DR. D.Y. PATIL INSTITUTE OF TECHNOLOGY, PUNE
DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

LAB MANUAL

Software Laboratory II
Subject Code: 317533

Prepared By:
Aboli Suryawanshi

DR. D.Y. PATIL INSTITUTE OF TECHNOLOGY, PUNE
DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

Lab Manual

Third Year Engineering

Semester-V

Software Laboratory II

Subject Code: 317533

Class: TE AI&DS

Academic Year 2023-24

Software Laboratory III
Subject Code: 317534

Teaching Scheme	Credit	Examination Scheme
PR: 04 Hours/Week	02	PR: 25 Marks TW: 50 Marks

Guidelines for Instructor's Manual

The instructor‘s manual is to be developed as a reference and hands-on resource. It should include prologue (about University/program/ institute/ department/foreword/ preface), curriculum of the course, conduction and Assessment guidelines, topics under consideration, concept, objectives, outcomes, set of typical applications/assignments/ guidelines, and references

Guidelines for Student Journal

The laboratory assignments are to be submitted by student in the form of journal. Journal consists of Certificate, table of contents, and handwritten write-up of each assignment (Title, Date of Completion, Objectives, Problem Statement, Software and Hardware requirements, Assessment grade/marks and assessor's sign, Theory- Concept in brief, algorithm, flowchart, test cases, Test Data Set(if applicable), mathematical model (if applicable), conclusion/analysis. Program codes with sample output of all performed assignments are to be submitted as softcopy. As a conscious effort and little contribution towards Green IT and environment awareness, attaching printed papers as part of write-ups and program listing to journal must be avoided. Use of DVD containing students programs maintained by Laboratory In-charge is highly encouraged. For reference one or two journals may be maintained with program prints in the Laboratory.

Guidelines for Laboratory /Term Work Assessment

Continuous assessment of laboratory work should be based on overall performance of Laboratory assignments by a student. Each Laboratory assignment assessment will assign grade/marks based on parameters, such as timely completion, performance, innovation, efficient codes, and punctuality.

Guidelines for Practical Examination

Problem statements must be decided jointly by the internal examiner and external examiner. During practical assessment, maximum weightage should be given to satisfactory implementation of the problem statement. Relevant questions may be asked at the time of evaluation to test the student’s understanding of the fundamentals, effective and efficient implementation. This will encourage, transparent evaluation and fair approach, and hence will not create any uncertainty or doubt in the minds of the students. So, adhering to these principles will consummate our team efforts to the promising start of student's academics.

Guidelines for Laboratory Conduction

The instructor is expected to frame the assignments by understanding the prerequisites, technological aspects, utility and recent trends related to the topic. The assignment framing policy need to address the average students and inclusive of an element to attract and promote the intelligent students. Use of open source software is encouraged. Based on the concepts learned. Instructor may also set one assignment or mini-project that is suitable to AI & DS branch beyond the scope of the syllabus.

Suggested List of Laboratory Experiments/Assignments
Group A (Any 6)
1. Write a Python program to plot a few activation functions that are being used in neural networks.
2. Generate ANDNOT function using McCulloch-Pitts neural net by a python program.
3. Write a Python Program using Perceptron Neural Network to recognise even and odd numbers. Given numbers are in ASCII form 0 to 9
4. With a suitable example demonstrate the perceptron learning law with its decision regions using Python. Give the output in graphical form.
5. Write a python Program for Bidirectional Associative Memory with two pairs of vectors.
6. Write a python program to recognize the number 0, 1, 2, 39. A 5 * 3 matrix forms the numbers. For any valid point it is taken as 1 and invalid point it is taken as 0. The net has to be trained to recognize all the numbers and when the test data is given, the network has to recognize the particular numbers
7. Implement Artificial Neural Network training process in Python by using Forward Propagation, Back Propagation.
8. Create a Neural network architecture from scratch in Python and use it to do multi-class classification on any data. Parameters to be considered while creating the neural network from scratch are specified as: (1) No of hidden layers : 1 or more (2) No. of neurons in hidden layer: 100 (3) Non-linearity in the layer : Relu (4) Use more than 1 neuron in the output layer. Use a suitable threshold value Use appropriate Optimisation algorithm
Group B (Any 4)
1. Write a python program to show Back Propagation Network for XOR function with Binary Input and Output
2. Write a python program to illustrate ART neural network.
3. Write a python program in python program for creating a Back Propagation Feed-forward neural network
4. Write a python program to design a Hopfield Network which stores 4 vectors
5. Write Python program to implement CNN object detection. Discuss numerous performance evaluation metrics for evaluating the object detecting algorithms' performance.
Group C (Any 3)
1. How to Train a Neural Network with TensorFlow/Pytorch and evaluation of logistic regression using tensorflow

2. TensorFlow/Pytorch implementation of CNN

3. For an image classification challenge, create and train a ConvNet in Python using TensorFlow. Also try to improve the performance of the model by applying various hyper parameter tuning to reduce the overfitting or under fitting problem that might occur. Maintain graphs of comparisons.

4. MNIST Handwritten Character Detection using PyTorch, Keras and Tensorflow

Mini Project

Car Object Detection using (ConvNet/CNN) Neural Network

Car Object Data: Data Source – https://www.kaggle.com/datasets/sshikamaru/car-object-detection
The dataset contains images of cars in all views.

Training Images – Set of 1000 files
Use Tensorflow, Keras & Residual Network resNet50

Constructs comparative outputs for various Optimisation algorithms and finds out good accuracy.
OR

Mini Project to implement CNN object detection on any data. Discuss numerous performance evaluation metrics for evaluating the object detecting algorithms' performance, Take outputs as a comparative results of algorithms.

Learning Resources

Text Books:

1. Neural Networks a Comprehensive Foundations, Simon Haykin, PHI edition.
2. Laurene Fausett:Fundamentals of Neural Networks: Architectures, Algorithms & Apps, Pearson, 2004.
3. Learn TensorFlow 2.0: Implement Machine Learning and Deep Learning Models with Python 1st ed. Edition, Apress publication

Reference Books:

1. Getting Started with TensorFlow, by Giancarlo Zaccone
2. AI and Machine learning for coders by Laurence Moroney, O'Reilly Media, Inc.

e- Books:

1. https://www.inf.ed.ac.uk/teaching/courses/nlu/assets/reading/Gurney_et_al.pdf
2. http://neuralnetworksanddeeplearning.com/

MOOC Courses:

1. **http://neuralnetworksanddeeplearning.com/**
2. **https://www.coursera.org/learn/convolutional-neural-networks-tensorflow**
3. **https://nptel.ac.in/courses/106106213**

Group A

ASSIGNMENT 1

TITLE: ACTIVATION FUNCTION

PROBLEM STATEMENT: -

Write a Python program to plot a few activation functions that are being used in neural networks.

OBJECTIVE:

1. To Learn and understand the concepts of activation function.
2. To learn and understand application of activation function in the neural network

PREREQUISITE: -

1 Basic of Python Programming

2 Concept of Artificial Neural Network

THEORY:

An activation function is a mathematical operation applied to each node (or neuron) in a neural network layer. The activation function determines whether a neuron should be activated or not based on the weighted sum of its input. It introduces non-linearity to the network, enabling it to learn complex relationships in the data.

It is used to determine the output of neural network like yes or no. It maps the resulting values in between 0 to 1 or -1 to 1 etc. (depending upon the function).

The Activation Functions can be basically divided into 2 types-

1) **Linear Activation Function**
2) **Non-linear Activation Functions**

1) Linear or Identity Activation Function

A linear function is also known as a straight-line function where the activation is proportional to the input i.e. the weighted sum from neurons. It has a simple function with the equation:

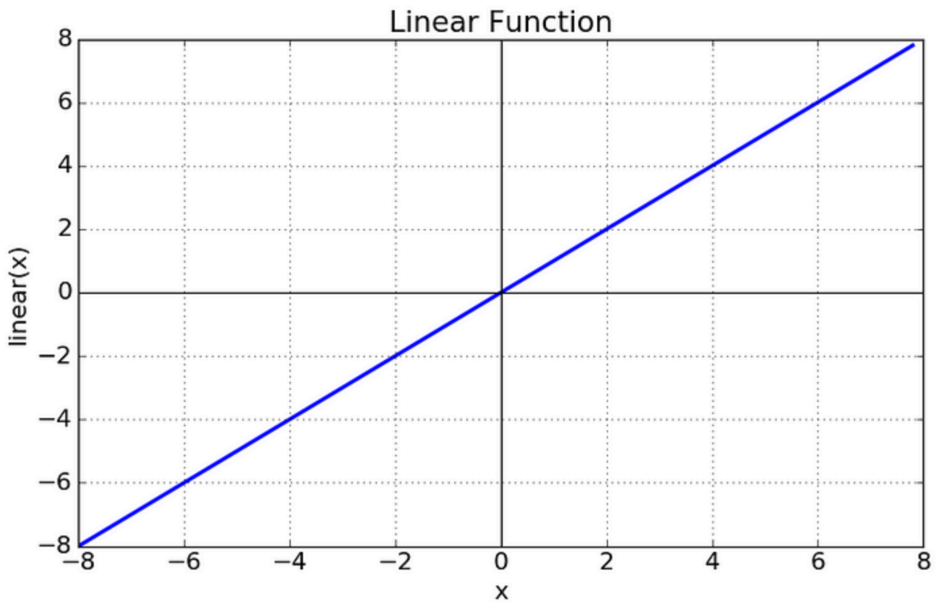


Fig: Linear Activation Function

Equation: f(x) = x
Range: (-infinity to infinity)

2)Non-Linear Activation Functions

The non-linear functions are known to be the most used activation functions. It makes it easy for a neural network model to adapt with a variety of data and to differentiate between the outcomes.

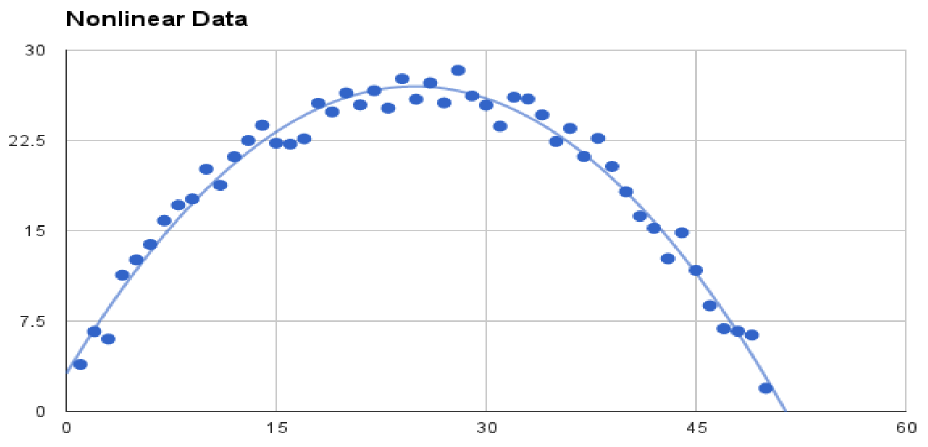


Fig: Non-linear Activation Function

Types of Non-linear Activation Function

1) Sigmoid Function

2) Tanh Function

3) RELU Function

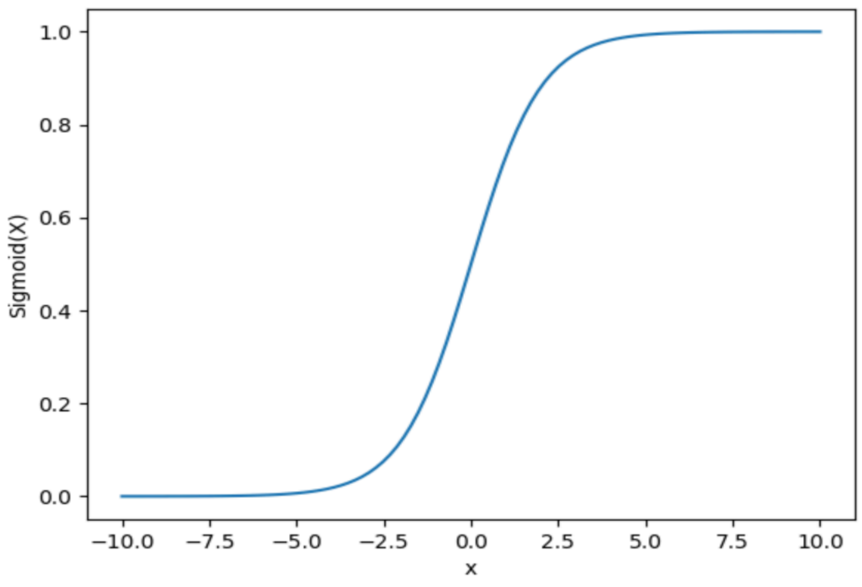
1) Sigmoid Function

It is a function which is plotted as ‘S’ shaped graph.

