K. K. Wagh Institute of Engineering Education and Research, Nashik.

Department of Artificial Intelligence & Data Science

LABORATORY MANUAL

2023-2024

Software Laboratory-II

TE AIDS (2019 Course)

SEMESTER-II

Course Code: 317533

TEACHING SCHEME:

PR: 04 Hours/Week

EXAMINATION SCHEME

Term Work : 25 Marks

Practical(PR): 25 Marks

Name of the Faculty

Prof. P. K. Shinde

K. K. Wagh Institute of Engineering Education and Research, Nashik.

Department of Artificial Intelligence & Data Science

TE(AIDS) Software Laboratory-II(317533)

Practical Assignment List

Sr. No	Problem Statement	Mapping with CO	Due Date
	Group A		
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 recognize even and odd numbers. Given numbers are in ASCII from 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	Implement Artificial Neural Network training process in Python by using Forward Propagation, Back Propagation.		
6	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 Optimization algorithm		
GROUP B			
7	Write a python program to illustrate ART neural network.		
8	Write a python program for creating a Back Propagation Feed-forward neural		
9	Write a python program to design a Hopfield Network which stores 4 vectors		
GROUP C			
10	How to Train a Neural Network with TensorFlow/Pytorch and evaluation of logistic regression using tensorflow.		
11	TensorFlow/Pytorch implementation of CNN.		
12	MNIST Handwritten Character Detection using PyTorch, Keras and Tensorflow.		

K. K. Wagh Institute of Engineering Education and Research, Nashik.

Department of Artificial Intelligence & Data Science

TE(AIDS) Software Laboratory-II(317533)

Course Objectives & Outcomes

COURSE OBJECTIVES:

On completion of the course, learner will be able to—

- To understand basic techniques and strategies of learning algorithms
- To understand various artificial neural network models
- To make use of tools to solve the practical problems in real field using PatternRecognition, Classification and Optimization.

COURSE OUTCOMES:

On completion of the course, learner will be able to—

CO1: Model artificial Neural Network, and to analyze ANN learning, and its applications CO2: Perform Pattern Recognition, Linear classification.
CO3: Develop different single layer/multiple layer Perceptron learning algorithms CO4: Design and develop applications using neural networks.

K. K. Wagh Institute of Engineering Education and Research, Nashik.

Department of Artificial Intelligence & Data Science

TE(AIDS) Software Laboratory-II(317533)

CO-PO MAPPING

@The CO-PO mapping table

PO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO1	2	2		2								2
CO2	1	2		2								2
CO3	2	2	2									2
CO4	2	2	2	2								2

EXPERIMENT NO. 1 (Group A)

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

Outcome: At end of this experiment, student will be able to Write a Python program to plot a few activation functions that are being used in neural networks.

Software Requirement: Ubuntu OS,Python Editor(Pyhton Interpreter)

Theory:

In the process of building a neural network, one of the choices you get to make is what Activation Function to use in the hidden layer as well as at the output layer of the network.

Elements of a Neural Network

Input Layer: This layer accepts input features. It provides information from the outside world to the network, no computation is performed at this layer, nodes here just pass on the information(features) to the hidden layer.

Hidden Layer: Nodes of this layer are not exposed to the outer world, they are part of the abstraction provided by any neural network. The hidden layer performs all sorts of computation on the features entered through the input layer and transfers the result to the output layer.

Output Layer: This layer bring up the information learned by the network to the outer world.

What is an activation function and why use them?

The activation function decides whether a neuron should be activated or not by calculating the weighted sum and further adding bias to it. The purpose of the activation function is to introduce non-linearity into the output of a neuron.

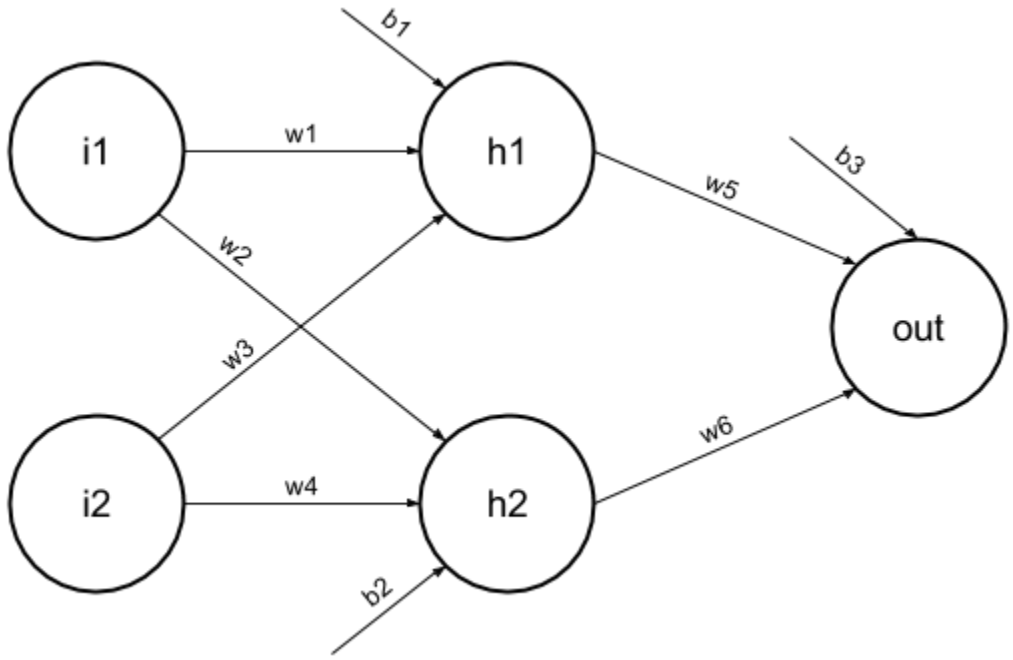
Explanation: We know, the neural network has neurons that work in correspondence with weight, bias, and their respective activation function. In a neural network, we would update the weights and biases of the neurons on the basis of the error at the output. This process is known as back-propagation. Activation functions make the back-propagation possible since the gradients are supplied along with the error to update the weights and biases.

Why do we need Non-linear activation function?

A neural network without an activation function is essentially just a linear regression model. The activation function does the non-linear transformation to the input making it capable to learn and perform more complex tasks.

Mathematical proof

Suppose we have a Neural net like this :-



z(1) = W(1)X + b(1) a(1)

Here,

- z(1) is the vectorized output of layer 1
- W(1) be the vectorized weights assigned to neurons of hidden layer i.e. w1, w2, w3 and w4
- X be the vectorized input features i.e. i1 and i2
- b is the vectorized bias assigned to neurons in hidden layer i.e. b1 and b2
- a(1) is the vectorized form of any linear function.

(Note: We are not considering activation function here) Layer 2 i.e. output layer :-
Note : Input for layer 2 is output from layer 1 z(2) = W(2)a(1) + b(2)
a(2) = z(2)

Calculation at Output layer

z(2) = (W(2) * [W(1)X + b(1)]) + b(2)

z(2) = [W(2) * W(1)] * X + [W(2)*b(1) + b(2)]

Let,

[W(2) * W(1)] = W

[W(2)*b(1) + b(2)] = b

Final output : z(2) = W*X + b which is again a linear function

This observation results again in a linear function even after applying a hidden layer, hence we can conclude that, doesn't matter how many hidden layer we attach in neural net, all layers will behave same way because the composition of two linear function is a linear function itself. Neuron can not learn with just a linear function

attached to it. A non-linear activation function will let it learn as per the difference w.r.t error. Hence we need an activation function.
Variants of Activation Function Linear Function

- Equation : Linear function has the equation similar to as of a straight line i.e. y = x

- No matter how many layers we have, if all are linear in nature, the final activation function of last layer is nothing but just a linear function of the input of first layer.

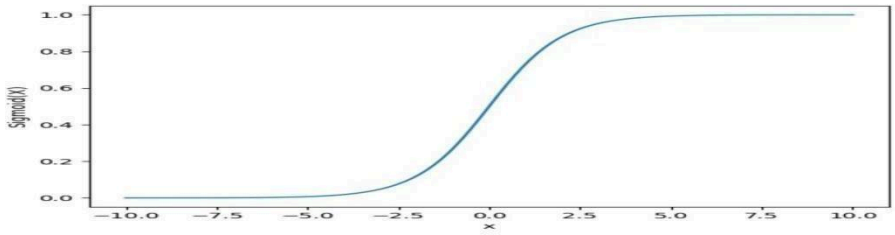
- Range : -inf to +inf

- Uses : Linear activation function is used at just one place i.e. output layer.

- Issues : If we will differentiate linear function to bring non-linearity, result will no more depend on input "x" and function will become constant, it won't introduce any ground-breaking behavior to our algorithm.

For example : Calculation of price of a house is a regression problem. House price may have any big/small value, so we can apply linear activation at output layer. Even in this case neural net must have any non-linear function at hidden layers.

Sigmoid Function



It is a function which is plotted as 'S' shaped graph.

