DATE:
EX.NO: 1

BASICS OF PTHON – INTRODUCTION TENSORFLOW METHODS AND OPERATION.

AIM:

To explore and learn basic operators and methods in tensor flow.

TENSORFLOW DISCRIPTION:

TensorFlow is an open-source library developed by Google primarily for deep learning applications. It also supports traditional machine learning.

TensorFlow was originally developed for large numerical computations without keeping deep learning in mind. However, it proved to be very useful for deep learning development as well, and therefore Google open-sourced it.

TensorFlow accepts data in the form of multi-dimensional arrays of higher dimensions called tensors. Multi-dimensional arrays are very handy in handling large amounts of data.

TensorFlow works on the basis of data flow graphs that have nodes and edges. As the execution mechanism is in the form of graphs, it is much easier to execute TensorFlow code in a distributed manner across a cluster of computers while using GPUs.

PROGRAM:

#Tensorflow can be installed in Python using:

pip install tensorflow

#Tensorflow can be imported in Python as:

import tensorflow as tf

#TensorFlow operates on multi-dimensional arrays called as tensors represented as **tf.Tensor().**
#Tensors have a rank which is the number of dimensions and the shape which is similar to numpy array.
#Tensors can be created using:

tf.constant(tf.Tensor(), dtype)

#Variable can be created using:

tf.variable(tf.Tensor())

```
x=torch.arange(12,dtype=torch.float32) #tensor and vectors
print(x)
```

OUTPUT: tensor([0., 1., 2., 3., 4., 5., 6., 7., 8., 9., 10., 11.])

```
#no of elements x.numel()
```

OUTPUT: 12

```
# shape x.shape
```

OUTPUT: torch.Size([12]) x=x.reshape(3,4)

```
X
```

OUTPUT: tensor([[0., 1., 2., 3.],
[4., 5., 6., 7.],
[8., 9., 10., 11.]])
y=torch.zeros(2,3,4,dtype=torch.int64) y

OUTPUT: tensor([[[0, 0, 0, 0],
[0, 0, 0, 0],
[0, 0, 0, 0]]
[[0, 0, 0, 0],
[0, 0, 0, 0],
[0, 0, 0, 0]]])
z=torch.ones(1,2,dtype=torch.int64) z

OUTPUT:
tensor([[1, 1]])
rand=torch.randn(3,4) rand

OUTPUT:
tensor([[0.1997, 0.8492,0.6358, -
0.6414], [-0.4781, 0.3819, -1.3364, -
0.2838],
[-1.3081, -0.9683, 1.1555, -0.4296]])

#indexing and slicing
#start and step are optional in slicing x[-1]

OUTPUT: tensor([8., 9., 10., 11.]) x[1:3]

OUTPUT: tensor([[4., 5., 6., 7.],
[8., 9., 10., 11.]])

<u>Evaluate:</u>
Returns the loss value and metric values for the model in test mode.

```
evaluate(
        x=None, y=None,
        batch_size=None,
        verbose='auto'
)
```

x – Input data
y- Target data
batch_size – Number of samples per batch of computation
verbose – verbosity mode.

<u>Fit:</u>
Trains the Sequential model using the training data

```
fit(
        x=None, y=None,
        epochs=1,
        batch_size=None,
        validation_data=None,
        validataion_split=0.0
```

)
x – Input data
y- Target data
batch_size – Number of samples per batch of computation
epochs – Number of epochs to train the model
validation_split – Fraction of the training data to be used as validation data
validation_data – Data on which to evaluate the loss and any model metrics at the end of each epoch. The model will not be trained on this data.

Summary:

Prints a string summary of the network/model.

Summary()

Predict:

Generates output predictions for the given input samples.

```
predict(
    x=None
)
```

x- Input data samples

KERAS DISCRIPSTION:

Keras is a high-level, deep learning API developed by Google for implementing neural networks. It is written in Python and is used to make the implementation of neural networks easy. It also supports multiple backend neural network computation.

Keras is relatively easy to learn and work with because it provides a python frontend with a high level of abstraction while having the option of multiple back-ends for computation purposes. This makes Keras slower than other deep learning frameworks, but extremely beginner-friendly.

tf.keras.Layers:

It contains all the layers available in the keras API.

Classes:-

AvergePooling2D() – average pooling operation for spatial data.
Conv2D() – 2D convolutional layer.
Dense() – Just a regular densely-connected neural network layer.
Dropout() – Randomly drops out some input units.
Flatten() – It falattens the input data.
Input() – Layer used as an entry point into a network.
LSTM() – Long short term memory layer.
Lambda() – We can pass our own arbitrary function as the layer.
MaxPool2D() – Max pooling operation for 2D spatial data.
SimpleRNN() – Fully connected RNN where the output is fed back as the input.

tf.keras.losses:

It contains all the built-in loss functions available in the keras API.

Classes:-

BinaryCrossentropy() – computes the cross-entropy loss between true labels and predicted labels for binary output.
SparseCategoricalCrossentropy() - computes the cross-entropy loss between true labels and predicted labels for multi-class output.
MeanAbsoluteError() - Computes the mean of absolute difference between labels and predictions.
MeanSquaredError() - Computes the mean of squares of errors between labels and predictions.

tf.keras.metrics:

It contains all the metrics available in the keras API.

Classes:-

Accuracy() – Calculates how often predictions equal labels.
MeanSquaredError() - Computes the mean of squares of errors between labels and predictions.
MeanAbsoluteError() - Computes the mean of absolute difference between labels and predictions.
RootMeanSquaredError() - Computes the root mean of squares of errors between labels and predictions.

tf.keras.optimizers:

It contains all the metrics available in the keras API.
Classes:-
Adam() - Optimizer that implements the Adam algorithm.
Adamax() - Optimizer that implements the Adamax algorithm.
SGD() - Optimizer that implements the SGD algorithm.

tf.keras.activations:
It contains all the metrics available in the keras API.

Classes:-

linear() – Linear activation function(pass-through).
relu() – Applies the rectified linear unit activation function.
sigmoid() –Sigmoid activation function sigmoid(x) = 1 / (1 + exp(-x)).
softmax() – Softmax converts a vector of values to a probability distribution.
tanh() – Hyperbolic tangent activation function.

tf.keras.datasets:
It contains all the metrics available in the keras API.

Classes:-

Cifar10() – CIFAR10 small image classification dataset.
mnist() – MNIST handwritten digits dataset.

Padding:

Adds padding to an integer Tensor

```
        Tf.pad(
                Tensor,
                paddings
        )
```

Tensor – the tensor that is passed.
Paddings – an integer value for the padding size.

<u>Expanding Dimensions:</u>

Returns a Tensor with a length 1, axis is inserted according to the passed value.

```
Tf.expand_dims(
        Input,
        axis )
```

Input – The input Tensor.
Axis – Integer specifying the dimension index at which to expand the shape of the input.

<u>Repeat:</u>

A Tensor which has the same shape as input, except along the given axis. If axis is None then the output array is flattened to match the flattened input array.

```
Tf.repeat(
        Input,
        Repeats,
        Axis=None )
```

Input – The input Tensor.
Repeats - A Tensor which has the same shape as input, except along the given axis. If axis is None then the output array is flattened to match the flattened input array.
Axis - An int. The axis along which to repeat values.

RESULT:

This basic tensor flow operation and methods are implemented successfully.

DATE:

LINEAR REGRESSION USING TENSORFLOW

EX.NO: 2(a)

AIM:

To implement Linear Regression using Tensorflow from scratch

DESCRIPTION:

Linear regression is a statistical method used to model and analyze the relationship between a dependent variable and one or more independent variables. It aims to find the best-fitting straight line (a linear equation) that represents the pattern of the data, allowing for predictions and understanding the nature of the connection between the variables.

The linear regression equation can be written as: y = $\widehat{\beta_0}$ + $\widehat{\beta_1}$ *x

where:

y is the dependent variable (the one being predicted or explained).
x is the independent variable (the one used for making predictions).
b0 is the y-intercept.
b1 is the slope or coefficient of the independent variable.

The least squares estimates of β_0 and β_1 are:

$$\hat{\beta}_1 = \frac{\sum_{i=1}^{n}(X_i - \bar{X})(Y_i - \bar{Y})}{\sum_{i=1}^{n}(X_i - \bar{X})^2}$$

$$\hat{\beta}_0 = \bar{Y} - \hat{\beta}_1 \bar{X}$$

The primary goal of linear regression is to estimate the values of b0 and b1 such that the line fits the data points as closely as possible. This is achieved by minimizing the sum of the squared differences between the predicted y-values and the actual y-values, known as the "method of least squares."

PROGRAM:

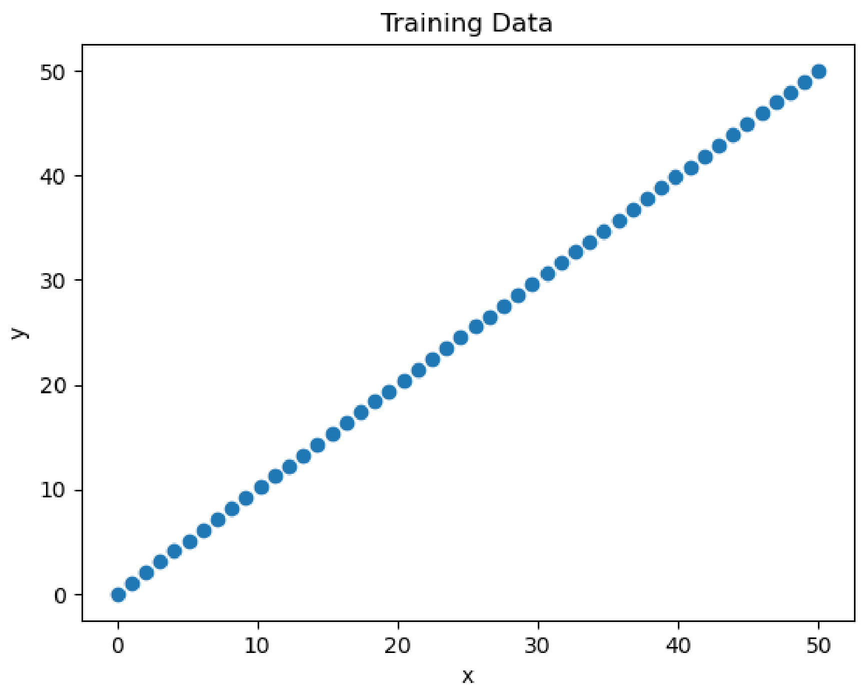
```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
np.random.seed(101)

#Generates random linear data points from 1 to 50
x = np.linspace(0,50,50) #(start,end,no of points to be generated)
y = np.linspace(0,50,50)
x,y,n
```

OUTPUT:

```
(array([ 0.        ,  1.02040816,  2.04081633,  3.06122449,  4.08163265,
         5.10204082,  6.12244898,  7.14285714,  8.16326531,  9.18367347,
        10.20408163, 11.2244898 , 12.24489796, 13.26530612, 14.28571429,
        15.30612245, 16.32653061, 17.34693878, 18.36734694, 19.3877551 ,
        20.40816327, 21.42857143, 22.44897959, 23.46938776, 24.48979592,
        25.51020408, 26.53061224, 27.55102041, 28.57142857, 29.59183673,
        30.6122449 , 31.63265306, 32.65306122, 33.67346939, 34.69387755,
        35.71428571, 36.73469388, 37.75510204, 38.7755102 , 39.79591837,
        40.81632653, 41.83673469, 42.85714286, 43.87755102, 44.89795918,
        45.91836735, 46.93877551, 47.95918367, 48.97959184, 50.        ]),
 array([ 0.        ,  1.02040816,  2.04081633,  3.06122449,  4.08163265,
         5.10204082,  6.12244898,  7.14285714,  8.16326531,  9.18367347,
        10.20408163, 11.2244898 , 12.24489796, 13.26530612, 14.28571429,
        15.30612245, 16.32653061, 17.34693878, 18.36734694, 19.3877551 ,
        20.40816327, 21.42857143, 22.44897959, 23.46938776, 24.48979592,
        25.51020408, 26.53061224, 27.55102041, 28.57142857, 29.59183673,
        30.6122449 , 31.63265306, 32.65306122, 33.67346939, 34.69387755,
        35.71428571, 36.73469388, 37.75510204, 38.7755102 , 39.79591837,
        40.81632653, 41.83673469, 42.85714286, 43.87755102, 44.89795918,
        45.91836735, 46.93877551, 47.95918367, 48.97959184, 50.        ]))
```

```
plt.scatter(x,y)
plt.xlabel('x')
plt.ylabel('y')
plt.title("Training Data")
plt.show()
```

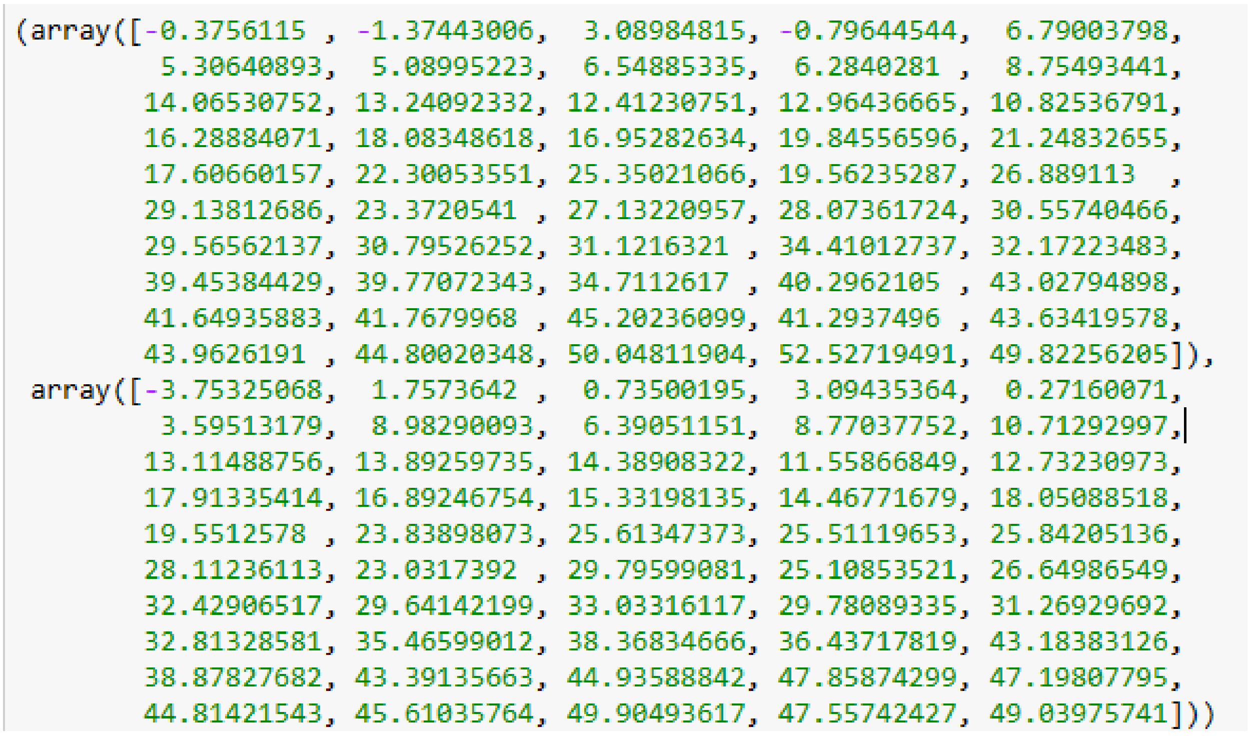
OUTPUT:



```
#Adding noise to the datapoints
x += np.random.uniform(-4,4,50)
y += np.random.uniform(-4,4,50)
n = len(x) #Number of Data  Points
```

x,y

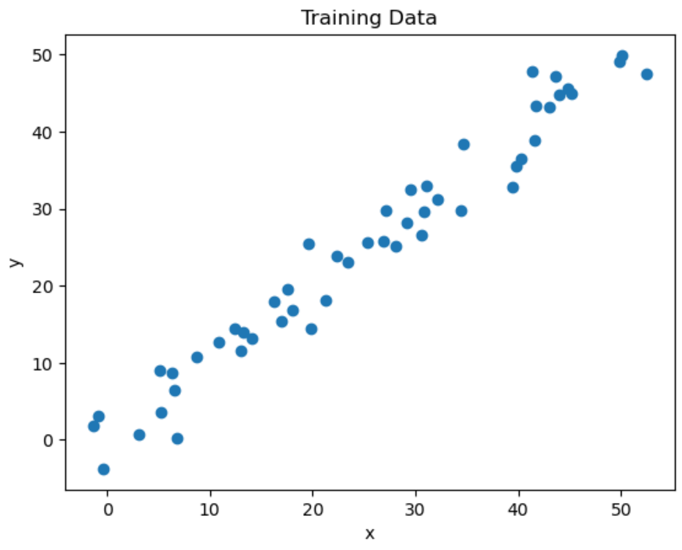
OUTPUT:



```
(array([-0.3756115 , -1.37443006,  3.08984815, -0.79644544,  6.79003798,
         5.30640893,  5.08995223,  6.54885335,  6.2840281 ,  8.75493441,
        14.06530752, 13.24092332, 12.41230751, 12.96436665, 10.82536791,
        16.28884071, 18.08348618, 16.95282634, 19.84556596, 21.24832655,
        17.60660157, 22.30053551, 25.35021066, 19.56235287, 26.889113  ,
        29.13812686, 23.3720541 , 27.13220957, 28.07361724, 30.55740466,
        29.56562137, 30.79526252, 31.1216321 , 34.41012737, 32.17223483,
        39.45384429, 39.77072343, 34.7112617 , 40.2962105 , 43.02794898,
        41.64935883, 41.7679968 , 45.20236099, 41.2937496 , 43.63419578,
        43.9626191 , 44.80020348, 50.04811904, 52.52719491, 49.82256205]),
 array([-3.75325068,  1.7573642 ,  0.73500195,  3.09435364,  0.27160071,
         3.59513179,  8.98290093,  6.39051151,  8.77037752, 10.71292997,
        13.11488756, 13.89259735, 14.38908322, 11.55866849, 12.73230973,
        17.91335414, 16.89246754, 15.33198135, 14.46771679, 18.05088518,
        19.5512578 , 23.83898073, 25.61347373, 25.51119653, 25.84205136,
        28.11236113, 23.0317392 , 29.79599081, 25.10853521, 26.64986549,
        32.42906517, 29.64142199, 33.03316117, 29.78089335, 31.26929692,
        32.81328581, 35.46599012, 38.36834666, 36.43717819, 43.18383126,
        38.87827682, 43.39135663, 44.93588842, 47.85874299, 47.19807795,
        44.81421543, 45.61035764, 49.90493617, 47.55742427, 49.03975741]))
```

