UNIT I

CHAPTER 1

Fundamentals of Deep Learning

Syllabus

What is Deep Learning?, Multilayer Perceptron ,Feed forward neural, Back propagation, Gradient descent, Vanishing gradient problem, Activation Functions: RELU, LRELU, ERELU, Optimization Algorithms, Hyper parameters: Layer size, Magnitude (momentum, learning rate),Regularization (dropout, drop connect, L1, L2)

1.1 INTRODUCTION TO DEEP LEARNING

- **Deep Learning** is a type of machine learning that imitates the way humans gain certain type of knowledge. It is a field that is based on learning and improving on its own by examining computer algorithms.
- Deep Learning is a subset of Artificial Intelligence also an important element of data science, which includes statistics and predictive modelling.
- Deep learning gets its name from the fact that it involves going deep into several layers of network, which also includes a hidden layer. The deeper you dive; you more complex information you extract.
- Deep learning methods rely on various complex programs to imitate human intelligence. This particular method teaches machines to recognise ideas so that they can be classified into distinct categories. To understand deep learning, imagine a toddler whose first word is dog.

- The toddler learns what a dog is and is not by pointing to objects and saying the word dog.
- The parent says, "Yes, that is a dog," or, "No, that is not a dog." As the toddler continues to point to objects, he becomes more aware of the features that all dogs possess.
- What the toddler does, without knowing it, is clarify a complex abstraction "the concept of dog" by building a hierarchy in which each level of abstraction is created with knowledge that was gained from the preceding layer of the hierarchy.

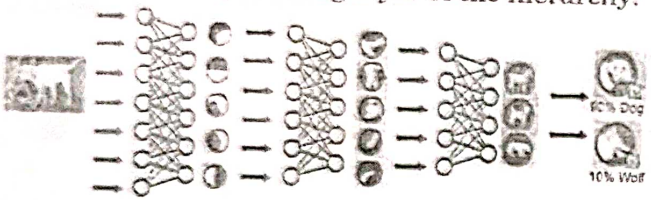


Fig. 1.1.1 : Understanding deep learning

1.2 MACHINE LEARNING VS DEEP LEARNING

- Deep learning is a type of machine learning that differs in how it solves problems. While traditional machine learning algorithms are linear, deep learning algorithms are highly complex and non-linear. To discover the most widely used features in machine learning, a domain expert is required.
- Deep learning, on the other hand, learns features incrementally, obviating the need for domain knowledge. Deep learning algorithms, on the other hand, require far longer to train than machine learning algorithms, which take anywhere from a few seconds to a few hours.
- During testing, on the other hand, the opposite is true. Deep learning algorithms conduct tests at a fraction of the time it takes machine learning algorithms, whose test time increases as the size of the data grows.
- Thus, deep learning is a specialised subset of machine learning. The Table 1.2.1 depicts the difference between Machine learning and Deep Learning.

1.2.1 Difference between Machine Learning & Deep Learning

Table 1.2.1

Sr. No.	Machine Learning	Deep Learning
1.	Machine Learning is a superset of Deep Learning	Deep Learning is a subset of Machine Learning
2.	Uses various types of automated algorithms that turn to model functions and predict future action from data.	Uses neural network that passes data through processing layers to the interpret data features and relations.
3.	The data representation in Machine Learning is quite different as compared to Deep Learning as it uses structured data	The data representation is used in Deep Learning is quite different as it uses neural networks (ANN).
4.	Machine Learning algorithms are linear.	Deep Learning algorithms are complex and non-linear.
5.	Machine learning consists of thousands of data points.	Deep learning works on Big Data, so millions of data points.
6.	To find best set of features in machine learning, domain expert is required.	Deep learning learns features incrementally, obviating the need for domain knowledge.
7.	Time required to train/test a machine learning model is comparatively very less than a deep learning model.	Time required to train/test a deep learning model is high as more layers are present.
8.	Not necessary to have costly high-end machines.	High-end machines and High performing GPUs are required.
9.	Outputs of a machine learning model can be some class label or numerical Value, like classification score.	Output of a deep learning model can be anything from numerical values to free-form elements, such as free text and sound.
10.	Algorithms are detected by data analysts to examine specific variables in data sets.	Algorithms are largely self-depicted on data analysis once they're put into production.

1.3 PERCEPTRON

Here, we shall study topics involving supervised learning networks and the associated single-layer and multilayer feed-forward networks. Also we shall discuss the perceptron learning rule for simple perceptron, the delta-rule and the back-propagation algorithms.

1.3.1 Perceptron Networks

Single-layer feed-forward network is also called as simple perception.

Single layer feedforward network

This type of network comprises two layers, the **input layer** and the **output layer**. The **input layer neurons** receive the input signals and the **output layer** neurons receive the output signals.

The synaptic links carrying the weights connect every **input** neuron to the **output neuron** but not in the reverse way. Such a network is said to be feedforward in type or acyclic in nature. Since the output single layer alone performs computation, hence it is called as single layer feedforward network.

1.4 KEY-POINTS

In a Perceptron network, the key points are

(i) Perceptron-network consists of three units, namely, sensory unit (input unit), associate unit (hidden unit), response unit (output unit).

(ii) The sensory units are connected to associator units with fixed weights having values 1, 0 or – 1, which are given at random.

(iii) The binary step is used with fixed threshold θ as activation for associator.

(iv) The response unit has an activation of 1, 0 or – 1.

(v) The output of the perception network is given by,

$$y = f(y_{in}), \text{ where}$$

$f(y_{in})$ is the activation function defined by

$$f(y_{in}) = \begin{cases} 1, & \text{if } y_{in} > \theta \\ 0, & \text{if } -\theta \le y_{in} \le \theta \\ -1, & \text{if } y_{in} < -\theta \end{cases}$$

y_{in} is the input unit

(vi) The perceptron learning rule is used in the weight updation between the associator unit and the response unit.

(vii) The weights on the connections from the units that send the non-zero signal will get adjusted suitably.

(viii) The weights will be adjusted if an error has occurred for a particular training pattern, i.e.,

$$w_i \text{ (new)} = w_i \text{ (old)} + \alpha + xi \quad \text{and} \quad b \text{ (new)}$$
$$= b \text{ (old)} + \alpha \cdot t$$

Here 't' is target value; and it is + 1 or – 1 and 'α' is the learning rate.

1.4.1 The Perceptron Rule Convergence Theorem

The theorem states that : "If there is a weight vector W, such that f(x(n) w) = t(n), for all n, then for any starting vector w_1, the perceptron learning rule will converge to a weight vector that gives the correct response for all training patterns, and this learning takes place within a finite number of steps provided that the solutions exists.

Refer Fig. 1.4.1.

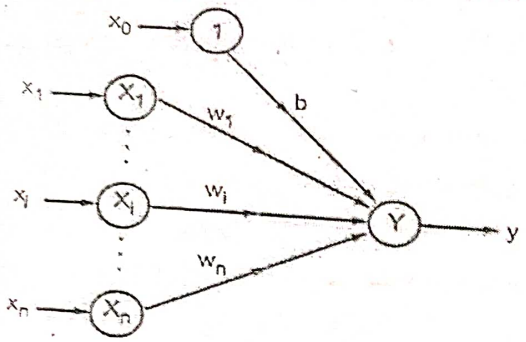


Fig. 1.4.1 : Single classification perceptron network :
[A simple perceptron network architecture]

1.5 ARCHITECTURE OF SIMPLE PERCEPTRON NETWORK

In the original perceptron network, the output obtained from the associator units is a binary vector, and hence that output can be taken as input signal to the response unit, and then classification can be made.

In Fig. 1.4.1, a simple perceptron network architecture is shown.

In the above diagram, there are 'n' input neurons, 1 output neuron and a bias. The input-layer and output-layer neurons are connected through a directed communication link, which is given weights. The aim of the perceptron net is to classify the input pattern as a member.

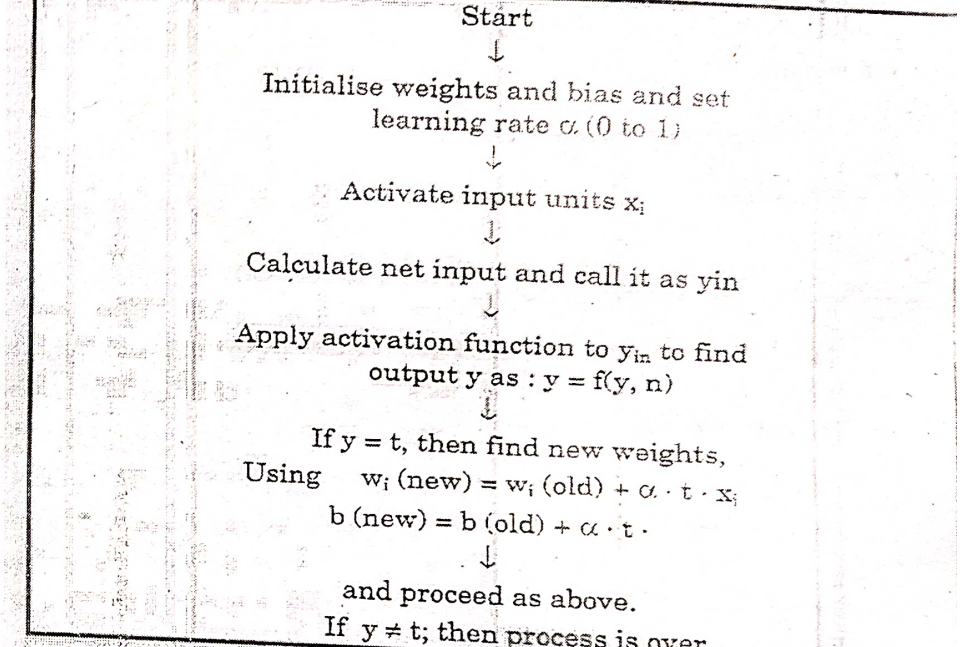
1.5.1 Flowchart for Training Process

The training is done on the basis of the comparison between the calculated and desired output. The loop stands terminated it there is no change in weights.

Remark

First one has to perform the basic. Initialisation required for the training process.

Flowchart for Perceptron network with single output



1.6 PERCEPTRON TRAINING ALGORITHM FOR SINGLE OUTPUT CLASSES

In the following algorithm the inputs are initially assigned and then the net input is calculated. Applying activation formula over the calculated net input, the output of the network is obtained.

We compare the calculated and desired output, and then carry out the weight updation. The perceptron algorithm is used for either binary or bipolar input vectors, having bipolar targets, with varying bias and fixed threshold. The entire network is based on stopping criterion.

The algorithm function as follows :

- **Step (I) :** First we initialise weights and bias. For easy calculation, we put them equal to zero. And learning rate α is between : $0 < \alpha \leq 1$. Conveniently, we take $\alpha = 1$.
- **Step (II) :** We perform the steps (III) to (vii) until the final stopping condition is false.
- **Step (III) :** We apply identity activation functions containing the input layer containing input, unit.
- **Step (IV) :** We obtain the net input

$$y_{in} = b + \sum_{i=1}^{n} x_i w_i$$

Where 'n' is the number of input neurons in the input layer.

