y)
print(z)
print(type(z))
Output:
<class 'list'>
[2, 4, 6, 3, 5, 1]
[5 9 7]
<class 'numpy.ndarray'>
```

Matrix: Matrix is a 2-D rectangular array consisting of numbers represents 2nd order tensor. Matrices should be written in uppercase, bold and italics. For Example: ***X.*** If p and q are positive integers, that is p, q ∈ ℕ then the p×q matrix contains p*q numbers, with p rows and q columns

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{a_{11}} & \boldsymbol{a_{12}} & \cdots & \boldsymbol{a_{1q}} \\ \boldsymbol{a_{21}} & \boldsymbol{a_{22}} & \cdots & \boldsymbol{a_{2q}} \\ \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{a_{p1}} & \boldsymbol{a_{p2}} & \cdots & \boldsymbol{a_{pq}} \end{bmatrix} \quad (1.3)$$

Full matrix component can be express as follows:

$$\boldsymbol{A} = [\boldsymbol{a_{ij}}]_{p\times q} \quad (1.4)$$

Some of the operations of matrices are as follows:

- **Matrix Addition:**

We can do addition of Matrices to scalars, vectors and other matrices. These precise techniques are often used in machine learning and deep learning.

```
# Python code for Matrix Addition

import numpy as np

x = np.matrix([[5, 3], [2, 6]])

sum = x.sum()

print(sum)
```

Output: 16

- **Matrix-Matrix Addition:**

Z=X+Y

Here, when shapes of two matrices are equal addition is possible otherwise not.

```
# Python code for Matrix-Matrix Addition
import numpy as np
p = np.matrix([[1, 1], [2, 2]])
q = np.matrix([[3, 3], [4, 4]])
m_sum = np.add(p, q)
print(m_sum)
```

Output :

```
[[ 4  4]
[ 6  6]]
```

Matrix-Scalar Addition:

Here, addition of given scalar with all elements of matrix takes place.

```
# Python code for  Matrix-Scalar Addition
import numpy as np
x = np.matrix([[1, 1], [3, 3]])
s_sum = x + 1
print(s_sum)
```

Output:

```
[[2 2]
 [4 4]]
```

- **Matrix Scalar Multiplication:**

Here, Multiplication of scalar element with matrix element takes place

```
# Python code for  Matrix Scalar Multiplication
import numpy as np
x = np.matrix([[2, 3], [3, 4]])
s_mul = x * 3
```

```
print(s_mul)
Output:
[[ 6  9]
[9  12]]
```

- **Matrix Transpose:**

With transpose operation horizontal row vector can be converted into vertical column vector and vice versa.

$$\mathbf{M} = \begin{bmatrix} a_{11} & a_{12} & \Lambda & a_{1c} \\ a_{21} & a_{22} & \Lambda & a_{2c} \\ \text{M} & \text{M} & & \text{M} \\ a_{r1} & a_{r2} & \Lambda & a_{rc} \end{bmatrix} \quad (1.5)$$

$$\mathbf{M}^T = \begin{bmatrix} a_{11} & a_{21} & \Lambda & a_{r1} \\ a_{12} & a_{22} & \Lambda & a_{r2} \\ \text{M} & \text{M} & & \text{M} \\ a_{1c} & a_{2c} & \Lambda & a_{rc} \end{bmatrix}, \text{i.e., } m_{ji}^T = m_{ij} \quad (1.6)$$

$$\text{examples : } \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}, \quad [x \quad y \quad z]^T = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$A=[a_{ij}]_{p\times q}$ (1.7)

$A^T = [a_{ji}]_{q\times p}$ (1.8)

```
# Python code for Matrix Transpose
import numpy as np
a = np.array([[2, 3], [5, 7]])
print(a)
a.transpose()
print(a)
Output:
[[2 3]
 [5 7]]
array([[2, 5],
       [3, 7]])
```

Tensors: Sometimes we need an array with more than 2 axis; a tensor is an array of numbers arranged on a grid. It wraps scalar, vector and matrix. We can represent tensor name "A" as **A.** we can access element of A at coordinates (i, j, k) by writing $A_{i,j,k}$

1.2.2 Multiplying Matrices and Vectors:

When A is a m × n matrix & B is a k × p matrix, AB is only possible if n=k. The result will be an m×p matrix.

Simple to know we can perform A*B IF: Number of columns in A = Number of rows in B

$$\mathbf{A*B} = \begin{bmatrix} 2 & 3 \\ -1 & -4 \\ 0 & 5 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 2\times1+3\times3 & 2\times2+3\times4 \\ -1\times1-4\times3 & -1\times2-4\times4 \\ 0\times1+5\times3 & 0\times2+5\times4 \end{bmatrix} = \begin{bmatrix} 11 & 16 \\ -13 & -16 \\ 15 & 20 \end{bmatrix}$$

Another example:

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

$$\mathbf{AB} = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} = \begin{bmatrix} 1\times4+2\times5+3\times6 \end{bmatrix} = \begin{bmatrix} 32 \end{bmatrix}$$

$$\mathbf{BA} = \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} = \begin{bmatrix} 4\times1 & 4\times2 & 4\times3 \\ 5\times1 & 5\times2 & 5\times3 \\ 6\times1 & 6\times2 & 6\times3 \end{bmatrix} = \begin{bmatrix} 4 & 8 & 12 \\ 5 & 10 & 15 \\ 6 & 12 & 18 \end{bmatrix}$$

The dot product operation is defined as:

$$C_{i,j} = A_{i\times k}\, B_{k\times j} = \sum_k A_{i\times k}\, B_{k\times j} \qquad (1.9)$$

Need to understand product of two matrices is not just a matrix having individual elements in that some operation are exist that called element wise product or hadmard product which is denoted as A ⊙ B.

Properties of the dot product:

- Simplification of the matrix product

- $(AB)^T = B^T A^T$ (1.10)

- Matrix multiplication is NOT commutative means order of multiplication is important

- $AB \neq BA$ (1.11)

- Matrix multiplication IS associative

- $A(BC)=(AB)C$ (1.12)

- Matrix multiplication IS distributive

- $A(B+C)=AB+AC$ (1.13)

- $(A+B)C=AC+BC$ (1.14)

1.2.3 Identity and Inverse Matrices:

Identity Matrix: An identity matrix is a square matrix where value of diagonal elements is one and rest of matrix elements values are equal to zero. The product is the matrix **A** and identity matrix is matrix **A** itself.

Another name for Identity Matrix is Unit Matrix or Elementary Matrix. Identity Matrix is denoted with the letter $\mathbf{I_{n\times n}}$ n, where n ×n represents the order of the matrix.

IA = A and AI = A (1.15)

AI = IA = A (1.16)

One of the important properties of identity matrix is: $\mathbf{AI_{n\times n}} = A$, where A is any square matrix of order n ×n

$I_1 = [1]$, $I_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, $I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

Square matrix: A matrix with the same number of rows and columns is called a square matrix.

Example: $\begin{bmatrix} 1 & 2 & 3 \\ 6 & 5 & 4 \\ 7 & 8 & 9 \end{bmatrix}_{3\times 3}$

Note: An identity matrix is a perpetually square matrix.

Inverse Matrix: Let '**A**' be any square matrix. An inverse matrix of '**A**' is denoted by '$\mathbf{A^{-1}}$' and is such a matrix that $AA^{-1}=A^{-1}A=I_n$. if we get $\mathbf{A^{-1}}$ of Matrix '**A**' then it is known as invertible. Non-square matrices do not have inverses. It is not necessary all square matrices have inverses. If a square matrix has an inverse then it is known as invertible or non-singular

Example

$\mathbf{A}=\begin{bmatrix}4 & 3\\ 3 & 2\end{bmatrix}$ and its inverse $\mathbf{A}^{-1}=\begin{bmatrix}-2 & 3\\ 3 & -4\end{bmatrix}$

$\mathbf{A}\,\mathbf{A}^{-1}=\begin{bmatrix}4 & 3\\ 3 & 2\end{bmatrix}\begin{bmatrix}-2 & 3\\ 3 & -4\end{bmatrix}=\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}$

$\mathbf{A}^{-1}\,\mathbf{A}=\begin{bmatrix}-2 & 3\\ 3 & -4\end{bmatrix}\begin{bmatrix}4 & 3\\ 3 & 2\end{bmatrix}=\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}$

Here, we can say that $\begin{bmatrix}4 & 3\\ 3 & 2\end{bmatrix}$ and $\begin{bmatrix}-2 & 3\\ 3 & -4\end{bmatrix}$ are inverses of each other.

1.2.4 Linear Dependence and Span:

Suppose we have ***Ax = b*,** to get $\mathbf{A}^{-1}$ here has exactly one solution for every value of ***b***. There is a possibility, to have no solution or infinitely many solutions for the systems of values ***b***. To understand and analyse what kind of different solutions the equation has we can consider columns of ***A*** which specifies different direction can travel from origin and determine number of ways reaching towards ***b*.**

A **linear combination** of a list $(\overrightarrow{v_1},\ \overrightarrow{v_2}+\ \ldots+\vec{v}_m)$ is some set of vector of the form which is given by multiplying each vector $v^{(i)}$ by respective scalar coefficient and adding the results.

$$\sum_i c_i v^{(i)} \qquad (1.17)$$

Given a set of vectors $\{\overrightarrow{v_1},\ \overrightarrow{v_2}+\ \ldots+\vec{v}_m\}$ in a vector space V, any vector of the form

v = $a_1\overrightarrow{v_1}+a_2\overrightarrow{v_2}+\ \ldots+a_m\,\overrightarrow{v_m}$ for some scalars $a_1, a_2, \ldots, a_k$ is called a **linear combination** of $v_1, v_1, \ldots, v_k$.

The set of all points obtainable by linear combination of the original vectors is called **Span**. In other words **Span** $((a_1\overrightarrow{v_1}+a_2\overrightarrow{v_2}+\ \ldots+a_m \in \mathbb{R})$

Now determining ***Ax = b*** need to test whether ***b*** is in the span of columns of ***A*,** this particular span is known as column space or the range of ***A*.**

Linear Independence: Given a set of vectors $\{\overrightarrow{v_1},\ \overrightarrow{v_2}+\ \ldots+\vec{v}_m\}$ in a vector space V, they are said to be linearly independent if the equation $c_1\overrightarrow{v_1}+c_2\overrightarrow{v_2}+\ \ldots+a_m\,\overrightarrow{v_m}=0$ has only the trivial solution.
If $(\overrightarrow{v_1},\ \overrightarrow{v_2}+\ \ldots+\vec{v}_m)$ are not linearly independent they are linearly dependent.

1.2.5 Norms:

Norm is a function that in return measures length/size of any vector (excluding zero vectors). In machine learning multiple times we measure the size of vectors by using norms. The norms are clubbed under p-norms or (l^p-norms) family, where p is arbitrary number greater than or equal to 1.

The p-norm of vector x can be presented as

$$||x||_p = (x_1^p + x_2^p + x_3^p + ... + x_n^p)^{1/p} \tag{1.17}$$

Every element of vector x is has the power p. Then their sum is raised to the power $(1/p)$

Simply represented as,

$$||x||_p = (\sum_{i=1}^{n} x_i^p)^{1/p} \tag{1.18}$$

Any norm function *f* should be satisfied following conditions:

1. If norm of x is greater than Zero then x is not equal to 0 (Zero Vector) and if norm is equal to Zero then x is a zero vector.

$$\text{If } f(x) > 0 \text{ then } x \neq 0 \tag{1.19}$$

$$\text{If } f(x) = 0 \text{ then } x = 0 \tag{1.20}$$

2. For any scalar quantity, say K

$$f(Kx) = K\, f(x) \tag{1.21}$$

3. Suppose we have another vector y

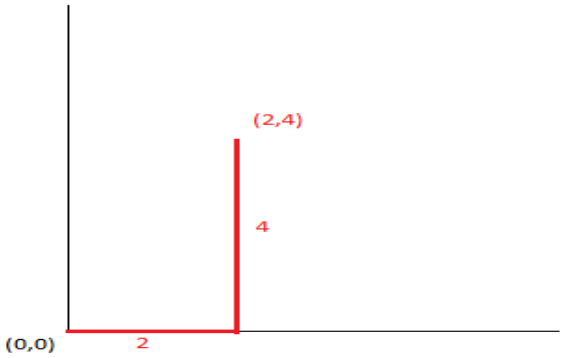
$$f(x + y) \leq f(x) + f(y) \tag{1.22}$$

If above three properties are satisfied then function f is norm.

- **Manhattan Distance (1-norm):**

Manhattan distance also called 1 norm it measure the distance between two points that can travel along orthogonal blocks.

Example:
$\vec{a}$ = [2,4]



- **Infinity-norm:**

The infinity-norm also called max-norm which returns absolute value in the given vector.

Suppose, we want to find infinity-norm of another vector, like ***a***

$\vec{a}$ = [2,4,-5]

$||a||_\infty = 5$

- **Euclidean Norm (2-norm):**

One of the most used Norms is 2-norm, used to calculate magnitude of vector.

$\left||a|\right|_2 = (2^2 + 4^2)^{1/2}$

$||a||_2 = (4 + 16)^{1/2}$

$||a||_2 = \sqrt{20}$

1.2.6 Special Kinds of Matrices and Vectors:

Some of the matrices:

- Diagonal Matrix: Diagonal matrix is a matrix Denoted by **D** where diagonal element values are nonzero and other entries of matrix are zero.

Example:

$$\mathbf{D} = \begin{bmatrix} \mathbf{2} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{4} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{6} \end{bmatrix}$$

As a Vector it will be represented as,

d= (d11, d22, d33)

With scalar values it can be represented as follows:

d = (1, 2, 3)

```
#Python code for  diagonal matrix
from numpy import array
from numpy import diag
A = array([[1, 2, 3], [1, 2, 3], [1, 2, 3]])
print(A)
# Print diagonal vector
d = diag(A)
print(d)
# create diagonal matrix from vector
D = diag(d)
print(D)
```

Output:

```
[[1 2 3]
 [1 2 3]
 [1 2 3]]

[1 2 3]

[[1 0 0]
 [0 2 0]
 [0 0 3]]
```

It is not compulsory diagonal matrix is square, there is possible to construct rectangular diagonal matrix as well. We can use Diagonal matrix in machine learning to obtain less expensive algorithm.

- **Triangular Matrix:** A triangular matrix is a type of square matrix where values are filled in the upper-right or lower-left of the matrix with the remaining elements of the matrix are filled with zero values. A triangular matrix with filled some values which lie above the main diagonal is called an upper triangular matrix. Whereas, a triangular matrix with filled values which lie below the main diagonal is called a lower triangular matrix.

Example of a 3×3 upper triangular matrix

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

Example of a 3×3 lower triangular matrix

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

```
# Python code for triangular Matrices

from numpy import array
from numpy import tril
from numpy import triu

A = array([[1, 2, 3], [1, 2, 3], [1, 2, 3]])

print(A)
lower = tril(A)
print(lower)
upper = triu(A)
print(upper)
```

Output:

```
[[1 2 3]
 [1 2 3]
 [1 2 3]]

[[1 0 0]
 [1 2 0]
 [1 2 3]]

[[1 2 3]
 [0 2 3]
 [0 0 3]]
```

- **Symmetric Matrix:** Symmetric matrix is a matrix where top-right triangle is the same as the bottom-left triangle.

Example:

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

Transpose of Symmetric Matrix is equal to original Symmetric Matrix.

$$\mathbf{A}= \mathbf{A}^{\mathbf{T}} \qquad (1.23)$$

- **Orthogonal Matrix:** Orthogonal matrix is a matrix if a dot product of two vectors is equals to zero, and then is called Orthogonal. It is a type of square matrix whose columns and rows are orthonormal unit vectors, e.g. it is perpendicular and have a length or magnitude of One. Here rows are mutually orthonormal and columns are mutually orthonormal.

$$\mathbf{A}^{\mathbf{T}}\mathbf{A} = \mathbf{A}\mathbf{A}^{\mathbf{T}} = \mathbf{I} \qquad (1.24)$$

```
# python code for orthogonal atrix
from numpy import array
from numpy.linalg
import inv
Q = array([[1, 0], [0, -1]])
print(Q)
# inverse equivalence
V = inv(Q)
print(Q.T)
print(V)
# identity equivalence
I = Q.dot(Q.T)
print(I)
```

Output:

```
[[ 1  0]
 [ 0 -1]]

[[ 1  0]
 [ 0 -1]]

[[ 1.  0.]
 [-0. -1.]]

[[1 0]
 [0 1]]
```


1.2.7 Eigende composition:

We usually used to see multiple mathematical structures which are complex to understand, so need to break it with some simplified form and understand useful properties for the same. We can decomposed integers with some prime factors, same way we can decompose matrix.

To make complex operations into simple structure matrix decompositions are very useful. Matrix decomposition tools helps for reducing a matrix to their constituent parts.

Eigende composition of a matrix is a widely used matrix decomposition which involves decomposition of a square matrix into a set of eigenvectors and eigenvalues. This kind of decomposition also helps in machine learning like **Principal Component Analysis method**.

A vector is an eigenvector of a matrix if it fulfils the following equation.

$\mathbf{A}\boldsymbol{v} = \boldsymbol{\lambda}$ (λ=lambda) (1.25)

The equation is called the eigenvalue equation, where **A** is the parent square matrix that we decompose, v is the non-zero eigenvector of the matrix, and lambda represents the eigenvalue scalar. We can also find left eigenvector like $\boldsymbol{v^T A = \lambda v^T}$, but we normally use right eigenvector.

It is possibility matrix can have one eigenvector and eigenvalue for each aspect of the parent matrix. Not necessary all square matrices can be decomposed into eigenvectors and eigenvalues, some may be decomposed in a way that requires complex numbers.

The parent matrix would be presented as to be a product of the eigenvectors and eigenvalues

$$\boldsymbol{A = V\, diag(\lambda) V^{-1}} \qquad (1.26)$$

Here, $\boldsymbol{V}$ is a matrix comprised of the eigenvectors, diag($\boldsymbol{\lambda}$) is a diagonal matrix comprised of the eigenvalues along the diagonal , $\boldsymbol{V^{-1}}$ is the inverse of the matrix comprised of the eigenvectors.

Eigen is not a name it is pronounced as "eye-gan" is a German word that means "own" as in belonging to the parent matrix.

Decomposition does not mean compression of the matrix perhaps it breaks it down into constituent parts to make certain operations on the matrix easier to perform.

The eigendecomposition of a matrix gives many useful facts about the matrix. If the eigen values are zero then the matrix is singular.

Eigenvectors and Eigenvalues: Eigenvectors are unit vectors that mean their length or magnitude is equal to one. They are often known as right vectors, which simply mean a column vector (as opposed to a row vector or a left vector).

A matrix contains only positive eigenvalues is known as a **positive definite matrix**, and if it contains all negative eigenvalues, it is known as a negative definite matrix.

Eigendecomposition Calculation:

```
# Python code for eigendecomposition
from numpy import array
from numpy.linalg import eig

# Define matrix
A = array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(A)
# Calculate eigendecomposition
val, vect = eig(A)
```

```
print(val)
print(vect)

Output:
[[1 2 3]
 [4 5 6]
 [7 8 9]]

[  1.61168440e+01  -1.11684397e+00  -9.75918483e-16]

