KUMARAGURU
college of technology
character is life

ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

LABORATORY RECORD

U18AII4202 - Neural Networks and Deep Learning

2022-2023 (EVEN)

KUMARAGURU
college of technology
character is life

Certificate

This is to certify that it is a Bonafide record of practical work done by Sri/Kum. ____________________________________ bearing the Register No.______________________of____________year____________________ __________________branch in the ______________________________ Laboratory during the academic year ____________________ under my supervision.

Course Instructor HoD

INTERNAL ASSESSMENT

Ex. No		Title	Data Visualization and preprocessing (5)	Model Building & Evaluation (10)	Report (5)	Total (20)	Viva (10)	Signature
1		McCulloch Pitts and Perceptron						
2		Multilayer Perceptron						
3		Convolutional Neural Network						
4		Estimating depth and width of Neural Networks						
5		Data Augmentation						
6		Restricted Boltzmann Machine						
7		Autoencoders						
8		Computer Vision						
9		Bidirectional LSTM						
					Total			

Staff in Charge

Ex No:1	**McCulloch Pitt's and Perceptron**

Aim:

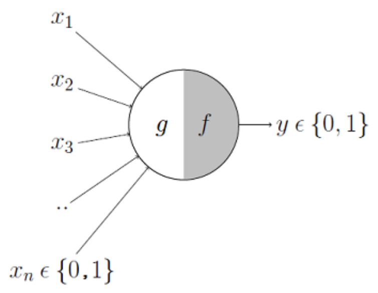
To represent the logic gates using MP neurons and Perceptron.

Introduction to AI neurons

An artificial neuron is a connection point in an artificial neural network. Artificial neural networks, like the human body's biological neural network, have a layered architecture and each network node (connection point) has the capability to process input and forward output to other nodes in the network. In both artificial and biological architectures, the nodes are called neurons and the connections are characterized by synaptic weights, which represent the significance of the connection. As new data is received and processed, the synaptic weights change, and this is how learning occurs.

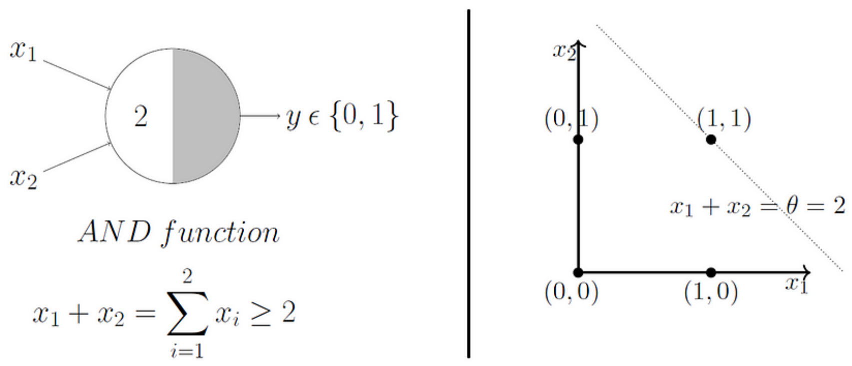
MuCulloch Pitts:

The first computational model of a neuron was proposed by Warren MuCulloch (neuroscientist) and Walter Pitts (logician) in 1943.



It is divided into 2 parts. The first part, ***g*** takes an input , performs an aggregation, and based on the aggregated value the second part, ***f*** makes a decision.

AND Function:

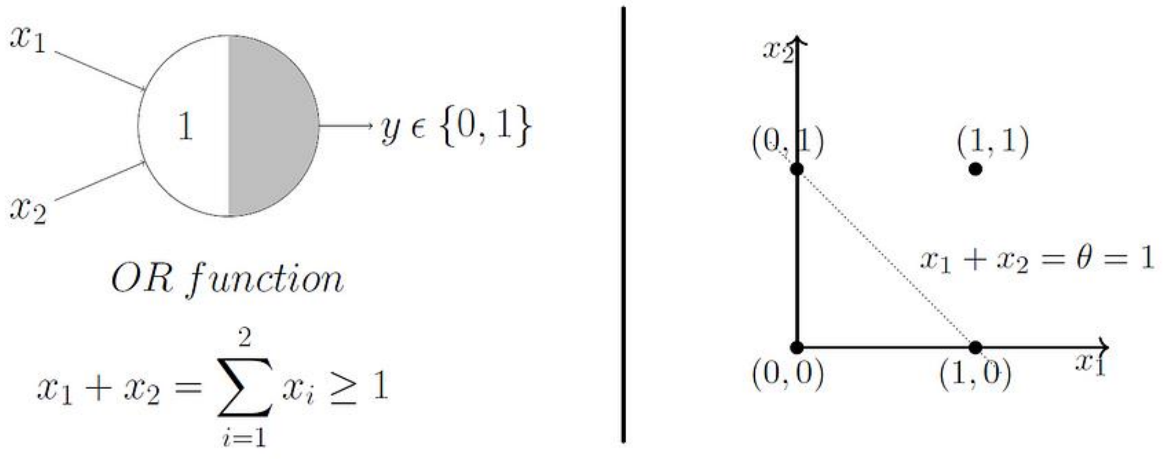


An AND function neuron would only fire when ALL the inputs are ON.

INPUTS		**OUTPUTS**
x1	**x2**	
0	0	0
0	1	0
1	0	0
1	1	1

```
import numpy as np
x1 = np.array([0,0,0,0,1,1,1,1])
x2 = np.array([0,0,1,1,0,0,1,1])
x3 = np.array([0,1,0,1,0,1,0,1])
f = x1+x2+x3
max_value = 0
l=list()
for (a, b, c) in zip(x1, x2, x3):
    y = a and b and c
    print(a, b, c, 'y = ', y)
    if y > max_value:
        max_value = a + b + c
print("Theta = ", max_value)
```

OR Function:

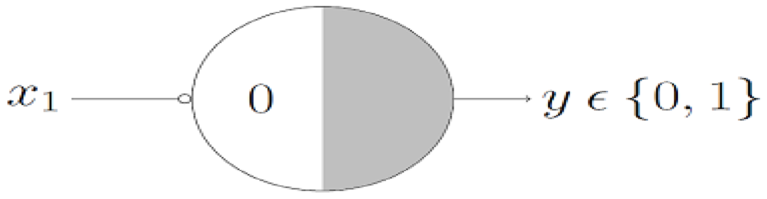


An OR function neuron would fire if ANY of the inputs is ON.

INPUTS		OUTPUTS
x1	**x2**	
0	0	0
0	1	1
1	0	1
1	1	1

```
import numpy as np
x1 = np.array([0,0,0,0,1,1,1,1])
x2 = np.array([0,0,1,1,0,0,1,1])
x3 = np.array([0,1,0,1,0,1,0,1])
f = x1+x2+x3
max_value = 0
l=list()
for (a, b, c) in zip(x1, x2, x3):
        y = a or b or c
        print(a, b, c, 'y = ', y)
        if y > max_value:
                max_value = a + b + c
print("Theta = ", max_value)
```

NOT Function:

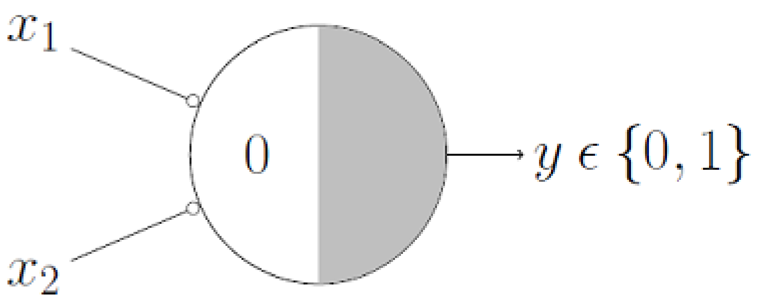


For a NOT neuron, 1 outputs 0 and 0 outputs 1. Take the input as an inhibitory input and set the thresholding parameter to 0.

INPUTS	OUTPUTS
0	1
1	0

```
import numpy as np
x = np.array([0,1])
y = np.array([1,0])
theta = 1
f = x
y_pred = (f < theta).astype(int)
if np.all(y == y_pred):
        print(f“f = {f}”)
        print(f “y_pred = {y_pred}”)
        print(f “threshold = {theta}”)
```

NOR Function:

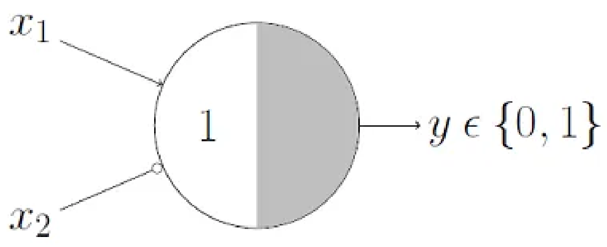


For a NOR neuron to fire, ALL the inputs to be 0 so the thresholding parameter should also be 0 and take them all as inhibitory input.

INPUTS		**OUTPUTS**
x1	**x2**	
0	0	1
0	1	0
1	0	0
1	1	0

```
def mp_neuron_nor_gate(x1, x2):
    threshold = 2
    if x1 == 0 and x2 == 0:
        return 1
   else:
       return 0
print(mp_neuron_nor_gate(0,0))
print(mp_neuron_nor_gate(0,1))
print(mp_neuron_nor_gate(1,0))
print(mp_neuron_nor_gate(1,1))
```

X1 AND !X2 :



$x_1\ AND\ !x_2{}^*$

Inhibitory input is x_2 so whenever x_2 is 1, the output will be 0. x_1 AND !x_2 would output 1 only when x_1 is 1 and x_2 is 0 so it is obvious that the threshold parameter should be 1.

g(x) i.e., $x_1 + x_2$ would be ≥ 1 in only 3 cases:

Case 1: when x_1 is 1 and x_2 is 0

Case 2: when x_1 is 1 and x_2 is 1

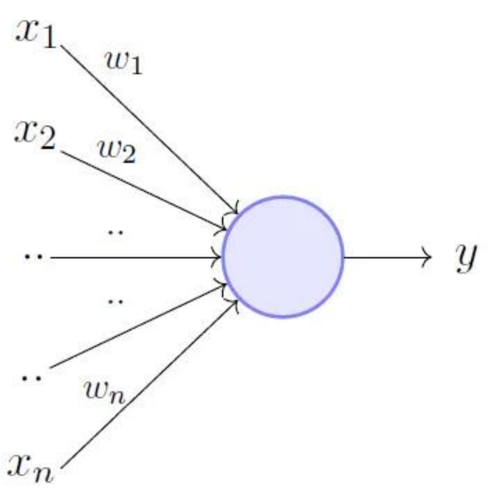
Case 3: when x_1 is 0 and x_2 is 1

INPUTS			OUTPUTS
x1	x2	!x2	x1 AND !x2
0	0	1	0
0	1	0	0
1	0	1	1
1	1	0	0

```
import numpy as np
x1 = np.array([0,0,1,1])
x2 = np.array([0,1,0,1])
f = x1 + x2 max_value = 0
l=list()
for (a, b) in zip(x1, x2):
    y = a and (not b)
    if y == True:
        y = 1
    else:
        y = 0
    print(a, b, 'y = ', y)
    if y > max_value:
    max_value = a + b
print("Theta = ", max_value)
```

PERCEPTRON:

One major limitation of MP neurons is that they can only model linearly separable functions. This means that they can only learn patterns that can be separated by a single line or hyperplane in the input space and also MP neurons are binary, which means that they can only output a 0 or 1. In contrast, perceptrons are able to learn more complex patterns by using multiple linear decision boundaries. This makes perceptrons more powerful than MP neurons in terms of their ability to learn complex functions. The perceptrons can also output continuous values between 0 and 1, allowing for a more fine-grained representation of output values.

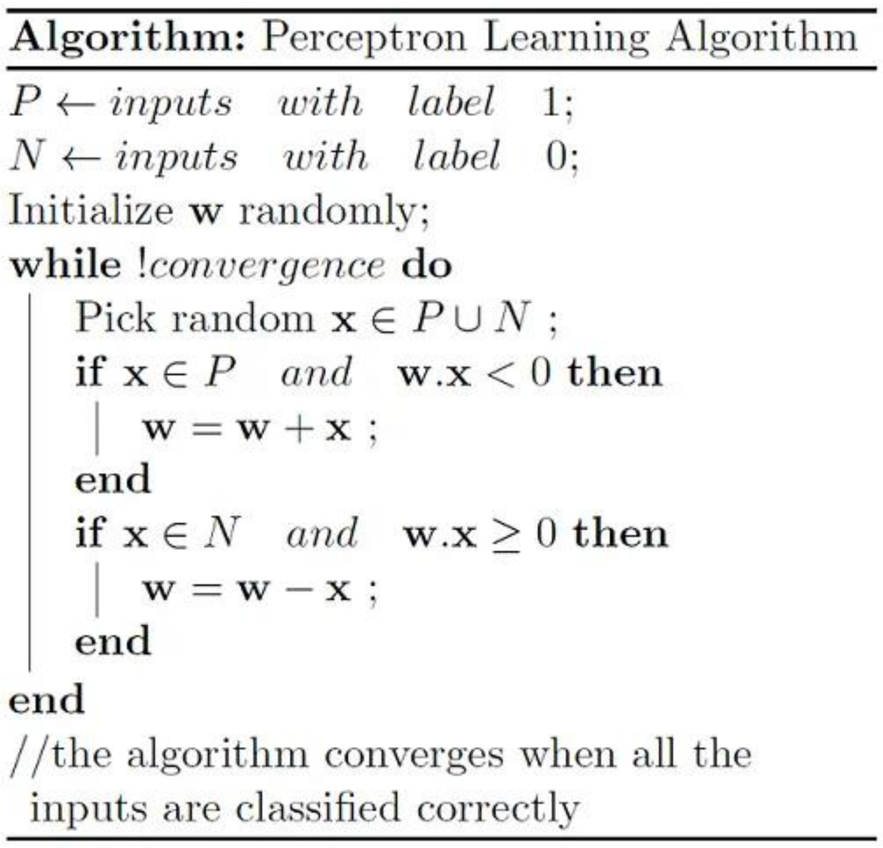


$$
\begin{aligned}
y &= 1 \quad if \sum_{i=1}^{n} w_i * x_i \geq \theta \\
&= 0 \quad if \sum_{i=1}^{n} w_i * x_i < \theta
\end{aligned}
$$

