ASSIGNMENT 7

TITLE: BACKPROPAGATION NEURAL NETWORK

PROBLEM STATEMENT: -

Write a python program to show Back Propagation Network for XOR function with Binary Input and Output

OBJECTIVE:

1. To Learn and understand the concepts of types of neural network
2. To learn and understand back propagation network
3. To understand implementation of XOR function with binary input using python.

PREREQUISITE: -

1 Basic of Python Programming

2 Concept of Artificial Neural Network

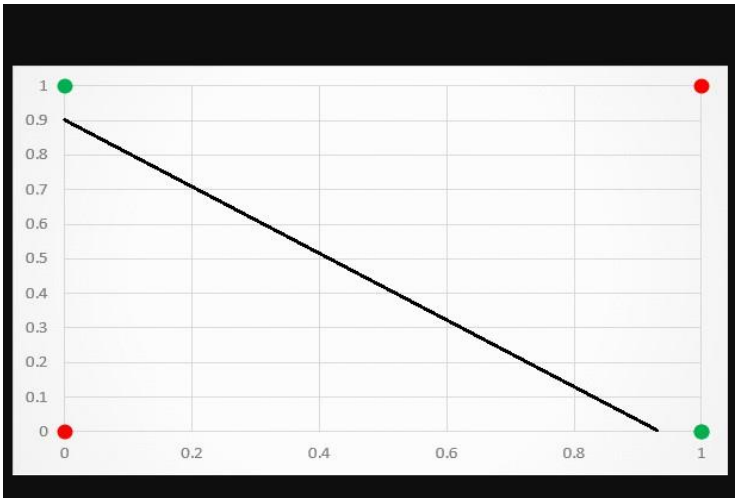
THEORY:

XOR is a problem that cannot be solved by a single layer perceptron, and therefore requires a multi-layer perceptron or a deep learning model. A single-layer perceptron can only learn linearly separable patterns, whereas a straight line or hyperplane can separate the data points. However, the XOR problem requires a non-linear decision boundary to classify the inputs accurately. This means that a single-layer perceptron fails to solve the XOR problem, emphasizing the need for more complex neural networks.

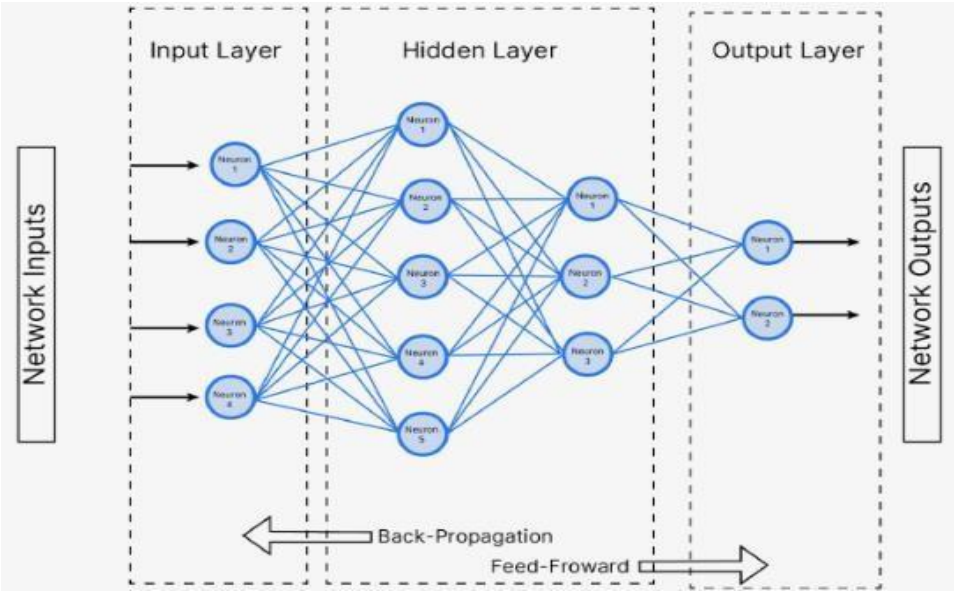
The XOR function is a binary function that takes two binary inputs and returns a binary output. The output is true if the number of true inputs is odd, and false otherwise. In other words, it returns true if exactly one of the inputs is true, and false otherwise.

Input A	Input B	Output (A XOR B)
0	0	0
0	1	1
1	0	1
1	1	0

The XOR function is not linearly separable, which means we cannot draw a single straight line to separate the inputs that yield different outputs.

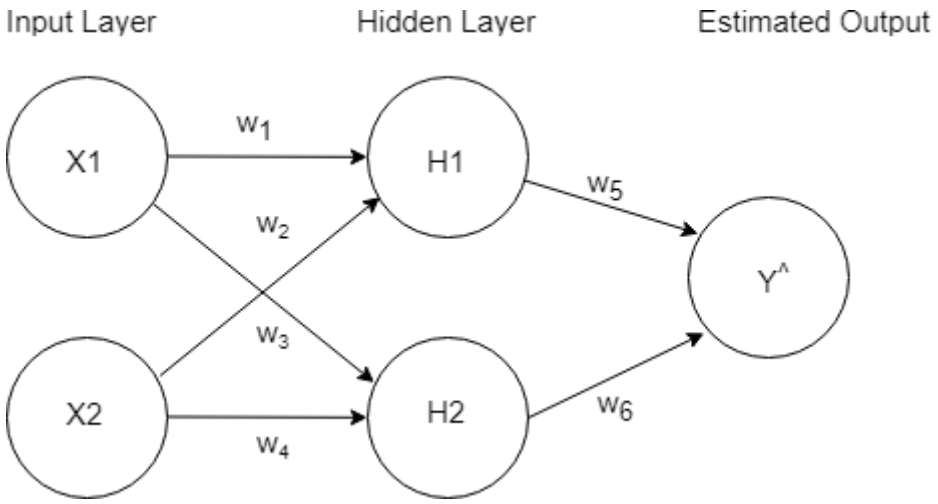


To solve the XOR problem, we need to introduce multi-layer perceptrons (MLPs) and the backpropagation algorithm. MLPs are neural networks with one or more hidden layers between the input and output layers. These hidden layers allow the network to learn non-linear relationships between the inputs and outputs.



The backpropagation algorithm is a learning algorithm that adjusts the weights of the neurons in the network based on the error between the predicted output and the actual output. It works by propagating the error backwards through the network and updating the weights using gradient descent.In addition to MLPs and the backpropagation algorithm, the choice of activation functions also plays a crucial role in solving the XOR problem. Activation functions introduce non-linearity into the network, allowing it to learn complex patterns. Popular

activation functions for solving the XOR problem include the sigmoid function and the hyperbolic tangent function.