Now to obtain the output, we apply activation formula over the net input :

$$y = f(y_{in}) = \begin{cases} 1, & \text{if } y_{in} > \theta \\ 0, & \text{if } -\theta \leq y_{in} \leq \theta, \ \theta \text{ is threshold} \\ -1, & \text{if } y_n < -\theta \end{cases}$$

- **Step (V) :** Now we compare the actual output (calculated output) and desired (target) output :

If $y \neq t$, then we use the formulae :

$$w_i \text{ (new)} = w_i \text{ (old)} + \alpha \cdot t \cdot x_i$$

$$b \text{ (new)} = b \text{ (old)} + \alpha \cdot t$$

else, we have

$$w_i \text{ (new)} = w_i \text{ (old)}$$

$$b \text{ (new)} = b \text{ (old)}$$

- **Step (VI) :** We train the network until there is no weight change.

This is called stopping condition for the network. If this condition is not satisfied, we start again from step (IV)

Remark

This algorithm is not sensitive to the initial values of the weights or the valve of the learning rate.

1.7 MULTILAYER FEEDFORWARD NETWORK

- This network is made up of multiple layers. The architecture of this class consists of one or more intermediary layers called as hidden layers, besides having an input and an output layer.
- The hidden layers perform intermediary computations before directing the input to the output layer.
- The input layer neurons are linked to the hidden layer neurons and the weights on these links are called as **input-hidden layer weights.**
- Again, the hidden layer neurons are linked to the output layer neurons and the corresponding weights are referred to as **hidden-output layer weights.**

1.7.1 Perceptron Training Algorithm for Multiple output Classes

We mention below the Algorithm of perceptron training for multiple output classes :

- **Step (I) :** Initialise the weights, biases and learning rate conveniently.
- **Step (II) :** Check for stopping condition; if it is not true, perform steps 3-7.

- **Step (III) :** For each bipolar or binary training vector pair S : t, perform steps 4–6.
- **Step (IV) :** Set activation function (identity function) of each input unit i = 1 to n : xi = si
- **Step (V) :** Calculate the net input as, $yinj = bj + \sum_{i=1}^{n} xi\ wij$

  Then calculate output response of each output unit j = 1 to m : To calculate the output response, apply activation formula over the net input :

$$y_j = f(y_{inj}) = \begin{cases} 1, & \text{if } y_{inj} > \theta \text{ ; } \theta \text{ is threshold valve} \\ 0, & \text{if } -\theta \le y_{inj} \le \theta \\ -1, & \text{if } y_{inj} < -\theta \end{cases}$$

- **Step (VI) :** If $t_j \neq y_j$, then make adjustment in weights and bias for j = 1 to m and i = 1 to n.

i.e. $w_{ij}\ (new) = w_{ij}(old) + \alpha\ t_j\ x_i$

$b_j\ (new) = b_j\ (old) + \alpha\ t_j$

Otherwise we have

$w_{ij}\ (new) = w_{ij}\ (old)$

$b_j\ (new) = b_j\ (old)$

- **Step (VII) :** If there is no change in weights then stop the training process, otherwise begin from step 3. We exhibit the architecture for the above algorithm :

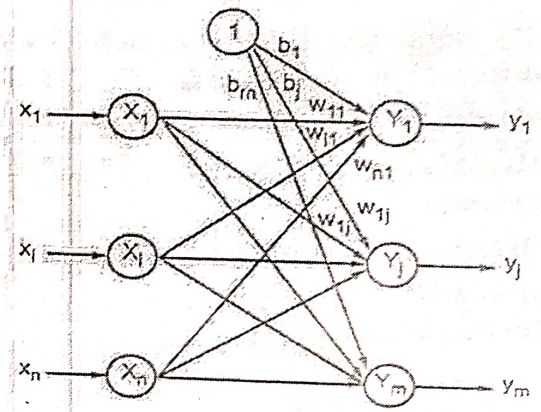


Fig. 1.7.1 : Network architecture for perceptron network for several output classes

1.8 PROGRAM FOR PERCEPTRON TRAINING ALGORITHM

- **Program 1.8.1 :** Write a program for solving linearly separable problem using Perceptron Model.

Source code

```
%Perceptron for AND function
clear;
clc;
x=[1 1 -1 -1;1 -1 1 -1];
t=[1 -1 -1 -1];
w=[0 0];
b=0;
alpha=input('Enter Learning rate=');
theta=input('Enter Threshold value=');
con=1;
epoch=0;
while con
  con=0;
    for i=1:4
      yin=b+x(1,i)*w(1)+x(2,i)*w(2);
        if yin>theta
```

```
        y=1;
        end
        if yin <=theta & yin>=-theta
        y=0;
        end
        if yin<-theta
        y=-1;
        end
        if y-t(i)
        con=1;
          for j=1:2
            w(j)=w(j)+alpha*t(i)*x(j,i);
          end
        b=b+alpha*t(i);
        end
    end
epoch=epoch+1;
end
disp('Perceptron for AND function');
disp('Final Weight matrix');
disp(w);
disp('Final Bias');
disp(b);
save 'result.mat','w','b';
```

Output

```
Enter Learning rate= 0.5
Enter Threshold value= 1
Perceptron for AND function
Final Weight matrix
   1.5000    1.5000
Final Bias
   -1.5000
```

1.8.1 Perceptron Network Testing Algorithm

(I) For efficient performance of the network, it is to be trained with more data.

We mention the testing algorithm :

- **Step (I) :** The initial weights to be used are to be taken from the final weights obtained during training.
- **Step (II) :** Perform steps 3-4 for each input vector X.
- **Step (III) :** Set activations formula of the input unit.
- **Step (IV) :** Obtain the response of output unit : $y_{in} = \sum_{i=1}^{n} x_i w_i$

$$\text{and} \quad y=f(y_{in}) = \begin{cases} 1 & \text{if } y_{inj} > \theta \quad \theta \text{ is threshold} \\ 0 & \text{if } -\theta \le y_{in} \le \theta \\ -1 & \text{if } y_{inj} < -\theta \end{cases}$$

With these steps the performance of the algorithm can be tested.

➢ **Program 1.8.2 :** Test the above trained perceptron network (Using MATLAB)

```
%This is the recall program for perceptron_AND
%Trained weight vector and bias value is saved in file 'result.mat' which is
loaded here using load command
load result.mat;
x=input('Enter the test pattern');
yin=b+x(1)*w(1)+x(2)*w(2);
 if yin>theta
   y=1;
 end
 if yin <=theta & yin>=-theta
   y=0;
 end
 if yin<-theta
  y=-1;
 end
 disp('output is');
 y
```

Output

```
Enter the test pattern
[-1 1]
output is
y= -1
```

1.9 CLASSES OF NEURAL NETWORKS

According to their learning mechanism, Neural Network are classified into several classes. We identify three fundamentally different classes of networks. All these three classes employ the digraph structure for their representation. The arrangement of neurons to form layers to from layers and the connection pattern formed within and between layers is called **network architecture**. There exist five basic types of neuron connection architectures.

They are :

1. Single-layer feed-forward network;
2. Multilayer feed-forward network;
3. Single node with its own feedback;
4. Single-layer recurrent network;
5. Multilayer recurrent network

1.9.1 Single Layer Feed Forward Network

(1) This type of network comprises two layers, namely the **input layer** and the **output layer**.

(2) The **input layer neurons** receive' the input signals,

(3) The **output layer neurons** receive the output signals.

(4) The synaptic links carrying the weights connect every input neuron to the output neuron but not conversely.

(5) This network is called as feed forward in type or acyclic in nature.

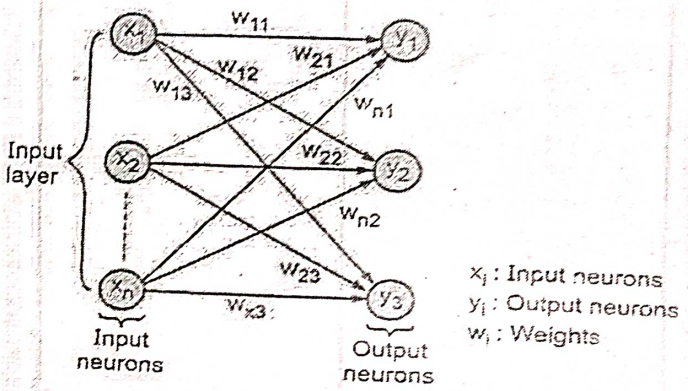


Fig. 1.9.1 : Single layer feedforward network

(6) The network is termed as single layer since the alone output layer, alone which performs computation.

(7) Even though there are two layers, it is called as single layer because of output layer.

(8) The input layer merely transmits the signals to the output layer.

We illustrate an example network as shown in Fig. 1.9.1.

1.9.2 Multilayer Feedforward Network

(1) As the name indicates, this network is made up of multiple layers.

(2) It has intermediary layers besides possessing input and output layers. These layers are called as **hidden layers.**

(3) The computational units of the hidden layer are called as hidden neurons or hidden units.

(4) The hidden layers adds in performing intermediary computations and then input layers are directed to the output layers.

(5) The weights on the links of input layer neurons and hidden layers are called as **Input-hidden layer weights.**

We illustrate this by the Fig. 1.9.2.

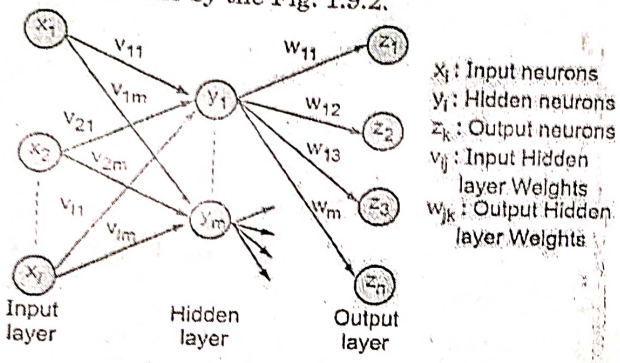


Fig. 1.9.2 : A multilayer feed forward network

1.9.3 Single Node with Its Own Feedback

- When outputs can be directed back as inputs to the same layer or preceding layer nodes, then it results in feedback networks.
- If the feedback of the output of the processing element is directed back as input to the processing element in the same layer then it is called lateral feedback.

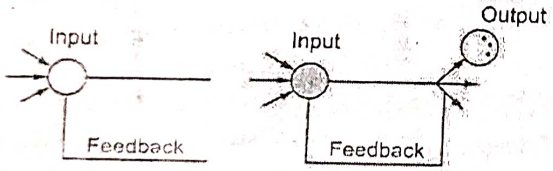


Fig. 1.9.3

1.9.4 Single Layer Recurrent Network

Recurrent networks are feedback networks with closed loop. Fig. 1.9.4 shows a single-layer network with a feedback connection in which a processing elements output can be directed back to the processing element itself or to the other processing element or to both.