Equation: A = 1/(1 + e-x)

Nature: Non-linear. Notice that X values lies between -2 to 2, Y values are very steep. This means, small changes in x would also bring about large changes in the value of Y.

Value Range: 0 to 1



Application: Usually used in output layer of a binary classification, where result is either 0 or 1, as value for sigmoid function lies between 0 and 1 only so, result can be predicted easily to be 1 if value is greater than 0.5 and 0 otherwise.

2) Tanh Function

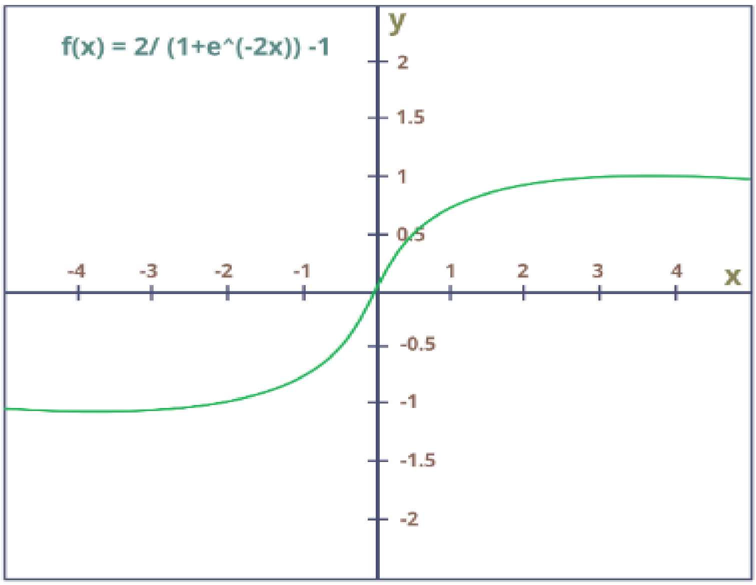
The activation that works almost always better than sigmoid function is Tanh function also known as Tangent Hyperbolic function. It’s actually mathematically shifted version of the sigmoid function. Both are similar and can be derived from each other.

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

Equation:-

Value Range: - -1 to +1

Nature: - non-linear



Uses: - Usually used in hidden layers of a neural network as it's values lies between -1 to 1 hence the mean for the hidden layer comes out be 0 or very close to it, hence helps in centering the data by bringing mean close to 0. This makes learning for the next layer much easier.

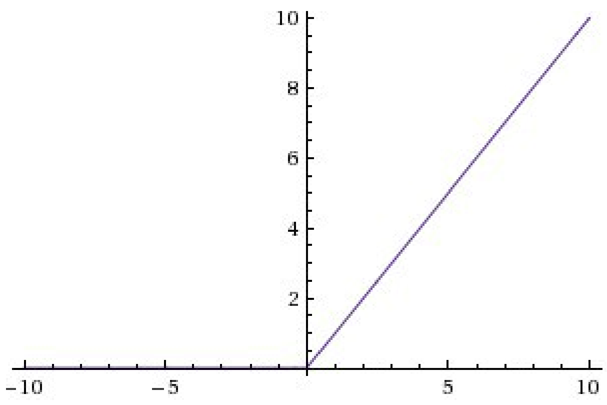
RELU Function

It Stands for Rectified linear unit. It is the most widely used activation function. Chiefly implemented in hidden layers of Neural network.

Equation :- A(x) = max(0,x). It gives an output x if x is positive and 0 otherwise.

Value Range :- [0, inf)

Nature :- non-linear, which means we can easily back propagate the errors and have multiple layers of neurons being activated by the ReLU function.



Uses: - ReLu is less computationally expensive than tanh and sigmoid because it involves simpler mathematical operations. At a time only a few neurons are activated making the network sparse making it efficient and easy for computation.

CONCLUSION: We have studied and implemented the activation function in python

ASSIGNMENT QUESTION

1) Explain the role of activation functions in artificial neural networks.
2) Compare and contrast linear and nonlinear activation functions. Provide examples of each.
3) Define the sigmoid and hyperbolic tangent (tanh) activation functions.
4) Discuss the advantages and disadvantages of using sigmoid and tanh functions in neural networks.
5) Describe the Rectified Linear Unit (ReLU) activation function.
6) Why is the sigmoid activation function commonly used in the output layer of binary classification problems?
7) Provide examples of scenarios where one activation function might be more suitable than another.
8) Given a specific problem (e.g., image classification, regression), recommend an activation function and justify your choice.

ASSIGNMENT 2

TITLE: McCulloch-Pitts neural Network

PROBLEM STATEMENT: -

Generate ANDNOT function using McCulloch-Pitts neural net by a python program.

OBJECTIVE:

1. To learn ANDNOT function using McCulloch-Pitts neural net

PREREQUISITE: -

1) Basic of Python Programming
2) Concept of Artificial Neural Network

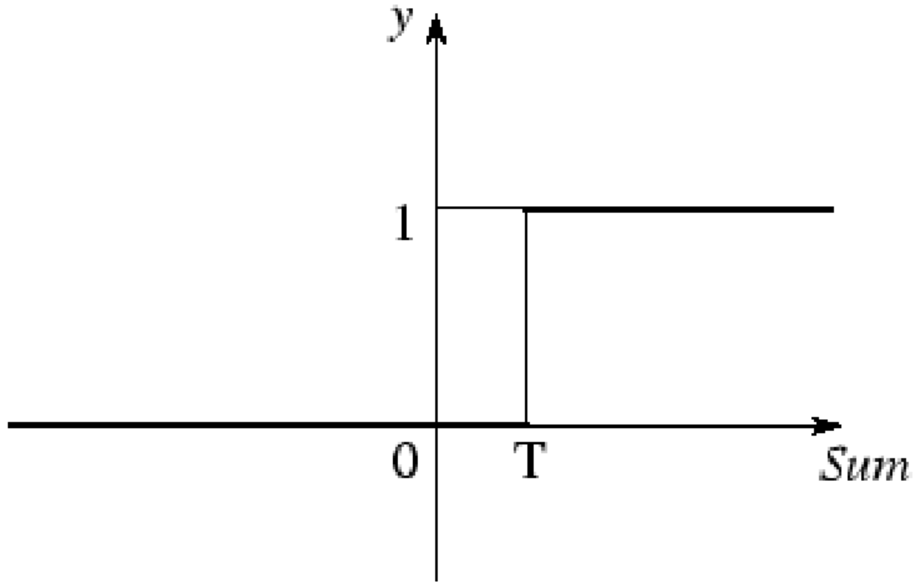
THEORY:

Theory: McCulloch-Pitts neural model: The early model of an artificial neuron is introduced by Warren McCulloch and Walter Pitts in 1943. The McCulloch-Pitts neural model is also known as linear threshold gate. It is a neuron of a set of inputs ***I_1, I_2, I_3……..I_m*** and one output ***y***. The linear threshold gate simply classifies the set of inputs into two different classes. Thus the output ***y*** is binary. Such a function can be described mathematically using these equations:

$$Sum = \sum_{i=1}^{N} I_i W_i,$$

$$y = f(Sum).$$

W_1, W_2,W_3……W_m are weight values normalized in the range of either ***(0,1) or (1,-1)***and associated with each input line, ***Sum*** is the weighted sum, and ***T*** is a threshold constant. The function ***f*** is a linear step function at threshold ***T*** as shown in figure below. The symbolic representation of the linear threshold gate is shown in figure.



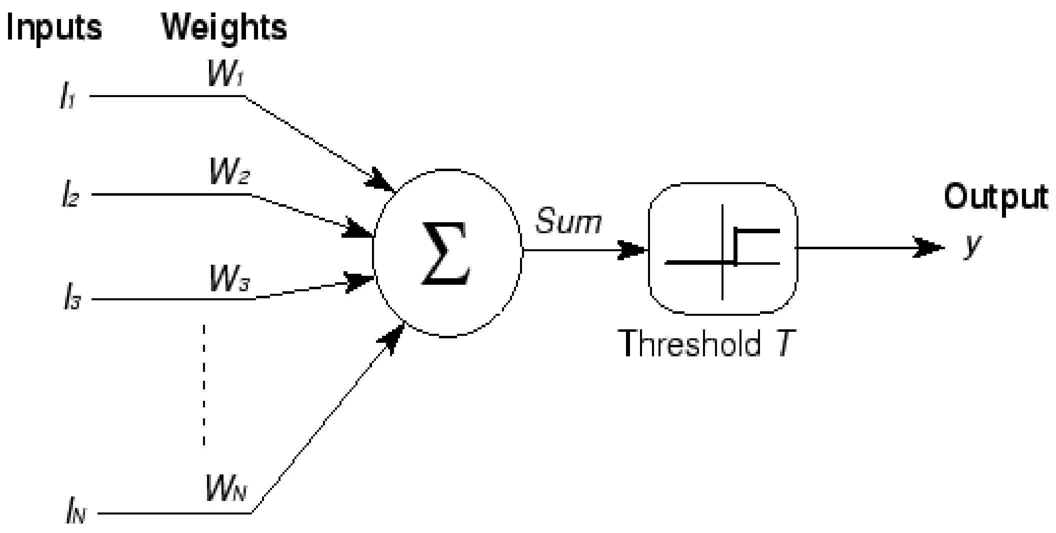


Figure A: Linear Threshold function B:Symbolic illustration of linear threshold gate

The McCulloch-Pitts model of a neuron is simple yet has substantial computing potential. It also has a precise mathematical definition. However, this model is so simplistic that it only generates a binary output and also the weight and threshold values are fixed. The neural computing algorithm has diverse features for various applications. Thus, we need to obtain the neural model with more flexible computational features.

Activation Value: X= $\left(\sum_{i=0}^{n} w_i x_i\right)$

Output Function

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

ANDNOT Function:

Truth Table:

X1	X2	Y
1	1	0
1	0	1
0	1	0
0	0	0

ANN with two input neurons and a single output neuron can operate as an ANDNOT logic function if we choose weights

W1 = 1, W2 = -1 and threshold ϴ = 1.

Yin is a activation value

X1=1, X2=1,

Yin= W1*X1+W2*X2=1*1+(-1)*1=0, Yin< ϴ , so Y=0

X1=1, X2=0

Yin=1*1+0*(-1)=1, Yin= ϴ , so Y=1

X1=0, X=1

Yin=0-1=-1, Yin< ϴ , so Y=0

X1=0, X2=0

Yin=0, Yin< ϴ , so Y=0

So , Y=[0 1 0 0]

EXPECTED OUTPUT / CALCULATION / RESULT:

Weights of Neuron: w1=1 w2=-1

Threshold: Theta=1

Output:

w1=1

w2=-1

Threshold:

Theta=1

With Output of Neuron: 0 1 0 0

CONCLUSION: We have studied and implemented ANDNOT function using McCulloch-Pitts neural net

ASSIGNMENT QUESTION

Q1. Explain MCP Model?

Q2. How to generate AND NOT function using MCP Model ?

Q3. Give Mc Culloch Pitt's artificial neuron model limitations?