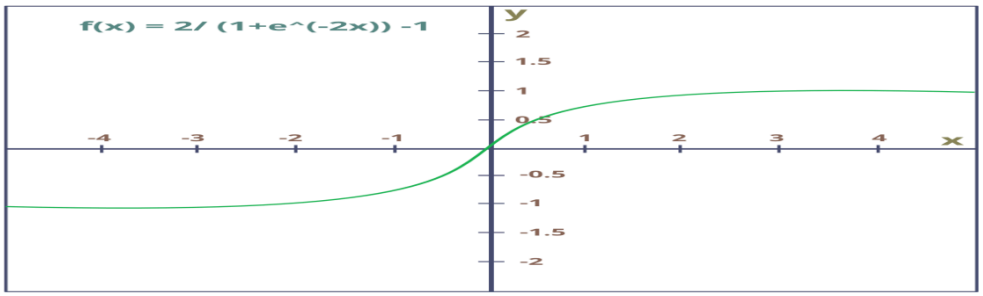
- Equation : A = 1/(1 + e-x)

- 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

- Uses : 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.

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.

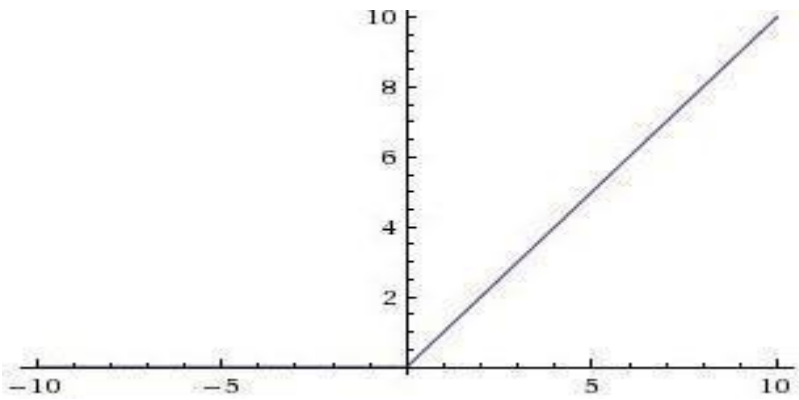
- Equation :-

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

- Value Range :- -1 to +1
- Nature :- non-linear

- Uses: - Usually used in hidden layers of a neural network as its 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.

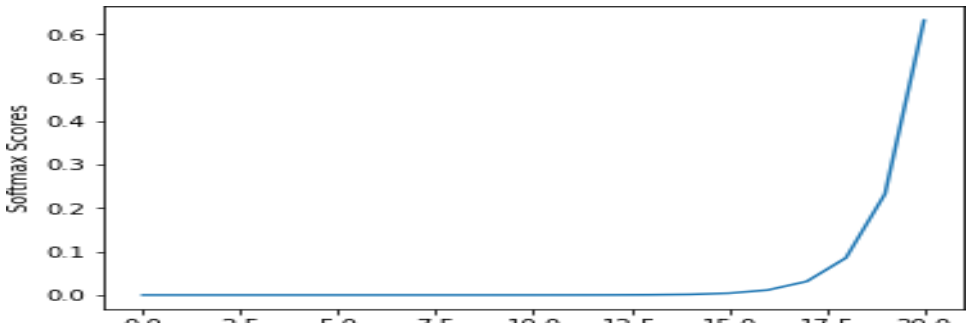
- Equation :- A(x) = max(0,x). It gives an output x if x is positive and 0 otherwise.

- Value Range :- [0, inf)

- Nature :- non-linear, which means we can easily backpropagate 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.
In simple words, RELU learns much faster than sigmoid and Tanh function.

Softmax Function



`The softmax function is also a type of sigmoid function but is handy when we are trying to handle multi- class classification problems.

- Nature :- non-linear

- Uses :- Usually used when trying to handle multiple classes. the softmax function was commonly found in the output layer of image classification problems.The softmax function would squeeze the outputs for each class between 0 and 1 and would also divide by the sum of the outputs.

- Output:- The softmax function is ideally used in the output layer of the classifier where we are actually trying to attain the probabilities to define the class of each input.

- The basic rule of thumb is if you really don't know what activation function to use, then simply use RELU as it is a general activation function in hidden layers and is used in most cases these days.

- If your output is for binary classification then, sigmoid function is very natural choice for output layer.

- If your output is for multi-class classification then, Softmax is very useful to predict the probabilities of each classes.

Conclusion: -

_____

_____

_____

_____

_____

Questions:

Q1. What is the role of the Activation functions in Neural Networks?

Q2. List down the names of some popular Activation Functions used in Neural Networks?

Q3. How to initialize Weights and Biases in Neural Networks? Q4. How are Neural Networks modeled?
Q5. What is an Activation Function

EXPERIMENT NO. 2 (Group A)

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

Outcome: At the end of this experiment, students will be able to Generate ANDNOT function using McCulloch-Pitts neural net by a python program.

Software Requirement: Ubuntu OS,Python Editor(Python Interpreter)

Theory:

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 and one output . The linear threshold gate simply classifies the set of inputs into two different classes. Thus the output is binary. Such a function can be described mathematically using these equations:

W1,w2,w3 are weight values normalized in the range of either or and associated with each input line, is the weighted sum, and is a threshold constant. The function is a linear step function at threshold as shown in figure below. The symbolic representation of the linear threshold gate is shown in figure.

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.

MLP Classifier trains iteratively since at each time step the partial derivative so the loss functions with respect to the model parameters are computed to update the parameters. It can also have a regularization term added to the loss function that shrinks model parameters to prevent over fitting.

This implementation works with data represented as dense numpy arrays or sparse scipy arrays of floating point values. Remove error values in the data set and train the model accordingly via various operations. To perform classification on data set and predict the outcomes, also to plot the outcomes of the experiment to get better understanding of the outcomes.

Firstly, to implement AND function using McCulloch Pitts Neuron you must have some knowledge about McCulloch Pitts Neuron. So, first of all, we have to draw the truth table of AND function. Therefore, In AND function, the Output will be High or 1, if both the inputs are High.

Truth Table of AND NOT Function

x1	x2	Y
1	1	0
1	0	1
0	1	0
0	0	0

After that, we have to assume two weights w1 = w2 = 1 for the inputs.

Conclusion: -

_____

_____

_____

Questions:

Q1. Explain MCP Model?

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

Ans:The MCP (McCulloch-Pitts) model is a simple model of artificial neurons that can be used to build logical functions. The AND NOT function can be generated using the following steps:

1. Start by defining two input neurons (A and B) and one output neuron (C).
2. Set the threshold for the output neuron to 1. This means that the output neuron will only fire if the combined input from the input neurons exceeds the threshold.
3. Connect the input neuron A to the output neuron with a weight of +1 (excitatory connection).
4. Connect the input neuron B to the output neuron with a weight of -1 (inhibitory connection).
5. This means that if A fires and B does not fire, the output neuron will fire (because the combined input will exceed the threshold). However, if both A and B fire, the inhibitory connection from B will cancel out the excitatory connection from A, and the output neuron will not fire.

Therefore, the resulting circuit implements the AND NOT function, as it produces an output only when input A is true and input B is false.

Q3. Discuss briefly Mc Culloch Pitt's artificial neuron model. Q4. Give Mc Culloch Pitt's artificial neuron model

limitations?

EXPERIMENT NO. 3 (Group A)

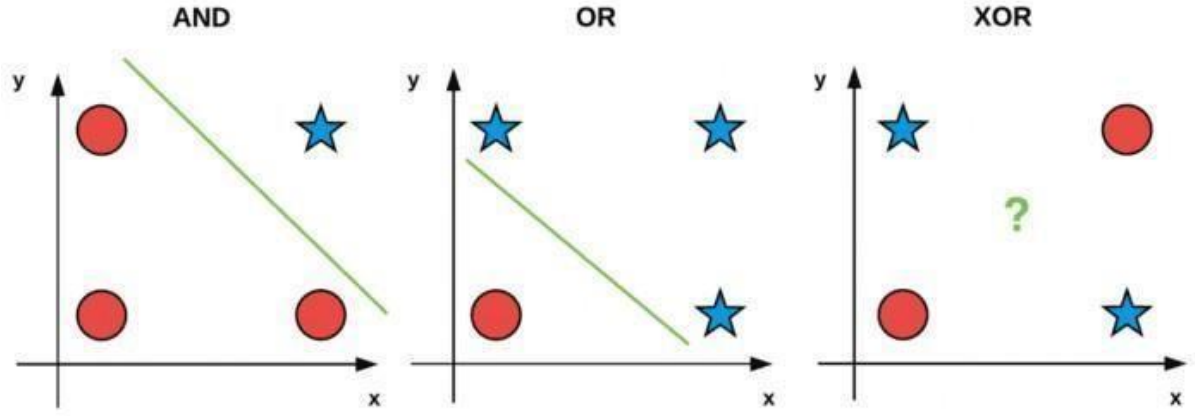
Aim: Write a Python Program using Perceptron Neural Network to recognize even and odd numbers. Given numbers are in ASCII from 0 to 9

Outcome: At end of this experiment, student will be able to Write a Python Program using Perceptron Neural Network to recognize even and odd numbers.