Inputs

$$\begin{aligned} \boldsymbol{x_1} &= [1,1]^T, \quad y_1 = +1 \\ \boldsymbol{x_2} &= [0,0]^T, \quad y_2 = +1 \\ \boldsymbol{x_3} &= [1,0]^T, \quad y_3 = -1 \\ \boldsymbol{x_4} &= [0,1]^T, \quad y_4 = -1 \end{aligned}$$

2 hidden neurons are used, each takes two inputs with different weights. After each forward pass, the error is back propogated. I have used sigmoid as the activation function at the hidden layer.

At hidden layer:

$$H_1 = x_1 w_1 + x_2 w_2$$

$$H_2 = x_1 w_3 + x_2 w_4$$

At output layer:

$$Y^{\wedge} = \sigma(H_1) w_5 + \sigma(H_2) w_6$$

here, σ represents sigmoid function.

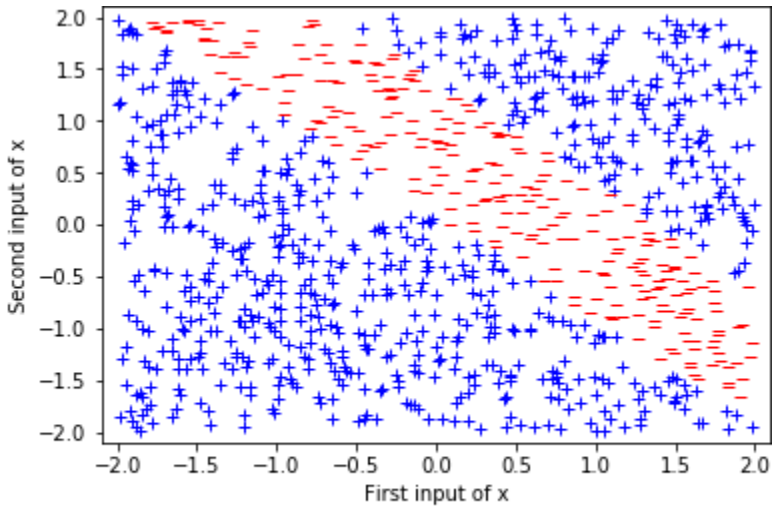
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Loss function:

$$\frac{1}{2}(Y - Y^{\wedge})^2$$

Results

Classification WITHOUT gaussian noise




```
(731, 269)
```

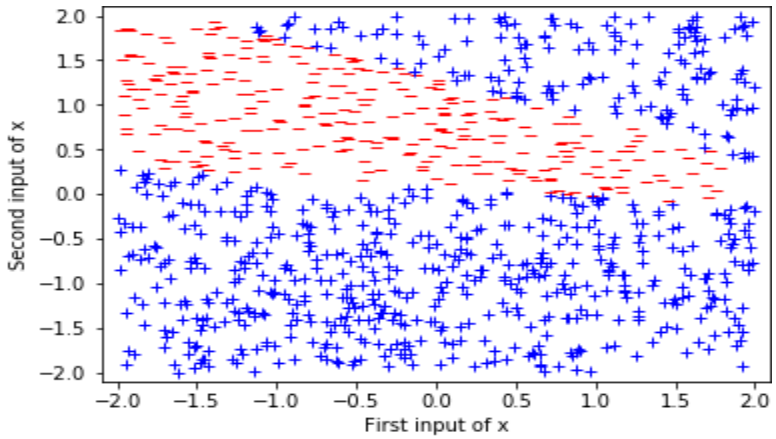
Observation: To classify, I generated 1000 x 2 random floats in range -2 to 2. Using weights from trained model, I classified each input & plotted it on 2-D space. Out of 1000, 731 points were classified as "+1" and 269 points were classified as "-1" It is clearly seen, classification region is not a single line, rather the 2-D region is separated by "-1" class. I did the same for classifications with Gaussian noise.

Classification WITH Gaussian noise

In real applications, we almost never work with data without noise. Now instead of using the above points generate Gaussian random noise centered on these locations.

$$\boldsymbol{x_1} \sim \boldsymbol{\mu_1} = [1,1]^T, \Sigma_1 = \Sigma \quad y_1 = +1$$
$$\boldsymbol{x_2} \sim \boldsymbol{\mu_1} = [0,0]^T, \Sigma_2 = \Sigma \quad y_1 = +1$$
$$\boldsymbol{x_3} \sim \boldsymbol{\mu_1} = [1,0]^T, \Sigma_3 = \Sigma \quad y_1 = -1$$
$$\boldsymbol{x_4} \sim \boldsymbol{\mu_1} = [0,1]^T, \Sigma_4 = \Sigma \quad y_1 = -1$$
$$\Sigma = \begin{bmatrix} \sigma & 0 \\ 0 & \sigma \end{bmatrix}$$

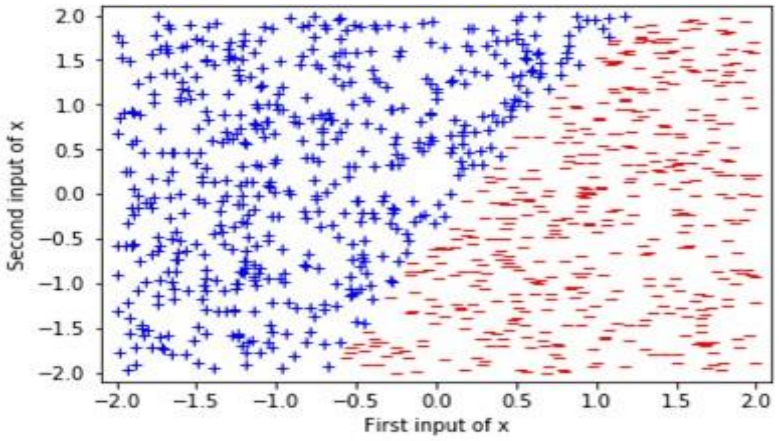
σ = 0.5




```
(702, 298)
```

Observation : We see a shift in classification regions. Here, classification is still not separated by a line

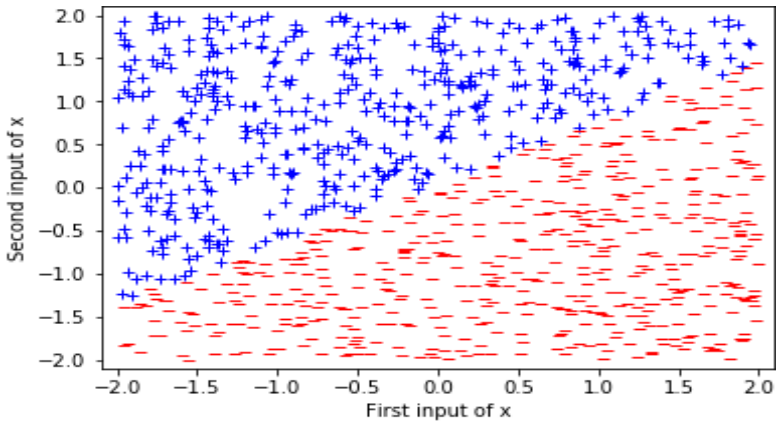
σ = 1.0




```
(538, 462)
```

Observation : We observe classifications being divided into two distinct regions.