[[-0.23197069 -0.78583024  0.40824829]
 [-0.52532209 -0.08675134 -0.81649658]
 [-0.8186735   0.61232756  0.40824829]]
```

1.3 SUMMARY

This chapter includes basic concepts and differentiation related to scalar, vector, matrix and tensor which is mostly used in machine learning as well as in deep learning. This chapter explain Linear Algebra concepts broadly. Different types of Matrix, Operation explain with python programs.

Specifically learned:

- Basic difference between scalar, vector, matrix and tensor.
- Matrix types, different operations related with matrices in Python with NumPy.
- Eigendecomposition and the role of eigenvectors and eigenvalues.

UNIT END EXERCISES

1. Differentiate between scalar and vector
2. Explain Arithmetic operations on scalar with example.
3. What is Matrix and explain matrices addition
4. Explain Matrix transpose with an example.
5. Write Properties of dot products.
6. Explain Inverse matrix with an example
7. Write a short note on Norms.
8. Explain Eigenvectors and Eigenvalues.
9. Compare Symmetric Matrix and Orthogonal Matrix
10. Write a note on linear combination

BIBLIOGRAPHY

1. **Ian Goodfellow, Yoshua Bengio, Aaron Courvile** "Deep Learning",1st edition 2016

2. https://hadrienj.github.io/posts/Deep-Learning-Book-Series-2.1-Scalars-Vectors-Matrices-and-Tensors/
3. https://hadrienj.github.io/posts/Deep-Learning-Book-Series-2.2-Multiplying-Matrices-and-Vectors/
4. https://medium.com/linear-algebra/part-18-norms-30a8b3739bb

*****

2

NUMERICAL COMPUTATION

Unit Structure

2.0 OBJECTIVES

After going through this unit, you will be able to:

- Define Overflow and Underflow techniques
- Clear Concepts like Poor Conditioning, Gradient based optimization
- Different ways of Constraint Optimization

2.1 INTRODUCTION TO NUMERICAL COMPUTATION

A lot amount of computation is needed in machine learning algorithms to solve complex and vital problems. Here need to follow iterative process where mathematical methods used to get best solutions. Numerical Computation needed large number of arithmetic calculation hence required efficient and fast computational devices. The approach is to formulate mathematical model and solve problems with arithmetic calculations..

2.2 OVERFLOW AND UNDERFLOW

Difficulties which are faced going from mathematics to computers are representing infinitely many real number on less or finite memory. This means that every calculations experience some approximation error. One of them is rounding error, where undergoes many operations, but algorithms which in work get failed due to not designing of minimum accumulation to the rounding error. In computers numbers are stored as discrete digits, so when we used to make arithmetic calculations which result in extra digits that result we cannot append into main result and face overflow or underflow error.

Underflow: Underflow is a situation it happens in a computer or similar devices when a result of mathematical operation is smaller than the device is capable of storing. It is kind of trickier to recognize because it has to work with precision in floating points. Here, cause of discrete nature of storage capacity in computers, we cannot store any small number either. The floating-point format comes up with some techniques to represent fractional numbers. When we implement these in calculations that result in a smaller number than our desire least value. This happens when numbers close to 0 are rounded to 0.

Like, suppose we want to perform a calculation, 0.005×0.005. The answer to this is 0.000025, but we don't have this many decimal places available. Here, we discard the least-significant bits and store 0.000, which is absolute an erroneous answer. Underflow is representational error and occurs mostly when dealing with decimal arithmetic. It also happens when two negative numbers are added and the result is out of range for the device to store

Overflow: Overflow shows that we have done a calculation, the result of that is so larger than we can hold and represent. Like, the processor is running an operation as an increment but the operand having same capacity, it cannot hold the result. The whole issue is occurs cause of the memory size in computers. In mathematics, numbers are absolute and there's no concept of precision. But in case of computers, due to hardware limitations and especially in memory size, real numbers are rounded to the nearest value.

Suppose we have an integer stored in 1 byte. We can store greatest number that is 255 in one byte, i.e. 11111111. Now, Let's add 2 to it to get 00000010. The result is 257, which is 100000001. The result has 9 bits, whereas the integers we are working with consist of only 8.

One of the function that must be stabilized against underflow and overflow name is the softmax function. The SoftMax Function is used to predict the probabilities associated with a Multinoulli Distribution

$$softmax(\vec{x})i = \frac{exp(x_i)}{\sum 1^n exp(x_j)} \tag{2.1}$$

If $\forall i, \vec{x}_i = c$, then we can show that $softmax(\vec{x}) = \frac{1}{n}$. But if we want to calculate the same value in computers numerically, depending on the value of c you could get different results.

If c is a big positive number then e^c will be overflow and the softmax will be undefined and return NaN. And if c is a big negative number e^c will be a underflow and softmax will become zero, resulting in "division by zero". The solution to both of these problems is a technique called *stabilization*.

Unfortunately, stabilization is not a complete solution. It is used to solve mathematical problem analytically. As for the softmax given above, let's define $\vec{z}$ as follow:

$$\vec{z} = \vec{x} - max_i(x_i) \tag{2.2}$$

$\vec{z}$ is a new vector where all the elements of $\vec{x}$ are subtracted by its biggest element. Doing so will result in a vector where at least one of its elements is zero ($\vec{x}$'s biggest element). While analytically you can show that $\forall i, softmax(\vec{x})_i = softmax(\vec{z})_i$, the problem of overflow is solved as well. At the same time, the problem of underflow and division by zero is fixed too. That's because at least one element of the denominator is one (the biggest element which is now zero). So there's no way division by zero could happen now. This process is called stabilization.

2.3 POOR CONDITIONING

When function changes with respect to small changes in its inputs it's called Conditioning. Functions that changes rapidly when their inputs are unsettle slightly can be problematic for scientific computation because rounding errors in the inputs can result in big changes in the output.

Consider the function on vector x:

$$f(\vec{x}) = A^{-1}\vec{x} \tag{2.3}$$

Assuming:

$$A \in \mathbb{R}^{nxn} \tag{2.4}$$

Has eigenvalue decomposition, its condition number is:

$$max_{i,j} = |\frac{\lambda_i}{\lambda_j}| \tag{2.5}$$

This ratio is the magnitude of the largest and smallest eigenvalues. When this number is big, matrix inversion is particularly sensitive to error in the input.

2.4 GRADIENT BASED OPTIMIZATION

Most Machine learning, deep learning algorithms involve optimization, Optimization presents to the task of either minimizing or maximizing some function f(x) by altering x. The function which needs to optimize is the Objective Function, sometimes called the **Criterion**.

We denote the value that minimizes or maximizes a function with a superscript *.

For a single variable function:

$$f(x) \Rightarrow f'(x) \Leftrightarrow \frac{d}{dx} f(x) \tag{2.6}$$

For a multi variable function:

$$f(\vec{x}) \Rightarrow \frac{\partial}{\partial x} f(\vec{x}) \Leftrightarrow \nabla_x f(\vec{x}) \tag{2.7}$$

• Minimize/maximize a function f (x) by altering x
– Usually stated a minimization
– Maximization accomplished by minimizing –f(x)
• f (x) referred to as objective function or criterion
– In minimization also referred to as loss function cost, or error
– Example is linear least squares $f(x) = \frac{1}{2} || Ax - b ||^2$
– Denote optimum value by x*=arg min f (x)

Calculus in Optimization:

Suppose function y=f (x), x, y real numbers, Derivative of function denoted: f'(x) or as dy/dx

- Derivative f'(x) gives the slope of f (x) at point x
- It specifies how to scale a small change in input to obtaina corresponding change in the output: f (x + ε) ≈ f (x) + ε f' (x)
  - It tells how you make a small change in input to make a small improvement in y
  - We know that f (x - ε sign (f'(x))) is less than f (x) for small ε. Thus we can reduce f (x) by moving x in small steps with opposite sign of derivative, this technique is called **gradient descent**

Gradient Descent Illustrated:

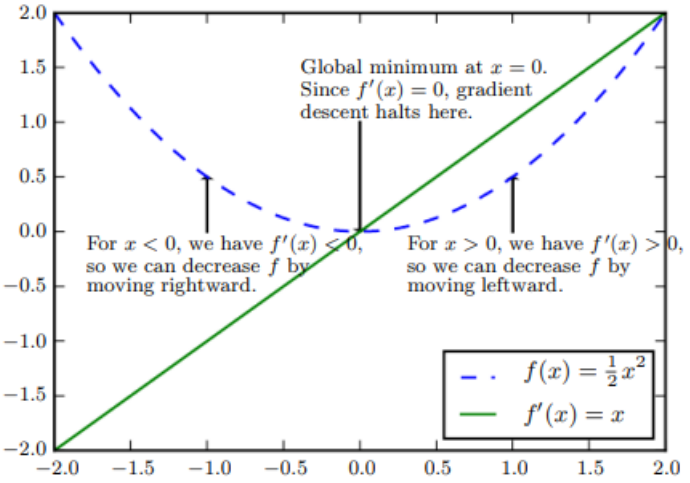


Figure1: An illustration of how the gradient descent algorithm uses the derivatives of a function can be used to follow the function downhill to a minimum

Source: Ian Goodfellow, Yoshua Bengio, Aaron Courvile "Deep Learning",1st edition 2016

For x>0, f(x) increases with x and f'(x)>0

- For x<0, f(x) is decreases with x and f'(x)<0
- Use f'(x) to follow function downhill
- Reduce f(x) by going in direction opposite sign of derivative f'(x)

Stationary points, Local Optima:

Here, When f'(x)=0 derivative provides no information about direction of move.

Points where f'(x)=0 are known as stationary or critical points.

– Local minimum/maximum: a point where f(x) lower/higher than all its neighbours

– Some critical points are neither maxima nor minima. These are known as saddle points

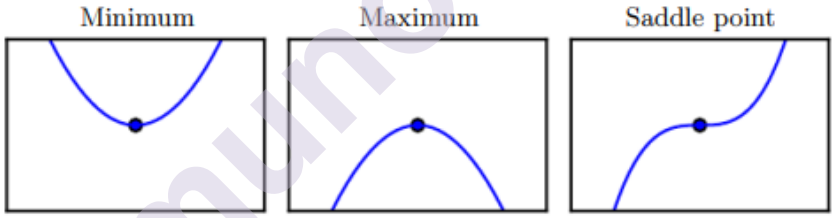


Figure 2: Minimum, Maximum, Saddle Point

Source: Ian Goodfellow, Yoshua Bengio, Aaron Courvile "Deep Learning",1st edition 2016

Optimization algorithms may fail to find global minimum
• Generally accept such solutions

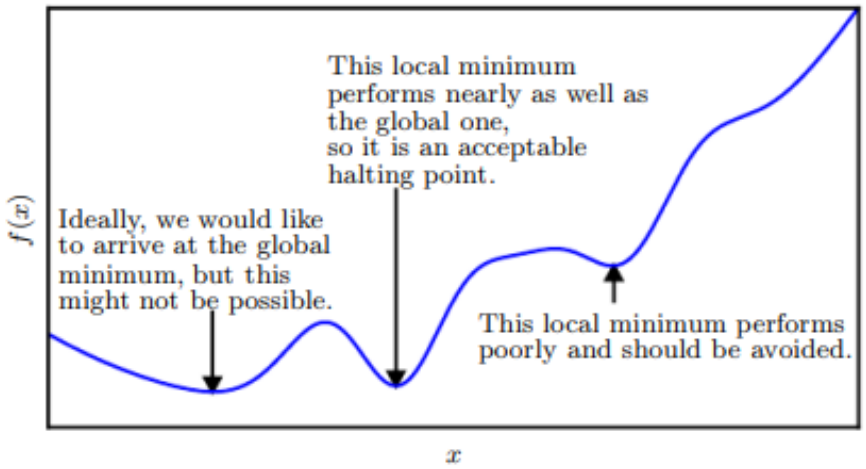


Figure 3: Local minimum

Source: Ian Goodfellow, Yoshua Bengio, Aaron Courvile "Deep Learning",1st edition 2016

We often minimize functions with multiple inputs: $f: R^n \rightarrow R$. For minimization to make sense there must still be only one (scalar) output.

Here , we need partial derivatives $\frac{\partial}{\partial x_i}$ f(x) measures how f changes as only variable x_i increases at point x

- Gradient generalizes notion of derivative where derivative is wrt a vector
- Gradient is vector containing all of the partial derivatives denoted ∇_x f(x)
  - Element i of the gradient is the partial derivative of f wrt x_i
  - Critical points are where every element of the gradient is equal to zero

Directional Derivative:

The directional derivate in direction u, a unit vector, is the slope of the function f in direction u. It's the derivative of the function f(x + au) with respect to a

$$\frac{\partial}{\partial x} f(\vec{x} + \alpha\vec{u}) = \vec{u}^T \nabla_x f(\vec{x}) \tag{2.8}$$

It is used to minimize f, we would like to find the direction in which f decreases the fastest., we can do this using the directional derivative:

$$min_{u,u^T u=1} = \vec{u}^T \nabla_x f(\vec{x}) \tag{2.9}$$

$$= min_{u,u^T u=1} ||\vec{u}||_2 \, ||\nabla_x f(\vec{x})||_2 \, cos(\theta)$$

(2.10)

Where theta is the angle between u and the gradient.

Where ||u||2 = 1 and ignoring all things dependent on u we get: $min_u cos(\theta)$

The gradient points directly uphill, and the negative gradient points directly downhill

- Thus we can decrease f by moving in the direction of the negative gradient
  - This is known as the **method of steepest descent or gradient descent**
- Steepest descent proposes a new point

$$\boldsymbol{x'} = \boldsymbol{x} - \varepsilon \nabla_x f\left(\boldsymbol{x}\right)$$

(2.11)

  - Where ε is the learning rate, a positive scalar. Set to a small constant.

Here, We can choose ε in several different ways like follows

- Popular approach: set ε to a small constant
- Another approach is called line search:
- Evaluate $f(\boldsymbol{x} - \varepsilon \nabla_x f\left(\boldsymbol{x}\right)$ for several values of ε and choose the one that results in smallest objective function value.

Steepest descent converges when every element of the gradient is zero

– In practice, very close to zero

- We may be able to avoid iterative algorithm and jump to the critical point by solving the equation $\nabla_x f\left(\boldsymbol{x}\right) = 0$ for x.

Gradient descent is limited to continuous spaces

- Concept of repeatedly making the best small move can be generalized to discrete spaces
- Ascending an objective function of discrete parameters is called **hill climbing**.

Beyond Gradient: Jacobian and Hessian matrices:

Sometimes we need to find all derivatives of a function whose input and output are both vectors

- If we have function *f:* $R^m \rightarrow R^n$ then the matrix of partial derivatives is known

as the Jacobian matrix J defined as $J_{i,j} = \frac{\partial}{\partial x_j} f(x)_i$

We are also sometimes interested in a derivative of a derivative.

• For a function *f:* $R^n \rightarrow R$ the derivative wrt x_i of the derivative of f wrt x_j

is denoted as $\frac{\partial^2}{\partial x_i \partial x_j} f$

- In a single dimension we can denote $\frac{\partial^2}{\partial x^2} f$ by f''(x)
- Tells us how the first derivative will change as we vary the input
- This important as it tells us whether a gradient step will cause as much of an improvement as based on gradient alone.

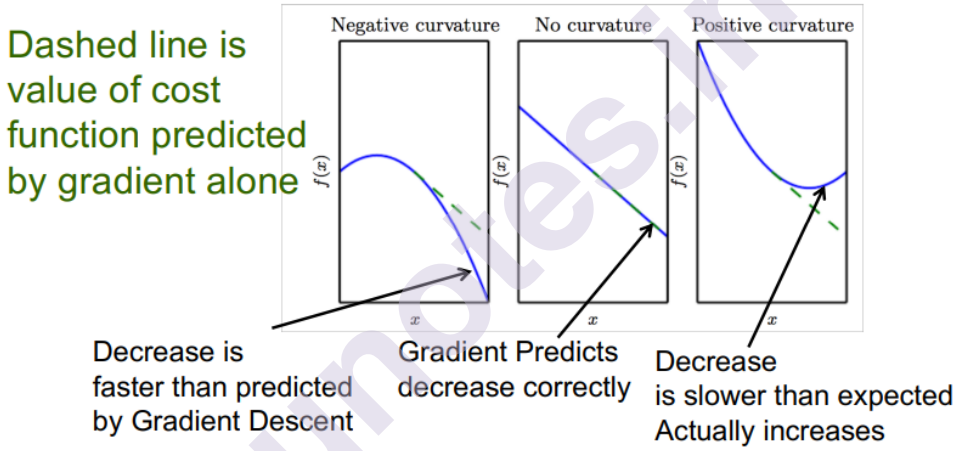


Figure 4: Quadratic functions with different curvatures

Source: https://cedar.buffalo.edu/~srihari/CSE676/

2.5 CONSTRAINT OPTIMIZATION

We may wish to optimize f(x) when the solution x is constrained to lie in set S

– Such values of x are feasible solutions

- Often we want a solution that is small, such as $||x|| \leq 1$
- Simple approach: modify gradient descent taking constraint into account (using Lagrangian formulation)

Least squares with Lagrangian:

We wish to minimize $f(\boldsymbol{x}) = \frac{1}{2} || A\boldsymbol{x} - \mathrm{b} ||^2$

- Subject to constraint $x^T x \leq 1$

- We introduce the Lagrangian $L(\boldsymbol{x}, \lambda) = f(\boldsymbol{x}) + \lambda\left(\boldsymbol{x}^T\boldsymbol{x} - 1\right)$

– And solve the problem $\min_{\boldsymbol{x}} \max_{\lambda, \lambda \geq 0} L(\boldsymbol{x}, \lambda)$

- For the unconstrained problem (no Lagrangian) the smallest norm solution is x=A+b
– If this solution is not feasible, differentiate

Lagrangian wrt x to obtain A^TAx-A^Tb+2λx=0

– Solution takes the form x = $(A^T A+2\lambda I)^{-1} A^T b$

– Choosing λ: continue solving linear equation and increasing λ until x has the correct norm.

Karush-Kuhn-Tucker is a very general solution to constrained optimization

- While Lagrangian allows equality constraints, KKT allows both equality and inequality constraints
- To define a generalized Lagrangian we need to describe S in terms of equalities and inequalities.

Generalized Lagrangian: Set S is described in terms of m functions g(i) and n functions h(j) so that

$$S = \left\{\boldsymbol{x} \mid \forall i, g^{(i)}(\boldsymbol{x}) = 0 \text{ and } \forall j, h^{(j)}(\boldsymbol{x}) \leq 0\right\} \tag{2.12}$$

– Functions of g are equality constraints and functions of h are inequality constraints

- Introduce new variables λ_i and α_j for each constraint (called KKT multipliers) giving the generalized Lagrangian and here We can now solve the unconstrained optimization problem

2.6 SUMMARY

This chapter is focused on need of Numerical computation. We faced many problems while calculations with respect to memory, multiple errors we faced like Overflow and Underflow, We can use softmax function to resolved problems. Use of Gradient Based Optimization and Constraint optimization explain well for better calculations.

UNIT END EXERCISES

1. Compare Overflow and Underflow
2. Explain Softmax function in detail
3. Write note on Poor Conditioning

4. What is a need of Gradient Optimization and Explain in detail.
5. Define Minimum, Maximum and Saddle point
6. Define Local minimum
7. Explain Constraint Optimization ways.

BIBLIOGRAPHY

1. Ian Goodfellow, Yoshua Bengio, Aaron Courvile "Deep Learning",1st edition 2016
2. https://towardsdatascience.com/deep-learning-chapter-4-numerical-computation-14de09526931
3. https://medium.com/computronium/numerical-method-considerations-for-machine-learning-e69331d28611
4. https://challengeenthusiast.com/2018/07/10/stabilizing-against-overflow-and-underflow/
5. https://cedar.buffalo.edu/~srihari/CSE676/

*****

UNIT II

3

DEEP NETWORKS

Unit Structure

3.0 OBJECTIVES

After going through this unit, you will be able to:

- Define Deep feedforward network and techniques
- State the characteristics in Deep feedforward network
- Describe the basic concept of Regularization for deep learning and its types.
- Explain Optimization for Training deep models

3.1 INTRODUCTION

A deep neural network (DNN) is an artificial neural network (ANN) with multiple layers between the input and output layers. There are different types of neural networks but they always consist of the same components: neurons, synapses, weights, biases, and functions. These components functioning similar to the human brains and can be trained like any other ML algorithm.

Deep learning is part of a broader family of machine learning methods based on artificial neural networks with representation learning. Learning can be supervised, semi-supervised or unsupervised.

For example, a DNN that is trained to recognize dog breeds will go over the given image and calculate the probability that the dog in the image is a certain breed. The user can review the results and select which probabilities the network should display (above a certain threshold, etc.) and return the proposed label. Each mathematical manipulation as such is considered a layer, and complex DNN have many layers, hence the name "deep" networks.

DNNs can model complex non-linear relationships. DNN architectures generate compositional models where the object is expressed as a layered composition of primitives. The extra layers enable composition of features from lower layers, potentially modeling complex data with fewer units than a similarly performing shallow network. For instance, it was proved that sparse multivariate polynomials are exponentially easier to approximate with DNNs than with shallow networks.

Deep architectures include many variants of a few basic approaches. Each architecture has found success in specific domains. It is not always possible to compare the performance of multiple architectures, unless they have been evaluated on the same data sets.

3.2 DEEP FEEDFORWARD NETWORK

Multi-layered Network of neurons is composed of many sigmoid neurons. MLNs are capable of handling the non-linearly separable data. The layers present between the input and output layers are called hidden layers. The hidden layers are used to handle the complex non-linearly separable relations between input and the output.

Deep feedforward networks, also often called feedforward neural networks, or multilayer perceptrons (MLPs), are the quintessential deep learning models. The goal of a feedforward network is to approximate some function $f*$. For example, for a classifier, $y = f*(x)$ maps an input x to a category y. A feedforward network defines a mapping $y = f(x; \theta)$ and learns the value of the parameters θ that result in the best function approximation.

These models are called feedforward because information flows through the function being evaluated from x, through the intermediate computations used to define f, and finally to the output y. There are no feedback connections in which outputs of the model are fed back into

itself. When feedforward neural networks are extended to include feedback connections, they are called recurrent neural networks. Feedforward networks are of extreme importance to machine learning practitioners. They form the basis of many important commercial applications.

3.2.1 Simple Deep Neural Network:

In simple neural network to solve complex non-linear decision boundaries. For example of a mobile phone like/dislike predictor with two variables: screen size, and the cost. It has a complex decision boundary as shown below,

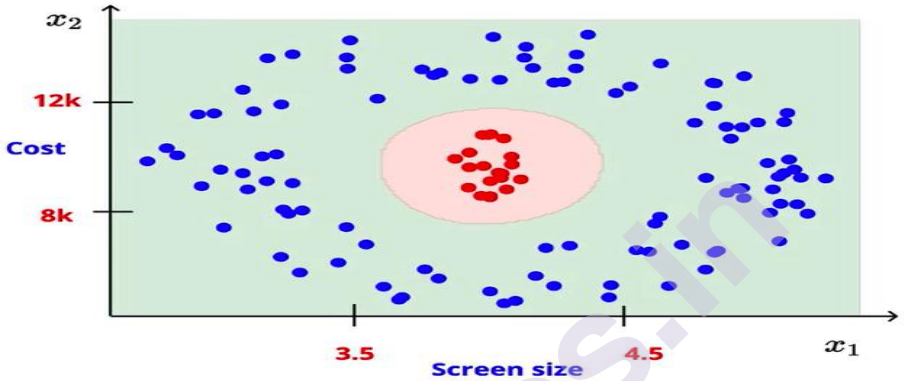


Fig. 1 Mobile phone Predictor

Decision Boundary:

In single sigmoid neuron, it is impossible to obtain this kind of non-linear decision boundary. Regardless of how we vary the sigmoid neuron parameters **w** and **b**. Now change the situation and use a simple network of neurons for the same problem and see how it handles.

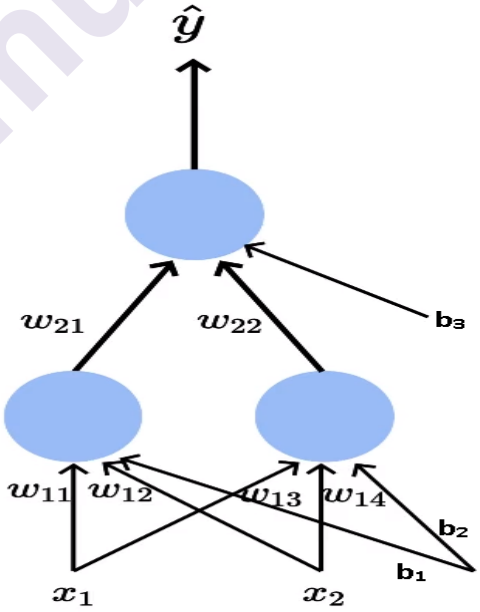


Fig. 2 Simple Neural Network

Simple Neural Network:

Here inputs x_1 - screen size and x_2 - price going into the network along with the bias b_1 and b_2 . In break down the model neuron by neuron to understand. We have our first neuron (leftmost) in the first layer which is connected to inputs x_1 and x_2 with the weights $w_{1\ 1}$ and $w_{1\ 2}$ and the bias b_1 and b_2 . The output of that neuron represented as h_1 , which is a function of x_1 and x_2 with parameters $w_{1\ 1}$ and $w_{1\ 2}$.

$$h_1 = f_1 (x_1, x_2)$$

Output of the first Neuron:

If we apply the sigmoid function to the inputs x_1 and x_2 with the appropriate weights $w_{1\ 1}$, $w_{1\ 2}$ and bias b_1 we would get an output h_1 , which would be some real value between 0 and 1. The sigmoid output for the first neuron h_1 will be given by the following equation,

$$h_1 = \frac{1}{1+e^{-(w_{11}*x_1+w_{12}*x_2+b_1)}}$$

Output Sigmoid for First Neuron:

Next up, we have another neuron in the first layer which is connected to inputs x_1 and x_2 with the weights $w_{1\ 3}$ and $w_{1\ 4}$ along with the bias b_3 and b_4 . The output of the second neuron represented as h_2 .

$$H_2 = f_2 (x_1, x_2)$$

Output of the second Neuron:

Similarly, the sigmoid output for the second neuron h_2 will be given by the following equation,

$$h_2 = \frac{1}{1+e^{-(w_{13}*x_1+w_{14}*x_2+b_2)}}$$

So far we have seen the neurons present in the first layer but we also have another output neuron which takes h_1 and h_2 as the input as opposed to the previous neurons. The output from this neuron will be the final predicted output, which is a function of h_1 and h_2 . The predicted output is given by the following equation,

$$\hat{y} = g(h_1, h_2)$$

$$\hat{y} = \frac{1}{1+e^{-(w_{21}*h_1+w_{22}*h_2+b_3)}}$$

$$= \frac{1}{1+e^{-(w_{21}*(\frac{1}{1+e^{-(w_{11}*x_1+w_{12}*x_2+b_1)}})+w_{22}*(\frac{1}{1+e^{-(w_{13}*x_1+w_{14}*x_2+b_2)}})+b_3)}}$$

We can adjust only w_1 , w_2 and b - parameters of a single sigmoid neuron. Now we can adjust the 9 parameters ($w_{1\,1}$, $w_{1\,2}$, $w_{1\,3}$, $w_{1\,4}$, $w_{2\,1}$, $w_{2\,2}$, b_1 , b_2 , b_3), which allows the handling of much complex decision boundary. By trying out the different configurations for these parameters we would be able to find the optimal surface where output for the entire middle area (red points) is one and everywhere else is zero, what we desire.

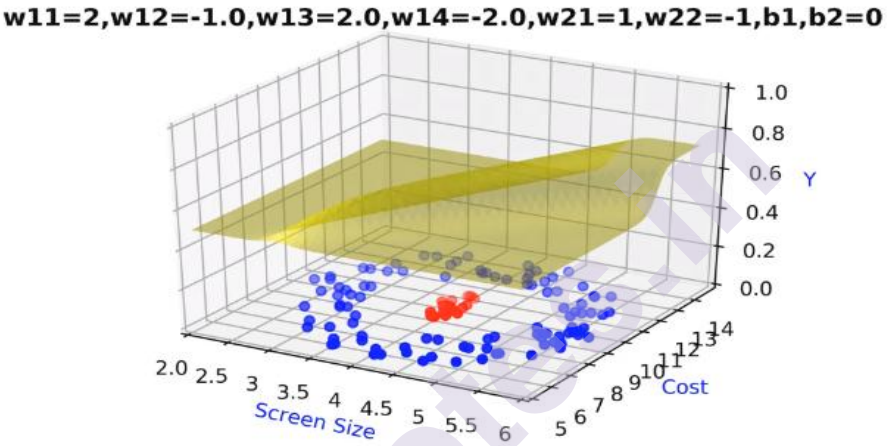


Fig. 3 Decision Boundary from Network

3.2.2 Generic Deep Neural Network:

Previously, we have seen the neural network for a specific task, now we will check about the neural network in generic form.

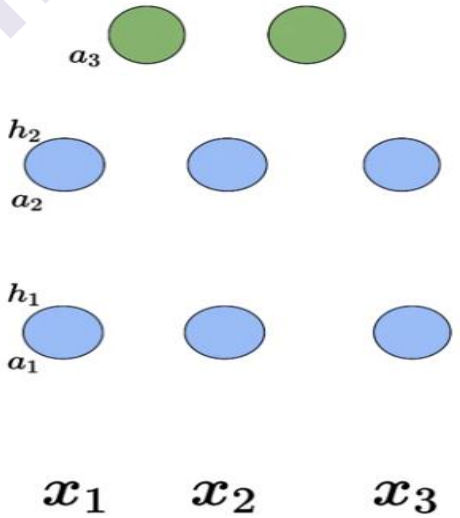


Fig. 4 Generic Network without Connections

In network of neurons with two hidden layers (in blue but there can more than 2 layers if needed) and each hidden layer has 3 sigmoid neurons there can be more neurons. In three inputs going into the network and there are two neurons in the output layer. We will take this network as it is and understand the intricacies of the deep neural network.

The terminology then we will go into how these neurons interact with each other. For each of these neurons, two things will happen

1. Pre-activation represented by 'a': It is a weighted sum of inputs plus the bias.
2. Activation represented by 'h': Activation function is Sigmoid function.

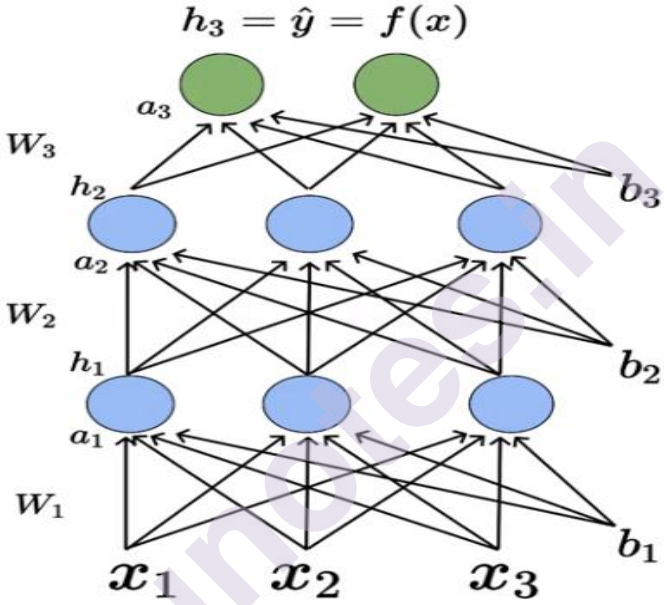


Fig. 5 Generic Network with Connections

Let's understand the network neuron by neuron. Consider the first neuron present in the first hidden layer. The first neuron is connected to each of the inputs by weight $\mathbf{W}_1$.

Going forward I will be using this format of the indices to represent the weights and biases associated with a particular neuron,

$W_{1\ 1\ 1}$ — Weight associated with the first neuron present in the first hidden layer connected to the first input.

$W_{1\ 1\ 2}$ — Weight associated with the first neuron present in the first hidden layer connected to the second input.

$b_{1\ 1}$ —Bias associated with the first neuron present in the first hidden layer.

$b_{1\ 2}$ —Bias associated with the second neuron present in the first hidden layer.

Here W_1 a weight matrix containing the individual weights associated with the respective inputs. The pre-activation at each layer is the weighted sum of the inputs from the previous layer plus bias. The mathematical equation for pre-activation at each layer 'i' is given by,

$$a_i(x) = W_i h_{i-1}(x) + b_i$$

Pre-activation Function:

The activation at each layer is equal to applying the sigmoid function to the output of pre-activation of that layer. The mathematical equation for the activation at each layer 'i' is given by,

$$h_i(x) = g(a_i(x))$$

where 'g' is called activation Function.

Finally, we can get the predicted output of the neural network by applying some kind of activation function (could be softmax depending on the task) to the pre-activation output of the previous layer. The equation for the predicted output is shown below,

$$f(x) = h_L = O(a_L)$$

Where 'O' is called as the output activation function.

3.2.3 Computations in Deep Neural Network:

Consider that you have 100 inputs and 10 neurons in the first and second hidden layers. Each of the 100 inputs will be connected to the neurons will be The weight matrix of the first neuron W_1 will have a total of 10 x 100 weights.

$$W_1 = \begin{bmatrix} w_{1\,1\,1} & w_{1\,1\,2} & \cdots & w_{1\,1\,99} & w_{1\,1\,100} \\ w_{1\,2\,1} & w_{1\,2\,2} & \cdots & w_{1\,2\,99} & w_{1\,2\,100} \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ w_{1\,10\,1} & w_{1\,10\,2} & \cdots & w_{1\,10\,99} & w_{1\,10\,100} \end{bmatrix} \quad X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{100} \end{bmatrix}$$

Weight Matrix:

Remember, we are following a very specific format for the indices of the weight and bias variable as shown below,

W(Layer number)(Neuron number in the layer)(Input number)

b(Layer number)(Bias number associated for that input)

Now let's see how we can compute the pre-activation for the first neuron of the first layer $a_{1\,1}$. We know that pre-activation is nothing but the weighted sum of inputs plus bias. In other words, it is the dot product between the first row of the weight matrix W_1 and the input matrix X plus bias $b_{1\,1}$.

$$a_{1\,1} = w_{1\,1\,1} * x_1 + w_{1\,1\,2} * x_2 + w_{1\,1\,3} * x_3 + ... + w_{1\,1\,100} * x_{100} + b_{11}$$

Similarly the pre-activation for other 9 neurons in the first layer given by,

$$a_{1\,2} = w_{1\,2\,1} * x_1 + w_{1\,2\,2} * x_2 + w_{1\,2\,3} * x_3 + ... + w_{1\,2\,100} * x_{100} + b_{12}$$

$$\vdots \qquad \vdots \qquad \vdots$$

$$a_{1\,10} = w_{1\,10\,1} * x_1 + w_{1\,10\,2} * x_2 + w_{1\,10\,3} * x_3 + ... + w_{1\,10\,100} * x_{100} + b_{1,10}$$

In short, the overall pre-activation of the first layer is given by,

$a_1 = W_1 * x + b_1$

Where,

W_1 is a matrix containing the individual weights associated with the corresponding inputs and b_1 is a vector containing($b_{1\,1}$, $b_{1\,2}$, $b_{1\,3}$,….,$b_{1\,0}$) the individual bias associated with the sigmoid neurons.
The activation for the first layer is given by,

$h_1 = g(a_1)$

Where 'g' represents the sigmoid function.

Remember that a_1 is a vector of 10 pre-activation values, here we are applying the element-wise sigmoid function on all these 10 values and storing them in another vector represented as h_1. Similarly, we can compute the pre-activation and activation values for 'n' number of hidden layers present in the network.

Output Layer of DNN:

So far we have talked about the computations in the hidden layer. Now we will talk about the computations in the output layer.

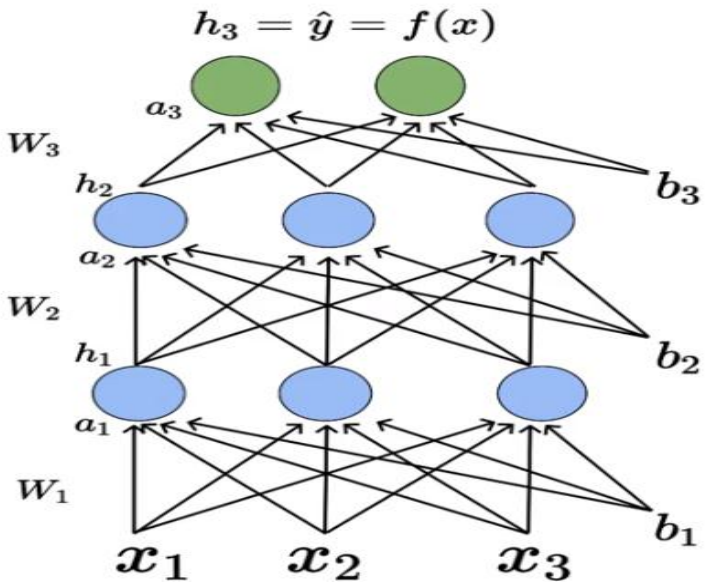


Fig. 6 The output Activation function is chosen depending on the task at hand, can be softmax or linear.

Softmax Function:

We will use the Softmax function as the output activation function. The most frequently used activation function in deep learning for classification problems.

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

Softmax Function:

In the Softmax function, the output is always positive, irrespective of the input.

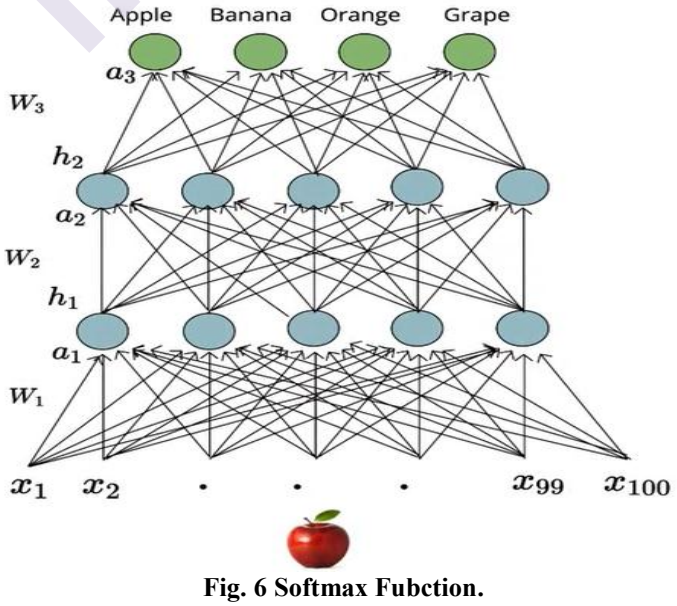


Fig. 6 Softmax Fubction.

Now, let us illustrate the Softmax function on the above-shown network with 4 output neurons. The output of all these 4 neurons is represented in a vector 'a'. To this vector, we will apply our softmax activation function to get the predicted probability distribution as shown below,

$$a = [a_1\ a_2\ a_3\ a_4]$$

$$softmax(a) = \left[\frac{e^{a_1}}{\Sigma_{j=1}^{k} e^{a_j}}\ \frac{e^{a_2}}{\Sigma_{j=1}^{k} e^{a_j}}\ \frac{e^{a_3}}{\Sigma_{j=1}^{k} e^{a_j}}\ \frac{e^{a_4}}{\Sigma_{j=1}^{k} e^{a_j}}\right]$$

By applying the softmax function we would get a predicted probability distribution and our true output is also a probability distribution, we can compare these two distributions to compute the loss of the network.

Loss Function:

In this section, we will talk about the loss function for binary and multi-class classification. The purpose of the loss function is to tell the model that some correction needs to be done in the learning process.