ASSIGNMENT 3

TITLE: PERCEPTRON NEURAL NETWORK

PROBLEM STATEMENT: -

Write a Python Program using Perceptron Neural Network to recognize even and odd numbers. Given numbers are in ASCII form 0 to 9

OBJECTIVE:

1. To learn types of neural network

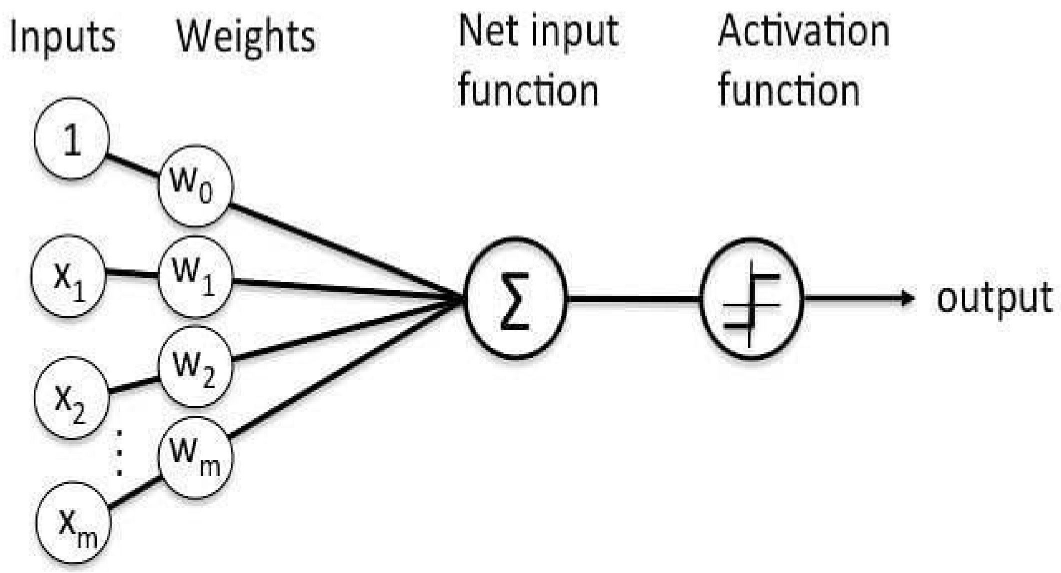
PREREQUISITE: -

1) Basic of Python Programming
2) Concept of Artificial Neural Network
3) Feed-forward neural network

THEORY:

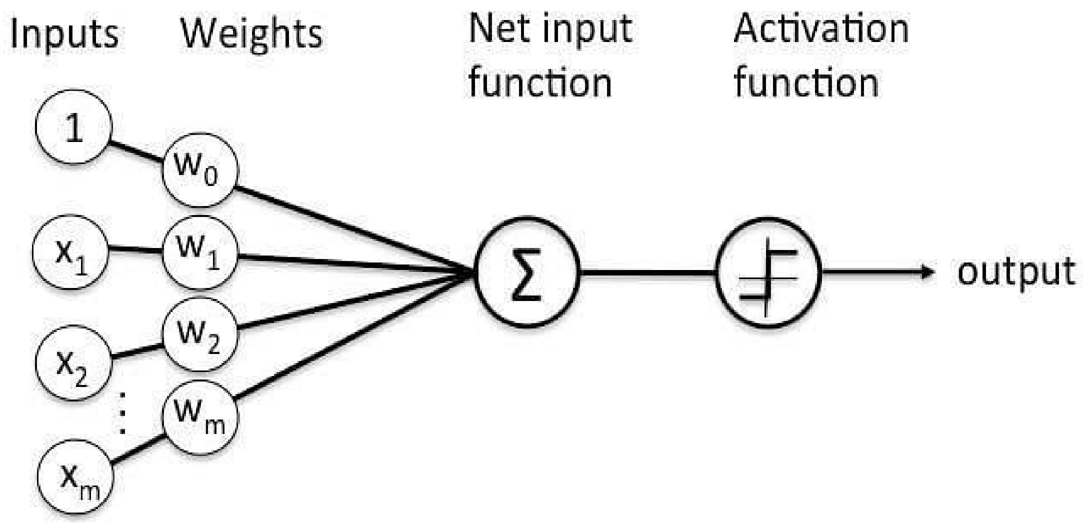
Perceptron

Perceptron was introduced by Frank Rosenblatt in 1957. He proposed a Perceptron learning rule based on the original MCP neuron. A Perceptron is an algorithm for supervised learning of binary classifiers. This algorithm enables neurons to learn and processes elements in the training set one at a time.



Perceptron

Perceptron was introduced by Frank Rosenblatt in 1957. He proposed a Perceptron learning rule based on the original MCP neuron. A Perceptron is an algorithm for supervised learning of binary classifiers. This algorithm enables neurons to learn and processes elements in the training set one at a time.



Basic Components of Perceptron

Perceptron is a type of artificial neural network, which is a fundamental concept in machine learning. The basic components of a perceptron are:

1. **Input Layer:** The input layer consists of one or more input neurons, which receive input signals from the external world or from other layers of the neural network.
2. **Weights:** Each input neuron is associated with a weight, which represents the strength of the connection between the input neuron and the output neuron.
3. **Bias:** A bias term is added to the input layer to provide the perceptron with additional flexibility in modeling complex patterns in the input data.
4. **Activation Function:** The activation function determines the output of the perceptron based on the weighted sum of the inputs and the bias term. Common activation functions used in perceptrons include the step function, sigmoid function, and ReLU function.
5. **Output:** The output of the perceptron is a single binary value, either 0 or 1, which indicates the class or category to which the input data belongs.
6. **Training Algorithm:** The perceptron is typically trained using a supervised learning algorithm such as the perceptron learning algorithm or backpropagation. During training, the weights and biases of the perceptron are adjusted to minimize the error between the predicted output and the true output for a given set of training examples.
7. Overall, the perceptron is a simple yet powerful algorithm that can be used to perform binary classification tasks and has paved the way for more complex neural networks used in deep learning today.

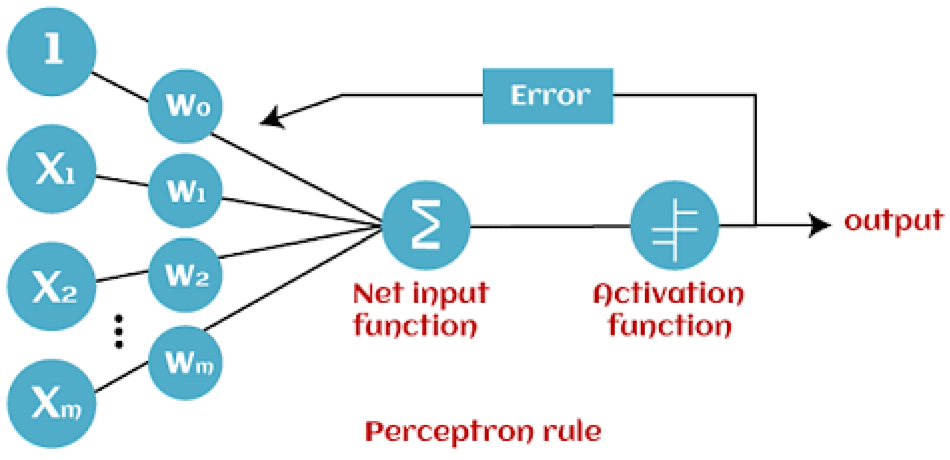
Types of Perceptron:

1. **Single layer:** Single layer perceptron can learn only linearly separable patterns.
2. **Multilayer:** Multilayer perceptrons can learn about two or more layers having a greater processing power.

The Perceptron algorithm learns the weights for the input signals in order to draw a linear decision boundary.

How Does Perceptron Work?

AS discussed earlier, Perceptron is considered a single-layer neural link with four main parameters. The perceptron model begins with multiplying all input values and their weights, then adds these values to create the weighted sum. Further, this weighted sum is applied to the activation function 'f' to obtain the desired output. This activation function is also known as the step function and is represented by 'f.'



This step function or Activation function is vital in ensuring that output is mapped between (0,1) or (-1,1). Take note that the weight of input indicates a node's strength. Similarly, an input value gives the ability the shift the activation function curve up or down.

Step 1: Multiply all input values with corresponding weight values and then add to calculate the weighted sum. The following is the mathematical expression of it:

∑wi*xi = x1*w1 + x2*w2 + x3*w3+……..x4*w4

Add a term called bias 'b' to this weighted sum to improve the model's performance.

Step 2: An activation function is applied with the above-mentioned weighted sum giving us an output either in binary form or a continuous value as follows:

Y=f(∑wi*xi + b)

Types of Perceptron models

We have already discussed the types of Perceptron models in the Introduction. Here, we shall give a more profound look at this:

1. **Single Layer Perceptron model:** One of the easiest ANN(Artificial Neural Networks) types consists of a feed-forward network and includes a threshold transfer inside the model. The main objective of the single-layer perceptron model is to analyze the linearly separable objects with binary outcomes. A Single-layer perceptron can learn only linearly separable patterns.

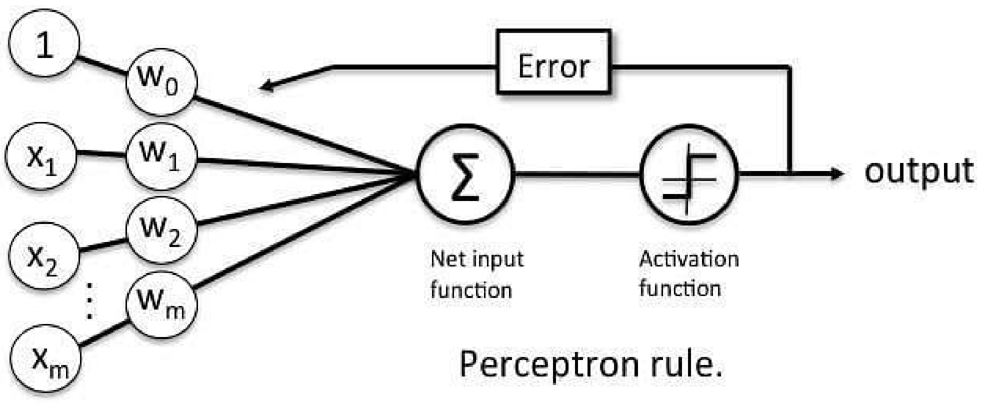
2. **Multi-Layered Perceptron model:** It is mainly similar to a single-layer perceptron model but has more hidden layers.

Forward Stage: From the input layer in the on stage, activation functions begin and terminate on the output layer.

Backward Stage: In the backward stage, weight and bias values are modified per the model's requirement. The backstage removed the error between the actual output and demands originating backward on the output layer. A multilayer perceptron model has a greater processing power and can process linear and non-linear patterns. Further, it also implements logic gates such as AND, OR, XOR, XNOR, and NOR.

Perceptron Learning Rule

Perceptron Learning Rule states that the algorithm would automatically learn the optimal weight coefficients. The input features are then multiplied with these weights to determine if a neuron fires or not.



Perceptron rule.

The Perceptron receives multiple input signals, and if the sum of the input signals exceeds a certain threshold, it either outputs a signal or does not return an output. In the context of supervised learning and classification, this can then be used to predict the class of a sample.

Perceptron Function

Perceptron is a function that maps its input "x," which is multiplied with the learned weight coefficient; an output value "f(x)"is generated.

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

In the equation given above:

- “w” = vector of real-valued weights
- “b” = bias (an element that adjusts the boundary away from origin without any dependence on the input value)
- “x” = vector of input x values

$$\sum_{i=1}^{m} w_i x_i$$

- “m” = number of inputs to the Perceptron

The output can be represented as “1” or “0.” It can also be represented as “1” or “-1” depending on which activation function is used.

Algorithm:

Inputs:

X: input features, representing an image of a number

y: binary target class label (0 for even, 1 for odd)

w: weight vector

b: bias term

alpha: learning rate

num_iterations: number of iterations to run gradient descent

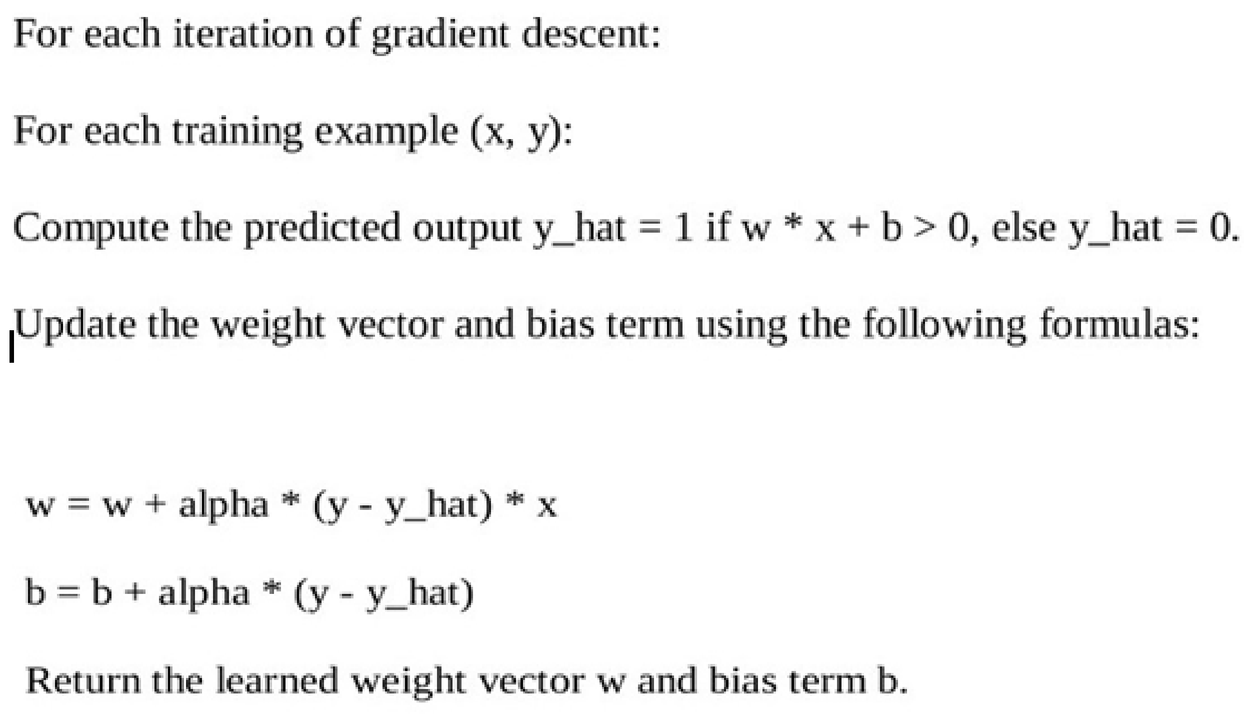
Outputs:

w: the learned weight vector

b: the learned bias term

Steps:

Initialize the weight vector w and bias term b to zero.