σ = 2.0




```
(467, 533)
```

Observation : We see a shift in classification regions (compared to of σ = 1). Here also, we observe two distinct regions of classification.

CONCLUSION: We have studied and implemented Back Propagation Network for XOR function with Binary Input and Output

ASSIGNMENT 8

TITLE: BACK PROPAGATION FEED-FORWARD NEURAL NETWORK

PROBLEM STATEMENT: -

. Write a python program in python program for creating a Back Propagation Feed-forward neural network

OBJECTIVE:

1. To Learn and understand the Back Propagation Feed-forward neural network
2. To learn and understand back propagation network

PREREQUISITE: -

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

THEORY:

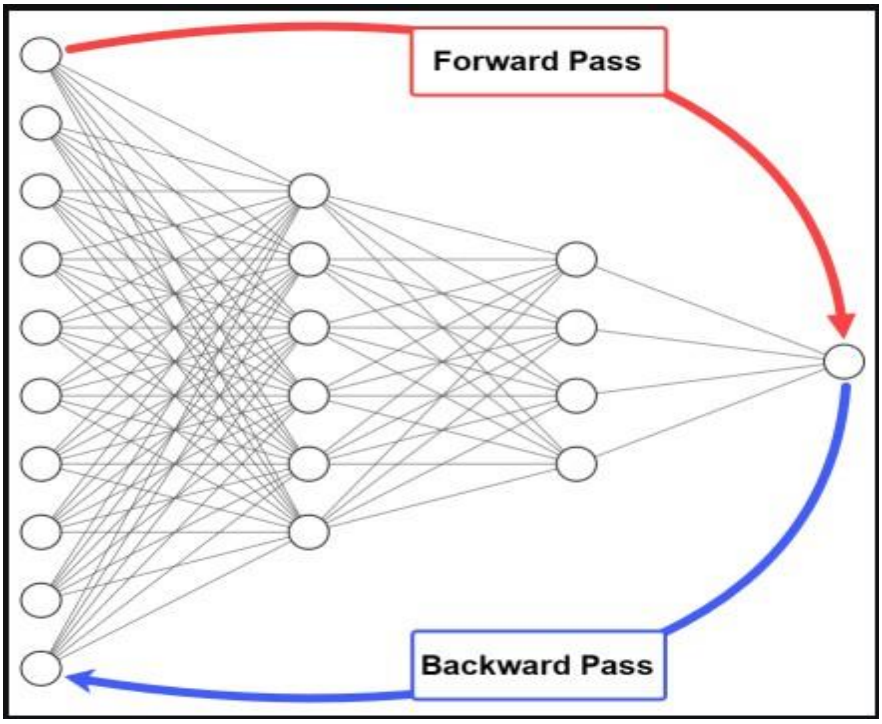
Backpropagation neural network is used to improve the accuracy of neural network and make them capable of self-learning. The backpropagation algorithm is one of the algorithms responsible for updating network weights with the objective of reducing the network error. Backpropagation means "backward propagation of errors". Here error is spread into the reverse direction in order to achieve better performance. Backpropagation is an algorithm for supervised learning of artificial neural networks that uses the gradient descent method to minimize the cost function. It searches for optimal weights that optimize the mean-squared distance between the predicted and actual labels.

The backpropagation algorithm consists of two phases:

The forward pass where our inputs are passed through the network and output predictions obtained (also known as the propagation phase).

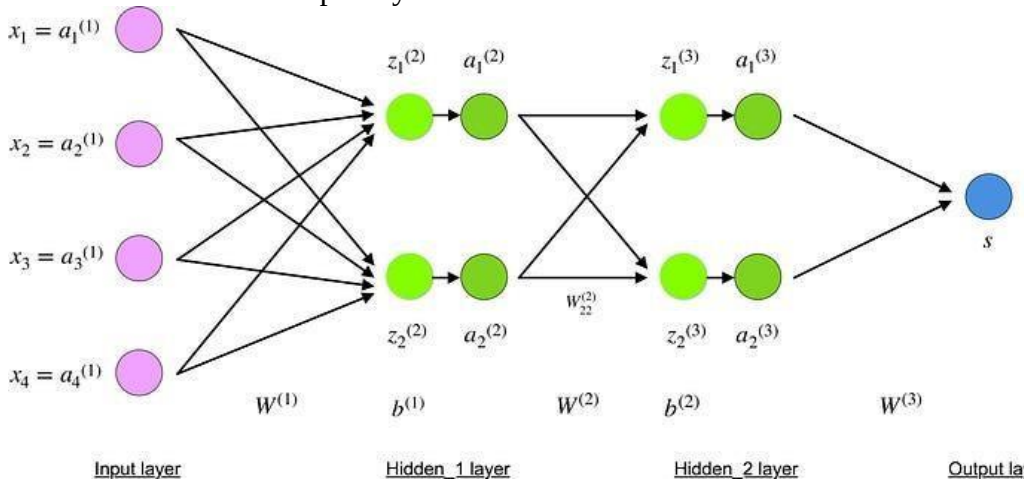
The backward pass where we compute the gradient of the loss function at the final layer (i.e., predictions layer) of the network and use this gradient to recursively apply the chain rule

to update the weights in our network (also known as the weight update phase).



At the end of the forward pass, the network error is calculated, and should be as small as possible. If the current error is high, the network didn't learn properly from the data. What does this mean? It means that the current set of weights isn't accurate enough to reduce the network error and make accurate predictions. As a result, we should update network weights to reduce the network error.

The 4-layer neural network consists of 4 neurons for the input layer, 4 neurons for the hidden layers and 1 neuron for the output layer.



Input layer

The neurons, colored in **purple**, represent the input data. These can be as simple as scalars or more complex like vectors or multidimensional matrices.

$$x_i = a_i^{(1)}, i \in 1,2,3,4$$

Equation for input x_i

The first set of activations (*a*) are equal to the input values. *NB: "activation" is the neuron's value after applying an activation function. See below.*

Hidden layers

The final values at the hidden neurons, colored in **green**, are computed using *z^l* — weighted inputs in layer *l*, and *a^l*— activations in layer *l*. For layer 2 and 3 the equations are:

- *l = 2*

$$z^{(2)} = W^{(1)}x + b^{(1)}$$

$$a^{(2)} = f(z^{(2)})$$

Equations for z^2 and a^2

- *l = 3*

$$z^{(3)} = W^{(2)}a^{(2)} + b^{(2)}$$

$$a^{(3)} = f(z^{(3)})$$

Equations for z^3 and a^3

W^2 and W^3 are the weights in layer 2 and 3 while b^2 and b^3 are the biases in those layers.