Rewriting the above,

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

1. **Input Values or One Input Layer**
2. **Weights and Bias**
3. **Net sum**
4. **Activation function**



Algorithm: Perceptron Learning Algorithm

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

AND LOGIC:

```
import numpy as np
x1 = np.array([0,0,1,1])
x2 = np.array([0,1,0,1])
y = np.array([0,0,0,1])
w1 = 1
w2 = 1
theta = 2                                        #threshold
f = x1*w1+x2*w2
y_pred = (f>=theta).astype(int)
if np.all(y == y_pred):
        print(f"f = {f}\ny_pred = {y_pred}\ncorrect weights and threshold")
        print("w1=",w1, “\n”,"w2=",w2)
        print("threshold=",theta)
else:
        print("change the weights/threshold")
```

OR LOGIC:

```
import numpy as np
x1 = np.array([0,0,1,1])
x2 = np.array([0,1,0,1])
y = np.array([0,1,1,1])
w1 = 1
w2 = 1
```

```
theta = 1                                    #threshold
f = x1*w1+x2*w2
y_pred = (f>=theta).astype(int)
if np.all(y == y_pred):
        print(f"f = {f}\ny_pred = {y_pred}\ncorrect weights and threshold")
        print("w1=",w1, “\n”,"w2=",w2)
        print("threshold=",theta)
else:
        print("change the weights/threshold")
```

NOR LOGIC:

```
import numpy as np
x1 = np.array([0,0,1,1])
x2 = np.array([0,1,0,1])
y = np.array([1,0,0,0])
w1 = w2 = 1
theta = 0                                    #threshold
f = x1*w1+x2*w2
y_pred = (f>=theta).astype(int)
if np.all(y == y_pred):
        print(f"f = {f}\ny_pred = {y_pred}\ncorrect weights and threshold")
        print("w1=",w1, “\n”,"w2=",w2) print("threshold=",theta)
else:
        print("change the weights/threshold")
```

NAND LOGIC:

```
import numpy as np
x1 = np.array([0,0,1,1])
x2 = np.array([0,1,0,1])
y = np.array([1,1,1,0])
w1 = w2 = 1
theta = 1                                   #threshold
f = x1*w1+x2*w2
y_pred = (f>=theta).astype(int)
if np.all(y == y_pred):
        print(f"f = {f}\ny_pred = {y_pred}\ncorrect weights and threshold")
        print("w1=",w1, “\n”,"w2=",w2, “\n”, “threshold =”, theta)
else:
        print("change the weights/threshold")
```

Conclusion: The logic gates using MP neurons and Perceptron have been implemented in python.

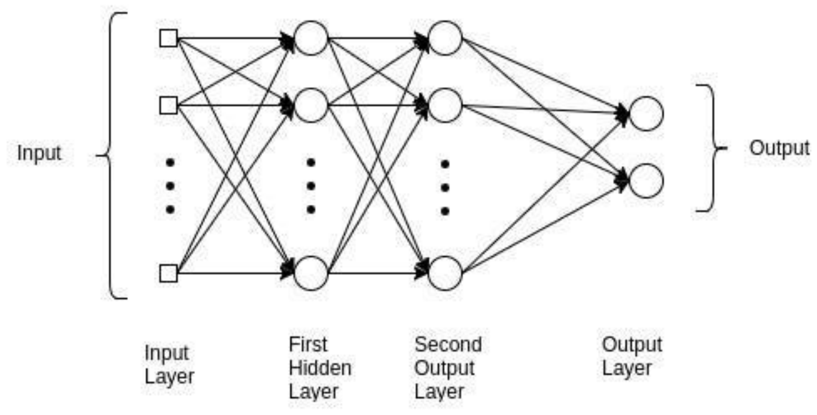
Ex : 2	Implementing Multilayer Perceptron

Aim:

To implement Multilayer Perceptron algorithm in MNIST dataset using back propagation and gradient descent.

Multilayer Perceptron (MLP):

Multi-layer perception is also known as MLP. It has fully connected dense layers, which transform any input dimension to the desired dimension. It is substantially formed from multiple layers of the perceptron.



Layers in MLP:

A multilayer perceptron (MLP) is a feed forward artificial neural network that generates a set of outputs from a set of inputs. It consists of three types of layers – the input layer, output layer and hidden layer.

Input layer: This is the initial layer of the network which takes in an input which will be used to produce an output.

Hidden Layer(s): The network needs to have at least one hidden layer. The hidden layer(s) perform computations and operations on the input data to produce something meaningful.

Output Layer: The neurons in this layer display a meaningful output.

Since MLPs are fully connected, each node in one layer connects with a certain weight to every node in the following layer. The connections between the layers are assigned weights. The weight of a connection specifies its importance.

Except for the input nodes, each node is a neuron that uses a nonlinear activation function. If a multilayer perceptron has a linear activation function in all neurons, that is, a linear function that maps the weighted inputs to the output of each neuron, then linear algebra shows that any number of layers can be reduced to a two-layer input-output model.

Back-propagation:

Back-propagation is a technique used to optimize the weights of an MLP using the outputs as inputs. In a conventional MLP, random weights are assigned to all the connections. These random weights propagate values through the network to produce the actual output.

The difference between the actual value and the expected value is called error. Back-propagation refers to the process of sending this error back through the network, readjusting the weights automatically so that eventually, the error between the actual and expected output is minimized. In this way, the output of the current iteration becomes the input and affects the next output. This is repeated until the correct output is produced. The weights at the end of the process would be the ones on which the neural network works correctly.

Gradient Descent:

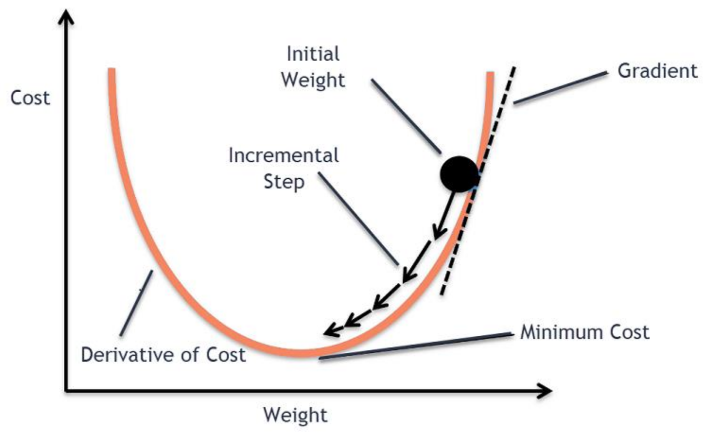
Gradient Descent is one of the most commonly used optimization algorithms to train machine learning models by means of minimizing errors between actual and expected results. Optimization is the task of minimizing the cost function parameterized by the model's parameters. The main objective of using a gradient descent algorithm is to minimize the cost function using iteration. It works by iteratively adjusting the weights or parameters of the model in the direction of the negative gradient of the cost function, until the minimum of the cost function is reached.

Hypothesis: $h_\theta(x) = \theta_0 + \theta_1 x$

Parameters: θ_0, θ_1

Cost function: $J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$

Goal: minimize $J(\theta_0, \theta_1)$



Dataset

Source: MNIST dataset

Size: 60000

Shape: (28, 28)

Code:

```
import tensorflow
from tensorflow import keras
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense,Flatten
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
import os
```

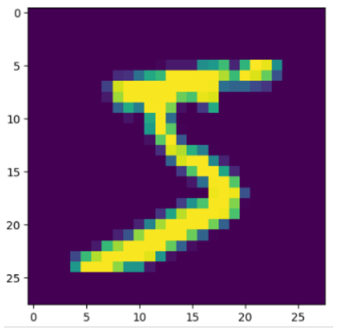
```
from keras.datasets import mnist
from keras.layers import *
from keras.models import *
from time import time
(X_train, y_train), (X_test, y_test) = keras.datasets.mnist.load_data()
X_train.shape
y_train
plt.imshow(X_train[0])
X_train[0]
X_train=X_train/ 255
X_test = X_test/255
model = Sequential()
model.add(Flatten(input_shape=(28,28)))
model.add(Dense(128,activation='relu'))
model.add(Dense(10,activation='softmax'))
model.summary()
model.compile(optimizer='adam',loss='sparse_categorical_crossentropy',metrics=['accuracy'])
model.fit(X_train,y_train,epochs=10,validation_split=0.2)
results = model.evaluate(X_test,  y_test, verbose = 0)
print('test loss, test acc:', results)
```

Output:

#number of images, size of image in pixels of training data

```
(60000, 28, 28)
array([5, 0, 4, ..., 5, 6, 8], dtype = uint8)
```

#sample image of x_train[0]



#x_train[0] – Matrix representation

#Greyscale image pixel values range from 0 to 255. The smaller numbers closer to zero represent the darker shade while the larger numbers closer to 255 represent the lighter or the white shade. In this image (x_train[0]), the highest is 255.

```
array([[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [ 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0], [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [ 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 3, 18, 18, 18, 126, 136, 175, 26, 166, 255, 247, 127, 0, 0, 0, 0], [ 0, 0, 0, 0, 0, 0, 0, 0, 30,
36, 94, 154, 170, 253, 253, 253, 253, 253, 225, 172, 253, 242, 195, 64, 0, 0, 0, 0], [ 0, 0, 0, 0, 0, 0,
0, 49, 238, 253, 253, 253, 253, 253, 253, 253, 253, 251, 93, 82, 82, 56, 39, 0, 0, 0, 0, 0], [ 0, 0, 0,
0, 0, 0, 0, 18, 219, 253, 253, 253, 253, 253, 198, 182, 247, 241, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [ 0, 0, 0,
0, 0, 0, 0, 0, 80, 156, 107, 253, 253, 205, 11, 0, 43, 154, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [ 0, 0, 0, 0, 0, 0,
0, 0, 0, 14, 1, 154, 253, 90, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 139,
253, 190, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 11, 190, 253, 70, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 35, 241, 225, 160, 108, 1, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0], [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 81, 240, 253, 253, 119, 25, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 45, 186, 253, 253, 150, 27, 0, 0, 0, 0, 0, 0, 0, 0], [ 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 16, 93, 252, 253, 187, 0, 0, 0, 0, 0, 0, 0, 0], [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 249, 253, 249, 64, 0, 0, 0, 0, 0, 0, 0], [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 46, 130,
183, 253, 253, 207, 2, 0, 0, 0, 0, 0, 0, 0], [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 39, 148, 229, 253, 253,
253, 250, 182, 0, 0, 0, 0, 0, 0, 0, 0], [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 24, 114, 221, 253, 253, 253, 253,
201, 78, 0, 0, 0, 0, 0, 0, 0, 0, 0], [ 0, 0, 0, 0, 0, 0, 0, 0, 23, 66, 213, 253, 253, 253, 253, 198, 81, 2,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [ 0, 0, 0, 0, 0, 0, 18, 171, 219, 253, 253, 253, 253, 195, 80, 9, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0], [ 0, 0, 0, 0, 55, 172, 226, 253, 253, 253, 253, 244, 133, 11, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0], [ 0, 0, 0, 0, 136, 253, 253, 253, 212, 135, 132, 16, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0], [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [ 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]], dtype=uint8)
```

#model summary

```
Model: "sequential_34"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_2 (Flatten)         (None, 784)               0

 dense_100 (Dense)           (None, 128)               100480

 dense_101 (Dense)           (None, 10)                1290

=================================================================
Total params: 101,770
Trainable params: 101,770
Non-trainable params: 0
_________________________________________________________________
```

#model fitting (no. of epochs = 10)

```
Epoch 1/10
1500/1500 [=============] - 4s 3ms/step - loss: 0.1209 - val_loss: 0.1073
Epoch 2/10
1500/1500 [=============] - 4s 3ms/step - loss: 0.0835 - val_loss: 0.0937
```

```
Epoch 3/10
1500/1500 [=============] - 5s 3ms/step - loss: 0.0627 - val_loss: 0.0873
Epoch 4/10
1500/1500 [=============] - 4s 3ms/step - loss: 0.0469 - val_loss: 0.0868
Epoch 5/10
1500/1500 [=============] - 4s 3ms/step - loss: 0.0375 - val_loss: 0.0809
Epoch 6/10
1500/1500 [=============] - 5s 3ms/step - loss: 0.0296 - val_loss: 0.0854
Epoch 7/10
1500/1500 [=============] - 4s 3ms/step - loss: 0.0228 - val_loss: 0.0816
Epoch 8/10
1500/1500 [=============] - 5s 3ms/step - loss: 0.0179 - val_loss: 0.0854
Epoch 9/10
1500/1500 [=============] - 4s 3ms/step - loss: 0.0149 - val_loss: 0.0906
Epoch 10/10
1500/1500 [=============] - 4s 3ms/step - loss: 0.0132 - val_loss: 0.1099


test loss, test acc: [0.10685475170612335, 0.9751999974250793]
```

Learning rate:

The default learning rate of Adam optimizer is 0.001.

Model Evaluation

Loss: 0.1068

Accuracy: 97.52 %

Conclusion:

Multilayer Perceptron has been implemented for the MNIST dataset, where backpropagation and gradient descent algorithm are applied for optimizing the weights and bias for better accuracy. The built model yields 97.52% accuracy for 10 epochs. The accuracy can be improved by increasing the epochs.

Ex No. 3	Convolutional Neural Network

Aim:

To implement Convolutional Neural Network model to classify digits.

Introduction:

Convolutional Neural Network is a type of artificial neural network commonly used for image and video processing tasks such as image classification, object detection, and segmentation.

CNNs are designed to automatically learn and extract features from input data by applying a series of filters or kernels to the input image. These filters detect specific patterns or features in the input image, such as edges, corners, or textures. The outputs of these filters are then combined and passed through additional layers of the network to gradually extract higher-level features.

CNNs are particularly effective for image processing tasks because they can leverage the spatial structure and correlations between pixels in an image. This allows them to learn complex patterns and features that may not be apparent to human observers.