For each iteration of gradient descent:

For each training example (x, y):

Compute the predicted output y_hat = 1 if w * x + b > 0, else y_hat = 0.

Update the weight vector and bias term using the following formulas:

w = w + alpha * (y - y_hat) * x

b = b + alpha * (y - y_hat)

Return the learned weight vector w and bias term b.

CONCLUSION: We have studied and implemented perceptron model for recognizing Even and odd numbers for ASCII 0-9.

ASSIGNMENT QUESTION

Q1. is it possible to train a NN to distinguish between odd and even numbers only using as input the numbers themselves?

Q2. Can Perceptron Generalize Non-linears Problems?

Q3. How to create a Multilayer Perceptron NN?

Q4. Is the multilayer perceptron (MLP) a deep learning method? Explain it?

Q5. What is the best way to fit a large amount of nonlinear data using a neural network?

ASSIGNMENT 4

TITLE: FEED-FORWARD NEURAL NETWORK

PROBLEM STATEMENT: -

With a suitable example demonstrate the perceptron learning law with its decision regions using python. Give the output in graphical form

OBJECTIVE:

1. To learn types of neural network
2. To implement the Back Propagation Feed-forward neural network

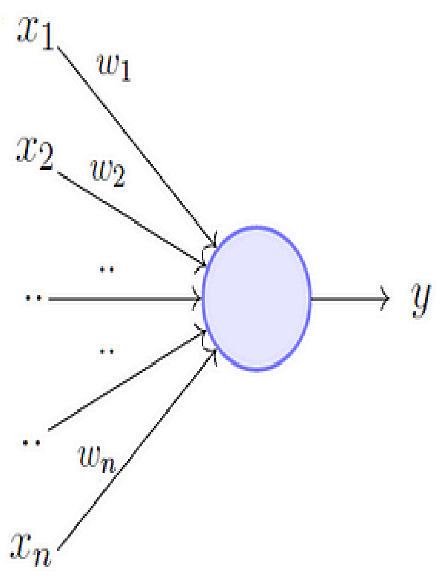
PREREQUISITE: -

1) Basic of Python Programming
2) Concept of Artificial Neural Network
3) Feed-forward neural network

THEORY:

1. Perceptron

You can just go through my previous post on the perceptron model (linked above) but I will assume that you won't. So here goes, a perceptron is not the Sigmoid neuron we use in ANNs or any deep learning networks today.

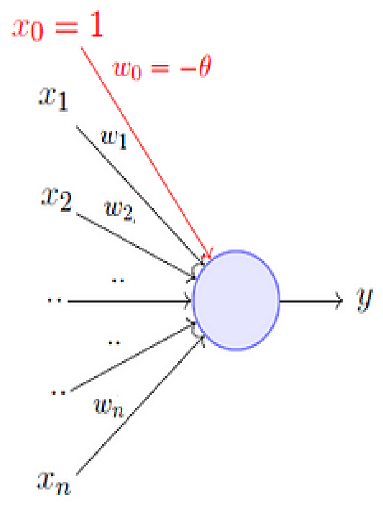


$$y = 1 \quad if \sum_{i=1}^{n} w_i * x_i \geq \theta$$
$$= 0 \quad if \sum_{i=1}^{n} w_i * x_i < \theta$$

Rewriting the above,

$$y = 1 \quad if \sum_{i=1}^{n} w_i * x_i - \theta \geq 0$$
$$= 0 \quad if \sum_{i=1}^{n} w_i * x_i - \theta < 0$$

The perceptron model is a more general computational model than McCulloch-Pitts neuron. It takes an input, aggregates it (weighted sum) and returns 1 only if the aggregated sum is more than some threshold else returns 0. Rewriting the threshold as shown above and making it a constant input with a variable weight, we would end up with something like the following:



A more accepted convention,

$$y = 1 \quad if \sum_{i=0}^{n} w_i * x_i \geq 0$$
$$= 0 \quad if \sum_{i=0}^{n} w_i * x_i < 0$$

$$where, \quad x_0 = 1 \quad and \quad w_0 = -\theta$$

A single perceptron can only be used to implement **linearly separable** functions. It takes both real and boolean inputs and associates a set of **weights** to them, along with a **bias** (the threshold thing I mentioned above). We learn the weights, we get the function. Let's use a perceptron to learn an OR function.

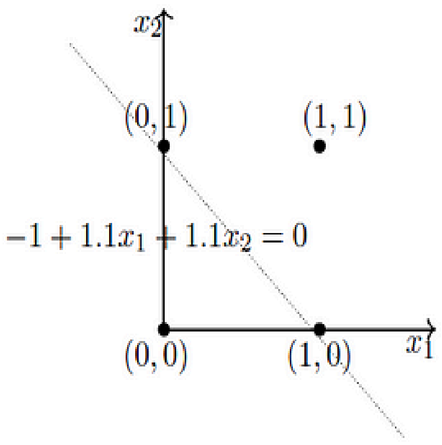
OR Function Using A Perceptron

x_1	x_2	OR	
0	0	0	$w_0 + \sum_{i=1}^{2} w_i x_i < 0$
1	0	1	$w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$
0	1	1	$w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$
1	1	1	$w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$

$$w_0 + w_1 \cdot 0 + w_2 \cdot 0 < 0 \implies w_0 < 0$$
$$w_0 + w_1 \cdot 0 + w_2 \cdot 1 \geq 0 \implies w_2 > -w_0$$
$$w_0 + w_1 \cdot 1 + w_2 \cdot 0 \geq 0 \implies w_1 > -w_0$$
$$w_0 + w_1 \cdot 1 + w_2 \cdot 1 \geq 0 \implies w_1 + w_2 > -w_0$$

One possible solution is

$$w_0 = -1,\ w_1 = 1.1,\ w_2 = 1.1$$



What's going on above is that we defined a few conditions (the weighted sum has to be more than or equal to 0 when the output is 1) based on the OR function output for various sets of inputs, we solved for weights based on those conditions and we got a line that perfectly separates positive inputs from those of negative.

Doesn't make any sense? Maybe now is the time you go through that post I was talking about. Minsky and Papert also proposed a more principled way of learning these weights using a set of examples (data). Mind you that this is NOT a Sigmoid neuron and we're not going to do any Gradient Descent.

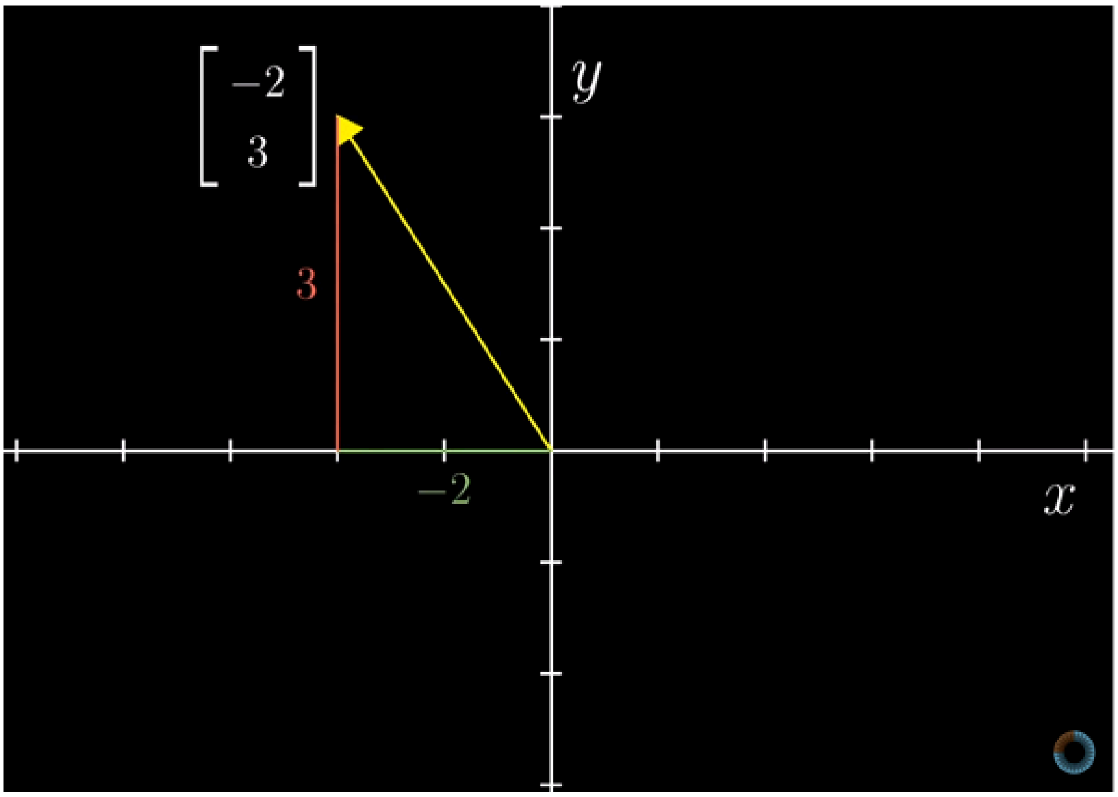
2. Warming Up — Basics Of Linear Algebra

Vector

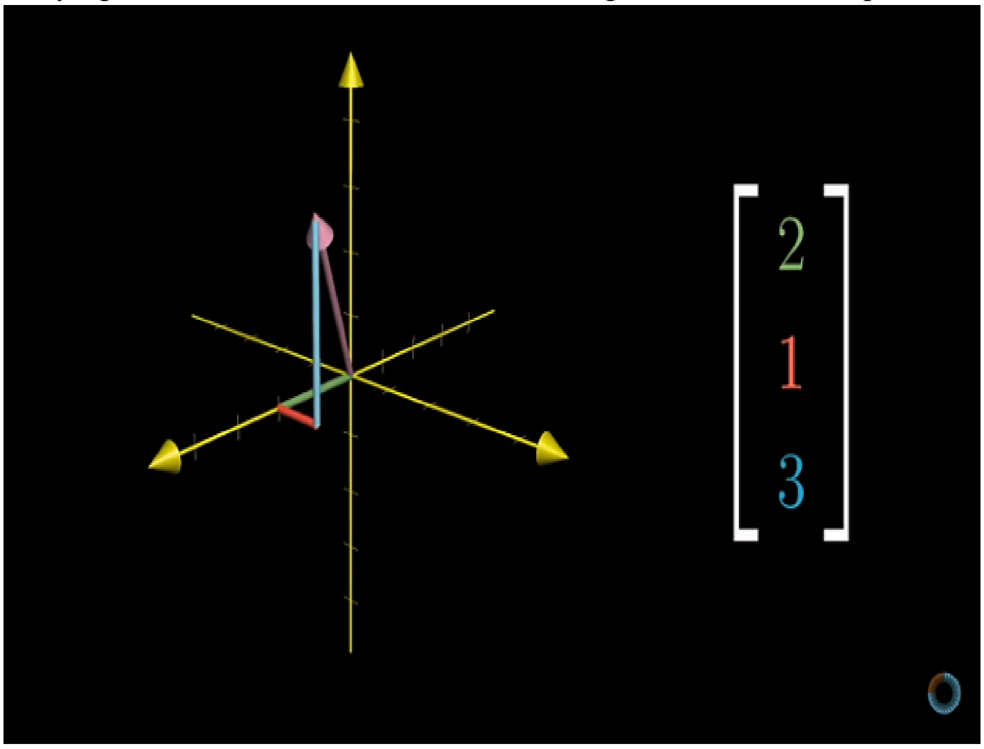
A vector can be defined in more than one way. For a physicist, a vector is anything that sits anywhere in space, has a magnitude and a direction. For a CS guy, a vector is just a data structure used to store some data — integers, strings etc. For this tutorial, I would like you to imagine a vector the Mathematician way, where a vector is an arrow spanning in space with its tail at the origin. This is not the best mathematical way to describe a vector but as long as you get the intuition, you're good to go.