Activations a^2 and a^3 are computed using an activation function f. Typically, this **function f is non-linear** (e.g. sigmoid, ReLU, tanh) and allows the network to learn complex patterns in data. We won't go over the details of how activation functions work, but, if interested, I strongly recommend reading this great article.Looking carefully, you can see that all of x, z^2, a^2, z^3, a^3, W^1, W^2, b^1 and b^2 are missing their subscripts presented in the 4-layer network illustration above. **The reason is that we have combined all parameter values in matrices, grouped by layers.** This is the standard way of working with neural networks and one should be comfortable with the calculations. However, I will go over the equations to clear out any confusion.

Let's pick layer 2 and its parameters as an example. The same operations can be applied to any layer in the network. W^1 is a weight matrix of shape (n, m) where n is the number of output neurons (neurons in the next layer) and m is the number of input neurons (neurons in the previous layer). For us, $n = 2$ and $m = 4$.

$$W^{(1)} = \begin{bmatrix} W_{11}^{(1)} & W_{12}^{(1)} & W_{13}^{(1)} & W_{14}^{(1)} \\ W_{21}^{(1)} & W_{22}^{(1)} & W_{23}^{(1)} & W_{24}^{(1)} \end{bmatrix}$$

Equation for W^1

NB: The first number in any weight's subscript matches the index of the neuron in the next layer (in our case this is the *Hidden_2 layer*) **and the second number matches the index of the neuron in previous layer** (in our case this is the *Input layer*).

- x is the input vector of shape $(m, 1)$ where m is the number of input neurons. For us, $m = 4$.

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

Equation for x

- b^1 is a bias vector of shape $(n , 1)$ where n is the number of neurons in the current layer.

  For us, $n = 2$.

$$b^{(1)} = \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \end{bmatrix}$$

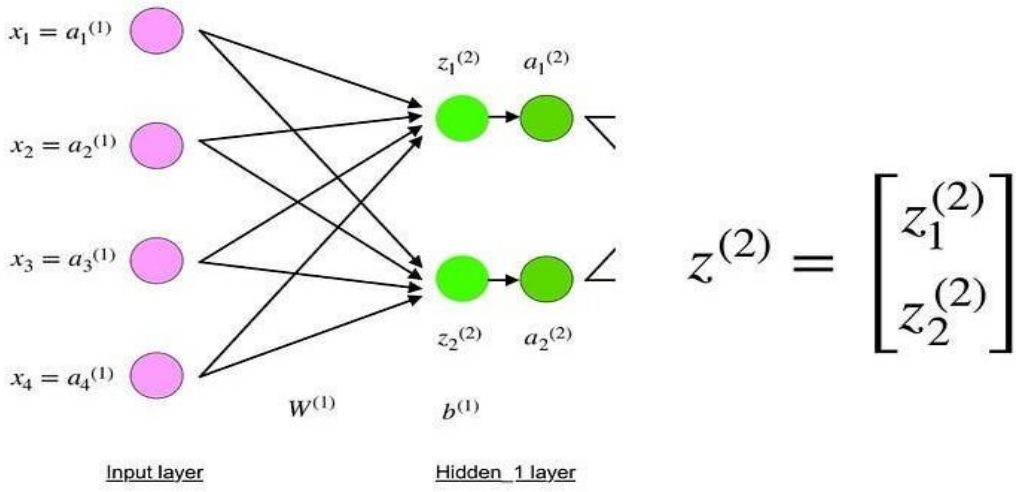
Equation for b^1

Following the equation for z^2, we can use the above definitions of W^1, x and b^1 to derive "*Equation for z^2*":

$$z^{(2)} = \begin{bmatrix} W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + W_{14}^{(1)}x_4 \\ W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + W_{24}^{(1)}x_4 \end{bmatrix} + \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \end{bmatrix}$$

Equation for z^2

Now carefully observe the neural network illustration from above.



Input and Hidden_1 layers

You will see that z^2 can be expressed using $(z\_1)^2$ and $(z\_2)^2$ where $(z\_1)^2$ and $(z\_2)^2$ are the sums of the multiplication between every input x_i with the corresponding weight $(W\_ij)^1$. This leads to the same "*Equation for z^2*" and proofs that the matrix representations for z^2, a^2, z^3 and a^3 are correct.

Output layer

The final part of a neural network is the output layer which produces the predicated value. In our simple example, it is presented as a single neuron, colored in blue and evaluated as follows:

$$s = W^{(3)}a^{(3)}$$

Equation for output s

Again, we are using the matrix representation to simplify the equation. One can use the above techniques to understand the underlying logic.

1. Forward propagation and evaluation

The equations above form network's forward propagation. Here is a short overview:

$$x = a^{(1)} \quad Input\ layer$$

$$z^{(2)} = W^{(1)}x + b^{(1)} \quad neuron\ value\ at\ Hidden_1\ layer$$

$$a^{(2)} = f(z^{(2)}) \quad activation\ value\ at\ Hidden_1\ layer$$

$$z^{(3)} = W^{(2)}a^{(2)} + b^{(2)} \quad neuron\ value\ at\ Hidden_2\ layer$$

$$a^{(3)} = f(z^{(3)}) \quad activation\ value\ at\ Hidden_2\ layer$$

$$s = W^{(3)}a^{(3)} \quad Output\ layer$$

Overview of forward propagation equations colored by layer

The final step in a forward pass is to evaluate the **predicted output** *s* against an **expected output** *y*.

The output *y* is part of the training dataset *(x, y)* where *x* is the input (as we saw in the previous section).

Evaluation between *s* and *y* happens through a **cost function**. This can be as simple as MSE (mean squared error) or more complex like cross-entropy.

We name this cost function *C* and denote it as follows:

$$C = cost(s, y)$$

Equation for cost function C were *cost* can be equal to MSE, cross-entropy or any other cost function.Based on *C*'s value, the model "knows" how much to adjust its parameters in order to get closer to the expected output *y*. This happens using the backpropagation algorithm.

1. Backpropagation and computing gradients

According to the paper from 1989, backpropagation: repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector. And the ability to create useful new features distinguishes back-propagation from earlier, simpler methods…

In other words, **backpropagation aims to minimize the cost function by adjusting network's weights and biases.** The level of adjustment is determined by the gradients of the cost function with respect to those parameters.

One question may arise — **why computing gradients**?