Overall, CNNs have become a widely used and powerful tool in the field of machine learning, particularly in computer vision applications.

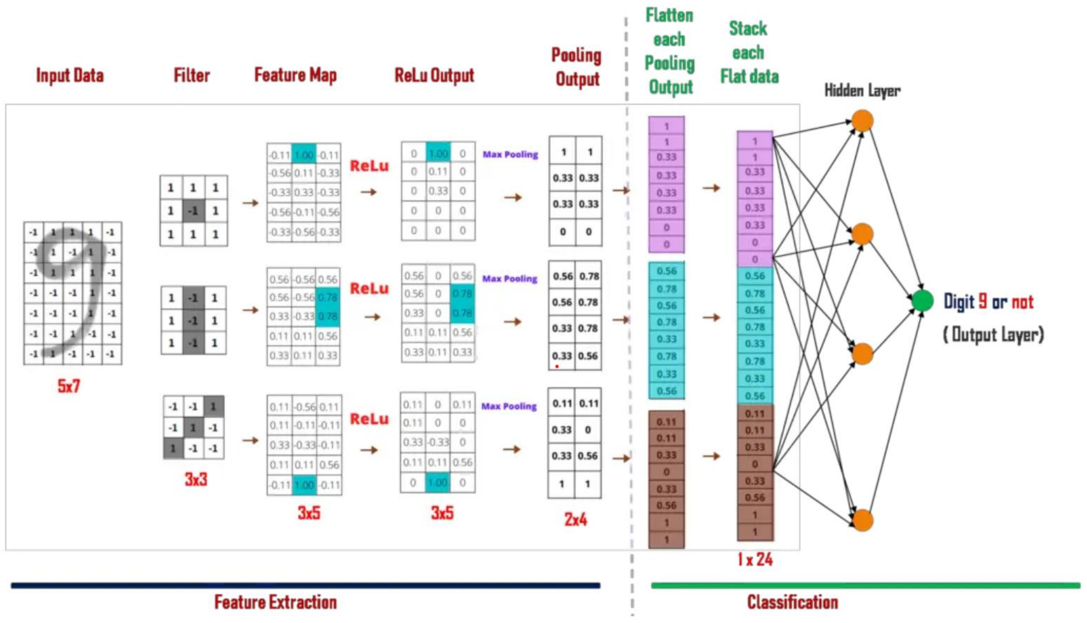
Basic Architecture:

There are two main parts to a CNN architecture.

1. Feature Extraction:
   Feature extraction in a CNN involves convolution layers that use filters to detect patterns and features in input data. These layers slide over the input, computing dot products to generate feature maps. Pooling layers then reduce spatial dimensions while retaining important information. By stacking multiple convolution and pooling layers, CNNs learn increasingly complex and abstract features. The output of feature extraction is then fed into fully connected layers for classification or regression. Additional components like skip connections and batch normalization can enhance the network's performance.

2. Classification:
   Classification in a CNN architecture involves the use of fully connected layers. These layers receive the output of the feature extraction stage and perform the task of assigning labels or classes to the input data. Each neuron in the fully connected layers is connected to all neurons in the previous layer, allowing for the learning of complex relationships between features. The final layer typically employs a softmax activation function to produce a probability distribution over the possible classes. The class with the highest probability is chosen as the predicted label for the input data. During

training, the model adjusts its weights through backpropagation to minimize the classification error.



The different layers:

1. Convolution layer:
   Convolutional layers in CNNs apply filters or kernels to the input data, allowing the network to automatically learn and extract relevant features from images. These filters slide over the input image, performing element-wise multiplications and summing the results to produce a feature map. By applying multiple filters, the network can learn to detect different types of features in the input image.
2. Pooling layer:
   Pooling layers in CNNs are used to downsample the output feature maps from convolutional layers. They help to reduce the spatial dimensionality of the feature maps while retaining the most important information. Common types of pooling layers include max pooling and average pooling, which take the maximum or average value within a fixed window, respectively.
3. Fully connected layer:
   A fully connected layer in a CNN is a traditional neural network layer where each neuron is connected to every neuron in the previous layer. It takes the flattened output from the previous convolutional and pooling layers and performs classification based on the learned features. The final output of the fully connected layer represents the predicted class probabilities for the input image.
4. Dropout:

Dropout is a regularization technique used in CNNs to prevent overfitting. It randomly drops out a fraction of the neurons in a layer during training, forcing the network to learn more robust and generalizable features. This helps to prevent the network from relying too heavily on any one set of features, and leads to improved performance on new, unseen data.

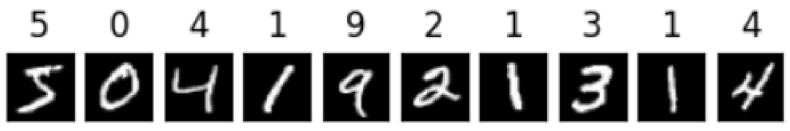
5. Activation Function:
   Activation functions in CNNs are applied to the output of each neuron in a layer to introduce non-linearity and improve the network's ability to model complex relationships in the input data. Popular activation functions include ReLU, sigmoid, and tanh. ReLU is the most used activation function in CNNs due to its simplicity and ability to prevent the vanishing gradient problem.

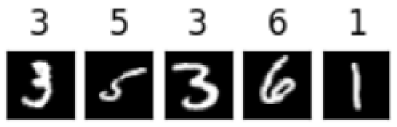
Code:

```
#importing libraries

from numpy import unique, argmax
from tensorflow.keras.datasets.mnist import load_data
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.layers import MaxPool2D
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Dropout
from tensorflow.keras.utils import plot_model
from matplotlib import pyplot
import matplotlib.pyplot as plt
import numpy as np


# train n test
(x_train, y_train), (x_test, y_test) = load_data()
# to reshape the train n test data
x_train = x_train.reshape((x_train.shape[0], x_train.shape[1],
x_train.shape[2], 1))
x_test = x_test.reshape((x_test.shape[0], x_test.shape[1], x_test.shape[2], 1))
# normalizing pixels
x_train = x_train.astype('float32')/255.0
x_test = x_test.astype('float32')/255.0
#plotting
fig = plt.figure(figsize=(5,3))
for i in range(15):
ax = fig.add_subplot(2, 10, i+1, xticks=[], yticks=[])
ax.imshow(np.squeeze(x_train[i]), cmap='gray')
ax.set_title(y_train[i])

# shape of input images
```

```
img_shape = x_train.shape[1:]
print(img_shape)
# defining the model
model = Sequential()
model.add(Conv2D(32, (3,3), activation='relu', input_shape=img_shape))
model.add(MaxPool2D(2,2))
model.add(Conv2D(48, (3,3), activation='relu'))
model.add(MaxPool2D(2,2))
model.add(Dropout(0.5))
model.add(Flatten())
model.add(Dense(500, activation='relu'))
model.add(Dense(10, activation='softmax'))
model.summary()
plot_model(model, 'model.jpg', show_shapes=True)
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
          metrics=['accuracy'])
x = model.fit(x_train, y_train, epochs=10, batch_size=128, verbose=2, validation_split=0.1)

loss, accuracy = model.evaluate(x_test, y_test, verbose=0)
print(f'Accuracy: {accuracy*100}')

image = x_train[5]

# predict
plt.imshow(np.squeeze(image), cmap='gray')
plt.show()

image = image.reshape(1, image.shape[0], image.shape[1], image.shape[2])
p = model.predict([image])
print('Predicted: {}'.format(argmax(p)))
```

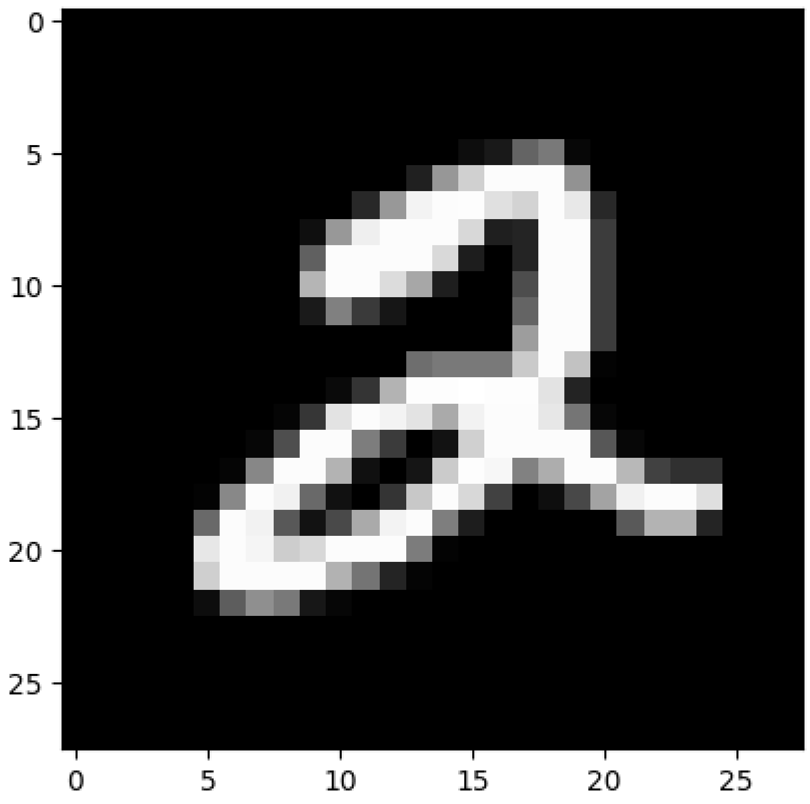
Sample images in the training dataset:





```
Epoch 1/10
422/422 - 13s - loss: 0.2403 - accuracy: 0.9256 - val_loss: 0.0565 - val_accuracy: 0.9837 - 13s/epoch - 31ms/step
Epoch 2/10
422/422 - 2s - loss: 0.0784 - accuracy: 0.9755 - val_loss: 0.0404 - val_accuracy: 0.9890 - 2s/epoch - 4ms/step
Epoch 3/10
422/422 - 2s - loss: 0.0581 - accuracy: 0.9820 - val_loss: 0.0349 - val_accuracy: 0.9903 - 2s/epoch - 6ms/step
Epoch 4/10
422/422 - 2s - loss: 0.0482 - accuracy: 0.9844 - val_loss: 0.0300 - val_accuracy: 0.9913 - 2s/epoch - 5ms/step
Epoch 5/10
422/422 - 2s - loss: 0.0417 - accuracy: 0.9869 - val_loss: 0.0309 - val_accuracy: 0.9922 - 2s/epoch - 5ms/step
Epoch 6/10
422/422 - 2s - loss: 0.0344 - accuracy: 0.9890 - val_loss: 0.0270 - val_accuracy: 0.9923 - 2s/epoch - 4ms/step
Epoch 7/10
422/422 - 2s - loss: 0.0313 - accuracy: 0.9899 - val_loss: 0.0321 - val_accuracy: 0.9912 - 2s/epoch - 4ms/step
Epoch 8/10
422/422 - 2s - loss: 0.0283 - accuracy: 0.9905 - val_loss: 0.0266 - val_accuracy: 0.9925 - 2s/epoch - 4ms/step
Epoch 9/10
422/422 - 2s - loss: 0.0260 - accuracy: 0.9914 - val_loss: 0.0274 - val_accuracy: 0.9932 - 2s/epoch - 4ms/step
Epoch 10/10
422/422 - 2s - loss: 0.0255 - accuracy: 0.9911 - val_loss: 0.0265 - val_accuracy: 0.9932 - 2s/epoch - 4ms/step
```

Test image



Output:

```
1/1 [==============================] - 0s 227ms/step
Predicted: 2
```

Result:

The CNN model to classify given digit is built and the model is evaluated, and it yields an accuracy of 99.29%.

Ex 04	Estimating depth and width of Neural Networks

Aim:

To estimate the depth and width of Neural Networks using GridSearchCV.

Theory:

Artificial neural networks have two main hyperparameters that control the architecture or topology of the network: the number of layers and the number of nodes in each hidden layer. Depth is the number of layers including output layer but not input layer and width is the maximum number of nodes in a layer. A node, also called a neuron or Perceptron, is a computational unit that has one or more weighted input connections, a transfer function that combines the inputs in some way, and an output connection. Nodes are then organized into layers to comprise a network. A single-layer artificial neural network, also called a single-layer, has a single layer of nodes, as its name suggests. Each node in the single layer connects directly to an input variable and contributes to an output variable. A single-layer network can be extended to a multiple-layer network, referred to as a Multilayer Perceptron. A Multilayer Perceptron, or MLP for short, is an artificial neural network with more than a single layer. It has an input layer that connects to the input variables, one or more hidden layers, and an output layer that produces the output variables.

Input Layer: Input variables, sometimes called the visible layer.

Hidden Layers: Layers of nodes between the input and output layers. There may be one or more of these layers.

Output Layer: A layer of nodes that produce the output variables.

The terms used to describe the shape and capability of a neural network are as follows:

Size: The number of nodes in the model.

Width: The number of nodes in a specific layer.

Depth: The number of layers in a neural network.

Capacity: The type or structure of functions that can be learned by a network configuration. Sometimes called “representational capacity“.

Architecture: The specific arrangement of the layers and nodes in the network.

A single-layer neural network can only be used to represent linearly separable functions. This means very simple problems where, say, the two classes in a classification problem can be neatly separated by a line. If the problem is relatively simple, perhaps a single layer network would be sufficient. Most problems that are interested in solving are not linearly separable. A Multilayer Perceptron can be used to represent convex regions. This means that in effect, they can learn to draw shapes around examples in some high-dimensional space that can separate and classify them, overcoming the limitation of linear separability.

Five approaches to solving this problem

1) Experimentation

2) Intuition

3) Go For Depth

4) Borrow Ideas

5) Search

Some popular search strategies include:

Random: Try random configurations of layers and nodes per layer.

Grid: Try a systematic search across the number of layers and nodes per layer.

Heuristic: Try a directed search across configurations such as a genetic algorithm or Bayesian optimization.

Exhaustive: Try all combinations of layers and the number of nodes; it might be feasible for small networks and datasets.