Software Requirement: Open Source Python, Programming tool like Jupyter Notebook, Pycharm, Spyder, Tensorflow.

Theory:

First introduced by Rosenblatt in 1958, The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain is arguably the oldest and most simple of the ANN algorithms. Following this publication, Perceptron-based techniques were all the rage in the neural network community. This paper alone is hugely responsible for the popularity and utility of neural networks today.



But then, in 1969, an "AI Winter" descended on the machine learning community that almost froze out neural networks for good. Minsky and Papert published Perceptrons: an introduction to computational geometry, a book that effectively stagnated research in neural networks for almost a decade — there is much controversy regarding the book (Olazaran, 1996), but the authors did successfully demonstrate that a single layer Perceptron is unable to separate nonlinear data points.

Given that most real-world datasets are naturally nonlinearly separable, this it seemed that the Perceptron, along with the rest of neural network research, might reach an untimely end.

Between the Minsky and Papert publication and the broken promises of neural networks revolutionizing industry, the interest in neural networks dwindled substantially. It wasn't until we started exploring deeper networks (sometimes called multi-layer perceptrons) along with the backpropagation algorithm (Werbos and Rumelhart et al.) that the "AI Winter" in the 1970s ended and neural network research started to heat up again.

All that said, the Perceptron is still a very important algorithm to understand as it sets the stage for more advanced multi-layer networks. We'll start this section with a review of the Perceptron architecture and explain the training procedure (called the delta rule) used to train the Perceptron. We'll also look at the termination criteria of the network (i.e., when the Perceptron should stop training). Finally, we'll implement the Perceptron algorithm in pure Python and use it to study and examine how the network is unable to learn nonlinearly separable datasets.

Perceptron Architecture

Rosenblatt (1958) defined a Perceptron as a system that learns using labeled examples (i.e., supervised learning) of feature vectors (or raw pixel intensities), mapping these inputs to their corresponding output class labels.

In its simplest form, a Perceptron contains N input nodes, one for each entry in the input row of the design matrix, followed by only one layer in the network with just a single node in that layer (Figure 2).

There exist connections and their corresponding weights w1, w2, …, wi from the input xi's to the single output node in the network. This node takes the weighted sum of inputs and applies a step function to determine the output class label. The Perceptron outputs either a 0 or a 1 — 0 for class #1 and 1 for class #2; thus, in its original form, the Perceptron is simply a binary, two- class classifier.

1. Initialize our weight vector $\boldsymbol{w}$ with small random values
2. Until Perceptron converges:
(a) Loop over each feature vector x_j and true class label d_j in our training set D
(b) Take $\boldsymbol{x}$ and pass it through the network, calculating the output value: $y_j = f(\boldsymbol{w}(t) \cdot \boldsymbol{x_j})$
(c) Update the weights $\boldsymbol{w}$: $w_i(t+1) = w_i(t) + \alpha(d_j - y_j)x_{j,i}$ for all features $0 <= i <= n$

Figure 3: The Perceptron algorithm training procedure.

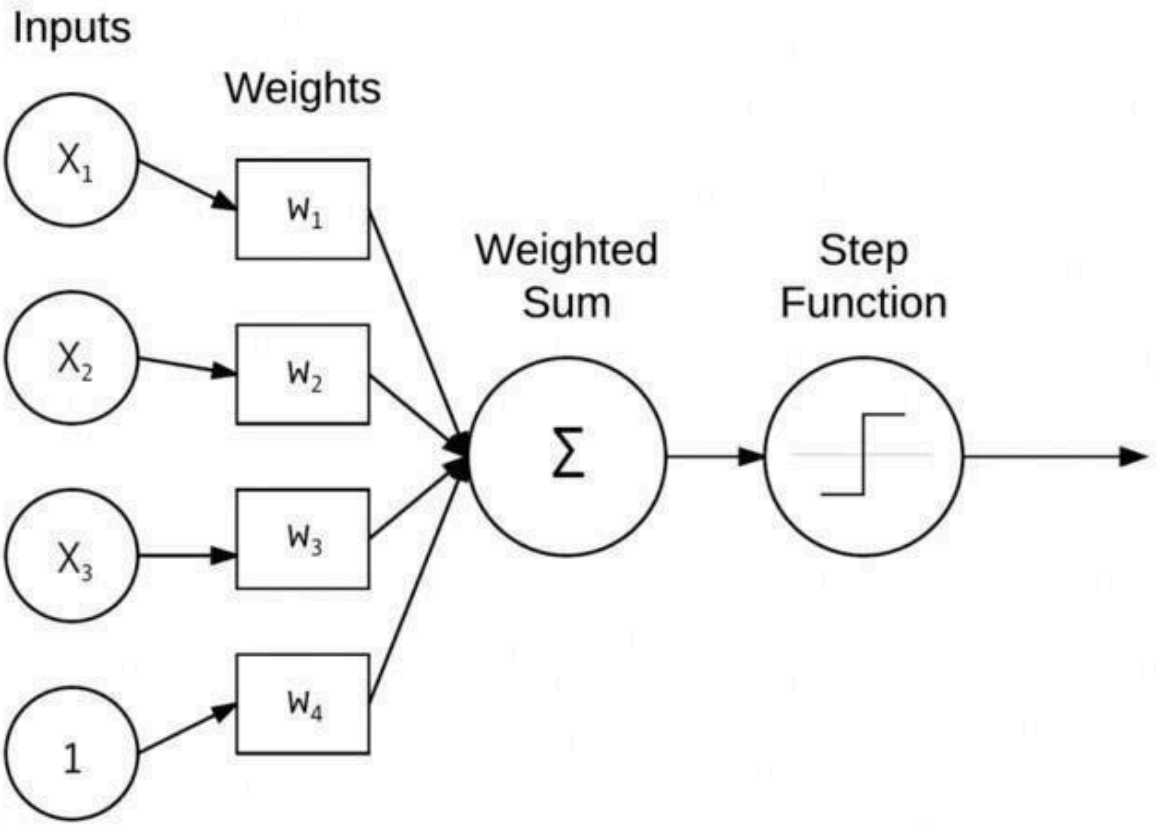


Figure 2: Architecture of the Perceptron network.

NN with two input neurons (one being the variable Number, the other being a bias neuron), nine neurons in the hidden layer and one output neuron using a very simple genetic algorithm: at each epoch, two sets of weights "fight" against each other; the one with the highest error loses and it's replaced by a modified version of the winner.

The script easily solve simple problems like the AND, the OR and the XOR operators but get stuck while trying to categorise odd and even numbers. Right now the best it managed to do is to identify 53 numbers out of 100 and it took several hours. Whether I normalize or not the inputs seems to make no difference.

Conclusion: -

Questions:

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?

EXPERIMENT NO. 4 (Group A)

Aim: With a suitable example demonstrate the perceptron learning law with its decision regions using python. Give the output in graphical form.
Outcome: At the end of this experiment, students will be able to demonstrate the perceptron learning law with its decision regions using python.
Hardware Requirement:

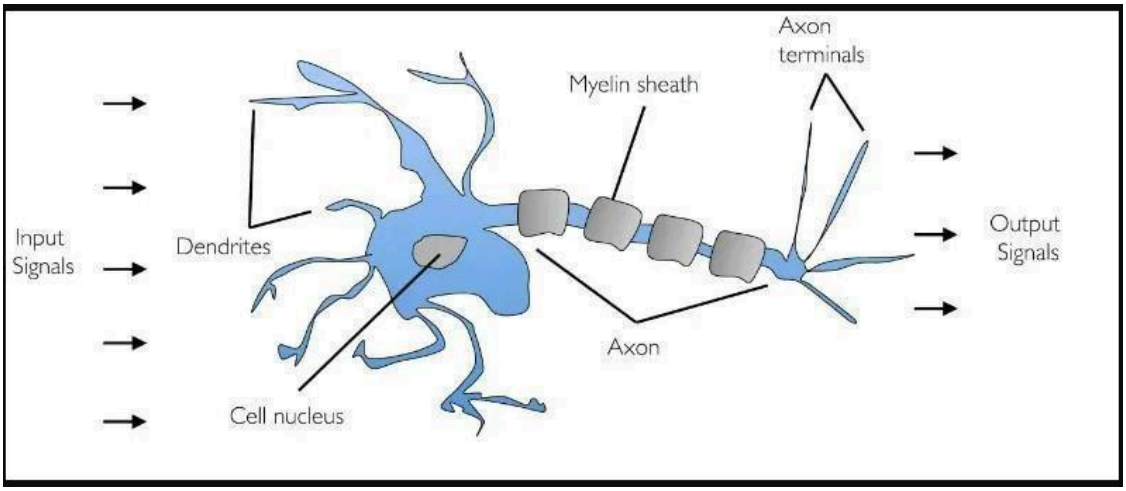
Software Requirement: Ubuntu OS,Python Editor(Pyhton Interpreter)

Theory:

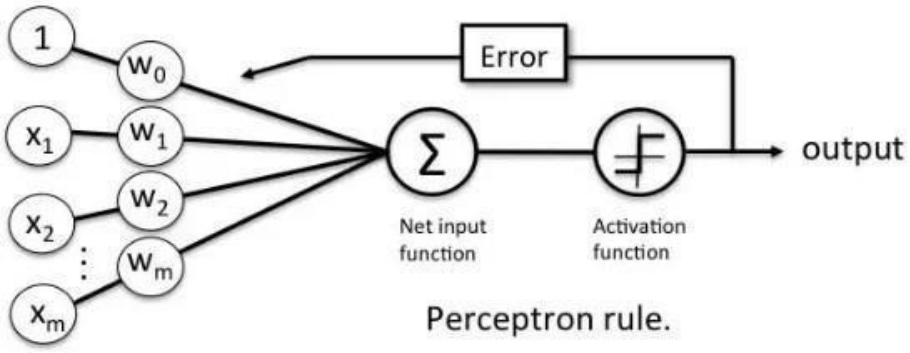
Perceptron is a machine learning algorithm which mimics how a neuron in the brain works. It is also called as **single layer neural network** consisting of a **single neuron.** The output of this neural network is decided based on the outcome of **just one activation function** associated with the single neuron. In perceptron, the **forward propagation** of information happens. Deep neural network consists of one or more perceptrons laid out in two or more layers. Input to different perceptrons in a particular layer will be fed from previous layer by combining them with different weights.