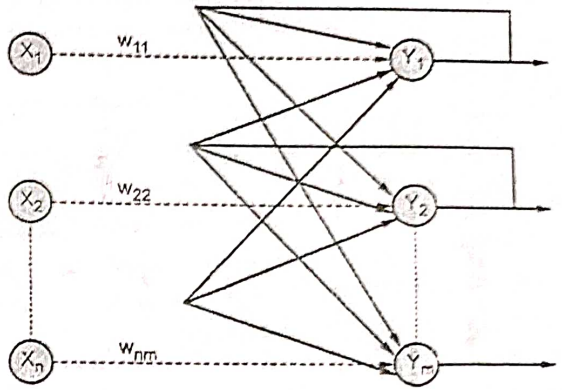


Fig. 1.9.4 : Single-layer recurrent network

1.9.5 Multilayer Recurrent Network

- The main feature of a multiplayer recurrent network is its hidden state, which captures information about a sequence.
- To form a multilayer recurrent network, a processing element output can be directed back to the nodes in a preceding layer. Also, in these networks, a processing element output can be directed back to the processing element itself and to other processing elements in the same layer.

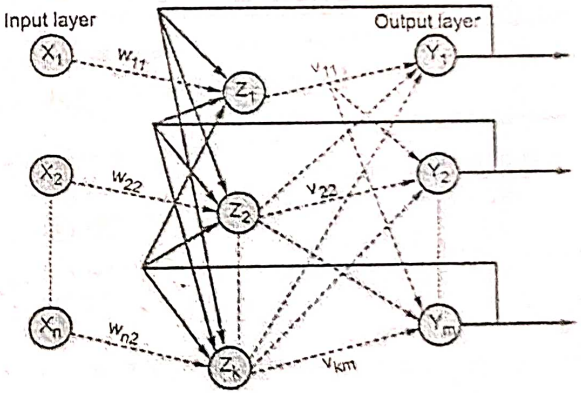
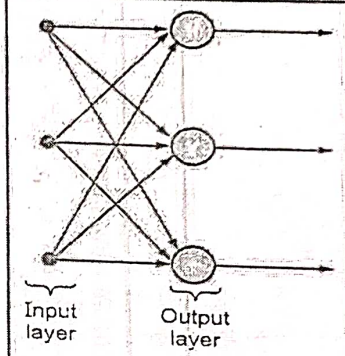
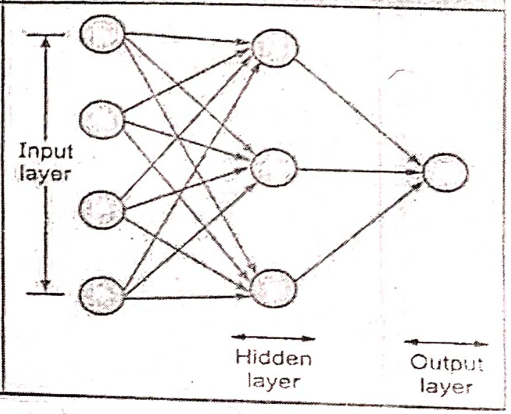


Fig. 1.9.5 : Multiplayer recurrent network

1.9.6 Difference between Multi Layer Feed Forward Neural Network and Signal Layer Feed Forward Neural Network

Sr. No.	Single layer feed forward Neural Network	Multi layer feed forward Neural Network
1.	Formation of layer is done by **processing** and **combining** elements with other processing elements.	Multi layer is formed by **interconnection** of **several** layers.
2.	There is **linking** between **input** and **output** layer.	Between input and output layers, there are **multiple layers,** which are known as **hidden layers.**
3.	Inputs are connected to the **processing** nodes with various **weights** resulting series of output **one for node.**	Input layers receive input and stores input signal and output layer generates output.
4.	There is no hidden layer.	There are **several hidden layers** in a network.
5.	In certain applications, it is **not efficient.**	More hidden layers create **complexity** of network, but the output is **efficient.**

Sr. No.	Single layer feed forward Neural Network	Multi layer feed forward Neural Network
		

1.10 APPLICATIONS OF NEURAL NETWORKS

RQ. What are the performance measures to see whether training of neural network is successful? Explain. (Ref. Q. 4(a) May 2016, 10 Marks)

Variety of problems can be solved by applications of neural networks.

Some of the common applications are :

(1) Pattern Recognition (PR)

(2) Image processing (IP)

(3) Neural Networks are very useful in visual images handwritten characters, printed characters, speech and other PR based tasks.

(4) Optimisation/Constraint Satisfactions

To obtain optimal solution satisfying given constraints, such problems can be solved by NN successfully.

Examples

(i) Manufacturing scheduling, (ii) Finding the shortest possible tour given a set of cities,

(iii) Problems of this nature arising out of industrial and manufacturing fields have been found acceptable solutions using NNS.

(5) **Forecasting and Risk Assessment :** Neural networks have shown capability to predict situations form past trends. There are ample applications in area such as meteorology, banking, stock market, econometrics with high success rates.

(6) **Control system :** NNS have gained commercial ground by applying applications in control systems. Companies incorporating NN technology have produced dozers of computer successfully.

(7) There are many applications for the control of chemical plants, robots and so on.

1.11 IMPORTANT TERMINOLOGIES OF ANN

1.11.1 Weights

(1) Each neuron, in the architecture of ANN, is connected to other neurons by means of direct communication links, and each links is associated with weights.

(2) Weights contain information about the input signal.

(3) This information is used by the net to solve a problem.

(4) The weight can be represented in terms of matrix. The weight matrix is also called as connection matrix.

(5) To make mathematical formulation, let there be 'n' processing elements in an ANN and each processing element has exactly m adaptive weights.

Thus the weight matrix w is defined as :

$$W = \begin{bmatrix} w_1^T \\ w_2^T \\ \vdots \\ w_n^T \end{bmatrix} \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1m} \\ w_{21} & w_{22} & \cdots & w_{2m} \\ \vdots & & & \\ w_{n1} & w_{n2} & \cdots & w_{nm} \end{bmatrix}$$

Where, $w_i = [w_{i1}, w_{i2}, \cdots, w_{im}]^T$

$i = 1$ to n. is the weight vector of processing element

And wij is the weight from processing element 'i' (source node) to processing element 'j' (destination node).

(6) The set of all W matrices will determine the set of all information for the ANN.

(7) The ANN can be realised by finding an appropriate matrix w.

(8) The weights encode long*term memory (LTM) and the activation states of neurons encode short-term memory (STM) in a neural network.

1.11.2 Bias

(1) The bias is included in the network.

(2) It has impact in calculating the net input.

(3) The input vector becomes, $X = \{1, X_1, X_2, \ldots, X_n\}$

(4) The bias is another weight, say $w = b_1$.

(5) Bias plays a major role in determining the output of the network.

(6) The bias is of two types :

(a) Positive bias increases the net input of the network and

(b) The negative bias decreases the net input of the network.

(7) Due to bias-effect, the output of the network can be varied.

1.11.3 Threshold

(1) Threshold is a set value on which the final output of the network may be calculated.

(2) In activation function, threshold value is used.

(3) To obtain the network output, a comparison is made between the calculated net input and the threshold.

(4) There is a threshold limit for each and every applications.

(5) The activations function using threshold is defined as

$$f(net) = \begin{cases} 1 & \text{if } net \geq \theta \\ -1 & \text{if } net < \theta; \end{cases}$$; Where θ is fixed threshold value.

1.12 LIMITATIONS OF NEURAL NETWORKS

(1) The NN needs training to operate

(2) The architecture of a neural network is different from the architecture of microprocessors.

Hence emulation is necessary –

(3) Requires high processing time for large neural networks.

1.12.1 Genetic Algorithm : (G.A.)

Genetic algorithms are computational procedures modelled on the mechanics of natural genetic systems.

- GAS are executed iteratively on a set of coded solutions, called population, with three basic operators Selections/reproduction, crossover and mutation.
- They use only objective function information for moving to the next iterations. They function in the following four ways :

(1) GAS work with the coding of the parameter set.

(2) GAS work simultaneously with multiple points.

(3) GAS search via sampling using only objective functions.

(4) GAS search using stochastic (probabilistic) operators, and not deterministic rules.

1.12.2 Applications of GAS

If has applications in solving problems in business, scientific and engineering circles

(1) Like synthesis of neural network architecture,

(2) Travelling salesman problem,

(3) Graph colouring

(4) Scheduling

(5) Numerical optimisations,

(6) Pattern recognition,

(7) Image processing.

1.12.3 Hybrid Systems

Hybrid systems can be classified into three different systems as shown in Fig. 1.12.1

1. **Neuro-fuzzy hybrid system**

It is a fuzzy system that uses a learning algorithm derived from network theory. It determines its parameters by processing data samples :

Its advantages are

(i) If can handle any kind of information (numeric, linguistic, logical, etc.)

(ii) It can manage partial, vague, imprecise or imperfect information.

(iii) It can resolve conflicts by collaboration.

(iv) It has self-learning, self-organising and self-tuning capabilities.

(v) It does not need prior knowledge of data.

(vi) It can organise human decision-making process.

(vii) It can make computation fast by using fuzzy number operations.

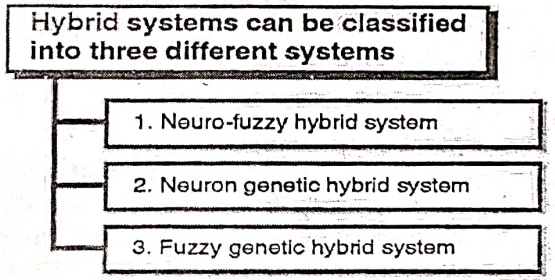


Fig 1.12.1 : Classification of Hybrid systems

2. **Neuro-Genetic Hybrid Systems**

Genetic algorithms (GAS) are increasingly applied in ANN design in various ways :

(i) **Topology optimisation :** GA is used to select a topology [i.e. number of hidden layers, number of hidden nodes, interconnection pattern] for ANN. And in turn using some training scheme, it is trained back propagation.

(ii) Many of the control parameters such as learning rate, momentum rate, tolerance level etc. can be optimised using GAS.

(iii) GAS have been used in many other ways innovative ways, e.g. select good indicators, evolve optimal trading systems etc.

3. **Fuzzy Genetic Hybrid System**

- The optimisation abilities of GAs are used to develop the best set of rules by a fuzzy inference engine. A fuzzy GA is a directed random search. Over all discrete fuzzy subsets of an interval and has features which make it applicable for solving the problem.
- It is capable of creating the classification rules for a fuzzy system where objects are classified by linguistic terms. Coding the rules genetically is more efficient as it is consistent with numeric coding of fuzzy examples. The training data and randomly generated rules are combined to create a better starting point for reproductions.

1.13 BACK-PROPAGATION NETWORK (BPN)

(i) Back-propagation network algorithm is applied to multilayer feed-forward networks consisting of processing elements.

(ii) The networks connected to back-propagation learning algorithm are also called as **back-propagation network (BPN).**

(iii) This algorithm provides a procedure for changing the weight in back-propagation network (BPN) to the given input pattern correctly.

(iv) The basic concept for this weight update is that where the error is propagated back to the hidden unit.

(v) In BPN, weights are calculated during the learning period of the network.

(vi) To update weights, the error must be calculated. There is no direct information of the error at the hidden layer. So we have to develop other techniques to calculate an error at the hidden layer, and this will cause minimisation of the output error.

(vii) Backpropagation is a systematic method of training multilayer artificial neural networks.

It is developed on high mathematical foundation and it has very good application potential.

(viii) Backpropagation learning rule is applicable on any feed forward network architecture.

(ix) Slow rate of convergence and local minima problems are its weaknesses.