Grid Search:

GridSearchCV is the process of performing hyperparameter tuning in order to determine the optimal values for a given model. As mentioned above, the performance of a model significantly depends on the value of hyperparameters. Note that there is no way to know in advance the best values for hyperparameters so ideally, need to try all possible values to know the optimal values. Doing this manually could take a considerable amount of time and resources and thus we use GridSearchCV to automate the tuning of hyperparameters. GridSearchCV is a function that comes in Scikit-learn's(or SK-learn) model_selection package. So, an important point here to note is that need to have the Scikit learn library installed on the computer. This function helps to loop through predefined hyperparameters and fit estimator (model) on training set.

Dataset:

Gender Classification

Source:

https://www.kaggle.com/datasets/elakiricoder/gender-classification-dataset

Dataset Description:

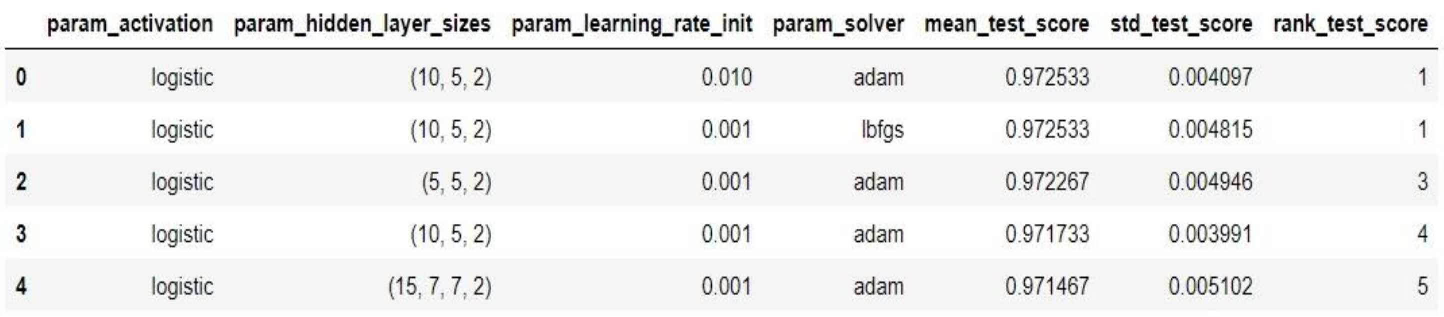
Sl. No	Variable	Type
1	long_hair	Categorical
2	forehead_width	Numerical
3	forehead_height	Numerical
4	nose_wide	Categorical
5	nose_long	Categorical
6	lips_thin	Categorical
7	distance_nose_to_lip_long	Categorical
8	gender	Categorical

Implementation:

```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
dataset = pd.read_csv(r"C:\Users\Siddhartha Devan V\Downloads\gender
classification\gender_classification_v7.csv")
dataset.isnull().sum()
dataset.gender.value_counts()
dataset.gender.replace({"Male":1, "Female":0}, inplace = True)
dataset.corr()
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
dataset_scaled = dataset.copy()
cols_to_scale = ["forehead_width_cm", "forehead_height_cm"]
dataset_scaled[cols_to_scale] = scaler.fit_transform(dataset_scaled[cols_to_scale])
dataset_scaled.head()
Y_s = dataset_scaled["gender"]
dataset_scaled.drop("gender",axis = "columns", inplace = True)
X_scaled = dataset_scaled.copy()
X_scaled.head()
X_train_s, X_test_s, Y_train_s, Y_test_s = train_test_split(X_scaled, Y_s, random_state = 0, stratify
= Y, test_size = 0.25)
clf2 = MLPClassifier(solver='lbfgs', alpha=0.001, hidden_layer_sizes=(5,5,2), max_iter = 100,
activation = 'relu')
clf2.fit(X_train_s, Y_train_s)

Y_pred_s = clf2.predict(X_test_s)
Y_pred_train_s = clf2.predict(X_train_s)
```

```
print(Y_pred_s[:100])
print(Y_pred_train_s[:100])
print(accuracy_score(Y_test_s, Y_pred_s))
print(accuracy_score(Y_train_s, Y_pred_train_s))
print(precision_score(Y_test_s, Y_pred_s))
print(recall_score(Y_test_s, Y_pred_s))
print(f1_score(Y_test_s, Y_pred_s))
from sklearn import metrics
y_pred = clf2.predict(X_test_s)
fpr, tpr, _ = metrics.roc_curve(Y_test_s, Y_pred_s)
auc = metrics.roc_auc_score(Y_test_s, Y_pred_s)
print('area under the curve is ',auc)
plt.plot(fpr,tpr,label="3 neighbours , AUC="+str(auc))
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.title('auc_roc_curve')
plt.show()
from sklearn.model_selection import GridSearchCV
mlp_gs = MLPClassifier()
parameters = {
   'hidden_layer_sizes' : [(5,5,2), (15, 7,7,2), (10, 5, 2), (5, 2)],
   'activation': ['relu', 'logistic', 'tanh', 'identity'],
   'solver' : ['sgd', 'adam', 'lbfgs'],
   'learning_rate_init' : [0.01, 0.001, 0.05]
}
clf = GridSearchCV(mlp_gs, parameters, cv = 5)
clf.fit(X_train_s, Y_train_s)
print(clf.best_params_)
res = pd.DataFrame(clf.cv_results_)
res.head()
res = res.sort_values('rank_test_score')
res.to_csv('grid_CV_results.csv')
res[['param_activation', 'param_hidden_layer_sizes',
      'param_learning_rate_init', 'param_solver','mean_test_score',
      'std_test_score', 'rank_test_score' ]].head()
clf_best= MLPClassifier(solver='adam', hidden_layer_sizes=(10,5,2), activation = 'logistic',
learning_rate_init= 0.01)
clf_best.fit(X_train_s, Y_train_s)
Y_pred_s = clf_best.predict(X_test_s)
Y_pred_train_s = clf_best.predict(X_train_s)
print(accuracy_score(Y_test_s, Y_pred_s))
print(accuracy_score(Y_train_s, Y_pred_train_s))
print(precision_score(Y_test_s, Y_pred_s))
print(recall_score(Y_test_s, Y_pred_s))
print(f1_score(Y_test_s, Y_pred_s))
y_pred = clf_best.predict(X_test_s)
fpr, tpr, _ = metrics.roc_curve(Y_test_s, Y_pred_s)
auc = metrics.roc_auc_score(Y_test_s, Y_pred_best)
print('area under the curve is ',auc)
plt.plot(fpr,tpr,label="3 neighbours , AUC="+str(auc))
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.title('auc_roc_curve')
plt.show()
```

Results:



	param_activation	param_hidden_layer_sizes	param_learning_rate_init	param_solver	mean_test_score	std_test_score	rank_test_score
0	logistic	(10, 5, 2)	0.010	adam	0.972533	0.004097	1
1	logistic	(10, 5, 2)	0.001	lbfgs	0.972533	0.004815	1
2	logistic	(5, 5, 2)	0.001	adam	0.972267	0.004946	3
3	logistic	(10, 5, 2)	0.001	adam	0.971733	0.003991	4
4	logistic	(15, 7, 7, 2)	0.001	adam	0.971467	0.005102	5

Initial Parameters:

Parameters	
solver	lbfgs
learning_rate_init	0.5
hidden_layer_sizes	{13,7,2}
activation	tanh

Training accuracy	0.9834666666666667
Test accuracy	0.9592326139088729
Precision	0.9783333333333334
Recall	0.9392
F1	0.9583673469387755
Area under the curve	0.9608242811501597

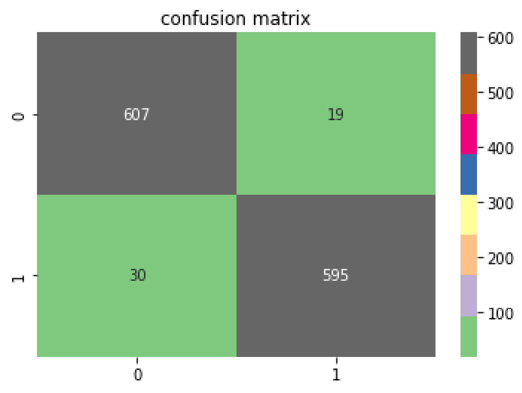
Best Parameters:

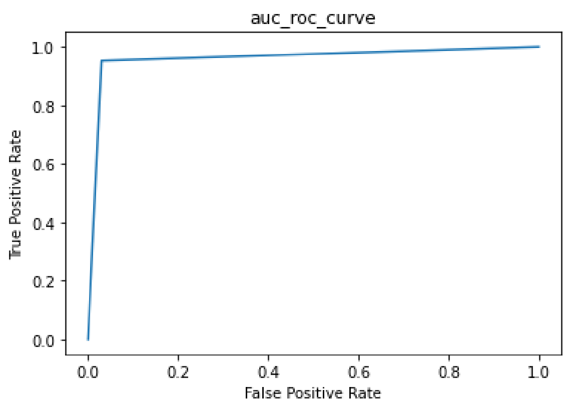
Parameters	
solver	adam
learning_rate_init	0.01
hidden_layer_sizes	{10,5,2}
activation	logistic

Training accuracy	0.9714666666666667
Test accuracy	0.9640287769784173
Precision	0.963258785942492
Recall	0.9648
F1	0.9640287769784173
Area under the curve	0.9640242811501598

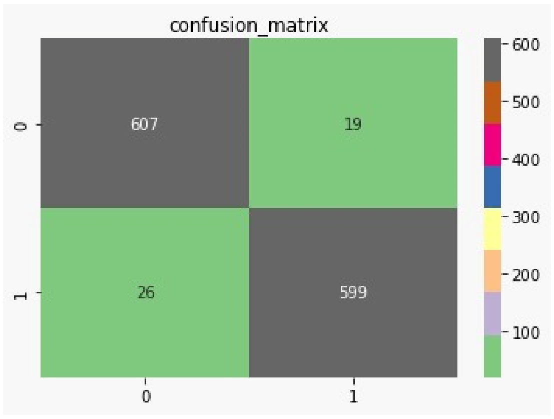
Visualizations:

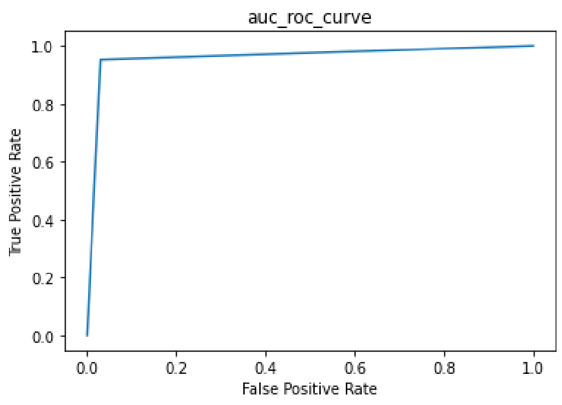
Before tuning:





After tuning:





Conclusion:

The width of the neural network is estimated as **{10,5,2}** and the depth is **3** with **Logistic** as activation function and **200** epochs for the experimented **Gender_classification** dataset.

Ex No : 5	Data Augmentation

Aim:

To implement Data Augmentation to the flower dataset to generalize the overfitting model.

Need for Regularisation:

Deep learning models are highly expensive and can learn complex patterns and representations from large amounts of data. However, they are also prone to overfitting, which occurs when a model becomes too complex and starts to memorize the training data rather than generalizing to new, unseen data.

Regularisation:

Regularization is a technique which makes slight modifications to the learning algorithm such that the model generalizes better. This in turn improves the model's performance on the unseen data as well.

Some common regularization techniques used in deep learning include:

- **L1 and L2 regularization:** These techniques add a penalty term to the loss function based on the magnitude of the weights in the model. L1 regularization encourages sparsity in the weights, while L2 regularization encourages smaller weight values.
- **Dropout:** This technique randomly drops out a fraction of the neurons in the model during training, forcing the remaining neurons to learn more robust representations.
- **Data augmentation:** This technique involves artificially increasing the size of the training data by applying transformations such as rotations, translations, and scaling to the input images.
- **Early stopping:** This technique stops the training process before the model overfits the training data by monitoring the validation loss and stopping when it starts to increase.
- **Batch normalization:** This technique involves normalizing the inputs to each layer of the network to have zero mean and unit variance. Batch normalization helps to stabilize the distribution of the inputs to each layer and improves the training speed and accuracy.
- **Multi-Task Learning:** In multitask learning, the network learns to share information between related tasks, which can improve the performance of each task and reduce the risk of overfitting.

Data Augmentation:

Data augmentation is a set of techniques to artificially increase the amount ofdata by gener ating new data points from existing data. This includes making small changes to data or using dee p learning models to generate new data points. For data augmentation, making simple alterations on visual data is popular Classic image processing activities for data augmentation are padding, random rotating, re-scaling, vertical and horizontal flipping, translation (image is moved along X, Y direction), cropping, zooming, darkening & brightening/colour modification, grayscaling, changing contrast, adding noise, random erasing.

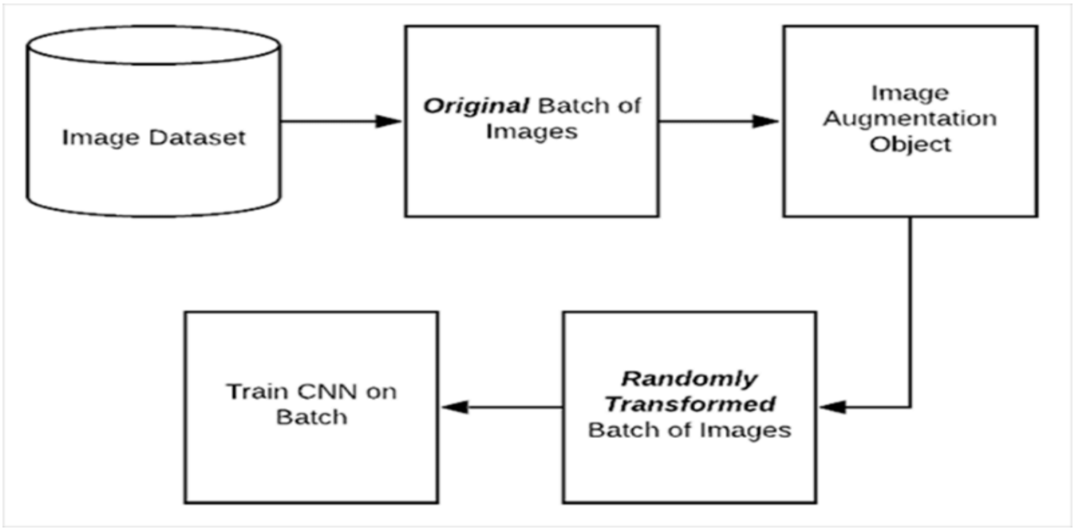


Image Dataset
Original Batch of Images
Image Augmentation Object
Randomly Transformed Batch of Images
Train CNN on Batch

Advanced models for data augmentation are

- Adversarial training/Adversarial machine learning
- Generative adversarial networks (GANs)
- Neural style transfer
- Reinforcement learning

Dataset

Source:
https://storage.googleapis.com/download.tensorflow.org/example_images/flower_photos.tgz

The flowers dataset has five classes.