In general, the number of neurons in the output layer would be equal to the number of classes. But in the case of binary classification, we can use only one sigmoid neuron which outputs the probability P(Y=1) therefore we can obtain P(Y=0) = 1-P(Y=1). In the case of classification, We will use the cross-entropy loss to compare the predicted probability distribution and the true probability distribution.

Cross-entropy loss for binary classification is given by,

Cross Entropy Loss:

$$L(\Theta) = \begin{cases} -log(\hat{y}) & \text{if } y = 1 \\ -log(1-\hat{y}) & \text{if } y = 0 \end{cases}$$

Cross-entropy loss for multi-class classification is given by,

Cross Entropy Loss:

$$L(\Theta) = -\sum_{i=1}^{k} y_i \log(\hat{y}_i)$$

3.2.4 Gradient-Based Learning:

Designing and training a neural network is not much different from training any other machine learning model with gradient descent. The largest difference between the linear models we have seen so far and neural networks is that the nonlinearity of a neural network causes most interesting loss functions to become non-convex. This means that neural networks are usually trained by using iterative, gradient-based optimizers that merely drive the cost function to a very low value, rather than the linear equation solvers used to train linear regression models or the convex

optimization algorithms with global convergence guarantees used to train logistic regression or SVMs.

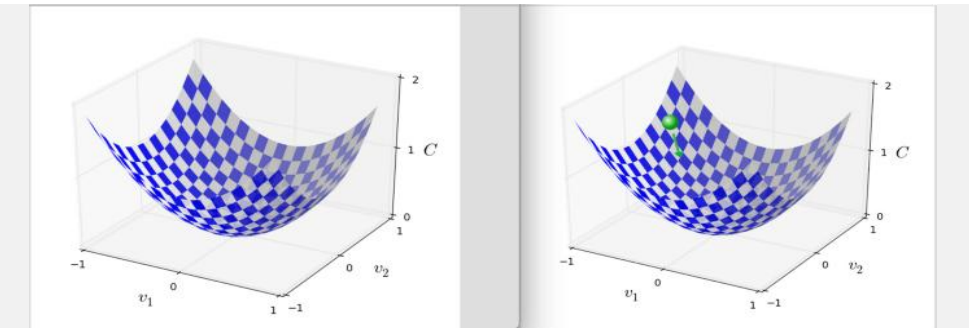


Fig 7. Ball diagram depicting the visualisation of how gradient based learning occur.

As we can see at right of figure 7, an analogy of a ball dropped in a deep valley and it settle downs at the bottom of the valley. Similarly we want our cost function to get minimize and get to the minimum value possible. When we move the ball a small amount Δv1 in the v1 direction, and a small amount Δv2 in the v2 direction. Calculus tells us that C changes as follows:

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2.$$

Change of the v1 and v2 such that the change in cost is negative is desirable. We can also denote $\Delta C \approx \nabla C \cdot \Delta v$. where ∇C is,

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \ldots, \frac{\partial C}{\partial v_m}\right)^T.$$

and Δv is,

$$\Delta v = (\Delta v_1, \ldots, \Delta v_m)^T$$

Indeed, there's even a sense in which gradient descent is the optimal strategy for searching for a minimum. Let's suppose that we're trying to make a move Δv in position so as to decrease C as much as possible. We'll constrain the size of the move so that $\|\Delta v\| = \epsilon$ for some small fixed $\epsilon > 0$. In other words, we want a move that is a small step of a fixed size, and we're trying to find the movement direction which decreases C as much as possible. It can be proved that the choice of Δv which minimizes $\nabla C \cdot \Delta v$ is $\Delta v = -\eta \nabla C$, where $\eta = \epsilon / \|\nabla C\|$ is determined by the size constraint $\|\Delta v\| = \epsilon$. So gradient descent can be viewed as a way of taking small steps in the direction which does the most to immediately decrease C. Now that the gradient vector ∇C has corresponding components $\partial C/\partial w_k$ and $\partial C/\partial b_\ell$. Writing out the gradient descent update rule in terms of components, we have

$$w_k \rightarrow w'_k = w_k - \eta \frac{\partial C}{\partial w_k}$$
$$b_l \rightarrow b'_l = b_l - \eta \frac{\partial C}{\partial b_l}.$$

Now there are many challenges in training gradient based learning. But for now I just want to mention one problem. When the number of training inputs is very large this can take a long time, and learning thus occurs slowly. An idea called ***stochastic gradient descent*** can be used to speed up learning. To make these ideas more precise, stochastic gradient descent works by randomly picking out a small number m of randomly chosen training inputs. We'll label those random training inputs X1,X2,…,Xm and refer to them as a *mini-batch*. Provided the sample size mm is large enough we expect that the average value of the ∇CXj will be roughly equal to the average over all ∇Cx, that is,

$$\frac{\sum_{j=1}^{m} \nabla C_{X_j}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C,$$

This modification helps us in reducing a good amount of computational load. Stochastic gradient descent applied to non-convex loss functions has no such convergence guarantee, and is sensitive to the values of the initial parameters. For feedforward neural networks, it is important to initialize all weights to small random values. The biases may be initialized to zero or to small positive values.

3.3 Regularization for deep learning

Regularization is a set of strategies used in Machine Learning to reduce the generalization error. Most models, after training, perform very well on a specific subset of the overall population but fail to generalize well. This is also known as overfitting. Regularization strategies aim to reduce overfitting and keep, at the same time, the training error as low as possible.

ResNet CNN architecture were originally proposed in 2015. A recent paper called "Revisiting ResNets: Improved Training and Scaling Strategies" applied modern regularization methods and achieved more than 3% test set accuracy on Imagenet. If the test set consists of 100K images, this means that 3K more images were classified correctly!

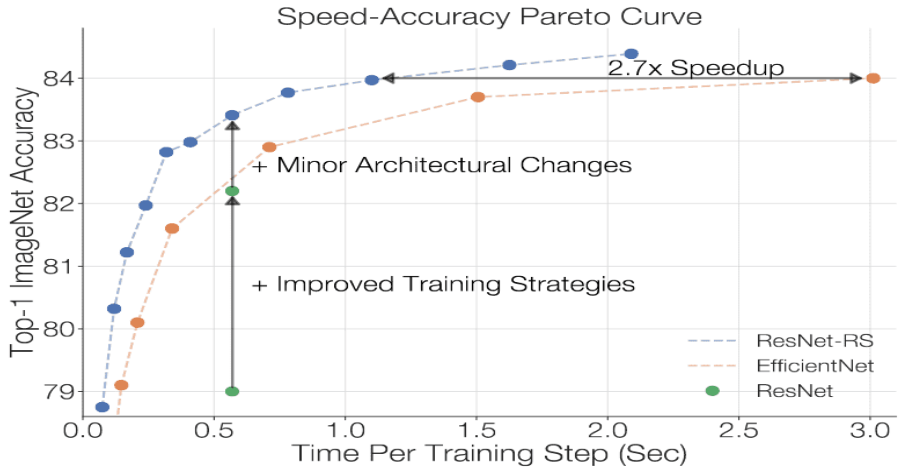


Fig. 8 Revisiting ResNets: Improved Training and Scaling Strategies

In the context of deep learning, most regularization strategies are based on regularizing estimators. Regularization of an estimator works by trading increased bias for reduced variance. An eff ective regularizer is one that makes a profitabletrade, reducing variance significantly while not overly increasing the bias.

In simple terms, regularization results in **simpler models**. And as the Occam's razor principle argues: the simplest models are the most likely to perform better. Actually, we constrain the model to a smaller set of possible solutions by introducing different techniques.

To get a better insight you need to understand the famous bias-variance tradeoff.

The bias-variance tradeoff: overfitting and underfitting

First, let's clarify that bias-variance tradeoff and overfitting-underfitting are equivalent.

Overfitting refers to the phenomenon where a neural network models the training data very well but fails when it sees new data from the same problem domain. Overfitting is caused by noise in the training data that the neural network picks up during training and learns it as an underlying concept of the data.

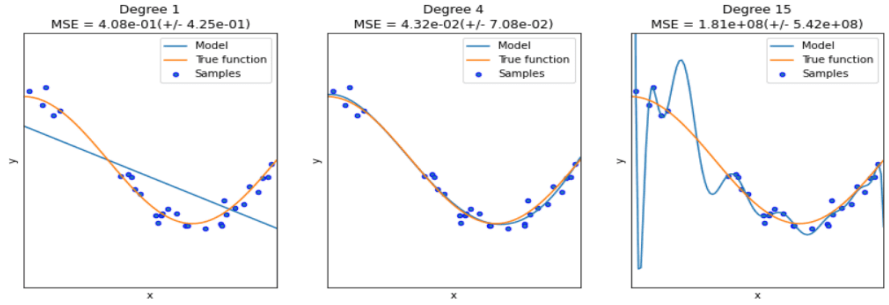


Fig. 9 Underfitting and overfitting

The **bias error** is an error from wrong assumptions in the learning algorithm. High bias can cause an algorithm to miss the relevant relations between features and target outputs. This is called **underfitting.**

The **variance** is an error from sensitivity to small fluctuations in the training set. High variance may result in modeling the random noise in the training data. This is called **overfitting.**

The **bias-variance tradeoff** is a term to describe the fact that we can reduce the variance by increasing the bias. Good regularization techniques strive to simultaneously minimize the two sources of error. Hence, achieving better generalization.

3.3.1 L2 regularization:

The L2 regularization is the most common type of all regularization techniques and is also commonly known as weight decay or Ride Regression.

The mathematical derivation of this regularization, as well as the mathematical explanation of why this method works at reducing overfitting, is quite long and complex. Since this is a very practical article I don't want to focus on mathematics more than it is required. Instead, I want to convey the intuition behind this technique and most importantly how to implement it so you can address the overfitting problem during your deep learning projects.

During the L2 regularization the loss function of the neural network as extended by a so-called regularization term, which is called here **Ω**.

The L2 regularizer will have a big impact on the directions of the weight vector that don't "contribute" much to the loss function. On the other hand, it will have a relatively small effect on the directions that contribute to the loss function. As a result, we reduce the variance of our model, which makes it easier to generalize on unseen data.

The regularization term **Ω** is defined as the Euclidean Norm (or L2 norm) of the weight matrices, which is the sum over all squared weight values of a weight matrix. The regularization term is weighted by the scalar alpha divided by two and added to the regular loss function that is chosen for the current task. This leads to a new expression for the loss function:

$$\hat{\mathcal{L}}(W) = \frac{\alpha}{2}||W||_2^2 + \mathcal{L}(W) = \frac{\alpha}{2}\sum_i \sum_j w_{ij}^2 + \mathcal{L}(W)$$

Eq 2. Regularization loss during L2 regularization.

Alpha is sometimes called as the **regularization rate** and is an additional hyperparameter we introduce into the neural network. Simply speaking alpha determines how much we regularize our model.

In the next step we can compute the gradient of the new loss function and put the gradient into the update rule for the weights:

$$\nabla_W \hat{\mathcal{L}}(W) = \alpha W + \nabla_W \mathcal{L}(W)$$

$$W_{new} = W_{old} - \epsilon(\alpha W_{old} + \nabla_W \mathcal{L}(W_{old}))$$

Eq. 3 Gradient Descent during L2 Regularization.

Some reformulations of the update rule lead to the expression which very much looks like the update rule for the weights during regular gradient descent:

$$W_{new} = (1 - \epsilon\alpha)W_{old} - \epsilon\nabla_W \mathcal{L}(W_{old})$$

Eq.4 Gradient Descent during L2 Regularization.

The only difference is that by adding the regularization term we introduce an additional subtraction from the current weights (first term in the equation).

In other words independent of the gradient of the loss function we are making our weights a little bit smaller each time an update is performed.

3.3.2 L1 regularization:

In the case of L1 regularization (also knows as Lasso regression), we simply use another regularization term **Ω**. This term is the sum of the absolute values of the weight parameters in a weight matrix:

Eq.5 Regularization Term for L1 Regularization.

As in the previous case, we multiply the regularization term by alpha and add the entire thing to the loss function.

$$\hat{\mathcal{L}}(W) = \alpha||W||_1 + \mathcal{L}(W)$$

Eq.6 Loss function during L1 Regularization.

The derivative of the new loss function leads to the following expression, which the sum of the gradient of the old loss function and sign of a weight value times alpha.

$$\nabla_W \hat{\mathcal{L}}(W) = \alpha sign(W) + \nabla_W \mathcal{L}(W)$$

Eq. 7 Gradient of the loss function during L1 Regularization.

As we can see, the regularization term does not scale linearly, contrary to L2 regularization, but it's a constant factor with an alternating sign. How does this affect the overall training?

The L1 regularizer introduces sparsity in the weights by forcing more weights to be zero instead of reducing the average magnitude of all weights (as the L2 regularizer does). In other words, L1 suggests that some features should be discarded whatsoever from the training process.

Elastic net:

Elastic net is a method that linearly combines L1 and L2 regularization with the goal to acquire the best of both worlds. More specifically the penalty term is as follows.

\Omega(\theta) = \lambda_1 ||w||_1 + \lambda_2||w||^2_2Ω(*θ*)=*λ*1||*w*||1 +*λ*2||*w*||22

Elastic Net regularization reduces the effect of certain features, as L1 does, but at the same time, it does not eliminate them. So it combines feature elimination from L1 and feature coefficient reduction from the L2.

What does Regularization achieve?:

- Performing L2 regularization encourages the weight values towards zero (but not exactly zero)
- Performing L1 regularization encourages the weight values to be zero

Intuitively speaking smaller weights reduce the impact of the hidden neurons. In that case, those hidden neurons become neglectable and the overall complexity of the neural network gets reduced.

As mentioned earlier less complex models typically avoid modeling noise in the data, and therefore, there is no overfitting.

But you have to be careful. When choosing the regularization term **α.** The goal is to strike the right balance between low complexity of the model and accuracy

- If your alpha value is too high, your model will be simple, but you run the risk of *underfitting* your data. Your model won't learn enough about the training data to make useful predictions.
- If your alpha value is too low, your model will be more complex, and you run the risk of *overfitting* your data. Your model will learn too

much about the particularities of the training data, and won't be able to generalize to new data.

L2 & L1 regularizatio:

L1 and L2 are the most common types of regularization. These update the general cost function by adding another term known as the regularization term.

Cost function = Loss (say, binary cross entropy) + Regularization term

Due to the addition of this regularization term, the values of weight matrices decrease because it assumes that a neural network with smaller weight matrices leads to simpler models. Therefore, it will also reduce overfitting to quite an extent.

However, this regularization term differs in L1 and L2.

In L2, we have:

$$Cost\,function = Loss + \frac{\lambda}{2m} * \sum ||w||^2$$

Here, lambda is the regularization parameter. It is the hyperparameter whose value is optimized for better results. L2 regularization is also known as weight decay as it forces the weights to decay towards zero (but not exactly zero).

In L1, we have:

$$Cost\,function = Loss + \frac{\lambda}{2m} * \sum ||w||$$

In this, we penalize the absolute value of the weights. Unlike L2, the weights may be reduced to zero here. Hence, it is very useful when we are trying to compress our model. Otherwise, we usually prefer L2 over it.
Similarly, we can also apply L1 regularization.

3.3.3 Entropy Regularization:

Entropy regularization is another norm penalty method that applies to probabilistic models. It has also been used in different Reinforcement Learning techniques such as A3C and policy optimization techniques. Similarly to the previous methods, we add a penalty term to the loss function.

The term "Entropy" has been taken from information theory and represents the average level of "information" inherent in the variable's

possible outcomes. An equivalent definition of entropy is the expected value of the information of a variable.

One very simple explanation of why it works is that it forces the probability distribution towards the uniform distribution to reduce variance.

In the context of Reinforcement Learning, one can say that the entropy term added to the loss, promotes action diversity and allows better exploration of the environment.

3.3.4 Dropout:

In addition to the L2 and L1 regularization, another famous and powerful regularization technique is called the dropout regularization. The procedure behind dropout regularization is quite simple.

In a nutshell, dropout means that during training with some probability **P** a neuron of the neural network gets turned off during training.

Another strategy to regularize deep neural networks is dropout. Dropout falls into noise injection techniques and can be seen as noise injection into the hidden units of the network.

In practice, during training, some number of layer outputs are randomly ignored (dropped out) with probability p*p*.

During test time, all units are present, but they have been scaled down by p*p*. This is happening because after dropout, the next layers will receive lower values. In the test phase though, we are keeping all units so the values will be a lot higher than expected. That's why we need to scale them down.

By using dropout, the same layer will alter its connectivity and will search for alternative paths to convey the information in the next layer. As a result, each update to a layer during training is performed with a different "view" of the configured layer. Conceptually, it approximates training a large number of neural networks with different architectures in parallel.

"Dropping" values means temporarily removing them from the network for the current forward pass, along with all its incoming and outgoing connections. Dropout has the effect of making the training process noisy. The choice of the probability p*p* depends on the architecture.

Other Dropout variations:

There are many more variations of Dropout that have been proposed over the years. To keep this article relatively digestible, I won't go into many details for each one. But I will briefly mention a few of them.

1. **Inverted dropout:** also randomly drops some units with a probability pp. The difference with traditional dropout is: During training, it also scales the activations by the inverse of the keep probability 1-p1−p. The reason behind this is: to prevent the activations from becoming too large thus the need to modify the network during the testing phase. The end result will be similar to the traditional dropout.

2. **Gaussian dropout**: instead of dropping units during training, is injecting noise to the weights of each unit. The noise is, more often than not ,Gaussian. This results in:

    1. A reduction in the computational effort during testing time.

    2. No weight scaling is required.

    3. Faster training overall

3. **DropConnect** follows a slightly different approach. Instead of zeroing out random activations (units), **it zeros random weights during each forward pass**. The weights are dropped with a probability of 1-p1−p. This essentially transforms a fully connected layer to a sparsely connected layer. Mathematically we can represent DropConnect as: r = a \left(\left(M * W\right){v}\right)$r=a((M∗W)v)$ where rr is the layers' output, vv the input, WW the weights and MM a binary matrix. MM is a mask that instantiates a different connectivity pattern from each data sample. Usually, the mask is derived from each training example. **DropConnect** can be seen as a generalization of Dropout to the full-connection structure of a layer.

4. **Variational Dropout**: we use the same dropout mask on each timestep. This means that we will drop the same network units each time.

5. **Attention Dropout**: popular over the past years because of the rapid advancements of attention-based models like Transformers. As you may have guessed, we randomly dropped certain attention units with a probability pp.

6. **Adaptive Dropout:** a technique that extends dropout by allowing the dropout probability to be different for different units. The intuition is that there may be hidden units that can individually make confident predictions for the presence or absence of an important feature or combination of features.

7. **Embedding Dropout**: a strategy that performs dropout on the embedding matrix and is used for a full forward and backward pass.

8. **DropBlock:** is used in Convolutional Neural networks and it discards all units in a continuous region of the feature map.

Stochastic Depth:

Stochastic depth goes a step further. It drops entire network blocks while keeping the model intact during testing. The most popular application is in large ResNets where we bypass certain blocks through their skip connections.

In particular, Stochastic depth drops out each layer in the network that has residual connections around it. It does so with a specified probability p*p* that is a function of the layer depth.

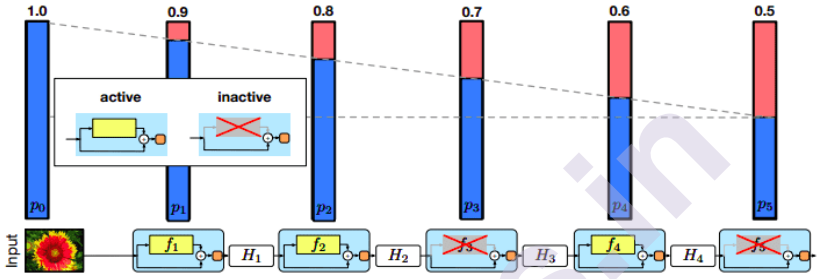


Fig. 10 Deep Networks with Stochastic Depth

Early stopping:

Early stopping is one of the most commonly used strategies because it is very simple and quite effective. It refers to the process of stopping the training when the training error is no longer decreasing but the validation error is starting to rise**.**

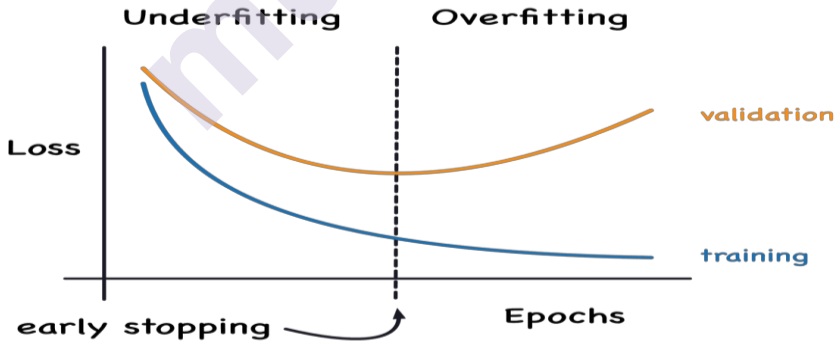


Fig. 11 Process of Early Stopping.

This implies that we store the trainable parameters periodically and track the validation error. After the training stopped, we return the trainable parameters to the exact point where the validation error started to rise, instead of the last ones.

A different way to think of early stopping is as a very efficient hyperparameter selection algorithm, which sets the number of epochs to

the absolute best. It essentially restricts the optimization procedure to a small volume of the trainable parameters space close to the initial parameters.

It can also be proven that in the case of a simple linear model with a quadratic error function and simple gradient descent, early stopping is equivalent to L2 regularization.

Parameter sharing:

Parameter sharing follows a different approach. Instead of penalizing model parameters, it **forces a group of parameters to be equal**. This can be seen as a way to apply our previous domain knowledge to the training process. Various approaches have been proposed over the years but the most popular one is by far Convolutional Neural Networks.

Convolutional Neural Networks take advantage of the spatial structure of images by sharing parameters across different locations in the input. Since each kernel is convoluted with different blocks of the input image, the weight is shared among the blocks instead of having separate ones.

Batch normalization:

Batch normalization (BN) can also be used as a form of regularization. Batch normalization fixes the means and variances of the input by bringing the feature in the same range. More specifically, we concentrate the features in a compact Gaussian-like space.

Visually this can be represented as:

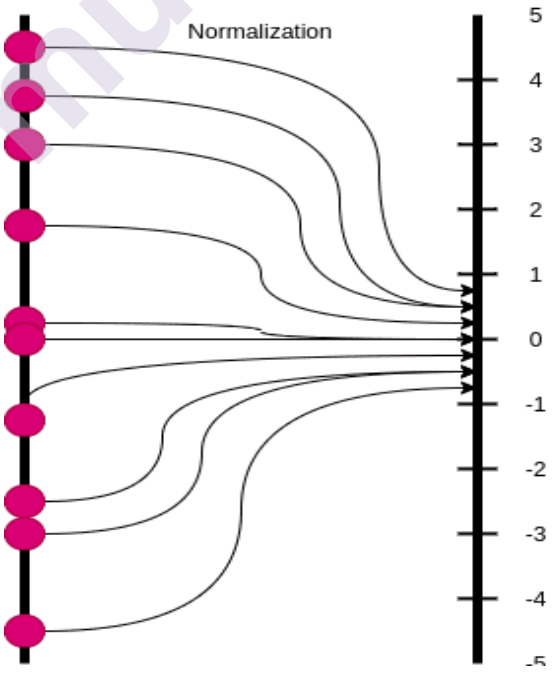


Fig. 12 Batch Normalization

Batch normalization can implicitly regularize the model and in many cases, it is preferred over Dropout.

One can think of batch normalization as a similar process with dropout because it essentially injects noise. Instead of multiplying each hidden unit with a random value, it multiplies them with the deviation of all the hidden units in the minibatch. It also subtracts a random value from each hidden unit at each step.

3.3.5 Data augmentation:

Data augmentation is the final strategy that we need to mention. Although not strictly a regularization method, it sure has its place here.

Data augmentation refers to the process of **generating new training examples to our dataset**. More training data means lower model's variance, a.k.a lower generalization error. Simple as that. It can also be seen as a form of noise injection in the training dataset.

Data augmentation can be achieved in many different ways. Let's explore some of them.

1. **Basic Data Manipulations**: The first simple thing to do is to perform geometric transformations on data. Most notably, if we're talking about images we have solutions such as: Image flipping, cropping, rotations, translations, image color modification, image mixing etc. **Cutout** is a commonly used idea where we remove certain image regions. Another idea, called **Mixup**, is the process of blending two images from the dataset into one image.

2. **Feature Space Augmentation:** Instead of transforming data in the input space as above, we can apply transformations on the feature space. For example, an autoencoder might be used to extract the latent representation. Noise can then be added in the latent representation which results in a transformation of the original data point.

3. **GAN-based Augmentation:** Generative Adversarial Networks have been proven to work extremely well on data generation so they are a natural choice for data augmentation.

4. **Meta-Learning:** In meta-learning, we use neural networks to optimize other neural networks by tuning their hyperparameters, improving their layout, and more. A similar approach can also be applied in data augmentation. In simple terms, we use a classification network to tune an augmentation network into generating better images. Example: We feed random images to an Augmentation Network (most likely a GAN), which will generate augmented images. Both the augmented image and the original are passed into a

second network, which compares them and tells us how good the augmented image is. After repeating the process the augmentation network becomes better and better at producing new images.

Regularization is an integral part of training Deep Neural Networks. In all the aforementioned strategies fall into two different high-level categories. They either penalize the trainable parameters or they inject noise somewhere along the training lifecycle. Whether this is on the training data, the network architecture, the trainable parameters or the target labels.

3.4 OPTIMIZATION FOR TRAINING DEEP MODELS

Deep learning algorithms involve optimization in many contexts. For example, performing inference in models such as PCA involves solving an optimization problem. We often use analytical optimization to write proofs or design algorithms. Of all of the many optimization problems involved in deep learning, the most difficult is neural network training. It is quite common to invest days to months of time on hundreds of machines in order to solve even a single instance of the neural network training problem. Because this problem is so important and so expensive, a specialized set of optimization techniques have been developed for solving it.

3.4.1 How Learning Differs from Pure Optimization:

Optimization algorithms used for training of deep models differ from traditional optimization algorithms in several ways. Machine learning usually acts indirectly. In most machine learning scenarios, we care about some performance measure P, that is defined with respect to the test set and may also be intractable. We therefore optimize P only indirectly. We reduce a different cost function J(θ) in the hope that doing so will improve P . This is in contrast to pure optimization, where minimizing J is a goal in and of itself. Optimization algorithms for training deep models also typically include some specialization on the specific structure of machine learning objective functions. In Machine Learning (ML), we care about a certain performance measure (say *P,* for e.g. accuracy) defined w.r.t the test set and optimize ***J(θ)*** (for e.g. cross-entropy loss) with the *hope* that it improves *P* as well. In pure optimization, optimizing ***J(θ)*** is the final goal.

- The expected *generalization error* (**risk**) is taken over the true data-generating distribution *p_data*. If we do have that, it becomes an optimization problem.

$$J^*(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},\mathrm{y})\sim p_{\mathrm{data}}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y)$$

Notice how the expectation is taken over the true data generating distribution.

When we don't have *p_data* but a finite training set, we have a ML problem. The latter can be converted back to an optimization problem by replacing *p_data* with the empirical distribution with p^_*data* obtained from the training set, thereby reducing the *empirical risk*. This is called *empirical risk minimization* (ERM):

$$\mathbb{E}_{\boldsymbol{x},\mathrm{y}\sim\hat{p}_{\mathrm{data}}(\boldsymbol{x},y)}[L(f(\boldsymbol{x};\boldsymbol{\theta}),y)] = \frac{1}{m}\sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}),y^{(i)})$$

Notice the change in distribution over which the expectation is taken.

Although this might look relatively similar to optimization, there are two main problems. Firstly, ERM is prone to overfitting with the possibility of the dataset being learned by high capacity models (models with the ability to learn extremely complex functions). Secondly, ERM might not be feasible. Most optimization algorithms now are based on *Gradient Descent* (GD) which requires a derivative calculation and hence, may not work with certain loss functions like the 0–1 loss (as it is not differentiable).

- It is for the reasons mentioned above that a **surrogate loss function** (SLF) is used instead, that acts as a proxy. For e.g. the *negative log-likelihood* of the true class is used as a surrogate for 0–1 loss. I've added a *code snippet* below that would help you understand why 0–1 loss won't work for Gradient Descent but cross-entropy, being a smooth function, would.

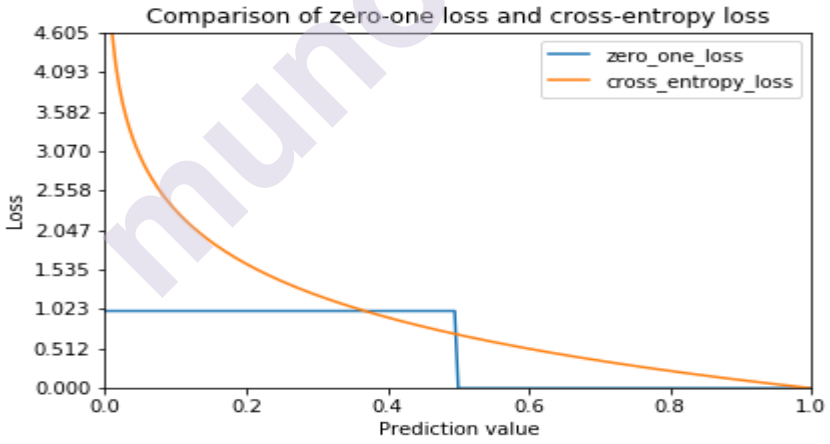


Fig. 13 Comparison of Zero-one loss and cross - entropy loss.

It can be seen that the 0–1 loss is a non-differentiable function and hence, not compatible with gradient-based algorithms like Gradient Descent. Cross-entropy is a smooth approximation of the 0–1 loss.

Using a SLF might even turn out to be beneficial as you can keep continuing to obtain a better test error by pushing the classes even further apart to get a more reliable classifier. By this, I mean that suppose we were using a 0–1 loss with a threshold of, say, 0.5 to assign each class. Here, in

case the true class is 1, our model would have no motivation to push the prediction score close to 1 once it's able to get it above 0.5. However, using cross-entropy, since we are using the raw prediction scores, the model keeps trying to push the prediction closer to the true class.

- Another common difference is that training might be halted following some convergence criterion based on Early Stopping to prevent overfitting, when the derivative of the surrogate loss function might still be *large*. This is different from pure optimization which is halted only when the derivative becomes very small.