Vector Representations

A 2-dimensional vector can be represented on a 2D plane as follows:



Carrying the idea forward to 3 dimensions, we get an arrow in 3D space as follows:



Dot Product Of Two Vectors

At the cost of making this tutorial even more boring than it already is, let's look at what a dot product is. Imagine you have two vectors oh size **n+1**, **w** and **x**, the dot product of these vectors (***w.x***) could be computed as follows:

$$\mathbf{w} = [w_0, w_1, w_2, ..., w_n]$$

$$\mathbf{x} = [1, x_1, x_2, ..., x_n]$$

$$\mathbf{w} \cdot \mathbf{x} = \mathbf{w}^{\mathrm{T}}\mathbf{x} = \sum_{i=0}^{n} w_i * x_i$$

The transpose is just to write it in a matrix multiplication form.

Here, **w** and **x** are just two lonely arrows in an **n+1 dimensional** space (and intuitively, their dot product quantifies how much one vector is going in the direction of the other). So technically, the perceptron was only computing a lame dot product (before checking if it's greater or lesser than 0). The decision boundary line which a perceptron gives out that separates positive examples from the negative ones is really just **w . x** = 0.

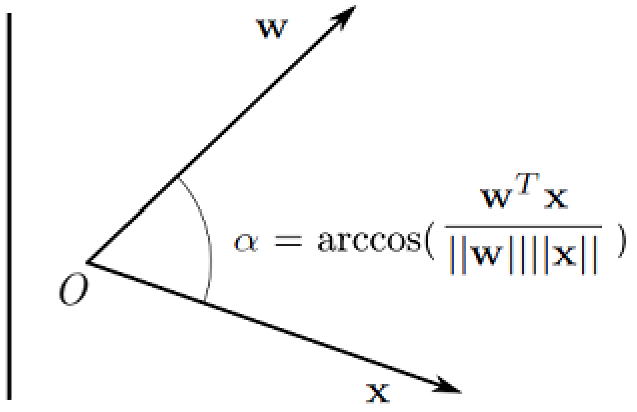
Angle Between Two Vectors

Now the same old dot product can be computed differently if only you knew the angle between the vectors and their individual magnitudes. Here's how:

$$\mathbf{w}^T\mathbf{x} = \|\mathbf{w}\|\|\mathbf{x}\| \, cos\alpha$$

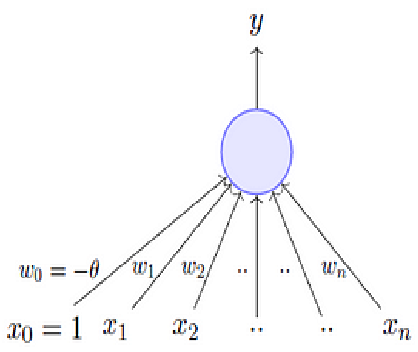
The other way around, you can get the angle between two vectors, if only you knew the vectors, given you know how to calculate vector magnitudes and their vanilla dot product.

$$cos\alpha = \frac{\mathbf{w}^T\mathbf{x}}{||\mathbf{w}||||\mathbf{x}||}$$


w
O
$\alpha = \arccos(\frac{\mathbf{w}^T\mathbf{x}}{||\mathbf{w}||||\mathbf{x}||})$
x

When I say that the cosine of the angle between **w** and **x** is 0, what do you see? I see arrow **w** being perpendicular to arrow **x** in an n+1 dimensional space (in 2-dimensional space to be honest). So basically, when the dot product of two vectors is 0, they are perpendicular to each other.

Setting Up The Problem



$$x_1 = isActorDamon$$
$$x_2 = isGenreThriller$$
$$x_3 = isDirectorNolan$$
$$x_4 = imdbRating(scaled\ to\ 0\ to\ 1)$$
$$\dots \quad \dots$$
$$x_n = criticsRating(scaled\ to\ 0\ to\ 1)$$

We are going to use a perceptron to estimate if I will be watching a movie based on historical data with the above-mentioned inputs. The data has positive and negative examples, positive being the movies I watched i.e., 1. Based on the data, we are going to learn the weights using the perceptron learning algorithm. For visual simplicity, we will only assume two-dimensional input.

3. Perceptron Learning Algorithm

Our goal is to find the **w** vector that can perfectly classify positive inputs and negative inputs in our data. I will get straight to the algorithm. Here goes:

Algorithm: Perceptron Learning Algorithm

```
$P \leftarrow inputs \quad with \quad label \quad 1;$
$N \leftarrow inputs \quad with \quad label \quad 0;$
Initialize $\mathbf{w}$ randomly;
while !convergence do
    Pick random $\mathbf{x} \in P \cup N$ ;
    if $\mathbf{x} \in P \quad and \quad \mathbf{w.x} < 0$ then
        $\mathbf{w} = \mathbf{w} + \mathbf{x}$ ;
    end
    if $\mathbf{x} \in N \quad and \quad \mathbf{w.x} \geq 0$ then
        $\mathbf{w} = \mathbf{w} - \mathbf{x}$ ;
    end
end
//the algorithm converges when all the
 inputs are classified correctly
```

We initialize **w** with some random vector. We then iterate over all the examples in the data, (*P* U *N*) both positive and negative examples. Now if an input **x** belongs to *P*, ideally what should the dot product **w.x** be? I'd say greater than or equal to 0 because that's the only thing what our perceptron wants at the end of the day so let's give it that. And if **x** belongs to *N*, the dot product MUST be less than 0. So if you look at the if conditions in the while loop:

```
while !convergence do
    Pick random $\mathbf{x} \in P \cup N$ ;
    if $\mathbf{x} \in P \quad and \quad \mathbf{w.x} < 0$ then
        $\mathbf{w} = \mathbf{w} + \mathbf{x}$ ;
    end
    if $\mathbf{x} \in N \quad and \quad \mathbf{w.x} \geq 0$ then
        $\mathbf{w} = \mathbf{w} - \mathbf{x}$ ;
    end
end
```

Case 1: When **x** belongs to *P* and its dot product **w.x** < 0
Case 2: When **x** belongs to *N* and its dot product **w.x** ≥ 0

Only for these cases, we are updating our randomly initialized **w**. Otherwise, we don't touch **w** at all because Case 1 and Case 2 are violating the very rule of a perceptron. So we are adding **x** to **w** (ahem vector addition ahem) in Case 1 and subtracting **x** from **w** in Case 2.

Why Would The Specified Update Rule Work?

But why would this work? If you get it already why this would work, you've got the entire gist of my post and you can now move on with your life, thanks for reading, bye. But if you are not sure why these seemingly arbitrary operations of **x** and **w** would help you learn that perfect **w** that can perfectly classify *P* and *N*, stick with me.

We have already established that when **x** belongs to *P*, we want **w.x** > 0, basic perceptron rule. What we also mean by that is that when **x** belongs to *P*, the angle between **w** and **x** should be _____ than 90 degrees. Fill in the blank.

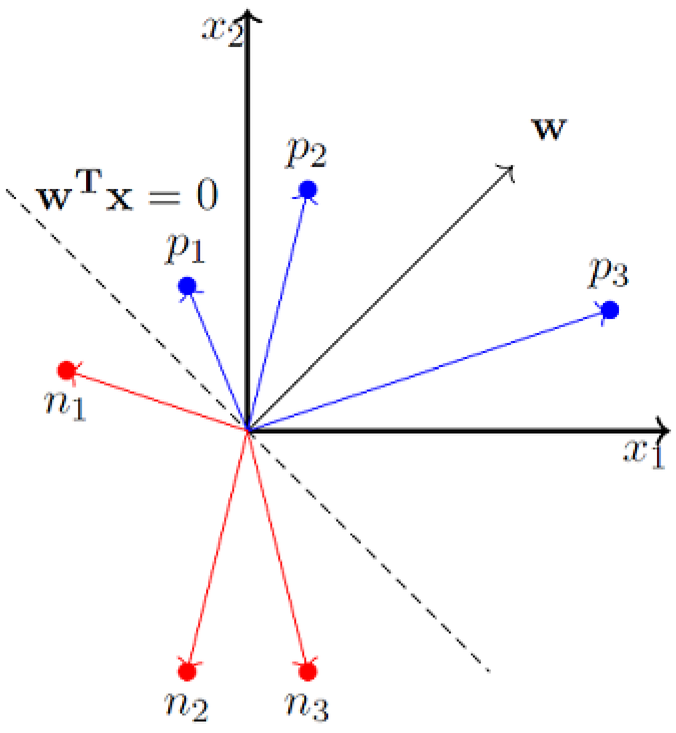
Answer: The angle between **w** and **x** should be less than 90 because the cosine of the angle is proportional to the dot product.

$$cos\alpha = \frac{\mathbf{w}^T\mathbf{x}}{||\mathbf{w}||||\mathbf{x}||} \quad \Bigg| \quad cos\alpha \propto \mathbf{w}^T\mathbf{x}$$

$$\text{So if } \mathbf{w}^T\mathbf{x} > 0 \quad \Rightarrow cos\alpha > 0 \quad \Rightarrow \alpha < 90$$

$$\text{Similarly, if } \mathbf{w}^T\mathbf{x} < 0 \quad \Rightarrow cos\alpha < 0 \quad \Rightarrow \alpha > 90$$

So whatever the **w** vector may be, as long as it makes an angle less than 90 degrees with the positive example data vectors (**x** E *P*) and an angle more than 90 degrees with the negative example data vectors (**x** E *N*), we are cool. So ideally, it should look something like this:



x_0 is always 1 so we ignore it for now.

So we now strongly believe that the angle between **w** and **x** should be less than 90 when **x** belongs to *P* class and the angle between them should be more than 90 when **x** belongs to *N* class. Pause and convince yourself that the above statements are true and you indeed believe them. Here's why the update works:

(α_{new}) when $\mathbf{w_{new}} = \mathbf{w} + \mathbf{x}$

$$\begin{aligned} cos(\alpha_{new}) &\propto \mathbf{w_{new}}^T \mathbf{x} \\ &\propto (\mathbf{w} + \mathbf{x})^T \mathbf{x} \\ &\propto \mathbf{w}^T\mathbf{x} + \mathbf{x}^T\mathbf{x} \\ &\propto cos\alpha + \mathbf{x}^T\mathbf{x} \\ cos(\alpha_{new}) &> cos\alpha \end{aligned}$$

(α_{new}) when $\mathbf{w_{new}} = \mathbf{w} - \mathbf{x}$

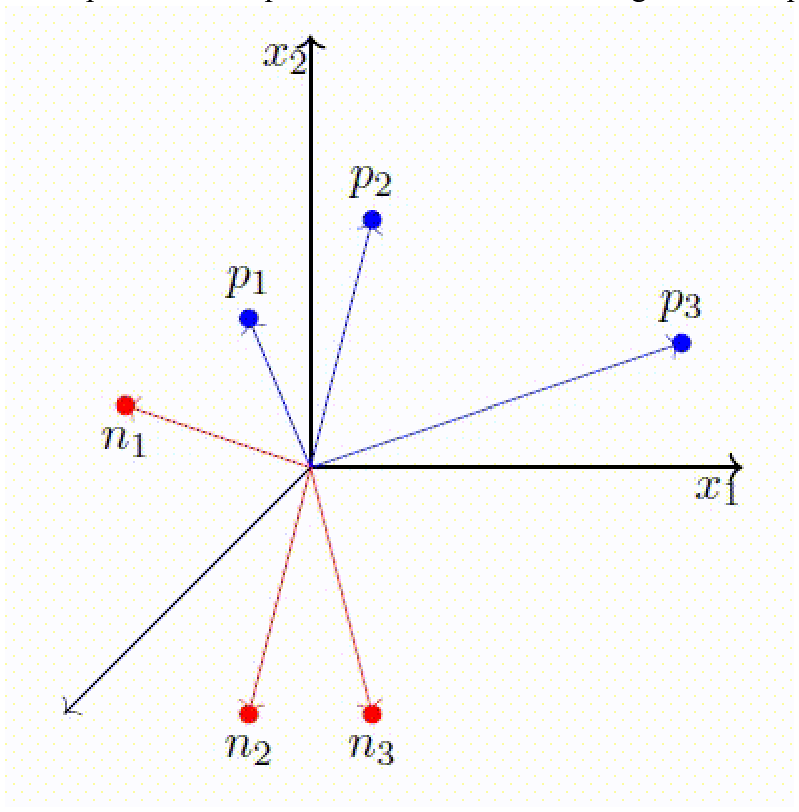
$$\begin{aligned} cos(\alpha_{new}) &\propto \mathbf{w_{new}}^T \mathbf{x} \\ &\propto (\mathbf{w} - \mathbf{x})^T \mathbf{x} \\ &\propto \mathbf{w}^T\mathbf{x} - \mathbf{x}^T\mathbf{x} \\ &\propto cos\alpha - \mathbf{x}^T\mathbf{x} \\ cos(\alpha_{new}) &< cos\alpha \end{aligned}$$

Now this is slightly inaccurate but it is okay to get the intuition.