- Daisy
- Dandelion
- Roses
- Sunflowers
- Tulips

Implementation

```
import matplotlib.pyplot as plt
import numpy as np
import cv2
import os
import PIL
import tensorflow as tf

from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.models import Sequential
import pathlib
dataset_url = "https://storage.googleapis.com/download.tensorflow.org/example_images/flower_
photos.tgz"
archive = tf.keras.utils.get_file(origin=dataset_url, extract=True)
data_dir = pathlib.Path(archive).with_suffix('')
list(data_dir.glob('*/*.jpg'))[:5]
image_count = len(list(data_dir.glob('*/*.jpg')))
print(image_count)
roses = list(data_dir.glob('roses/*'))
roses[:5]
tulips = list(data_dir.glob('tulips/*'))
PIL.Image.open(str(tulips[0]))

flowers_images_dict = {
    'roses': list(data_dir.glob('roses/*')),
    'daisy': list(data_dir.glob('daisy/*')),
    'dandelion': list(data_dir.glob('dandelion/*')),
    'sunflowers': list(data_dir.glob('sunflowers/*')),
    'tulips': list(data_dir.glob('tulips/*')),
}
flowers_labels_dict = {
    'roses': 0,
    'daisy': 1,
    'dandelion': 2,
    'sunflowers': 3,
    'tulips': 4,
}
img = cv2.imread(str(flowers_images_dict['roses'][0]))
img.shape
cv2.resize(img,(180,180)).shape
X, y = [], []

for flower_name, images in flowers_images_dict.items():
    for image in images:
        img = cv2.imread(str(image))
        resized_img = cv2.resize(img,(180,180))
        X.append(resized_img)
        y.append(flowers_labels_dict[flower_name])
X = np.array(X)
y = np.array(y)
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
X_train_scaled = X_train / 255
X_test_scaled = X_test / 255
num_classes = 5

model = Sequential([
  layers.Conv2D(16, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(32, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(64, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Flatten(),
  layers.Dense(128, activation='relu'),
  layers.Dense(num_classes)
])

model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

history=model.fit(X_train_scaled, y_train,validation_data=(X_test_scaled,y_test), epochs=30)
plt.figure(figsize=(10,8))
plt.plot(np.arange(0,30),history.history['accuracy'],label='Training_accuracy')
plt.plot(np.arange(0,30),history.history['loss'],label='Training_loss')
plt.plot(np.arange(0,30),history.history['val_accuracy'],label='Validation_accuracy')
plt.plot(np.arange(0,30),history.history['val_loss'],label='validation_loss')
plt.xlabel('#Epochs')
plt.ylabel('Percentage')
plt.legend()
plt.title('Training accuracy and Loss plot')
plt.show()
```

Improving test accuracy using Data Augmentation

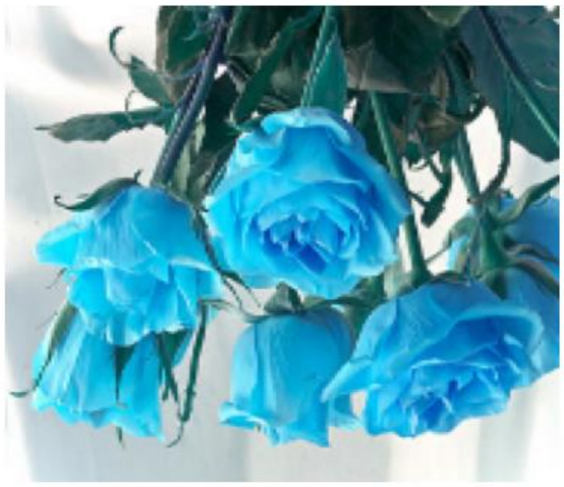
```
data_augmentation = keras.Sequential(
  [
    layers.experimental.preprocessing.RandomFlip("horizontal"),
    layers.experimental.preprocessing.RandomRotation(0.1),
    layers.experimental.preprocessing.RandomZoom(0.1),
  ]
)
plt.axis('off')
print("Image Dimensions were: {0}x{1}".format(X.shape[0], X.shape[1]))
plt.imshow(X[0])
plt.axis('off')
img=data_augmentation(X)[0].numpy().astype("uint8")
plt.imshow(img)
print("Resized Dimensions were: {0}x{1}".format(img.shape[0], img.shape[1]))
num_classes = 5
model = Sequential([
  data_augmentation,
  layers.Conv2D(16, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(32, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(64, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Dropout(0.2),
  layers.Flatten(),
  layers.Dense(128, activation='relu'),
  layers.Dense(num_classes)
])

model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

history = model.fit(X_train_scaled, y_train,validation_data=(X_test_scaled,y_test),epochs=30)
plt.figure(figsize=(10,8))
plt.plot(np.arange(0,30),history.history['accuracy'],label='Training_accuracy')
plt.plot(np.arange(0,30),history.history['loss'],label='Training_loss')
plt.plot(np.arange(0,30),history.history['val_accuracy'],label='Validation_accuracy')
plt.plot(np.arange(0,30),history.history['val_loss'],label='validation_loss')
plt.xlabel('#Epochs')
plt.ylabel('Percentage')
plt.legend()
plt.title('Training accuracy and Loss plot')
plt.show()
```

Result:

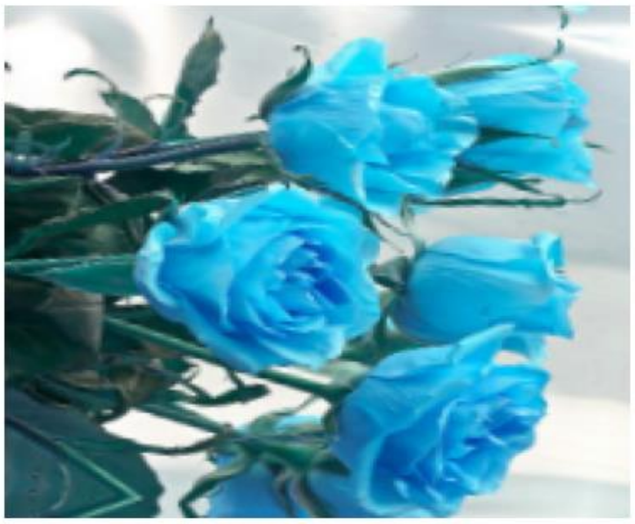
Data Augmentation : Flipping, rotation and zooming.

Before Data Augmentation

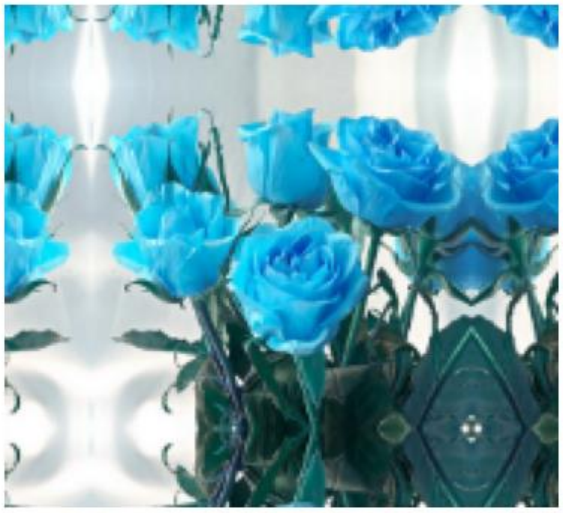
After Data Augmentation (Flipping)



After Data Augmentation (Rotation)



After Data Augmentation (Zooming)

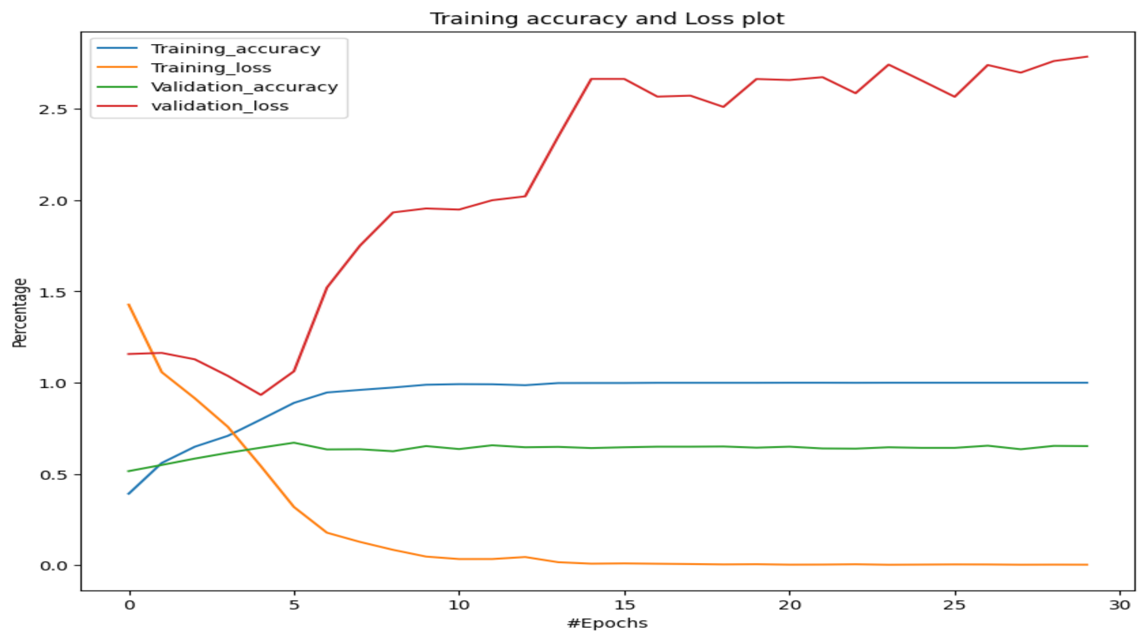


Performance metrices

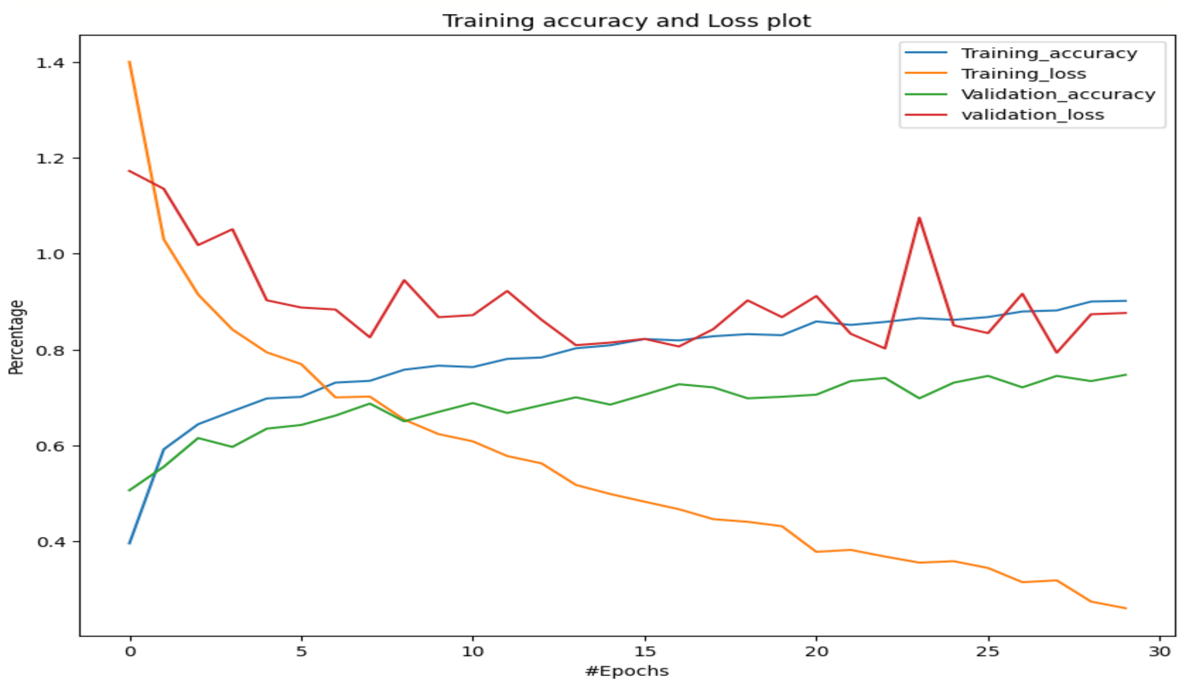
	Before Data Augmentation	After Data Augmentation
Number of samples	3670	3722
Training accuracy	99.96%	90.15%
Training Loss	0.0025	0.2607
Validation accuracy	65.25%	74.73%
Validation Loss	2.7867	0.8763
Image size	3670x180	180x180

Visualizations:

Before data augmentation



After Data Augmentation



Conclusion:

The Data Augmentation is implemented to regularize the CNN model from overfitting and the performance metrices are tabulated and visualised.