(x) The multilayer feedforward (MLFF) network with back propagation (BP) learning is also called as 'multilayer perception' because of its similarity to perceptron networks with more than one layer.

(xi) We carry out BPN as follows :

(a) The feed forward of the input training pattern

(b) The back-propagation of the error and

(c) Updation of weights.

1.13.1 Architecture

A back-propagation neural network (BPN) consist of an input layer, hidden layer and an output layer. The neurons at the hidden and output layers have biases. The bias terms also act as weights. The outputs obtained could be either binary (0, 1) or bipolar (– 1, 1). Refer Fig. 1.13.1.

We Exhibit the Architecture of BPN

The terminologies used in the algorithm are as follows :

x = input training vector($x_1, x_2, ..., x_n$)

t = target output vector

($t_1, t_2, ... t_k, ..., t_m$)

α = learning rate parameter ;

v_{oj} = bias on j^{th} hidden unit,

w_{ok} = bias on k^{th} output unit ;

Z_j = hidden unit j.

x_j = input unit j,

(since identity activation function is used for input layer, the input and output signals are same).

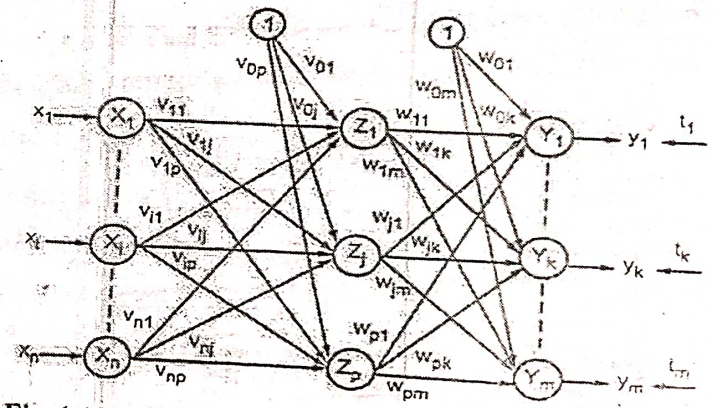


Fig. 1.13.1 : Architecture of a back-propagation network

The net input to Z_j is,

$$Z_{inj} = V_{oj} + \sum_{i=1}^{n} x_i v_{ij}$$

and the output is, $Z_j = f(Z_{inj})$

y_k = output unit k.

The net input to y_k is,

$$y_{ink} = w_{ok} + \sum_{j=1}^{p} Z_j w_{jk} \quad \text{and} \quad \text{the output is} \quad y_k = f(y_{ink})$$

δ_k = error correction weight adjustment for w_{jk}, which is back-propagated to the hidden units that feed into unit y_k. and

δ_j = error connection weight adjustment for v_{ij} and to the hidden unit Z_j.

1.13.2 Algorithm (Training)

We mention the error-back-propagation learning algorithm :

- **Step (I) :** Small random values for weights and learning rate,
- **Step (II)** : Carry on the steps 3-10 when stopping condition fails.
- **Step (III)** : Carry on steps 3-9 for each training pair.

Phase (I) : Feed-forward phase

- **Step (IV) :** Each input receives input signal xi and sends to hidden unit for i = 1 to n.
- **Step (V) :** Calculate net input

$$Z_{inj} = V_{oj} + \sum_{i=1}^{n} x_i v_{ij}$$

(To calculate output, we apply activation function Z_{inj})(binary or bipolar sigmoidal activation function) : $Z_j = f(Z_{inj})$ and send the output signal to the input of output layer units.

- **Step (VI) :** For each output y_k (k = 1 to m), calculate the net input :

$$y_{ink} = w_{ok} + \sum_{j=1}^{p} Z_j w_{jk}$$

and we apply activation function to evaluate output signal : $y_k = f(y_{ink})$

Phase II + (Back-Propagation of Error)

- **Step (VII) :** We compute error correction term for each output unit y_k (K = 1 to m)

$$\delta_k = (t_k - y_k) f'(y_{ink})$$

Where $f'(y_{ink})$ is derivative of $f(y_{ink})$.

Now, on the basis of the calculated error correction term, update the change in weights and bias :

$$\Delta w_{jk} = \alpha \delta_k z_j \; ; \quad \Delta w_{ok} = \alpha \delta_k$$

The error δ_k is to be sent to the **hidden layer backwards.**

- **Step (VIII) :** For each hidden unit (Z_j, J = 1 to p) ;

$$\delta_{inj} = \sum_{k=1}^{m} \delta_k w_{jk}$$

and to calculate the error term : $\delta_j = \delta_{inj} \cdot f'(Z_{inj})$

Now, we update the change in weights and bias ;

$$\Delta V_{ij} = \alpha \delta_j x_i \; ; \quad \Delta V_{oj} = \alpha \delta_j$$

Phase (III) : Weight and Bias Updation

- **Step (IX) :** Each output unit (y_k, k = 1 to m) updates the bias and weights :

$$w_{jk}(\text{new}) = w_{jk}(\text{old}) + \Delta w_{jk} \; ;$$

$$w_{ok}(\text{new}) = w_{ok}(\text{old}) + \Delta w_{ok}$$

Each hidden unit (Z_j, j = 1 to p) update its bias and weights :

$$V_{ij}(\text{new}) = V_{ij}(\text{old}) + \Delta V_{ij} \text{ and } v_{oj}(\text{new})$$

$$= V_{oj}(\text{old}) + \Delta V_{oj}$$

- **Step (X) :** Check for stopping condition.

1.13.3 Flow-chart for Back-Propagation Network Training

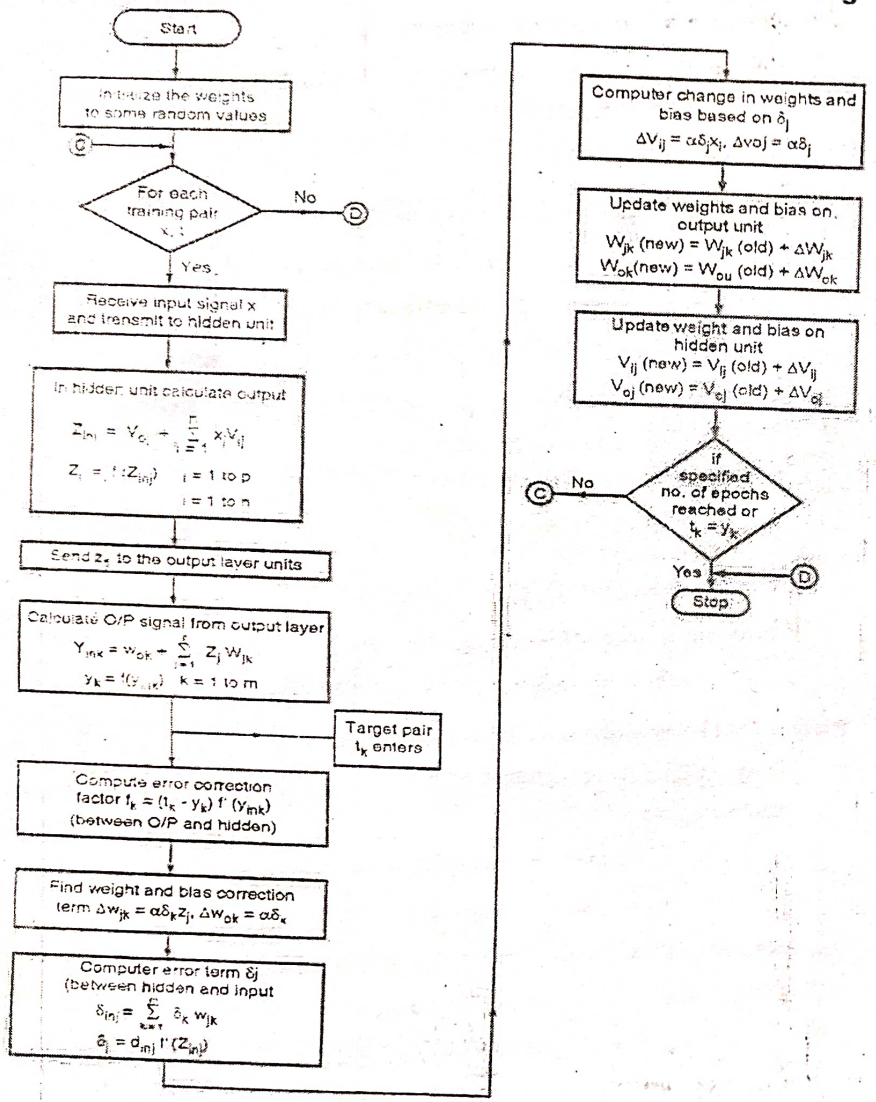


Fig. 1.13.2 : Flow-chart for Back-Propagation Network Training

1.14 LEARNING FACTORS OF BACK PROPAGATION NETWORK

(i) Learning rate α determines the size of the weight adjustments made at each iteration and hence influences the rate of convergence.

(ii) Poor choice of the rate can result in a failure in convergence.

It is convenient to keep the rate constant through all the iterations for best results. If the learning rate α is too large, the search path will oscillate and converges more slowly than a direct descent. The range of α from 10^{-3} to 0.9 is optimistic and has been used successfully for several back-propagation algorithmic experiments. If the rate is too small, the descent will progress in small steps, increasing the time to converge. Jacobs has suggested the use of adaptive coefficient where the value of the learning rate is the function of error derivative on successive updates.

1.15 CALCULATION OF ERROR

Consider any r^{th} output neuron. Let 'O' be the output for which the target output is 'T'. The **error norm** in output for the r^{th} output neuron is given by

$$E_r^1 = \frac{1}{2} e_r^2 = \frac{1}{2} (T - 0)^2 \quad ...(i)$$

Where 'e_r' is the error in the r^{th} neuron. Error may be positive or negative. To consider only positive values, we consider the square of the error, i.e., we consider only absolute valve.

For the **first training pattern,** the Euclidean norm of error E^1 is given by

$$E^I = \frac{1}{2} \sum_{r=1}^{n} (T_{or} - D_{or})^2 \quad \text{...(ii)}$$

If we consider error function for all the training patterns, we get

$$E(V, W) = \sum_{j=1}^{n \text{ set}} E^j (V, W, I), \text{ where}$$

E is the error function depending on the m (1 + n) weights of W and V. Based on error signal, neural network modifies its synaptic connections to improve the system performance.

1.16 METHOD OF STEEPEST DESCENT

The error surface is given by, $E = \sum_{j=1}^{n \text{ set}} E^p (V, W, I)$ and is shown in the Fig. 1.16.1

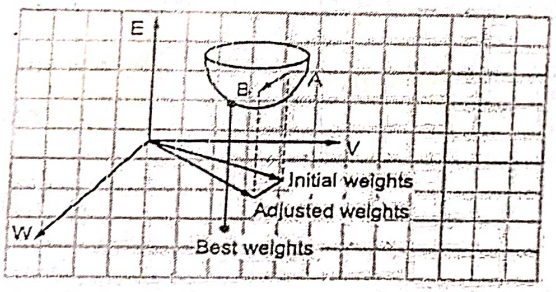


Fig. 1.16.1 : Euclidean norm of errors

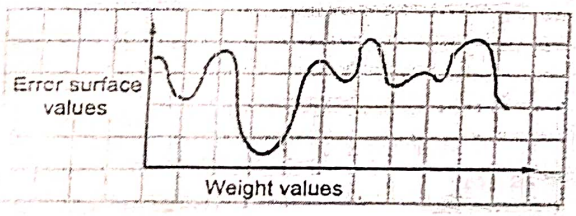


Fig. 1.16.2 : Typical error surface with non-linear activation function

(i) Multilayer feed forward networks with non-linear activation functions have mean squared error (MSE) surface, which is not in general, a smooth parabolic surface. In general, the error surface is complex and consists of many local and global minima,

(ii) In a single-layer linear activation case, the surface is smooth parabolic, as shown in Fig. 1.16.2.