To answer this, we first need to revisit some calculus terminology:

- *Gradient of a function C(x_1, x_2, ..., x_m) in point x is a vector of the partial derivatives of C in x.*

$$\frac{\partial C}{\partial x} = [\frac{\partial C}{\partial x_1}, \frac{\partial C}{\partial x_2}, \ldots, \frac{\partial C}{\partial x_m}]$$

Equation for derivative of C in x

- *The derivative of a function C measures the sensitivity to change of the function value (output value) with respect to a change in its argument x (input value). In other words, the derivative tells us the direction C is going.*

- *The gradient shows how much the parameter x needs to change (in positive or negative direction) to minimize C.*

Compute those gradients happens using a technique called chain rule.

For a single weight *(w_jk)^l*, the gradient is:

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{jk}^l} \qquad chain\ rule$$

$$z_j^l = \sum_{k=1}^{m} w_{jk}^l a_k^{l-1} + b_j^l \qquad by\ definition$$

$$m\ -\ number\ of\ neurons\ in\ l-1\ layer$$

$$\frac{\partial z_j^l}{\partial w_{jk}^l} = a_k^{l-1} \qquad by\ differentiation\ (calculating\ derivative)$$

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l}a_k^{l-1} \qquad final\ value$$

Equations for derivative of C in a single weight *(w_jk)^l*

Similar set of equations can be applied to *(b_j)^l*:

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial b_j^l} \qquad chain\ rule$$

$$\frac{\partial z_j^l}{\partial b_j^l} = 1 \qquad by\ differentiation\ (calculating\ derivative)$$

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l}1 \qquad final\ value$$

Equations for derivative of C in a single bias *(b_j)^l* The common part in both equations is often called *"local gradient"* and is expressed as follows:

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} \qquad local\ gradient$$

Equation for local gradient

The *"local gradient"* can easily be determined using the chain rule. I won't go over the process now but if you have any questions, please comment below.

The gradients allow us to optimize the model's parameters:

$$while\ (termination\ condition\ not\ met)$$

$$w := w - \epsilon \frac{\partial C}{\partial w}$$

$$b := b - \epsilon \frac{\partial C}{\partial b}$$

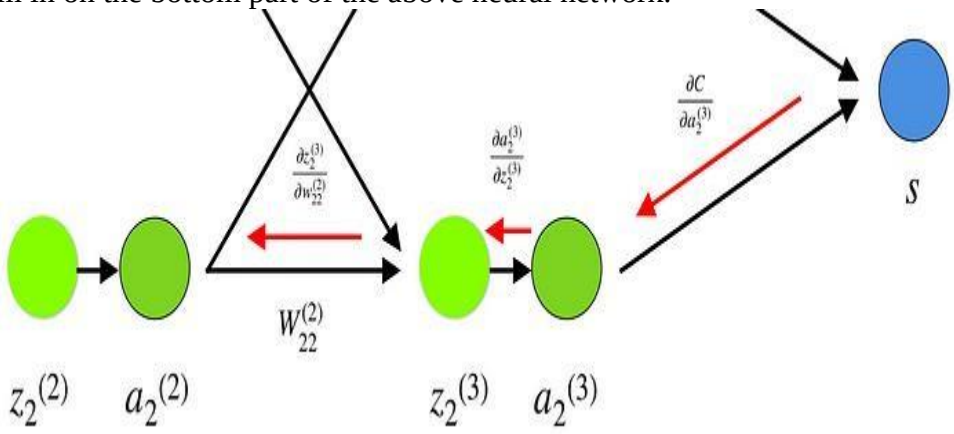
$$end$$

Algorithm for optimizing weights and biases (also called "Gradient descent")

- Initial values of *w* and *b* are randomly chosen.
- Epsilon (*e*) is the learning rate. It determines the gradient's influence.
- *w* and *b* are matrix representations of the weights and biases. Derivative of *C* in *w* or *b* can be calculated using partial derivatives of *C* in the individual weights or biases.
- Termination condition is met once the cost function is minimized.

I would like to dedicate the final part of this section to a simple example in which we will calculate the gradient of *C* with respect to a single weight *(w_22)²*.

Let's zoom in on the bottom part of the above neural network:



Visual representation of backpropagation in a neural network

Weight *(w_22)²* connects *(a_2)²* and *(z_2)²*, so computing the gradient requires applying the chain rule through *(z_2)³* and *(a_2)³:*

$$\frac{\partial C}{\partial w_{22}^{(2)}} = \frac{\partial C}{\partial z_2^{(3)}} \cdot \frac{\partial z_2^{(3)}}{\partial w_{22}^{(2)}} = \frac{\partial C}{\partial a_2^{(3)}} \cdot \frac{\partial a_2^{(3)}}{\partial z_2^{(3)}} \cdot a_2^{(2)} = \frac{\partial C}{\partial a_2^{(3)}} \cdot f'(z_2^{(3)}) \cdot a_2^{(2)}$$

Equation for derivative of C in $(w\_22)^2$

Calculating the final value of derivative of *C* in $(a\_2)^3$ requires knowledge of the function *C*. Since *C* is dependent on $(a\_2)^3$, calculating the derivative should be fairly straightforward.

CONCLUSION: We have studied and implemented Back Propagation Network for XOR function with Binary Input and Output

ASSIGNMENT 9

TITLE: HOPFIELD NETWORK

PROBLEM STATEMENT: -

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

OBJECTIVE:

- To Learn and understand the concepts of types of neural network
- To learn and understand back propagation network
- To understand implementation of XOR function with binary input using python.

PREREQUISITE: -

- Basic of Python Programming
- Concept of Artificial Neural Network

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.

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.

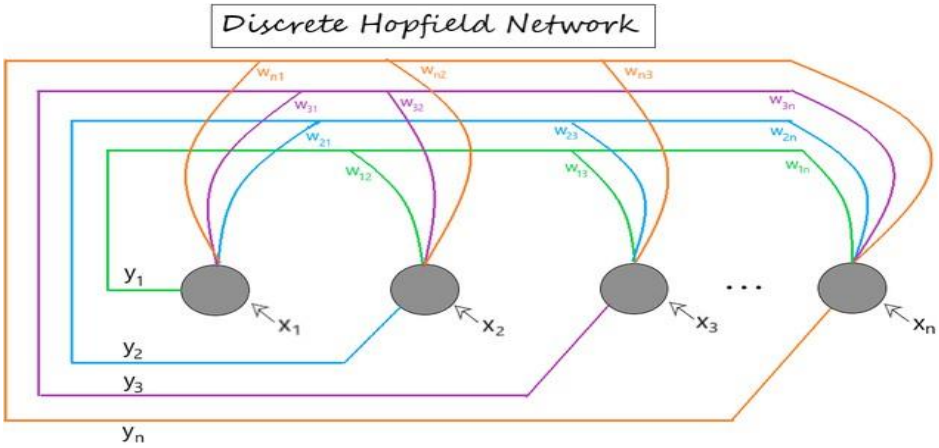
$$1.\ w_{ij} = w_{ji}$$
$$2.\ w_{ii} = 0$$

- **Structure & Architecture of Hopfield Network**

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 the

self. The below figure shows a sample representation of a Discrete Hopfield Neural Network architecture having the following elements.