So when we are adding **x** to **w**, which we do when x belongs to P and **w.x** < 0 (Case 1), we are essentially **increasing the *cos(alpha)*** value, which means, we are **decreasing**

the *alpha* value, the angle between **w** and **x**, **which is what we desire**. And the similar intuition works for the case when **x** belongs to *N* and **w.x** ≥ 0 (Case 2).

Here's a toy simulation of how we might up end up learning **w** that makes an angle less than 90 for positive examples and more than 90 for negative examples.



We start with a random vector **w**.

<u>CONCLUSION:</u> We have studied and implemented perceptron learning algorithm for classification problem.

<u>ASSIGNMENT QUESTION</u>

Q1. What do you mean by Perceptron?

Q2. What are the different types of Perceptrons?

Q3. What is the use of the Loss functions?

Q4. What is Perceptron Learning Law?

Q5. How Perceptron Learning Law works? Explain with Example.

ASSIGNMENT 5

TITLE: BIDIRECTIONAL ASSOCIATIVE MEMORY

PROBLEM STATEMENT: -

Write a python Program for Bidirectional Associative Memory with two pairs of vectors.

OBJECTIVE:

1. To learn types of associative memory
2. To implement the Bidirectional Associative Memory with two pairs of vectors

PREREQUISITE: -

1) Basic of Python Programming
2) Concept of Artificial Neural Network
3) Bidirectional Associative Memory

THEORY:

Bidirectional Associative Memory (BAM) is a supervised learning model in Artificial Neural Network. This is ***hetero-associative memory***, for an input pattern, it returns another pattern which is potentially of a different size. This phenomenon is very similar to the human brain. Human memory is necessarily associative. It uses a chain of mental associations to recover a lost memory like associations of faces with names, in exam questions with answers, etc.

In such memory associations for one type of object with another, a Recurrent Neural Network (RNN) is needed to receive a pattern of one set of neurons as an input and generate a related, but different, output pattern of another set of neurons.

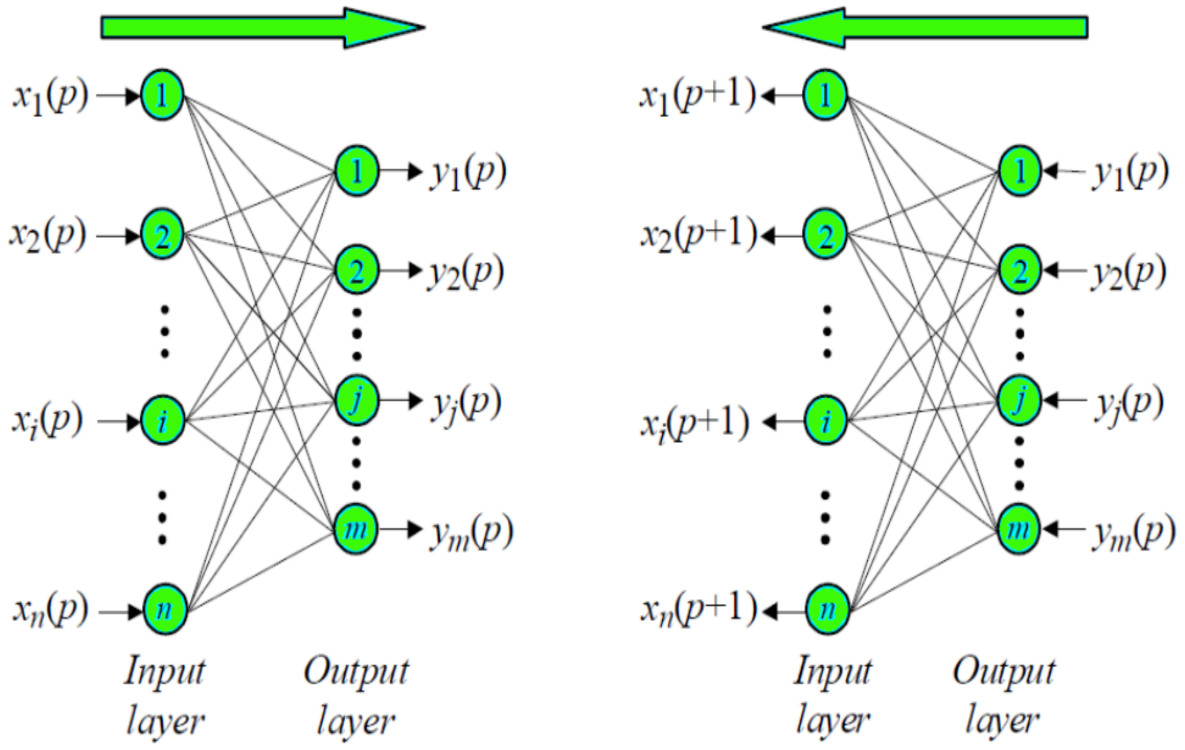
Why BAM is required?

The main objective to introduce such a network model is to store hetero-associative pattern pairs.

This is used to retrieve a pattern given a noisy or incomplete pattern.

BAM Architecture:

When BAM accepts an input of ***n***-dimensional vector ***X*** from set ***A*** then the model recalls ***m***-dimensional vector ***Y*** from set ***B***. Similarly when ***Y*** is treated as input, the BAM recalls ***X***.



(*a*) Forward direction. (*b*) Backward direction.

1. **Storage (Learning):** In this learning step of BAM, weight matrix is calculated between M pairs of patterns (fundamental memories) are stored in the synaptic weights of the network following the equation

$$W = \sum_{m=1}^{M} X_m Y_m^T$$

2. **Testing:** We have to check that the BAM recalls perfectly

Y_m

for corresponding

X_m

and recalls

X_m

for corresponding

Y_m

. Using,

$$Y_m = \text{sign}\left(W^T X_m\right), \quad m = 1.2, \ldots, M$$
$$X_m = \text{sign}\left(W Y_m\right), \quad m = 1.2, \ldots, M$$

All pairs should be recalled accordingly.

3. **Retrieval:** For an unknown vector ***X*** (a corrupted or incomplete version of a pattern from set ***A*** or ***B***) to the BAM and retrieve a previously stored association:

$$X \neq X_m, \quad m = \mathbf{1}, \mathbf{2}, \ldots, M$$

- ***Initialize the BAM:***

$$X(0) = X, \quad p = 0$$

- ***Calculate the BAM output at iteration*** p :

$$Y(p) = \text{sign}\left[W^T X(p)\right]$$

- ***Update the input vector*** $X(p)$:

$$X(p+1) = \text{sign}[WY(p)]$$

- ***Repeat*** the iteration until convergence, when input and output remain unchanged.

Limitations of BAM:

- **Storage capacity of the BAM:** In the BAM, stored number of associations should not be exceeded the number of neurons in the smaller layer.
- **Incorrect convergence:** Always the closest association may not be produced by BAM.

<u>CONCLUSION:</u> We have studied and implemented Bidirectional Associative Memory with two pairs of vectors in artificial neural network

<u>ASSIGNMENT QUESTION</u>

1. What are the key components of a BAM model?
2. How do you represent vectors in Python, and what data structures would you use to store them for this program?
3. Can you describe the process of initializing and training a BAM model with two pairs of vectors?

4. What mathematical operations are involved in the activation and update functions of the BAM algorithm?
5. Can you discuss the role of the weight matrix in the BAM model and how it is updated during the learning process?
6. How do you handle noise or errors in the input vectors in the context of BAM?

ASSIGNMENT 6

TITLE: FEED-FORWARD NEURAL NETWORK

PROBLEM STATEMENT: -

Write a python program to recognize the number 0, 1, 2, 39. A 5 * 3 matrix forms the numbers. For any valid point it is taken as 1 and invalid point it is taken as 0. The net has to be trained to recognize all the numbers and when the test data is given, the network has to recognize the particular numbers.

Aim: Write a python program to recognize the number 0, 1, 2, 39. A 5 * 3 matrix forms the numbers. For any valid point it is taken as 1 and invalid point it is taken as 0.

Objective: To learn digit recognition in python.

Theory:

Handwritten digit recognition using MNIST dataset is a major project made with the help of Neural Network. It basically detects the scanned images of handwritten digits.

We have taken this a step further where our handwritten digit recognition system not only detects scanned images of handwritten digits but also allows writing digits on the screen with the help of an integrated GUI for recognition.

Approach:

We will approach this project by using a three-layered Neural Network.

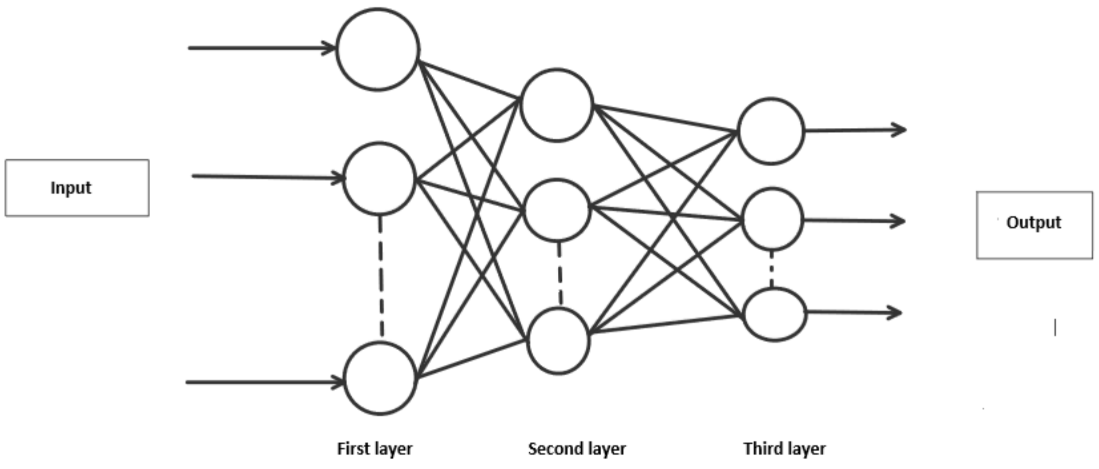
- **The input layer:** It distributes the features of our examples to the next layer for calculation of activations of the next layer.

- **The hidden layer:** They are made of hidden units called activations providing nonlinear ties for the network. A number of hidden layers can vary according to our requirements.
- **The output layer:** The nodes here are called output units. It provides us with the final prediction of the Neural Network on the basis of which final predictions can be made.

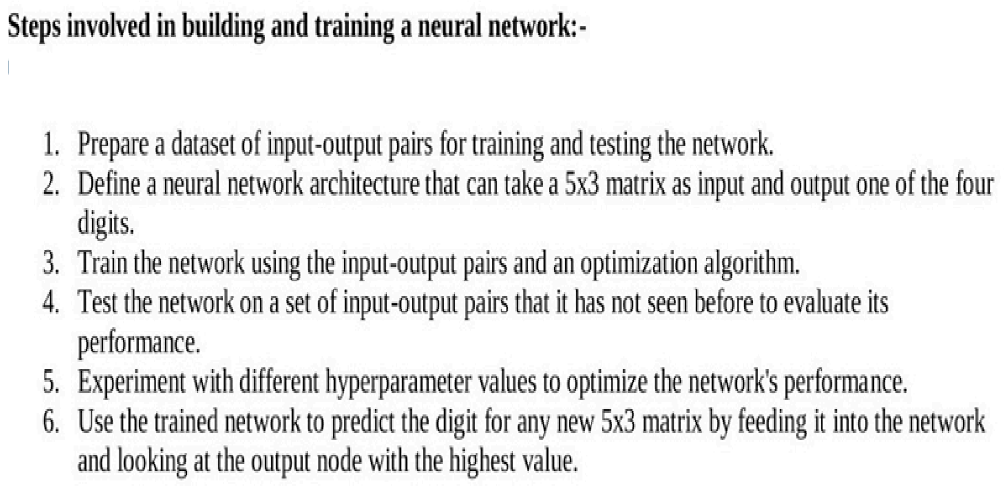
A neural network is a model inspired by how the brain works. It consists of multiple layers having many activations, this activation resembles neurons of our brain. A neural network tries to learn a set of parameters in a set of data which could help to recognize the underlying relationships. Neural networks can adapt to changing input; so the network generates the best possible result without needing to redesign the output criteria.