Let's first understand how a neuron works. The diagram below represents a neuron in the brain. The input signals (x1, x2, …) of different strength (observed weights, w1, w2 …) is fed into the neuron cell as **weighted sum** via dendrites. The weighted sum is termed as the **net input**. The net input is processed by the neuron and output signal (observer signal in AXON) is appropriately fired. In case the combined signal strength is not appropriate based on decision function within neuron cell (observe activation function), the neuron does not fire any output signal.

The following is an another view of understanding an artificial neuron, a perceptron, in relation to a biological neuron from the viewpoint of how input and output signals flows:



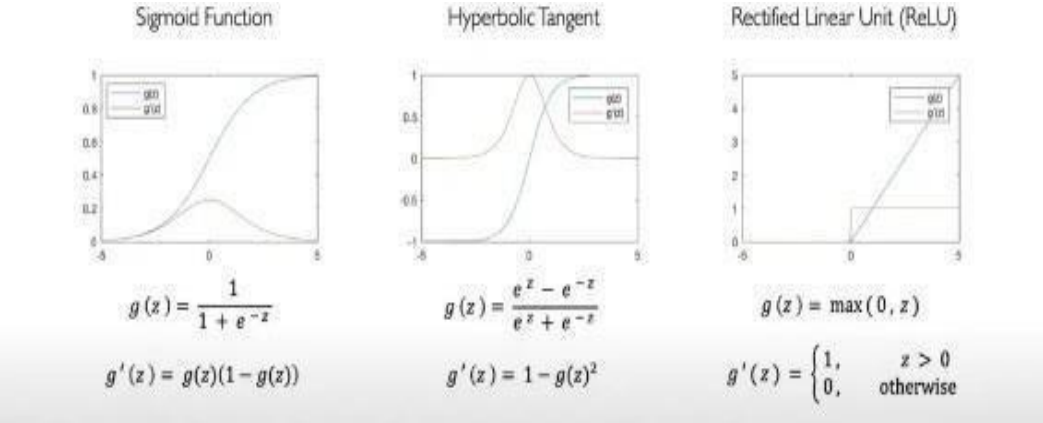
The perceptron when represented as line diagram would look like the following with mathematical notations:



Pay attention to some of the following in relation to what's shown in the above diagram representing a neuron:

- **Step 1 – Input signals weighted and combined as net input**: Weighted sum of input signal reaches to the neuron cell through dendrites. The weighted inputs do represent the fact that different input signals may have different strength, and thus, weighted sum. This weighted sum can as well be termed as **net input** to the neuron cell.
- **Step 2 – Net input fed into activation function:** Weighted The weighted sum of inputs or **net input** is fed as input to what is called as **activation function**. The activation function is a non-linear activation function. The activation functions are of different types such as the following:
- Unit step function
- Sigmoid function (Popular one as it outputs number between 0 and 1 and thus can be used to represent probability)
- Rectilinear (ReLU) function
- Hyperbolic tangent

The diagram below depicts different types of non-linear activation functions



- **Step 3A – Activation function outputs binary signal appropriately**: The activation function processes the net input based on the **unit step** (Heaviside**) function** and outputs the binary signal appropriately as either 1 or 0. The activation function for perceptron can be said to be a unit step function. Recall that the **unit step function**, u(t), outputs the value of 1 when t >= 0 and 0 otherwise. In the case of a shifted unit step function, the

function u(t-a) outputs the value of 1 when t >= a and 0 otherwise.

- **Step 3B – Learning input signal weights based on prediction vs actuals:** A parallel step is a neuron sending the feedback to strengthen the input signal strength (weights) appropriately such that it could create an output signal appropriately that matches the actual value. The feedback is based on the outcome of the activation function which is a unit step function. Weights are updated based on the **gradient descent learning algorithm.**

Conclusion: -

Questions:

Q1. What do you mean by Perceptron?
Q2. What are the different types of Perceptrons?
Q3. What is the use of the Loss functions?

EXPERIMENT NO. 5 (Group A)

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

Outcome: At end of this experiment, student will be able to Implement Artificial Neural Network training process in Python by using Forward Propagation, Back Propagation

Theory:

Backpropagation (short for "backward propagation of errors") is a popular algorithm used in artificial neural networks to train the weights of the network's connections. The goal of backpropagation is to adjust the weights of the network in order to minimize the difference between the predicted output and the actual output.

Backpropagation works by first making a forward pass through the network to obtain a prediction. The difference between the predicted output and the actual output is then calculated, and this difference (the error) is propagated backwards through the network. The algorithm then adjusts the weights of the connections in the network to reduce the error.

The adjustment of the weights is done using a process called gradient descent, which involves calculating the gradient (derivative) of the error with respect to each weight in the network. The weights are then updated by moving in the direction of the negative gradient, which helps to reduce the error.

The backpropagation algorithm is often used in conjunction with an optimization algorithm such as stochastic gradient descent (SGD) to efficiently update the weights of the network. It is a powerful technique that has been used to train neural networks for a wide range of applications, including image recognition, speech recognition, and natural language processing. A Forward propagation, also known as feedforward, is the process of transmitting input data through a neural network in order to obtain an output prediction.

In a neural network, forward propagation involves a series of calculations that are performed on the input data as it passes through the network's layers. Each layer of the network consists of a set of neurons, which are connected to neurons in the previous layer by weighted connections

The input data is fed into the first layer of the network, and each neuron in the layer calculates a weighted sum of its inputs, which is then passed through an activation function to produce an output value. This output is then passed to the next layer of the network as input, and the process is repeated until the output of the final layer is obtained.

The weights of the connections between the neurons are typically initialized randomly and then adjusted during the training process using an algorithm such as backpropagation. The goal of the training process is to adjust the weights so that the network's predictions become more accurate over time.

Forward propagation is a key component of neural network training and inference. It allows the network to process input data and produce output predictions, which can be compared to the actual output values in order to calculate an error. This error can then be used to update the network's weights and improve its accuracy.

The training process consists of the following steps:

Forward Propagation:

Take the inputs, multiply by the weights (just use random numbers as weights) Let Y = WiIi = W1I1+W2I2+W3I3
Pass the result through a sigmoid formula to calculate the neuron's output. The Sigmoid function is used to normalize the result between 0 and 1:
1/1 + e-y

Back Propagation

Calculate the error i.e the difference between the actual output and the expected output. Depending on the error, adjust the weights by multiplying the error with the input and again with the gradient of the Sigmoid curve:
Weight += Error Input Output (1-Output) ,here Output (1-Output) is derivative of sigmoid curve. Note: Repeat the whole process for a few thousand iterations.
Let's code up the whole process in Python. We'll be using the Numpy library to help us with all the calculations on matrices easily. You'd need to install a numpy library on your system to run the code

Conclusion: -

_____

_____

_____

_____

_____

Questions:

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

EXPERIMENT NO. 6 (Group A)

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

Outcome: At end of this experiment, student will be able to study Create a Neural network architecture from scratch in Python and use it to do multi-class classification on any data.

Hardware Requirement:

Software Requirement: Ubuntu OS,Python Editor.

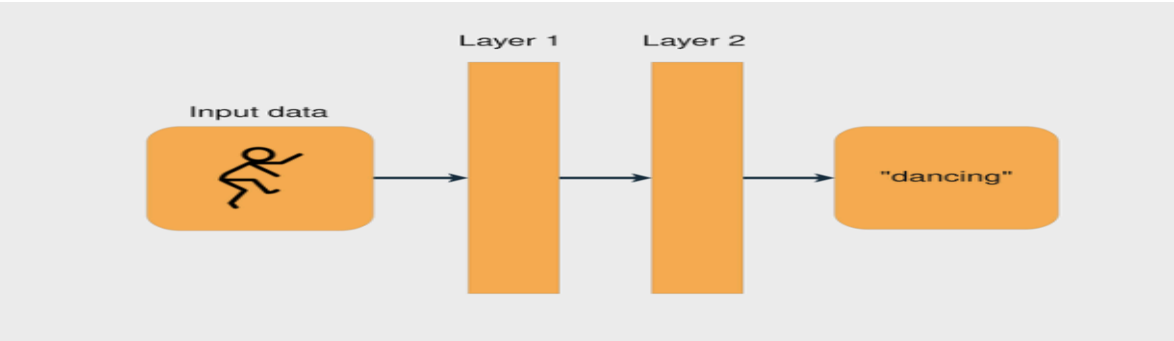
Theory:
A neural network is a system that learns how to make predictions by following these steps:

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 cool 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.