Discrete Hopfield Network Architecture

[x1 , x2 , … , xn] -> Input to the n given neurons.
[y1 , y2 , … , yn] -> Output obtained from the n given neurons
W_{ij} -> weight associated with the connection between the i^{th} and the j^{th} neuron.

Training Algorithm

For storing a set of input patterns S(p) [p = 1 to P], where **S(p) = S_1(p) … S_i(p) … S_n(p)**, the weight 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 in the training of a Hopfield Network are as mapped below:

- Initialize weights **(w_{ij})** to store patterns **(using training algorithm)**.
- For each input vector y_i, perform **steps 3-7**.
- Make the initial activators of the network equal to the external input vector x.

$$y_i = x_i : (for \ i = 1 \ to \ n)$$

- For each vector y_i, perform **steps 5-7**.
- Calculate the total input of the network **y_{in}** using the equation given below.

$$y_{in_i} = x_i + \sum_j [y_j w_{ji}]$$

- Apply activation over the total input to calculate the output as per the equation given below:

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

(where θ_i (threshold) and is normally taken as 0)

- Now feedback the obtained output y_i to all other units. Thus, the activation vectors are updated.
- Test the network for convergence.

Continuous Hopfield Network

Unlike the discrete Hopfield networks, here the time parameter is treated as a continuous variable. So, instead of getting binary/bipolar outputs, we can obtain values that lie between 0 and 1. It can be used to solve constrained optimization and associative memory problems. The output is defined as:

$$v_i = g(u_i)$$

where,

- v_i = *output from the continuous hopfield network*
- u_i = *internal activity of a node in continuous hopfield network.*

CONCLUSION: We have studied and implemented Back Propagation Network for XOR function with Binary Input and Output.

ASSIGNMENT 10

TITLE: OBJECT DETECTION

PROBLEM STATEMENT: -

Write Python program to implement CNN object detection. Discuss numerous performance evaluation metrics for evaluating the object detecting algorithms' performance.

OBJECTIVE:

- To Learn and understand the concepts of types of neural network
- To learn and understand CNN
- To understand implementation of object detection.

PREREQUISITE: -

- Basic of Python Programming
- Concept of Artificial Neural Network

THEORY:

"A convolutional neural network is a class of deep neural networks, most commonly applied to analyzing visual imagery. CNN is an improved version of multilayer perceptron". It's a class of deep neural network inspired by human's visual cortex.

Basically CNN works by collecting matrix of features then predicting whether this image contains a class or another class based on these features using softmax probabilities.

Object detection is a computer vision technique for locating instances of objects in images or videos. Region-Based Convolutional Neural Networks, or R-CNNs, are a family of techniques for addressing object localization and recognition tasks, designed for model performance.

The evaluation metrics for object detection model assess its ability to accurately identify and locate objects in an image. It's typically measured through metrics like Average Precision (AP) or mAP (mean Average Precision), which consider the precision and recall of the model across different object categories and detection thresholds. Average Precision (AP) and mean Average Precision (mAP) are the most popular metrics used to evaluate object detection models, such as Faster R_CNN, Mask R-CNN, and YOLO, among others.

Evaluating the performance of an object detector determines if the detection is correct.

Definition of terms:

True Positive (TP) — Correct detection made by the model.

False Positive (FP) — Incorrect detection made by the detector.

False Negative (FN) — A Ground-truth missed (not detected) by the object detector.

True Negative (TN) —this is the background region correctly not detected by the model. This metric is not used in object detection because such regions are not explicitly annotated when preparing the annotations.

Intersection Over Union (IOU)

IOU is a metric that finds the difference between ground truth annotations and predicted bounding boxes. This metric is used in most state of art object detection algorithms. In object detection, the model predicts multiple bounding boxes for each object, and based on the confidence scores of each bounding box it removes unnecessary boxes based on its threshold value. We need to declare the threshold value based on our requirements.

IOU = Area of union / area of intersection

Precision: Is a critical metric in model evaluation as it serves to quantify the accuracy of the positive predictions made by the model. It specifically assesses how well the model distinguishes true objects from false positives. In essence, precision provides insight into the model's ability to make positive predictions that are indeed accurate. A high precision score indicates that the model is skilled at avoiding false positives and provides reliable positive predictions.

$$\text{precision} = \frac{TP}{TP + FP}$$

Recall: Recall, also known as sensitivity or true positive rate, is another essential metric used in evaluating model performance, especially in object detection tasks. Recall measures the model's capability to capture all relevant objects in the image. In essence, recall assesses the model's completeness in identifying objects of interest. A high recall score indicates that the model effectively identifies most of the relevant objects in the data.

$$\text{recall} = \frac{TP}{TP + FN}$$

F1-Score: Is the harmonic mean of precision and recall. It provides a balanced measure of the model's performance, considering both false positives and false negatives. This metric is particularly useful when there is an imbalance between positive and negative classes in the dataset.

$$F_1 = \frac{2 * (\text{precision} * \text{recall})}{(\text{precision} + \text{recall})}$$

Average Precision(AP) : It is calculated using area under the curve (AUC) of the Precision x Recall curve. As AP curves are often zigzag curves, comparing different curves (different detectors) in the same plot usually is not an easy task. In practice AP is the precision averaged across all recall values between 0 and 1.

$$\text{Average Precision (AP)} = \int_{r=0}^{1} p(r)dr$$

Mean Average Precision(mAP) : The mAP score is calculated by taking the mean AP over all classes and/or over all IoU thresholds, depending on the competition. AP value can be calculated for each class. The mean average precision is calculated by taking the average of AP across all the classes under consideration. i.e

$$mAP = \frac{1}{k} \sum_{i}^{k} AP_i$$

CONCLUSION: We have studied and implemented Back Propagation Network for XOR function with Binary Input and Output

ASSIGNMENT 11

TITLE: LOGISTIC REGRESSION USING TENSORFLOW

PROBLEM STATEMENT:

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

OBJECTIVE:

- Training a Neural Network
- Evaluation of Logistic Regression (using TensorFlow)

THEORY:

What is Tensor Flow:

TensorFlow allows developers to create dataflow graphs—structures that describe how data moves through a graph, or a series of processing nodes. Each node in the graph represents a mathematical operation, and each connection or edge between nodes is a multidimensional data array, or tensor.