#plot of training data
plt.scatter(x,y)
plt.xlabel('x')
plt.ylabel('y')
plt.title("Training data")
plt.show()

OUTPUT:



import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()
X=tf.placeholder("float")

```
Y=tf.placeholder("float")

W=tf.Variable(np.random.randn(),name="W")
b=tf.Variable(np.random.randn(),name="b")

learning_rate = 0.01
training_epochs = 1000

#Hypothesis
y_pred = tf.add(tf.multiply(X,W),b)

#MSE Cost Function
cost = tf.reduce_sum(tf.pow(y_pred-Y,2))/(2*n)

#Gradient Descent Optimizer
optimizer =
tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)

#Global Variable Initializer
init = tf.global_variables_initializer()

#Starting TensorFlow Session
with tf.Session() as sess:
    sess.run(init)
    for epoch in range(training_epochs):
        for(_x,_y) in zip(x,y):
            sess.run(optimizer, feed_dict = {X : _x, Y : _y})

        if(epoch + 1) % 50 == 0:
            c = sess.run(cost, feed_dict = {X : _x, Y : _y})
            print("Epoch",(epoch+1),": cost =", c, "W =", sess.run(W),
"b=",sess.run(b))

    #Storing necessary values to be used outside the session
    training_cost = sess.run(cost,  feed_dict = {X : _x, Y : _y})
    weight = sess.run(W)
    bias = sess.run(b)
```

OUTPUT:

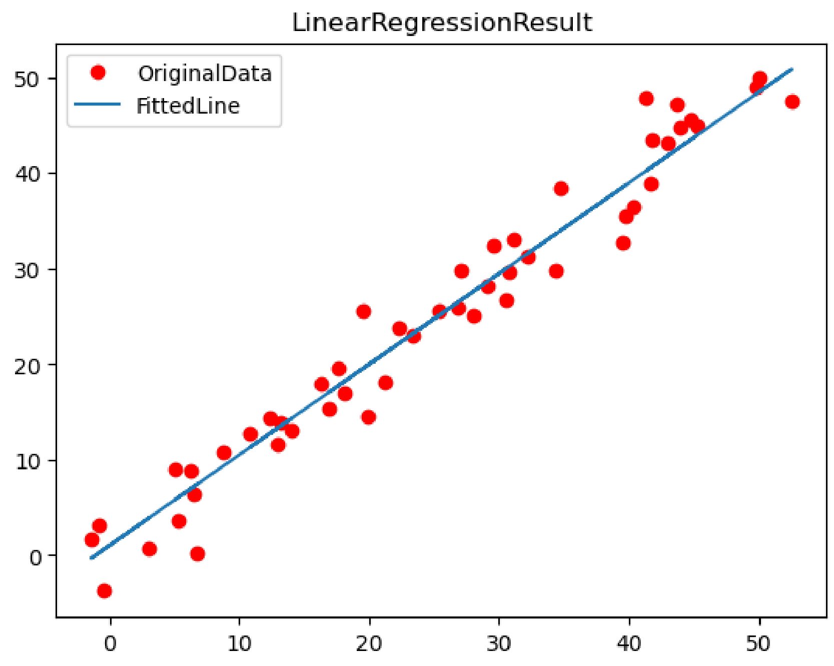
```
Epoch 800 :cost= 0.03479725 W= 0.7661446 b= 8.090939
Epoch 850 :cost= 0.03498694 W= 0.76410013 b= 8.201936
Epoch 900 :cost= 0.03515396 W= 0.76230603 b= 8.299341
Epoch 950 :cost= 0.03530045 W= 0.760731 b= 8.38486
Epoch 1000 :cost= 0.03542972 W= 0.7593485 b= 8.459916
```

```
#Calculating the predictions

predictions=weight*x + bias
```

print("Training cost =",training_cost,"Weight=",weight,"bias=",bias,'\n')

OUTPUT:

```
Training cost = 0.03542972
Weight= 0.7593485
bias= 8.459916
#Plotting the results
plt.plot(x,y,'ro',label='Original data')
plt.plot(x,prediction,label)
```



RESULT:

The given linear regression model is implemented using Tensorflow. The model is trained and tested and the line of regression is plotted for the data

AD18511- DEEP LEARNING LABORATORY
LINEAR REGRESSION USING SCIKIT LEARN

DATE:
EX.NO: 2(b)

AIM:

To implement Linear Regression using Tensorflow from scratch.

DESCRIPTION:

Linear regression is a statistical method used to model and analyze the relationship between a dependent variable and one or more independent variables. It aims to find the best-fitting straight line (a linear equation) that represents the pattern of the data, allowing for predictions and understanding the nature of the connection between the variables.
The linear regression equation can be written as:

$$y = \hat{0} + \hat{1} * x$$

where:

y is the dependent variable (the one being predicted or explained).
x is the independent variable (the one used for making predictions).
b0 is the y-intercept.
b1 is the slope or coefficient of the independent variable.

The least squares estimates of β_0 and β_1 are:

$$\begin{aligned}\hat{\beta}_1 &= \frac{\sum_{i=1}^{n}(X_i - \bar{X})(Y_i - \bar{Y})}{\sum_{i=1}^{n}(X_i - \bar{X})^2} \\ \hat{\beta}_0 &= \bar{Y} - \hat{\beta}_1\bar{X}\end{aligned}$$

The primary goal of linear regression is to estimate the values of b0 and b1 such that the line fits the data points as closely as possible. This is achieved by minimizing the sum of the squared differences between the predicted y-values and the actual y-values, known as the "method of least squares."

PROGRAM:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
dataset=pd.read_csv('/content/Salary_Data.csv')
dataset.head()#in json format
```

OUTPUT:

```
[{"index":0,"YearsExperience":"1.1","Salary":"39343.0"},{"index":1,"YearsExperience":"1.3","
Salary":"46205.0"},{"index":2,"YearsExperience":"1.5","Salary":"37731.0"},{"index":3,"Years
Experience":"2.0","Salary":"43525.0"},{"index":4,"YearsExperience":"2.2","Salary":"39891.0"}
]
```

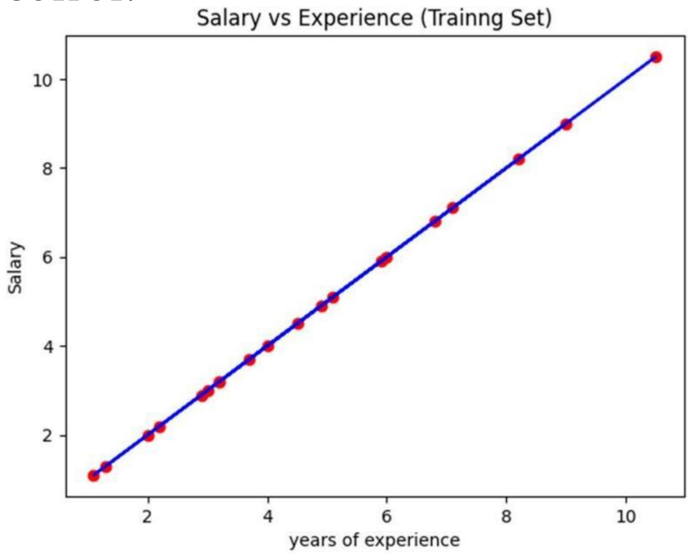
```
#data preprocessing
X=dataset.iloc[:,:-1].values #independent variable array x =all rows and columns
y=dataset.iloc[:,:1].values #dependent variable vector y= all rows and last column alone
```

OUTPUT:

```
array([[ 1.1], [ 1.3], [ 1.5], [ 1.5], [ 2. ],[ 2.2], [ 2.9], [ 3. ], [ 3.2], [ 3.2], [ 3.7], [ 3.9], [ 4. ], [ 4. ],
[ 4.1], [ 4.5], [ 4.9], [ 5.1], [ 5.3], [ 5.9], [ 6. ], [ 6.8], [ 7.1], [ 7.9], [ 8.2], [ 8.7], [ 9. ], [ 9.5], [9.6],
[10.3], [10.5]])
```

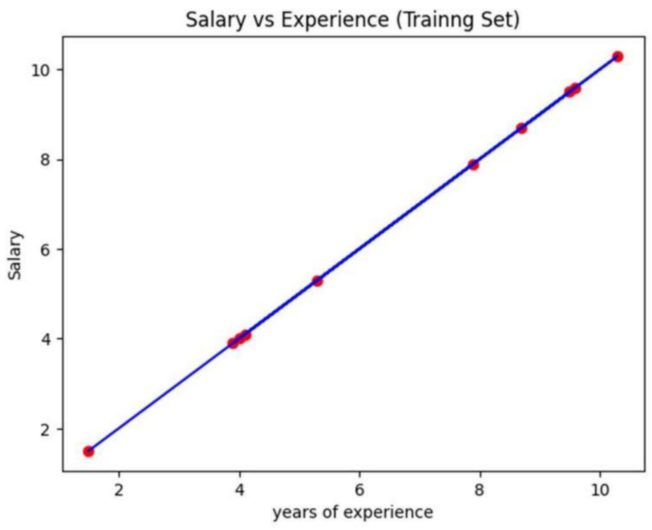
```
#splitting the dataset
from sklearn.model_selection
import train_test_split
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=1/3,random_state=0) #fitting the regression model
from sklearn.linear_model
import LinearRegression
regressor=LinearRegression()
regressor.fit(X_train,y_train)

#Predicting the test results
y_pred=regressor.predict(X_test)

#visualizing the results
#plot for the TRAIN
get_ipython().run_line_magic('matplotlib','inline')
plt.scatter(X_train,y_train,color='red')#plotting the observation line plt.plot(X_train,regressor.predict(X_train),color='blue')
plt.title("Salary vs Experience (Trainng Set)")#stating the title of the graph
plt.xlabel("years of experience")#adding name of x-axis
plt.ylabel("Salary")
plt.show()
```

OUTPUT:



```
#visualizing the results #plot for the train
plt.scatter(X_test,y_test,color='red')#plotting the observation line
plt.plot(X_test,regressor.predict(X_test),color='blue')
plt.title("Salary vs Experience (Trainng Set)")#stating the title of the graph

plt.xlabel("years of experience")#adding name of x-axis
plt.ylabel("Salary")
plt.show()
```

OUTPUT:



RESULT:

The given linear regression model is implemented using scikit learn module. The model is trained and tested and the line of regression is plotted for the data.

DATE:

PERCEPTRON MODEL WITHOUT USING PACKAGE.

EX.NO: 3(a)

AIM:

To implement perceptron from scratch and use it to for classification in the iris dataset.

PERCEPTRON DISCRIPTION:

- A Perceptron is an algorithm used for supervised learning of binary classifiers. Binary classifiers decide whether an input, usually represented by a series of vectors, belongs to a specific class.
- In short, a perceptron is a single-layer neural network. They consist of four main parts including input values, weights and bias, net sum, and an activation function.
- Working of Perceptron:

```
Step 1: X1W1 +X2W2  + .......  +XnWn
  Σ Wi *Xi

Step 2: Unit step activation function

  If   Σ Wi *Xi+ b > 0:
      Output = 1
  else:
      Output =0
```

ALGORITHM:

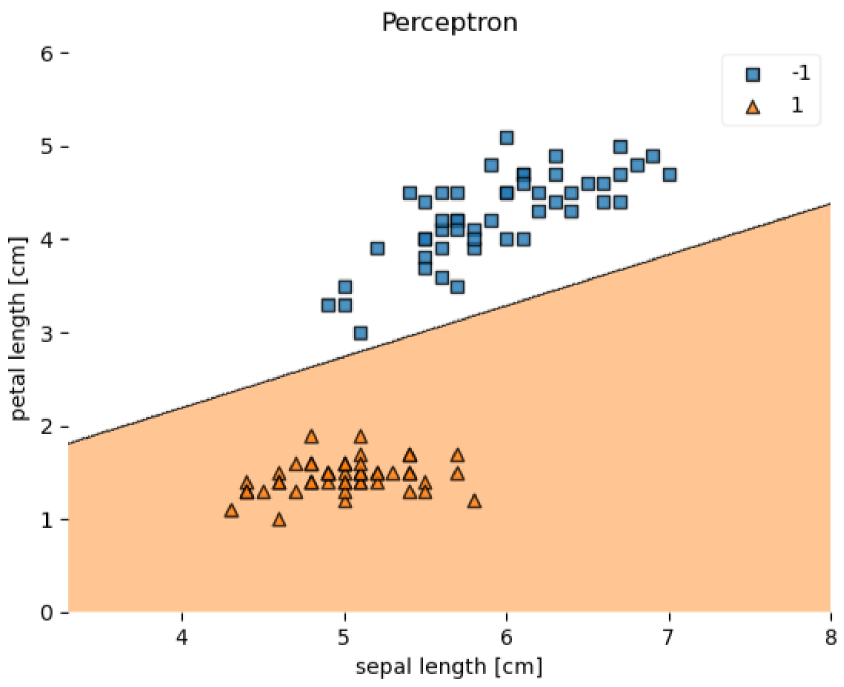
1. Start
2. import numpy
3. declare class perceptron with learning rate = 0.01, epochs = 50.
4. Define functions train, net input and predict.
5. import pandas and read the csv file into a dataframe.
6. plot the graph.
7. Stop.

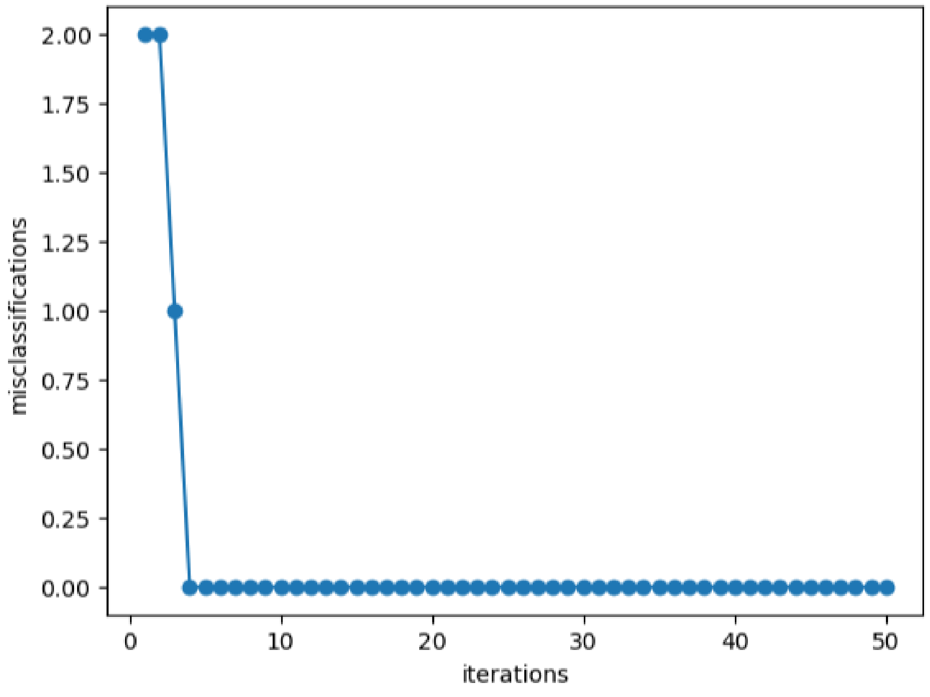
PROGRAM:

```
import numpy as np

class Perceptron(object):

   def __init__(self,eta=0.01,epochs=50):
      self.eta=eta
      self.epochs=epochs

   def train(self,X,y):
      self.w_=np.zeros(1+X.shape[1])
      self.errors_=[]

      for _ in range(self.epochs):
          errors=0
          for xi,target in zip(X,y):
              update=self.eta*(target - self.predict(xi))
              self.w_[1:] += update* xi
              self.w_[0] += update
              errors += int(update !=0.0)
          self.errors_.append(errors)
      return self

   def net_input(self,X):
      return np.dot(X,self.w_[1:] + self.w_[0])

   def predict(self,X):
      return np.where(self.net_input(X) > 0.0,1,-1)

import pandas as pd
df=pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data',header=None)

y=df.iloc[0:100,4].values
y=np.where(y=='Iris-setosa',1,-1)

X=df.iloc[0:100,[0,2]].values


import matplotlib.pyplot as plt
from mlxtend.plotting import plot_decision_regions

ppn=Perceptron()

ppn.train(X,y)
print('weights:%s'%ppn.w_)
plot_decision_regions(X,y,clf=ppn)
plt.title('Perceptron')
plt.xlabel('sepal length [cm]')
plt.ylabel('petal length [cm]')
plt.show()


plt.plot(range(1,len(ppn.errors_)+1),ppn.errors_,marker='o')
plt.xlabel('iterations')
plt.ylabel('misclassifications')
plt.show()
```

OUTPUT:

```
weights:[ 0.02   0.026 -0.104]
```





RESULT:

The Perceptron model is implemented using Tensorflow. The model is trained and tested and then the loss and accuracy are displayed.

DATE:
EX.NO: 3(b)

PERCEPTRON MODEL USING PACKAGES.

AIM:

To write a program that builds a perceptron model with the use of packages.

DESCRIPTION:

- A Perceptron is an algorithm used for supervised learning of binary classifiers. Binary classifiers decide whether an input, usually represented by a series of vectors, belongs to a specific class.
- In short, a perceptron is a single-layer neural network. They consist of four main parts including input values, weights and bias, net sum, and an activation function.
- Working of Perceptron:

```
Step 1: X1W1 +X2W2  + .......  +XnWn
          Σ Wi *Xi
Step 2: Unit step activation function

     If   Σ Wi *Xi+ b > 0:
              Output = 1
    else:
              Output =0
```

PROGRAM:

```
import pandas as pd
df = pd.read_csv("/home/user/anaconda3/lib/python3.10/site-packages/bokeh/sampledata/_data/iris.csv")
y = df.iloc[0:100, 4].values
y = np.where(y == "setosa", 1, -1)
X = df.iloc[0:100, [0,2]].values
from sklearn.linear_model import Perceptron as per
from sklearn.model_selection import train_test_split as tts
from sklearn.metrics import accuracy_score as ac
x1,x2,y1,y2 = tts(X,y,test_size=0.2, random_state=42)
model = per(alpha=0.001, max_iter=100)
model.fit(x1,y1)
pred = model.predict(x2)
acc = ac(y2, pred)
print("acc=",acc)
```

OUTPUT:

```
acc= 1.0
```

RESULT:

Hence, the program to build a perceptron model using packages is complete and the output is verified.

DATE:

EX NO: 4

SOFTMAX REGRESSION.

AIM:

To write a program to build a softmax regression model using tensorflow.

DESCRIPTION:

- Softmax Regression or multimodal logistic regression is a generalization of logistic regression to the case where we want to handle multiple classes.
- Softmax Regression allows us to handle

yi ∈ {1,2,3,......,k} where k is the number of classes

$$Pi = \frac{e^{wiTx}}{\sum_{i=1}^{k} e^{wiTx}}$$

where $1 \leq i < k$

- Softmax function is the extension of sigmoid function in multiclass.

PROGRAM:

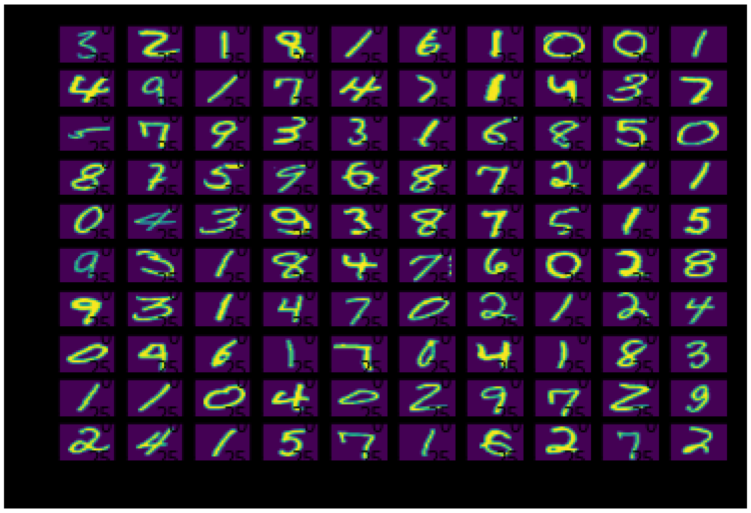
```
import tensorflow as tf
import tensorflow.compat.v1 as tf1
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

(X_train,Y_train),\
(X_val,Y_val)=tf.keras.datasets.mnist.load_data()
print("shape of features matrix:",X_train.shape)
print("shape of target matrix:",Y_train.shape)
```

OUTPUT:

```
shape of features matrix: (60000, 28, 28)
shape of target matrix: (60000,)
```

```
fig,ax=plt.subplots(10,10)
for i in range(10):
        for j in range(10):
        k=np.random.randint(0,X_train.shape[0])
        ax[i][j].imshow(X_train[k].reshape(28,28),aspect='auto')
plt.show()
```

OUTPUT:



```
num_features=784
num_labels=10
learning_rate=0.05
batch_size=128
num_steps=5001
train_dataset=X_train.reshape(-1,784)
train_labels=pd.get_dummies(Y_train).values
valid_dataset=X_val.reshape(-1,784)
valid_labels=pd.get_dummies(Y_val).values

graph=tf.Graph()
with graph.as_default():
        tf_train_dataset=tf1.placeholder(tf.float32,shape=(batch_size,num_features))
        tf_train_labels=tf1.placeholder(tf.float32,shape=(batch_size,num_labels))
        tf_valid_dataset=tf.constant(valid_dataset)
        weights=tf.Variable(tf.random.truncated_normal([num_features,num_labels]))
        biases=tf.Variable(tf.zeros([num_labels]))
        logits=tf.matmul(tf_train_dataset,weights)+biases
        loss=tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=tf_train_labels,logits=logits))
        optimizer=tf1.train.GradientDescentOptimizer(learning_rate).minimize(loss)
        train_prediction=tf.nn.softmax(logits)
        tf_valid_dataset=tf.cast(tf_valid_dataset,tf.float32)
        valid_prediction=tf.nn.softmax(tf.matmul(tf_valid_dataset,weights)+biases)
def accuracy(predictions,labels):
        correctly_predicted=np.sum(np.argmax(predictions,1)==np.argmax(labels,1))
        acc=(100.0*correctly_predicted)/predictions.shape[0]
        return acc
with tf1.Session(graph=graph) as session:
        tf1.global_variables_initializer().run()
        print("Initialized")
        for step in range(num_steps):
        offset = np.random.randint(0, train_labels.shape[0] - batch_size - 1)
        batch_data = train_dataset[offset:(offset + batch_size), :]
        batch_labels = train_labels[offset:(offset + batch_size), :]
        feed_dict = {tf_train_dataset: batch_data,tf_train_labels: batch_labels}
        _, l, predictions = session.run([optimizer, loss, train_prediction],feed_dict=feed_dict)
        if (step % 500 == 0):
        print("Minibatch loss at step {0}: {1}".format(step, l))
        print("Minibatch accuracy: {:.1f}%".format(accuracy(predictions, batch_labels)))
        print("Validation accuracy: {:.1f}%".format(accuracy(valid_prediction.eval(), valid_labels)))
```

OUTPUT:

```
Initialized
Minibatch loss at step 0: 3228.845703125
Minibatch accuracy: 14.1%
Validation accuracy: 30.5%
Minibatch loss at step 500: 470.5828857421875
Minibatch accuracy: 88.3%
Validation accuracy: 88.5%
Minibatch loss at step 1000: 810.6234741210938
Minibatch accuracy: 83.6%
Validation accuracy: 82.7%
Minibatch loss at step 1500: 217.19741821289062
Minibatch accuracy: 93.8%
Validation accuracy: 90.8%
Minibatch loss at step 2000: 400.3426818847656
Minibatch accuracy: 88.3%
Validation accuracy: 90.1%
Minibatch loss at step 2500: 587.1039428710938
Minibatch accuracy: 89.8%
Validation accuracy: 87.5%
Minibatch loss at step 3000: 793.615966796875
Minibatch accuracy: 85.9%
Validation accuracy: 88.4%
Minibatch loss at step 3500: 742.802978515625
Minibatch accuracy: 85.9%
Validation accuracy: 86.7%
Minibatch loss at step 4000: 181.66873168945312
Minibatch accuracy: 94.5%
Validation accuracy: 89.6%
Minibatch loss at step 4500: 692.54296875
Minibatch accuracy: 86.7%
Validation accuracy: 89.5%
Minibatch loss at step 5000: 200.63148498535156
Minibatch accuracy: 94.5%
Validation accuracy: 91.0%
```

RESULT:

Hence, the implementation of softmax regression model was done successfully using tensor flow.

DATE:

EX.NO: 5 **MULTI LAYER PERCEPTRON USING IMAGE CLASSIFICATION.**

AIM:

To write a program to build a multi layer perceptron model using tensor flow.

DESCRIPTION:

- Multi layer perceptron (MLP) is a supplement of feed forward neural network.
- It consists of three types of layers—the input layer, output layer and hidden layer .
- The input layer receives the input signal to be processed. The required task such as prediction and classification is performed by the output layer.
- An arbitrary number of hidden layers that are placed in between the input and output layer are the true computational engine of the MLP.
- Similar to a feed forward network in a MLP the data flows in the forward direction from input to output layer. The neurons in the MLP are trained with the back propagation learning algorithm.
- The computations taking place at every neuron in the output and hidden layer are as follows,

$$ox=Gb2+W2hx$$

$$hx=\Phi x=sb1+W1x$$

- with bias vectors b(1), b(2); weight matrices W(1), W(2) and activation functions G and s. The set of parameters to learn is the set $\theta = \{W(1), b(1), W(2), b(2)\}$.

PROGRAM:

```
import numpy as np
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense,Flatten
from keras.utils import to_categorical
import matplotlib.pyplot as plt

#load and preprocess the mnist dataset
(x_train,y_train),(x_test,y_test)=mnist.load_data()
x_train=x_train.reshape((-1,28*28))/255.0
x_test=x_test.reshape((-1,28*28))/255.0
y_train=to_categorical(y_train,num_classes=10)
y_test=to_categorical(y_test,num_classes=10)

#build the MLP model
model=Sequential()
model.add(Dense(128,activation='relu',input_shape=(28*28,)))
model.add(Dense(64,activation='relu'))
model.add(Dense(10,activation='softmax'))
model.compile(optimizer='adam',loss='categorical_crossentropy',metrics=['accuracy'])
model.compile(optimizer='adam', loss='categorical_crossentropy',metrics=['accuracy'])

#train the model
history=model.fit(x_train,y_train,epochs=50,batch_size=64,validation_split=0.2)
```

OUTPUT:

```
Epoch 1/50
750/750 [==============================] - 1s 1ms/step - loss: 0.3140 - accuracy: 0.9103 - val_loss: 0.1590 -
val_accuracy: 0.9539
Epoch 2/50
750/750 [==============================] - 1s 888us/step - loss: 0.1314 - accuracy: 0.9614 - val_loss: 0.1173 -
val_accuracy: 0.9647
Epoch 3/50
750/750 [==============================] - 1s 904us/step - loss: 0.0881 - accuracy: 0.9736 - val_loss: 0.1034 -
val_accuracy: 0.9696
Epoch 4/50
750/750 [==============================] - 1s 900us/step - loss: 0.0669 - accuracy: 0.9794 - val_loss: 0.0947 -
val_accuracy: 0.9714
Epoch 5/50
750/750 [==============================] - 1s 890us/step - loss: 0.0534 - accuracy: 0.9829 - val_loss: 0.0909 -
val_accuracy: 0.9743
…………
Epoch 48/50
750/750 [==============================] - 1s 900us/step - loss: 0.0049 - accuracy: 0.9987 - val_loss: 0.2094 -
val_accuracy: 0.9752
Epoch 49/50
750/750 [==============================] - 1s 914us/step - loss: 0.0053 - accuracy: 0.9983 - val_loss: 0.1832 -
val_accuracy: 0.9762
Epoch 50/50
750/750 [==============================] - 1s 906us/step - loss: 0.0020 - accuracy: 0.9994 - val_loss: 0.2035 -
val_accuracy: 0.9755
```

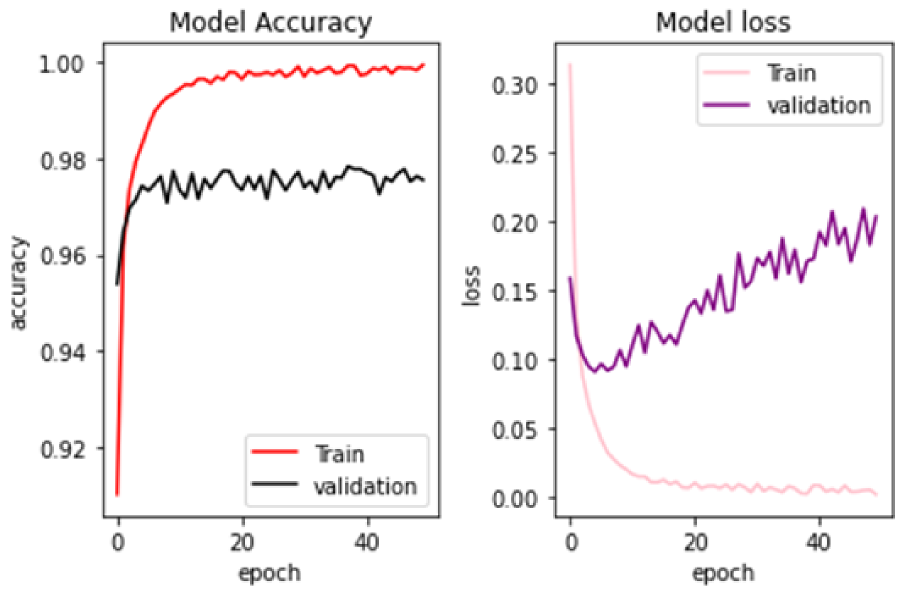
```
# visualize the training progress
plt.figure(figsize=(12,4))
```

OUTPUT:

```
<Figure size 864x288 with 0 Axes>
<Figure size 864x288 with 0 Axes>
```

```
#plot training and validation accuracy values
plt.subplot(1,2,1)
plt.plot(history.history['accuracy'],label='Train',color='r')
plt.plot(history.history['val_accuracy'],label='validation',color='black')
plt.title('Model Accuracy')
plt.xlabel("epoch")
plt.ylabel("accuracy")
plt.legend()

#plot training and validation loss values
plt.subplot(1,2,2)
plt.plot(history.history['loss'],label="Train",color='pink')
plt.plot(history.history['val_loss'],label='validation',color='purple')
plt.title('Model loss')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.legend()
plt.tight_layout()
plt.show()
```

OUTPUT:



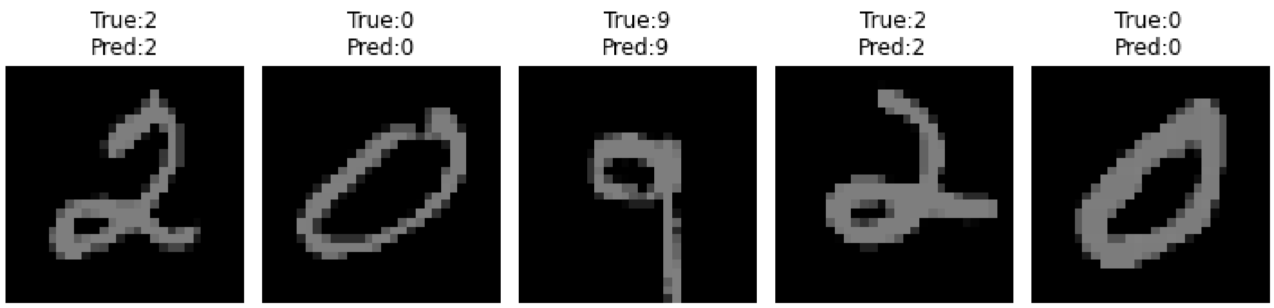
```
from sklearn.metrics import confusion_matrix,ConfusionMatrixDisplay
import random

#generate random indices for example predictions
example_indices=random.sample(range(len(x_test)),5)

#get the predicted labels for the example prediction
example_predictions=model.predict(x_test[example_indices])
example_predicted_labels=np.argmax(example_predictions,axis=1)
```

OUTPUT:

```
        1/1 [==============================] - 0s 11ms/step
```

```
# get the true labels for the example predictions
example_true_labels=np.argmax(y_test[example_indices],axis=1)


# plot the example predictions with images
        plt.figure(figsize=(10,6))
        for i,index in enumerate(example_indices):
             plt.subplot(2,5,i+1)
             plt.imshow(x_test[index].reshape(28,28),cmap='gray')
             plt.title(f"True:{example_true_labels[i]}\nPred:{example_predicted_labels[i]}")
             plt.axis('off')
        plt.tight_layout()
```

OUTPUT:



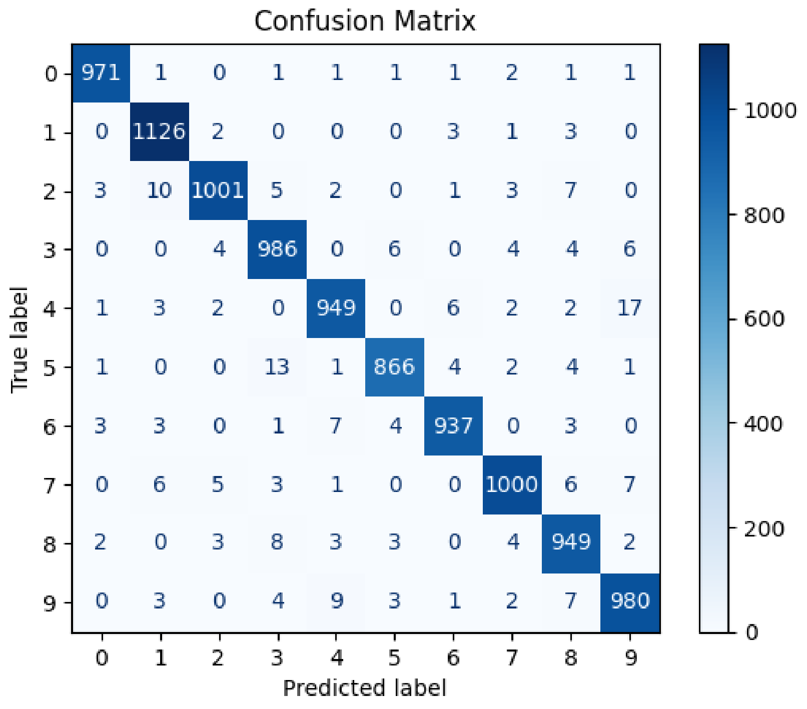
```
#compute confusion matrix
test_predictions=model.predict(x_test)
test_predicted_labels=np.argmax(test_predictions,axis=1)
conf_matrix=confusion_matrix(np.argmax(y_test,axis=1),test_predicted_labels)
```

OUTPUT:

```
        313/313 [==============================] - 0s 454us/step
```

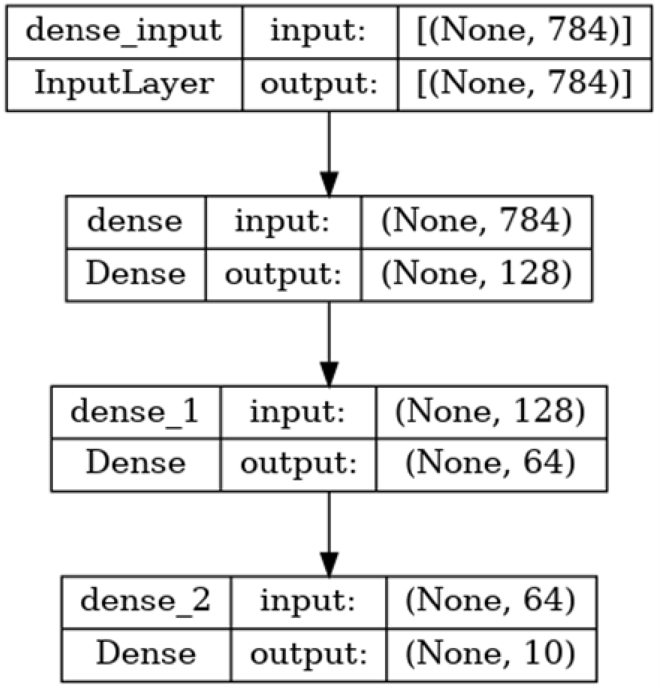
```
#display confusion matrix
plt.figure(figsize=(8,8))
ConfusionMatrixDisplay(conf_matrix,display_labels=range(10)).plot(cmap=plt.cm.Blues)
plt.title('confusion matrix')
plt.show()
```

OUTPUT:



```
from keras.utils.vis_utils import plot_model
from keras.utils import plot_model
plot_model(model, to_file='model.png', show_shapes=True)
```

OUTPUT:


dense_input | input: | [(None, 784)]
InputLayer | output: | [(None, 784)]
dense | input: | (None, 784)
Dense | output: | (None, 128)
dense_1 | input: | (None, 128)
Dense | output: | (None, 64)
dense_2 | input: | (None, 64)
Dense | output: | (None, 10)

RESULT:

The Perceptron model is implemented using Tensorflow. The model is trained and tested and then the loss and accuracy are displayed**.**

DATE:

EX.NO: 6(a) **EDGE DETECTION.**

AIM:

To perform edge detection for an image using three algorithms canny, sobel and prewitt using a CNN.

DESCRIPTION:

Edge Detection is a method of segmenting an image into regions of discontinuity. It is a widely used technique in digital image processing

Edge Detection Operators are of two types:

- Gradient – based operator which computes first-order derivations in a digital image like, Sobel operator, Prewitt operator, Robert operator
- Gaussian – based operator which computes second-order derivations in a digital image like, Canny edge detector, Laplacian of Gaussian

Sobel Operator: It is a discrete differentiation operator. It computes the gradient approximation of image intensity function for image edge detection. At the pixels of an image, the Sobel operator produces either the normal to a vector or the corresponding gradient vector. It uses two 3 x 3 kernels or masks which are convoluted with the input image to calculate the vertical and horizontal derivative approximations respectively –

Prewitt Operator: This operator is almost similar to the sobel operator. It also detects vertical and horizontal edges of an image. It is one of the best ways to detect the orientation and magnitude of an image. It uses the kernels or masks –

Canny Operator: It is a Gaussian-based operator in detecting edges. This operator is not susceptible to noise. It extracts image features without affecting or altering the feature. Canny edge detector have advanced algorithm derived from the previous work of Laplacian of Gaussian operator. It is widely used an optimal edge detection technique.

PROGRAM:

```
pip install opencv-python matplotlib numpy

import cv2
import numpy as np
import matplotlib.pyplot as plt
image=cv2.imread("/content/dog.jpg")
plt.imshow(image)

#convert it to grayscale
gray = cv2.cvtColor(image,cv2.COLOR_BGR2GRAY)
#show the grayscale image
plt.imshow(gray,cmap="gray")
plt.show()

blur=cv2.GaussianBlur(gray,(5,5),0)
```

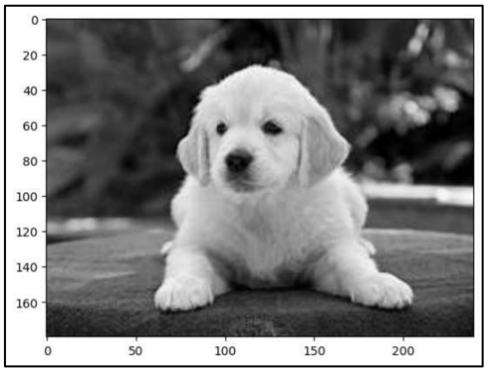
```
plt.imshow(blur,cmap="gray")
plt.show()

#perform the canny edge detector to detect image edges
edges=cv2.Canny(gray,threshold1=30,threshold2=100)#blur or gray or image

plt.imshow(edges,cmap="gray")
plt.show()

#perform the canny edge detector to detect image edges
edges=cv2.Canny(blur,threshold1=30,threshold2=100)#blur or gray or imag

plt.imshow(edges,cmap="gray")
plt.show()

#perform the canny edge detector to detect image edges
edges=cv2.Canny(image,threshold1=30,threshold2=100)#blur or gray or image
plt.imshow(edges,cmap="gray")
plt.show()

#Sobel operator
#Read the original image
img=cv2.imread("/home/user/Desktop/tiger.jpeg")
#display
cv2.imshow('Original',img)
cv2.waitKey(0)
#Convert to gray scale
img_gray=cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
#Blur the image for better edge detection
img_blur=cv2.GaussianBlur(img_gray,(3,3),0)

#Sobel Edge Detection
sobelx=cv2.Sobel(src=img_blur,ddepth=cv2.CV_64F,dx=1,dy=0,ksize=5)
sobely=cv2.Sobel(src=img_blur,ddepth=cv2.CV_64F,dx=0,dy=1,ksize=5)
sobelxy=cv2.Sobel(src=img_blur,ddepth=cv2.CV_64F,dx=1,dy=1,ksize=5)

#Display Sobel Edge Detetcion Images
cv2.imshow('Sobel X',sobelx)
cv2.waitKey(0)
cv2.imshow('Sobel Y',sobely)
cv2.waitKey(0)
cv2.imshow('Sobel XY using Sobel() function',sobelxy)
cv2.waitKey(0)

#cv2.destroyAllWindows()

#Display Sobel Edge Detetcion Images
cv2.imshow('Sobel X',sobelx)
cv2.waitKey(0)
cv2.imshow('Sobel Y',sobely)
cv2.waitKey(0)
cv2.imshow('Sobel XY using Sobel() function',sobelxy)
cv2.waitKey(0)
```

```
# Prewitt
# Defining the kernels
kernelx = np.array([[1, 1, 1], [0, 0, 0], [-1, -1, -1]])
kernely = np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]])

# Applying convolution
img_prewittx = cv2.filter2D(blur, -1, kernelx)
img_prewitty = cv2.filter2D(blur, -1, kernely)
img_prewitt = img_prewittx + img_prewitty

fig, axs = plt.subplots(2, 2)
titles = ['Original', 'Kernel X', 'Kernel Y', 'Kernel X and Y']
axs[0, 0].imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
axs[0, 0].set_title(titles[0])
axs[0, 0].axis('off')
axs[0, 1].imshow(img_prewitt, cmap='gray')
axs[0, 1].set_title(titles[3])
axs[0, 1].axis('off')
axs[1, 0].imshow(img_prewittx, cmap='gray')
axs[1, 0].set_title(titles[1])
axs[1, 0].axis('off')
axs[1, 1].imshow(img_prewitty, cmap='gray')
axs[1, 1].set_title(titles[2])
axs[1, 1].axis('off')
```

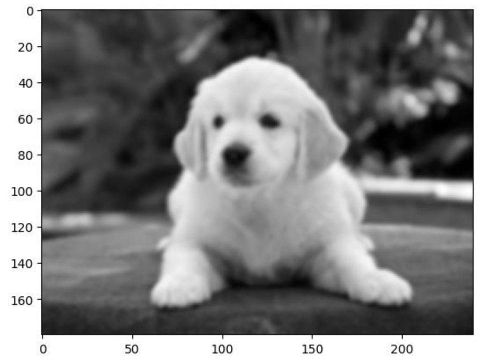
OUTPUT:

Original

Grayscale

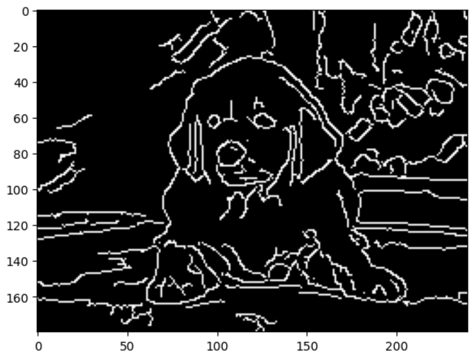


Blur

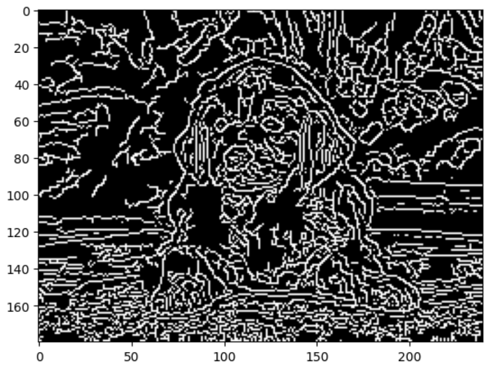


Canny edge detector

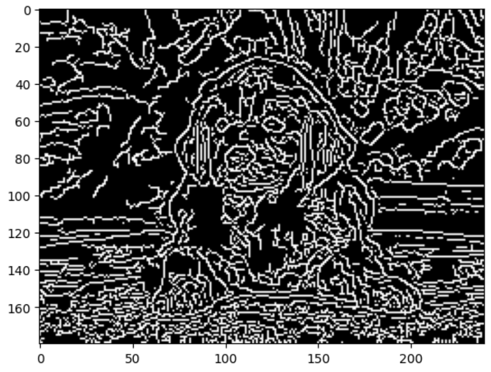
Blur



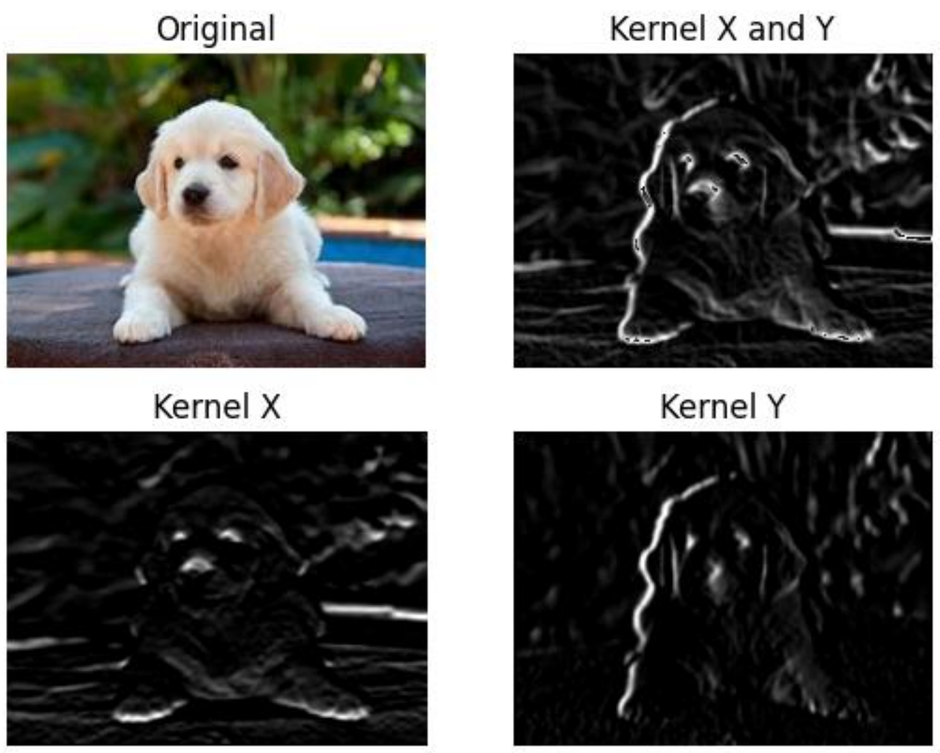
Image



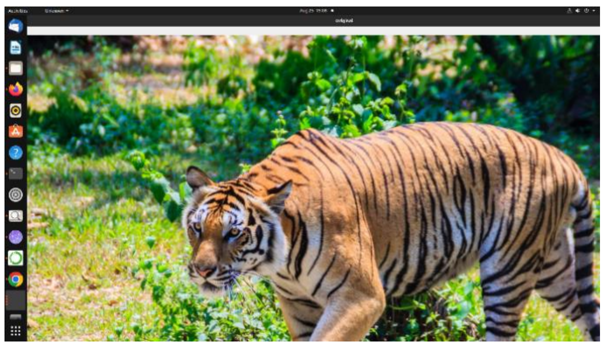
Image

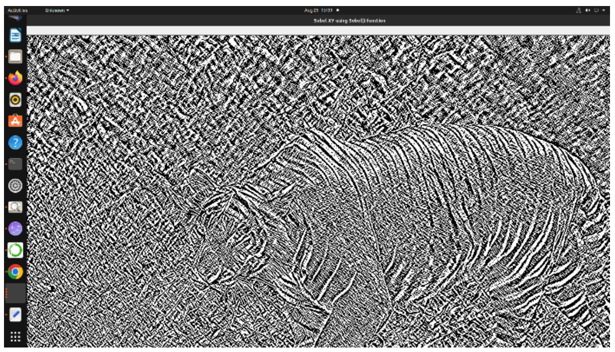


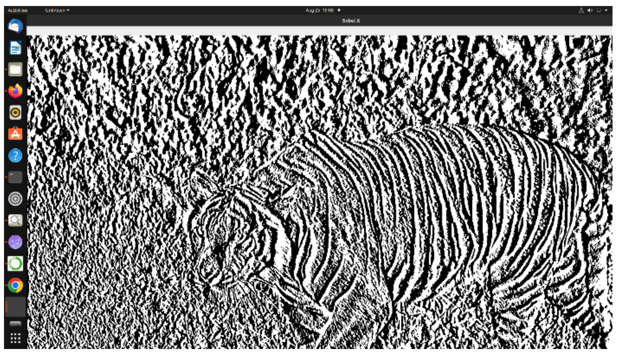
Prewitt:

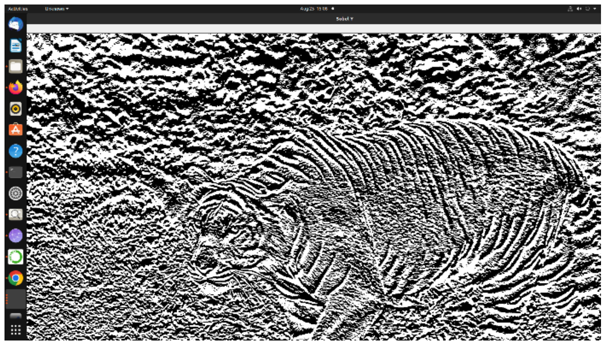


Sobel:









RESULT:

The image is loaded and edge detection is done using the 3 algorithms with a CNN successfully.

DATE:

EX.NO: 6(b) **EDGE DETECTION.**

AIM:

To implement edge detection for an image using Laplacian and Scharr operator.

DESCRIPTION:

Edge Detection is a method of segmenting an image into regions of discontinuity. It is a widely used technique in digital image processing.

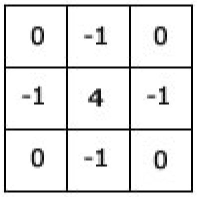
Sch-arr operator: This is a filtering method used to identify and highlight gradient edges/features using the first derivative. Performance is quite similar to the Sobel filter.

Scharr Operator [X-axis] = [-3 0 3; -10 0 10; -3 0 3];

Scharr Operator [Y-axis] = [3 10 3; 0 0 0; -3 -10 -3];

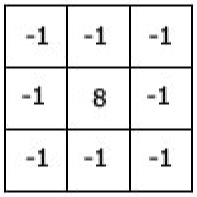
Laplacian operator: Unlike the Sobel edge detector, the Laplacian edge detector uses only one kernel.

It calculates second order derivatives in a single pass. Here's the kernel used for it:



0	-1	0
-1	4	-1
0	-1	0

The laplacian operator



-1	-1	-1
-1	8	-1
-1	-1	-1

The laplacian operator (include diagonals)

PROGRAM:

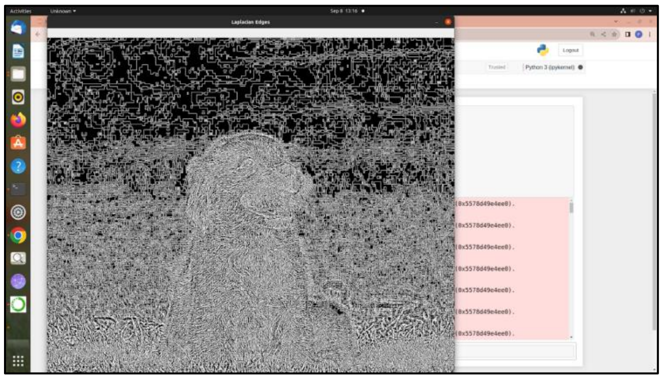
```
#Laplacian

import cv2

#load an image
image=cv2.imread('/home/user/Desktop/pro.jpg',cv2.IMREAD_GRAYSCALE)