- In ML optimization algorithms, the objective function decomposes as a sum over the examples and we can perform updates by randomly sampling a batch of examples and taking the average over the examples in that batch. If we consider *n* random variables, each having the true mean given by **μ**, the Standard Error (S.E.) of the mean *estimated* from those *n* random variables is given as follows:

$$S.E.(\hat{\mu}) = \frac{\sigma}{\sqrt{n}}$$

This indicates that as we include more examples for making an update, the *returns* of additional examples in improving the error is *less than linear*. Thus, if we use 100 and 10000 examples separately to make an update, the latter takes 100 times more compute, but reduces the error only by a factor of 10. Thus, it's better to compute *rapid approximate updates* rather than a *slow exact update*.

- There are 3 types of sampling based algorithms — *batch* gradient descent (BGD), where the **entire** training set is used to make a single update, *stochastic* gradient descent (SGD), where a **single** example is used to make a weight update and *mini-batch* gradient descent (MBGD), where a batch (not to be confused with BGD) of examples is randomly sampled from the entire training set and is used to make an update. *Mini-batch GD is nowadays commonly referred to as Stochastic GD.*

It is a common practise to use batch sizes of powers of 2 to offer better run-time with certain hardware. Small batches tend to have a regularizing effect because of the noise they inject as each update is made by seeing only a very small portion of the entire training set.

- The mini-batches should be selected randomly. It is sufficient to shuffle the dataset once and iterate over it multiple times. In the first epoch, the network sees each example for the first time and hence, the estimate of gradient is an *unbiased* estimate of the gradient of the true generalization error. However, from the second epoch onward, the estimate becomes biased as it is re-sampling from data that it has already seen. Such a sampling algorithm is called Random Reshuffling

and although their analysis even for generalized linear models, which are strongly convex, is an open problem till date, reasonable efforts have been made to show that this biased estimate of the gradient is decent enough.

3.4.2 Challenges in Neural Network Optimization:

Optimization in general is an extremely difficult task. Traditionally, machine learning has avoided the difficulty of general optimization by carefully designing the objective function and constraints to ensure that the optimization problem is convex. When training neural networks, we must confront the general non-convex case. Even convex optimization is not without its complications. The optimization problem for training neural networks is generally non-convex. Some of the challenges faced are mentioned below:

3.4.2.1 Ill-conditioning of the Hessian Matrix:

Some challenges arise even when optimizing convex functions. Of these, the most prominent is ill-conditioning of the Hessian matrix H. This is a very general problem in most numerical optimization. For the sake of completion, the Hessian matrix **H** of a function f with a vector-valued input $\boldsymbol{x}$ is given as:

$$(\mathbf{H}_f)_{i,j} = \frac{\partial^2 f}{\partial x_i \, \partial x_j}.$$

$$\mathbf{H}_f = \begin{bmatrix} \dfrac{\partial^2 f}{\partial x_1^2} & \dfrac{\partial^2 f}{\partial x_1 \, \partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_1 \, \partial x_n} \\ \dfrac{\partial^2 f}{\partial x_2 \, \partial x_1} & \dfrac{\partial^2 f}{\partial x_2^2} & \cdots & \dfrac{\partial^2 f}{\partial x_2 \, \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{\partial^2 f}{\partial x_n \, \partial x_1} & \dfrac{\partial^2 f}{\partial x_n \, \partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_n^2} \end{bmatrix},$$

Ill-conditioning: is said to happen when the first term exceeds the second term as then the cost would be increasing. In many cases, the gradient might be large leading to a large gradient norm (i.e. **g'g**). However, **g'Hg** might be even larger than the gradient norm. This would lead to slower learning as we would need to reduce the learning rate to make the first term lower than the second. To clarify more on this, since the first term contains the 2nd power of ϵ, and ϵ being less than 1, $\epsilon^2 < \epsilon$. So, to prevent ill-conditioning, the first term should be lower than the second, but

given that **g'Hg > g'g,** this can be achieved only by reducing the learning rate, leading to slower learning. Thus, although ideally the gradient norm should decrease during training (since our primary aim is to reach a global minima where the gradient is 0), we can still get successful training even with the gradient norm increasing as shown below:

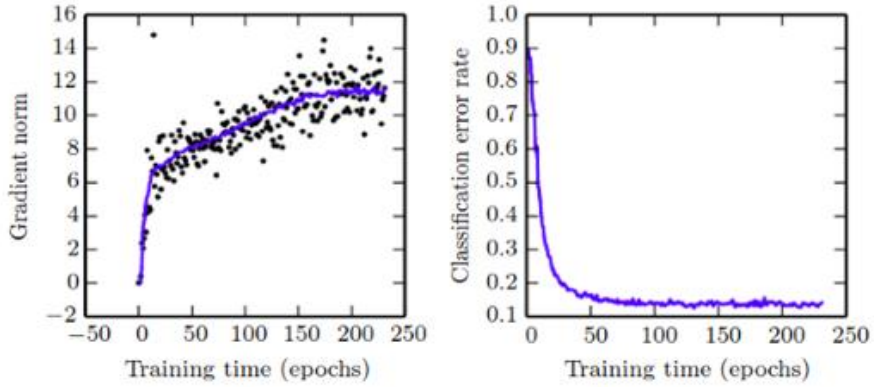


Fig 14 Gradient norm

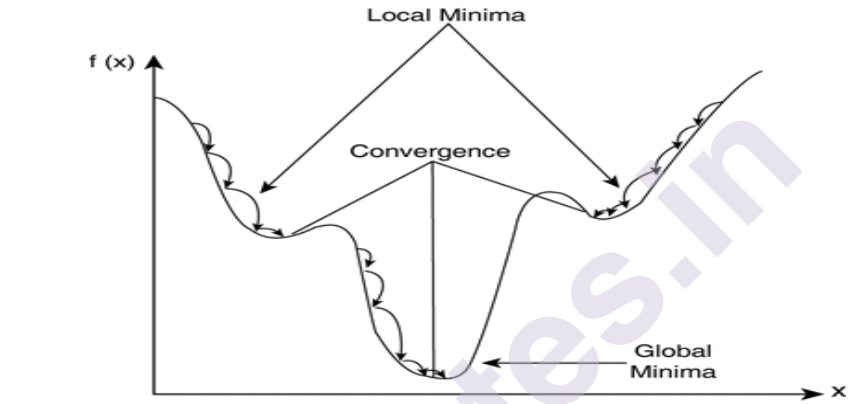
Figure 14 Gradient descent often does not arrive at a critical point of any kind. In this example, the gradient norm increases throughout training of a convolutional network used for object detection. (Left)A scatterplot showing how the norms of individual gradient evaluations are distributed over time. To improve legibility, only one gradient norm is plotted per epoch. The running average of all gradient norms is plotted as a solid curve. The gradient norm clearly increases over time, rather than decreasing as we would expect if the training process converged to a critical point. (Right)Despite the increasing gradient, the training process is reasonably successful. The validation set classification error decreases to a low level.

3.4.2.2 Local minima:

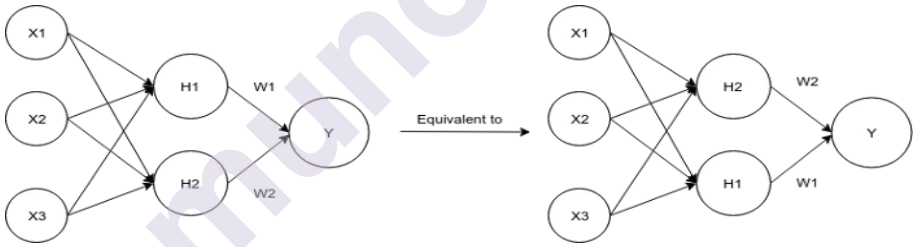
One of the most prominent features of a convex optimization problem is that it can be reduced to the problem of finding a local minimum. Any local minimum is guaranteed to be a global minimum. Some convex functions have a flat region at the bottom rather than a single global minimum point, but any point within such a flat region is an acceptable solution. When optimizing a convex function, we know that we have reached a good solution if we find a critical point of any kind. With non-convex functions, such as neural nets, it is possible to have many local minima. Indeed, nearly any deep model is essentially guaranteed to have an extremely large number of local minima. However, as we will see, this is not necessarily a major problem. Neural networks and any models with multiple equivalently parametrized latent variables all have multiple local minima because of the model identifiability problem. A model is said to be identifiable if a sufficiently large training set can rule out all but one setting of the model's parameters. Models with latent variables are often not identifiable because we can obtain equivalent models by exchanging latent

variables with each other. For example, we could take a neural network and modify layer 1 by swapping the incoming weight vector for unit i with the incoming weight vector for unit j, then doing the same for the outgoing weight vectors. If we have m layers with n units each, then there are n!m ways of arranging the hidden units. This kind of non-identifiability is known as weight space symmetry. Nearly any Deep Learning (DL) model is guaranteed to have an extremely large number of local minima (LM) arising due to the *model identifiability* problem.

A model is said to be *identifiable* if a *sufficiently large* training set can rule out *all but one* setting of the model parameters. In case of neural networks, we can obtain equivalent models by swapping the position of the neurons. Thus, they are not identifiable.



(a)



(b)

Fig. 15 Local Minima

Swapping the two hidden nodes leads to equivalent models. Thus, even after having a sufficiently large training set, there is not a unique setting of parameters. This is the model identifiability problem that neural networks suffer from.

However, all the local minima caused due to this have the same value of the cost function, thus not being a problem. However, if local minima with high cost are common, it becomes a serious problem as shown above. Many points other than local minima can lead to low gradients. Nowadays, it's common to aim for a *low but not minimal* cost value.

3.4.2.3 Plateaus, Saddle Points and Other Flat Regions:

Saddle point (SP) is another type of point with zero gradient where some points around it have higher value and the others have lower. Intuitively, this means that a saddle point acts as both a local minima for some neighbors and a local maxima for the others. Thus, Hessian at SP has both positive and negative eigenvalues for a function to curve upwards or downwards around a point as in the case of local minima and local maxima, the eigenvalues should have the same sign, positive for local minima and negative for local maxima.

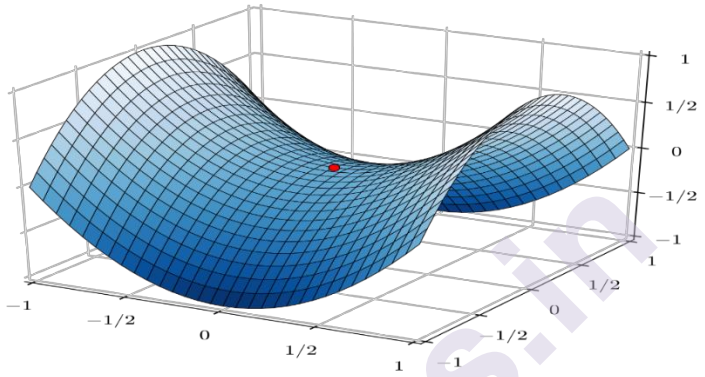


Fig. 16 Saddle Point

It's called a saddle point as it looks like the saddle of a horse. For many classes of random functions, saddle points *become more common* at high dimensions with the ratio of number of SPs to LMs growing exponentially with *n* for an n-dimensional space. Many random functions have an amazing property that near points with low cost, the Hessian tends to take up mostly positive eigenvalues. SGD empirically tends to rapidly avoid encountering a high-cost saddle point.

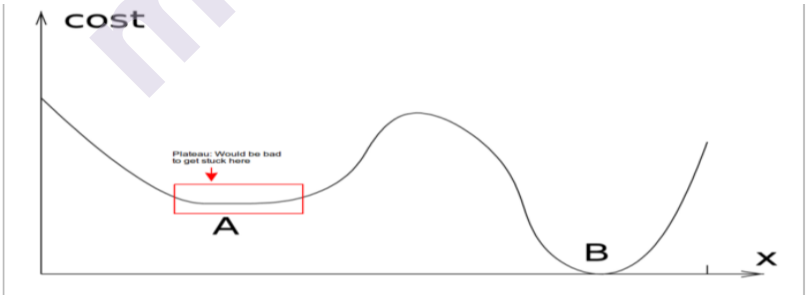


Fig 17 Position of Plateau

It is problematic to get stuck in a plateau where the value of the cost function is high.

- **Cliffs and Exploding Gradients**: Neural Networks (NNs) might sometimes have extremely steep regions resembling cliffs due to the repeated multiplication of weights. Suppose we use a 3-layer (input-hidden-output) neural network with all the activation functions as

linear. We choose the same number of input, hidden and output neurons, thus, using the same weight W for each layer. The output layer y = W*h where h = W*x represents the hidden layer, finally giving y = W*W x. So, deep neural networks involve multiplication of a large number of parameters leading to sharp non-linearities in the parameter space. These non-linearities give rise to high gradients in some places. At the edge of such a cliff, an update step might throw the parameters extremely far.

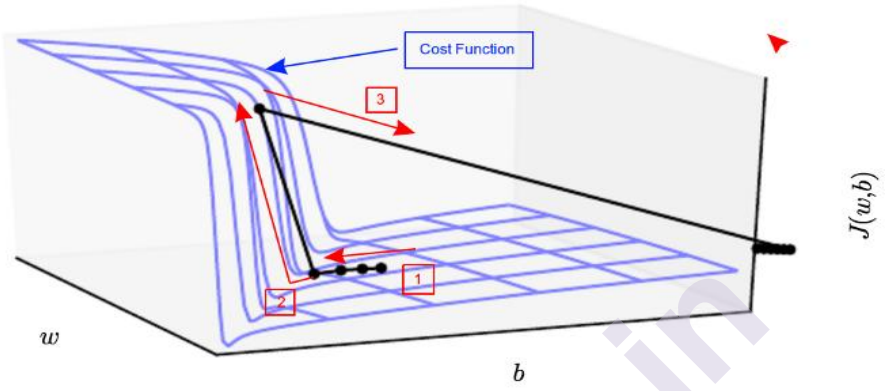


Fig. 18 Cliffs and Exploding Gradients

Image depicting the problem of exploding gradients when approaching a cliff. 1) Usual training going on with the parameters moving towards the lower cost region. 2) The gradient at the bottom left-most point pointed downwards (correct direction) but the step-size was too large, which caused the parameters to land at a point having large cost value. 3) The gradient at this new point moved the parameters in a completely different position undoing most of the training done until that point.

It can be taken care of by using **Gradient Clipping (GC)**. The gradient indicates only the direction in which to make the update. If the GD update proposes making a very large step, GC intervenes to reduce the step size.

- **Long-Term Dependencies**: This problem is encountered when the NN becomes sufficiently deep. For example, if the same weight matrix W is used in each layer, after *t* steps, we'd get W *W * W … (*t* times). Using the eigendecomposition of W:

$$\mathbf{W} = \mathbf{V} diag(\lambda) \mathbf{V}^T$$
$$\mathbf{W}^t = \mathbf{V} diag(\lambda)^t \mathbf{V}^T$$

Here, **V** is an orthonormal matrix, i.e. V V' = I

Thus, any eigenvalues not near an absolute value of one would either explode or vanish leading to the *Vanishing and Exploding Gradient* problem. The use of the same weight matrix is especially the case in Recurrent NNs (RNNs), where this is a serious problem.

- **Inexact Gradients**: Most optimization algorithms use a noisy/biased estimate of the gradient in cases where the estimate is based on sampling, or in cases where the true gradient is intractable for e.g. in the case of training a *Restricted Boltzmann Machine* (RBM), an approximation of the gradient is used. For RBM, the *contrastive divergence* algorithm gives a technique for approximating the gradient of its intractable log-likelihood.

- Neural Networks might not end up at any critical point at all and such critical points might not even necessarily exist. A lot of the problems might be avoided if there exists a space connected reasonably directly to the solution by a path that local descent can follow and if we are able to initialize learning within that well-behaved space. Thus, choosing good initial points should be studied.

Stochastic Gradient Descent:

This has already been described before but there are certain things that should be kept in mind regarding SGD. The learning rate ϵ is a very important parameter for SGD. ϵ should be reduced after each epoch in general. This is due to the fact that the random sampling of batches acts as a source of noise which might make SGD keep oscillating around the minima without actually reaching it. This is shown below:

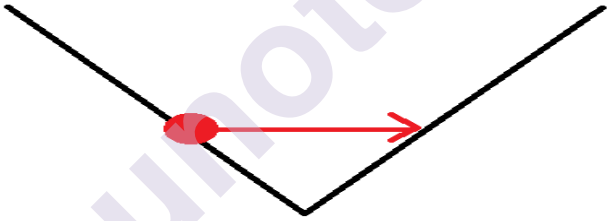


Fig. 19 Stochastic Gradient Descent

The true gradient of the total cost function (involving the entire dataset) *actually becomes 0* when we reach the minimum. Hence, BGD can use a fixed learning rate. The following conditions guarantee convergence under convexity assumptions in case of SGD:

$$\sum_{k=1}^{\infty}(\epsilon_k) = \infty$$

$$\sum_{k=1}^{\infty}(\epsilon\, k^2) = \infty$$

Setting it too low makes the training proceed slowly which might lead to the algorithm being stuck at a high cost value. Setting it too high would lead to large oscillations which might even push the learning outside the optimal region. The best way is to monitor the first several iterations and

set the learning rate to be higher than the best performing one, but not too high to cause instability.

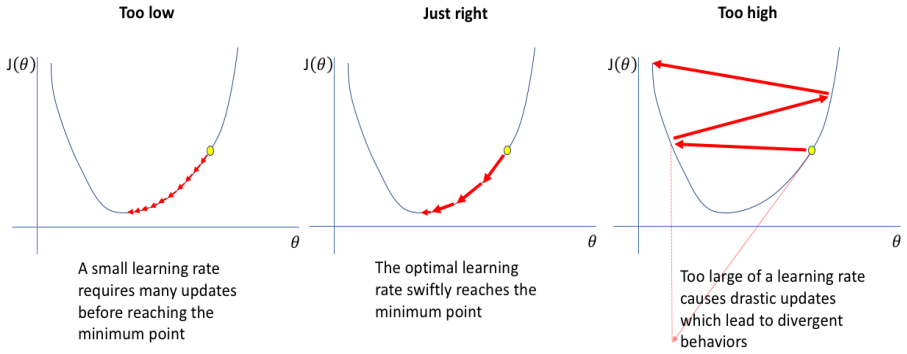


Fig. 20 Learning Rate

A big advantage of SGD is that the time taken to compute a weight update doesn't grow with the number of training examples as each update is computed after observing a batch of samples which is independent of the total number of training examples. Theoretically, for a convex problem, BGD makes the error rate **O(1/k)** after *k* iterations whereas SGD makes it **O(1/√k).** However, SGD compensates for this with its advantages after a few iterations along with the ability to make rapid updates in the case of a large training set.

- **Momentum**: The momentum algorithm accumulates the exponentially decaying moving average of past gradients (called as **velocity**) and uses it as the direction in which to take the next step. Momentum is given by mass times velocity, which is *equal* to velocity if we're using unit mass. The momentum update is given by:

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L\left(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)} \right) \right)$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}.$$

Momentum weight update

The step size (earlier equal to learning rate * gradient) now depends on how *large* and *aligned* the *sequence of gradients* are. If the gradients at each iteration point in the same direction (say ***g***), it will lead to a higher value of the step size as they just keep accumulating. Once it reaches a constant (terminal) velocity, the step size becomes **ϵ || g|| / (1 — α)**. Thus, using α as 0.9 makes the speed 10 times. Common values of α are 0.5, 0.9 and 0.99.

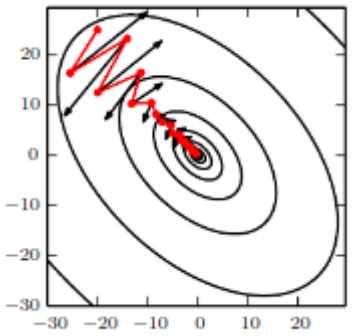


Fig. 21 Momentum

Momentum aims primarily to solve two problems: poor conditioning of the Hessian matrix and variance in the stochastic gradient. Here, we illustrate how momentum overcomes the first of these two problems. The contour lines depict a quadratic loss function with a poorly conditioned Hessian matrix. The red path cutting across the contours indicates the path followed by the momentum learning rule as it minimizes this function. At each step along the way, we draw an arrow indicating the step that gradient descent would take at that point. We can see that a poorly conditioned quadratic objective looks like a long, narrow valley or canyon with steep sides. Momentum correctly traverses the canyon lengthwise, while gradient steps waste time moving back and forth across the narrow axis of the canyon.
Viewing it as the Newtonian dynamics of a particle sliding down a hill, the momentum algorithm consists of solving a set of differential equations via numerical simulation. There are two kinds of forces involved as shown below:

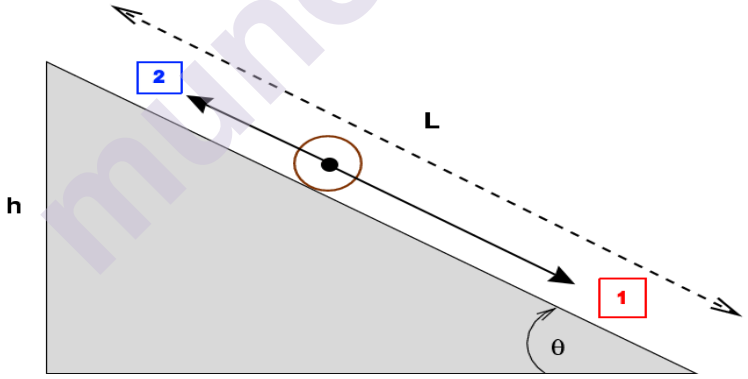


Fig. 22 Momentum (forces)

Momentum can be seen as two forces operating together. 1) Proportional to the negative of the gradient such that whenever it descends a steep part of the surface, it gathers speed and continues sliding in that direction until it goes uphill again. 2) A viscous drag force (friction) proportional to **- v(t)** without the presence of which the particle would keep oscillating back and forth as the negative of the gradient would keep forcing it to move downhill . Viscous force is suitable as it is weak enough to allow the gradient to cause motion and strong enough to resist any motion if the gradient doesn't justify moving

- **Nesterov Momentum**: This is a slight modification of the usual momentum equation. Here, the gradient is calculated after applying the current velocity to the parameters, which can be viewed as adding a correction factor:

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \nabla_{\boldsymbol{\theta}} \left[\frac{1}{m} \sum_{i=1}^{m} L\left(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta} + \alpha \boldsymbol{v}), \boldsymbol{y}^{(i)} \right) \right]$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v},$$

Nesterov momentum weight update

The intuition behind Nesterov momentum is that upon being at a point $\boldsymbol{\theta}$ in the parameter space, the momentum update is going to shift the point by $\boldsymbol{\alpha v}$. So, we are soon going to end up in the vicinity of $\boldsymbol{(\theta + \alpha v)}$. Thus, it might be better to compute the gradient from that point onward. The figure below describes this visually:

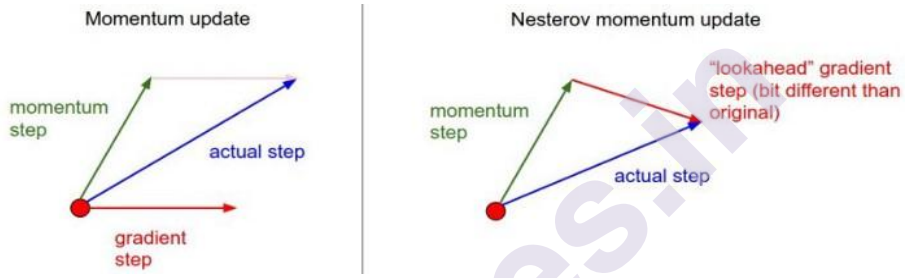


Fig. 23 Nesterov momentum weight update

3.5 SUMMARY

Feedforward networks continue to have unfulfilled potential. In the future, we expect they will be applied to many more tasks, and that advances in optimization algorithms and model design will improve their performance even further. In This unit, firstly described the neural network family of models. This module introduced the basic concepts of generalization, underfitting, overfitting, bias, variance and regularization. In Second part, we describe regularization in more detail, focusing on regularization strategies for deep models or models that may be used as building blocks to form deep models. In second part described most of the general strategies used to regularize neural networks. In third part begin with a description of how optimization used as a training algorithm for a machine learning task differs from pure optimization. We then define several practical algorithms, including both optimization algorithms themselves and strategies for initializing the parameters. We have now described the basic family of neural network models and how to regularize and optimize them. In the chapters ahead, we turn to specializations of the neural network family, that allow neural networks to scale to very large sizes and process input data that has special structure. The optimization methods discussed in this chapter are often directly applicable to these specialized architectures with little or no modification.

3.6 LIST OF REFERENCES

- Ackley, D. H., Hinton, G. E., and Sejnowski, T. J. (1985). A learning algorithm for Boltzmann machines. Cognitive Science, 9, 147–169.
- Alain, G. and Bengio, Y. (2013). What regularized auto-encoders learn from the data generating distribution. In ICLR'2013, arXiv:1211.4246.
- Baldi, P. and Hornik, K. (1989). Neural networks and principal component analysis: Learning from examples without local minima. Neural Networks, 2, 53–58.
- Bayer, J. and Osendorfer, C. (2014). Learning stochastic recurrent networks. ArXiv e-prints.
- Bengio, Y., Lamblin, P., Popovici, D., and Larochelle, H. (2007). Greedy layer-wise training of deep networks. In NIPS'2006 .
- Chapelle, O., Weston, J., and Schölkopf, B. (2003). Cluster kernels for semi-supervised learning. In S. Becker, S. Thrun, and K. Obermayer, editors, Advances in Neural Information Processing Systems 15 (NIPS'02), pages 585–592, Cambridge, MA. MIT Press.
- Choromanska, A., Henaff, M., Mathieu, M., Arous, G. B., and LeCun, Y. (2014). The loss surface of multilayer networks.
- Dauphin, Y., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., and Bengio, Y. (2014). Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In NIPS'2014.
- Desjardins, G., Simonyan, K., Pascanu, R., et al. (2015). Natural neural networks. In Advances in Neural Information Processing Systems , pages 2062–2070. 320
- E. Aljalbout, V. Golkov, Y. Siddiqui, and D. Cremers. Clustering with deep learning: Taxonomy and new methods. arXiv:1801.07648, 2018. https://arxiv.org/abs/1801.07648
- R. Al-Rfou, B. Perozzi, and S. Skiena. Polyglot: Distributed word representations for multilingual nlp. arXiv:1307.1662, 2013. https://arxiv.org/abs/1307.1662

UNIT END EXERCISES

- Define and explain Deep Networks with example?
- Describe Deep feedforward network with its types.
- What Simple Deep Neural Network? Explain with Example.
- How to compute Deep Neural Network. Explain.
- Explain Gradient-Based Learning?
- Define Regularizaion wit example?
- Compare L1 Regularization and L2 Regularization?
- Define Underfitting and overfitting.
- What is Dropout. Explain in details?
- Define and Explain Data Augmentation.
- Explain Local Minima with Diagram.

- Explain Momentum. Give details.
- Define Plateaus, Saddle Points and Other Flat Regions. Explain with Diagram.
- Explain Challenges in Neural Network Optimization.

BIBLIOGRAPHY

- Goodfellow, Ian; Bengio, Yoshua; Courville, Aaron (2016). Deep Learning. MIT Press. ISBN 978-0-26203561-3. Archived from the original on 2016-04-16. Retrieved 2021-05-09, introductory textbook.
- Charu C. Aggarwa, Neural Networks and Deep Learning, ISBN 978-3-319-94462-3 IBM T. J. Watson Research Center, International Business Machines, Yorktown Heights, NY, USA.
- R. Ahuja, T. Magnanti, and J. Orlin. Network flows: Theory, algorithms, and applications. Prentice Hall, 1993.

*****

UNIT III

4

CONVOLUTIONAL NEURAL NETWORK

Unit Structure

4.0 OBJECTIVES:

In this chapter the student will learn about:

- Convolution concept
- Convolution Operations
- Convents over Feed-Forward Neural Nets
- Examples
- Applications

4.1 INTRODUCTION

Artificial Intelligence has been witnessing a monumental growth in bridging the gap between the capabilities of humans and machines. Researchers and enthusiasts alike, work on numerous aspects of the field to make amazing things happen. One of many such areas is the domain of Computer Vision. A **Convolutional Neural Network (ConvNet/CNN)** is a Deep Learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from the other. The pre-processing required in a ConvNet is much lower as compared to other classification algorithms. While in primitive methods filters are hand-engineered, with enough training, ConvNets can learn these filters/characteristics.

4.2 WHAT IS CONVOLUTIONAL NEURAL NETWORK?

Convolutional Neural Network is one of the main categories to do image classification and image recognition in neural networks. Scene labelling, objects detections, and face recognition, etc., are some of the areas where convolutional neural networks are widely used.

As shown in fig. 4.1, CNN takes an image as input, which is classified and process under a certain category such as car, truck, van, etc. The computer sees an image as an array of pixels and depends on the resolution of the image. Based on image resolution, it will see as **h * w * d**, where h= height w= width and d= dimension. For example, An RGB image is **6 * 6 * 3** array of the matrix, and the grayscale image is **4 * 4 * 1** array of the matrix.

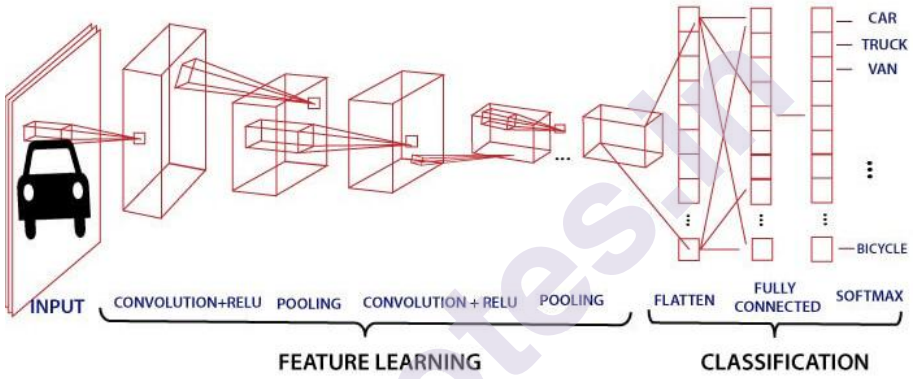


Fig. 4.1: Convolution Neural Network

In CNN, each input image will pass through a sequence of convolution layers along with pooling, fully connected layers, filters (Also known as kernels). After that, we will apply the Soft-max function to classify an object with probabilistic values 0 and 1.

4.2.1 Convolution Layer:

Convolution layer is the first layer to extract features from an input image. By learning image features using a small square of input data, the convolutional layer preserves the relationship between pixels. It is a mathematical operation which takes two inputs such as image matrix and a kernel or filter.

- The dimension of the image matrix is **h×w×d**.
- The dimension of the filter is $\mathbf{f_h \times f_w \times d}$.
- The dimension of the output is $\mathbf{(h-f_h+1)\times(w-f_w+1)\times 1}$.

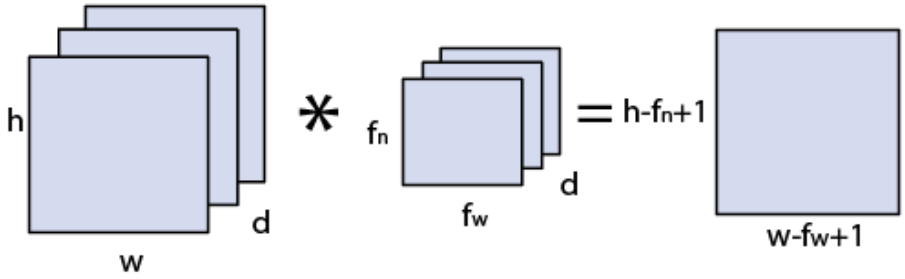


Image matrix multiplies kernl or filter matrix

Let's start with consideration a 5*5 image whose pixel values are 0, 1, and filter matrix 3*3 as:

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

5 × 5 – Image Matrix 3 × 3 – Filter Matrix

The convolution of 5*5 image matrix multiplies with 3*3 filter matrix is called "**Features Map**" and show as an output.

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 4 & 3 & 4 \\ 2 & 4 & 3 \\ 2 & 3 & 4 \end{bmatrix}$$

Convolved Feature

Convolution of an image with different filters can perform an operation such as blur, sharpen, and edge detection by applying filters.

4.2.2 STRIDES

Stride is the number of pixels which are shift over the input matrix. When the stride is equalled to 1, then we move the filters to 1 pixel at a time and similarly, if the stride is equalled to 2, then we move the filters to 2 pixels at a time. The following figure shows that the convolution would work with a stride of 2.

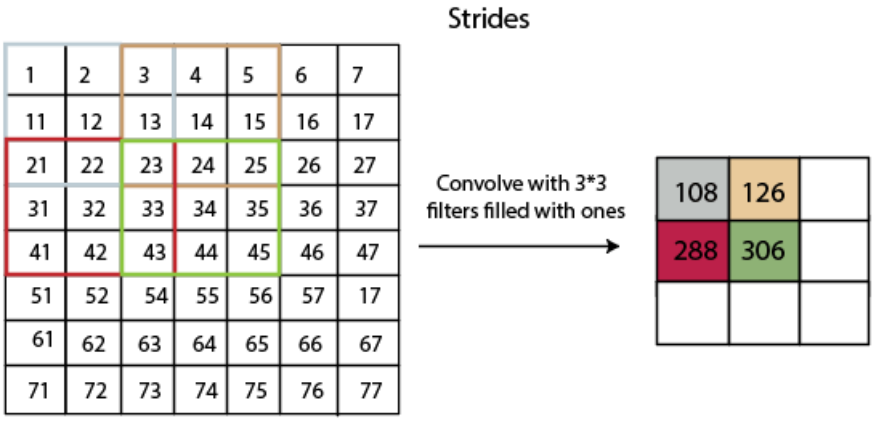


Fig. 4.2: Convolutional Strides

4.2.3 Padding

Padding plays a crucial role in building the convolutional neural network. If the image will get shrink and if we will take a neural network with 100's of layers on it, it will give us a small image after filtered in the end.
If we take a three by three filter on top of a grayscale image and do the convolving then what will happen?

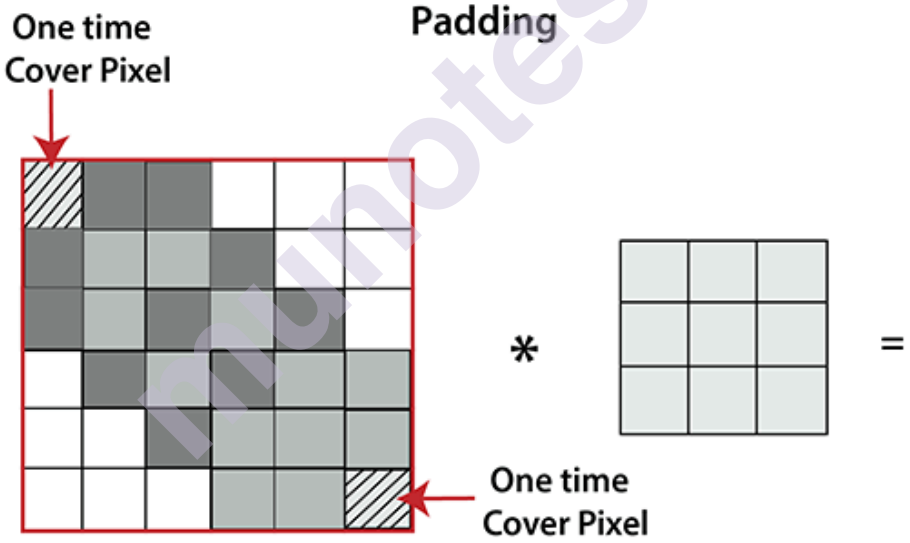


Fig. 4.3: Convolutional Padding

It is clear from the above picture that the pixel in the corner will only get covers one time, but the middle pixel will get covered more than once. It means that we have more information on that middle pixel, so there are two downsides:

- Shrinking outputs
- Losing information on the corner of the image.

To overcome this, we have introduced padding to an image. **"Padding is an additional layer which can add to the border of an image."**

4.3 WHY CONVNETS OVER FEED-FORWARD NEURAL NETS?

The architecture of a ConvNet is inspired by the organisation of the Visual Cortex and is akin to the connectivity pattern of Neurons in the Human Brain. Individual neurons can only respond to stimuli in a small area of the visual field called the Receptive Field. A group of similar fields will encompass the full visual region if they overlap.

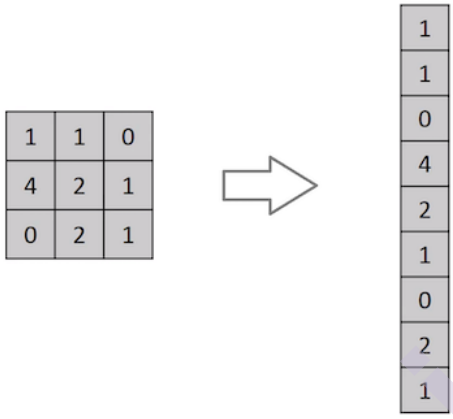


Fig 4.4: Flattening of a 3x3 image matrix into a 9x1 vector

An image is nothing, but a matrix of pixel values as shown in fig. 4.4. In cases of extremely basic binary images, the method might show an average precision score while performing prediction of classes but would have little to no accuracy when it comes to complex images having pixel dependencies throughout.

A ConvNet is able to **successfully capture the Spatial and Temporal dependencies** in an image through the application of relevant filters. The architecture performs a better fitting to the image dataset due to the reduction in the number of parameters involved and reusability of weights. In other words, the network can be trained to understand the sophistication of the image better.

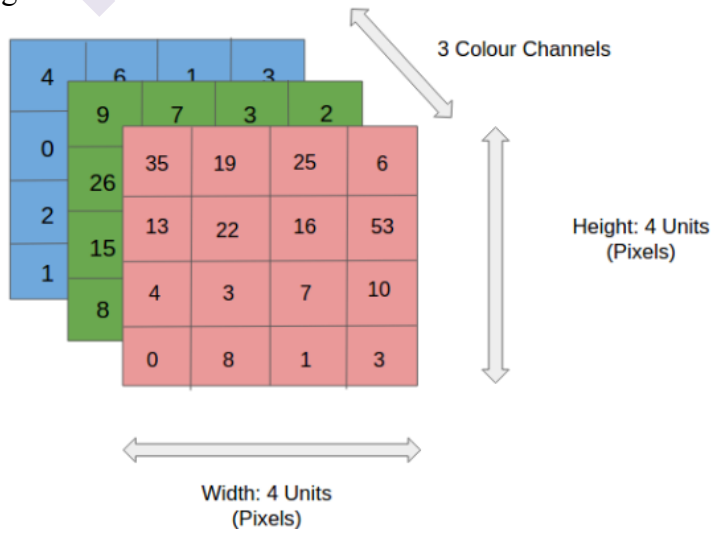


Fig 4.5: 4x4x3 RGB Image

In the figure 4.5, there is an RGB image which has been separated by its three color planes — Red, Green, and Blue. There are several such color spaces in which images exist — Grayscale, RGB, HSV, CMYK, etc. You can imagine how computationally hard things will get once the photos exceed 8K (76804320) dimensions. The ConvNet's job is to compress the images into a format that is easier to process while preserving elements that are important for obtaining a decent prediction. This is critical for designing an architecture that is capable of learning features while also being scalable to large datasets.

4.4 CONVOLUTIONAL OPERATION

Convolution is a specialized kind of linear operation. Convolution is an operation on two functions of a real- valued argument. To motivate the definition of convolution, we start with examples of two functions we might use. Convnets are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers.

4.4.1 Convolution Kernels:

A kernel is a small 2D matrix whose contents are based upon the operations to be performed. A kernel maps on the input image by simple matrix multiplication and addition, the output obtained is of lower dimensions and therefore easier to work with.

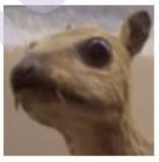
Original	*Gaussian Blur*	*Sharpen*	*Edge Detection*
$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$
			

Fig 4.6: Kernel types