In the image below, you can see an example of a network architecture with two layers:



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.

These are the steps for trying to hit the center of a dartboard:

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.

These are the basics of how the neural network mechanism works. Now it's time to see how to apply these concepts using Python.

Conclusion: -

Questions:

Q1. How to choose the number of hidden layers and nodes in a feedforward neural network?
Q2. Why do we use ReLU in neural networks? Q3. How do we use ReLU in NN?
Q4. What is ReLU in NN?
Q5. What happen when you iniatialize ReLU in Neural Network?

EXPERIMENT NO. 7 (Group B)

Aim: Write a python program to illustrate ART neural network.

Outcome: At end of this experiment, student will be able to illustrate ART neural network.
Hardware Requirement:

Software Requirement: Python IDE

Theory:

Adaptive Resonance Theory (ART)

Adaptive resonance theory is a type of neural network technique developed by Stephen Grossberg and Gail Carpenter in 1987. The basic ART uses unsupervised learning technique. The term "adaptive" and "resonance" used in this suggests that they are open to new learning(i.e. adaptive) without discarding the previous or the old information(i.e. resonance). The ART networks are known to solve the stability-plasticity dilemma i.e., stability refers to their nature of memorizing the learning and plasticity refers to the fact that they are flexible to gain new information. Due to this the nature of ART they are always able to learn new input patterns without forgetting the past. ART networks implement a clustering algorithm. Input is presented to the network and the algorithm checks whether it fits into one of the already stored clusters. If it fits then the input is added to the cluster that matches the most else a new cluster is formed.

Types of Adaptive Resonance Theory(ART)

Carpenter and Grossberg developed different ART architectures as a result of 20 years of research. The ARTs can be classified as follows:

- **ART1 –** It is the simplest and the basic ART architecture. It is capable of clustering binary input values.
- **ART2 –** It is an extension of ART1 that is capable of clustering continuous-valued inputdata.
- **Fuzzy ART –** It is the augmentation of fuzzy logic and ART.
- **ARTMAP –** It is a supervised form of ART learning where one ART learns based on the previous ART module. It is also known as predictive ART.
- **FARTMAP –** This is a supervised ART architecture with Fuzzy logic included.

Basic of Adaptive Resonance Theory (ART) Architecture

The adaptive resonance theory is a type of neural network that is self-organizing and competitive. It can be of both types, the unsupervised ones(ART1, ART2, ART3, etc) or the supervised ones(ARTMAP). Generally, the supervised algorithms are named with the suffix "MAP".But the basic ART model is unsupervised in nature and consists of :

- F1 layer or the comparison field(where the inputs are processed)
- F2 layer or the recognition field (which consists of the clustering units)
- The Reset Module (that acts as a control mechanism)

The **F1 layer** accepts the inputs and performs some processing and transfers it to the F2 layer that best matches with the classification factor.
There exist **two sets of weighted interconnection** for controlling the degree of similarity between the units in the F1 and the F2 layer.
The **F2 layer** is a competitive layer. The cluster unit with the large net input becomes the candidate to learn the input pattern first and the rest F2 units are ignored.
The **reset unit** makes the decision whether or not the cluster unit is allowed to learn the input pattern depending on how similar its top-down weight vector is to the input vector and to the decision. This is called the vigilance test.
Thus we can say that the **vigilance parameter** helps to incorporate new memories or new information. Higher vigilance produces more detailed memories, lower vigilance produces more general memories.

Generally **two types of learning** exists,slow learning and fast learning.In fast learning,weight update during resonance occurs rapidly. It is used in ART1.In slow learning, the
weight change occurs slowly relative to the duration of the learning trial. It is used in ART2.

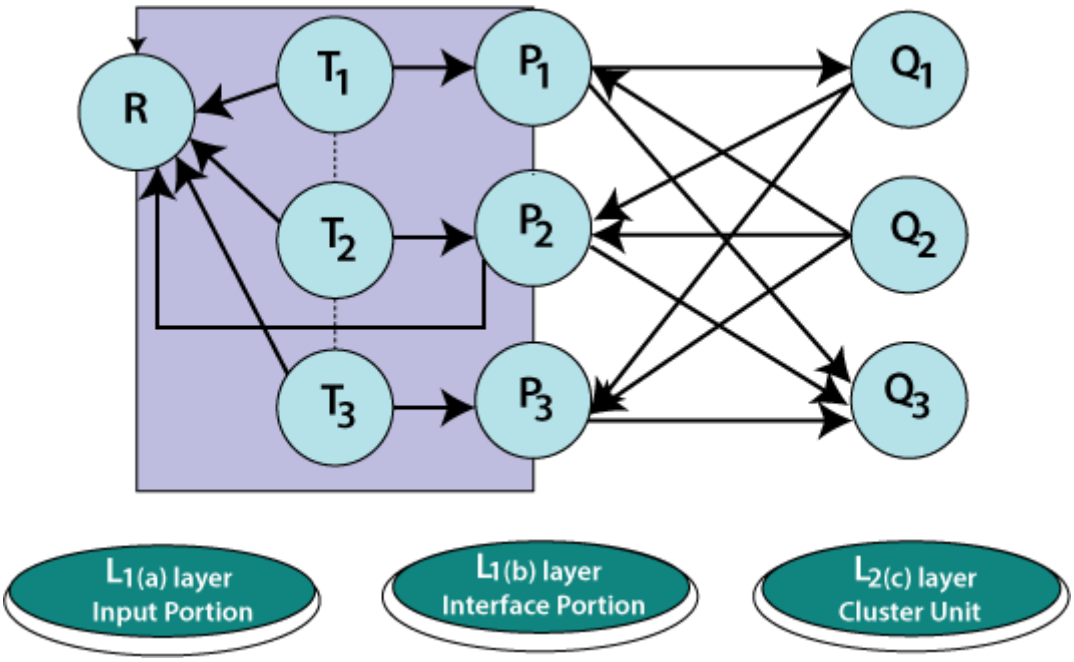
Advantage of Adaptive Resonance Theory (ART)

- It exhibits stability and is not disturbed by a wide variety of inputs provided to its network.
- It can be integrated and used with various other techniques to give more good results.
- It can be used for various fields such as mobile robot control, face recognition, land cover classification, target recognition, medical diagnosis, signature verification,
clustering web users, etc.
- It has got advantages over competitive learning (like bpnn etc). The competitive learning lacks the capability to add new clusters when deemed necessary.
- It does not guarantee stability in forming clusters.

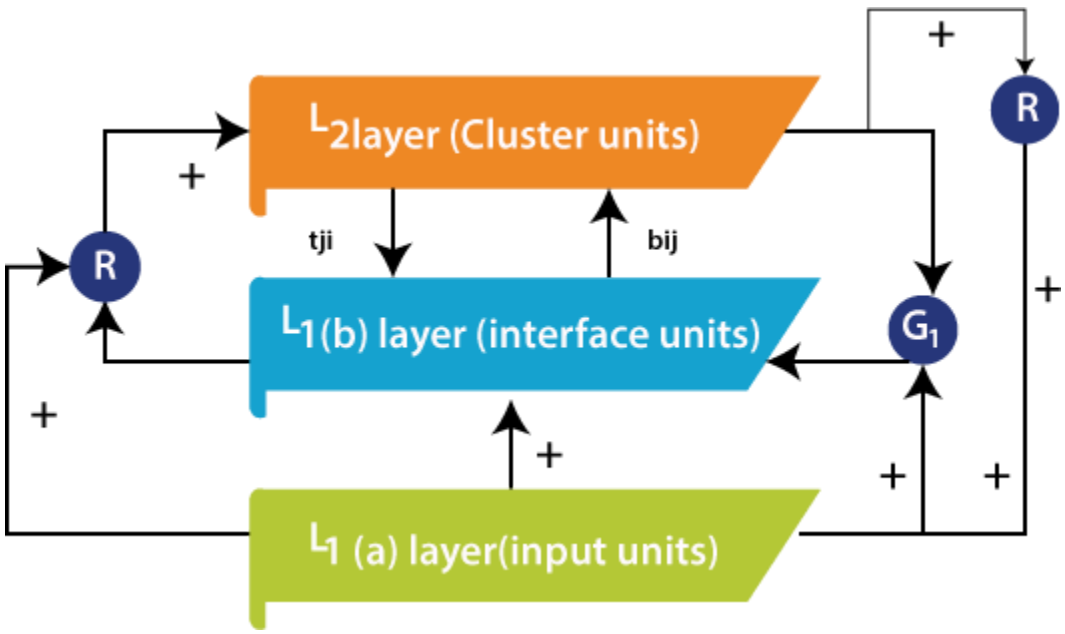
Limitations of Adaptive Resonance Theory

Some ART networks are inconsistent (like the Fuzzy ART and ART1) as they depend uponthe order in which training data, or upon the learning rate.