Ex No: 6	Restricted Boltzmann Machine

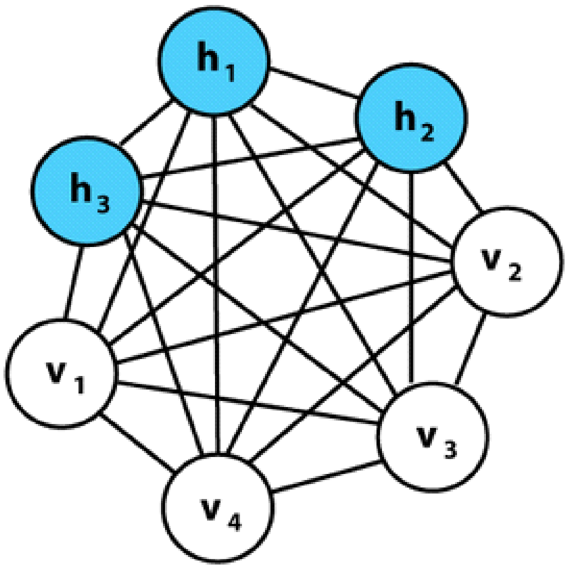
Aim:

To implement Restricted Boltzmann machine in MOVIELENS dataset for Recommendation System.

Boltzmann Machine:

Boltzmann machines are one of the first neural networks capable of learning internal representations and can solve, in a specific time, difficult combinatoric problems. Unlike neural networks like CNN, RNN, etc. Boltzmann Machine are **undirected models**. It contains 2 layers:

- An input layer
- A hidden layer



V1, V2, V3, V4 are visible or input nodes and h1, h2, h3 are hidden nodes.

Restricted Boltzmann Machine

RBMs are shallow neural nets that learn to reconstruct data by themselves in an unsupervised fashion. Application of RBMs is Collaborative Filtering, dimensionality reduction, classification, regression, feature learning, topic modeling, Recommendation System and even Deep Belief Networks (DBN) .

The energy function for the RBM is defined as:

$$E(v,h) = -\sum_i a_i v_i - \sum_j b_j h_j - \sum_i \sum_j v_i w_{i,j} h_j$$

where a and b are biases and v is visible node and h is hidden unit node and w is weight between visible and hidden node. Energy function depends on the visible/input states, hidden states, weights and biases. The training of RBM consists in finding of the parameters for given input values so that the energy reaches a minimum. It can automatically extract meaningful features from a given input.

Restricted Boltzmann Machines are probabilistic. So, probability distributions over hidden and/or visible vectors are defined in terms of the energy function:

$$P(v,h) = \frac{1}{Z} e^{-E(v,h)}$$

Here Z is called the 'partition function' that is the summation over all possible pairs of visible and hidden vectors. In other words, just a normalizing constant to ensure the probability distribution sums to 1.

Dataset:

https://www.kaggle.com/datasets/sherinclaudia/movielens

Sl. No	**Variable**	**Type**
1.	MovieID	Integer
2.	Title	Object
3.	Genres	Object
4.	UserID	Integer
5.	Rating	Integer

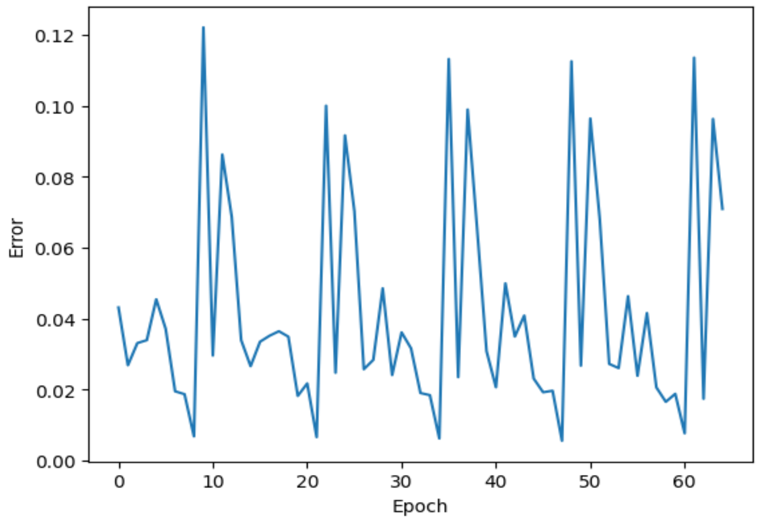
Implementation of RBM:

```
import tensorflow as tf
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
#Loading in the movies dataset
movies = pd.read_csv('/content/drive/MyDrive/datasets/movies.dat', sep = '::', header = None,
engine = 'python', encoding = 'latin-1')
#Loading in the ratings dataset
Users = pd.read_csv('/content/drive/MyDrive/datasets/users.dat', sep = '::', header = None,
engine = 'python', encoding = 'latin-1')
ratings = pd.read_csv('/content/drive/MyDrive/datasets/ratings.dat', sep = '::', header = None,
engine = 'python', encoding = 'latin-1')
user_rating_df = ratings.pivot(index='UserID', columns='MovieID', values='Rating')
ratings=ratings.drop('Timestamp',axis=1)
ratings.head()
norm_user_rating_df = user_rating_df.fillna(0) / 5.0
trX = norm_user_rating_df.values
trX[0:5]
hiddenUnits = 20
visibleUnits =  len(user_rating_df.columns)

vb = tf.Variable(tf.zeros([visibleUnits]), tf.float32)
hb = tf.Variable(tf.zeros([hiddenUnits]), tf.float32)
W = tf.Variable(tf.zeros([visibleUnits, hiddenUnits]), tf.float32)
v0 = tf.zeros([visibleUnits], tf.float32)
#testing to see if the matrix product works
tf.matmul([v0], W)
def hidden_layer(v0_state, W, hb):
   h0_prob = tf.nn.sigmoid(tf.matmul([v0_state], W) + hb)  #probabilities of the hidden units
   h0_state = tf.nn.relu(tf.sign(h0_prob - tf.random.uniform(tf.shape(h0_prob))))
#sample_h_given_X
   return h0_state
#printing output of zeros input
h0 = hidden_layer(v0, W, hb)
print("first 15 hidden states: ", h0[0][0:15])

def reconstructed_output(h0_state, W, vb):
   v1_prob = tf.nn.sigmoid(tf.matmul(h0_state, tf.transpose(W)) + vb)
   v1_state = tf.nn.relu(tf.sign(v1_prob - tf.random.uniform(tf.shape(v1_prob))))
#sample_v_given_h
   return v1_state[0]
v1 = reconstructed_output(h0, W, vb)
print("hidden state shape: ", h0.shape)
print("v0 state shape:  ", v0.shape)
print("v1 state shape:  ", v1.shape)
def error(v0_state, v1_state):
   return tf.reduce_mean(tf.square(v0_state - v1_state))
```

```
err = tf.reduce_mean(tf.square(v0 - v1))
print("error" , err.numpy())
epochs = 5
batchsize = 500
errors = []
weights = []
K=1
alpha = 0.1
#creating datasets
train_ds = \tf.data.Dataset.from_tensor_slices((np.float32(trX))).batch(batchsize)
v0_state=v0
for epoch in range(epochs):
    batch_number = 0
    for batch_x in train_ds:
       for i_sample in range(len(batch_x)):
           for k in range(K):
               v0_state = batch_x[i_sample]
               h0_state = hidden_layer(v0_state, W, hb)
               v1_state = reconstructed_output(h0_state, W, vb)
               h1_state = hidden_layer(v1_state, W, hb)
              delta_W          =          tf.matmul(tf.transpose([v0_state]),          h0_state)          -
tf.matmul(tf.transpose([v1_state]), h1_state)
               W = W + alpha * delta_W
               vb = vb + alpha * tf.reduce_mean(v0_state - v1_state, 0)
               hb = hb + alpha * tf.reduce_mean(h0_state - h1_state, 0)
               v0_state = v1_state

           if i_sample == len(batch_x)-1:
               err = error(batch_x[i_sample], v1_state)
               errors.append(err)
               weights.append(W)
               print ( 'Epoch: %d' % (epoch + 1),
                      "batch #: %i " % batch_number, "of %i" % (len(trX)/batchsize),
                      "sample #: %i" % i_sample,
                      'reconstruction error: %f' % err)
        batch_number += 1
plt.plot(errors)
plt.ylabel('Error')
plt.xlabel('Epoch')
plt.show()
mock_user_id = 215
#Selecting the input user
inputUser = trX[mock_user_id-1].reshape(1, -1)
inputUser = tf.convert_to_tensor(trX[mock_user_id-1],"float32")
v0 = inputUser
print(v0)
v0.shape
v0test = tf.zeros([visibleUnits], tf.float32)
v0test.shape
```

```
#Feeding in the user and reconstructing the input
hh0 = tf.nn.sigmoid(tf.matmul([v0], W) + hb)
vv1 = tf.nn.sigmoid(tf.matmul(hh0, tf.transpose(W)) + vb)
rec = vv1
tf.maximum(rec,1)
for i in vv1:
    print(i)
scored_movies_df_mock = movies[movies['MovieID'].isin(user_rating_df.columns)]
scored_movies_df_mock = scored_movies_df_mock.assign(RecommendationScore = rec[0])
scored_movies_df_mock.sort_values(["RecommendationScore"], ascending=False).head(20)
movies_df_mock = ratings[ratings['UserID'] == mock_user_id]
movies_df_mock.head()
#Merging movies_df with ratings_df by MovieID
merged_df_mock = scored_movies_df_mock.merge(movies_df_mock, on='MovieID', how='outer')
merged_df_mock['UserID']=merged_df_mock['UserID'].astype(pd.Int64Dtype()

merged_df_mock.sort_values(["RecommendationScore"], ascending=False).head(10)
```

Visualization:



The model has five epoch and each epoch undergoes 13 batch normalization. The minimum error 0.004404 is found at batch 8 of epoch 2.

Result:

Recommended movie for a single person with id 215. (The watched movie by the person is indicated with the user id)

Sl. No	Movie ID	Title	Genres	Recommendation Score	User ID	Rating
1148	1240	Terminator, The (1984)	Action|Sci-Fi|Thriller	0.815360	<NA>	NaN
1120	1210	Star Wars: Episode VI - Return of the Jedi (1983)	Action|Adventure|Romance|Sci-Fi|War	0.785401	215	5.0
2374	2571	Matrix, The (1999)	Action|Sci-Fi|Thriller	0.762195	<NA>	NaN
253	260	Star Wars: Episode IV - A New Hope (1977)	Action|Adventure|Fantasy|Sci-Fi	0.756881	215	5.0
1106	1196	Star Wars: Episode V - The Empire Strikes Back...	Action|Adventure|Drama|Sci-Fi|War	0.714569	<NA>	NaN
2708	2916	Total Recall (1990)	Action|Adventure|Sci-Fi|Thriller	0.663168	215	4.0
863	924	2001: A Space Odyssey (1968)	Drama|Mystery|Sci-Fi|Thriller	0.611130	<NA>	NaN
1025	1097	E.T. the Extra-Terrestrial (1982)	Children's|Drama|Fantasy|Sci-Fi	0.595124	215	4.0
1178	1270	Back to the Future (1985)	Comedy|Sci-Fi	0.576349	<NA>	NaN
2557	2762	Sixth Sense, The (1999)	Thriller	0.533358	<NA>	NaN

Conclusion:

Thus, the Restricted Boltzmann Machine have been implemented successfully for Movie Recommendation system.

Ex No: 7	Autoencoders

AIM:

To implement Deep CNN and denoising Autoencoders using MNIST dataset in python.

Introduction

Autoencoders are a specific type of feedforward neural networks where the input is the same as the output. They compress the input into a lower-dimensional *code* and then reconstruct the output from this representation. The code is a compact "summary" or "compression" of the input, also called the *latent-space representation.*

An autoencoder consists of 3 components: encoder, code and decoder. The encoder compresses the input and produces the code, the decoder then reconstructs the input only using this code. Three methods are required to build an autoencoder, an encoding method, decoding method, and a loss function to compare the output with the target.

The following four hyperparameters are set before training an autoencoder:

Code size: number of nodes in the middle layer. Smaller size results in more compression.

Number of layers: the autoencoder can be deep.

Number of nodes per layer: the autoencoder architecture used is called a *stacked autoencoder* since the layers are stacked one after another. Usually stacked autoencoders look like a "sandwitch". The number of nodes per layer decreases with each subsequent layer of the encoder, and increases back in the decoder. Also the decoder is symmetric to the encoder in terms of layer structure.

Loss function: Use *mean squared error (mse)* or *binary crossentropy*. If the input values are in the range [0, 1] then use crossentropy, otherwise use the mean squared error.

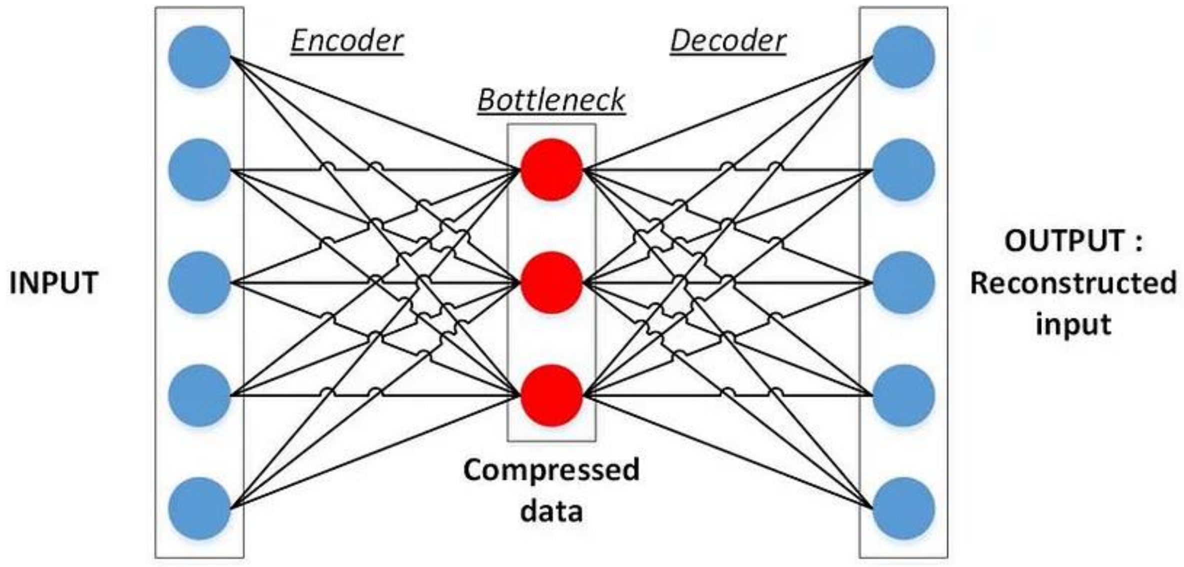
Training the model works as follows:

1. Pass the input image to the model and generate the output using the Sigmoid activation function.

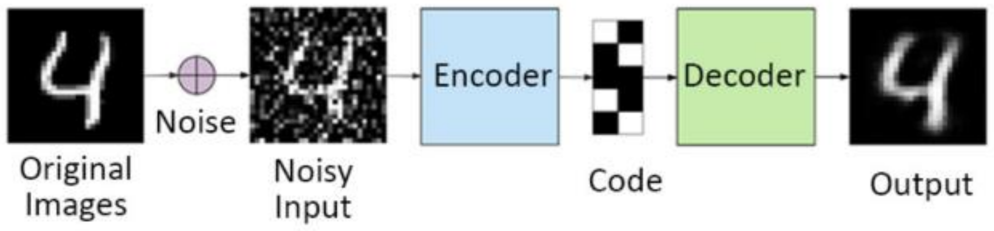
2. Calculate the loss of actual and predicted values.