#Apply Laplacian edge detection
edges=cv2.Laplacian(image,cv2.CV_64F)

#Display the resulting edges
cv2.imshow('Laplacian Edges',edges)
cv2.waitKey(0)
cv2.DestroyAllWindows()
```

OUTPUT:

```
shape of features matrix: (60000, 28, 28)
shape of target matrix: (60000,)
```

```
fig,ax=plt.subplots(10,10)
for i in range(10):
    for j in range(10):
    k=np.random.randint(0,X_train.shape[0])
    ax[i][j].imshow(X_train[k].reshape(28,28),aspect='auto')
plt.show()
```

OUTPUT:

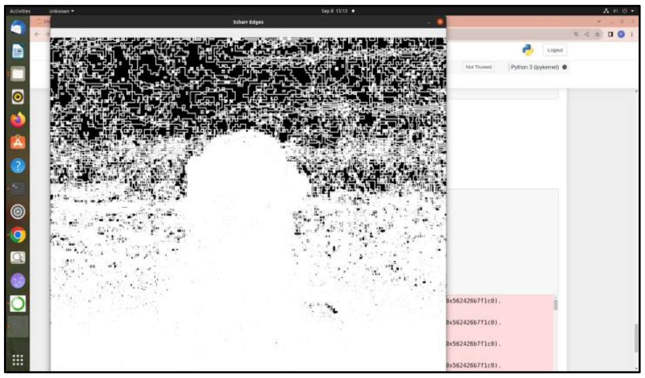


```
(0x5578d49e4ee0).
(0x5578d49e4ee0).
(0x5578d49e4ee0).
(0x5578d49e4ee0).
(0x5578d49e4ee0).
(0x5578d49e4ee0).
(0x5578d49e4ee0).
```

```
#Scharr

import cv2

#load an image
image=cv2.imread('/home/user/Desktop/pro.jpg',cv2.IMREAD_GRAYSCALE)

#Apply Scharr edge detection
scharr_x=cv2.Scharr(image,cv2.CV_64F,1,0)
scharr_y=cv2.Scharr(image,cv2.CV_64F,0,1)
#Laplacian

import cv2

#load an image
image=cv2.imread('/home/user/Desktop/pro.jpg',cv2.IMREAD_GRAYSCALE)

#Apply Laplacian edge detection
edges=cv2.Laplacian(image,cv2.CV_64F)

#Display the resulting edges
cv2.imshow('Laplacian Edges',edges)
cv2.waitKey(0)
cv2.DestroyAllWindows()
#Display the resulting edge map
cv2.imshow('Scharr Edges',edges)
```

```
cv2.waitKey(0)
cv2.DestroyAllWindows()
```

OUTPUT:



```
0x562426b7f1c0).

0x562426b7f1c0).

0x562426b7f1c0).

0x562426b7f1c0).
```

RESULT:

The image is loaded and edge detection is done using the Laplacian and Scharr operator algorithms.

DATE:

EX.NO: 7

KERNEL IMPLEMENTATION

AIM:

To implement different kernels and perform convolution on an image using CNN.

DESCRIPTION:

- In image processing, a kernel, convolution matrix, or mask is a small matrix used for blurring, sharpening, embossing, edge detection, and more.
- This is accomplished by doing a convolution between the kernel and an image.
- When each pixel in the output image is a function of the nearby pixels (including itself) in the input image, the kernel is that function.

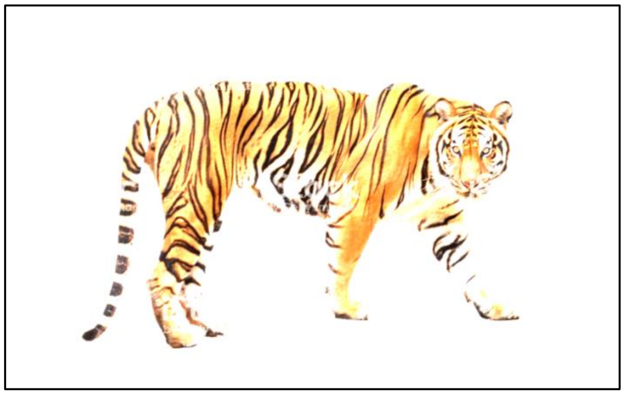
PROGRAM:

```
#Grascale image vector 0-255 pixels(2D)

import cv2
import numpy as np

#load an image
image=cv2.imread('/home/user/Desktop/pro.jpg')

#Define different kernels
kernel_identity=np.array([[0,0,0], [0,1,0], [0,0,0]])

kernel_edge_detection=np.array([[-1,-1,-1],[-1,8,-1],  [-1,-1,-1]])
kernel_sharpen=np.array([[0,-1,0], [-1,5,-1], [0,-1,0]])
kernel_blur=np.array([[1,2,1],[2,4,2], [1,2,1]])/16.0

#Apply kernels to the image
result_identity=cv2.filter2D(image,-1,kernel_identity)
result_edge_detection=cv2.filter2D(image,-1,kernel_edge_detection)
result_sharpen=cv2.filter2D(image,-1,kernel_sharpen)
result_blur=cv2.filter2D(image,-1,kernel_blur)

#Display the original and processed image
cv2.imshow('Original image',image)
cv2.imshow('Identity Kernel',result_identity)
cv2.imshow('Edge Detection Kernel',result_edge_detection)
cv2.imshow('Sharpen Kernel',result_sharpen)
cv2.imshow('Blur Kernel',result_blur)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

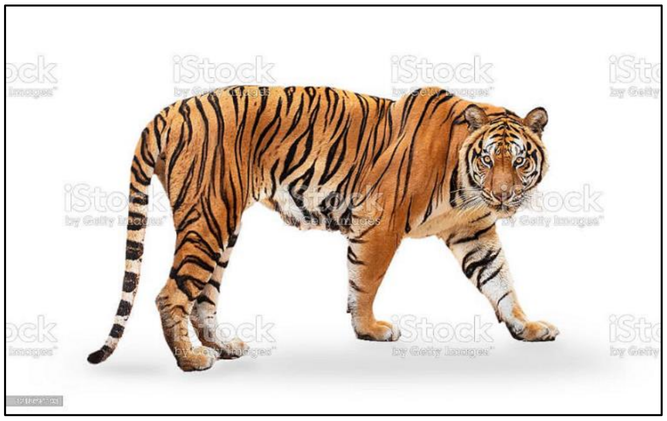
OUTPUT:

Original

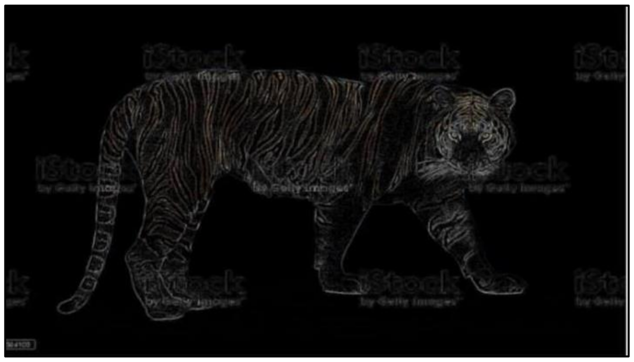
Identity Kernel



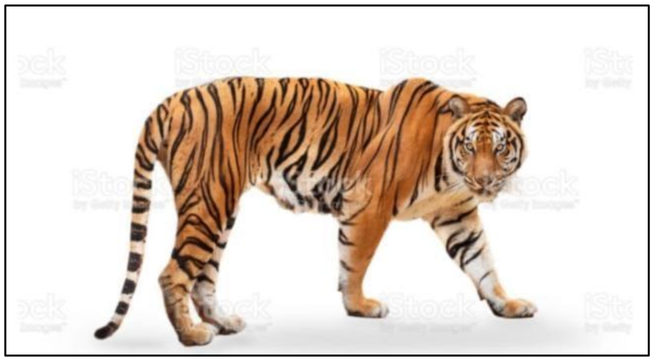
Sharpen



Edge detection



Blur Image



RESULT:

The kernels are created and applied to the image and the result is displayed successfully.

DATE:

EX.NO: 8(a) **IMPLEMENTATION OF BASIC CONVOLUTION NETWORK FOR MNIST DATASETS FROM SCRATCH**

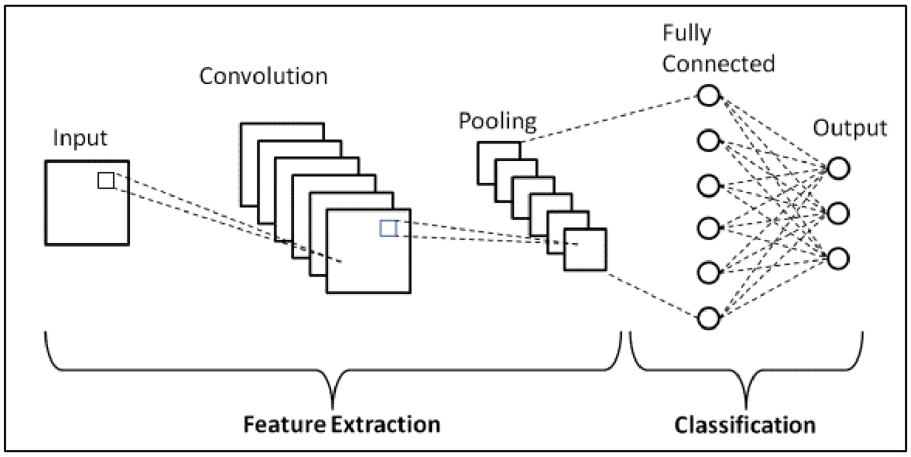
AIM:

To implement a basic Convolution Neural Network for MNIST datasets from scratch

DESCRIPTION:

- A **Convolutional Neural Network (CNN)** is a type of **artificial neural network** designed specifically for processing structured grid data, such as images and videos.

- CNNs have revolutionized computer vision tasks and have been widely used in various applications, including image classification, object detection, facial recognition, and more.



PROGRAM:

```
import tensorflow as tf
from tensorflow.keras import layers,models
from tensorflow.keras.datasets import mnist
#Load and preprocess the MNIST datasets
(train_images,train_labels),(test_images,test_labels)=mnist.load_data()
```

OUTPUT:

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz 11490434/11490434
[==============================] - 0s 0us/step
```

```
#Normalize pixel values to be between 0 and 1
train_images,test_images=train_images/255.0,test_images/255.0
#Create CNN model
model=models.Sequential([
        layers.Conv2D(32,(3,3),activation='relu',input_shape=(28,28,1)),
                    layers.MaxPooling2D((2,2)), layers.Conv2D(64,(3,3),activation='relu'),
                    layers.MaxPooling2D((2,2)),  layers.Flatten(),
                    layers.Dense(64,activation='relu'), layers.Dense(10,activation='softmax'),
                                   ])
#Compile the model
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',  metrics=['accuracy'])
#Print the model summary
model.summary()
#Train the model
model.fit(train_images.reshape(1,28,28,1),train_labels,epochs=5,validation_data=(test_images.reshape(1,28,28,1),test_lab
els))
```

OUTPUT:

```
Model: "sequential" _________________________________________________________________ Layer (type) Output
Shape Param # ================================================================= conv2d
(Conv2D) (None, 26, 26, 32) 320 max_pooling2d (MaxPooling2 (None, 13, 13, 32) 0 D) conv2d_1 (Conv2D) (None, 11,
11, 64) 18496 max_pooling2d_1 (MaxPoolin (None, 5, 5, 64) 0 g2D) flatten (Flatten) (None, 1600) 0 dense (Dense)
(None, 64) 102464 dense_1 (Dense) (None, 10) 650
================================================================= Total params: 121930 (476.29
KB) Trainable params: 121930 (476.29 KB) Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________ Epoch 1/5 1875/1875
[==============================] - 54s 28ms/step - loss: 0.0150 - accuracy: 0.9950 - val_loss: 0.0288 -
val_accuracy: 0.9908 Epoch 2/5 1875/1875 [==============================] - 53s 28ms/step - loss: 0.0104 -
accuracy: 0.9968 - val_loss: 0.0313 - val_accuracy: 0.9915 Epoch 3/5 1875/1875
[==============================] - 52s 28ms/step - loss: 0.0096 - accuracy: 0.9969 - val_loss: 0.0349 -
val_accuracy: 0.9908 Epoch 4/5 1875/1875 [==============================] - 53s 28ms/step - loss: 0.0067 -
accuracy: 0.9979 - val_loss: 0.0367 - val_accuracy: 0.9910 Epoch 5/5 1875/1875
[==============================] - 53s 28ms/step - loss: 0.0075 - accuracy: 0.9974 - val_loss: 0.0314 -
val_accuracy: 0.9925 313/313 [==============================] - 3s 8ms/step - loss: 0.0314 - accuracy: 0.9925
Test accuracy:(test_accuracy*100:.2f)%
```

```
#Evaluate the model on the test dataset
test_loss,test_accuracy=model.evaluate(test_images.reshape(-1,28,28,1),test_labels)
print(f"Test accuracy:{test_accuracy*100:.2f}%")
```

OUTPUT:

```
313/313 [==============================] - 3s 8ms/step - loss: 0.0314 - accuracy: 0.9925 Test

accuracy:99.25%
```

RESULT:

Thus the CNN architecture has been implemented successfully.

DATE:

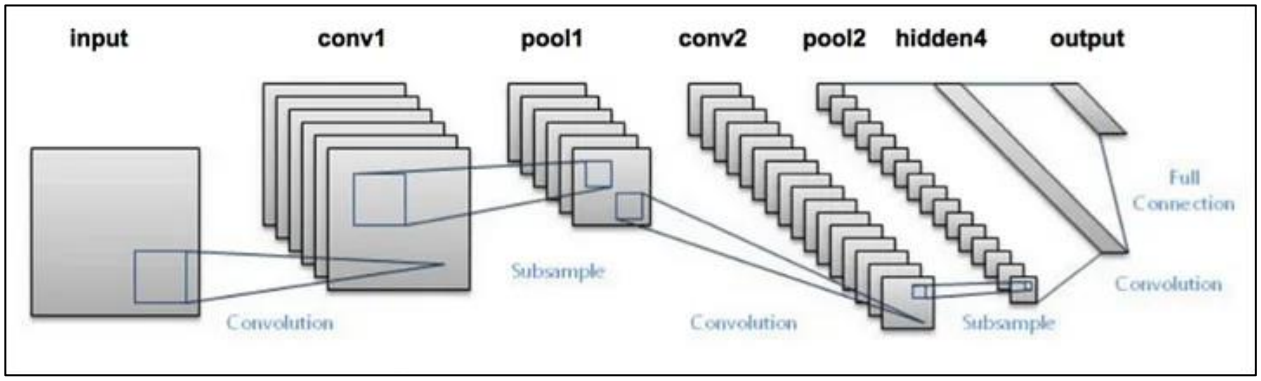
EX.NO: 8(b) **IMPLEMENTATION OF LENET ARCHITECTURE FROM SCRATCH**

AIM:

To use the LeNet architecture to perform handwritten digit detection using Tensorflow.

DESCRIPTION:

- LeNet-5, a pioneering 7-level convolutional network by LeCun et al in 1998, that classifies digits, was applied by several banks to recognise hand-written numbers on checks (cheques) digitized in 32x32 pixel grayscale input images.
- The ability to process higher resolution images requires larger and more convolutional layers, so this technique is constrained by the availability of computing resources



PROGRAM:

```
import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.keras import datasets,layers,models,losses

(x_train,y_train),(x_test,y_test)=tf.keras.datasets.mnist.load_data()
```

OUTPUT:

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
        11490434/11490434 [==============================] - 0s 0us/step
```

```
x_train=tf.pad(x_train,[[0,0],[2,2],[2,2]])/255
x_test=tf.pad(x_test,[[0,0],[2,2],[2,2]])/255
x_train=tf.expand_dims(x_train,axis=3,name=None)
x_test=tf.expand_dims(x_test,axis=3,name=None)
x_val=x_train[-2000:,:,:]
y_val=y_train[-2000:]
```

```
x_train=x_train[:-2000,:,:,:]
y_train=y_train[:-2000]

model=models.Sequential([
            layers.Conv2D(6,5,activation='tanh',input_shape=x_train.shape[1:]),
            layers.AveragePooling2D(2),
            layers.Activation('sigmoid'),
            layers.Conv2D(16,5,activation='tanh'),
            layers.AveragePooling2D(2),
            layers.Activation('sigmoid'),
            layers.Conv2D(120,5,activation='tanh'),
            layers.Flatten(),
            layers.Dense(84,activation='tanh'),
            layers.Dense(10,activation='softmax')
        ])
model.summary()
```