TensorFlow applications can be run on most any target that's convenient: a local machine, a cluster in the cloud, iOS and Android devices, CPUs or GPUs. If you use Google's own cloud, you can run TensorFlow on Google's custom TensorFlow Processing Unit (TPU) silicon for further acceleration. Models created by TensorFlow can be deployed on most any device to serve predictions.

what is Pytorch?

PyTorch is an optimized Deep Learning tensor library based on Python and Torch and is mainly used for applications using GPUs and CPUs. PyTorch is favored over other Deep Learning frameworks like TensorFlow and Keras since it uses dynamic computation graphs and is completely Pythonic. It allows scientists, developers, and neural network debuggers to run and test portions of the code in real-time. Thus, users don't have to wait for the entire code to be implemented to check if a part of the code works or not.

The two main features of PyTorch are:

Tensor Computation (similar to NumPy) with strong GPU (Graphical Processing Unit) acceleration support

Automatic Differentiation for creating and training deep neural networks

What is regression?

Regression is a statistical method used to model the relationship between a dependent variable (often denoted as Y) and one or more independent variables (often denoted as X). The goal of regression analysis is to understand how the independent variables influence the dependent variable and to predict the value of the dependent variable based on the values of the independent variables.

What is logistic regression?

Logistic regression is the appropriate regression analysis to conduct when the dependent variable is dichotomous (binary). Like all regression analyses, logistic regression is a predictive analysis. It is used to describe data and to explain the relationship between one dependent binary variable and one or more nominal, ordinal, interval or ratio-level independent variables.

Logistic Regression is another statistical analysis method borrowed by Machine Learning. It is used when our dependent variable is dichotomous or binary. It just means a variable that has only 2 outputs, for example, A person will survive this accident or not, The student will pass this exam or not. The outcome can either be yes or no (2 outputs). This regression technique is similar to linear regression and can be used to predict the Probabilities for classification problems.

Type of Logistic Regression:

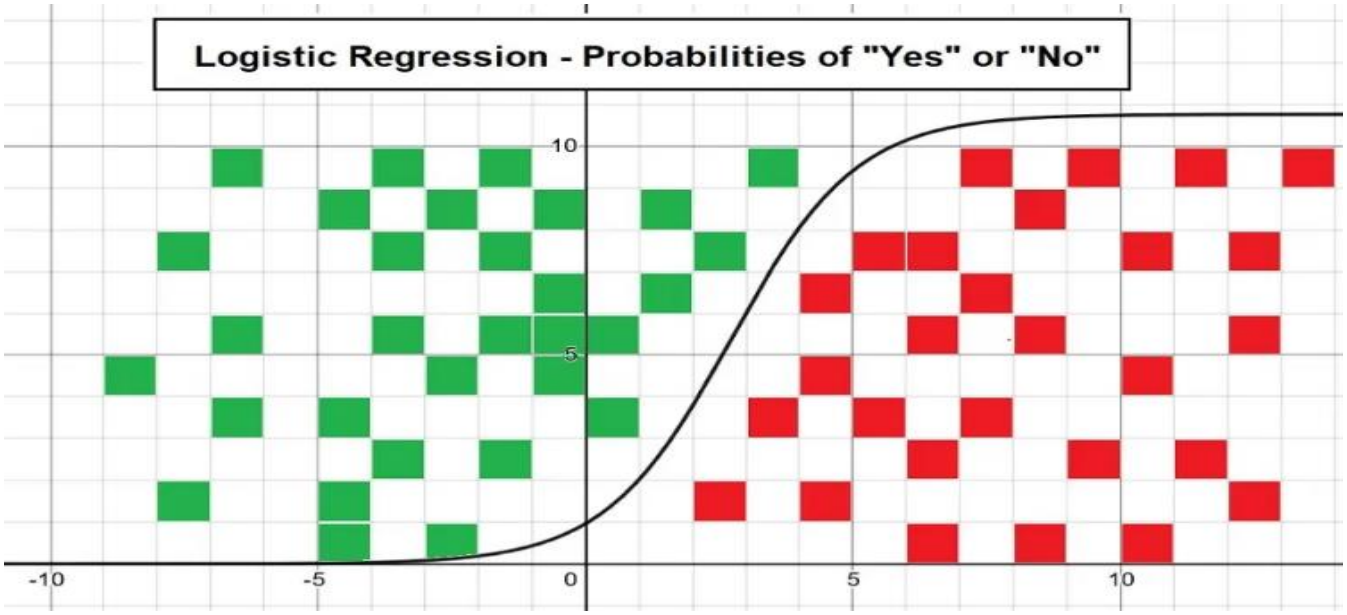
Binomial: There can be only two possible types of dependent variables, such as 0 or 1, Yes or No, etc.

Multinomial: There can be three or more possible unordered types of the dependent variable, such as “cat,” “dogs,” or “sheep.”

Ordinal: There can be three or more possible ordered types of dependent variables, such as “low,” “Medium,” or “High.”

Example of Logistic Regression Plot

The following plot shows logistic regression by considering the probabilities of “Yes” and “No”:



Assumptions of logistic regression

The output data is yes and no

Linearity

Little outliers

Independence in value

As we mentioned above, the answer to our logistic regression model is between 0 and 1. To ensure our output is at that value, we have to squash that data into that range. The best tool for doing this would be a sigmoid function.

Algorithm:

Step 1: Import all necessary modules

Step 2: Loading and preparing the mnist data set

Step 3: Setting up hyper parameters and data set parameters.

Step 4: Shuffling and batching the data

Step 5: Initialize weight and biases

Step 6: Define logistic regression and cost function

Step 7: Defining the optimizers and updating weight and biases

Step 8: Optimization process and updating weight and biases

Step 9: The training loop

Step 10: Testing model accuracy using the test data.

CONCLUSION:

We implement Neural network with TensorFlow/ Pytorch and evaluation of logistic regression using tensorflow

ASSIGNMENT 12

TITLE: TENSORFLOW/PYTORCH IMPLEMENTATION OF CNN

PROBLEM STATEMENT:

TensorFlow/Pytorch implementation of CNN

OBJECTIVE:

- Implement CNN using TensorFlow/Pytorch

THEORY:

What is CNN:

Convolutional Neural Networks (CNN) is a type of Deep Learning algorithm which is highly instrumental in learning patterns and features in images. CNN has a unique trait which is its ability to process data with a grid-like topology whereas a typical Artificial Neural Network (Dense or Sparse) generally takes input by flattening the tensors into a one-dimensional vector. This facilitates it to learn and differentiate between features in images, which when represented digitally are essentially a grid of numbers.

Convolutional Neural Networks are typically comprised of multiple layers. Usually, the initial layers are used to detect simple features such as edges, and complex features are detected down the line, as we go deeper into the network.