Methodology:

We have implemented a Neural Network with 1 hidden layer having *100* activation units (excluding bias units). The data is loaded from a *.mat* file, features(X) and labels(y) were extracted. Then features are divided by *255* to rescale them into a range of *[0,1]* to avoid overflow during computation. Data is split up into *60,000* training and *10,000* testing examples. Feedforward is performed with the training set for calculating the hypothesis and then backpropagation is done in order to reduce the error between the layers. The regularization parameter lambda is set to 0.1 to address the problem of overfitting. Optimizer is run for 70 iterations to find the best fit model.



Layers of Neural Network



Steps involved in building and training a neural network:-

1. Prepare a dataset of input-output pairs for training and testing the network.
2. Define a neural network architecture that can take a 5x3 matrix as input and output one of the four digits.
3. Train the network using the input-output pairs and an optimization algorithm.
4. Test the network on a set of input-output pairs that it has not seen before to evaluate its performance.
5. Experiment with different hyperparameter values to optimize the network's performance.
6. Use the trained network to predict the digit for any new 5x3 matrix by feeding it into the network and looking at the output node with the highest value.

Conclusion:

In this way, we have studied how to recognize the number 0, 1, 2, and 39. A 5 * 3 matrix forms the numbers. For any valid point it is taken as 1 and invalid point it is taken as 0.

ASSIGNMENT QUESTION

1. Can you explain the overall approach or algorithm you would use to recognize numbers in a 5x3 matrix?
2. How would you represent the training data for each number? What structure would you use to store this information in Python?
3. What kind of neural network architecture would you choose for this recognition task, and why?
4. How do you initialize the weights in the neural network for training?
5. Describe the process of training the neural network to recognize the numbers 0, 1, 2, and 39 using the provided matrix representations.
6. How do you handle biases in your neural network, and why are they important for this task?

ASSIGNMENT 7

TITLE: FEED-FORWARD NEURAL NETWORK

PROBLEM STATEMENT: -

Implement Artificial Neural Network training process in Python by using Forward Propagation, Back Propagation.

OBJECTIVE:

1. To learn types of neural network
2. To implement the Forward and Back Propagation Feed-forward neural network

PREREQUISITE: -

1) Basic of Python Programming
2) Concept of Artificial Neural Network
3) Forward and Back Propagation Feed-forward neural network

THEORY:

Why Neural Networks?

According to *Universal Approximate Theorem*, Neural Networks can approximate as well as learn and represent any function given a large enough layer and desired error margin. The way neural network learns the true function is by building complex representations on top of simple ones. On each hidden layer, the neural network learns new feature space by first compute the affine (linear) transformations of the given inputs and then apply non-linear function which in turn will be the input of the next layer. This process will continue until we reach the output layer. Therefore, we can define neural network as information flows from inputs through hidden layers towards the output. For a 3-layers neural network, the learned function would be: *f(x) = f_3(f_2(f_1(x)))* where:

- *f_1(x)*: Function learned on first hidden layer
- *f_2(x)*: Function learned on second hidden layer
- *f_3(x)*: Function learned on output layer

Therefore, on each layer we learn different representation that gets more complicated with later hidden layers.Below is an example of a 3-layers neural network (we don't count input layer):

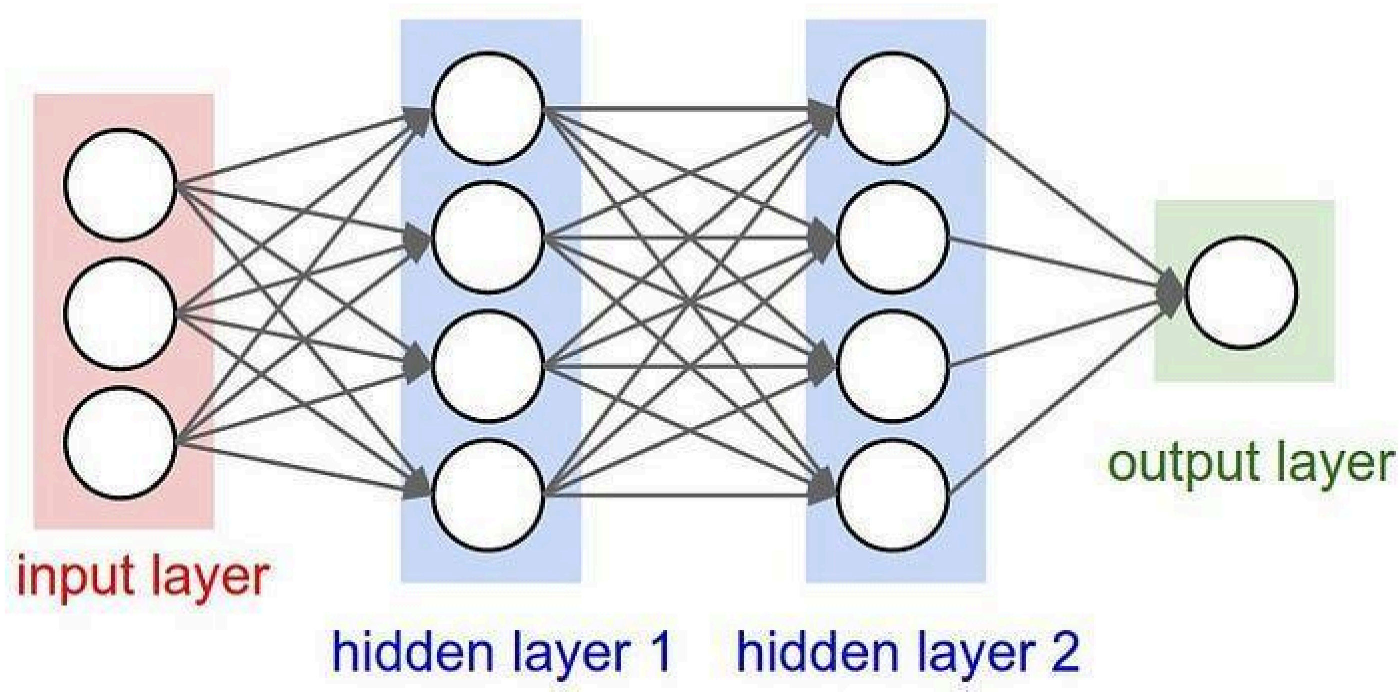


Figure 1: Neural Network with two hidden layers

For example, computers can't understand images directly and don't know what to do with pixels data. However, a neural network can build a simple representation of the image in the early hidden layers that identifies edges. Given the first hidden layer output, it can learn corners and contours. Given the second hidden layer, it can learn parts such as nose. Finally, it can learn the object identity.

Since **truth is never linear** and representation is very critical to the performance of a machine learning algorithm, neural network can help us build very complex models and leave it to the algorithm to learn such representations without worrying about feature engineering that takes practitioners very long time and effort to curate a good representation.

Backpropagation Algorithm

The Backpropagation algorithm is a supervised learning method for multilayer feed-forward networks from the field of Artificial Neural Networks.

Feed-forward neural networks are inspired by the information processing of one or more neural cells, called a neuron. A neuron accepts input signals via its dendrites, which pass the electrical signal down to the cell body. The axon carries the signal out to synapses, which are the connections of a cell's axon to other cell's dendrites.

The principle of the backpropagation approach is to model a given function by modifying internal weightings of input signals to produce an expected output signal. The system is trained using a supervised learning method, where the error between the system's output and a known expected output is presented to the system and used to modify its internal state.

Technically, the backpropagation algorithm is a method for training the weights in a multilayer feed-forward neural network. As such, it requires a network structure to be defined of one or more layers where one layer is fully connected to the next layer. A standard network structure is one input layer, one hidden layer, and one output layer.

Backpropagation can be used for both classification and regression problems, but we will focus on classification in this tutorial.

In classification problems, best results are achieved when the network has one neuron in the output layer for each class value. For example, a 2-class or binary classification problem with the class values of A and B. These expected outputs would have to be transformed into binary vectors with one column for each class value. Such as [1, 0] and [0, 1] for A and B respectively. This is called a one hot encoding.

This tutorial is broken down into 5 parts:

1. Initialize Network.
2. Forward Propagate.

3. Back Propagate Error.
4. Train Network.
5. Predict.

1. Initialize Network

Let’s start with something easy, the creation of a new network ready for training.

Each neuron has a set of weights that need to be maintained. One weight for each input connection and an additional weight for the bias. We will need to store additional properties for a neuron during training, therefore we will use a dictionary to represent each neuron and store properties by names such as ‘**weights**‘for the weights.

A network is organized into layers. The input layer is really just a row from our training dataset. The first real layer is the hidden layer. This is followed by the output layer that has one neuron for each class value.

We will organize layers as arrays of dictionaries and treat the whole network as an array of layers.

It is good practice to initialize the network weights to small random numbers. In this case, will we use random numbers in the range of 0 to 1.

Below is a function named **initialize_network()** that creates a new neural network ready for training. It accepts three parameters, the number of inputs, the number of neurons to have in the hidden layer and the number of outputs.

You can see that for the hidden layer we create **n_hidden** neurons and each neuron in the hidden layer has **n_inputs + 1** weights, one for each input column in a dataset and an additional one for the bias.

You can also see that the output layer that connects to the hidden layer has **n_outputs** neurons, each with **n_hidden + 1** weights. This means that each neuron in the output layer connects to (has a weight for) each neuron in the hidden layer.

2. Forward Propagate

We can calculate an output from a neural network by propagating an input signal through each layer until the output layer outputs its values.

We call this forward-propagation.

It is the technique we will need to generate predictions during training that will need to be corrected, and it is the method we will need after the network is trained to make predictions on new data.
We can break forward propagation down into three parts:

1. Neuron Activation.
2. Neuron Transfer.
3. Forward Propagation.

3. Back Propagate Error

The backpropagation algorithm is named for the way in which weights are trained.
Error is calculated between the expected outputs and the outputs forward propagated from the network. These errors are then propagated backward through the network from the output layer to the hidden layer, assigning blame for the error and updating weights as they go.
The math for backpropagating error is rooted in calculus, but we will remain high level in this section and focus on what is calculated and how rather than why the calculations take this particular form.
This part is broken down into two sections.

1. Transfer Derivative.
2. Error Backpropagation.

4. Train Network

The network is trained using stochastic gradient descent.
This involves multiple iterations of exposing a training dataset to the network and for each row of data forward propagating the inputs, backpropagating the error and updating the network weights.
This part is broken down into two sections:

1. Update Weights.
2. Train Network.

5. Predict

Making predictions with a trained neural network is easy enough.

We have already seen how to forward-propagate an input pattern to get an output. This is all we need to do to make a prediction. We can use the output values themselves directly as the probability of a pattern belonging to each output class.
It may be more useful to turn this output back into a crisp class prediction. We can do this by selecting the class value with the larger probability. This is also called the arg max function.
Below is a function named **predict ()** that implements this procedure. It returns the index in the network output that has the largest probability. It assumes that class values have been converted to integers starting at 0.

Algorithm:

Inputs:

X: a training example input vector

w: the weight matrix for the network

b: the bias vector for the network

Outputs:

a_L: the output of the final layer of the network

Steps:

Set a_0 = X, the input vector for the network.