To smoothen the image before processing, Sharpen image(enhance the depth of edges) and edge detection the above example shown in fig 4.6. The shape of a kernel is heavily dependent on the input shape of the image and architecture of the entire network, mostly the size of kernels is **(MxM)** i.e a square matrix. The movement of a kernel is always from left to right and top to bottom.

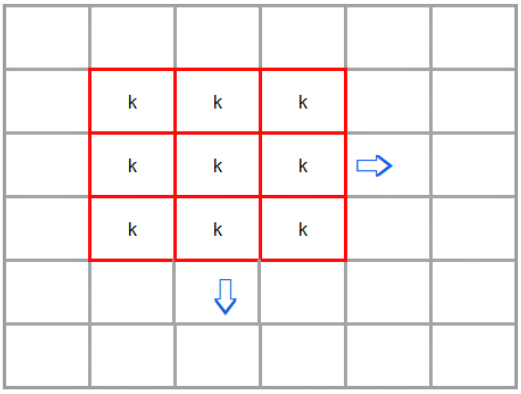
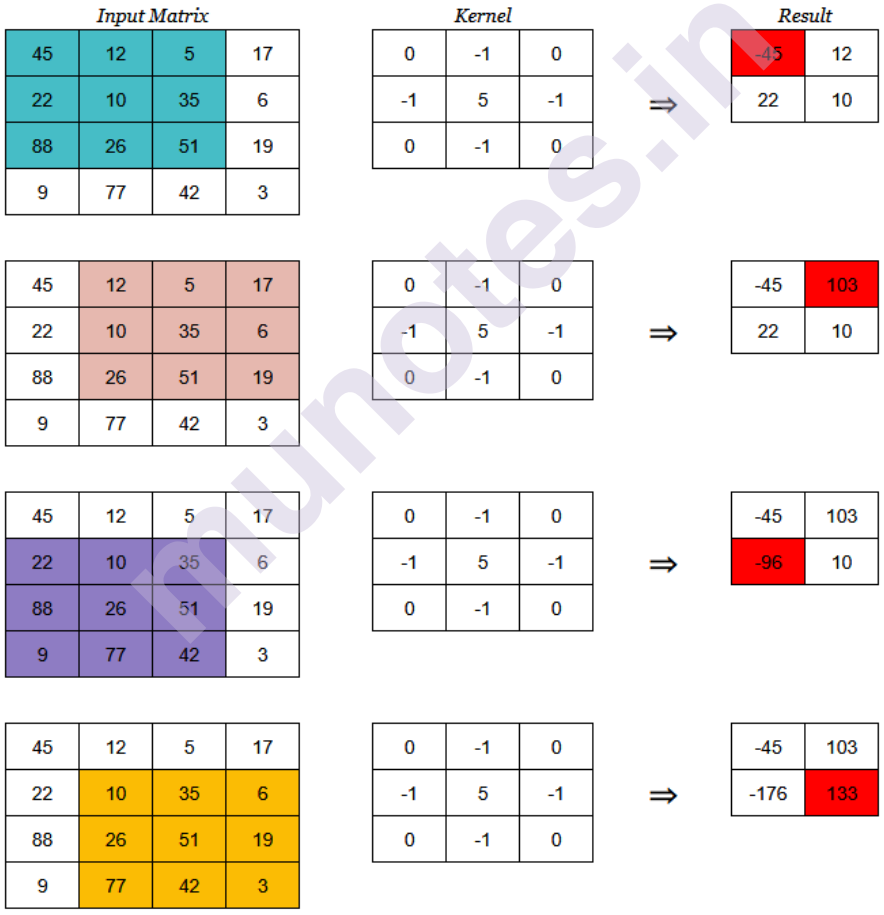


Fig. 4.7: Kernel Movement

As discussed above the stride defines for example, a stride of 1 causes the kernel to slide by one row/column at a time, whereas a stride of 2 causes the kernel to slide by two rows/columns.



the input matrix has shape 4x4x1 and the kernel is of size 3x3 since the shape of input is larger than the kernel, we are able to implement a sliding window protocol and apply the kernel over entire input. First entry in the convoluted result is calculated as:

*45*0 + 12*(-1) + 5*0 + 22*(-1) + 10*5 + 35*(-1) + 88*0 + 26*(-1) + 51*0 = -45*

4.4.2 Sliding window protocol:

1. The kernel gets into position at the top-left corner of the input matrix.
2. Then it starts moving left to right, calculating the dot product and saving it to a new matrix until it has reached the last column.
3. Next, kernel resets its position at first column but now it slides one row to the bottom. Thus following the fashion left-right and top-bottom.
4. Steps 2 and 3 are repeated till the entire input has been processed.

The kernel will move from front to back, left to right, and top to bottom for a 3D input matrix. You should have a fundamental knowledge of the Convolution operation, which is the heart of a Convolutional Neural Network by now.

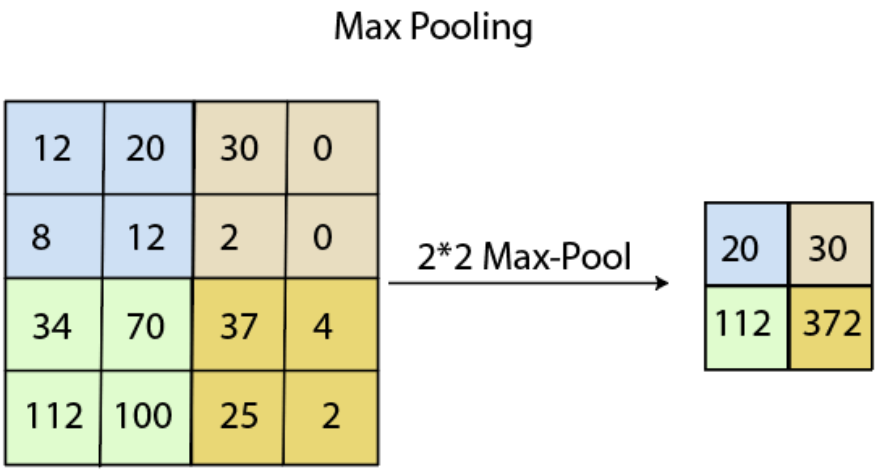
4.5 POOLING

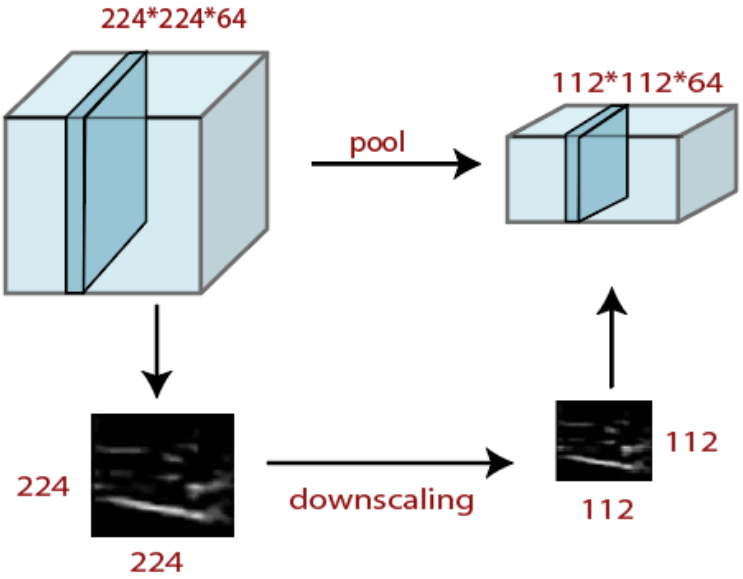
Pooling layer plays an important role in pre-processing of an image. Pooling layer reduces the number of parameters when the images are too large. Pooling is "**downscaling**" of the image obtained from the previous layers. It can be compared to shrinking an image to reduce its pixel density. Spatial pooling is also called downsampling or subsampling, which reduces the dimensionality of each map but retains the important information. There are the following types of spatial pooling:

Max Pooling:

Max pooling is a **sample-based discretization process**. Its main objective is to downscale an input representation, reducing its dimensionality and allowing for the assumption to be made about features contained in the sub-region binned.

Max pooling is done by applying a max filter to non-overlapping sub-regions of the initial representation.





Average Pooling:

Down-scaling will perform through average pooling by dividing the input into rectangular pooling regions and computing the average values of each region.

layer = averagePooling2dLayer(poolSize)

layer = averagePooling2dLayer(poolSize,Name,Value)

Sum Pooling:

The sub-region for **sum pooling** or **mean pooling** are set exactly the same as for **max-pooling** but instead of using the max function we use sum or mean.

4.6 DATA TYPES

The data used with a convolutional network usually consist of several channels, each channel being the observation of a different quantity at some point in space or time. One advantage to convolutional networks is that they can also process inputs with varying spatial extents. These kinds of input simply cannot be represented by traditional, matrix multiplication-based neural networks. This provides a compelling reason to use convolutional networks even when computational cost and overfitting are not significant issues. For example, consider a collection of images in which each image has a different width and height. It is unclear how to model such inputs with a weight matrix of fixed size. Convolution is straightforward to apply; the kernel is simply applied a different number of times depending on the size of the input, and the output of the convolution operation scales accordingly. Convolution may be viewed as matrix multiplication; the same convolution kernel induces a different size of

doubly block circulant matrix for each size of input. Sometimes the output of the network as well as the input is allowed to have variable size, for example, if we want to assign a class label to each pixel of the input. In this case, no further design work is necessary. In other cases, the network must produce some fixed-size output, for example, if we want to assign a single class label to the entire image. In this case, we must make some additional design steps, like inserting a pooling layer whose pooling regions scale in size proportional to the size of the input, to maintain a fixed number of pooled outputs.

	Single channel	**Multichannel**
1-D	Audio waveform: The axis we convolve over corresponds to time. We discretize time and measure the amplitude of the waveform once per time step.	Skeleton animation data: Animations of 3-D computer-rendered characters are generated by altering the pose of a "skeleton" over time. At each point in time, the pose of the character is described by a specification of the angles of each of the joints in the character's skeleton
2-D	Audio data that has been pre-processed with a Fourier transform: We can transform the audio waveform into a 2-D tensor with different rows corresponding to different frequencies and different columns corresponding to different points in time. Using convolution in the time makes the model equivariant to shifts in time.	Color image data: One channel contains the red pixels, one the green pixels, and one the blue pixels. The convolution kernel moves over both the horizontal and the vertical axes of the image, conferring translation equivariance in both directions
3-D	Volumetric data: A common source of this kind of data is medical imaging technology, such as CT	Color video data: One axis corresponds to time, one to the height of the video frame, and one to the width of the video frame.

4.7 CONVOLUTION ALGORITHMS:

Convolution is equivalent to converting both the input and the kernel to the frequency domain using a Fourier transform, performing point-wise multiplication of the two signals, and converting back to the time domain using an inverse Fourier transform. For some problem sizes, this can be faster than the naive implementation of discrete convolution. When a d-dimensional kernel can be expressed as the outer product of d vectors, one vector per dimension, the kernel is called separable. When the kernel is

separable, naive convolution is inefficient. It is equivalent to compose d one-dimensional convolutions with each of these vectors. The composed approach is significantly faster than performing one d-dimensional convolution with their outer product. The kernel also takes fewer parameters to represent as vectors. If the kernel is w elements wide in each dimension, then naive multidimensional convolution requires O(wd) runtime and parameter storage space, while separable convolution requires O(w × d) runtime and parameter storage space.

4.8 RELATION OF CONVOLUTIONAL NETWORK WITH DEEP LEARNING

A Convolutional Neural Network (ConvNet/CNN) is a Deep Learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from the other. The pre-processing required in a ConvNet is much lower as compared to other classification algorithms. While in primitive methods filters are hand-engineered, with enough training, ConvNets can learn these filters/characteristics. The architecture of a ConvNet is analogous to that of the connectivity pattern of Neurons in the Human Brain and was inspired by the organization of the Visual Cortex. Individual neurons respond to stimuli only in a restricted region of the visual field known as the Receptive Field. A collection of such fields overlaps to cover the entire visual area.

4.9 DIFFERENCE BETWEEN CNN AND RNN

S.no	CNN	RNN
1	**CNN** stands for **Convolutional Neural Network**.	**RNN** stands for **Recurrent Neural Network**.
2	CNN is more potent than RNN.	RNN includes less feature compatibility when compared to CNN.
3	CNN is ideal for images and video processing.	RNN is ideal for text and speech Analysis.
4	It is suitable for spatial data like images.	RNN is used for temporal data, also called sequential data.
5	The network takes fixed-size inputs and generates fixed size outputs.	RNN can handle arbitrary input/ output lengths.

6	CNN is a type of feed-forward artificial neural network with variations of multilayer perceptron's designed to use minimal amounts of pre-processing.	RNN, unlike feed-forward neural networks- can use their internal memory to process arbitrary sequences of inputs.

4.10 CONCLUSION

In this chapter we learned about the fundamental concept of neural network with its application for classifying the image. A Convolutional Neural Network (ConvNet/CNN) is a Deep Learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from the other. Pooling layer reduces the number of parameters when the images are too large. Convolution is straightforward to apply; the kernel is simply applied a different number of times depending on the size of the input, and the output of the convolution operation scales accordingly. Convolution may be viewed as matrix multiplication; the same convolution kernel induces a different size of doubly block circulant matrix for each size of input. CNN is a type of feed-forward artificial neural network with variations of multilayer perceptron's designed to use minimal amounts of pre-processing. RNN, unlike feed-forward neural networks- can use their internal memory to process arbitrary sequences of inputs.

EXERCISE

1. What is convolution neural network? How it is different from neural network.
2. Explain the mechanism of convolution neural network.
3. What is Pooling? Explain the role of pooling.
4. Explain the working of MAX and Average Pooling.
4. Explain different types of data types.
6. Write a note on Convolution algorithm.
7. Give comparison between recurrent neural network and convolutional neural network.

REFERENCES

1. Ian Goodfellow, Yoshua Bengio, Aaron Courvile, Deep Learning, MIT Press, 2016
2. Nikhil Buduma, Fundamentals of Deep Learning, O'Reilly,

*****

5

SEQUENCE MODELLING

Unit Structure

5.0 OBJECTIVES

In this chapter the student will learn about:

- Recurrent neural networks
- Use of Sequence modelling
- Applications of sequence modelling

5.1 INTRODUCTION

Having a solid grasp on deep learning techniques feels like acquiring a super power these days. From classifying images and translating languages to building a self-driving car, all these tasks are being driven by computers rather than manual human effort. Deep learning has penetrated multiple and diverse industries, and it continues to break new ground on an almost weekly basis. Sequence Modelling is the task of predicting what word/letter comes next. Unlike the FNN and CNN, in sequence modelling, the current output is dependent on the previous input and the length of the input is not fixed. The ability to predict what comes next in a sequence is fascinating. Sequence models, in supervised learning, can be used to address a variety of applications including financial time series prediction, speech recognition, music generation, sentiment classification, machine translation and video activity recognition. The obvious question that always pop-ups that, Why not a standard network?. We can say that

Traditional feedforward neural networks do not share features across different positions of the network. In other words, these models assume that all inputs (and outputs) are independent of each other. This model would not work in sequence prediction since the previous inputs are inherently important in predicting the next output. For example, if you were predicting the next word in a stream of text, you would want to know at least a couple of words before the target word.

5.2 AUTO-COMPLETION

Auto-completion is a feature for predicting the rest of a query a user is typing, which can improve the user search experience and accelerate the shopping process before checkout. It can also improve the search response quality and thus create higher revenue by providing well-formatted queries.

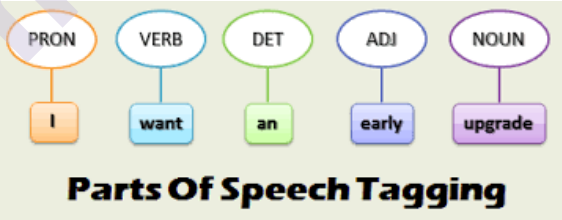
5.2.1 Parts of Speech Tagging:

Part-of-Speech tagging is a well-known task in Natural Language Processing. It refers to the process of classifying words into their parts of speech (also known as words classes or lexical categories). This is a supervised learning approach. Parts of speech tags are the properties of the words, which define their main context, functions, and usage in a sentence. Some of the commonly used parts of speech tags are

Nouns: Which defines any object or entity

Verbs: That defines some action.

Adjectives and Adverbs: This acts as a modifier, quantifier, or intensifier in any sentence.



Further, Has and purchased belong to the verb indicating that they are the actions. The Laptop and Apple store are the nouns. New is the adjective whose role is to modify the context of the laptop. Parts of speech tags are defined by the relationship of words with the other words in the sentence.

5.2.2 Sequence Classification:

Sequence classification is a predictive modeling problem where you have some sequence of inputs over space or time and the task is to predict a

category for the sequence. Few examples where sequence models are used in real-world scenarios.

Speech recognition:

Here, the input is an audio clip and the model has to produce the text transcript. The audio is considered a sequence as it plays over time. Also, the transcript is a sequence of words.

"The quick brown fox jumped over the lazy dog."

Sentiment Classification:

Another popular application of sequence models. We pass a text sentence as input and the model has to predict the sentiment of the sentence (positive, negative, angry, elated, etc.). The output can also be in the form of ratings or stars.

"There is nothing to like in this movie."

DNA sequence analysis:

Given a DNA sequence as input, we want our model to predict which part of the DNA belongs to which protein.

AGCCCCTGTGAGGAACTAG → AGCCCCTGTGAGGAACTAG

Machine Translation:

We input a sentence in one language, say French, and we want our model to convert it into another language, say English. Here, both the input and the output are sequences:

Voulez-vous chanter avec moi? → Do you want to sing with me?

Video activity recognition:

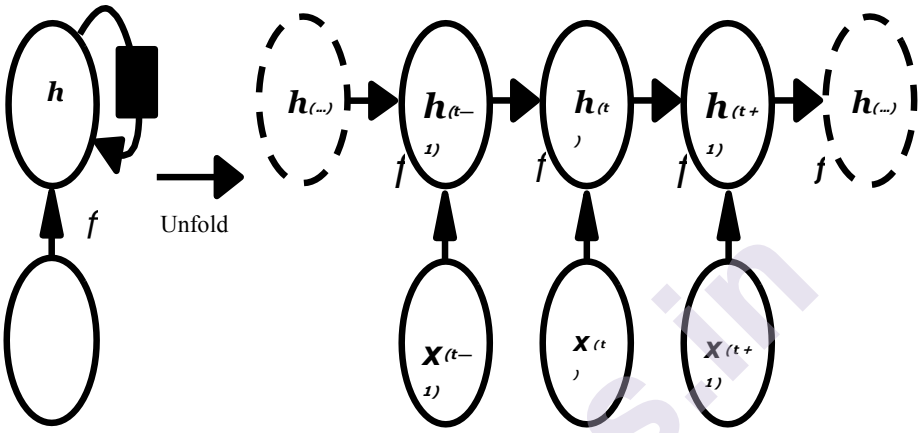
This is actually a very upcoming (and current trending) use of sequence models. The model predicts what activity is going on in a given video. Here, the input is a sequence of frames.

Running

Modelling Sequence Learning Problems

5.3 UNFOLDING COMPUTATIONAL GRAPHS

A computational graph is a way to formalize the structure of a set of computations, such as those involved in mapping inputs and parameters to outputs and loss. we explain the idea of unfolding a recursive or recurrent computation into a computational graph that has a repetitive structure, typically corresponding to a chain of events. Unfolding this graph results in the sharing of parameters across a deep network structure.



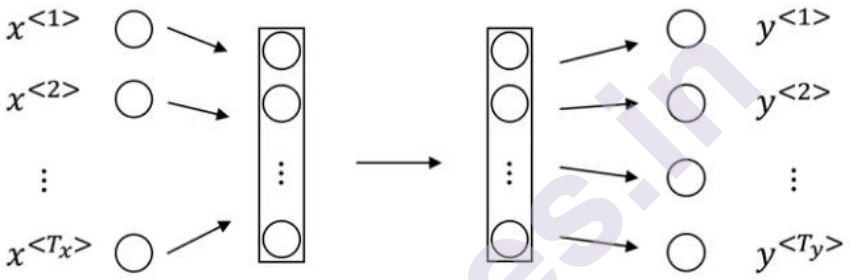
- Each node represents the state at some time t, and the function f maps the state at t to the state at t+ 1.
- The same parameters (the same value of θ used to parametrize f) are used for all time steps.
- These cycles represent the influence of the present value of a variable on its own value at a future time step. Such computational graphs allow us to define recurrent neural networks. We then describe many different ways to construct, train, and use recurrent neural networks. A computational graph is a way to formalize the structure of a set of computations, such as those involved in mapping inputs and parameters to outputs and loss.
- We explain the idea of unfolding a recursive or recurrent computation into a computational graph that has a repetitive structure, typically corresponding to a chain of events.
- Unfolding this graph results in the sharing of parameters across a deep network structure.

A recurrent network with no outputs. This recurrent network just processes information from the input x by incorporating it into the state h that is passed forward through time. (Left) Circuit diagram. The black square indicates a delay of a single time step. (Right) The same network seen as an unfolded computational graph, where each node is now associated with one particular time instance. Recurrent neural networks can be built in many diff erent ways. Much as almost any function can be considered a

feedforward neural network, essentially any function involving recurrence can be considered a recurrent neural network. When the recurrent network is trained to perform a task that requires predicting the future from the past, the network typically learns to use h(t) as a kind of lossy summary of the task-relevant aspects of the past sequence of inputs up to t.

5.4 RECURRENT NEURAL NETWORKS

Recurrent Neural Networks are used to learn mapping from X to Y, when either X or Y, or both X and Y, are some sequences. But it is not use as a standard neural network for these sequence problems? Some examples of important design patterns for recurrent neural networks include the following: Recurrent networks that produce an output at each time step and have recurrent connections between hidden units, illustrated in figure

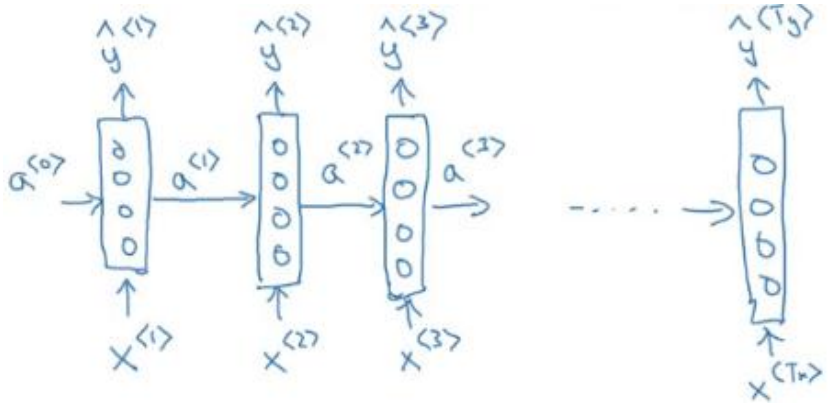


There are primarily two problems with this:

1 Inputs and outputs do not have a fixed length, i.e., some input sentences can be of 10 words while others could be <> 10. The same is true for the eventual output

2 We will not be able to share features learned across different positions of text if we use a standard neural network

We need a representation that will help us to parse through different sentence lengths as well as reduce the number of parameters in the model. This is where we use a recurrent neural network. This is how a typical RNN looks like:



A RNN takes the first word (x<1>) and feeds it into a neural network layer which predicts an output (y'<1>). This process is repeated until the last time step x<Tx> which generates the last output y'<Ty>. This is the network where the number of words in input as well as the output are same. The RNN scans through the data in a left to right sequence. Note that the parameters that the RNN uses for each time step are shared. We will have parameters shared between each input and hidden layer (Wax), every timestep (Waa) and between the hidden layer and the output (Wya). So if we are making predictions for x<3>, we will also have information about x<1> and x<2>. A potential weakness of RNN is that it only takes information from the previous timesteps and not from the ones that come later. This problem can be solved using bi-directional RNNs which we will discuss later. For now, let's look at forward propagation steps in a RNN model:

a<0> is a vector of all zeros and we calculate the further activations similar to that of a standard neural network:

- $a^{<0>} = 0$
- $a^{<1>} = g(W_{aa} * a^{<0>} + W_{ax} * x^{<1>} + b_a)$
- $y^{<1>} = g'(W_{ya} * a^{<1>} + b_y)$

Similarly, we can calculate the output at each time step. The generalized form of these formulae can be written as:

$$a^{<t>} = g(W_{aa}a^{<t>} + W_{ax}x^{<t>} + b_a)$$

$$\hat{y}^{<t>} = g(W_{ya}a^{<t>} + b_y)$$

We horizontally stack W_{aa} and W_{ya} to get W_a. $a^{<t-1>}$ and $x^{<t>}$ are stacked vertically. Rather than carrying around 2 parameter matrices, we now have just 1 matrix. And that, in a nutshell, is how forward propagation works for recurrent neural networks.

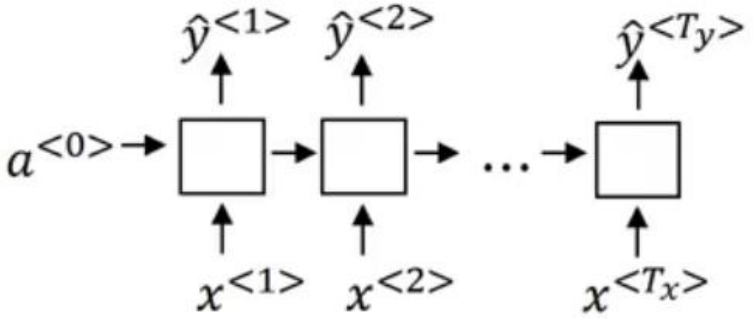
5.5 TYPES OF RNNS

We can have different types of RNNs to deal with use cases where the sequence length differs. These problems can be classified into the following categories:

Many-to-many:

The name entity recognition examples we saw earlier fall under this category. We have a sequence of words, and for each word, we have to

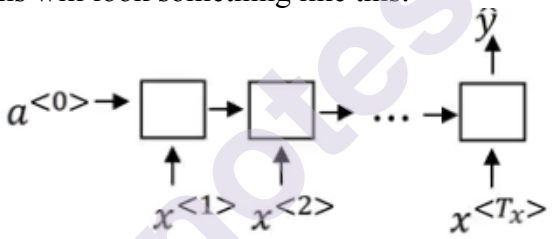
predict whether it is a name or not. The RNN architecture for such a problem looks like this:



For every input word, we predict a corresponding output word.

Many-to-one:

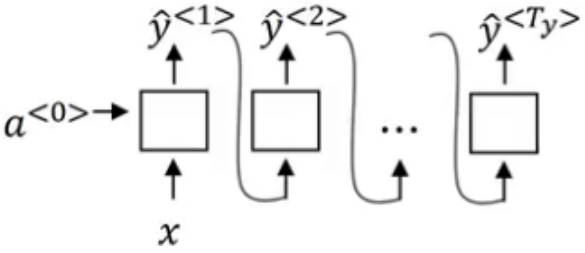
Consider the sentiment classification problem. We pass a sentence to the model and it returns the sentiment or rating corresponding to that sentence. This is a many-to-one problem where the input sequence can have varied length, whereas there will only be a single output. The RNN architecture for such problems will look something like this:



Here, we get a single output at the end of the sentence.

One-to-many:

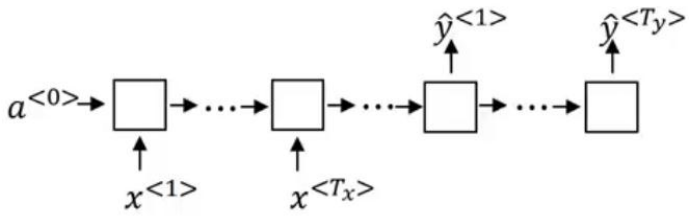
Consider the example of music generation where we want to predict the lyrics using the music as input. In such scenarios, the input is just a single word (or a single integer), and the output can be of varied length. The RNN architecture for this type of problems looks like the below:



There is one more type of RNN which is popularly used in the industry. Consider the machine translation application where we take an input sentence in one language and translate it into another language. It is a

many-to-many problem but the length of the input sequence might or might not be equal to the length of output sequence.

In such cases, we have an encoder part and a decoder part. The encoder part reads the input sentence and the decoder translates it to the output sentence:



5.6 NATURAL LANGUAGE PROCESSING AND WORD EMBEDDINGS

Natural language processing with deep learning is an important combination. Using word vector representations and embedding layers you can train recurrent neural networks with outstanding performances in a wide variety of industries. Examples of applications are sentiment analysis, named entity recognition and machine translation.

5.6.1 Introduction to Word Embeddings:

Word Representation:

- One of the weaknesses of one-hot representation is that it treats each word as a thing unto itself, and it doesn't allow an algorithm to easily generalize across words.
  - Because the any product between any two different one-hot vector is zero.
  - It doesn't know that somehow apple and orange are much more similar than king and orange or queen and orange.
- Instead we can learn a futurized representation.
  - But by a lot of the features of apple and orange are actually the same, or take on very similar values. And so, this increases the odds of the learning algorithm that has figured out that orange juice is a thing, to also quickly figure out that apple juice is a thing.
  - The features we'll end up learning, won't have a easy to interpret interpretation like that component one is gender, component two is royal, component three is age and so on. What they're representing will be a bit harder to figure out.
  - But nonetheless, the featurized representations we will learn, will allow an algorithm to quickly figure out that apple and orange are more similar than say, king and orange or queen and orange.

features\ words	Man (5391)	Woman (9853)	King (4914)	Queen (7157)	Apple (456)	Orange (6257)
Gender	-1	1	-0.95	0.97	0.00	0.01
Royal	0.01	0.02	0.93	0.95	-0.01	0.00
Age (adult?)	0.03	0.02	0.7	0.69	0.03	-0.02
Food	0.09	0.01	0.02	0.01	0.95	0.97
Size	...	...	...	...	...	...
...	...	...	...	...	...	...

Using word embeddings:

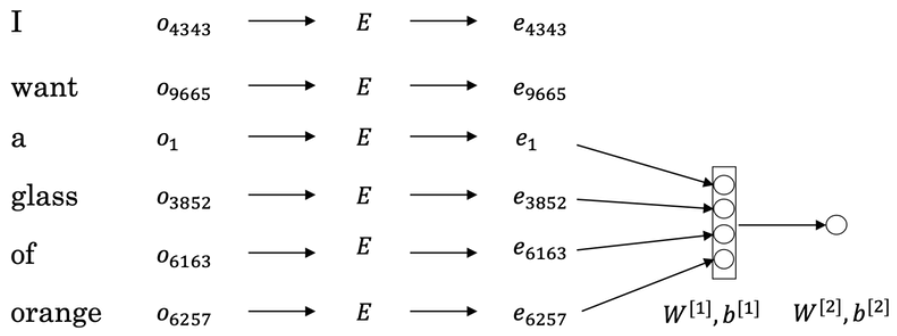
Word embeddings tend to make the biggest difference when the task you're trying to carry out has a relatively smaller training set.

- Useful for NLP standard tasks.
- Named entity recognition
- Text summarization
- Co-reference
- Parsing

5.6.2 Learning Word Embeddings: Word2vec:

Learning word embeddings:

- In the history of deep learning as applied to learning word embeddings, people actually started off with relatively complex algorithms. And then over time, researchers discovered they can use simpler and simpler and simpler algorithms and still get very good results especially for a large dataset.
- A more complex algorithm: a neural language model, by Yoshua Bengio, Rejean Ducharme, Pascals Vincent, and Christian Jauvin: A Neural Probabilistic Language Model.
    - Let's start to build a neural network to predict the next word in the sequence below.
    - I want a glass of orange _____.
    - 4343 9665 1 3852 6163 6257



- If we have a fixed historical window of 4 words (4 is a hyperparameter), then we take the four embedding vectors and stack them together, and feed them into a neural network, and then feed this neural network output to a softmax, and the softmax classifies among the 10,000 possible outputs in the vocab for the final word we're trying to predict. These two layers have their own parameters W1,b1 and W2, b2.
- This is one of the earlier and pretty successful algorithms for learning word embeddings.

- A more generalized algorithm.
  - We have a longer sentence: I want a glass of orange juice to go along with my cereal. The task is to predict the word juice in the middle.
  - If it goes to build a language model then is natural for the context to be a few words right before the target word. But if your goal isn't to learn the language model per se, then you can choose other contexts.
  - Contexts:

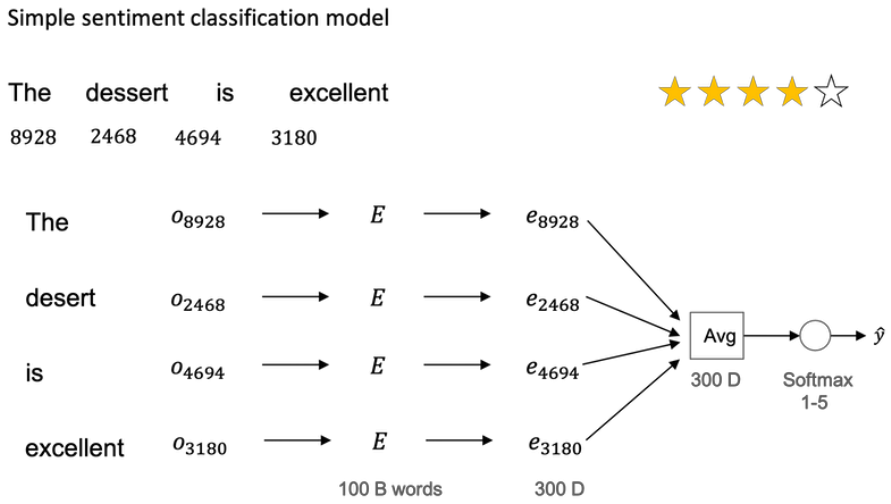
- Last 4 words: descibed previously.
- 4 words on left & right: a glass of orange ___ to go along with
- Last 1 word: orange, much more simpler context.
- Nearby 1 word: glass. This is the idea of a **Skip-Gram** model, which works surprisingly well.
  - If your main goal is really to learn a word embedding, then you can use all of these other contexts and they will result in very meaningful work embeddings as well.

5.6.3 Applications using Word Embeddings:

Sentiment Classification:

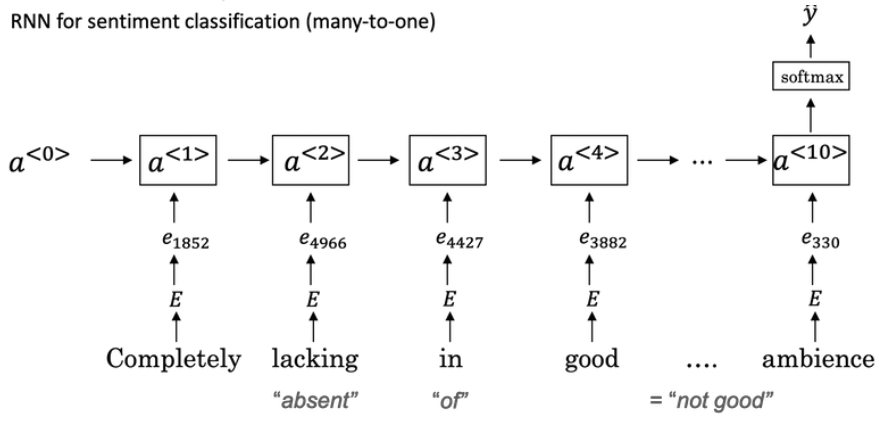
Comments	Stars
The dessert is excellent.	4
Service was quite slow.	2
Good for a quick meal, but nothing special.	3
Completely lacking in good taste, good service, and good ambience.	1

A simple sentiment classification model*:*



- So one of the challenges of sentiment classification is you might not have a huge label data set.
- If this was trained on a very large data set, like a hundred billion words, then this allows you to take a lot of knowledge even from infrequent words and apply them to your problem, even words that weren't in your labeled training set.
- Notice that by using the average operation here, this particular algorithm works for reviews that are short or long because even if a review that is 100 words long, you can just sum or average all the feature vectors for all hundred words and so that gives you a representation, a 300-dimensional feature representation, that you can then pass into your sentiment classifier.
- One of the problems with this algorithm is it **ignores word order**.
  - "Completely *lacking* in *good* taste, *good* service, and *good* ambiance".
  - This is a very negative review. But the word good appears a lot.

***A more sophisticated model*:**



- Instead of just summing all of your word embeddings, you can instead use a RNN for sentiment classification.

- In the graph, the one-hot vector representation is skipped.
- This is an example of a many-to-one RNN architecture.

Debiasing word embeddings:

Word embeddings maybe have the bias problem such as gender bias, ethnicity bias and so on. As word embeddings can learn analogies like man is to woman like king to queen. The paper shows that a learned word embedding might output:

Man: Computer_Programmer as Woman: Homemaker

Learning algorithms are making very important decisions and so I think it's important that we try to change learning algorithms to diminish as much as is possible, or, ideally, eliminate these types of undesirable biases.

- **Identify bias direction:**
  - The first thing we're going to do is to identify the direction corresponding to a particular bias we want to reduce or eliminate.
  - And take a few of these differences and basically average them. And this will allow you to figure out in this case that what looks like this direction is the gender direction, or the bias direction. Suppose we have a 50-dimensional word embedding.
- $g_1 = e_{she} - e_{he}$
- $g_2 = e_{girl} - e_{boy}$
- $g_3 = e_{mother} - e_{father}$
- $g_4 = e_{woman} - e_{man}$
  - $g = g_1 + g_2 + g_3 + g_4 + ...$ for gender vector.
  - Then we have
- cosine_similarity(sophie, g)) = 0.318687898594
- cosine_similarity(john, g)) = -0.23163356146
- to see male names tend to have positive similarity with gender vector whereas female names tend to have a negative similarity. This is acceptable.
  - But we also have
- cosine_similarity(computer, g)) = -0.103303588739
- cosine_similarity(singer, g)) = 0.185005181365
- It is astonishing how these results reflect certain unhealthy gender stereotypes.
  - The bias direction can be higher than 1-dimensional. Rather than taking an average, SVD (singular value decomposition) and PCA might help.
- Neutralize
  - For every word that is not definitional, project to get rid of bias.

5.7 CONCLUSION

Recurrent neural networks, or RNNs are a family of neural networks for processing sequential data. A computational graph is a way to formalize the structure of a set of computations, such as those involved in mapping inputs and parameters to outputs and loss. Traditional neural networks require the input and output sequence lengths to be constant across all predictions.Recurrent neural networks can be built in many diff erent ways. When the recurrent network is trained to perform a task that requires predicting the future from the past. Natural language processing with deep learning is an important combination. Using word vector representations and embedding layers you can train recurrent neural networks with outstanding performances in a wide variety of industries.