3. Backpropagate and adjust the parameters in order to minimize the loss.

Deep CNN Autoencoder



Denoising Autoencoder



ALGORITHM:

Algorithm 1 Convolutional Auto-encoder.

Step1: *Encoding*
Convert input data $\mathbf{X}$ into output of hidden layers, $\mathbf{Z}$, by

$$\mathbf{Z} = f(\mathbf{X}) = \eta(w.\mathbf{X} + b) \tag{1}$$

where η denotes a non-linear activation function.

Step2: (Decoding)
Regenerate input value $\mathbf{Y}$ based on output hidden layer $\mathbf{Z}$ by

$$\mathbf{Y} = \hat{f}(\mathbf{Z}) = \zeta\left(\hat{w}.\mathbf{Z} + \hat{b}\right) \tag{2}$$

where the activation function ζ and η are analogous.

Step3: (Calculation)
Compute square error norm, $\rho(\mathbf{X}, \mathbf{Y}) = \|\mathbf{X} - \mathbf{Y}\|^2$, where ρ is the error cost function.

Algorithm 2 Denoising Autoencoder Training

```
1: procedure DA Training(e,b,$\boldsymbol{x}$,c,l,$\boldsymbol{\theta}$)
2:     $\boldsymbol{x} = [x_1, x_2, ...x_n] \in R^{n*m}$ is the input matrix, in which $x_i \in [0,1]^m$ $(1 \leq i \leq m)$ is a single input data
3:     $e$ is the amount of epochs to be iterated
4:     $b$ is the amount of batches
5:     $l$ is the learning rate
6:     $c$ is the corruption level
7:     $\boldsymbol{\theta} = \{W, \boldsymbol{b}, \boldsymbol{b_h}\}$ where $W \in R^{n*d}, b \in R^d, b_h \in R^d$, $\boldsymbol{\theta}$ is the parameters of a DA
8:     for 0 to $e$ do
9:         for 0 to $b$ do
10:            $\tilde{\boldsymbol{x}}$ = getCorruptedInput($\boldsymbol{x}$,c), in which c is the corrupted level
11:            $\boldsymbol{h} = sigmoid(\tilde{\boldsymbol{x}} * W + \boldsymbol{b})$
12:            $\widehat{\boldsymbol{x}} = sigmoid(\boldsymbol{h} * W^T + \boldsymbol{b_h})$
13:            $\boldsymbol{L(x, \widehat{x})} = -\sum_{i=1}^{d} [\boldsymbol{x}_i log \widehat{\boldsymbol{x}}_i + (1 - \boldsymbol{x}_i) log(1 - \widehat{\boldsymbol{x}}_i)]$
14:            cost = mean($\boldsymbol{L(x, \widehat{x})}$)
15:            $\boldsymbol{g}$ = compute the gradients of the cost with respect to $\boldsymbol{\theta}$
16:            for $\theta_i, g_i$ in $(\boldsymbol{\theta}, \boldsymbol{g})$ do
17:                $\theta_i = \theta_i - l * g_i$
18:            end for
19:        end for
20:    end for
21: end procedure
```

Dataset

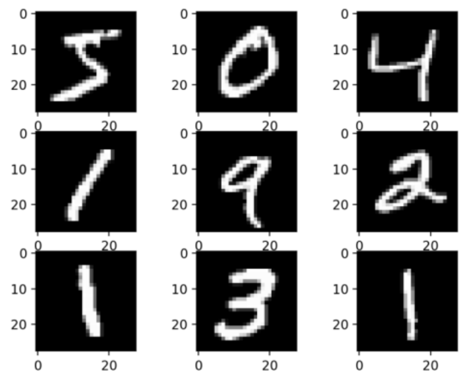
Modified National Institute of Standards and Technology dataset

Description:

size	60,000 images
pixels	28 X 28 pixel gray scale images
content	handwritten single digits between 0 and 9

Data Visualization /preprocessing

First nine images in the dataset



Implementation

```
from keras.layers import Dense,Conv2D,MaxPooling2D,UpSampling2D
from keras import Input, Model
from keras.datasets import mnist
import numpy as np
import matplotlib.pyplot as plt
encoding_dim = 15
input_img = Input(shape=(784,))
from keras.models import Sequential
#Deep CNN
#The encoder will be made up of a stack of Conv2D and max-
pooling layer and the decoder will have a stack of Conv2D and Upsampling Layer.
model = Sequential()
# encoder network
model.add(Conv2D(30, 3, activation= 'relu', padding='same', input_shape = (28,28,1)))
model.add(MaxPooling2D(2, padding= 'same'))
model.add(Conv2D(15, 3, activation= 'relu', padding='same'))
model.add(MaxPooling2D(2, padding= 'same'))
#decoder network
model.add(Conv2D(15, 3, activation= 'relu', padding='same'))
model.add(UpSampling2D(2))
model.add(Conv2D(30, 3, activation= 'relu', padding='same'))
model.add(UpSampling2D(2))
model.add(Conv2D(1,3,activation='sigmoid', padding= 'same')) # output layer
model.compile(optimizer= 'adam', loss = 'binary_crossentropy')
model.summary()
(x_train, _), (x_test, _) = mnist.load_data()
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = np.reshape(x_train, (len(x_train), 28, 28, 1))
x_test = np.reshape(x_test, (len(x_test), 28, 28, 1))
model.fit(x_train, x_train,
          epochs=15,
                batch_size=128,
                validation_data=(x_test, x_test))
pred = model.predict(x_test)
plt.figure(figsize=(20, 4))
for i in range(5):
   # Display original
   ax = plt.subplot(2, 5, i + 1)
   plt.imshow(x_test[i].reshape(28, 28))
   plt.gray()
   ax.get_xaxis().set_visible(False)
   ax.get_yaxis().set_visible(False)
   # Display reconstruction
   ax = plt.subplot(2, 5, i + 1 + 5)
   plt.imshow(pred[i].reshape(28, 28))
   plt.gray()
   ax.get_xaxis().set_visible(False)
   ax.get_yaxis().set_visible(False)
plt.show()


#Denoising Autoencoder
#how the model performs with noise in the image.
#What we mean by noise is blurry images, changing the color of the images, or even white markers on the
image.

noise_factor = 0.7
x_train_noisy = x_train + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=x_train.shape)
x_test_noisy = x_test + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=x_test.shape)
x_train_noisy = np.clip(x_train_noisy, 0., 1.)
x_test_noisy = np.clip(x_test_noisy, 0., 1.)
#Here is how the noisy images look right now.
plt.figure(figsize=(20, 2))
for i in range(1, 5 + 1):
   ax = plt.subplot(1, 5, i)
   plt.imshow(x_test_noisy[i].reshape(28, 28))
   plt.gray()
   ax.get_xaxis().set_visible(False)
   ax.get_yaxis().set_visible(False)
plt.show()
model = Sequential()
# encoder network
model.add(Conv2D(35, 3, activation= 'relu', padding='same', input_shape = (28,28,1)))
model.add(MaxPooling2D(2, padding= 'same'))
model.add(Conv2D(25, 3, activation= 'relu', padding='same'))
model.add(MaxPooling2D(2, padding= 'same'))
#decoder network
model.add(Conv2D(25, 3, activation= 'relu', padding='same'))
model.add(UpSampling2D(2))
model.add(Conv2D(35, 3, activation= 'relu', padding='same'))
model.add(UpSampling2D(2))
model.add(Conv2D(1,3,activation='sigmoid', padding= 'same')) # output layer
model.compile(optimizer= 'adam', loss = 'binary_crossentropy')
model.fit(x_train_noisy, x_train,
                epochs=15,
                batch_size=128,
                validation_data=(x_test_noisy, x_test))
pred = model.predict(x_test_noisy)
plt.figure(figsize=(20, 4))
for i in range(5):
    # Display original
    ax = plt.subplot(2, 5, i + 1)
    plt.imshow(x_test_noisy[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    # Display reconstruction
    ax = plt.subplot(2, 5, i + 1 + 5)
    plt.imshow(pred[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```

Results:

CNN
Summary

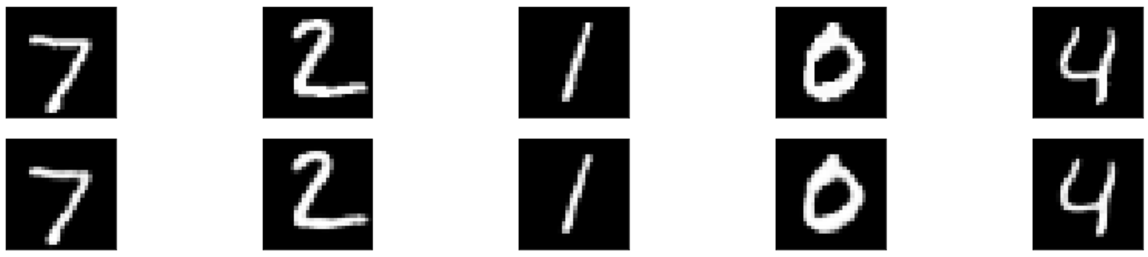
Layer (type)	**Output Shape**	**Param #**
conv2d (Conv2D)	(None, 28, 28, 30)	300
max_pooling2d (MaxPooling2D)	(None, 14, 14, 30)	0
conv2d_1 (Conv2D)	(None, 14, 14, 15)	4065
max_pooling2d_1 (MaxPooling2D)	(None, 7, 7, 15)	0
conv2d_2 (Conv2D)	(None, 7, 7, 15)	2040
up_sampling2d (UpSampling2D)	(None, 14, 14, 15)	**0**
conv2d_3 (Conv2D)	(None, 14, 14, 30)	4080
conv2d_4 (Conv2D)	(None, 28, 28, 1)	271

Total params: 10,756
Trainable params: 10,756
Non-trainable params: 0

Loss

```
Epoch 1/15
469/469 [=============] - 167s 353ms/step - loss: 0.1510 - val_loss: 0.0815
Epoch 2/15
469/469 [=============] - 154s 329ms/step - loss: 0.0786 - val_loss: 0.0752
Epoch 3/15
469/469 [=============] - 152s 324ms/step - loss: 0.0748 - val_loss: 0.0734
Epoch 4/15
469/469 [=============] - 154s 329ms/step - loss: 0.0732 - val_loss: 0.0719
Epoch 5/15
469/469 [=============] - 151s 323ms/step - loss: 0.0721 - val_loss: 0.0710
Epoch 6/15
469/469 [=============] - 150s 320ms/step - loss: 0.0713 - val_loss: 0.0704
Epoch 7/15
469/469 [=============] - 149s 317ms/step - loss: 0.0707 - val_loss: 0.0699
Epoch 8/15
469/469 [=============] - 148s 316ms/step - loss: 0.0703 - val_loss: 0.0694
Epoch 9/15
469/469 [=============] - 149s 317ms/step - loss: 0.0699 - val_loss: 0.0691
Epoch 10/15
469/469 [=============] - 148s 316ms/step - loss: 0.0696 - val_loss: 0.0688
Epoch 11/15
469/469 [=============] - 148s 316ms/step - loss: 0.0693 - val_loss: 0.0687
Epoch 12/15
469/469 [=============] - 147s 314ms/step - loss: 0.0690 - val_loss: 0.0683
Epoch 13/15
469/469 [=============] - 148s 315ms/step - loss: 0.0688 - val_loss: 0.0681
Epoch 14/15
469/469 [=============] - 149s 317ms/step - loss: 0.0686 - val_loss: 0.0679
Epoch 15/15
469/469 [=============] - 149s 319ms/step - loss: 0.0684 - val_loss: 0.0678
```

Encoded and Decoded images of CNN

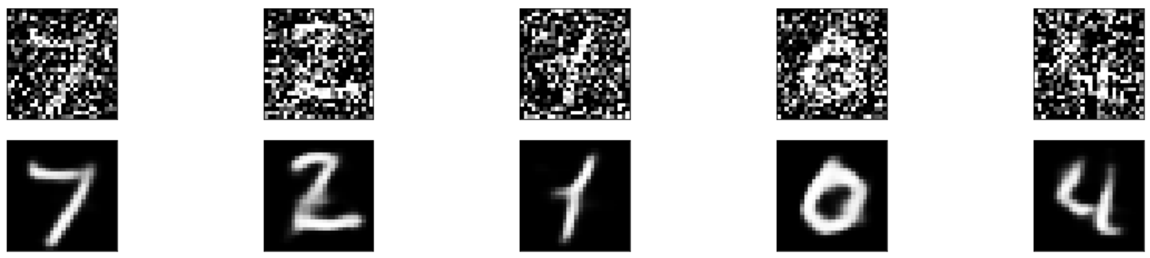


Denoising Autoencoder

Loss