ART1 is an unsupervised learning model primarily designed for recognizing binary patterns. It comprises an attentional subsystem, an orienting subsystem, a vigilance parameter, and a reset module, as given in the figure given below. The vigilance parameter has a huge effect on the system. High vigilance produces higher detailed memories. The ART1 attentional comprises of two competitive networks, comparison field layer L1 and the recognition field layer L2, two control gains, Gain1 and Gain2, and two short-term memory (STM) stages S1 and S2. Long term memory (LTM) follows somewhere in the range of S1 and S2 multiply the signal in these pathways.



Gains control empowers L1 and L2 to recognize the current stages of the running cycle. STM reset wave prevents active L2 cells when mismatches between bottom-up and top- down signals happen at L1. The comparison layer gets the binary external input passing it to the recognition layer liable for coordinating it to a classification category. This outcome is given back to the comparison layer to find out when the category coordinates the input vector. If there is a match, then a new input vector is read, and the cycle begins once again. If there is a mismatch, then the orienting system is in charge of preventing the previous category from getting a new category match in the recognition layer. The given two gains control the activity of the recognition and the comparison layer, respectively. The reset wave specifically and enduringly prevents active L2 cell until the current is stopped. The offset of the input pattern ends its processing L1 and triggers the offset of Gain2. Gain2 offset causes consistent decay of STM at L2 and thereby prepares L2 to encode the next input pattern without bais.



Conclusion: -

Questions:

Q1. Explain adaptive resonance learning with its applications ?
Q2. Explain the task of feedforward ANN in pattern recognition ?
Q3. What are the advantages or disadvantages of simulated annealing ?
Q4. Discuss use of hopfield networks in associative learning ?
Q5. What is the purpose of ART networks?

EXPERIMENT NO. 8 (Group B)

Aim: Write a python program for creating a Backpropagation Feed-forward neural network.

Outcome: At end of this experiment, student will be able to creating a Back Propagation Feed-forward neural network.

Software Requirement: Python IDE

Theory:

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 anda 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 networkstructure 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 binaryvectors with one column for each class value. Such as [1, 0] and [0, 1] for A and Brespectively. 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.

Conclusion: -

Questions:

Q1. What is the feed-forward back propagation method?
Q2. What are the factors affecting backpropagation networks?
Q3. Which function is used on the back propagation network?
Q4. What is the difference between backpropagation and feed forward?
Q5. What are the characteristics of a backpropagation algorithm?

EXPERIMENT NO. 9 (Group B)

Aim: Write a python program to design a Hopfield Network which stores 4 vectors

Outcome: At end of this experiment, student will be able to design a Hopfield Network.
Hardware Requirement:

Software Requirement: Python IDE

Theory:

The Hopfield Neural Networks, invented by Dr John J. Hopfield consists of one layer of 'n' fully connected recurrent neurons. It is generally used in performing auto association and optimization tasks. It is calculated using a converging interactive process and it generates a different response than our normal neural nets.

Hopfield network is a special kind of neural network whose response is different from other neural networks. It is calculated by converging iterative processes. It has just one layer of neurons relating to the size of the input and output, which must be the same. When such a network recognizes, for example, digits, we present a list of correctly rendered digits to the network. Subsequently, the network can transform a noise input to the related perfect output.

Discrete Hopfield Network: It is a fully interconnected neural network where each unit is connected to every other unit. It behaves in a discrete manner, i.e. it gives finite distinct output, generally of two types:

- Binary (0/1)
- **Bipolar (-1/1)**

The weights associated with this network are symmetric in nature and have the following properties.

Structure & Architecture

- Each neuron has an inverting and a non-inverting output.
- Being fully connected, the output of each neuron is an input to all other neurons but not self.

Fig 1 shows a sample representation of a Discrete Hopfield Neural Network architecture having the following elements.

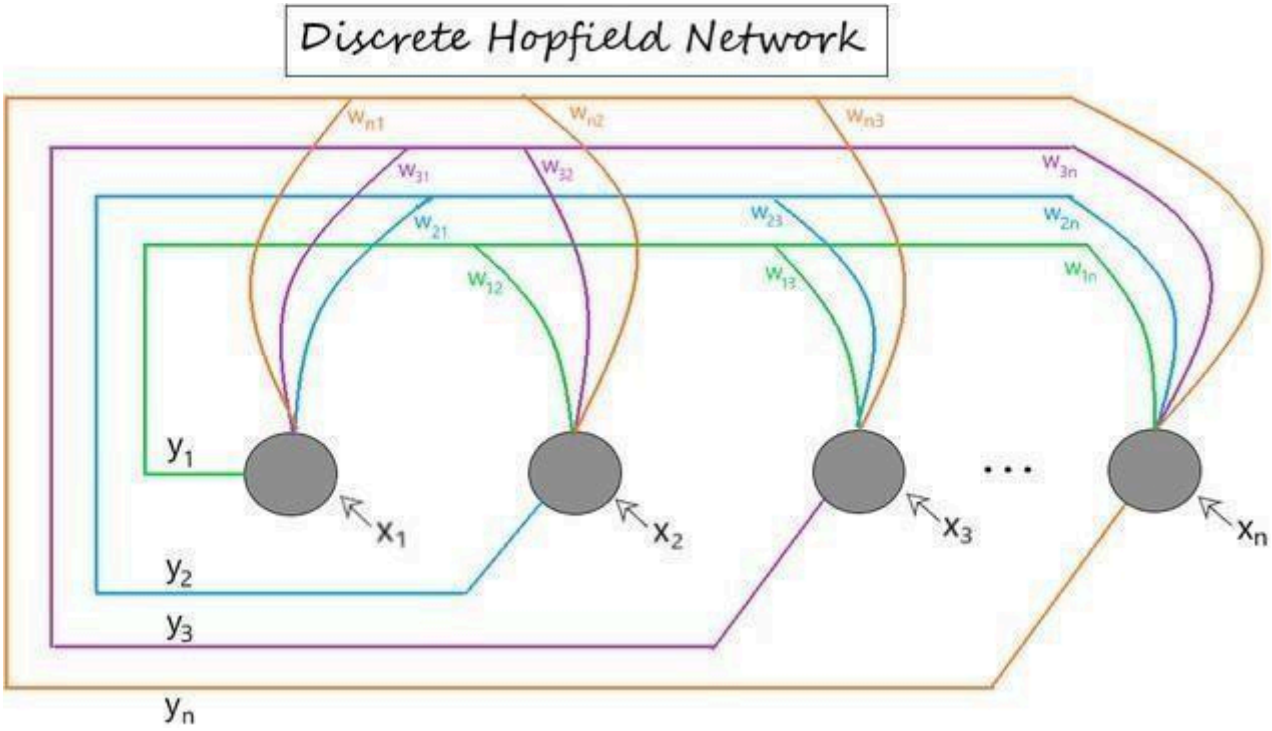


Fig 1: Discrete Hopfield Network Architecture

Training Algorithm

For storing a set of input patterns S(p) [p = 1 to P], where S(p) = S1(p) … Si(p) … Sn(p), theweight matrix is given by:

For binary patterns

w_{ij} = \sum_{p = 1}^{P} [2s_{i}(p) - 1][2s_{j}(p) - 1]\ (w_{ij}\ for\ all\ i\neq j)

For bipolar patterns

w_{ij} = \sum_{p = 1}^{P} [s_{i}(p)s_{j}(p)]\ (where\ w_{ij} = 0\ for\ all\ i= j)(i.e. weights here have no self connection)

Steps Involved

Step 1 - Initialize weights (wij) to store patterns (using training algorithm).

Step 2 - For each input vector yi, perform steps 3-7.

Step 3 - Make initial activators of the network equal to the external input vector x.y_i = x_i : (for\ i = 1\ to\ n)

Step 4 - For each vector yi, perform steps 5-7.

Step 5 - Calculate the total input of the network yin using the equation given below.

y_{in_{i}} = x_i + \sum_{j} [y_jw_{ji}]

Step 6 - Apply activation over the total input to calculate the output as per the equation givenbelow:

y_i = \begin{cases} & 1 \text{ if } y_{in}>\theta_i \\ &y_i \text{ if } y_{in}=\theta_i \\ & 0 \text{ if } y_{in}<\theta_i \end{cases}

(whereθi (threshold) and is normally taken as 0)

Step 7 - Now feedback the obtained output yi to all other units. Thus, the activation vectorsare updated.

Step 8 - Test the network for convergence.

Example:

Suppose we have only two neurons: N = 2

There are two non-trivial choices for connectivities:

w12 = w21 = 1 w12= w21 = -1
Asynchronous updating:

In the first case, there are two attracting fixed points termed as [-1,-1] and [-1,-1]. All orbit converges to one of these. For a second, the fixed points are [-1,1] and [1,-1], andall orbits are joined through one of these. For any fixed point, swapping all the signs gives another fixed point.