UNIT END EXERCISE

1. What is sequence modelling? State its applications.
2. Explain the mechanism of sequence modelling.
3. Write a note on Parts of Speech.
4. Explain the classification process using sequence modelling.
5. Explain the architecture of RNN.
5. What is word embedding?
7. Explain different types of RNN.

REFERENCES

1. Ian Goodfellow, Yoshua Bengio, Aaron Courvile, Deep Learning, MIT Press, 2016
2. Nikhil Buduma, Fundamentals of Deep Learning, O'Reilly, 2017
3. Shamsi Fatma Al and Guessoum Ahmed. 2005. A hidden Markov model-based POS tagger for Arabic. In Proceedings of the 8th International Conference on the Statistical Analysis of Textual Data. 31–42

*****

6

APPLICATIONS

Unit Structure

6.0 OBJECTIVES

In this chapter the student will learn about:

- how to use deep learning to solve applications in computer vision, speech recognition, natural language processing, and other application areas of commercial interest.

6.1 INTRODUCTION

Deep learning is based on the philosophy of connectionism. Deep Learning is a field of Artificial Intelligence (AI) that aims to imbibe human brain function in data processing machines. The way the human brain works serves as the foundation of deep learning which is also called deep neural learning. Deep learning (DL) has achieved promising results on a wide spectrum of AI application domains ranging from computer vision. Data processing is a significant field and deep learning helps in processing vast amounts of data with the help of identified and verified patterns established by the human brain. A revolutionary technique of machine learning, deep learning has helped the field of technology advance manifold. The rise of deep learning in AI has helped the digital domain excel and evolve unstoppably. Numerous applications and advantages of deep learning can further be used to understand the concept in a better manner.

6.2 LARGE-SCALE DEEP LEARNING

Deep learning (DL) has achieved promising results on a wide spectrum of AI application domains ranging from computer vision, natural language processing and machine translation, information retrieval and many others. The scale is the main driver behind the rise of DL. Larger datasets and neural networks consistently yield better performance across all tasks that generally require more computation and longer training time. Therefore, recent years have witnessed a surge of interests from both academia and industry in scaling up DL with distributed training on a large cluster of devices such as TPUs and GPUs with higher computation capability and memory limit. Data parallelism has become a dominant practice for distributed training. It distributes a large batch to multiple devices, where each device holds an identical model replica, computes the gradient of a local batch, and finally gathers the gradients at each iteration for synchronous parameter update. With recent optimization techniques, it is now able to train very large batches on thousands of GPU devices. However, training at such scales requires overcoming both algorithmic and systems related challenges. One of the main challenges is the degradation of model accuracy with large batch size beyond a certain point (e.g., 32k). Naively increasing the batch size typically results in degradation of generalization performance and reduces computational benefits. Additionally, we cannot always improve the training speed by just using more processors as the communication cost is a non-negligible overhead. Intuitively multiple processors collaboratively training one task can reduce the overall training time, but the corresponding communication cost between processors is heavy and limits the model scalibility. Worse still, models with tens of billions to trillions of parameters clearly do not fit into memory of a single device, and simply adding more devices will not help scale the training.

6.2.2 Fast CPU Implementations:

Traditionally, neural networks were trained using the CPU of a single machine. Today, this approach is generally considered insufficient. We now mostly use GPU computing or the CPUs of many machines networked together. Before moving to these expensive setups, researchers worked hard to demonstrate that CPUs could not manage the high computational workload required by neural networks. Each new model of CPU has different performance characteristics, so sometimes floating-point implementations can be faster too. The important principle is that careful specialization of numerical computation routines can yield a large payoff.

6.2.3 GPU Implementations:

Graphics processing units (GPUs) are specialized hardware components that were originally developed for graphics applications. The consumer

market for video gaming systems spurred development of graphics processing hardware. The performance characteristics needed for good video gaming systems turn out to be beneficial for neural networks as well. GPU hardware was originally so specialized that it could only be used for graphics tasks. Over time, GPU hardware became more flexible, allowing custom subroutines to be used to transform the coordinates of vertices or assign colors to pixels.

6.2.4 Large-Scale Distributed Implementations:

Distributing inference is simple, because each input example we want to process can be run by a separate machine. This is known as data parallelism. It is also possible to get model parallelism, where multiple machines work together on a single datapoint, with each machine running a different part of the model. This is feasible for both inference and training. Data parallelism during training is somewhat harder. We can increase the size of the minibatch used for a single SGD step, but usually we get less than linear returns in terms of optimization performance. It would be better to allow multiple machines to compute multiple gradient descent steps in parallel.

6.3 COMPUTER VISION

Computer vision is a very broad field encompassing a wide variety of ways of processing images, and an amazing diversity of applications. Applications of computer vision range from reproducing human visual abilities, such as recognizing faces, to creating entirely new categories of visual abilities. As an example of the latter category, one recent computer vision application is to recognize sound waves from the vibrations they induce in objects visible in a video. Most deep learning research on computer vision has not focused on such exotic applications that expand the realm of what is possible with imagery but rather a small core of AI goals aimed at replicating human abilities. Most deep learning for computer vision is used for object recognition or detection of some form, whether this means reporting which object is present in an image, annotating an image with bounding boxes around each object, transcribing a sequence of symbols from an image, or labelling each pixel in an image with the identity of the object it belongs to.

6.3.1 Preprocessing:

Computer vision usually requires relatively little of this kind of pre-processing. The images should be standardized so that their pixels all lie in the same, reasonable range, like [0,1] or [-1, 1]. Mixing images that lie in [0,1] with images that lie in [0, 255] will usually result in failure. Formatting images to have the same scale is the only kind of pre-processing that is strictly necessary. Many computer vision architectures require images of a standard size, so images must be cropped or scaled to

fit that size. Even this rescaling is not always strictly necessary. Some convolutional models accept variably sized inputs and dynamically adjust the size of their pooling regions to keep the output size constant. Dataset augmentation is an excellent way to reduce the generalization error of most computer vision models. A related idea applicable at test time is to show the model many different versions of the same input (for example, the same image cropped at slightly different locations) and have the different instantiations of the model vote to determine the output.

6.3.2 Dataset Augmentation:

It is an excellent way to reduce the generalization error of most computer vision models. A related idea applicable at test time is to show the model many different versions of the same input (for example, the same image cropped at slightly different locations) and have the different instantiations of the model vote to determine the output. Object recognition is a classification task that is especially amenable to this form of dataset augmentation because the class is invariant to so many transformations and the input can be easily transformed with many geometric operations. As described before, classifiers can benefit from random translations, rotations, and in some cases, flips of the input to augment the dataset. In specialized computer vision applications, more advanced transformations are commonly used for dataset augmentation.

6.4 SPEECH RECOGNITION

The task of speech recognition is to map an acoustic signal containing a spoken natural language utterance into the corresponding sequence of words intended by the speaker. Let $X = (x^{(1)}, x^{(2)}, \ldots, x^{(T)})$ denote the sequence of acoustic input vectors (traditionally produced by splitting the audio into 20ms frames). Most speech recognition systems pre-process the input using specialized hand-designed features, but some deep learning systems learn features from raw input. Let y = (y1, y2, . . ., yN) denote the target output sequence (usually a sequence of words or characters). The automatic speech recognition (ASR) task consists of creating a function $f*$ ASR that computes the most probable linguistic sequence y given the acoustic sequence X:

$$f* \text{ ASR}(X) = \arg\max_y P*(y \mid X = X)$$

where $P*$ is the true conditional distribution relating the inputs X to the targets y.

6.5 NATURAL LANGUAGE PROCESSING

Natural language processing (NLP) is the use of human languages, such as English or French, by a computer. Computer programs typically read and emit specialized languages designed to allow efficient and unambiguous

parsing by simple programs. More naturally occurring languages are often ambiguous and defy formal description. Natural language processing includes applications such as machine translation, in which the learner must read a sentence in one human language and emit an equivalent sentence in another human language. Many NLP applications are based on language models that define a probability distribution over sequences of words, characters or bytes in a natural language.

6.5.1 n-grams':

An n-gram is a sequence of n tokens. Models based on n-grams define the conditional probability of the n-th token given the preceding n − 1 tokens. The model uses products of these conditional distributions to define the probability distribution over longer sequences:

$$P(x_1, \ldots, x_\tau) = P(x_1, \ldots, x_{n-1}) \prod_{t=n}^{\tau} P(x_t \mid x_{t-n+1}, \ldots, x_{t-1})$$

This decomposition is justified by the chain rule of probability. The probability distribution over the initial sequenceP (x1, . . . , xn−1) may be modeled by a different model with a smaller value of n. Training n-gram models is straightforward because the maximum likelihood estimate can be computed simply by counting how many times each possible n gram occurs in the training set.

6.5.2 Hierarchical Softmax:

A classical approach (Goodman, 2001) to reducing the computational burden of high-dimensional output layers over large vocabulary sets V is to decompose probabilities hierarchically. Instead of necessitating a number of computations proportional to |V| (and also proportional to the number of hidden units, nh), the |V| factor can be reduced to as low as log |V|. To predict the conditional probabilities required at each node of the tree, we typically use a logistic regression model at each node of the tree, and provide the same context C as input to all of these models. Because the correct output is encoded in the training set, we can use supervised learning to train the logistic regression models.

6.6 OTHER APPLICATIONS

The types of applications of deep learning that are different from the standard object recognition, speech recognition and natural language processing tasks discussed above.

6.6.1 Recommender Systems:

One of the major families of applications of machine learning in the information technology sector is the ability to make recommendations of

items to potential users or customers. Two major types of applications can be distinguished: online advertising and item recommendations. Both rely on predicting the association between a user and an item, either to predict the probability of some action or the expected gain. If an ad is shown or a recommendation is made regarding that product to that user. Companies including Amazon and eBay use machine learning, including deep learning, for their product recommendations.

6.6.2 Exploration Versus Exploitation:

Many recommendation problems are most accurately described theoretically as contextual bandits. The issue is that when we use the recommendation system to collect data, we get a biased and incomplete view of the preferences of users: we only see the responses of users to the items they were recommended and not to the other items. This would be like training a classifier by picking one class yˆ for each training example x (typically the class with the highest probability according to the model) and then only getting as feedback whether this was the correct class or not. The bandit problem is easier in the sense that the learner knows which reward is associated with which action. In the general reinforcement learning scenario, a high reward or a low reward might have been caused by a recent action or by an action in the distant past.

6.6.3 Knowledge Representation, Reasoning and Question Answering:

Deep learning approaches have been very successful in language modeling, machine translation and natural language processing due to the use of embeddings for symbols and words. These embeddings represent semantic knowledge about individual words and concepts.

6.7 SUMMARY

Deep learning has been applied to many other applications besides the ones described here, and will surely be applied to even more after this writing. Given larger datasets and bigger models consistently yielding significant improvements in accuracy, large-scale deep learning has become an inevitable trend. As datasets increase in size and DNNs in complexity, the computational intensity, communication cost and memory demands of deep learning increase proportionally. Recent years have witnessed a surge of interests from both academia and industry in scaling up DL with distributed training on a large cluster of devices such as TPUs and GPUs with higher computation capability and memory limit. Eventually, knowledge of relations combined with a reasoning process and understanding of natural language could allow us to build a general question answering system.

UNIT END EXERCISE

1. Explain deep learning application with reference to Natural Language Processing
2. Explain deep learning role in speech recognition.
3. Write a note on large scale deep learning.
4. Write a note on computer vision.
5. Give comparison between Exploration and Exploitation.
6. Explain in brief about recommender system.

REFERENCES

1. Ian Goodfellow, Yoshua Bengio, Aaron Courvile, Deep Learning, MIT Press, 2016
2. Nikhil Buduma, Fundamentals of Deep Learning, O'Reilly, 2017
3. Shamsi Fatma Al and Guessoum Ahmed. 2006. A hidden Markov model-based POS tagger for Arabic. In Proceedings of the 8th International Conference on the Statistical Analysis of Textual Data. 31–42.

*****

UNIT IV

7

DEEP LEARNING RESEARCH

Unit Structure

7.0 OBJECTIVES

After going through this unit, you will be able to:

- Understand and classify Linear factor Models.
- Understand significance of Autoencoders
- Understand representation learning and its use in deep learning models.

7.1 INTRODUCTION

In the earlier units we have studied about supervised learning algorithms and other models. We understand that a good accuracy can be achieved with a good amount of labeled data, which at times doesn't seem to exist. Quality of data used in supervised learning algorithms is always questionable. Thus, we look ahead for developing general models in deep learning which can work better in absence of labeled data or supervised data. This will also help in good applicability and achieve higher accuracy.

In this chapter we shall focus on unsupervised learning. Though unsupervised learning doesn't solve the problem as supervised learning does, but a lot can be explored and researched. A major problem with unsupervised learning is the high dimensionality of the problem which eventually generates other problems like computations, statistical calculations etc. Many of this can be handled with proper design and other approaches. Let us have a look at existing unsupervised learning approaches in the following part.

7.2 LINEAR FACTOR MODELS

Probabilistic models use probabilistic inference to predict variables using other given variables. A linear factor model is a probabilistic model where X is obtained by using a linear decoder function. This linear decoder function is created by combining linear transformation(Wh+b) with noise.

X=Wh+b+noise

Where,

h is the latent variable,

W is the weight matrix

b denotes bias

noise , is normally distributed or diagonal.

In linear factor model, h is an explanatory factor obtained from a factorial distribution.

h~p(h), where, p(h)=$\prod_i p(hi)$

here all hi's are independent.

Graphically relation between x_i's and h_i's can be represented as follows,

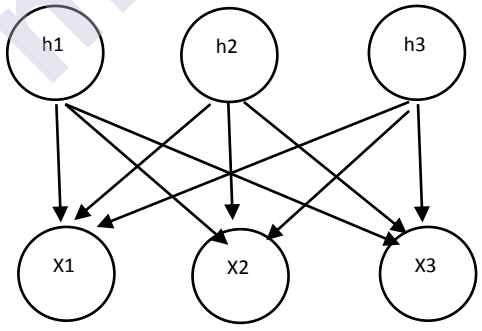


Figure 7.1: relation between x_i's and h_i's

We can see above how the observed variables x_i are obtained from hidden variables h_i.

The probabilistic PCA(principal component analysis), factor analysis and ICA(Independent component analysis) models are typically the variations

of Linear factor models with different choices of noise and the distribution of latent variable.

7.2.1 Probabilistic PCA:

Principal component analysis (PCA) is a popular technique used in variable selection, the concept of PCA is to find the set of axes which communicate the most information of the data set, thus reducing the dimensions to work with. As we know variance corresponds to information, the process in PCA is to find vectors with maximum variance and keep repeating it till we find the vectors equal to the dimension of the dataset. We select m vectors (axes) for a dataset with dimension d, such that m<d. These m axes store maximum information.

PCA is been widely used for dimensionality reduction but it fails to capture information if the relationships are nonlinear. We have probabilistic Principal component analysis which takes advantage of the observations that most variations in the data can be captured by the latent variables with residual error σ^2.As $\sigma^2\ tends\ to\ 0$ the probabilistic PCA becomes PCA.

In probabilistic PCA**,** noise is considered to be drawn from a diagonal covariance Gaussian distribution, where the

covariance matrix is a diagonal matrix of equal variances.

Covariance matrix=diag(σ^2)

Where $\sigma^2 = [\sigma^2, \sigma^2, \sigma^2, \dots\dots\dots\dots, \sigma^2]^T$, a variance vector.
In this case h captures, the dependencies between variables x.

Thus we have

h ~N(h;0,I)

x ~N(x;b,$WW^T+\sigma^2 I$)

Thus, x=Wh+b+ σz, where z is Gaussian noise.

7.2.2 Factor Analysis:

In contrast to PCA factor analysis focusses on locating independent variables. Considering the Linear factor models discussed above in Factor analysis linear model, the latent variable h is unit variance Gaussian and the variable x are assumed to be conditionally dependent such that h is given.

In Factor analysis noise is considered to be drawn from a diagonal covariance Gaussian distribution, where the covariance matrix is a diagonal matrix of variances.

Covariance matrix=diag(σ^2)

Where $\sigma^2 = [\sigma_1^2, \sigma_2^2, \sigma_3^2, \ldots\ldots\ldots\ldots \sigma_n^2]^T$, a variance vector.

In this case h captures the dependencies between variable x.

Thus we have
h ~N(h;0,I)
x ~N(x;b,WW^T+diag(σ^2))

7.2.3 Independent Component Analysis (ICA):

ICA deals with separating an observed signal into the underlying signals which are scaled and added together to form observed data. These signals are supposed to be independent.

For example, separating speech signal of people talking simultaneously

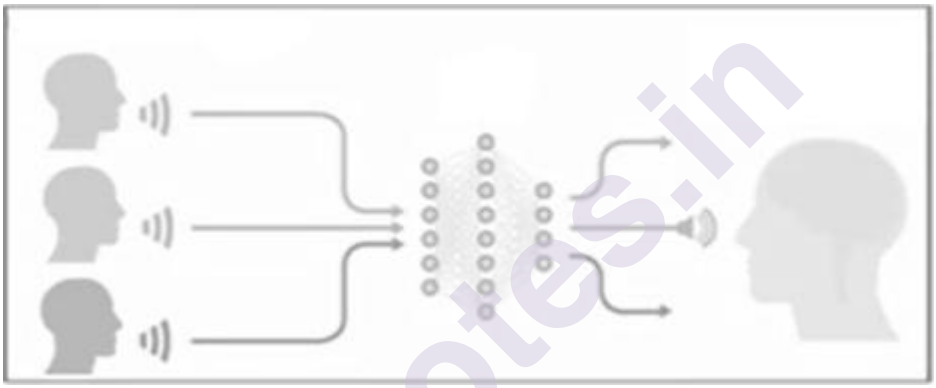


Figure 7.2: Algorithm separates speech signal

In this model the prior distribution p(h) from which h is generated, is fixed well before.

Thus,
x=Wh

This Model can be trained using Maximum likelihood.
If p(h) is independent, then we can obtain underlying factors that are likely to be independent.

For example, if we have n microphones placed in different locations such that x_i is an observation of mixed signals,
h_i is an estimate of original independent signal
ICA can help separation of signals i.e. each hi contains only one person speaking clearly.

Many variants of ICA are possible. And all these variants require p(h) to be non-Gaussian.

7.2.4 Slow Feature Analysis (SFA):

Slow Feature Analysis is based on slowness principle; it is applied to any model trained with gradient descent. In slow feature analysis the information is extracted from time signals. It's an unsupervised learning algorithm which helps in extracting slowly varying features from a quick varying signal.

For example, in video frame of moving zebra, a zebra moves from left to right, an individual pixel will quickly change from black to white and back again as the zebra's stripes pass over the pixel. But the feature indicating whether a zebra is in the image will not change, and the feature describing the zebra's position will change slowly.

Slow feature analysis is efficient because it is used in linear feature extractor. It is possible theoretically to predict the feature SFA will learn.

7.2.5 Sparse coding:

Sparse coding is a class of unsupervised learning methods for learning sets to represent data efficiently. This is the most used Linear factor model from the unsupervised feature learning. It helps in deciding the value of latent variable in the model.
In this model also, x is obtained using decoder and reconstructions.
$X=Wh+b+noise$

The model assumes that linear factor model has Gaussian noise with isotropic precision β.
$p(x|h)=N(x;Wh+b,(1/\beta)I)$

7.2.6 Manifold Interpretation of PCA:

Linear factor models can be explained as a manifold. We know that Principal Component analysis helps in dimensionality reduction, but it works well when the data has linear relationship. Problem arises when data has nonlinear relationships. These problems can be handled using manifold learning. Manifold learning describes the high dimensional datasets into low dimensional manifolds. Probabilistic PCA can be viewed as a region of the shape of a thin pancake with high probability. PCA can be interpreted as aligning this pancake with a linear manifold in a higher-dimensional space.

7.3 AUTOENCODERS

Autoencoder neural network is trained using unsupervised learning. Autoencoders are feedforward NN having outpust same as input.The input is compressed and then reconstructed using Autoencoders.

Input-->Encoder-->code-->Decoder-->Output (same as input)

An auto encoder has 3 components:

1. **Encoder:** The encoder compresses the input and produces the code. This is done with the help of an encoding method.
2. **Code:** it is the latent representation of the code
3. **Decoder:** input is reconstructed by the decoder. This is done with the help of a decoding method.

Autoencoders is mainly used in dimensionality reduction, but they also exhibit other properties.

They are Data-specific. Autoencoders are lossy they don't yield the exact same ouput as the input, it can be a degraded representation. To train an autoencoder we need the raw input data. Autoencoders generate their own labels from the training data and therefore they are termed as self-supervised or unsupervised algorithms.

7.3.1 Undercomplete Autoencoders:

An autoencoder whose code dimension is less than the input dimension is called undercomplete. Learning an undercomplete representation pushes the autoencoder to capture the most significant features of the training data. Thus, to obtain main features from the autoencoder is by constraining h to have smaller dimension than x.

7.3.2 Regularized Autoencoders:

We have seen above that, Undercomplete autoencoders, with code dimension less than the input dimension, can learn the most salient features of the data distribution. Also, autoencoders fail to learn anything useful if the encoder and decoder are given too much amount of data. Possibly, one can train any architecture of autoencoder by taking the code dimension and the capacity of the encoder and decoder based on the complexity of distribution to be modeled. This can be achieved with Regularized autoencoders. Instead of limiting the model capacity, regularized autoencoders use a loss function that promotes the model to have other properties apart from copying its input to its output. These properties include:

1. sparseness of the representation
2. compactness of the derivative of the representation
3. robustness to noise / missing inputs.

A regularized autoencoder can be nonlinear and overcomplete however it can still learn something important about the data distribution.

Typically, sparse autoencoders are used to learn features for a different job, such as classification. Instead of just operating as an identity function,

an autoencoder that has been regularized to be sparse must respond to unique statistical properties of the dataset it has been trained on. In this sense, using a sparsity penalty to train a model to execute a copying task can result in a model that has learnt important features as a side effect.

Autoencoders with Denoising Rather than adding a penalty to the cost function, we can change the reconstruction error term of the cost function to obtain an autoencoder that learns anything meaningful. Instead, a denoising autoencoder, minimizes $L(x, g(f(\tilde{x}))$, where $\tilde{x}$ is a duplicate of x that has been distorted by noise. Instead of just copying their input, denoising autoencoders must repair the damage.

Autoencoders are frequently learned using just a single layer encoder and decoder. This is not, however, a must. Deep encoders and decoders provide numerous benefits. Remember that depth in a feedforward network has a lot of advantages. These benefits also apply to autoencoders because they are feedforward networks. Furthermore, because the encoder and decoder are both feedforward networks, each of these autoencoder components can benefit from depth on its own. The universal approximator theorem implies that a feedforward neural network with at least one hidden layer can achieve non-trivial depth.

Given enough hidden units, a deep autoencoder with at least one extra hidden layer inside the encoder can approximate any mapping from input to code arbitrarily accurately. The computational cost of modelling some functions can be reduced by an order of magnitude when using depth. Depth can also reduce the quantity of training data required to learn some functions tremendously. Because greedily pretraining the deep architecture by training a stack of shallow autoencoders is a typical technique for training a deep autoencoder, we frequently meet shallow autoencoders, even when the end goal is to train a deep autoencoder.

7.3.3 Denoising Autoencoders:

An alternate way to make the autoencoder to learn some important features is by adding random noise to the input. Because the input contains random noise,it wont be possible for an autoencoder to replicate the input to the output.
Thus the input can be decoded by removing the noise, this process is called denoising autoencoder.

Autoencoder is expected to generate the input image even if it not actually seeing that input.

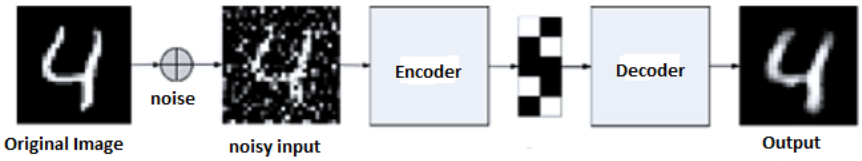


Figure 7.3 Autoencoder

7.3.4 Applications of Autoencoders:

- Autoencoders have been effectively used in dimensionality reduction and information retrieval tasks.
- Autoencoders are good at denoising of images.
- Autoencoders are used in Feature Extraction, they help to learn important hidden features of the input data.
- Variational Autoencoder (VAE), is used to generate images.
- Autoencoders are used in Sequence-to-Sequence Prediction.
- Deep Autoencoders can be used in recommending movies, books, or other items.

7.4 REPRESENTATION LEARNING

Representation learning is learning representations of input data typically by transforming it or extracting features from it (by some means), that makes it easier to perform a task like classification or prediction. Representation makes subsequent learning easier.

Information processing tasks can be easy or difficult depending on how the information is represented and viewed. Thus, a representation is said to be good if it makes the subsequent learning tasks easier, and the choice of representation depends on what is the successive task. This determines if the representation is good or bad. For example, it will be easy for a person to divide 670 by 6 using long division method, but once we change the representation of numbers to other forms like hexadecimal or roman then it becomes difficult. Thus representation matters.

Feedforward networks taught by supervised learning can be thought of as doing a sort of representation learning. A linear classifier, such as a SoftMax regression classifier, is often used as the network's final layer. The rest of the network learns to give this classifier a representation. The representation at every hidden layer takes on qualities that make the classification task easier when trained with a supervised criterion.