CNN has countless qualities that make it so suitable for processing images. Let's take a look at some of them:-

They require much less data pre-processing than other Deep Learning Algorithms.

A well-trained CNN model has the ability to learn and classify features in an image, which gives much better accuracy in the classification and detection of features in images.

It can save a lot of computational resources by methods like increasing the convolutional and pooling layers.

What are Convolutional and Pooling Layers in CNN?

Convolutional Layers:

These are the first layers in a CNN, and they can be thought of as “Filters” for an image. Just like Filters in Instagram detect our face, a convolutional layer detects features or filters such as edges in an image, wherever they might be present.

Pooling Layers:

The pooling layers mainly reduce the computational cost by reducing the spatial size of the image. The best way to describe it would be that it makes the grids of information smaller by taking a “lump-sum” of the images’ spatial resolution.

Stepwise implementation

Step 1: Load and Preprocess the Dataset

Load the dataset you want to work with (e.g., CIFAR-10, MNIST).

Preprocess the data by normalizing pixel values to the range [0, 1] and reshaping if necessary.

Split the dataset into training and testing sets.

Step 2: Define the CNN Model Architecture

Initialize a Sequential model using tf.keras.models.Sequential().

Add convolutional layers using tf.keras.layers.Conv2D() with appropriate parameters such as number of filters, kernel size, and activation function.

Add pooling layers using tf.keras.layers.MaxPooling2D() to downsample the feature maps.

Optionally, add additional convolutional and pooling layers for deeper network architecture.

Flatten the 2D feature maps into a 1D vector using tf.keras.layers.Flatten().

Add fully connected (Dense) layers using tf.keras.layers.Dense() for classification, with appropriate activation functions.

Step 3: Compile the Model

Compile the model using model.compile() function.

Choose an optimizer (e.g., Adam, SGD) and specify its parameters.

Specify the loss function (e.g., categorical cross-entropy, sparse categorical cross-entropy) appropriate for your task.

Optionally, specify additional metrics to monitor during training (e.g., accuracy).

Step 4: Train the Model

Train the model on the training data using model.fit() function.

Specify the training data, number of epochs, batch size, and validation data (if available).

Monitor the training progress and adjust hyperparameters as needed.

Step 5: Evaluate the Model

Evaluate the trained model on the testing data using model.evaluate() function.

Calculate metrics such as accuracy, loss, and any additional metrics specified during compilation.

Step 6: Fine-tuning and Optimization

Experiment with different model architectures, hyperparameters, and optimization algorithms to improve performance.

Apply regularization techniques such as dropout or L2 regularization to prevent overfitting.

Perform hyperparameter tuning using techniques such as grid search or random search.

Explore techniques such as data augmentation to increase the diversity of training data and improve generalization.

Step 7: Deployment and Inference

Once satisfied with the model performance, deploy the model for inference on new data.

Serialize the trained model using TensorFlow's SavedModel format or HDF5 format.

Integrate the model into your application or deploy it to a production environment for real-time predictions.

By following these step-by-step instructions, you can implement a CNN using TensorFlow for image classification tasks and achieve state-of-the-art performance on various datasets.

CONCLUSION:

We learn implementation of CNN using TensorFlow for image classification

ASSIGNMENT NO 13

TITLE: MNIST HANDWRITTEN CHARACTER DETECTION

PROBLEM STATEMENT:

MNIST Handwritten Character Detection using PyTorch, Keras and Tensorflow

OBJECTIVE:

- Implement MNIST Handwritten Character Detection using PyTorch, Keras and Tensorflow

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.

The handwritten digit recognition is the ability of computers to recognize human handwritten digits. It is a hard task for the machine because handwritten digits are not perfect and can be made with many different flavors. The handwritten digit recognition is the solution to this problem which uses the image of a digit and recognizes the digit present in the image.

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.

What is Keras?

Keras is an effective high-level neural network Application Programming Interface (API) written in Python. This open-source neural network library is designed to provide fast experimentation with deep neural networks, and it can run on top of CNTK, TensorFlow, and Theano.

Keras focuses on being modular, user-friendly, and extensible. It does not handle low-level computations; instead, it hands them off to another library called the Backend.

Keras was adopted and integrated into TensorFlow in mid-2017. Users can access it via the tf.keras module. However, the Keras library can still operate separately and independently.

What is PyTorch?

PyTorch is a relatively new deep learning framework based on Torch. Developed by Facebook's AI research group and open-sourced on GitHub in 2017, it's used for natural language processing applications. PyTorch has a reputation for simplicity, ease of use, flexibility, efficient memory usage, and dynamic computational graphs. It also feels native, making coding more manageable and increasing processing speed.

What is TensorFlow?

TensorFlow is an end-to-end open-source deep learning framework developed by Google and released in 2015. It is known for documentation and training support, scalable production and deployment options, multiple abstraction levels, and support for different platforms, such as Android.

TensorFlow is a symbolic math library used for neural networks and is best suited for dataflow programming across a range of tasks. It offers multiple abstraction levels for building and training models.

Implementation

Step 1: Import the libraries and load the dataset

we are going to import all the modules that we are going to need for training our model. The Keras library already contains some datasets and MNIST is one of them. So we can easily import the dataset and start working with it. The mnist.load_data() method returns us the training data, its labels and also the testing data and its labels.

Step 2: Preprocess the data

The image data cannot be fed directly into the model so we need to perform some operations and process the data to make it ready for our neural network. The dimension of the training data is (60000,28,28). The CNN model will require one more dimension so we reshape the matrix to shape (60000,28,28,1).

Step 3: Create the model

Now we will create our CNN model in Python data science project. A CNN model generally consists of convolutional and pooling layers. It works better for data that are represented as grid structures, this is the reason why CNN works well for image classification problems. The dropout layer is used to deactivate some of the neurons and while training, it reduces offer fitting of the model. We will then compile the model with the Adadelta optimizer.

Step 4: Train the model

The model.fit() function of Keras will start the training of the model. It takes the training data, validation data, epochs, and batch size.

It takes some time to train the model. After training, we save the weights and model definition in the ‘mnist.h5’ file.

Step 5: Evaluate the model

We have 10,000 images in our dataset which will be used to evaluate how good our model works. The testing data was not involved in the training of the data therefore, it is new data for our model. The MNIST dataset is well balanced so we can get around 99% accuracy.

Step 6: Create GUI to predict digits

Now for the GUI, we have created a new file in which we build an interactive window to draw digits on canvas and with a button, we can recognize the digit. The Tkinter library comes in the Python standard library. We have created a function predict_digit() that takes the image as input and then uses the trained model to predict the digit.

CONCLUSION:

Hence, we implemented MNIST Handwritten Character Detection using PyTorch, Keras and Tensorflow