1.16.1 Some Applications

Variety of problems have been successfully solved by Neural Networks. Pattern recognition and image processing Neural Networks are used in the recognition of visual images, handwritten characters, printed characters etc.

1. **Optimisation-constraint satisfaction :** Such type of problems have to satisfy constraints and to obtain optimal solutions. For examples, manufacturing scheduling, finding shortest possible tour given a set of countries, etc.

   Forecasting Neural Networks can predict situations from past trends and can make risk assessment. Therefore in areas such as meteorology, stock market, banking they have ample applications.

2. **Control Systems**

   Neural Networks have ample applications in control systems. Computer products, chemical plants, robots are standing examples of neural networks.

1.17 GRADIENT DESCENT

- Gradient Descent is an optimization algorithm and it is used to train **machine learning models** and **neural networks.**

- Training the data helps these models learn over time and the cost function within gradient descent gauges its accuracy with each iteration of parameter updates.
- The model will continue to adjust its parameter so that possible error will be minimum. The happens when function become, near to zero.
- Once machine learning models are optimized for accuracy, they become powerful tools for computer science application.

1.17.1 Working of Gradient Descent

GQ. How gradient descent works ?

- Let us revise the concept of linear regression.
- The equation $y = mx + c$, represents a line with 'm ' as slope and 'c' as intercept on Y-axis.
- We can plot a scatter – diagram and find the line of best fit. It can calculate the error between the actual output and the predicted output, using the mean - squared formula.
- The gradient descent algorithm behaves similarly, but it is based on a convex function, such as :

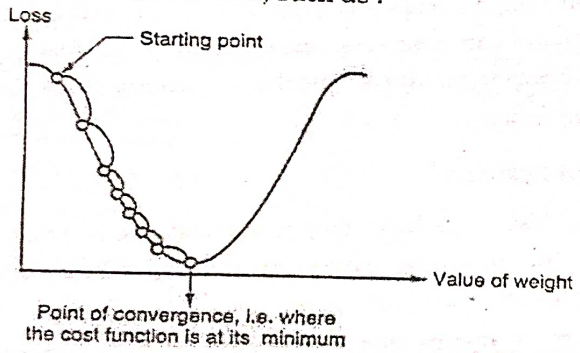
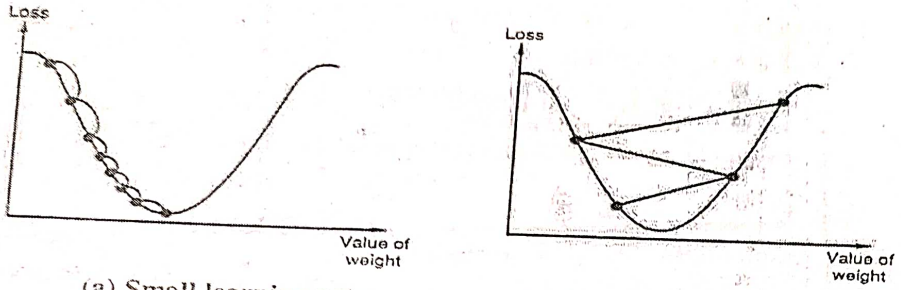


Fig.1.17.1 : Gradient descent algorithm

Point of convergence, i.e. where the cost function is at its minimum

- The starting point is chosen arbitrarily to evaluate the performance.
- From that starting point, we calculate derivative (or slope) and draw tangent line and note the steepness of the slope.
- The slope will yield the updates of the parameters i.e. the weight and bias.
- The slope at the starting point will be steeper, but as new parameters are generated, the steepness reduces till it reaches, the lowest point on the curve, known as the point of convergence.
- The aim of gradient descent is to minimize the cost function or the error between predicated and actual value.
- To do this, we require two data points : a direction and a learning rate.
- Using these factors, we can calculate partial derivatives of future iterations, and thereby we get local or global minimum i.e. point of convergence.
- **Learning rate :** This is also called as step size or alpha.
- This is the size of the steps to reach to the minimum. This is typically a small value and it is evaluate and updated, depending upon the behaviour of the cost function.
- High learning rates result in larger steps but the risk involved is minimum.
- Conversely, a low learning rate has small step sizes.
- The advantage of this method is gives more precision. Since the number of iterations are more, it takes more time and computations to reach the minimum.



(a) Small learning rate (b) Large learning rate

Fig. 1.17.2

1.17.2 The Cost (or loss) Function

- This function measures the difference, or error between actual y and predicted y at its current position.
- This provides feedback to the model so that it can adjust parameters to minimize the error and can find local or global minimum. This way model's efficacy is improved.
- It iterates continuously i.e. it moves along the direction of steepest descent, until the cost function is minimum.
- At this point, the model stops learning. Here, we note that there is a slight difference between cost function and cost function and loss function.
- A loss function refers to the error of one training example, while a cost function calculates the average error across an entire training set.

1.17.3 Types of Gradient Descent

There are three types of gradient descent learning algorithms :

(i) batch-gradient descent

(ii) stochastic-gradient descent and

(iii) mini-batch gradient descent

1.17.3(A) Batch Gradient Descent

- Batch gradient descent sums the error at each point in a training set, updating the model only after all training examples have been evaluated.
- This process is referred to as a training epoch.
- While this batching provides computation efficiency, it can have long processing time for large training datasets. It needs actually to store all of the data into memory.
- Batch descent method produces a stable error gradient and convergence, but the convergence point is not most ideal

1.17.3(B) Stochastic Gradient Descent

- Stochastic gradient descent (SGD) runs a training epoch for each example in the dataset.
- It updates each training examples parameter one at a time.
- Since it is only one training example, they are easier to keep in memory.
- These frequent updates can offer more detail and also speed, it results in losses in computational efficiency when compared to batch gradient descent.
- Its frequent updates create noisy gradients, but this helps to find global minimum.

1.17.3(C) Mini-batch Gradient Descent

- Mini-batch gradient descent combines concepts from both batch gradient and stochastic gradient descent.
- It splits the training dataset into small batch sizes and performs updates on each of those batches.
- This approach makes a baloma between the computational efficiency of batch gradient descent and the speed of stochastic gradient descent.

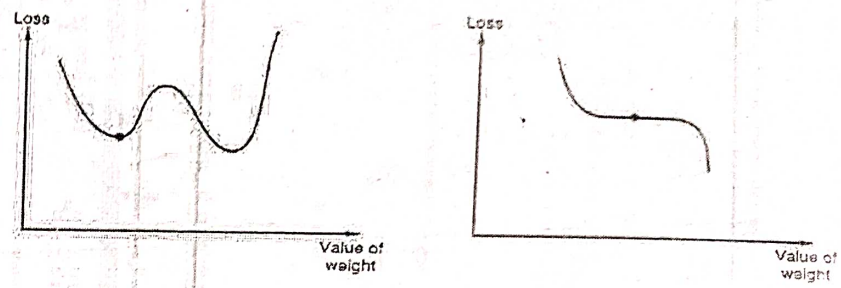
1.17.4 Challenges with Gradient Descent

Gradient descent is the most common approach for optimization problems, but it has its own set of challenges.

For example :

Local minima and saddle points :

- For, convex problem, local minimum can be found easily. But for non-convex problems, gradient descent has to struggle to find global minimum.
- Note that when the slope of the cost function is at zero (or close to zero), the model stops learning.
- At local minima and saddle points the slope of the cost function is zero. But the slope of cost function, increases on either side of the current point.
- Noisy gradients can help the gradient escape local minimum and saddle points.



(a) (b)

Fig. : 1.17.3 : value of weight

1.17.5 Vanishing and Exploding Gradients

- We can also come across two other problems when the model is trained with gradient descent and back propagation :

(i) Vanishing gradient :

This occurs when the gradient is too small. As we go backwards during backpropagation, the gradient continues to become smaller, causing the earlier layers in the network to learn more slowly than later layers.

When this happens, the weight parameters update until they become insignificant - i.e. (almost) O. This results in an algorithm which stops learning.

(ii) Exploding gradients

This happens when the gradient is too large. This creates an unstable model.

In this case, the model weights will grow too large.

One solution to this issue is to leverage a dimensionality reduction technique, and this helps to minimize the complexity within the model.

1.17.6 Method of Resolving the Vanishing Oradlert Problem

There are various methods that help in overcoming the vanishing gradient problems :

(1) Multi-level hierarchy

(2) The long-short term memory

(3) Residual neural network

(4) ReLU

1.17.6(A) Multi-level Hierarchy

It is a simple method, it trains one level at a time and fine - tunes the level by backpropagatio. So that every layer learns a compressed observation which goes ahead for the next level.

1.17.6(B) Long-Short Term Memory (LSTM)

- A simple LSTM helps the gradient size to remain constant.
- The activation function we use in the LSTM often works as an identify function.
- Hence in gradient back-propagation the size of the gradient does not vanish. The image is as :

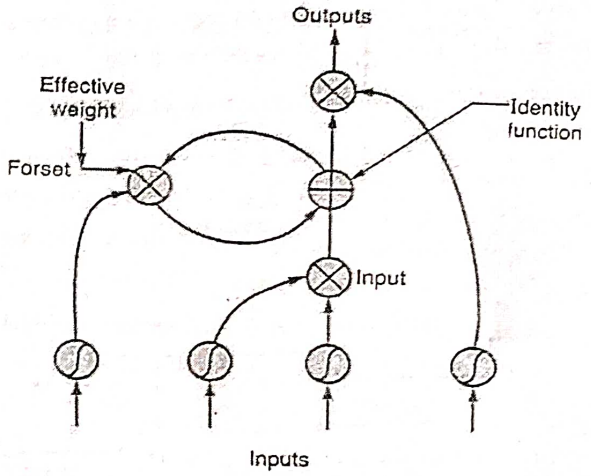


Fig. 1.17.4 : Image of gradient back-propagation

- From the above Fig. 1.17.4 the effective weight of the gradient is equal to the forget gate activation.
- Hence, if the **forget gate** is on (i.e.) activation close to 1.01, then the gradient does not vanish.
- Therefore, LSTM is one of the best options to deal with long-range dependences. Especially, in the recurrent neural network.

1.17.7 Residual Neural Network

- Residual connections in the neural network make the model learn well and the batch normalization feature makes sure that gradients will not vanish.
- These batch normalization features are obtained by the skip connection.