Synchronous updating:

In the first and second cases, although there are fixed points, none can be attracted to nearby points, i.e., they are not attracting fixed points. Some orbits oscillate forever.

Energy function evaluation:

Hopfield networks have an energy function that diminishes or is unchanged with asynchronous updating.

For a given state X ∈ {−1, 1} N of the network and for any set of association weights Wij with Wij = wji and wii =0 let,

Hopfield Network

Here, we need to update Xm to X'm and denote the new energy by E' and show that. E'-E = (Xm-X'm) ∑i≠mWmiXi.
Using the above equation, if Xm = Xm' then we have E' = E
If Xm = -1 and Xm' = 1 , then Xm - Xm' = 2 and hm= ∑iWmiXi ? 0 Thus, E' - E ≤ 0
Similarly if Xm =1 and Xm'= -1 then Xm - Xm' = 2 and hm= ∑iWmiXi< 0 Thus, E - E' < 0.

Conclusion: -

Questions:

Q1. Write note on competitive learning ?
Q2. How SOM works?
Q3. What are the issues faced while training in Recurrent Networks?
Q4. Explain the different layers of CNN?
Q5. Explain Hopfield Model?

EXPERIMENT NO. 10 (Group C)

Aim: How to Train a Neural Network with TensorFlow/Pytorch and evaluation of logistic regression using tensorflow.

Outcome: At end of this experiment, student will be able to Train a Neural Network with TensorFlow/Pytorch and evaluation of logistic regression using tensorflow.

Software Requirement: Python IDE

Theory:
What is regression?

For example, if the model that we built should predict discrete or continuous values like a person's age, earnings, years of experience, or need to find out how these values are correlated with the person, it shows that we are facing a regression problem.

What is a neural network?

Just like a human brain, a neural network is a series of algorithms that detect basic patterns ina set of data. The neural network works as a neural network in the human brain. A "neuron" in a neural network is a mathematical function that searches for and classifies patterns according to a specific architecture.

It is possible and important to talk about each of these topics in detail and for a long time, butmy goal in this article is to build a model and work on it together after briefly touching on theimportant points. If you start to write codes with me and get the results by yourself, everything will be more fun and memorable. So let's get our hands dirty.

Become a Full Stack Data Scientist

Transform into an expert and significantly impact the world of data science. First, let's start with **importing some libraries** that we will use at the beginning:
import tensorflow as tf print(tf. version)
import numpy as np import matplotlib.pyplot as plt
We are dealing with a regression problem, and we will **create our dataset:**

One important point in NN is the input shapes and the output shapes. The input shape is the shape of the data that we train the model on, and the output shape is the shape of data that we expect to come out of our model. Here we will use X and aim to predict y, so, X is our input and y is our output.

X. shape,y.shape
>>((74,),(74,))
Here we can see that our tensors have the same shape, but in real life, it may not be that way always, so, we should check and fix that if needed before we build a model. Let's start building our model with TensorFlow. There are 3 typical steps to creating a model inTensorFlow:

- **Creating a model** – connect the layers of the neural network yourself, here we either use Sequential or Functional API, also we may import a previously built model that we call transfer learning.
- **Compiling a model** – at this step, we define how tomeasure a model's performance, which optimizer should be used.

- **Fitting a model** – In this step, we introduce the model to the data and let it find patterns.

We've created our dataset, that is why we can directly start modeling, but first, we need tosplit our train and test set

```
len(X)
>> 74
X_train=X[:60]y_train
=y[:60]

X_test=X[60:]y_test
=y[60:]
len(X_train), len(X_test)
>> (60,14)
```

The best way of getting more insight into our data is by **visualizing** it! So, let's do it!

```
plt.figure(figsize=(12,6))
plt.scatter(X_train, y_train,c='b',label='Trainingdata')plt.scatter(X_test, y_test, c='g', label='Testing data')
plt.legend()
```

Improve the Regression model with neural network

```
tf.random.set_seed(42) model_2 =tf.keras.Sequential([ tf.keras.layers.Dense(1), tf.keras.layers.Dense(1)
])
model_2.compile(loss=tf.keras.losses.mae, optimizer=tf.keras.optimizers.SGD(), metrics=['mae'])
model_2.fit(X_train, y_train, epochs=100, verbose=0)
```

Here we just replicated the first model, and add an extra layer to see how it works?

```
preds_2 =model_2.predict(X_test) plot_preds(predictions=preds_2)
```

Logistic Regression is Classification algorithm commonly used in Machine Learning. It allows categorizing data into discrete classes by learning the relationship from a given set of labeled data. It learns a linear relationship from the given dataset and then introduces a non-linearity in the form of the Sigmoid function.

In case of Logistic regression, the hypothesis is the Sigmoid of a straight line, i.e,h(x) = \sigma(wx + b) where \sigma(z) = \frac{1}{1 + e^{-z}}
Where the vector w represents the Weights and the scalar b represents the Bias of the model.

Conclusion: -

_____

_____

_____

_____

Questions:

Q1. Write short note on Softmax regression?

Q2. What are the deep learning frameworks or tools? Q3**.** What are the applications of deep learning?
Q4. What is the meaning of term weight initialization in neural networks? Q5. What are the essential elements of PyTorch?

EXPERIMENT NO. 11 (Group C)

Aim: TensorFlow/Pytorch implementation of CNN.

Outcome: At end of this experiment, student will be able to implement TensorFlow/Pytorch of CNN

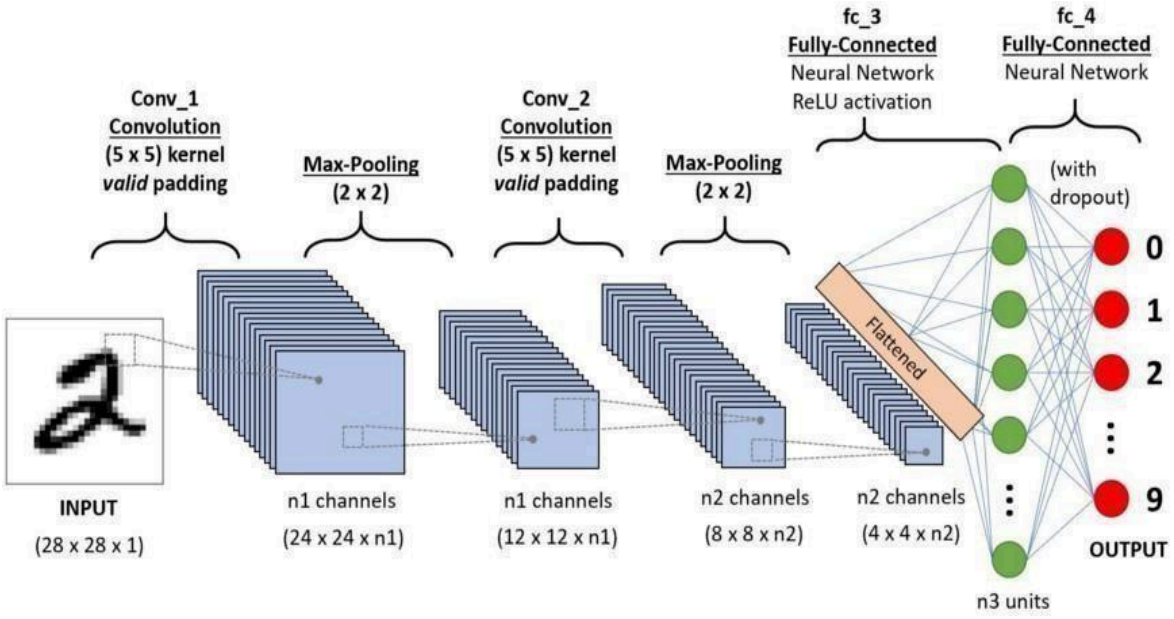
Software Requirement: Python IDE

Theory:

Convolutional neural networks, also called ConvNets, were first introduced in the 1980s by YannLeCun, a computer science researcher who worked in the background. LeCun built on the work of Kunihiko Fukushima, a Japanese scientist, a basic network for image recognition.

The old version of CNN, called LeNet (after LeCun), can see handwritten digits. CNN helps find pin codes from postal. But despite their expertise, ConvNets stayed close to computer vision and artificial intelligence because they faced a major problem: They could not scale much. CNN's require a lot of data and integrate resources to work well for large images.

At the time, this method was only applicable to low-resolution images. Pytorch is a library that can do deep learning operations. We can use this to perform Convolutional neural networks. Convolutional neural networks contain many layers of artificial neurons. Synthetic neurons, complex simulations of biological counterparts, are mathematical functions that calculate the weighted mass of multiple inputs and product value activation.



The above image shows us a CNN model that takes in a digit-like image of 2 and gives us the result of what digit was shown in the image as a number. We will discuss in detail how we get this in this article.

CIFAR-10 is a dataset that has a collection of images of 10 different classes. This dataset is widely used for research purposes to test different machine learning models and especially for computer vision problems. In this article, we will try to build a Neural network model using Pytorch and test it on the CIFAR-10 dataset to check what accuracy of prediction can be obtained.