We frequently have a lot of unlabeled training data and a small amount of annotated training data. On the labelled subset, supervised learning techniques often result in significant overfitting. Semi-supervised learning

can help overcome the overfitting problem by learning from unlabeled data as well. We can develop good unlabeled data representations and then use these representations to perform the supervised learning challenge.

The procedure to train a deep supervised network without requiring architectural specializations like convolution or recurrence is termed as greedy layer wise unsupervised pretraining. This process shows how a representation is learned for one task can be useful for another task.

Greedy layer-wise pretraining gets its name from the fact that it's a greedy algorithm, which means it optimizes each piece of the solution separately, one at a time, rather than all at once. Layer-wise refers to the independent elements that make up the network's layers. Because each layer is trained with an unsupervised representation learning algorithm, it is called unsupervised.

Other unsupervised learning techniques, such as deep autoencoders and probabilistic models with multiple layers of latent variables, can benefit from greedy layer-wise unsupervised pretraining.

5.6 SUMMARY

- Linear factor models are the simplest generative models.
- Probabilistic PCA, Factor models, independent component Analysis are all obtained with variations in Linear factor models.
- Slow Feature Analysis is based on slowness principle and is used in important feature extraction.
- Autoencoders are feedforward neural networks where the input is same as the output.
- An autoencoder has 3 components: Encoder, Code & Decoder.
- Significant features can be learnt using undercomplete, regularized and denoised autoencoders.
- The concept of representation learning links together all the many forms of deep learning. Feedforward and recurrent networks, autoencoders and probabilistic models all learn and develop representations.

QUESTIONS

1. What are Linear factor Models.

2. Explain the concept of Probabilistic PCA

3. Compare Factor Analysis & Independent Component Analysis

4. Write short note on Slow feature analysis

5. What is Sparse coding?

7. What is Manifold Interpretation of PCA?

7. What are Autoencoders?

8. How do we obtain Undercomplete Autoencoders & Regularized Autoencoders?

9. What is the significance of Denoising Autoencoders

10. List Applications of Autoencoders

11. Write a short not on importance of Representation learning in deep learning.

REFERENCES & BIBLIOGRAPHY

- Deep Learning Ian Goodfellow, Yoshua Bengio, Aaron Courvile An MIT Press book 1st 2016

- Deep Learning: Methods and Applications Deng & Yu Now Publishers 1st 2013

- https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798

- https://towardsdatascience.com/tagged/representation-learning

- https://cedar.buffalo.edu/~srihari/CSE676/

- Youtube Channel : Sargur Srihari

*****

UNIT V

8

APPROXIMATE INFERENCE

Unit Structure

8.0 OBJECTIVES

This chapter would make you understand the following concepts:

- Fundamentals of approximate inference
- Algorithm and working of Expectation Maximization

8.1 INTRODUCTION

Approximate inference methods useful to learn realistic models from large dataset (image or video or textual or any other.) by falling computation time for accuracy.

Exact inference is carried out with the posterior probability of the parameters. However, we often do not have access to that posterior — it may be difficult to compute, sample, or both! In those cases, if we can find a — necessarily biased, but simpler — approximation to the posterior, we can use that approximation to carry out inference.

8.1.1 Posterior Probability Definition:

- A posterior probability, in Bayesian statistics, is the revised or updated probability of an event occurring after taking into consideration new information.

- The posterior probability is calculated by updating the prior probability using Bayes' theorem.
- In statistical terms, the posterior probability is the probability of event A occurring given that event B has occurred.

Bayes' Theorem Formula:

The formula to calculate a posterior probability of A occurring given that B occurred:

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)} = \frac{P(A) \times P(A)}{P(B)}$$

Remark:

A, B = events
P (B|A) = the probability of B occurring given that Ais true
P (B) and P(B) = the probabilities of A occurring and B occurring independently of each other

Bayes' theorem can be used in many applications, such as economics, medicine, and finance. In finance, Bayes' theorem can be used to update a previous belief once new information is obtained.

Prior probability represents what is originally believed before new evidence is introduced, and posterior probability takes this new information into account.

8.2 APPROXIMATE INFERENCE IN MACHINE LEARNING

From a probabilistic perspective, we can frame the problem of learning as an inference problem. We have a model M which is parameterized by a set of variablesθ. We also have access to some observed data $D = \{x_i\}_{i=1}^{N}$ and that can include labels or anything else. Our goal in learning is to find a setting for θ such that our model is useful. In the language of probability, we are looking for a posterior distribution over the parameters, and Bayes theorem tells us exactly what to do:

$$p(\theta|D, M) = \frac{p(\theta, M)p(M)}{p(D|M)}$$

where $p(x|\theta, M), p(\theta|M)$ are the likelihood and prior distribution (respectively) and are specified by M.

The simplicity of the Bayes theorem has complexities that in most cases we cannot exactly solve the inference problem. In most cases, $p(D|M)$ is completely intractable, and we cannot really do anything with that equation.

The notion of approximate inference addresses the above issue: so we can approximately solve Bayes theorem for complex cases, ie. We scale up Bayesian learning to the types of interesting, high-dimensional datasets that we want to deal with today's Machine Learning.
We can roughly divide approximate inference schemes into two categories: deterministic and stochastic. Stochastic methods are based on the idea of Monte-Carlo sampling i.e., we can approximate any expectation w.r.t. a distribution as a mean of samples from it:

$$Ep(x)[f(x)] \approx \frac{1}{L}\sum_{l=1}^{L} f(x^l) \ with\, x^l \sim p(x)$$

The problem of inference can be converted into sampling from the posterior distribution. Happily, there are many cases where the posterior is intractable, but we can still sample from it. Here, Markov-Chain Monte Carlo algorithms dominate the landscape.

Deterministic methods substitute the problem of inference with optimization. We can parameterize the approximation with some *variational* parameters, and then minimize a probabilistic divergence w.r.t. the variational parameters. We then use the trained approximate distribution instead of the true, intractable one.

8.3 APPROXIMATE INFERENCE IN DEEP LEARNING

In the context of deep learning, we usually have a set of visible variables v and a set of latent variables h. The challenge of inference usually refers to the difficult problem of computing $p(h \mid v)$ or taking expectations with respect to it. Such operations are often necessary for tasks like maximum likelihood learning.

In most graphical models with multiple layers of hidden variables have intractable posterior distributions. Exact inference requires an exponential amount of time in these models. Even some models with only a single layer, such as sparse coding, have the same problem.

Intractable inference problems in deep learning usually arise from interactions between latent variables in a structured graphical model. (Ex. Refer Figure No 8.1)

Intractable inference problems in deep learning are usually the result of interactions between latent variables in a structured graphical model. These interactions can be due to edges directly connecting one latent variable to another or longer paths that are activated when the child of a V-structure is observed.

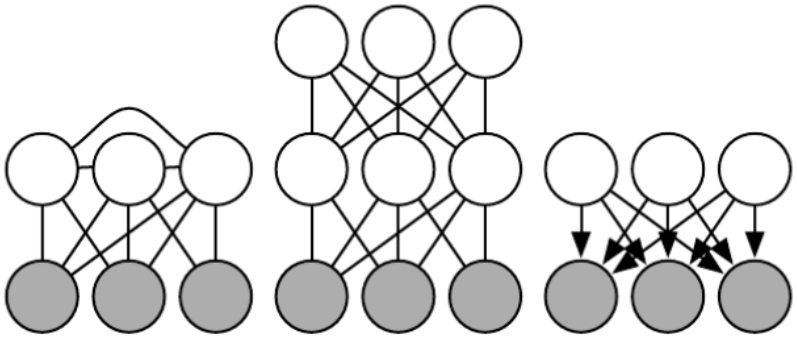


Figure No 8.1: Sample fully connected Network

- (Left part of Figure No 8.1) A semi-restricted Boltzmann machine with connections between hidden units. These direct connections between latent variables make the posterior distribution intractable because of the large cliques of latent variables.
- (Center part of Figure No 8.1)A deep Boltzmann machine, organized into layers of variables without intra-layer connections, still has an intractable posterior distribution because of the connections between layers.
- (Right part of Figure No 8.1)This directed model has interactions between latent variables when the visible variables are observed because every two latent variables are coparents.
- Some probabilistic models are able to provide tractable inference over the latent variables despite having one of the graph structures depicted above.

8.4 INFERENCE AS OPTIMIZATION

Approximate inference algorithms may then be derived by approximating the underlying optimization problem.

To compute the log-probability of the observed data, $log\ p(v;\ \theta)$. we can compute a lower bound $L(v, \theta, q)$ on $log\ p(v; \theta)$. This bound is called the **evidence lower bound** (ELBO). Another commonly used name for this lower bound is the negative **variational free energy**.

The evidence lower bound is defined to be

$$L(v, \theta, q) = log\ p(v;\ \theta) - D_{KL}\ (q(h\ |v)\ ||\ p(h\ |\ v;\ \theta))$$

Remark:

Observed variables v
Latent variables h
q is an arbitrary probability distribution over h.

The difference between $log\, p(v)$ and $L(v, \theta, q)$ is given by the KL divergence, and because the KL divergence is always nonnegative, we can see that L always has at most the same value as the desired log probability. The two are equal if and only if q is the same distribution as $p(h \mid v)$.

The canonical definition of the evidence lower bound

$$L(v, \theta, q) = E_{h \sim q}\,[log\, p(h, v)] + H(q).$$

For an appropriate choice of q, L is tractable to compute. For any choice of q, L provides a lower bound on the likelihood. For $q(h \mid v)$ that are better approximations of $p(h \mid v)$, the lower bound L will be tighter, in other words, closer to $log\, log\, p(v)$. When $q(h \mid v) = p(h \mid v)$, the approximation is perfect, and $L(v, \theta, q) = log\, log\, p(v; \theta)$.

The procedure for finding the q that maximizes L. Exact inference maximizes L perfectly by searching over a family of functions q that includes $p(h \mid v)$.

To derive different forms of approximate inference from approximate optimization to find q.

It can make the optimization procedure less expensive but approximate by restricting the family of distributions q that the optimization is allowed to search over or by using an imperfect optimization procedure that may not completely maximize L
but may merely increase it by a significant amount. No matter what choice of q we use, L is a lower bound. It can get tighter or looser bounds that are cheaper or more expensive to compute depending on how it chooses to approach this optimization problem.

It can obtain a poorly matched q but reduce the computational cost by using an imperfect optimization procedure, or by using a perfect optimization procedure over a restricted family of q distributions.

8.5 EXPECTATION MAXIMIZATION

An expectation-maximization algorithm is an approach for performing maximum likelihood estimation in the presence of latent variables. It does this by first estimating the values for the latent variables, then optimizing the model, then repeating these two steps until convergence. It is an effective and general approach and is most commonly used for density estimation with missing data, such as clustering algorithms like the Gaussian Mixture Model.

The EM algorithm is an iterative approach that cycles between two modes. The first mode attempts to estimate the missing or latent variables called the estimation-step or E-step. The second mode attempts to optimize the

parameters of the model to best explain the data called the maximization-step or M-step.

- E-Step. Estimate the missing variables in the dataset.
- M-Step. Maximize the parameters of the model in the presence of the data.

The E-step (expectation step):

Let $\theta^{(0)}$ denote the value of the parameters at the beginning of the step.

Set $q(h(i) \mid v) = p(h(i) \mid v(i); \theta(0))$ for all indices i of the training examples $v^{(i)}$; to train on (both batch and minibatch variants are valid).

q is defined in terms of the *current* parameter value of $\theta^{(0)}$; if we vary θ, then $p(h \mid v; \theta)$ will change, but q(h | v) will remain equal to $p(h \mid v; \theta(0))$.

The M-step (maximization step):

Completely or partially maximize

$$\sum_i \mathrm{L}(\mathrm{v}^{(i)}, \theta, \mathrm{q})$$

with respect to θ using any optimization algorithm can be used.

8.5.1 Algorithm of Expectation Maximization:

1. Given a set of incomplete data, consider a set of starting parameters.
2. Expectation step (E – step): Using the observed available data of the dataset, estimate (guess) the values of the missing data.
3. Maximization step (M – step): Complete data generated after the expectation (E) step is used in order to update the parameters.
4. Repeat Step 2 to 3 until convergence

The EM algorithm is useful for discovering the values of latent variables. It can be used for the purpose of estimating the parameters of the Hidden Markov Model (HMM). It can be used to fill the missing data in a sample. It can be used as the basis of unsupervised learning of clusters.

The E-step and M-step are often pretty easy for many problems in terms of implementation.

It is always guaranteed that likelihood will increase with every iteration. Solutions to the M-steps often exist in the closed-form.

EM algorithm has slow convergence. It makes convergence to the local optima only. It requires both the probabilities, forward and backward (numerical optimization requires only forward probability).

8.6 MAXIMUM A POSTERIORI (MAP)

The term *inference is* referring to computing the probability distribution over one set of variables given another. When training probabilistic models with latent variables, we are usually interested in computing $p(h \mid v)$. An alternative form of inference is to compute the single most likely value of the missing variables, rather than to infer the entire distribution over their possible values. In the context of latent variable models, this means computing

$$h^* = arg\ max_h\ (p(h \mid v))$$

This is known as **maximum a posteriori (MAP)** inference.
Bayes theorem provides a principled way of calculating conditional probability.

It involves calculating the conditional probability of one outcome given another outcome, using the inverse of this relationship, stated as follows:

$$P(A \mid B) = (P(B \mid A) * P(A)) / P(B)$$

The quantity that we are calculating is typically referred to as the posterior probability of A given B and $P(A)$ is referred to as the prior probability of A. The normalizing constant of $P(B)$ can be removed, and the posterior can be shown to be proportional to the probability of B given A multiplied by the prior.

$P(A \mid B)$ is proportional to $P(B \mid A) * P(A)$

Or, simply:

$$P(A \mid B) = P(B \mid A) * P(A)$$

This is a helpful simplification as we are not interested in estimating a probability, but instead in optimizing a quantity. A proportional quantity is good enough for this purpose.

We can now relate this calculation to our desire to estimate a distribution and parameters (theta) that best explains our dataset (X),

$$P(theta \mid X) = P(X \mid theta) * P(theta)$$

Maximizing this quantity over a range of theta solves an optimization problem for estimating the central tendency of the posterior probability (e.g. the model of the distribution).

$$maximize\ P(X \mid theta) * P(theta)$$

In machine learning, Maximum a Posteriori optimization provides a Bayesian probability framework for fitting model parameters to training data and an alternative and sibling to the perhaps more common Maximum Likelihood Estimation framework.

8.7 VARIATIONAL INFERENCE AND LEARNING

The evidence lower bound $L(v, \theta, q)$ is a lower bound on $log\ p(v; \theta)$, how inference can be viewed as maximizing L with respect to q, and how learning can be viewed as maximizing L with respect to θ. The EM algorithm enables us to make large learning steps with a fixed q and that learning algorithms based on MAP inference enable us to learn using a point estimate of $p(h \mid v)$ rather than inferring the entire distribution.

The core idea behind variational learning is that we can maximize L over a restricted family of distributions q. This family should be chosen so that it is easy to compute $Eq\ log\ p(h, v)$. A typical way to do this is to introduce assumptions about how q factorizes. A common approach to variational learning is to impose the restriction that q is a factorial distribution:

$$q(h \mid v) = \prod_i q(hi \mid v)$$

This is called the **mean-field** approach. More generally, we can impose any graphical model structure we choose on q, to flexibly determine how many interactions we want our approximation to capture. This fully general graphical model approach is called **structured variational inference**

The beauty of the variational approach is that we do not need to specify a specific parametric form for q.

8.8 DISCRETE LATENT VARIABLES

Variational inference with discrete latent variables is relatively straightforward. We define a distribution q, typically one where each factor of q is just defined by a lookup table over discrete states. In the simplest case, h is binary and we make the mean-field assumption that q factorizes over each individual h_i. In this case, we can parametrize q with a vector $\widehat{h}$ whose entries are probabilities. Then $q(h_i = 1 \mid v) = \widehat{h_i}$

After determining how to represent q, we simply optimize its parameters. With discrete latent variables, this is just a standard optimization problem. In principle, the selection of q could be done with any optimization algorithm, such as gradient descent.

Because this optimization must occur in the inner loop of a learning algorithm, it must be very fast. To achieve this speed, we typically use special optimization algorithms that are designed to solve comparatively small and simple problems in few iterations. A popular choice is to iterate fixed-point equations, in other words, to solve

$$\frac{\partial}{\partial \widehat{h}_i} L = 0$$

for $\widehat{h}_i$. We repeatedly update different elements of $\widehat{h}$ until we satisfy a convergence criterion.

8.9 SUMMARY

- Approximate inference methods useful to learn realistic models from large dataset
- approximate inference schemes into two categories: deterministic and stochastic.
- The challenge of inference usually refers to the difficult problem of computing p(h | v) or taking expectations with respect to it.
- The optimization procedure less expensive than approximate
- An expectation-maximization algorithm is an approach for performing maximum likelihood estimation in the presence of latent variables.
- Maximum a Posteriori optimization provides a Bayesian probability framework for fitting model parameters to training data

Experiment Practice:

- Implement Bayes' Theorem Formula.
- Implement Bayesian-Neural-Networks
- Implement Expectation-Maximization Algorithm
- Implement MAP Algorithm

LIST OF REFERENCES

[1] Heskes, Tom & Albers, Kees & Kappen, Hilbert. (2012). Approximate Inference and Constrained Optimization. Proceedings of Uncertainty in AI.

[2] Shin Kamada, Takumi Ichimura.: An adaptive learning method of Deep Belief Network by layer generation algorithm. IEEE Xplore. Nov 2016.

[3] Shin Kamada, Takumi Ichimura.: An adaptive learning method of Deep Belief Network by layer generation algorithm. IEEE Xplore. Nov 2016.

[4] Mohamed A,Dahl G E, Hinton. G.: Acoustic modeling using deep belief networks[J]. Audio Speech and Language Processing IEEE Transactions an, vol. 20, no. 1, pp. 14-22, 2012

[5] Boureau Y, Cun L. Y.: Sparse feature learning for deep belief networks [C] //Advances in neural information processing systems. pp. 1185-1192, 2008

[6] Hinton. G.: A practical guide to training restricted Boltzmann machines[J]. Momentum, vol. 9, no. 1, pp. 926, 2010

[7] Bengio Y, Lamblin P, Popovici Det al.:Greedy layer-wise training of deep networks. Advances in neural information processing systems, vol. 19, pp. 153, 2008.

[8] Ranzato A M, Szummer M.: Semi-supervised learning of compact document representations with deep networks[C]//Proceedings of the 25th international conference on Machine learning. ACM, pp. 792-799, 2008.

[9] Neal R M, Hinton G. E.: A view of the EM algorithm that justifies incremental sparse and other variants[M]//Learning in graphical models. pp. 355-368, 1998.

[10] Hinton G E, Salakhutdinov R. R.: Reducing the dimensionality of data with neural networks[J]. Science, vol. 313, no. 5786, pp 504-507, 2006.

[11] Goodfellow and Bengio, "Deep learning " 2016

MODEL QUESTIONS

1. Explain how approximate Inference works machine learning
2. Explain relationship between approximate Inference deep learning
3. Define Posterior Probability
4. Explain Expectation Maximization algorithm
5. Write Expectation Maximization algorithm
6. Write Maximum a Posteriori (MAP) algorithm
8. Compare Supervised and Unsupervised Learning

*****

9

DEEP GENERATIVE MODELS

Unit Structure

9.0 OBJECTIVES

This chapter would make you understand the following concepts

- Fundamentals of Deep Generative Models
- Algorithm and working of Restricted Boltzmann Machines
- Algorithm and working of Deep Belief Networks

9.1 INTRODUCTION

A generative model includes the distribution of the data itself, and tells us how likely a given examples are. For example, models that predict the next word in a sequence are typically generative models (usually much simpler than GANs) because they can assign a probability to a sequence of words.

Some applications are as follows:

Generate Examples for Image Datasets,
Generate Realistic Photographs
Image-to-Image Translation,
Face Frontal View Generation
Generate Photographs of Human Faces
Photos to Emojis
Photograph Editing
Face Aging
Photo Blending

Super Resolution
Photo Inpainting
Generate Cartoon Characters
Text-to-Image Translation
Semantic-Image-to-Photo Translation
Generate New Human Poses
Clothing Translation
Video Prediction
3D Object Generation

9.1 DEEP GENERATIVE MODELS

Deep generative models (DGM) are neural networks with many hidden layers trained to approximate complicated, high-dimensional probability distributions using a large number of samples. When trained successfully, we can use the DGMs to estimate the likelihood of each observation and to create new samples from the underlying distribution. Applications of deep generative models (DGM), such as creating fake portraits from celebrity images are quite popular.

The ambitious goal in DGM training is to learn an unknown or intractable probability distribution from a typically small number of independent and identically distributed samples. When trained successfully, we can use the DGM to estimate the likelihood of a given sample and to create new samples that are similar to samples from the unknown distribution. These problems have been at the core of probability and statistics for decades but remain computationally challenging to solve, particularly in high dimensions.

Three key mathematical challenges in DGM:

1. DGM training is an ill-posed problem since uniquely identifying a probability distribution from a finite number of samples is impossible. Hence, the performance of the DGM will depend heavily on so-called hyper-parameters, which include the design of the network, the choice of training objective, regularization, and training algorithms.

2. Training the generator requires a way to quantify its samples' similarity to those from the intractable distribution. In the approaches considered here, this either requires the inversion of the generator or comparing the distribution of generated samples to the given dataset. Both of these avenues have their distinct challenges. Inverting the generator is complicated in most cases, particularly when it is modeled by a neural network that is nonlinear by design. Quantifying the similarity of two probability distributions from samples leads to two-sample test problems, which are especially difficult without prior assumptions on the distributions.

3. Most common approaches for training DGMs assume that we can approximate the intractable distribution by transforming a known and much simpler probability distribution (for instance, a Gaussian) in a latent space of a known dimension. In most practical cases, determining the latent space dimension is impossible and is left as a hyper-parameter that the user needs to choose. This choice is both difficult and important. With an overly conservative estimate, the generator may not approximate the data well enough, and an overestimate can render the generator non-injective, which complicates the training.

9.1.1 Supervised vs. Unsupervised Learning:

A typical machine learning problem involves using a model to make a prediction, e.g. predictive modeling.

This requires a training dataset that is used to train a model, comprised of multiple examples, called samples, each with input variables (X) and output class labels (y). A model is trained by showing examples of inputs, having it predict outputs, and correcting the model to make the outputs more like the expected outputs.

In the predictive or supervised learning approach, the goal is to learn a mapping from inputs x to outputs y, given a labeled set of input-output pairs.

Examples of supervised learning problems include classification and regression, and examples of supervised learning algorithms include logistic regression and random forest.

There is another paradigm of learning where the model is only given the input variables (X) and the problem does not have any output variables (y). A model is constructed by extracting or summarizing the patterns in the input data. There is no correction of the model, as the model is not predicting anything.

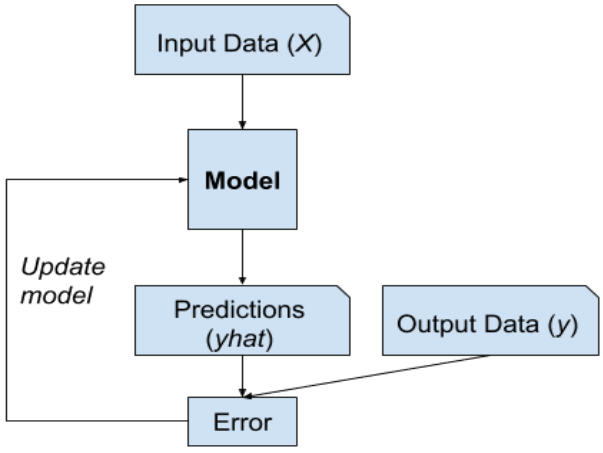


Figure 9.1 Example of Supervised Learning

The second main type of machine learning is the descriptive or unsupervised learning approach. Here we are only given inputs, and the goal is to find "interesting patterns" in the data. […] This is a much less well-defined problem, since we are not told what kinds of patterns to look for, and there is no obvious error metric to use (unlike supervised learning, where we can compare our prediction of y for a given x to the observed value). This lack of correction is generally referred to as an unsupervised form of learning or unsupervised learning.

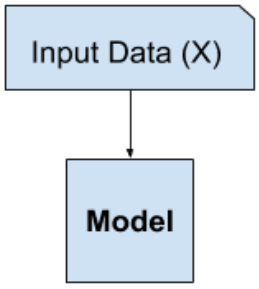


Figure 9.2 Unsupervised Learning

Example of Unsupervised Learning:

Examples of unsupervised learning problems include clustering and generative modeling, and examples of unsupervised learning algorithms are K-means and Generative Adversarial Networks.

9.1.2 Discriminative vs. Generative Modeling:

In supervised learning, we may be interested in developing a model to predict a class label given an example of input variables.

This predictive modeling task is called classification.

Classification is also traditionally referred to as discriminative modeling. We use the training data to find a discriminant function f(x) that maps each x directly onto a class label, thereby combining the inference and decision stages into a single learning problem.

This is because a model must discriminate examples of input variables across classes; it must choose or make a decision as to what class a given example belongs to.

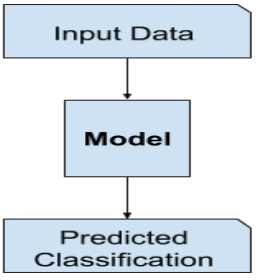


Figure 9.3 Example of Discriminative Modeling

Alternately, unsupervised models that summarize the distribution of input variables may be able to be used to create or generate new examples in the input distribution. As such, these types of models are referred to as generative models.

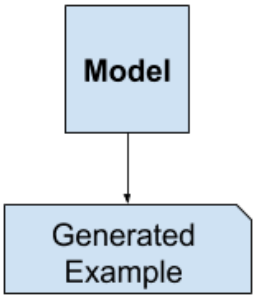


Figure 9.4 Example of Generative Modeling

For example, a single variable may have a known data distribution, such as a Gaussian distribution, or bell shape. A generative model may be able to sufficiently summarize this data distribution, and then be used to generate new variables that plausibly fit into the distribution of the input variable.

Approaches that explicitly or implicitly model the distribution of inputs, as well as outputs, are known as generative models because by sampling from them it is possible to generate synthetic data points in the input space.

In fact, a really good generative model may be able to generate new examples that are not just plausible, but indistinguishable from real examples from the problem domain.
Examples of Generative Models

Naive Bayes is an example of a generative model that is more often used as a discriminative model.

For example, Naive Bayes works by summarizing the probability distribution of each input variable and the output class. When a prediction is made, the probability for each possible outcome is calculated for each variable, the independent probabilities are combined, and the most likely outcome is predicted. Used in reverse, the probability distributions for each variable can be sampled to generate newly plausible (independent) feature values.