- The skip or bypass connection is useful in any network to bypass data from a few layers. Using these connection, information can be transferred from layer in to layer n + t.
- We connect the activation function of layer n to the activation function of n + t.
- This causes the gradient to pass between the layers without any modification in size. We exhibit the image :

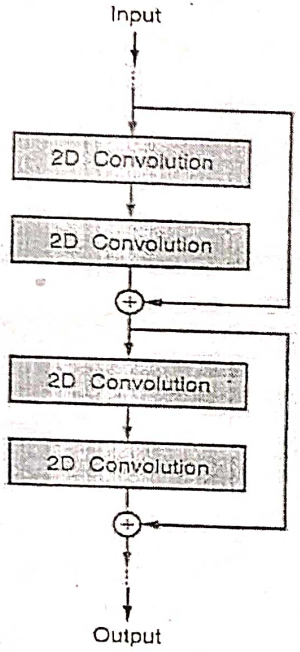


Fig. 1.17.5 : Image of Activation function

1.17.8 Rectified Linear Unit (ReLU) Activation Function :

- ReLU is an activation function, like a sigmoid and tanh activation function but better than them.
- The basic functions for ReLU is given by,

$$f(x) = \max(0, x)$$

- Here gradient is one when the output of the function is > 0.

1.18 ACTIVATION FUNCTION

RQ. Explain common activation functions used in neural network. **(Ref. Q.1(b) May 2019, 5 Marks)**

(1) A model of the behaviour of a neuron can be presented as shown in Fig. 1.18.1.

Here $x_1, x_2, \ldots, x_n$ are the n-inputs to the artificial neuron. $w_1, w_2, \ldots, w_n$ are the weights attached to the input links.

(2) **Biological** neuron receives all inputs through the dendrites, sums them and produces an output. If the sum is greater than a threshold value. The input signals are passed on to the cell body through the synapse which may accelerate or retard an arriving signal.

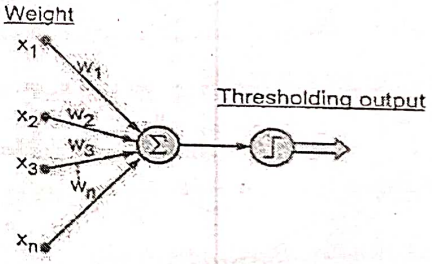


Fig. 1.18.1 : Simple model of an artificial neuron

(3) Here weights model the acceleration or retardation of the input signals. In short, weights are multiplicative factors of the inputs.

(4) The total input (say I) received by the soma (body) of the artificial neuron is

$$I = w_1 x_1 + w_2 x_2 + \ldots + w_n x_n = \sum_{i=1}^{n} w_i x_i \quad \ldots(i)$$

(5) This sum is passed on to a non-linear filter ϕ called Activation function, or Transfer function, or Squash Function which releases the outputs.

$$\text{i.e. } y = f(I) \quad \ldots(ii)$$

1.18.1 Activation Functions

RQ. What are the important properties of activation function used in neural networks ? **(Ref - Q. 1(c), May 2016, 5 Marks)**

RQ. List the different activation functions used in neural network. **(Ref - Q. 1(d), May 2017, 5 Marks)**

RQ. With mathematical list four different activation functions used in neurons. **(Ref - Q. 1(e), May 2019, 5 Marks)**

The obtain exact output, the activation function is applied over the net input of an ANN.

There are several activation functions. Here we consider a few

1. **Linear function :** It is defined as

$$f(x) = x \quad \text{for all } x$$

Here input = output

2. **Bipolar Step function :** The function is defined as :

$$f(x) = \begin{cases} 1, & \text{if } x \geq \theta \\ -1, & \text{if } x < \theta \end{cases}$$

Where θ is threshold value. The function is also used in single-layer nets to convert the net input to an bipolar output $(+1, -1)$.

3. **Binary step function :** The function is defined as :

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

This function is used in single-layer nets to convert the net input to an output that is binary (0 or 1).

4. **Sigmoidal function :** This function is used in back-propagation nets. They are of two types :

(i) Binary sigmoidal function : It is also called as unipolar sigmoid function or a logistic sigmoid function, and defined as

$f(x) = \frac{1}{1 + e^{-\lambda x}}$, where λ is steepness parameter

The derivative of f(x) is :

$$f'(x) = \frac{-1}{[1 + e^{-\lambda x}]^2} \cdot (-\lambda e^{-\lambda x}) = \frac{\lambda (e^{-\lambda x})}{[1 + e^{-\lambda x}]^2}$$

$$= \frac{\lambda [1 + e^{-\lambda x} - 1]}{[1 + e^{-\lambda x}]^2}$$

$$= \lambda \left\{ \frac{1}{(1 + e^{-\lambda x})} - \frac{1}{(1 + e^{-\lambda x})^2} \right\}$$

$$= \lambda \left[\frac{1}{(1 + e^{-\lambda x})} \left\{ 1 - \frac{1}{(1 + e^{-\lambda x})} \right\} \right]$$

$$= \lambda [f(x) (1 - f(x))] = \lambda f(x) [1 - f(x)]$$

Range of this sigmoid function is 0 to 1 [∵ denominator is always greater than or equal to 1]

(ii) Bipolar sigmoid function :

The function is given by

$$f(x) = \frac{2}{1 + e^{-\lambda x}} - 1 = \frac{2 - 1 - e^{-\lambda x}}{1 + e^{-\lambda x}} = \frac{1 - e^{-\lambda x}}{1 + e^{-\lambda x}}$$

where again, λ is steepress parameter and the range is between -1 and $+1$.

Derivative of $f(x)$ is :

$$f'(x) = \frac{[1+e^{-\lambda x}][\lambda e^{-\lambda x}] - (1-e^{-\lambda x})(-\lambda e^{-\lambda x})}{(1+e^{-\lambda x})^2}$$

$$= \frac{\lambda e^{-\lambda x}[1+e^{-\lambda x}+1-e^{-\lambda x}]}{(1+e^{-\lambda x})^2} = \frac{2\lambda e^{-\lambda x}}{(1+e^{-\lambda x})^2}$$

$$= \frac{\lambda}{2}\left[\frac{4e^{-\lambda x}}{(1+e^{-\lambda x})^2}\right]$$

$$= \frac{\lambda}{2}\left[\frac{(1+e^{-\lambda x})^2 - (1-e^{-\lambda x})^2}{(1+e^{-\lambda x})^2}\right]$$

$$= \frac{\lambda}{2}\left[1 - \left(\frac{1-e^{-\lambda x}}{1+e^{-\lambda x}}\right)^2\right]$$

$$= \frac{\lambda}{2}[1 - f(x)^2]$$

$$= \frac{\lambda}{2}[(1+f(x))(1-f(x))]$$

Remark $\quad [\because\ a^2 - b^2 = (a+b)(a-b)]$

(1) Here, $f(x) = \frac{1-e^{-\lambda x}}{1+e^{-\lambda x}} = \tanh\left(\frac{\lambda x}{2}\right)$

$$\therefore\ f'(x) = \frac{\lambda}{2}\operatorname{sech}^2\left(\frac{x}{2}\right)$$

$$= \frac{\lambda}{2}\left[1 - \tanh^2\left(\frac{x}{2}\right)\right]$$

$$= \frac{\lambda}{2}\left[\left(1+\tanh\frac{x}{2}\right)\left(1-\tanh\frac{x}{2}\right)\right]$$

$$f'(x) = \frac{\lambda}{2}[(1+f(x))(1-f(x))]$$

(2) Sigmoid functions are so-called because their graphs are "S-shaped".

(i) Simple sigmoids defined to be odd, are monotone functions of one variable,

(ii) Hyperbolic sigmoids are subset of simple sigmoids and natural generalisation of the hyperbolic tangent function.

(3) Ramp function

It is defined as $f(x) = \begin{cases} 1, & \text{if } x > 1 \\ x, & 0 \le x \le 1 \\ 0, & x < 0 \end{cases}$

☞ **Graphs of activation functions**

(1) Linear function

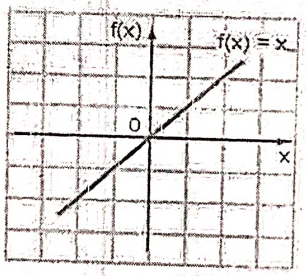


Fig. 1.18.2

(2)

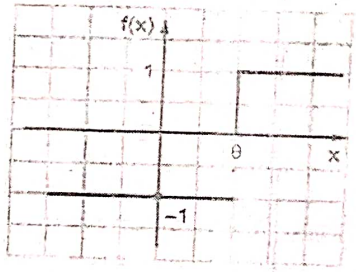


Fig. 1.18.3

(3) Binary-step function

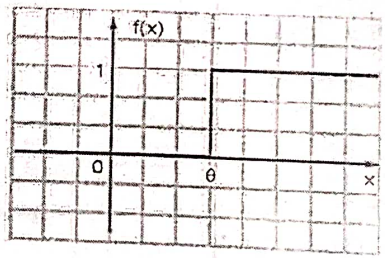


Fig. 1.18.4

(4)

(i) Binary sigmoidal function

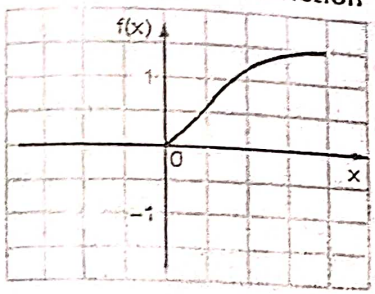


Fig. 1.18.5(i)

(ii) Bipolar sigmoidal function

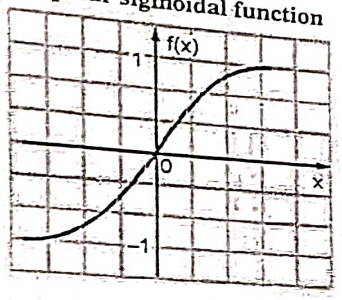


Fig. 1.18.5(ii)

(5) Ramp function

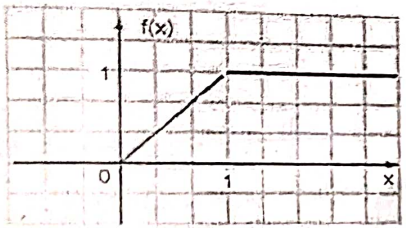


Fig. 1.18.6

1.18.2 Illustrative Examples for Logistic Function

Ex. 1.18.1 : Logistic function is given by, $F(v) = \frac{1}{1 + \exp(-av)}$

Show that the derivative of f(v) w.r.t. v is given by, $\frac{df}{dv} = af(v)\,[1 - f(v)]$

What is the value of this derivative at the origin ?