For each layer l in the network, compute the weighted input z_l for that layer:

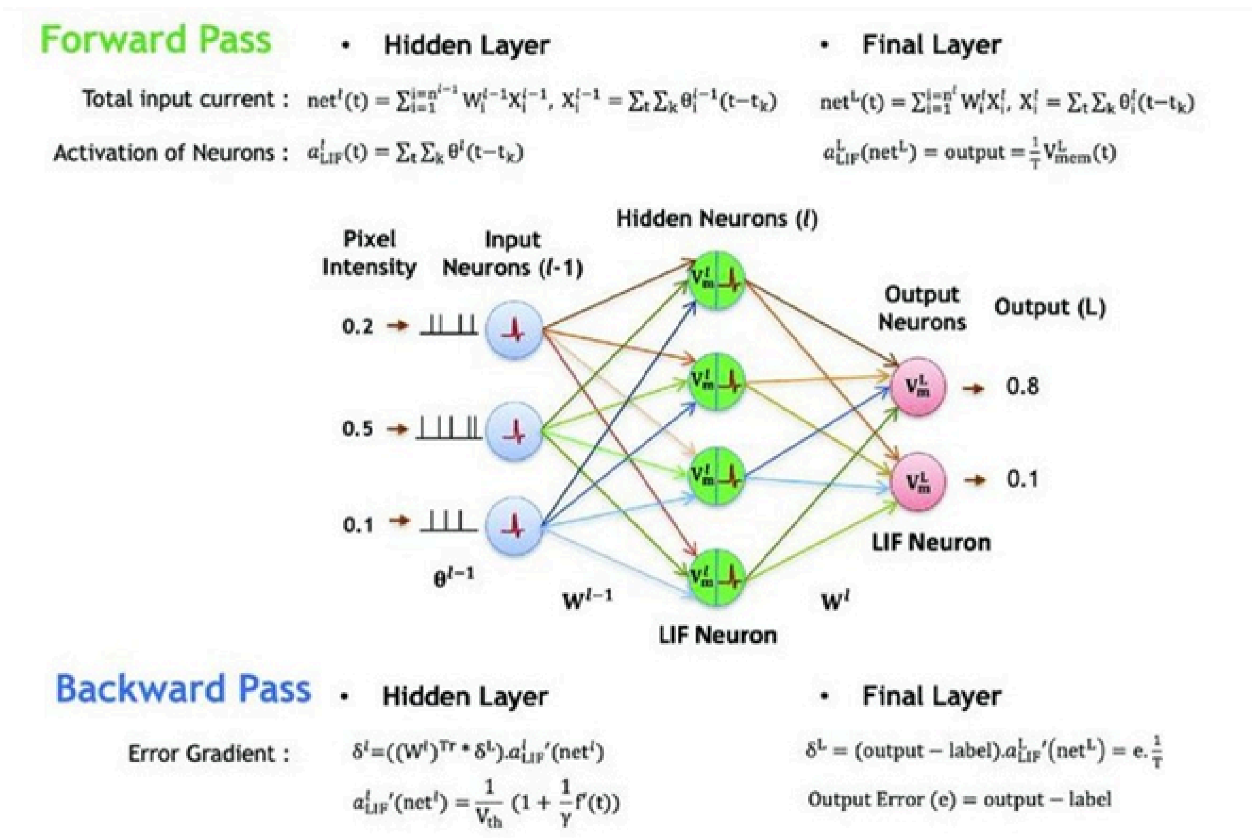
z = w * a-1 + b

Apply the activation function g to the weighted input for each layer to compute the activation a_l:

a = g(z)

Repeat steps 2 and 3 for all layers in the network, up to the final layer L.

Return a_L as the output of the network for input X.


Forward Pass
• Hidden Layer
• Final Layer
Total input current : $net^l(t) = \sum_{i=1}^{i=n^{l-1}} W_i^{l-1} X_i^{l-1}$, $X_i^{l-1} = \sum_t \sum_k \theta_i^{l-1}(t-t_k)$
$net^L(t) = \sum_{i=1}^{i=n^l} W_i^l X_i^l$, $X_i^l = \sum_t \sum_k \theta_i^l(t-t_k)$
Activation of Neurons : $a_{LIF}^l(t) = \sum_t \sum_k \theta^l(t-t_k)$
$a_{LIF}^L(net^L) = output = \frac{1}{T} V_{mem}^L(t)$
Hidden Neurons (l)
Pixel Intensity
Input Neurons (l-1)
Output Neurons
Output (L)
0.2
0.5
0.1
0.8
0.1
θ^{l-1}
W^{l-1}
W^l
LIF Neuron
LIF Neuron
Backward Pass
• Hidden Layer
• Final Layer
Error Gradient : $\delta^l = ((W^l)^{Tr} \cdot \delta^L).a_{LIF}^l{}'(net^l)$
$a_{LIF}^l{}'(net^l) = \frac{1}{V_{th}}(1 + \frac{1}{\gamma} f'(t))$
$\delta^L = (output - label).a_{LIF}^L{}'(net^L) = e.\frac{1}{T}$
Output Error (e) = output – label

CONCLUSION: In this way, we have studied how to implement Artificial Neural Network training process by using Forward Propagation, Back Propagation using python.

ASSIGNMENT QUESTION

Q1. What Is Forward And Backward Propagation?

Q2. How do Forward And Backward Propagation work?

Q3. Write Difference between Forward And Backward Propagation?

Q4. What are steps involved in Forward Propagation?

Q5. What are steps involved in Backward Propagation?

Q6.What is Preactivation and activation in Forward Propagation?

ASSIGNMENT 8

TITLE: FEED-FORWARD NEURAL NETWORK

PROBLEM STATEMENT: -

8. Create a Neural network architecture from scratch in Python and use it to do multi-class classification on any data.

Parameters to be considered while creating the neural network from scratch are specified as:

(1) No of hidden layers : 1 or more

(2) No. of neurons in hidden layer: 100

(3) Non-linearity in the layer : Relu

Use more than 1 neuron in the output layer. Use a suitable threshold value Use appropriate Optimization algorithm

OBJECTIVE:

1. To learn types of neural network
2. To implement the multiclass classification algorithm using ANN

PREREQUISITE: -

1) Basic of Python Programming
2) Basic Concept of Artificial Neural Network

THEORY:

A neural network is a system that learns how to make predictions by following these steps:

1.Taking the input data

2.Making a prediction
3.Comparing the prediction to the desired output
4.Adjusting its internal state to predict correctly the next time

Vectors, layers, and linear regression are some of the building blocks of neural networks. The data is stored as vectors, and with Python you store these vectors in arrays. Each layer transforms the data that comes from the previous layer. You can think of each layer as a feature engineering step, because each layer extracts some representation of the data that came previously.

One good thing about neural network layers is that the same computations can extract information from any kind of data. This means that it doesn't matter if you're using image data or text data. The process to extract meaningful information and train the deep learning model is the same for both scenarios.
Each layer transforms the data that came from the previous layer by applying some mathematical operations.

The Process to Train a Neural Network
Training a neural network is similar to the process of trial and error. Imagine you're playing darts for the first time. In your first throw, you try to hit the central point of the dartboard. Usually, the first shot is just to get a sense of how the height and speed of your hand affect the result. If you see the dart is higher than the central point, then you adjust your hand to throw it a little lower, and so on.

Notice that you keep assessing the error by observing where the dart landed (step 2). You go on until you finally hit the center of the dartboard.

With neural networks, the process is very similar: you start with some random weights and bias vectors, make a prediction, compare it to the desired output, and adjust the vectors to predict more accurately the next time. The process continues until the difference between the prediction and the correct targets is minimal.

Knowing when to stop the training and what accuracy target to set is an important aspect of training neural networks, mainly because of overfitting and underfitting scenarios.

Vectors and Weights

Working with neural networks consists of doing operations with vectors. You represent the vectors as multidimensional arrays. Vectors are useful in deep learning mainly because of one particular operation: the dot product. The dot product of two vectors tells you how similar they are in terms of direction and is scaled by the magnitude of the two vectors.

The main vectors inside a neural network are the weights and bias vectors. Loosely, what you want your neural network to do is to check if an input is similar to other inputs it's already seen. If the new input is similar to previously seen inputs, then the outputs will also be similar. That's how you get the result of a prediction.

The Linear Regression Model

Regression is used when you need to estimate the relationship between a dependent variable and two or more independent variables. Linear regression is a method applied when you approximate the relationship between the variables as linear. The method dates back to the nineteenth century and is the most popular regression method.

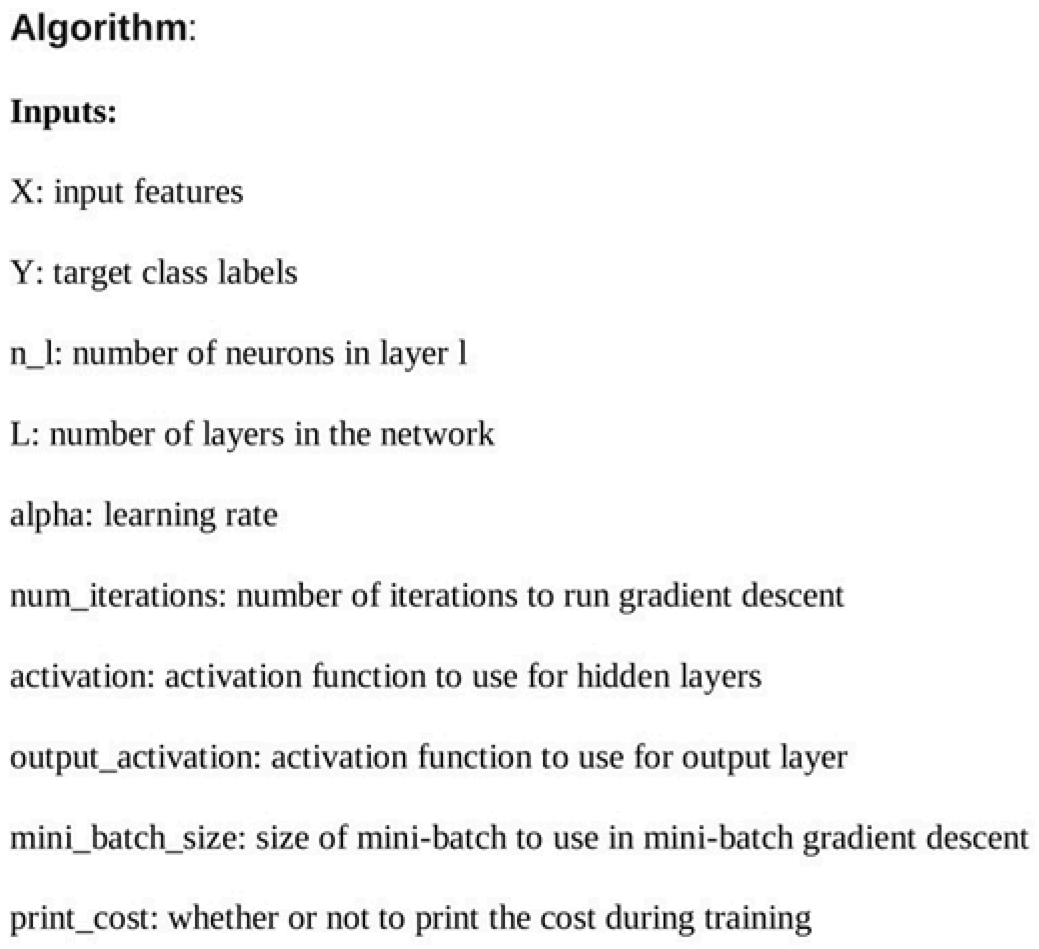
Note: A linear relationship is one where there's a direct relationship between an independent variable and a dependent variable.

By modeling the relationship between the variables as linear, you can express the dependent variable as a weighted sum of the independent variables. So, each independent variable will be multiplied by a vector called weight. Besides the weights and the independent variables, you also add another vector: the bias. It sets the result when all the other independent variables are equal to zero.

As a real-world example of how to build a linear regression model, imagine you want to train a model to predict the price of houses based on the area and how old the house is. You decide to model this relationship using linear regression. The following code block shows how you can write a linear regression model for the stated problem in pseudocode:

price = (weights_area * area) + (weights_age * age) + bias

In the above example, there are two weights: weights_area and weights_age. The training process consists of adjusting the weights and the bias so the model can predict the correct price value. To accomplish that, you'll need to compute the prediction error and update the weights accordingly.



Algorithm:

Inputs:

X: input features

Y: target class labels

n_l: number of neurons in layer l

L: number of layers in the network

alpha: learning rate

num_iterations: number of iterations to run gradient descent

activation: activation function to use for hidden layers

output_activation: activation function to use for output layer

mini_batch_size: size of mini-batch to use in mini-batch gradient descent

print_cost: whether or not to print the cost during training

Outputs:

parameters: a dictionary of learned parameters

Steps:

Initialize the parameters for the neural network:

Use random initialization for weight matrices W^[l] and bias vectors b^[l] for all layers l in range(1, L)

Perform mini-batch gradient descent to train the network:

For each iteration of gradient descent:

Randomly shuffle the training examples X and corresponding labels Y

Split the data into mini-batches of size mini_batch_size

For each mini-batch:

Perform forward propagation to compute the activations for each layer of the network

Compute the cost using the output activations and the true labels Y

Use backpropagation to compute the gradients of the cost with respect to the parameters for each layer of the network

Update the parameters using the gradients and the learning rate alpha

Optionally, print the cost after each iteration

Return the learned

CONCLUSION: We have studied and implemented multiclass classification algorithm using ANN

ASSIGNMENT QUESTION

Q1. How to choose the number of hidden layers and nodes in a feedforward neural network?