OUTPUT:

```
Model: "sequential" _________________________________________________________________ Layer (type) Output
Shape Param # ================================================================= conv2d (Conv2D)
(None, 28, 28, 6) 156 average_pooling2d (Average (None, 14, 14, 6) 0 Pooling2D) activation (Activation) (None, 14, 14, 6) 0
conv2d_1 (Conv2D) (None, 10, 10, 16) 2416 average_pooling2d_1 (Avera (None, 5, 5, 16) 0 gePooling2D) activation_1
(Activation) (None, 5, 5, 16) 0 conv2d_2 (Conv2D) (None, 1, 1, 120) 48120 flatten (Flatten) (None, 120) 0 dense (Dense)
(None, 84) 10164 dense_1 (Dense) (None, 10) 850
================================================================= Total params: 61706 (241.04 KB)
Trainable params: 61706 (241.04 KB) Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
from keras.src.engine.training import optimizer
model.compile(optimizer='adam',loss=losses.sparse_categorical_crossentropy,metrics=['accuracy'])
history=model.fit(x_train,y_train,batch_size=64,epochs=30,validation_data=(x_val,y_val))
```

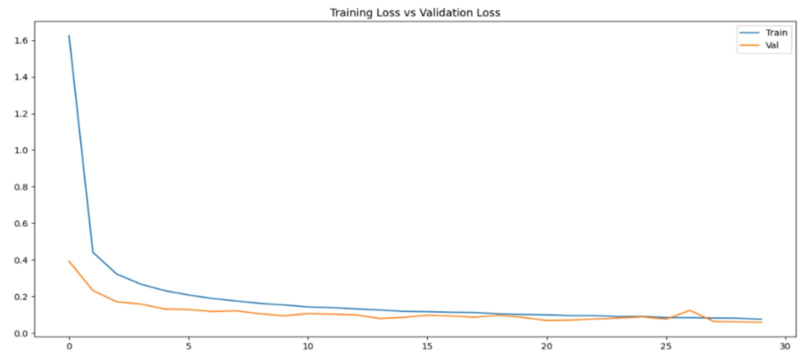
OUTPUT:

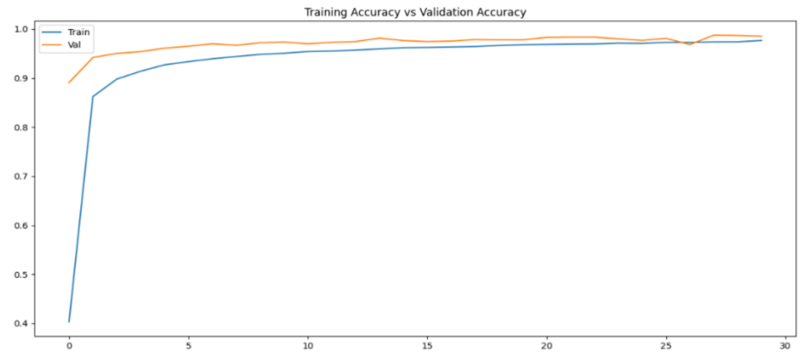
```
Epoch 27/30 907/907 [==============================] - 31s 35ms/step - loss: 0.0845 - accuracy: 0.9727 - val_loss:
0.1241 - val_accuracy: 0.9685 Epoch 28/30 907/907 [==============================] - 30s 33ms/step - loss: 0.0823
- accuracy: 0.9739 - val_loss: 0.0636 - val_accuracy: 0.9875 Epoch 29/30 907/907 [==============================]
- 30s 33ms/step - loss: 0.0815 - accuracy: 0.9740 - val_loss: 0.0621 - val_accuracy: 0.9870 Epoch 30/30 907/907
[==============================] - 35s 39ms/step - loss: 0.0755 - accuracy: 0.9767 - val_loss: 0.0582 - val_accuracy:
0.9855
```

```
fig, axs = plt.subplots(2, 1, figsize=(15,15))

axs[0].plot(history.history['loss'])
axs[0].plot(history.history['val_loss'])
axs[0].title.set_text('Training Loss vs Validation Loss')
axs[0].legend(['Train', 'Val'])
axs[1].plot(history.history['accuracy'])
axs[1].plot(history.history['val_accuracy'])
axs[1].title.set_text('Training Accuracy vs Validation Accuracy')
axs[1].legend(['Train', 'Val'])
```

```
results = model.evaluate(x_test, y_test)
print("Loss = {}, Accuray = {}".format(results[0], results[1]))
```

OUTPUT:





RESULT:

Thus the LeNet architecture has been implemented successfully

DATE:

EX.NO: 9 **IMPLEMENTATION OF ALEXNET ARCHITECTURE.**

AIM:

To implement alexnet architecture for image recognition and classification tasks.

DESCRIPTION:

- The AlexNet CNN architecture won the 2012 ImageNet ILSVRC challenges of deep learning algorithm by a large variance by achieving 17% with top-5 error rate as the second best achieved 26%!
- It was introduced by Alex Krizhevsky (name of founder), The Ilya Sutskever and Geoffrey Hinton are quite similar to LeNet-5, only much bigger and deeper and it was introduced first to stack convolutional layers directly on top of each other models, instead of stacking a pooling layer top of each on CN network convolutional layer.
- AlexNNet has 60 million parameters as AlexNet has total 8 layers, 5 convolutional and 3 fully connected layers.
- AlexNNet is first to execute (ReLUs) Rectified Linear Units as activation functions.
- it was the first CNN architecture that uses GPU to improve the performance.

PROGRAM:

```
# Importing Libraries
import tensorflow as tf
from tensorflow import keras
from keras.layers import Conv2D, MaxPooling2D, Dense, Flatten, Dropout, BatchNormalization
from tensorflow.keras.optimizers import Adam,SGD
from tensorflow.keras.callbacks import TensorBoard

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

import os
import time


train_df = pd.read_csv('/fashion-mnist_train.csv',sep=',')
test_df = pd.read_csv('/fashion-mnist_test.csv', sep = ',')

from google.colab import drive
drive.mount('/content/drive')

train.head (10)
train_data= np.array(train_df, dtype= 'float32')
test_data= np.array(test_df, dtype= 'float32')

x_train = train_data[:,1:]/255
y_train = train_data[:,0]
x_test= test_data[:,1:]/255
y_test=test_data[:,0]
```

```
# Example of training label content
print(y_train[0], y_train[43], y_train[1923])
print("Minimum value of training labels", y_train.min())
print("Maximum value of training labels", y_train.max())
```

OUTPUT:

```
0.0 6.0 2.0
Minimum value of training labels 0.0
Maximum value of training labels 9.0
```

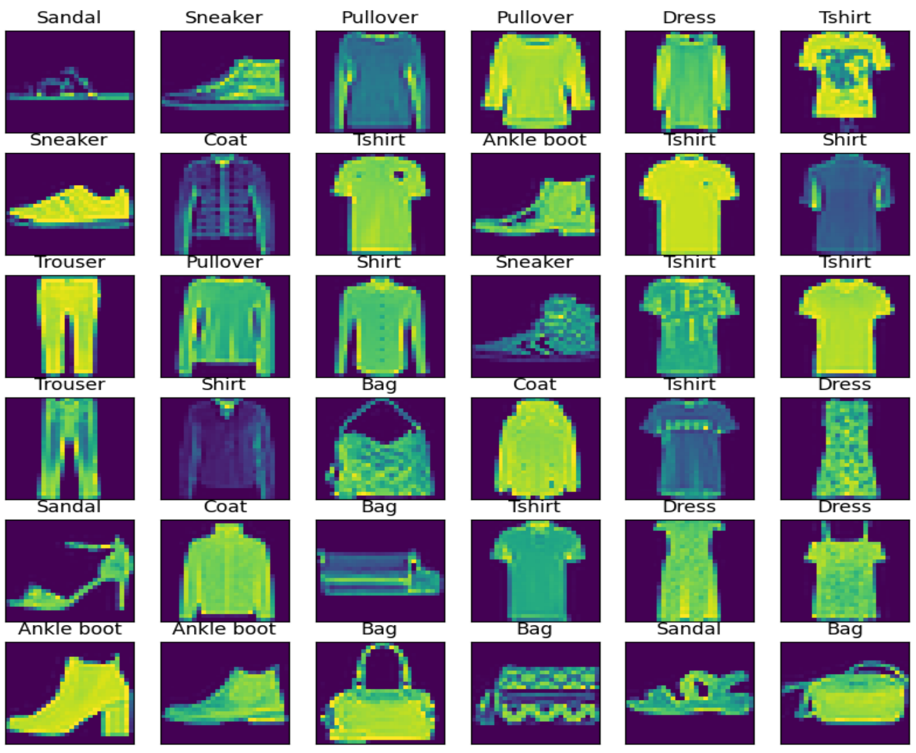
```
x_train,x_validate,y_train,y_validate = train_test_split(x_train,y_train,test_size = 0.2,random_state = 12345)
x_train.shape, x_validate.shape,y_train.shape,y_validate.shape
```

OUTPUT:

```
((8000, 784), (2000, 784), (8000,), (2000,))
```

```
class_names= ['Tshirt', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
plt.figure(figsize=(10,10))
for i in range(36):
    plt.subplot(6,6, i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(x_train[i].reshape((28,28)))
    label_index= int(y_train[i])
    plt.title(class_names[label_index])
plt.show()
```

OUTPUT:



a

```
image_rows = 28
image_cols = 28
batch_size = 4096
image_shape = (image_rows,image_cols,1)

x_train= x_train.reshape(x_train.shape[0], *image_shape)
x_test = x_test.reshape(x_test.shape[0],*image_shape)
x_validate = x_validate.reshape(x_validate.shape[0],*image_shape)
x_train.shape, x_test.shape, x_validate.shape

# Alexnet
model= tf.keras.Sequential([Conv2D(filters=96, kernel_size=(11,11), strides=(4,4), activation= 'relu', input_shape= image_shape),
                            BatchNormalization(),
                            MaxPooling2D(pool_size=(3,3), strides=(2,2), padding='same'),
                            Conv2D(filters=256, kernel_size=(5,5), strides=(1,1), activation='relu', padding='same'),
                            BatchNormalization(),
                            MaxPooling2D(pool_size=(3,3), strides=(2,2), padding='same'),
                            Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), activation='relu', padding='same'),
                            BatchNormalization(),
                            Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), activation='relu', padding='same'),
                            BatchNormalization(),
                            Conv2D(filters=256, kernel_size=(3,3), strides=(1,1), activation='relu', padding='same'),
                            BatchNormalization(),
                            MaxPooling2D(pool_size=(3,3), strides=(2,2), padding='same'),
                            Flatten(),
                            Dense(4096, activation='relu'),
                            Dropout(0.5),
                            Dense(4096, activation='relu'),
                            Dropout(0.5),
                            Dense(10, activation='softmax')
                            ])
model.summary()
```

OUTPUT:

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 5, 5, 96)          11712

 batch_normalization (Batch  (None, 5, 5, 96)          384
 Normalization)

 max_pooling2d (MaxPooling2  (None, 3, 3, 96)          0
 D)

 conv2d_1 (Conv2D)           (None, 3, 3, 256)         614656

 batch_normalization_1 (Bat  (None, 3, 3, 256)         1024
 chNormalization)

 max_pooling2d_1 (MaxPoolin  (None, 2, 2, 256)         0
 g2D)

 conv2d_2 (Conv2D)          (None, 2, 2, 384)         885120

 batch_normalization_2 (Bat  (None, 2, 2, 384)         1536
 chNormalization)

 conv2d_3 (Conv2D)          (None, 2, 2, 384)         1327488

 batch_normalization_3 (Bat  (None, 2, 2, 384)         1536
 chNormalization)

 conv2d_4 (Conv2D)          (None, 2, 2, 256)         884992

 batch_normalization_4 (Bat  (None, 2, 2, 256)         1024
 chNormalization)

 max_pooling2d_2 (MaxPoolin  (None, 1, 1, 256)         0
 g2D)

 flatten (Flatten)          (None, 256)               0

 dense (Dense)              (None, 4096)              1052672

 dropout (Dropout)          (None, 4096)              0

 dense_1 (Dense)            (None, 4096)              16781312

 dropout_1 (Dropout)        (None, 4096)              0

 dense_2 (Dense)            (None, 10)                40970

=================================================================
Total params: 21604426 (82.41 MB)
Trainable params: 21601674 (82.40 MB)
Non-trainable params: 2752 (10.75 KB)
_________________________________________________________________
```

```
from tensorflow.keras.optimizers import SGD

def lr_schedule(epoch):
    lr = 0.01
    if epoch > 50:
        lr *= 0.1
    elif epoch > 75:
        lr *= 0.01
    return lr

sgd = SGD(learning_rate=0.01, momentum=0.9, nesterov=True)
model.compile(loss="sparse_categorical_crossentropy", optimizer=sgd, metrics=["accuracy"])
from tensorflow.keras.callbacks import LearningRateScheduler
lr_scheduler = LearningRateScheduler(lr_schedule)
```

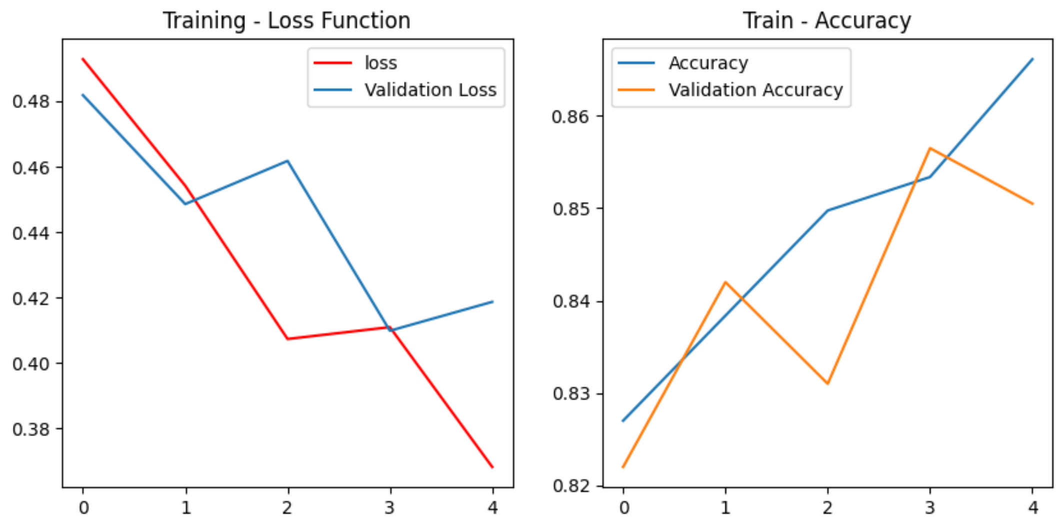
```
# train!
early_stopping_cb = keras.callbacks.EarlyStopping(monitor='val_loss', patience=30, verbose=1, mode='min')
history= model.fit(x_train, y_train, epochs=5, verbose=1, callbacks=[early_stopping_cb],
        validation_data=(x_validate, y_validate))
```

OUTPUT:

```
Epoch 1/5
250/250 [==============================] - 220s 881ms/step - loss: 0.4928 - accuracy: 0.8270 - val_loss:
0.4818 - val_accuracy: 0.8220
Epoch 2/5
250/250 [==============================] - 218s 872ms/step - loss: 0.4541 - accuracy: 0.8384 - val_loss:
0.4485 - val_accuracy: 0.8420
Epoch 3/5
250/250 [==============================] - 213s 852ms/step - loss: 0.4073 - accuracy: 0.8497 - val_loss:
0.4617 - val_accuracy: 0.8310
Epoch 4/5
250/250 [==============================] - 216s 863ms/step - loss: 0.4110 - accuracy: 0.8534 - val_loss:
0.4099 - val_accuracy: 0.8565
Epoch 5/5
250/250 [==============================] - 222s 890ms/step - loss: 0.3683 - accuracy: 0.8661 - val_loss:
0.4187 - val_accuracy: 0.8505
```

```
plt.figure(figsize=(10,10))
plt.subplot(2,2,1)
plt.plot(history.history['loss'], label='loss',color='r')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.legend()
plt.title('Training - Loss Function')

plt.subplot(2, 2, 2)
plt.plot(history.history['accuracy'], label='Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.legend()
plt.title('Train - Accuracy')
```

OUTPUT:



```
model_evaluation_results = model.evaluate(x_test, y_test, batch_size=32, verbose=2)
print("The test loss is", model_evaluation_results[0])
print("The test accuracy is", model_evaluation_results[1])
```

OUTPUT:

```
533/533 - 42s - loss: nan - accuracy: 0.8438 - 42s/epoch - 79ms/step
The test loss is nan
The test accuracy is 0.8438380360603333
```

```
# Prediction on test images using model.predict() method
practical_test_images =  x_test[:10]
prediction_probabilites = model.predict(practical_test_images)
prediction_probabilites[:3]
```

OUTPUT:

```
1/1 [============================= = = = = = = = = = = = =  = = = = = = = = = = = = = = = = = = = = = =
= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
= = = = =
 1.62392721e-01, 6.00572117e-03, 3.07703376e-01, 3.86583386e-03,
 8.07231665e-02, 4.55609756e-03],
 [2.13454390e-04, 1.31126042e-04, 2.42721246e-04, 1.57689094e-04,
 2.41081478e-04, 5.40605932e-03, 3.75541451e-04, 3.94529492e-01,
 3.35998135e-04, 5.983666857e-01],
 [1.77102792e-03, 9.13637981e-04, 1.02652036e-01, 3.12519167e-03,
 5.54388940e-01, 6.66437962e-04, 3.32357943e-01, 6.95253024e-04,
 2.90616718e-03, 5.23336872e-04], dtype=float32)
```

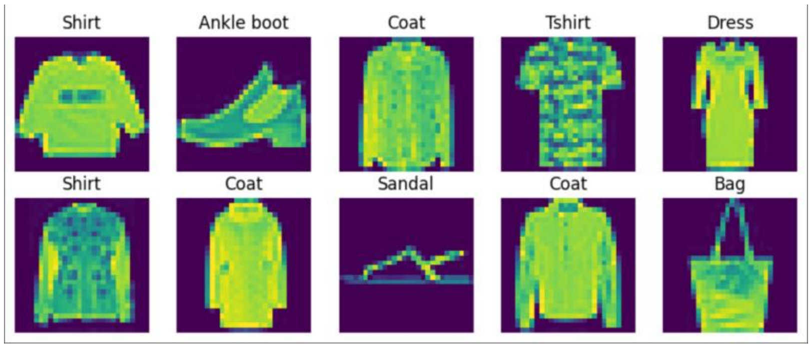
```
#  Clean up model prediction using argmax to find the index of the largest probablity
def derive_predicted_classes(prediction_probabilites):
    batch_prediction = []
    for vector in prediction_probabilites:
        batch_prediction.append(np.argmax(vector))
    return batch_prediction

model_prediction = derive_predicted_classes(prediction_probabilites)
model_prediction
```

OUTPUT:

```
[6, 9, 4, 0, 3, 6, 4, 5, 4, 8]
```

```
# Visualise the prediction result
plt.figure(figsize=(10,10))
for i in range(len(practical_test_images)):
    plt.subplot(5,5, i+1)
    plt.axis("off")
    plt.grid(False)
    plt.imshow(practical_test_images[i])
    plt.title(class_names[model_prediction[i]])
plt.show()
```

OUTPUT:



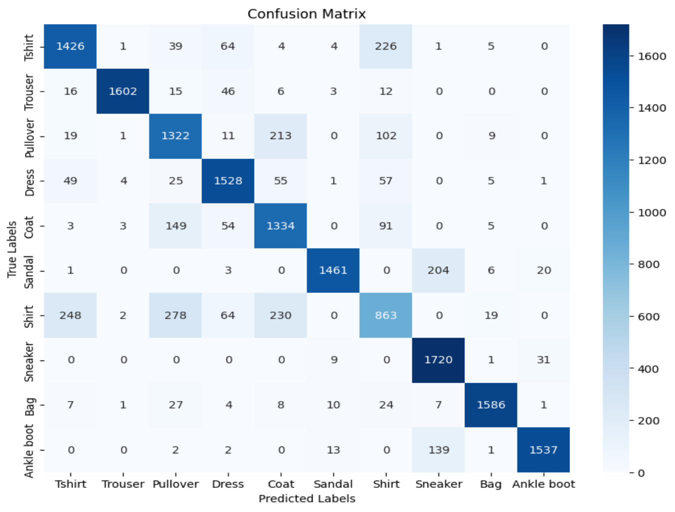
```
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

# Reshape x_test to have the correct shape
x_test = x_test.reshape(x_test.shape[0], image_rows, image_cols, 1)

# Calculate accuracy score
y_pred = model.predict(x_test)
y_pred_classes = np.argmax(y_pred, axis=1)
accuracy = accuracy_score(y_test, y_pred_classes)
print("Accuracy:", accuracy)
```

OUTPUT:

```
533/533 [==============================] - 41s 78ms/step Accuracy: 0.8438380281690141
```

```
# Compute confusion matrix
confusion_mtx = confusion_matrix(y_test, y_pred_classes)

# Display confusion matrix as a heatmap
import seaborn as sns
plt.figure(figsize=(10,8))
sns.heatmap(confusion_mtx, annot=True, fmt='d', cmap='Blues', xticklabels=class_names, yticklabels=class_names)
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.title('Confusion Matrix')
plt.show()
```

OUTPUT:



```
# Generate and display classification report
classification_rep = classification_report(y_test, y_pred_classes, target_names=class_names)
print("Classification Report:\n", classification_rep)
```

OUTPUT:

```
Classification Report:
              precision    recall  f1-score   support

      Tshirt       0.81      0.81      0.81      1770
     Trouser       0.99      0.94      0.97      1700
    Pullover       0.71      0.79      0.75      1677
       Dress       0.86      0.89      0.87      1725
        Coat       0.72      0.81      0.76      1639
      Sandal       0.97      0.86      0.91      1695
       Shirt       0.63      0.51      0.56      1704
     Sneaker       0.83      0.98      0.90      1761
         Bag       0.97      0.95      0.96      1675
  Ankle boot       0.97      0.91      0.94      1694

    accuracy                           0.84     17040
   macro avg       0.85      0.84      0.84     17040
weighted avg       0.85      0.84      0.84     17040
```

RESULT:

Thus the ALexNet architecture has been implemented successfully.

DATE:

EX.NO: 10 **IMPLEMENTATION OF VGG ARCHITECTURE.**

AIM:

To use the VGG (Visual Geometric group) architecture to CIFAR-10 datasets using Tensorflow.

DESCRIPTION:

- VGG stands for Visual Geometry Group; it is a standard deep Convolutional Neural Network (CNN) architecture with multiple layers.
- The “deep” refers to the number of layers with VGG-16 or VGG-19 consisting of 16 and 19 convolutional layers.
- The VGG architecture is the basis of ground-breaking object recognition models. Developed as a deep neural network, the VGGNet also surpasses baselines on many tasks and datasets beyond ImageNet. Moreover, it is now still one of the most popular image recognition architectures.
- The VGG network is constructed with very small convolutional filters. The VGG-16 consists of 13 convolutional layers and three fully connected layers.
- Input:The VGGNet takes in an image input size of 224×224
- Convolutional Layers: VGG’s convolutional layers leverage a minimal receptive field, i.e., 3×3, the smallest possible size that still captures up/down and left/right.
- Hidden Layers: All the hidden layers in the VGG network use ReLU. VGG does not usually leverage Local Response Normalization (LRN) as it increases memory consumption and training time.
- Fully-Connected Layers: The VGGNet has three fully connected layers. Out of the three layers, the first two have 4096 channels each, and the third has 1000 channels, 1 for each class.

PROGRAM:

```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, classification_report
import numpy as np

# Load and preprocess CIFAR-10 dataset
(train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_data()
train_images, test_images = train_images / 255.0, test_images / 255.0
```

OUTPUT:

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 [==============================] - 14s 0us/step
```

```
model = models.Sequential()
model.add(layers.Conv2D(64, (3, 3), activation='relu', input_shape=(32, 32, 3), padding='same'))
model.add(layers.BatchNormalization(scale=False))  # Add BatchNormalization layer
model.add(layers.Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu', input_shape=(32, 32, 3), padding='same'))
model.add(layers.BatchNormalization(scale=False))  # Add BatchNormalization layer
model.add(layers.Conv2D(128, (3, 3), activation='relu', padding='same'))
model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(256, (3, 3), activation='relu', input_shape=(32, 32, 3), padding='same'))
model.add(layers.BatchNormalization(scale=False))  # Add BatchNormalization layer
model.add(layers.Conv2D(256, (3, 3), activation='relu', padding='same'))
model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(512, (3, 3), activation='relu', input_shape=(32, 32, 3), padding='same'))
model.add(layers.BatchNormalization(scale=False))  # Add BatchNormalization layer
model.add(layers.Conv2D(512, (3, 3), activation='relu', padding='same'))
model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(512, (3, 3), activation='relu', input_shape=(32, 32, 3), padding='same'))
model.add(layers.BatchNormalization(scale=False))  # Add BatchNormalization layer
model.add(layers.Conv2D(512, (3, 3), activation='relu', padding='same'))
model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Flatten())
model.add(layers.Dense(4096, activation='relu'))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(4096, activation='relu'))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(10, activation='softmax'))

# creating the model

model.summary()
```

OUTPUT:

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 32, 32, 64)        1792

 batch_normalization (Batch  (None, 32, 32, 64)        192
 Normalization)

 conv2d_1 (Conv2D)           (None, 32, 32, 64)        36928

 max_pooling2d (MaxPooling2D)  (None, 16, 16, 64)        0


 conv2d_2 (Conv2D)           (None, 16, 16, 128)       73856

 batch_normalization_1 (Bat  (None, 16, 16, 128)       384
 chNormalization)

 conv2d_3 (Conv2D)           (None, 16, 16, 128)       147584
 max_pooling2d_1 (MaxPoolin  (None, 8, 8, 128)         0
 g2D)

 conv2d_4 (Conv2D)           (None, 8, 8, 256)         295168

 batch_normalization_2 (Bat  (None, 8, 8, 256)         768
 chNormalization)

 conv2d_5 (Conv2D)           (None, 8, 8, 256)         590080

 max_pooling2d_2 (MaxPoolin  (None, 4, 4, 256)         0
 g2D)

 conv2d_6 (Conv2D)           (None, 4, 4, 512)         1180160

 batch_normalization_3 (Bat  (None, 4, 4, 512)         1536
 chNormalization)

 conv2d_7 (Conv2D)           (None, 4, 4, 512)         2359808

 max_pooling2d_3 (MaxPoolin  (None, 2, 2, 512)         0
 g2D)

 conv2d_8 (Conv2D)           (None, 2, 2, 512)         2359808

 batch_normalization_4 (Bat  (None, 2, 2, 512)         1536
 chNormalization)

 conv2d_9 (Conv2D)           (None, 2, 2, 512)         2359808

 max_pooling2d_4 (MaxPoolin  (None, 1, 1, 512)         0
 g2D)

 flatten (Flatten)           (None, 512)               0

 dense (Dense)               (None, 4096)              2101248

 dropout (Dropout)           (None, 4096)              0

 dense_1 (Dense)             (None, 4096)              16781312

 dropout_1 (Dropout)         (None, 4096)              0

 dense_2 (Dense)             (None, 10)                40970

=================================================================
Total params: 28332938 (108.08 MB)
Trainable params: 28329994 (108.07 MB)
Non-trainable params: 2944 (11.50 KB)
_________________________________________________________________
```

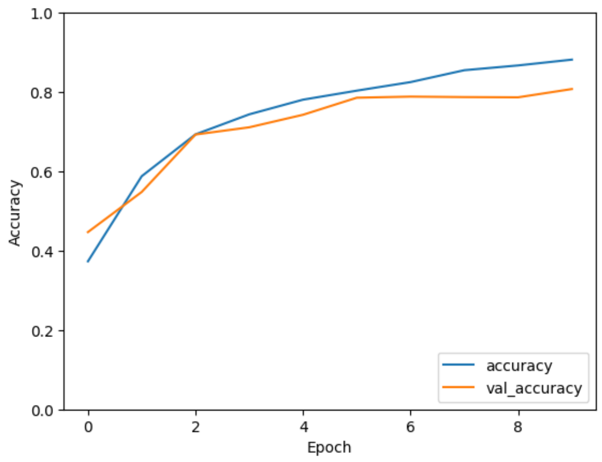
```
model.compile(optimizer='adam',
            loss='sparse_categorical_crossentropy',
            metrics=['accuracy'])

# Train the model
history = model.fit(train_images, train_labels, epochs=10,
                 validation_data=(test_images, test_labels))
```

OUTPUT:

```
Epoch 1/10
1563/1563 [==============================] - 57s 27ms/step - loss: 1.6695 - accuracy: 0.3731 - val_loss:
1.6695 - val_accuracy: 0.4469
Epoch 2/10
1563/1563 [==============================] - 39s 25ms/step - loss: 1.1939 - accuracy: 0.5878 - val_loss:
1.3521 - val_accuracy: 0.5481
Epoch 3/10
1563/1563 [==============================] - 40s 25ms/step - loss: 0.9247 - accuracy: 0.6933 - val_loss:
0.8926 - val_accuracy: 0.6929
Epoch 4/10
1563/1563 [==============================] - 40s 25ms/step - loss: 0.7902 - accuracy: 0.7438 - val_loss:
0.8881 - val_accuracy: 0.7111
Epoch 5/10
1563/1563 [==============================] - 40s 25ms/step - loss: 0.6856 - accuracy: 0.7808 - val_loss:
0.8698 - val_accuracy: 0.7427
Epoch 6/10
1563/1563 [==============================] - 40s 25ms/step - loss: 0.6300 - accuracy: 0.8035 - val_loss:
0.8138 - val_accuracy: 0.7857
Epoch 7/10
1563/1563 [==============================] - 41s 26ms/step - loss: 0.5603 - accuracy: 0.8251 - val_loss:
1.0362 - val_accuracy: 0.7885
Epoch 8/10
1563/1563 [==============================] - 45s 29ms/step - loss: 0.4660 - accuracy: 0.8551 - val_loss:
0.7054 - val_accuracy: 0.7873
Epoch 9/10
1563/1563 [==============================] - 39s 25ms/step - loss: 0.4313 - accuracy: 0.8671 - val_loss:
2.1532 - val_accuracy: 0.7867
Epoch 10/10
1563/1563 [==============================] - 39s 25ms/step - loss: 0.3916 - accuracy: 0.8817 - val_loss:
4.2583 - val_accuracy: 0.8076
```

```
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label='val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0, 1])
plt.legend(loc='lower right')
plt.show()
```

OUTPUT:



```
test_loss, test_acc = model.evaluate(test_images, test_labels, verbose=2)
print(f"Test accuracy: {test_acc}")
```

OUTPUT:

```
313/313 - 2s - loss: 4.2583 - accuracy: 0.8076 - 2s/epoch - 6ms/step Test accuracy: 0.8076000213623047
```

```
# Make predictions
predictions = np.argmax(model.predict(test_images), axis=-1)
```

OUTPUT:

```
313/313 [==============================] - 2s 6ms/step
```

```
# Make predictions
predictions = np.argmax(model.predict(test_images), axis=-1)
# Generate classification report
report = classification_report(test_labels, predictions)

# Print confusion matrix and classification report
print("Confusion Matrix:\n", cm)
print("Classification Report:\n", report)
```

OUTPUT:

```
Confusion Matrix:
 [[825   8  43  37  10   3   7  26   6  35]
 [  7 923   8   3   2   0   5   3   1  48]
 [ 46   0 655  31  76  81  56  43   0  12]
 [  4   3  29 549  35 258  62  44   2  14]
 [  4   1  16  20 795  39  25  97   1   2]
 [  1   1   9  73  19 838   7  50   0   2]
 [  2   1  28  33  19  16 886  10   2   3]
 [  4   1   4  12  14  26   1 931   0   7]
 [ 79  52  10  11   4   2  17  10 748  67]
 [  8  26   2   5   1   3   4  14  11 926]]


Classification Report:
              precision    recall  f1-score   support

           0       0.84      0.82      0.83      1000
           1       0.91      0.92      0.92      1000
           2       0.81      0.66      0.73      1000
           3       0.71      0.55      0.62      1000
           4       0.82      0.80      0.81      1000
           5       0.66      0.84      0.74      1000
           6       0.83      0.89      0.86      1000
           7       0.76      0.93      0.84      1000
           8       0.97      0.75      0.84      1000
           9       0.83      0.93      0.88      1000

    accuracy                           0.81     10000
   macro avg       0.81      0.81      0.81     10000
weighted avg       0.81      0.81      0.81     10000
```

RESULT:

Thus the VGG architecture has been implemented successfully.

DATE:

EX.NO: 11 **IMPLEMENTATION OF GOOGLENET ARCHITECTURE.**

AIM:

To implement the application using Google-net architecture.

DESCRIPTION:

GoogLeNet is a convolutional neural network that is 22 layers deep.

GoogLeNet, also known as Inception v1, is a deep convolutional neural network architecture that was introduced in 2014. It won the ImageNet Large-Scale Visual Recognition Challenge 2014 (ILSVRC-2014) with a classification performance of 92.3%, which was a significant improvement over previous state-of-the-art models.

The success of GoogLeNet was attributed to a number of factors, including:

The use of inception modules, which are a type of building block that combines convolutions of different kernel sizes in parallel. This allows the network to learn features at different scales, which is important for image classification.

The use of 1x1 convolutions to reduce the number of filter channels in the network, which makes it more efficient and reduces the risk of overfitting.

The use of global average pooling at the end of the network, which allows the network to learn features that are invariant to translation and scale.

PROGRAM:

```
import tensorflow as tf
from tensorflow.keras.layers import Input,
Conv2D,MaxPooling2D,AveragePooling2D,concatenate,Flatten,Dense,Dropout
from tensorflow.keras.models import Model
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.utils import to_categorical