Importing the PyTorch Library

```
import numpy as np

import pandas as pd

import torch
import torch.nn.functional as F from torchvision
import datasets,transforms from torch
 import non import matplotlib.pyplot as plt
 import numpy as np
 import seaborn as sns #from tqdm.notebook
 import tqdm from tqdm
import tqdm
```

In this step, we import the required libraries. We can see we use NumPy for numerical operations and pandas for data frame operations. The torch library is used to import Pytorch.

Pytorch has an nn component that is used for the abstraction of machine learning operations and functions. This is imported as F. The torchvision library is used so that we can import theCIFAR-10 dataset. This library has many image datasets and is widely used for research. The transforms can be imported so that we can resize the image to equal size for all the images. The tqdm is used so that we can keep track of the progress during training and is used for visualization.

Analyzing the data with PyTorch

```
print("Number of points:",trainData.shape[0]) print("Numberof features:",trainData.shape[1])
print("Features:",trainData.columns.values) print("Number of Unique Values")
for col in trainData: print(col,":",len(trainData[col].unique())) plt.figure(figsize=(12,8))
```

Output:

Number of points: 50000 Number of features: 2 Features: ['id"label']Number of Unique Valuesid : 50000
label: 10

In this step, we analyze the dataset and see that our train data has around 50000 rows withtheir id and associated label. There is a total of 10 classes as in the name CIFAR-10.

Getting the validation set using PyTorch

fromtorch.utils.dataimportrandom_splitval_size = 5000 train_size=len(dataset)- val_size

train_ds,val_ds=random_split(dataset,[train_size,val_size])len(train_ds), len(val_ds)

This step is the same as the training step, but we want to split the data into train andvalidation sets.

(45000,5000)
fromtorch.utils.data.dataloaderimportDataLoader

batch_size=64
train_dl=DataLoader(train_ds,batch_size, shuffle=True, num_workers=4,pin_memory=True)
val_dl=DataLoader(val_ds, batch_size, num_workers=4,pin_memory=True)

The torch.utils have a data loader that can help us load the required data bypassing variousparams like worker number or batch size.

Defining the required functions

@torch.no_grad() def accuracy(outputs,labels):
_, preds = torch.max(outputs, dim=1) return torch.tensor(torch.sum(preds== labels).item()/len(preds))

classImageClassificationBase(nn.Module):
deftraining_step(self, batch):
images, labels= batch
out=self(images)
#Generate Predictions Loss =F.cross_entropy(out, labels)
Calculate loss accu =accuracy(out,labels)
return loss,accu

defvalidation_step(self, batch):
images, labels= batch
out=self(images) #Generate Predictions Loss=
F.cross_entropy(out, labels) # Calculate loss
acc=accuracy(out, labels) # Calculate Accuracy Return
{'Loss': loss.detach(), 'Accuracy': acc}

defvalidation_epoch_end(self, outputs):
batch_losses =[x['Loss'] forxinoutputs] epoch_loss=torch.stack(batch_losses).mean() #Combinelosses
batch_accs = [x['Accuracy'] for x in outputs]
epoch_acc=torch.stack(batch_accs).mean() # Combineaccuraciesreturn
{'Loss': epoch_loss.item(), 'Accuracy':epoch_acc.item()}

defepoch_end(self,epoch,result):
print("Epoch:",epoch+ 1)
print(f'Train Accuracy:{result["train_accuracy"]*100:.2f}% Validation

Accuracy:{result["Accuracy"]*100:.2f}%')
print(f'Train Loss:{result["train_loss"]:.4f} ValidationLoss:{result["Loss"]:.4f}')
Image 1

Convolutional neural networks, also called ConvNets, were first introduced in the 1980s by YannLeCun, a computer science researcher who worked in the background. LeCun built on the work of Kunihiko Fukushima, a Japanese scientist, a basic network for image recognition.

The old version of CNN, called LeNet (after LeCun), can see handwritten digits. CNN helpsfind pin codes from postal. But despite their expertise, ConvNets stayed close to computer vision and artificial intelligence because they faced a major problem: They could not scale much. CNN's require a lot of data and integrate resources to work well for large images.

At the time, this method was only applicable to low-resolution images. Pytorch is a library that can do deep learning operations. We can use this to perform Convolutional neural networks. Convolutional neural networks contain many layers of artificial neurons. Syntheticneurons, complex simulations of biological counterparts, are mathematical functions that
calculate the weighted mass of multiple inputs and product value activation.

As we can see here we have used class implementation of ImageClassification and it takes one parameter that is nn.Module. Within this class, we can implement the various functionsor various steps like training, validation, etc. The functions here are simple python implementations.

The training step takes images and labels in batches. we use cross-entropy for loss functionand calculate the loss and return the loss. This is similar to the validation step as we can see in the function. The epoch ends combine losses and accuracies and finally, we print the
accuracies and losses.

Conclusion: -

Questions:

Q1. How do we find the derivatives of the function in PyTorch?

Q2. Give any one difference between torch.nn and torch.nn.functional? Q3. Why it is difficult for the network is showing the problem?
Q4. Are tensor and matrix the same?

Q5. What is PyTorch?

EXPERIMENT NO. 12 (Group C)

Aim: MNIST Handwritten Character Detection using PyTorch, Keras and Tensorflow.

Outcome: At end of this experiment, student will be able to detect Handwritten Character using PyTorch, Keras and Tensorflow.

Software Requirement: Python IDE

Theory:

The MNIST handwritten digit classification problem is a standard dataset used in computervision and deep learning.

Although the dataset is effectively solved, it can be used as the basis for learning and practicing how to develop, evaluate, and use convolutional deep learning neural networks forimage classification from scratch. This includes how to develop a robust test harness for estimating the performance of the model, how to explore improvements to the model, and how to save the model and later load it to make predictions on new data.

MNIST Handwritten Digit Classification Dataset

The MNIST dataset is an acronym that stands for the Modified National Institute ofStandards and Technology dataset.

It is a dataset of 60,000 small square 28×28 pixel grayscale images of handwritten singledigits between 0 and 9.

The task is to classify a given image of a handwritten digit into one of 10 classes representinginteger values from 0 to 9, inclusively.

It is a widely used and deeply understood dataset and, for the most part, is “solved.” Top-performing models are deep learning convolutional neural networks that achieve a classification accuracy of above 99%, with an error rate between 0.4 %and 0.2% on the holdout test dataset.

The example below loads the MNIST dataset using the Keras API and creates a plot of thefirst nine images in the training dataset.

example of loading the mnist dataset from tensorflow.keras.datasets import mnistfrom matplotlib import pyplot as plt
load dataset
(trainX, trainy), (testX, testy) = mnist.load_data()# summarize loaded dataset print('Train: X=%s, y=%s' % (trainX.shape, trainy.shape))print('Test: X=%s, y=%s' % (testX.shape,

testy.shape)) # plot first few imagesfor i in range(9):
define subplot

plt.subplot(330 + 1 + i)# plot raw pixel data plt.imshow(trainX[i],
cmap=plt.get_cmap('gray'))# show the figure plt.show()

Running the example loads the MNIST train and test dataset and prints their shape.

We can see that there are 60,000 examples in the training dataset and 10,000 in the testdataset and that images are indeed square with 28×28 pixels.

Train: X=(60000, 28, 28), y=(60000,)
Test: X=(10000, 28, 28), y=(10000,)

Model Evaluation Methodology

Although the MNIST dataset is effectively solved, it can be a useful starting point for developing and practicing a methodology for solving image classification tasks using convolutional neural networks.

Instead of reviewing the literature on well-performing models on the dataset, we can developa new model from scratch.

The dataset already has a well-defined train and test dataset that we can use.

In order to estimate the performance of a model for a given training run, we can further split the training set into a train and validation dataset. Performance on the train and validation dataset over each run can then be plotted to provide learning curves and insight into how wella model is learning the problem.

The Keras API supports this by specifying the “validation_data” argument to the model.fit()function when training the model, that will, in turn, return an object that describes model performance for the chosen loss and metrics on each training epoch.

record model performance on a validation dataset during traininghistory = model.fit(..., validation_data=(valX, valY))

In order to estimate the performance of a model on the problem in general, we can use k-foldcross-validation, perhaps five-fold cross-validation. This will give some account of the models variance with both respect to differences in the training and test datasets, and in termsof the stochastic nature of the learning algorithm. The performance of a model can be taken as the mean performance across k-folds, given the standard deviation, that could be used to estimate a confidence interval if desired.

We can use the KFold class from the scikit-learn API to implement the k-fold cross-validation evaluation of a given neural network model. There are many ways to achievethis, although we can choose a flexible approach where the KFold class is only used to specify the row indexes used for each spit.

```
# example of k-fold cv for a neural netdata = ...
# prepare cross validation

kfold = KFold(5, shuffle=True, random_state=1)# enumerate splits for train_ix, test_ix in
kfold.split(data):model = ...
...
```

Conclusion: -

Questions:

Q1. What is Deep Learning?
Q2. Explain Difference between keras and tensorflow?
Q3. What is Convolution operation? Explain in details?
Q4. What is the simple working of an algorithm in TensorFlow?
Q5. What are the languages that are supported in TensorFlow?