```
Epoch 1/15
469/469 [=============] - 161s 340ms/step - loss: 0.1943 - val_loss: 0.1469
Epoch 2/15
469/469 [=============] - 155s 330ms/step - loss: 0.1436 - val_loss: 0.1380
Epoch 3/15
469/469 [=============] - 155s 331ms/step - loss: 0.1370 - val_loss: 0.1331
Epoch 4/15
469/469 [=============] - 155s 331ms/step - loss: 0.1329 - val_loss: 0.1305
Epoch 5/15
469/469 [=============] - 156s 332ms/step - loss: 0.1305 - val_loss: 0.1281
Epoch 6/15
469/469 [=============] - 155s 331ms/step - loss: 0.1287 - val_loss: 0.1267
Epoch 7/15
469/469 [=============] - 154s 329ms/step - loss: 0.1275 - val_loss: 0.1256
Epoch 8/15
469/469 [=============] - 153s 327ms/step - loss: 0.1265 - val_loss: 0.1251
Epoch 9/15
469/469 [=============] - 154s 328ms/step - loss: 0.1256 - val_loss: 0.1240
Epoch 10/15
469/469 [=============] - 153s 327ms/step - loss: 0.1248 - val_loss: 0.1233
Epoch 11/15
469/469 [=============] - 153s 327ms/step - loss: 0.1242 - val_loss: 0.1228
Epoch 12/15
469/469 [=============] - 153s 326ms/step - loss: 0.1236 - val_loss: 0.1221
Epoch 13/15
469/469 [=============] - 153s 325ms/step - loss: 0.1231 - val_loss: 0.1220
Epoch 14/15
469/469 [=============] - 152s 325ms/step - loss: 0.1226 - val_loss: 0.1214
Epoch 15/15
469/469 [=============] - 152s 325ms/step - loss: 0.1223 - val_loss: 0.1217
```

Encoded and Decoded images of Denoising Autoencoder



Conclusion:

Deep CNN Autoencoder and Denoising Autoencoder were implemented using Python.

Ex. No. 8	Computer Vision

Aim:

To visualise and detect objects in image data using openCV.

Introduction

Computer vision is a discipline that studies how to reconstruct, interrupt and understand a 3d scene from its 2d images, in terms of the properties of the structure present in the scene.
OpenCV functions for Reading, Showing, Writing an Image File
imread() function − reading an image. OpenCV imread() supports various image formats like PNG, JPEG, JPG, TIFF, etc.
imshow() function − for showing an image in a window. The window automatically fits to the image size. OpenCV imshow() supports various image formats like PNG, JPEG, JPG, TIFF, etc.
imwrite() function − for writing an image. OpenCV imwrite() supports various image formats like PNG, JPEG, JPG, TIFF, etc.

Color Space Conversion
In OpenCV, the images are not stored by using the conventional RGB color, rather they are stored in the reverse order i.e. in the BGR order. Hence the default color code while reading an image is BGR. The cvtColor() color conversion function in for converting the image from one color code to other.
Edge Detection
Humans, after seeing a rough sketch, can easily recognize many object types and their poses. OpenCV provides very simple and useful function called Canny()for detecting the edges.

Face and Eye Detection

Face and eye detection is one of the fascinating applications of computer vision which makes it more realistic as well as futuristic. OpenCV has a built-in facility to perform face detection. Haar cascade classifier is used for face detection.

Data

Implementation

```
!pip install opencv-python
import cv2
import numpy as np
face_detection=cv2.CascadeClassifier('/usr/local/lib/python3.7/dist-
packages/cv2/data/haarcascade_frontalface_alt.xml')
img = cv2.imread('/jas.jpg')
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
cv2.imwrite('/edges.jpg',cv2.Canny(img,200,300))
faces = face_detection.detectMultiScale(gray, 1.3, 5)
for (x,y,w,h) in faces:
   img = cv2.rectangle(img,(x,y),(x+w, y+h),(255,0,0),3)
cv2.imwrite('/Face.jpg',img)
eye_cascade = cv2.CascadeClassifier('/usr/local/lib/python3.7/dist-
packages/cv2/data/haarcascade_eye.xml')
eyes = eye_cascade.detectMultiScale(gray, 1.03, 5)
for (ex,ey,ew,eh) in eyes:
   img = cv2.rectangle(img,(ex,ey),(ex+ew, ey+eh),(0,255,0),2)
cv2.imwrite('/Eye.jpg',img)
```

Results

Canny Edge detection	Face Detection	Eye Detection

Conclusion

Thus, using OpenCV an image is read, its edges, face and eyes are detected using haar cascade classifier.

Ex. No. 9	Bidirectional LSTM

Aim

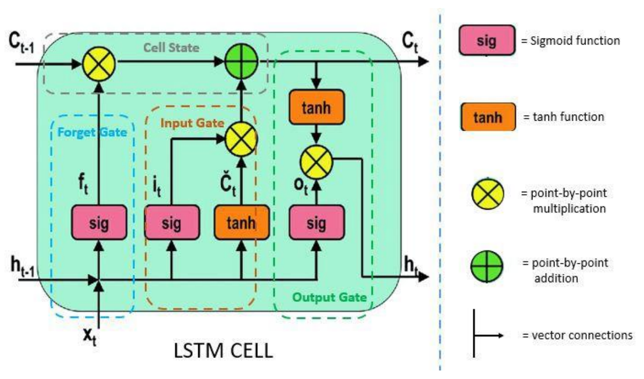
To perform a movie review (text classification) on the IMDB dataset to predict the polarity of the review is positive or negative.

Introduction

Recurrent Neural Network (RNN) considers the sequence of words in a sentence for the prediction. Vanishing gradient is a big problem in deep neural networks. It vanishes or explodes quickly in earlier layers and this makes RNN unable to hold information of longer sequence. Thus, RNN becomes short-term memory. LSTMs are a special kind of RNN — capable of learning long-term dependencies by remembering information for long periods is the default behavior. Long Short-Term Memory (LSTM) is an RNN architecture specifically designed to address the vanishing gradient problem.

LSTM networks

Every LSTM network basically contains three gates to control the flow of information and cells to hold information. The Cell States carries the information from initial to later time steps without getting vanished.



Gates

Gates make use of sigmoid activation or you can say tanh activation. values ranges in tanh activation are 0 -1.

Forget Gate: This gate decides what information should be carried out forward or what information should be ignored. Information from previous hidden states and the current state information passes through the sigmoid function. Values that come out from sigmoid are always between 0 and 1. if the value is closer to 1 means information should proceed forward and if value closer to 0 means information should be ignored.

Input Gate: After deciding the relevant information, the information goes to the input gate, Input gate passes the relevant information, and this leads to updating the cell states. simply saving updating the weight. Input gate adds the new relevant information to the existing information by updating cell states.

Output Gate: After the information is passed through the input gate, now the output gate comes into play. Output gate generates the next hidden states. and cell states are carried over the next time step.

Dataset

This is a dataset of 25,000 movies reviews from IMDB, labelled by sentiment (positive/negative). Reviews have been pre-processed, and each review is encoded as a list of word indexes (integers). For convenience, words are indexed by overall frequency in the dataset, so that for instance the integer "3" encodes the 3rd most frequent word in the data. This allows for quick filtering operations such as: "only consider the top 10,000 most common words, but eliminate the top 20 most common words". As a convention, "0" does not stand for a specific word, but instead is used to encode any unknown word.

Implementation

1.Import Libraries

2. Import the dataset

num_words = 10000 signifies that only 10000 unique words will be taken for our dataset.

x_train, x_test: List of movie reviews text data. having an uneven length.

y_train, y_test: Lists of integer target labels (1 or 0).

3. Feature Extraction

Since the text data in x_train and x_test have an uneven length, convert the text data into a numerical form in order to feed it into the model.

Make the length of texts equal using padding. Define **max_len = 200**. If a sentence is having a length greater than 200 it will be trimmed off otherwise it will be padded by 0.

4. Designing the Bi-directional LSTM

input_length = maxlen Since we have already made all sentences in our dataset have an equal length of 200 using pad_sequence.

The Embedding layer takes **n_unique_words** as the size of the vocabulary in our dataset which is already declared as 10000.

After the Embedding layer, we are adding Bi-directional LSTM units.

Using sigmoid activation and then compiling the model

5. Training the model

```
import numpy as np
```

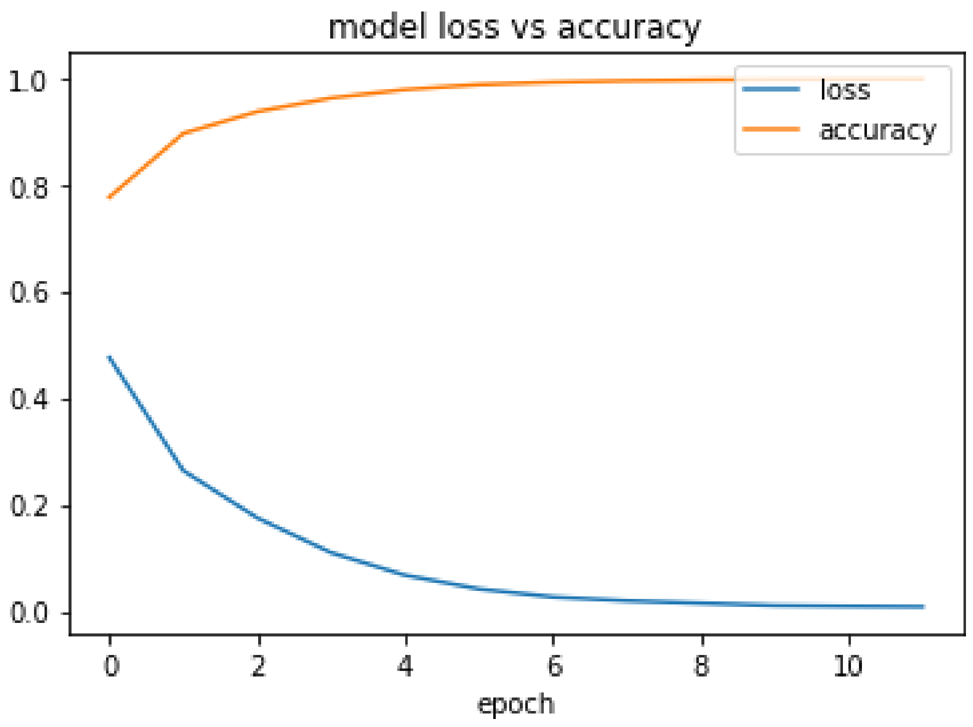
```
from keras.models import Sequential
from keras.preprocessing import sequence
from keras.layers import Dropout
from keras.layers import  Dense, Embedding, LSTM, Bidirectional
from keras.datasets import imdb
(x_train, y_train),(x_test, y_test) = imdb.load_data(num_words=10000)
maxlen = 200
x_train = sequence.pad_sequences(x_train, maxlen=maxlen)
x_test = sequence.pad_sequences(x_test, maxlen=maxlen)
y_test = np.array(y_test)
y_train = np.array(y_train)
n_unique_words=10000
model = Sequential()
model.add(Embedding(n_unique_words, 128, input_length=maxlen))
model.add(Bidirectional(LSTM(64)))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
history=model.fit(x_train, y_train,
         batch_size=5,
         epochs=12,
         validation_data=[x_test, y_test])
print(history.history['loss'])
print(history.history['accuracy'])
from matplotlib import pyplot
pyplot.plot(history.history['loss'])
pyplot.plot(history.history['accuracy'])
pyplot.title('model loss vs accuracy')
pyplot.xlabel('epoch')
pyplot.legend(['loss', 'accuracy'], loc='upper right')
pyplot.show()
```

Results

```
Epoch 1/12
5000/5000 [=======] - 900s 179ms/step - loss: 0.4747 - accuracy: 0.7761 - val_loss: 0.3691 - val_accuracy: 0.8323
Epoch 2/12
5000/5000 [=======] - 880s 176ms/step - loss: 0.2633 - accuracy: 0.8955 - val_loss: 0.3073 - val_accuracy: 0.8680
Epoch 3/12
5000/5000 [=======] - 876s 175ms/step - loss: 0.1747 - accuracy: 0.9363 - val_loss: 0.3301 - val_accuracy: 0.8752
Epoch 4/12
```

```
5000/5000 [=======] - 876s 175ms/step - loss: 0.1094 - accuracy: 0.9618 - val_loss: 0.3444 - val_accuracy: 0.8675
Epoch 5/12
5000/5000 [=======] - 873s 175ms/step - loss: 0.0671 - accuracy: 0.9776 - val_loss: 0.4523 - val_accuracy: 0.8668
Epoch 6/12
5000/5000 [=======] - 865s 173ms/step - loss: 0.0413 - accuracy: 0.9870 - val_loss: 0.5443 - val_accuracy: 0.8632
Epoch 7/12
5000/5000 [=======] - 867s 173ms/step - loss: 0.0263 - accuracy: 0.9914 - val_loss: 0.6314 - val_accuracy: 0.8590
Epoch 8/12
5000/5000 [=======] - 863s 173ms/step - loss: 0.0191 - accuracy: 0.9940 - val_loss: 0.7184 - val_accuracy: 0.8625
Epoch 9/12
5000/5000 [=======] - 856s 171ms/step - loss: 0.0147 - accuracy: 0.9956 - val_loss: 0.7398 - val_accuracy: 0.8590
Epoch 10/12
5000/5000 [=======] - 858s 172ms/step - loss: 0.0103 - accuracy: 0.9967 - val_loss: 0.8531 - val_accuracy: 0.8615
Epoch 11/12
5000/5000 [=======] - 860s 172ms/step - loss: 0.0091 - accuracy: 0.9972 - val_loss: 0.8500 - val_accuracy: 0.8576
Epoch 12/12
5000/5000 [=======] - 891s 178ms/step - loss: 0.0081 - accuracy: 0.9974 - val_loss: 0.8511 - val_accuracy: 0.8608
```

Loss Vs Accuracy



Conclusion

A Bi-LSTM is built to predict the polarity of a movie review and the model is evaluated. The loss and accuracy during each epoch are calculated and the accuracy in predicting the polarity of a movie review is 99.7% and it is visualised.