☑ **Soln. :**

Step (I) : We have, $f(v) = \frac{1}{1 + \exp(-av)}$

Different w.r.t. v ; we get

$$f(v) = \frac{-1}{[1 + \exp(-av)]^2}\,[\exp(-av)(-a)]$$

$$= \frac{a \exp(-av)}{[1 + \exp(-av)]^2}$$

$$= \frac{a}{[1 + \exp(-av)]}\,\frac{\exp(-av)}{[1 + \exp(-av)]}$$

$$= a\,f(v)\left[\frac{\exp(-av)}{[1 + \exp(-av)]}\right]$$

$$= a\,f(v)\left[1 - 1 - \frac{\exp(-av)}{1 + \exp(-av)}\right]$$

$$= a\,f(v)\left[1 - \left\{1 + \frac{\exp(-av)}{1 + \exp(-av)}\right\}\right]$$

$$= a\,f(v)\left[1 - \left\{\frac{1 + \exp(-av) - \exp(-av)}{1 + \exp(-av)}\right\}\right]$$

$$= a\,f(v)\left[1 - \left\{\frac{1}{1 + \exp(-av)}\right\}\right]$$

$$\therefore f'(v) = a\,f(v)\,[1 - f(v)]$$

Step (II) : As $V \to 0$; $\exp(-av) = \exp(0) = 1$

$$\therefore f(v) = \frac{1}{1+1} = \frac{1}{2}$$

$\therefore$ f'(v) at v = 0 is,

$$= a \cdot \frac{1}{2}\left[1 - \frac{1}{2}\right] = \frac{9}{4}$$

Ex. 1.18.2 : An odd sigmoid function is given by

$f(v) = \frac{1 - \exp(-av)}{1 + \exp(-av)} = \tan h\left(\frac{av}{2}\right)$.

Show that the derivative of f(v) w.r.t. V is given by,

$\frac{df}{dv} = \frac{a}{2}[1 - f^2(v)]$.

What is the valve of this derivative at the origin ? Suppose that the slope parameter a is made infinitely large. What is the resulting form of f(v).

☑ Soln. :

Step (I) : For convenience we take, $f(v) = \tan h\left(\frac{av}{2}\right)$

Differentiating w.r.t. V;

$$\frac{df}{dv} = \operatorname{sech}^2\left(\frac{av}{2}\right) \cdot \frac{a}{2} = \frac{a}{2}\left[1 - \tan h^2\left(\frac{av}{2}\right)\right]$$

$$[\because \operatorname{sech}^2 x = 1 - \tanh^2 x]$$

$$= \frac{a}{2}[1 - f^2(v)]$$

$$\therefore \frac{df}{dv} = \frac{a}{2}[1 - f^2(v)] \quad \ldots(i)$$

Step (II) : We have $f(v) = \tanh\left(\frac{av}{2}\right)$

At $v = 0$, $f(0) = \tan h(0) = 0$. ∴ At origin, from Equation (i),

$\frac{df}{dv} = \frac{a}{2}[1 - 0] = \frac{a}{2}$

As $a \to \infty$, $\tan h(\infty) = 1$. ∴ As $a \to \infty$, $f(v) \to 1$.

Ex. 1.18.3 : The algebraic sigmoid function is given by,

$$f(v) = \frac{v}{\sqrt{1 + v^2}}$$

Show that the derivative of f(v) w.r.t. v is given by, $\frac{df}{dv} = \frac{\phi^3(v)}{v^3}$

What is the value of this derivative at origin ?

☑ Soln. :

Step (I) :

Differentiating $f(v) = \frac{v}{\sqrt{1 + v^2}}$; we get

$$f'(v) = \left[\frac{\sqrt{1 + v^2} \cdot 1 - v \cdot \frac{1 \cdot 2v}{2\sqrt{1 + v^2}}}{(1 + v^2)}\right]$$

$$= \left[\frac{\sqrt{1 + v^2} - \frac{v^2}{\sqrt{1 + v^2}}}{(1 + v^2)}\right]$$

$$= \left[\frac{(1 + v^2) - v^2}{(1 + v^2)^{3/2}}\right]$$

$$\therefore f'(v) = \frac{1}{(1 + v^2)^{3/2}} \quad \ldots(i)$$

$$\therefore f'(v) = \frac{1}{v^3}\left[\frac{v^3}{(1 + v^2)^{3/2}}\right] = \frac{1}{v^3}\left[\frac{v}{(1 + v^2)}\right]^3$$

$$= \frac{1}{v^3}[f(v)]^3 = \frac{f^3(v)}{v^3} \quad \ldots(ii)$$

Step (II) : As $v \to 0$ $f'(v) = \frac{1}{(1 + v^2)^{3/2}} \to 1$.

∴ At origin, $f'(v) = 1$.

1.19 RELU

ReLU is rectified linear unit ReLu is a non-linear activation function, used in multi – layer deep neural networks. This function is represented as :

$$f(x) = \max(0, x),$$

where x = an input value

An output is equal to zero when the input valve is negative and is equal to input value when input value is positive,

$$\text{i.e. } f(x) = \begin{cases} 0, & \text{if } x \le 0 \\ x, & \text{if } x > 0 \end{cases}$$

where x is an input value.

1.20 OPTIMISATION ALGORITHM

- The concept of loss in deep learning tells us how poorly the model is performing at that current instant.
- We want to minimize loss, because lower loss implies that the model is performing well.
- The process of minimizing (or maximizing) any mathematical expression is called **optimisation.**
- Optimizers are algorithms (or methods) used to change the attributes of the neural network such as weights and learning rate to reduce the losses. Optimizers are used to solve optimization problems by minimizing the function.

1.20.1 Working of Optimizers

GQ. How Optimizers Work ?

Different types of optimizers work to minimize loss function are :

1. Gradient Descent
2. Stochastic Gradient Descent (SGD)
3. Mini Batch Gradient Descent (MB - SGD)

1.20.1(A) Gradient Descent

- Gradient descent is an optimization algorithm, based on a convex function and changes its parameters iteratively to minimize a given function to its local minimum.

1.20.1(B) Stochastic Gradient Descent (SGD)

- Gradient descent requires a lot of memory to load the entire dataset of n-points at a time to compute the derivative of loss function.
- But SGD algorithm computes derivative taking one point at a time.
- SGD performs a parameter update for each training example x(i) and y(i).

$$\theta = \theta - \alpha \frac{\partial}{\partial \theta} [\, J(\theta\,;\, x(i), y(i))\,]$$

where { x(i), y(i) } are the training examples.

- To make the training even faster, we take a Gradient Descent step for each training example.
- SGD is quite noisy and may not converge to a minimum but it is much faster.

1.20.1(C) Mini Batch Gradient Descent (MB - SGD)

- Now, to get the best out of both algorithms, we used Mini – Batch Gradient Descent (MGD). Mini – batch Gradient Descent is relatively more stable than stochastic Gradient Descent (SGD).
- MB – SGD algorithm is an extension of the SGD – algorithm. It overcomes the problem of large time complexity in the case of SGD algorithm.
- MB-SGD algorithm taken a batch of points or subset of points from the dataset to compute derivative.
- But the derivative of the loss – function for MB – SGD is almost the same as a derivative of the loss function for GD after some number of iterations. But the number of iterations

to get minima is large for MB-SGD compared to GD and the cost of Computation is also high.

- The updates in the case of MB – SGD are much noisy because derivative is not always towards minima.
- MB - SGD divides the dataset into various batches and after every batch the parameters are updated :

$$\theta = \theta - \alpha \frac{\partial}{\partial \theta} [J(\theta ; B(i))]$$

where { B(i) } are the batches training examples.

☞ **Advantages**

Less time complexity to converge compared to standard SGD algorithm.

☞ **Disadvantages**

(i) The update of MB-SGD is much noisy compared to the update of the GD algorithm.

(ii) Take a longer time to converge than the GD algorithm.

(iii) It may stuck at local minima.

1.20.1(D) SGD with Momentum

- A major disadvantage of the MB-SGD algorithm is that updates of weight are very noisy. SGD with momentum overcomes this disadvantage. It denoises the gradients.
- Updates of weight depend on noisy derivative and if we denoise the derivatives, then convergence time decreases.
- To denoise derivative, more weightage is given to recent updates compared to previous updates.
- It accelerates the conversence towards the relevant direction and reduces the fluctuation to the irrelevant direction.

- One more hyper parameter is used and is known as momentum, symbolised by Y.

$$\therefore V(t) = Y \cdot V(t-1) + a \frac{\partial}{\partial \theta} [J(\theta)]$$

Now, the weights are updated by

$$\theta = \theta - V(t)$$

- The momentum term θ Y is usually set to 0.9 or a similar value.
- Momentum at time 't' is computed using all previous updates giving more weightage to recent updates compared to previous update. This speeds up convergence.
- The momentum term increase for those dimensions whose gradients points in the same directions and reduces updates for dimensions whose gradient change directions.
- This way faster convergence and reduced oscillations occur.

☞ **Advantages**

(1) It has all advantages of the SGD algorithm.

(2) Converges faster than the GD algorithm.

☞ **Disadvantages**

We need to compute one more variable for each update.

1.20.1(E) Nesterov Accelerated Gradient (NAG)

- The idea of the NAG algorithm is similar to SGD with momentum.
- If the momentum is too high, the algorithm may miss the local minima and may continue to rise up. Here NAG algorithm comes to our help.

- We have use $y \cdot V(t-1)$ to modify weights, hence $\theta - yV(t-1)$ can predict the future location. Now, we can calculate the cost based on this future parameter rather than the current one.

$$V(t) = y \cdot V(t-1) + \alpha \frac{\partial}{\partial \theta} [J(\theta - yV(t-1))]$$

and then update the parameters using

$$\theta = \theta - V(t)$$

- NAG first makes a big jump in the direction of the previously accumulated gradient, then measures the gradient and then makes a correction and that results in the complete NAG update.
- This update prevents us from going too fast and results in increased responsiveness.
- This increases the performance of RNNS on a number of tasks.

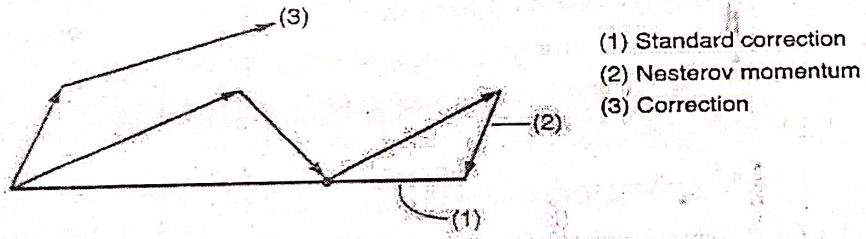


Fig. 1.20.1 : RNN performance

1.20.1(F) Adaptive Gradient Descent (AdaGrad)

- In all the algorithm we have seen, the learning rate remains constant.
- The key idea of AdaGrad is to have an adaptive learning rate for each of the weights.
- *It performs smaller updates for parameters associated with frequently occurring features, and larger updates for parameters associated with infrequently occurring features.*

- For convenience, we use the following notations :
  - (i) q_t is gradient at time step t,
  - (ii) q_t, i is the partial derivative of the objective function w.r.t. the parameter θ_i at time step t, η is the learning rate and ∇_θ is the partial derivative of the loss function $J(\theta_i)$

$$q_t, i = \nabla_\theta \cdot J(\theta_t, i)$$

The SGD update for every parameter θ_i at each time step t becomes.

$$\theta_{t+1, i} = \theta_{t, i} - \eta \cdot q_{t, i}$$

- In update rule, AdaGrad modifies the general learning rate η at each time step t for every parameter θ_i based on the past gradients for θ_i :

$$\theta_{t+1, i} = \theta_{t_1} i - \frac{\eta}{G_t \sqrt{i\, i + \epsilon}} \cdot q_t, i$$

- Where G_t is the sum of the squares of the past gradients w.r.t. all parameters θ.
- The AdaGrad eliminates the need to manually tune the learning rate.
- The main weakness is the accumulation of the squared gradients (G_t) in the denominator. Since every added term is positive, the accumulated sum keeps growing during training.
- *It causes learning rate to shrink and it becomes very small and* that causes gradient problem to vanish.