Other examples of generative models include Latent Dirichlet Allocation, or LDA, and the Gaussian Mixture Model, or GMM.

Deep learning methods can be used as generative models. Two popular examples include the Restricted Boltzmann Machine, or RBM, and the Deep Belief Network, or DBN.

Two modern examples of deep learning generative modeling algorithms include the Variational Autoencoder, or VAE, and the Generative Adversarial Network, or GAN.

9.2 GENERATIVE ADVERSARIAL NETWORKS

Generative Adversarial Networks, or GANs, are a deep-learning-based generative model.

More generally, GANs are a model architecture for training a generative model, and it is most common to use deep learning models in this architecture.

The GAN architecture was first described in the 2014 paper by Ian Goodfellow, et al. titled "Generative Adversarial Networks."
A standardized approach called Deep Convolutional Generative Adversarial Networks, or DCGAN, that led to more stable models was later formalized by Alec Radford, et al. in the 2015 paper titled "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks".

The GAN model architecture involves two sub-models: a generator model for generating new examples and a discriminator model for classifying whether generated examples are real, from the domain, or fake, generated by the generator model.

1. **Generator**. The model is used to generate new plausible examples from the problem domain.
2. **Discriminator**. The model that is used to classify examples as real (from the domain) or fake (generated).

The generator network directly produces samples. Its adversary, the discriminator network, attempts to distinguish between samples drawn from the training data and samples drawn from the generator.

9.2.1 The Generator Model:

The generator model takes a fixed-length random vector as input and generates a sample in the domain.

The vector is drawn randomly from a Gaussian distribution, and the vector is used to seed the generative process. After training, points in this multidimensional vector space will correspond to points in the problem domain, forming a compressed representation of the data distribution.

This vector space is referred to as a latent space, or a vector space comprised of latent variables. Latent variables, or hidden variables, are those variables that are important for a domain but are not directly

observable. A latent variable is a random variable that we cannot observe directly.

We often refer to latent variables, or a latent space, as a projection or compression of data distribution. That is, a latent space provides compression or high-level concepts of the observed raw data such as the input data distribution. In the case of GANs, the generator model applies meaning to points in a chosen latent space, such that new points drawn from the latent space can be provided to the generator model as input and used to generate new and different output examples.

Machine-learning models can learn the statistical latent space of images, music, and stories, and they can then sample from this space, creating new artworks with characteristics similar to those the model has seen in its training data.

After training, the generator model is kept and used to generate new samples.

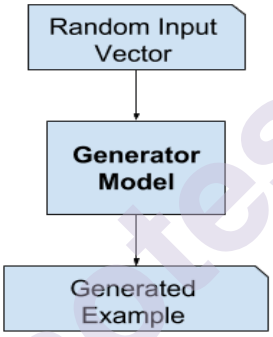


Figure 9.5 Example of the GAN Generator Model

9.2.2 The Discriminator Model:

The discriminator model takes an example from the domain as input (real or generated) and predicts a binary class label of real or fake (generated).

The real example comes from the training dataset. The generated examples are output by the generator model.

The discriminator is a normal (and well understood) classification model. After the training process, the discriminator model is discarded as we are interested in the generator.

Sometimes, the generator can be repurposed as it has learned to effectively extract features from examples in the problem domain. Some or all of the feature extraction layers can be used in transfer learning applications using the same or similar input data.

We propose that one way to build good image representations is by training Generative Adversarial Networks (GANs), and later reusing parts

of the generator and discriminator networks as feature extractors for supervised tasks

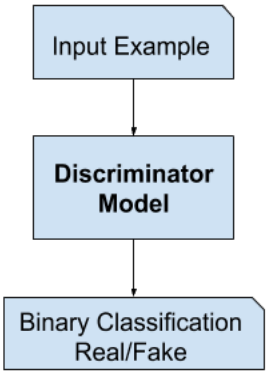


Figure 9.6 Example of the GAN Discriminator Model

9.3 GANS AS A TWO-PLAYER GAME

Generative modeling is an unsupervised learning problem, as we discussed in the previous section, although a clever property of the GAN architecture is that the training of the generative model is framed as a supervised learning problem.

The two models, the generator and discriminator, are trained together. The generator generates a batch of samples, and these, along with real examples from the domain, are provided to the discriminator and classified as real or fake.

The discriminator is then updated to get better at discriminating real and fake samples in the next round, and importantly, the generator is updated based on how well, or not, the generated samples fooled the discriminator. We can think of the generator as being like a counterfeiter, trying to make fake money, and the discriminator as being like police, trying to allow legitimate money and catch counterfeit money. To succeed in this game, the counterfeiter must learn to make money that is indistinguishable from genuine money, and the generator network must learn to create samples that are drawn from the same distribution as the training data.

In this way, the two models are competing against each other, they are adversarial in the game theory sense, and are playing a zero-sum game.

Because the GAN framework can naturally be analyzed with the tools of game theory, we call GANs “adversarial”.

In this case, zero-sum means that when the discriminator successfully identifies real and fake samples, it is rewarded or no change is needed to the model parameters, whereas the generator is penalized with large updates to model parameters.

Alternately, when the generator fools the discriminator, it is rewarded, or no change is needed to the model parameters, but the discriminator is penalized and its model parameters are updated.

At a limit, the generator generates perfect replicas from the input domain every time, and the discriminator cannot tell the difference and predicts "unsure" (e.g. 50% for real and fake) in every case. This is just an example of an idealized case; we do not need to get to this point to arrive at a useful generator model.

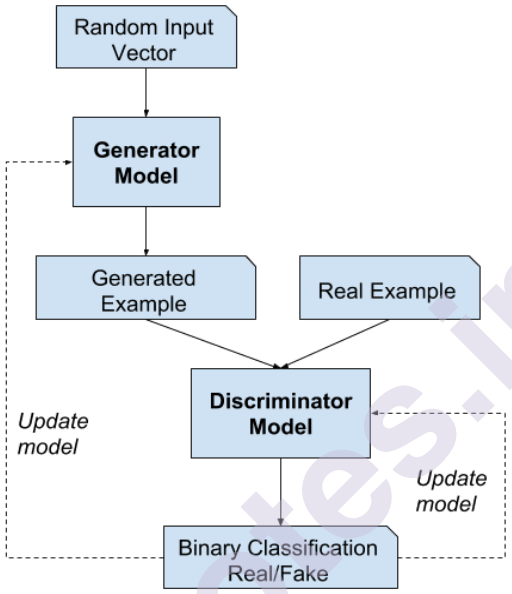


Figure 9.7 Example of the Generative Adversarial Network Model Architecture

Training drives the discriminator to attempt to learn to correctly classify samples as real or fake. Simultaneously, the generator attempts to fool the classifier into believing its samples are real. At convergence, the generator's samples are indistinguishable from real data, and the discriminator outputs 1/2 everywhere. The discriminator may then be discarded.

9.3.1 GANs and Convolutional Neural Networks:

GANs typically work with image data and use Convolutional Neural Networks, or CNNs, as the generator and discriminator models.

The reason for this may be both because the first description of the technique was in the field of computer vision and used CNNs and image data, and because of the remarkable progress that has been seen in recent years using CNNs more generally to achieve state-of-the-art results on a suite of computer vision tasks such as object detection and face recognition.

Modeling image data means that the latent space, the input to the generator, provides a compressed representation of the set of images or photographs used to train the model. It also means that the generator generates new images or photographs, providing an output that can be easily viewed and assessed by developers or users of the model.
It may be this fact above others, the ability to visually assess the quality of the generated output, that has both led to the focus of computer vision applications with CNNs and on the massive leaps in the capability of GANs as compared to other generative models, deep learning-based or otherwise.

9.3.2 Conditional GANs:

An important extension to the GAN is in its use for conditionally generating an output.

The generative model can be trained to generate new examples from the input domain, where the input, the random vector from the latent space, is provided with (conditioned by) some additional input.

The additional input could be a class value, such as male or female in the generation of photographs of people, or a digit, in the case of generating images of handwritten digits.

Generative adversarial nets can be extended to a conditional model if both the generator and discriminator are conditioned on some extra information y. y could be any kind of auxiliary information, such as class labels or data from other modalities. We can perform the conditioning by feeding y into both the discriminator and generator as [an] additional input layer.

The discriminator is also conditioned, meaning that it is provided both with an input image that is either real or fake and the additional input. In the case of a classification label type conditional input, the discriminator would then expect that the input would be of that class, in turn teaching the generator to generate examples of that class in order to fool the discriminator.

In this way, a conditional GAN can be used to generate examples from a domain of a given type.

Taken one step further, the GAN models can be conditioned on an example from the domain, such as an image. This allows for applications of GANs such as text-to-image translation, or image-to-image translation. This allows for some of the more impressive applications of GANs, such as style transfer, photo colorization, transforming photos from summer to winter or day to night, and so on.

In the case of conditional GANs for image-to-image translation, such as transforming day to night, the discriminator is provided examples of real

and generated nighttime photos as well as (conditioned on) real daytime photos as input. The generator is provided with a random vector from the latent space as well as (conditioned on) real daytime photos as input.

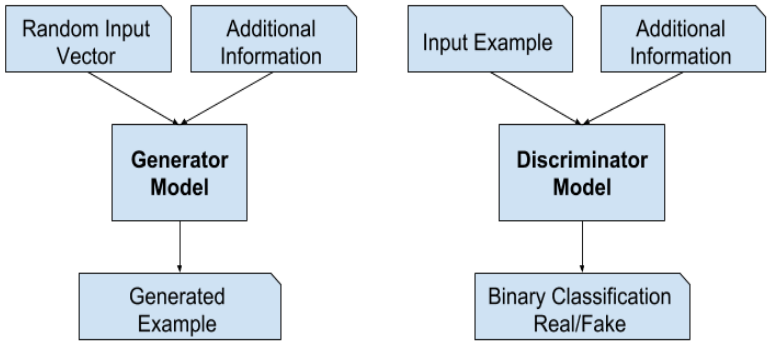


Figure 9.8 Example of a Conditional Generative Adversarial Network Model Architecture

One of the many major advancements in the use of deep learning methods in domains such as computer vision is a technique called data augmentation.

Data augmentation results in better-performing models, both increasing model skill and providing a regularizing effect, reducing generalization error. It works by creating new, artificial but plausible examples from the input problem domain on which the model is trained.

The techniques are primitive in the case of image data, involving crops, flips, zooms, and other simple transforms of existing images in the training dataset.

Successful generative modeling provides an alternative and potentially more domain-specific approach for data augmentation. In fact, data augmentation is a simplified version of generative modeling, although it is rarely described this way.

In complex domains or domains with a limited amount of data, generative modeling provides a path towards more training for modeling. GANs have seen much success in this use case in domains such as deep reinforcement learning.

Among these reasons, he highlights GANs' successful ability to model high-dimensional data, handle missing data, and the capacity of GANs to provide multi-modal outputs or multiple plausible answers.

Perhaps the most compelling application of GANs is in conditional GANs for tasks that require the generation of new examples.

- Image Super-Resolution. The ability to generate high-resolution versions of input images.

- Creating Art. The ability to create new and artistic images, sketches, paintings, and more.
- Image-to-Image Translation. The ability to translate photographs across domains, such as day to night, summer to winter, and more.

Perhaps the most compelling reason that GANs are widely studied, developed, and used is because of their success. GANs have been able to generate photos so realistic that humans are unable to tell that they are of objects, scenes, and people that do not exist in real life.

Astonishing is not a sufficient adjective for their capability and success.

9.4 BOLTZMANN MACHINES

Boltzmann Machines is an unsupervised DL model in which every node is connected to every other node. That is, unlike the ANNs, CNNs, RNNs, and SOMs, the Boltzmann Machines are undirected (or the connections are bidirectional). Boltzmann Machine is not a deterministic DL model but a stochastic or generative DL model. It is rather a representation of a certain system. There are two types of nodes in the Boltzmann Machine — Visible nodes — those nodes which we can and do measure, and the Hidden nodes – those nodes which we cannot or do not measure. Although the node types are different, the Boltzmann machine considers them as the same and everything works as one single system. The training data is fed into the Boltzmann Machine and the weights of the system are adjusted accordingly. Boltzmann machines help us understand abnormalities by learning about the working of the system in normal conditions.

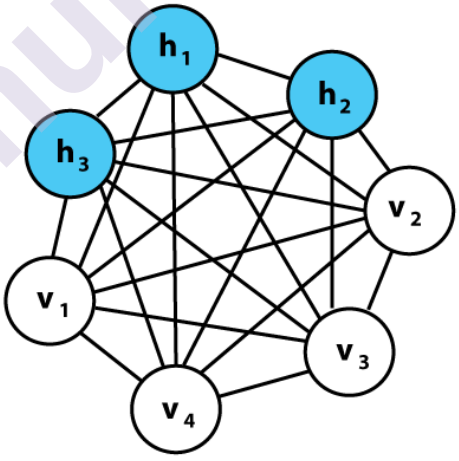


Figure 9.9 Boltzmann Machine

Boltzmann Distribution is used in the sampling distribution of the Boltzmann Machine. The Boltzmann distribution is governed by the equation –

$P_i = e^{(-\epsilon i/kT)} / \sum e^{(-\epsilon j/kT)}$

P_i - the probability of the system being in state i

ϵ_i - Energy of system in state i

T - Temperature of the system

k - Boltzmann constant

$\sum e^{(-\epsilon j/kT)}$ - Sum of values for all possible states of the system

9.5 RESTRICTED BOLTZMANN MACHINES

In a full Boltzmann machine, each node is connected to every other node and hence the connections grow exponentially. This is the reason we use RBMs. The restrictions in the node connections in RBMs are as follows –

- Hidden nodes cannot be connected to one another.
- Visible nodes connected to one another.

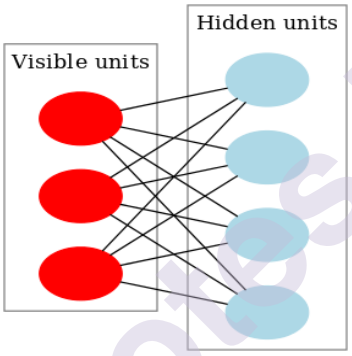


Figure 9.10 Boltzmann Machine units structure

9.5.1 Energy function example for Restricted Boltzmann Machine:

$E(v, h) = -\sum a_i v_i - \sum b_j h_j - \sum\sum v_i w_{i,j} h_j$

a, v - biases in the system - constants

v_i, h_j - visible node, hidden node

P(v, h) = Probability of being in a certain state

$P(v, h) = e^{(-E(v, h))}/Z$

Z - sum if values for all possible states

9.5.2 Features of Restricted Boltzmann Machine:

- They use recurrent and symmetric structures.
- RBMs in their learning process try to associate a high probability with low energy states and vice-versa.
- There are no intralayer connections.
- It is an unsupervised learning algorithm i.e, it makes inferences from input data without labeled responses.

9.5.3 Working of Restricted Boltzmann Machine:

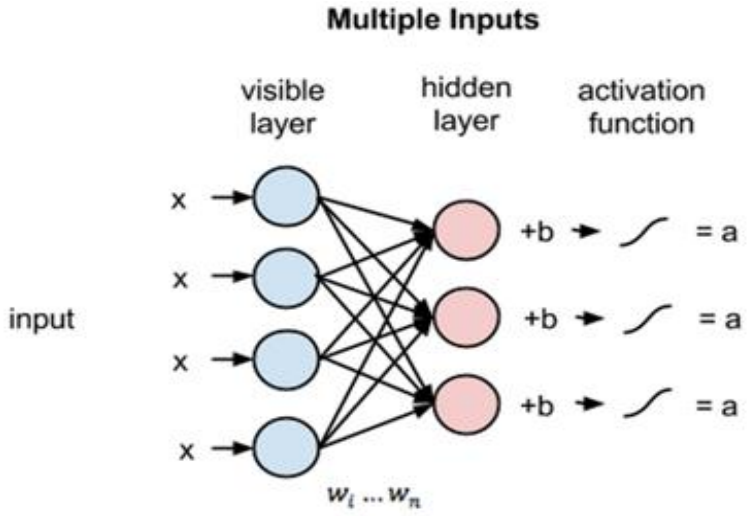


Figure 9.11 working of Restricted Boltzmann Machine

The above image shows the first step in training an RBM with multiple inputs. The inputs are multiplied by the weights and then added to the bias. The result is then passed through a sigmoid activation function and the output determines if the hidden state gets activated or not. Weights will be a matrix with the number of input nodes as the number of rows and the number of hidden nodes as the number of columns. The first hidden node will receive the vector multiplication of the inputs multiplied by the first column of weights before the corresponding bias term is added to it.

And if you are wondering what a sigmoid function is, here is the formula:

$$S(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{1+e^x}$$

So the equation that we get in this step would be,

$$\mathbf{h}^{(1)} = S(\mathbf{v}^{(0)T} W + \mathbf{a})$$

where h(1) and v(0) are the corresponding vectors (column matrices) for the hidden and the visible layers with the superscript as the iteration v(0) means the input that we provide to the network) and a is the hidden layer bias vector. (Note that we are dealing with vectors and matrices here and not one-dimensional values.)

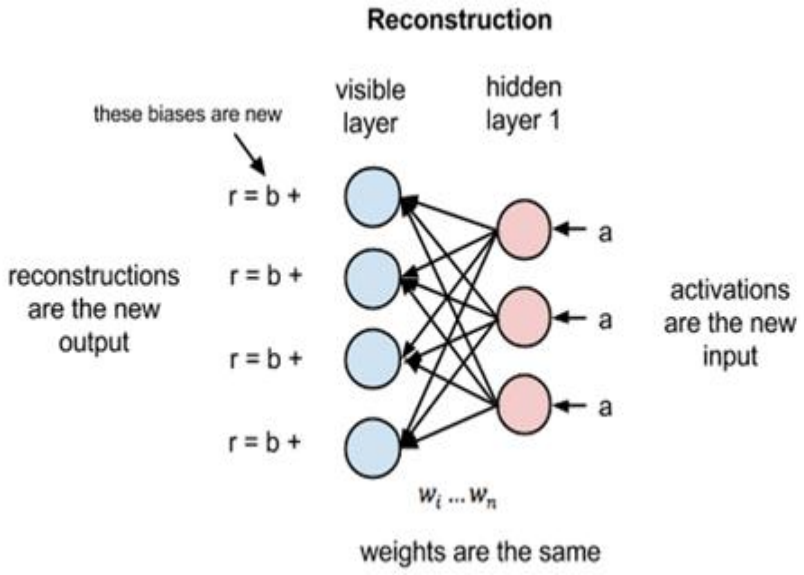


Figure 9.12 reconstructions in Restricted Boltzmann Machine

Now this image shows the reverse phase or the reconstruction phase. It is similar to the first pass but in the opposite direction. The equation comes out to be:

$$\mathbf{v}^{(1)} = S(\mathbf{h}^{(1)}W^T + \mathbf{b})$$

where v(1) and h(1) are the corresponding vectors (column matrices) for the visible and the hidden layers with the superscript as the iteration and b is the visible layer bias vector.

9.5.4 The learning process:

Now, the difference v(0)−v(1) can be considered as the reconstruction error that we need to reduce in subsequent steps of the training process. So the weights are adjusted in each iteration so as to minimize this error and this is what the learning process essentially is. Now, let us try to understand this process in mathematical terms without going too deep into mathematics. In the forward pass, we are calculating the probability of output h(1) given the input v(0) and the weights W denoted by:

$$p(\mathbf{h}^{(1)} \mid \mathbf{v}^{(0)}; W)$$

And in the backward pass, while reconstructing the input, we are calculating the probability of output v(1) given the input h(1) and the weights W denoted by:

$$p(\mathbf{v}^{(1)} \mid \mathbf{h}^{(1)}; W)$$

The weights used in both the forward and the backward pass are the same. Together, these two conditional probabilities lead us to the joint distribution of inputs and the activations:

$$p(\mathbf{v}, \mathbf{h})$$

Reconstruction is different from regression or classification in that it estimates the probability distribution of the original input instead of associating a continuous/discrete value to an input example. This means it is trying to guess multiple values at the same time. This is known as generative learning as opposed to discriminative learning that happens in a classification problem (mapping input to labels).

9.5.5Advantages and Disadvantages of RBM:

9.5.5.1 Advantages:

- Expressive enough to encode any distribution and computationally efficient.
- Faster than traditional Boltzmann Machine due to the restrictions in terms of connections between nodes.
- Activations of the hidden layer can be used as input to other models as useful features to improve performance

9.5.5.1 Disadvantages:

- Training is more difficult as it is difficult to calculate the Energy gradient function.
- The CD-k algorithm used in RBMs is not as familiar as the backpropagation algorithm.
- Weight Adjustment

9.5.6 Applications of RBM:

- Hand Written Digit Recognition
- Real-time intrapulse recognition of radar

9.5.7 Difference between Autoencoders & RBMs:

Autoencoder is a simple 3-layer neural network where output units are directly connected back to input units. Typically, the number of hidden units is much less than the number of visible ones. The task of training is to minimize an error or reconstruction, i.e. find the most efficient compact representation for input data.

RBM shares a similar idea, but it uses stochastic units with particular distribution instead of deterministic distribution. The task of training is to

find out how these two sets of variables are actually connected to each other.

One aspect that distinguishes RBM from other autoencoders is that it has two biases. The hidden bias helps the RBM produce the activations on the forward pass, while the visible layer's biases help the RBM learn the reconstructions on the backward pass.

9.6 DEEP BELIEF NETWORKS

Deep belief nets are probabilistic generative models that are composed of multiple layers of stochastic, latent variables. The latent variables typically have binary values and are often called hidden units or feature detectors. The top two layers have undirected, symmetric connections between them and form an associative memory. The lower layers receive top-down, directed connections from the layer above. The states of the units in the lowest layer represent a data vector.

The two most significant properties of deep belief nets are:

There is an efficient, layer-by-layer procedure for learning the top-down, generative weights that determine how the variables in one layer depend on the variables in the layer above.

After learning, the values of the latent variables in every layer can be inferred by a single, bottom-up pass that starts with an observed data vector in the bottom layer and uses the generative weights in the reverse direction.

Deep belief nets are learned one layer at a time by treating the values of the latent variables in one layer, when they are being inferred from data, as the data for training the next layer. This efficient, greedy learning can be followed by, or combined with, other learning procedures that fine-tune all of the weights to improve the generative or discriminative performance of the whole network.

Discriminative fine-tuning can be performed by adding a final layer of variables that represent the desired outputs and backpropagating error derivatives. When networks with many hidden layers are applied to highly structured input data, such as images, backpropagation works much better if the feature detectors in the hidden layers are initialized by learning a deep belief net that models the structure in the input data

Deep Belief Networks have two phases:

- Pre-train Phase
- Fine-tune Phase

The pre-train phase is nothing but multiple layers of RBNs, while Fine Tune Phase is a feed-forward neural network. Let us visualize both the steps:-

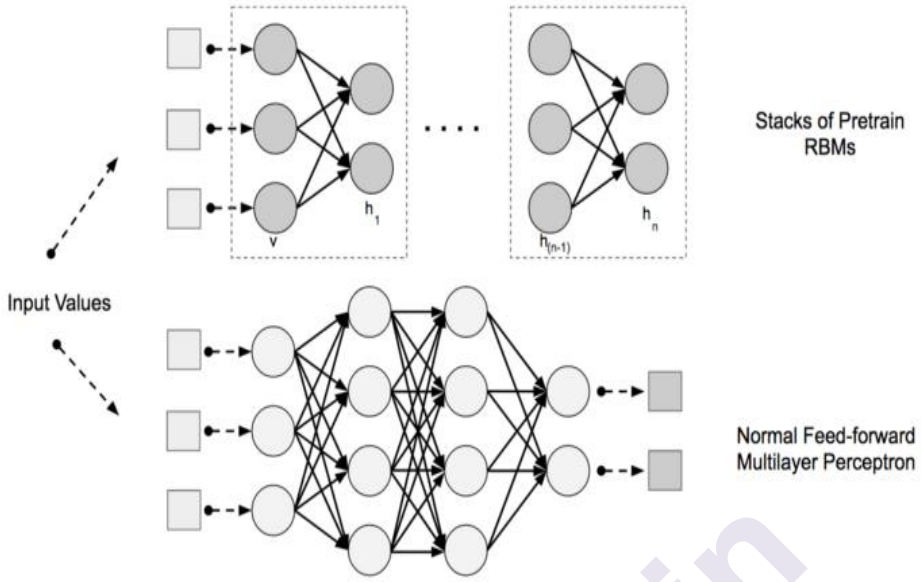


Figure 9.13 Deep Belief Networks

Algorithm for Training DBN:

Input: Dataset

Output: Trained network

Step 1: Train the first layer as RBM that models the input $a = h^{(0)}$ as its visible layer.

Step 2: By using the first layer obtain the representation of the input that will be used as

input for the next layer

$p(h^{(0)})$ or $p(h^{(0)})$

Step 3: Train the second layer as an RBM

Step 4: Repeat step 2 and step 3 for all the number of layers

Steps to update the weight

Input: Random Weight initialized

Output: Wight updated based on the error rate

Update the weight of Edge

$$upd(w_{ij}+\eta/2*(\text{positive}(E_{ij})-\text{negative}(E_{ij}))$$

Positive phase:

Compute positive statistics for edge E_{ij}

Positive (Eij)⇒ $p(HJ=1|v)$

The individual activation probabilities for the hidden layer is

$$p(H_J=1|v) = \sigma(B_j+\sum_{i=1}^{m} w_{ij} v_i)$$

Negative Phase:

Compute negative statistics for edge E_{ij}

Negative (Eij)⇒ $p(v_i=1|H)$

The individual activation probabilities for visible layer is

$$p(v_i=1|H) = \sigma(A_i+\sum_{j=1}^{n} w_{ij} H_j)$$

Deep belief networks can substitute for a deep feedforward network or, given more complex arrangements, even a convolutional neural network. They have the advantages that they are less computationally expensive (they grow linearly in computational complexity with the number of layers, instead of exponentially, as with feedforward neural networks); and that they are significantly less vulnerable to the vanishing gradients problem.

However, because deep belief networks impart significant restrictions on their weight connections, they are also vastly less expressive than a deep neural network, which outperforms them on tasks for which sufficient input data is available.

Even in their prime, deep belief networks were rarely used in direct application. They were instead used as a pretraining step: a deep belief network with the same overall architecture as a corresponding deep neural network is defined and trained. Its weights are then taken and placed into the corresponding deep neural network, which is then fine-tuned and put to application.

Deep belief networks eventually fell out of favor in this application as well. For one thing, RBMs are just a special case of autoencoders, which were found to be more broadly flexible and useful both for pretraining and for other applications. For another thing, the introduction of ReLU activation and its further refinement into leaky ReLU, along with the introduction of more sophisticated optimizers, learning late schedulers, and dropout techniques, have worked to greatly alleviate the vanishing gradient problem in practice, at the same time that increased data volumes and computes power have made direct deep neural network applications to problems more tractable.

9.7 SUMMARY

- We have discovered a large number of applications of Generative Adversarial Networks, or GANs.
- Broad catagory of GAN are live supervised and unsupervised
- The GAN model architecture involves two sub-models: a generator and discriminator
- The generator model takes a fixed-length random vector as input and generates a sample in the domain
- The discriminator model takes an example from the domain as input (real or generated) and predicts a binary class label of real or fake (generated).
- Boltzmann Machines is an unsupervised DL model in which every node is connected to every other node.
- Deep belief nets are probabilistic generative models that are composed of multiple layers of stochastic, latent variables.

Experiment Practice:

- **Apply Restricted Boltzmann Machine implementation for Recommender System (Amazon product suggestions or Netflix movie)**
- **Apply Deep Belief Networks implementation for Recommender System (Amazon product suggestions or Netflix movie)**

LIST OF REFERENCES

[1] Heskes, Tom & Albers, Kees & Kappen, Hilbert. (2012). Approximate Inference and Constrained Optimization. Proceedings of Uncertainty in AI.

[2] Shin Kamada, Takumi Ichimura.: An adaptive learning method of Deep Belief Network by layer generation algorithm. IEEE Xplore. Nov 2016.

[3] Shin Kamada, Takumi Ichimura.: An adaptive learning method of Deep Belief Network by layer generation algorithm. IEEE Xplore. Nov 2016.

[4] Mohamed A, Dahl G E, Hinton. G.: Acoustic modeling using deep belief networks[J]. Audio Speech and Language Processing IEEE Transactions an, vol. 20, no. 1, pp. 14-22, 2012

[5] Boureau Y, Cun L. Y.: Sparse feature learning for deep belief networks [C] //Advances in neural information processing systems. pp. 1185-1192, 2008

[6] Hinton. G.: A practical guide to training restricted Boltzmann machines[J]. Momentum, vol. 9, no. 1, pp. 926, 2010

[7] Bengio Y, Lamblin P, Popovici Det al.:Greedy layer-wise training of deep networks. Advances in neural information processing systems, vol. 19, pp. 153, 2007.

[8] Ranzato A M, Szummer M.: Semi-supervised learning of compact document representations with deep networks[C]//Proceedings of the 25th international conference on Machine learning. ACM, pp. 792-799, 2009.

[9] Neal R M, Hinton G. E.: A view of the EM algorithm that justifies incremental sparse and other variants[M]//Learning in graphical models. pp. 355-368, 1999.

[10] Hinton G E, Salakhutdinov R. R.: Reducing the dimensionality of data with neural networks[J]. Science, vol. 313, no. 5786, pp 504-507, 2006.

[11] Goodfellow and Bengio, "Deep learning " 2016

[12] Goodfellow, Ian, et al. "Generative adversarial nets." Advances in neural information processing systems. 2014.

[13] Kingma, Diederik P., and Max Welling. "Auto-encoding variational bayes." arXiv preprint arXiv:1312.6114 (2013).

[14] Radford, Alec, Luke Metz, and Soumith Chintala. "Unsupervised representation learning with deep convolutional generative adversarial networks." arXiv preprint arXiv:1511.06434 (2015).

[15] Nowozin, Sebastian, Botond Cseke, and Ryota Tomioka. "f-GAN: Training generative neural samplers using variational divergence

minimization." Advances in Neural Information Processing Systems. 2016.

[16] Denton, Emily L., Soumith Chintala, and Rob Fergus. "Deep Generative Image Models using a Laplacian Pyramid of Adversarial Networks."

Advances in neural information processing systems. 2015.

[17] Chen, Xi, et al. "Infogan: Interpretable representation learning by information maximizing generative adversarial nets." Advances in Neural

Information Processing Systems. 2016.

[18] Zhao, Junbo, Michael Mathieu, and Yann LeCun. "Energy-based generative adversarial network." arXiv preprint arXiv:1609.03126 (2016).

[19] Doersch, Carl. "Tutorial on variational autoencoders." arXiv preprint arXiv:1606.05908 (2016).

[20] Wang, K-C., and Richard Zemel. "classifying NBA offensive plays using neural networks." MIT Sloan Sports Analytics Conference, 2016.

[21] Wikipedia "Kernel density estimation"

[22] Wikipedia “Principal component analysis”

[23] Wikipedia “Autoencoder

MODEL QUESTIONS

1. Explain how approximate Inference works machine learning
2. Explain relationship between approximate Inference deep learning
3. Define Posterior Probability
4. Explain Expectation Maximization algorithm
5. Write Expectation Maximization algorithm
6. Write Maximum a Posteriori (MAP) algorithm
7. Compare Supervised and Unsupervised Learning
9. Compare Discriminative Generative Modeling
9. Explain Generative Adversarial Networks in detail.
9. Explain the Generator Model and the Discriminator Model of GAN
11. Explain Conditional GANs
12. Write short note on Boltzmann Machines
13. Explain Restricted Boltzmann Machines
14. Write Features of Restricted Boltzmann Machine
15. Compare Boltzmann Machines and Restricted Boltzmann Machine
16. Write Advantages and Disadvantages of RBM
17. Write Applications of RBM also explain any one
19. Explain Deep Belief Networks with its working.
19. Write Algorithm for Training DBN
20. Explain how weights are updated in DBN.

*****