☞ **Advantage**

No need to update the learning rate manually as it changes adaptively with iterations.

☞ **Disadvantage**

As the number of iteration becomes very large, the learning rate decreases and that causes slow convergence.

1.21 HYPER PARAMETERS

- Hyper parameters are the variables which determine the network structure (e.g. number of hidden units) and the variables which determine how the network is trained (e.g. : learning rate)
- Hyper parameters are set before training (i.e. before optimizing the weights and bias).
- The prefix 'hyper' – suggests that they are 'top level' parameters that control the learning process and the model parameters that result from it.
- Hyper parameters should not be confused with parameters. In machine learning, the label parameter is used to identify variables whose values are learned during training.
- Hyperparameters control the learning process.
- Every variable that an **AI engineer** or **ML engineer** chooses before model training begins can be referred to as a hyper parameter – so far as the value of the variable remains the same when training ends.
- It is necessary to choose the right hyper parameters before training begins because this type of variable has a direct impact on the performance of the resulting machine learning model.

1.21.1 Layer Size

- Two main hyper parameters control the architecture or topology of the network ; the number of layers and the number of nodes in each hidden layer.
- Here we shall study the roles of layers and nodes.

We divide this section into four parts :

1. The multilayer perceptron
2. How to count layers
3. Why have multiple layers
4. How many layers and Nodes to use

1.21.1(A) The Multilayer Perception

- A node, also called as neuron or perceptron is a computational unit that has one or more weighted input connections and a transfer function that combines the inputs and an output connection.
- Nodes are organized into **layers** to comprise a network.
- A single layer networks have just one layer of active units. Inputs connect directly to the output through a single layer of weights.
- A single layer network can be extended to a multiple – layer – network.
- It has an input layer that connects to the input variables, one or more hidden layers, and an output layer that produces the output variables.

1.21.1(B) Counting of Layers

GQ. How to Count Layers.

- There is an argument that input layer is not to be counted. It is because inputs are not active, they are just the input variables.
- Hence, an MLP that has an input layer, one hidden layer and one output layer is a 2 - layer MLP.
- The number of nodes in each layer is specified as an integer, in order from the input layer to the output layer.
- For example, a network with two variables in the input layer, one hidden layer with eight nodes, and an output node with one node will be described using the notation : 2/8/1.

1.21.1(C) Need of Multiple Layers

GQ. Why have Multiple Layers ?

- A single layer neural network can represent only linearly separable functions. This means that very simple problems where, say, two classes in a classification problem can be separated by a line.
- Here a single layer network would be sufficient. But the majority of the problems are not linearly separable.
- Here, a multilayer perceptron can be used to represent convex regions. It can learn to draw shapes in some high - dimensional space that can separate and classify them, and it overcomes the limitation of linear separability.

1.21.1(D) Layers and Nodes to use

GQ. How many layers and nodes to use.

We enumerate different approaches to solve the problem :

(i) **Experimentation :** It is analytically difficult to calculate the number of layers or the number of nodes per layers. The number of layers and the number of nodes in each layer are model hyper – parameters that are can specify. One can discover the answer using a robust test harness and controlled experiments.

(ii) **Intuition :** The network can be decided via intuition. Given an understanding of the problem domain, we may believe that a deep hierarchical model is required to solve prediction problem. In which case, we may choose a network that has many layers of depth.

(iii) **Co for Depth :** Greater depth does seem to result in better generalization for a wide variety of tasks. This indicates that using deep architectures does indeed express a useful space of functions the model learns.

(iv) **Borrow Ideas :** A simple but time consuming approach is to use findings reported in the literature. Transferability of model hyper parameters that result in skillful models from one problem to another is a challenging open problem. The network layers and number of nodes used on related problems is a good starting point for testing ideas.

(v) **Search :** Some popular search strategies are :

(i) **Random :** Try random configuration of layers.

(ii) **Grid :** Systematic search across the number of layers and nodes per layer.

(iii) **Heuristic** : Try a directed search as a genetic algorithm or Bayesian optimization.

(iv) **Exhaustive** : Try all combinations of layers and number of nodes it may work for small networks and datasets.

☞ **Momentum, Learning Rate**

- A technique that can help the network to escape from local minima is the use of a **momentum** term. This is perhaps the most popular extension of backdrop algorithm. With momentum in, the weight update at a given time t becomes.

$$\Delta w_{ij}(t) = \mu_i \delta_i y_j + m \Delta w_{ij}(t-1),$$

Where $0 < m < 1$, is a new parameter which must be determined trial and error method.

- When the gradient keeps changing direction momentum will remove the variations. This is particularly useful when the network is not well – conditioned.

☞ **Learning Rate Adaptation**

We can employ simple heuristics to arrive at a reasonable guesses for the global and local learning rates. We can have some methods that can do **automatically** by adapting the learning rates during training.

(1) **Bold Driver** : A useful method for adapting the global learning rate μ is the **bold driver** algorithm.

Its operation is simple, after each epoch, compare the networks loss E(t) to its previous value E(t – 1). If the error has decreased, increase μ by a small proportion (say, 1% – 5%).

It the error has increased by more than a tiny proportion (say 10^{-10}), undo the last weight change, and decrease μ sharply, say be 50%.

Thus bold driver will keep growing μ slowly till it finds itself taking a step too far on the opposite slope of the error function. This show that the network has arrived in a trickey area of the error surface, here we reduce the step size quite drastically.

(2) **Annealing** : Bold driver cannot be used for online learning : the stochastic fluctuations in E(t) confuse the algorithm. In order to reach the minimum, and stay there, we must anneal (gradually lower) the global learning rate.

A simple, non-adaptive annealing schedule for this is 'search – then – converge' schedule.

$$\mu(t) = \frac{\mu(0)}{1 + \frac{t}{T}}$$

It keeps μ nearly constant for the first T training patterns. The characteristic time T of this schedule is a new free parameter that must be determined by trial and error method.

☞ **Regularization L_1 and L_2 Methods**

- Regularization is a technique which makes slight modifications to the learning algorithm such that the model generalises better. This improves model's performance on the unseen data also.

☞ **L_1 and L_2 Regularization**

- L_1 and L_2 are the most common types of regularization. These update the cost function by adding another term known as the regularization term.

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

- The addition of this regularization term the values of weigh – matrices decrease, because is assumes that a neural network with smaller weight matrices leads to simpler models. That reduces over fitting to some extent.

In L_2, we have, cost function = Loss + $\frac{\lambda}{2m} \cdot \sum || w ||^2$

- Here, lambda is the regularization parameter. It is hyper parameter whose value is optimised for better results. L_2 - regularization is also known as **weight decay** as it forces the weights to decay towards zero (but not exactly zero).

In L_1, we have, cost function = Loss + $\frac{\lambda}{2m} \cdot \sum || w ||$

- Here, we generalise the absolute value of the weights. Unlike L_2 the weights may be reduced to zero. Hence, it is useful when we are trying to compress our model. Otherwise we usually prefer L_2.

Dropout

- This is one of the most interesting types of regularisation techniques. It is frequently used in the field of deep learning. It produces very good results.

- To make the concept of dropout clear, we consider an example :

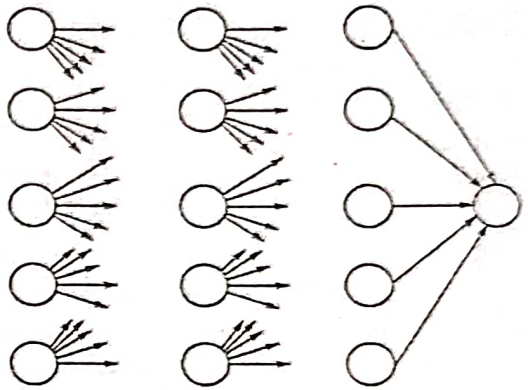


Fig. 1.21.1 : Dropout

Function of dropout :

At every iteration, it randomly selects some nodes and removes them along with all of their incoming and outgoing connections as shown :

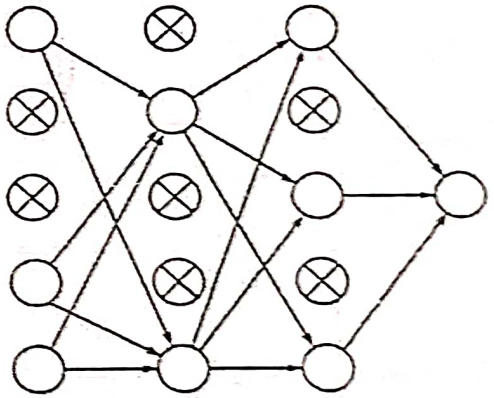
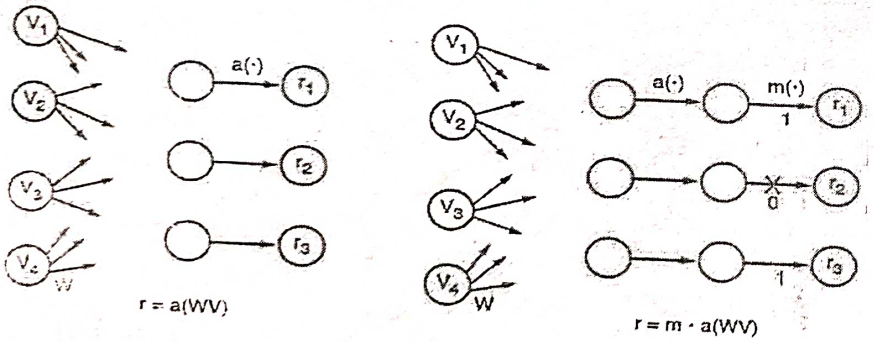


Fig. 1.21.2 : Iteration diagram

- Each iteration has a different set of nodes and this results in a different set of outputs. From the above figure, we can see that dropout can be applied to both the hidden layers as well as the input layer.
- The probability of choosing how may nodes should be dropped depends upon the hyper parameters of the dropout function.
- Dropout is usually preferred when we have a large neural network structure in order to introduce more randomness.

☞ **Drop connect**

- Drop connect is generalization of dropout for regularizing large fully – connected layers within neural networks. Drop connect sets a randomly selected subset of weights within the network to zero. Each unit receives input from a random subset of units in the previous layer.
- We derive a bound on the generalization performance of both Dropout and Drop Connect. We then evaluate Drop Connect on a range of datasets, comparing to Dropout by aggregating multiple Drop Connect trained models.



(a) No-Drop Network (b) Dropout Network

Fig. 1.21.3

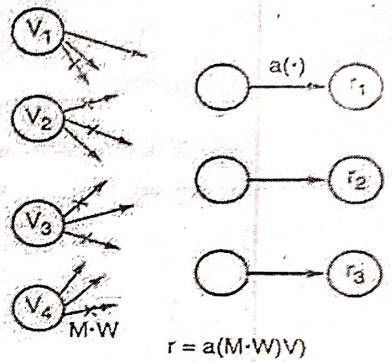


Fig. 1.21.4 : Deep Connect Network

...Chapter End

□□□