(x_train,y_train),(x_test,y_test)=cifar10.load_data()
x_train=x_train/255.0
x_test=x_test/255.0
y_train=to_categorical(y_train,num_classes=10)
y_test=to_categorical(y_test,num_classes=10)

def inception_module(x,filters):
   conv1x1=Conv2D(filters[0],(1,1),padding='same',activation='relu')(x)
   conv3x3=Conv2D(filters[1],(3,3),padding='same',activation='relu')(x)
   conv5x5=Conv2D(filters[2],(5,5),padding='same',activation='relu')(x)
   maxpool=MaxPooling2D((3,3),strides=(1,1),padding='same')(x)
   pool_conv=Conv2D(filters[3],(1,1),padding='same',activation='relu')(maxpool)
   return concatenate([conv1x1,conv3x3,conv5x5,pool_conv],axis=-1)
```

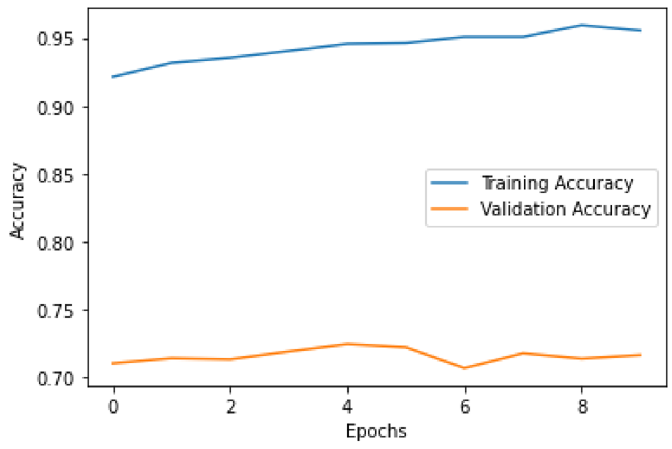
```
input_layer=Input(shape=(32,32,3))
x=Conv2D(64,(7,7),padding='same',activation='relu',strides=(2,2))(input_layer)
x=MaxPooling2D((3,3),strides=(2,2))(x)
x=Conv2D(64,(1,1),padding='same',activation='relu')(x)
x=Conv2D(192,(3,3),padding='same',activation='relu')(x)
x=MaxPooling2D((3,3),strides=(2,2))(x)
x=inception_module(x,[64,128,32,32])
x=inception_module(x,[128,192,96,64])
x=MaxPooling2D((3,3),strides=(2,2))(x)

x=Flatten()(x)
x=Dropout(0.4)(x)
x=Dense(256,activation='relu')(x)
output_layer=Dense(10,activation='softmax')(x)

model=Model(inputs=input_layer,outputs=output_layer)

model.compile(optimizer='adam',loss='categorical_crossentropy',metrics=['accuracy'])

history=model.fit(x_train,y_train,epochs=10,batch_size=128,validation_data=(x_test,y_test))
```

OUTPUT:

```
l_accuracy: 0.7068
Epoch 8/10
391/391 [==============================] - 24s 61ms/step - loss: 0.1421 - accuracy: 0.9507 - val_loss: 1.3139 -
val_accuracy: 0.7177
Epoch 9/10
391/391 [==============================] - 23s 59ms/step - loss: 0.1206 - accuracy: 0.9594 - val_loss: 1.5549 -
val_accuracy: 0.7138
Epoch 10/10
391/391 [==============================] - 23s 60ms/step - loss: 0.1277 - accuracy: 0.9554 - val_loss: 1.5585 -
val_accuracy: 0.7164
```

```
#Plot the training accuracy graph
import matplotlib.pyplot as plt
import numpy as np
plt.plot(history.history['accuracy'],label='Training Accuracy')
plt.plot(history.history['val_accuracy'],label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()

#Generate predictions
predictions=model.predict(x_test)
predicted_labels=np.argmax(predictions,axis=1)
true_labels=np.argmax(y_test,axis=1)

#Print classification report
from sklearn.metrics import classification_report
print(classification_report(true_labels,predicted_labels))
```

OUTPUT:



```
313/313 [==============================] - 1s 4ms/step


              precision    recall  f1-score   support

           0       0.76      0.79      0.78      1000
           1       0.85      0.78      0.81      1000
           2       0.60      0.61      0.61      1000
           3       0.54      0.53      0.53      1000
           4       0.63      0.74      0.68      1000
           5       0.67      0.55      0.61      1000
           6       0.77      0.77      0.77      1000
           7       0.73      0.80      0.76      1000
           8       0.85      0.81      0.83      1000
           9       0.78      0.79      0.78      1000

    accuracy                           0.72     10000
   macro avg       0.72      0.72      0.72     10000
weighted avg       0.72      0.72      0.72     10000
```

RESULT:

Thus the application using googlenet architecture has been implemented successfully.

DATE:

EX.NO: 12 **IMPLEMENTATION OF RESNET ARCHITECTURE.**

AIM:

To implement the ResNet architecture using Tensorflow.

DESCRIPTION:

- A novel architecture called **Residual Network** was launched by Microsoft Research experts in 2015 with the proposal of **ResNet.**
- The Residual Blocks idea was created by this design to address the issue of the vanishing/exploding gradient.
- The method called skip connection is applied in this network.It bypasses some levels in between to link-layer activations to subsequent layers.
- This creates a leftover block. These leftover blocks are stacked to create resnets.
- Pooling Layers: Between residual blocks, ResNet often includes pooling layers (e.g., max-pooling) to downsample feature maps and reduce spatial dimensions.
- Fully Connected Layer: Towards the end of the network, there is typically a fully connected layer or global average pooling layer to flatten the feature maps and produce class scores in the case of image classification tasks.
- Output Layer: The final layer produces the network's output, which could be class probabilities for classification tasks or regression values for regression tasks.
- Final Activation Function: The output layer is often followed by a softmax activation for classification tasks or a linear activation for regression tasks.

PROGRAM:

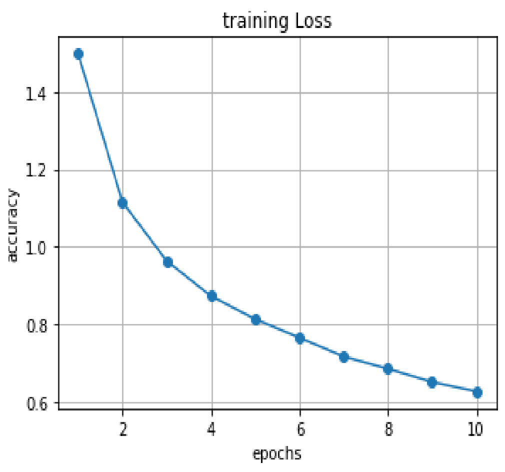
```
!pip install torch torchvision scikit-learn

import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from sklearn.metrics import classification_report,confusion_matrix
import numpy as np
import matplotlib.pyplot as plt

#define
class ResidualBlock(nn.Module):
    def __init__(self,in_channels,out_channels,stride=1):
        super(ResidualBlock,self).__init__()
        self.conv1=nn.Conv2d(in_channels,out_channels,kernel_size=3,stride=stride,padding=1,bias=False)
        self.bn1=nn.BatchNorm2d(out_channels)
        self.relu=nn.ReLU()
        self.conv2=nn.Conv2d(out_channels,out_channels,kernel_size=3,stride=1,padding=1,bias=False)
        self.bn2=nn.BatchNorm2d(out_channels)
        self.downsample=None
        if in_channels!=out_channels or stride!=1:
            self.downsample=nn.Sequential(
                nn.Conv2d(in_channels,out_channels,kernel_size=1,stride=stride,bias=False),
                nn.BatchNorm2d(out_channels)
            )

    def forward(self,x):
        residual=x

        out=self.conv1(x)
        out=self.bn1(out)
        out=self.relu(out)

        out=self.conv2(out)
        out=self.bn2(out)

        if self.downsample is not None:
            residual=self.downsample(x)


        out+=residual
        out=self.relu(out)

        return out

#define the resnet architecture
class ResNet(nn.Module):
    def __init__(self,block,num_blocks,num_classes=10):
        super(ResNet,self).__init__()
        self.in_channels=16
        self.conv1=nn.Conv2d(3,16,kernel_size=3,stride=1,padding=1,bias=False)
        self.bn=nn.BatchNorm2d(16)
        self.relu=nn.ReLU()
        self.layer1=self.make_layer(block,16,num_blocks[0],stride=1)
        self.layer2=self.make_layer(block,32,num_blocks[1],stride=2)
        self.avg_pool=nn.AdaptiveAvgPool2d((1,1))
        self.fc=nn.Linear(32,num_classes)

    def make_layer(self,block,out_channels,num_blocks,stride):
        layers=[block(self.in_channels,out_channels,stride)]
        self.in_channels=out_channels
        for _ in range(1,num_blocks):
            layers.append(block(out_channels,out_channels,stride=1))
        return nn.Sequential(*layers)

    def forward(self,x):
        out=self.conv1(x)
        out=self.bn(out)
        out=self.relu(out)
        out=self.layer1(out)
        out=self.layer2(out)
        out=self.avg_pool(out)
        out=out.view(out.size(0),-1)
        out=self.fc(out)
        return out

def ResNet18():
    return ResNet(ResidualBlock,[2,2])



def train_model(model,trainloader,criterion,optimizer,num_epochs=10):
    model.train()
    train_losses=[]
    train_accuracies=[]

    for epoch in range(num_epochs):
        running_loss=0.0
        correct=0
        total=0

        for data in trainloader:
            inputs,labels=data
            optimizer.zero_grad()
            outputs=model(inputs)
            loss=criterion(outputs,labels)
            loss.backward()
            optimizer.step()


            running_loss+=loss.item()
            _,predicted=torch.max(outputs.data,1)
            total+=labels.size(0)
            correct+=(predicted==labels).sum().item()


        train_loss=running_loss/len(trainloader)
        train_accuracy=100*correct/total
        train_losses.append(train_loss)
        train_accuracies.append(train_accuracy)


        print(f'Epoch{epoch+1}/{num_epochs},Loss:{train_loss:.4f},Accuracy:{train_accuracy:.2f}%')
    return model,train_losses,train_accuracies

transform=transforms.Compose([transforms.ToTensor(),transforms.Normalize((0.5,0.5,0.5),(0.5,0.5,0.5))])
trainset=torchvision.datasets.CIFAR10(root='./data',train=True,download=True,transform=transform)
trainloader=torch.utils.data.DataLoader(trainset,batch_size=64,shuffle=True)


resnet=ResNet18()

criterion=nn.CrossEntropyLoss()
optimizer=optim.SGD(resnet.parameters(),lr=0.01,momentum=0.9)

resnet,train_losses,train_accuarcies=train_model(resnet,trainloader,criterion,optimizer,num_epochs=10)
```

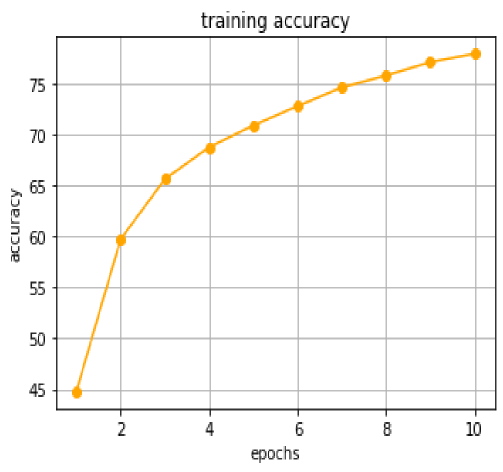
OUTPUT:

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./data/cifar-10-python.tar.gz
100%|██████████| 170498071/170498071 [00:18<00:00, 9283771.40it/s]

Extracting ./data/cifar-10-python.tar.gz to ./data
Epoch1/10,Loss:1.4989,Accuracy:44.78%
Epoch2/10,Loss:1.1158,Accuracy:59.72%
Epoch3/10,Loss:0.9633,Accuracy:65.63%
Epoch4/10,Loss:0.8727,Accuracy:68.76%
Epoch5/10,Loss:0.8130,Accuracy:70.90%
Epoch6/10,Loss:0.7650,Accuracy:72.80%
Epoch7/10,Loss:0.7162,Accuracy:74.63%
Epoch8/10,Loss:0.6845,Accuracy:75.81%
Epoch9/10,Loss:0.6504,Accuracy:77.12%
Epoch10/10,Loss:0.6258,Accuracy:77.94%
```

```
num_epochs=10
plt.figure(figsize=(10,4))
plt.subplot(1,2,1)
plt.plot(range(1,num_epochs+1),train_losses,marker='o')
plt.title('training Loss')
plt.xlabel('epochs')
plt.ylabel('accuracy')
plt.grid()


plt.subplot(1,2,2)
plt.plot(range(1,num_epochs+1),train_accuarcies,marker='o',color='orange')
plt.title('training accuracy')
plt.xlabel('epochs')
plt.ylabel('accuracy')
plt.grid()

plt.tight_layout()
plt.show()

trainset=torchvision.datasets.CIFAR10(root='./data',train=False,download=True,transform=transform)
testloader=torch.utils.data.DataLoader(trainset,batch_size=64,shuffle=False)

resnet.eval()
all_preds=[]
all_labels=[]

with torch.no_grad():
    for data in testloader:
        inputs,labels=data
        outputs=resnet(inputs)
        _,predicted=torch.max(outputs,1)
        all_preds.extend(predicted.tolist())
        all_labels.extend(labels.tolist())


print(classification_report(all_labels,all_preds))
```

OUTPUT:





```
Files already downloaded and verified

              precision    recall  f1-score   support

           0       0.79      0.72      0.75      1000
           1       0.97      0.73      0.83      1000
           2       0.77      0.41      0.53      1000
           3       0.53      0.46      0.49      1000
           4       0.77      0.62      0.69      1000
           5       0.45      0.85      0.59      1000
           6       0.69      0.86      0.77      1000
           7       0.87      0.66      0.75      1000
           8       0.86      0.85      0.85      1000
           9       0.76      0.93      0.83      1000

    accuracy                           0.71     10000
   macro avg       0.74      0.71      0.71     10000
weighted avg       0.74      0.71      0.71     10000
```

RESULT:

The ResNet architecture is implemented and the model is trained and tested successfully.

DATE:

EX.NO: 13 **AUTOREGRESSIVE(SEQUENCE TO SEQUENCE).**

AIM:

To implement the Autoregressive (Sequence to Sequence)model for a toy example using Tensorflow.

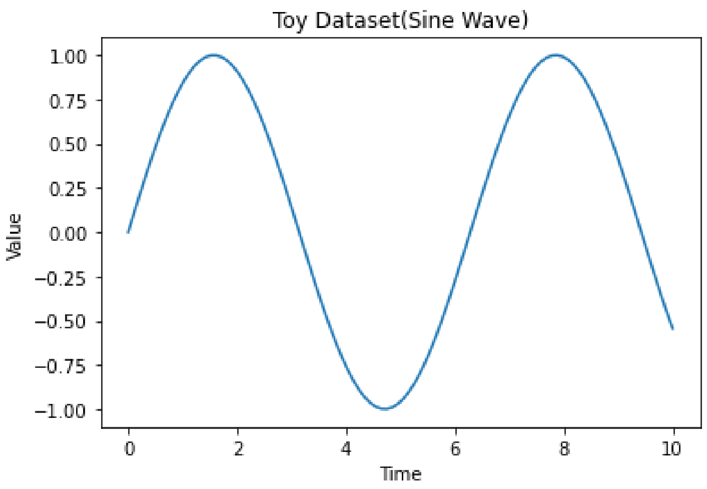
DESCRIPTION:

- An autoregressive sequence-to-sequence model is a type of neural network architecture.
- The core idea behind an autoregressive sequence-to-sequence model is to transform one sequence into another sequence.
- Seq2Seq models are typically composed of two main components: an encoder and a decoder.
- The encoder processes the input sequence and compresses it into a fixed-size context vector, also known as the hidden state or thought vector.
- The decoder takes the context vector and generates the output sequence one step at a time.
- Many modern sequence-to-sequence models incorporate attention mechanisms, which allow the model to focus on different parts of the input sequence while generating the output.
- Autoregressive sequence-to-sequence models are applied to a wide range of tasks, including machine translation, text summarization, speech synthesis, and image captioning.

PROGRAM:

```
import numpy as np
import matplotlib.pyplot as plt

num_points=100

t=np.linspace(0,10,num_points)
data=np.sin(t)

plt.plot(t,data)
plt.title("Toy Dataset(Sine Wave)")
plt.xlabel("Time")
plt.ylabel("Value")
plt.show()
```

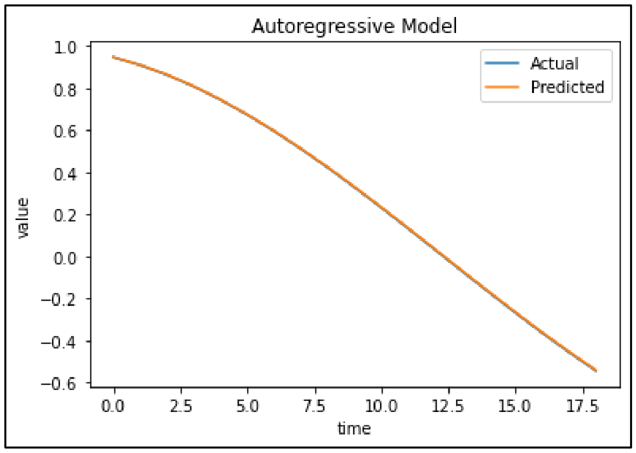
OUTPUT:



```
n=5
num_samples=len(data)

X=[]
y=[]

for i in range(n,num_samples):
    X.append(data[i-n:i])
    y.append(data[i])

X=np.array(X)
y=np.array(y)
split_ratio=0.8
split_index=int(split_ratio*len(X))
X_train,X_test=X[:split_index],X[split_index:]
y_train,y_test=y[:split_index],y[split_index:]

class AutoregressiveModel:
    def __init__(self,n):
        self.n=n
        self.weights=np.random.randn(n)
        self.bias=np.random.randn(1)
    def predict(self,input_sequence):
        return np.dot(self.weights,input_sequence) + self.bias
    def train(self,X,y,learning_rate=0.01,num_epochs=100):
        for epoch in range(num_epochs):
            for i in range(len(X)):
                input_sequence=X[i]
                target=y[i]
                prediction=self.predict(input_sequence)
                error=target-prediction


                self.weights+=learning_rate*error*input_sequence
                self.bias+=learning_rate*error

model=AutoregressiveModel(n)
model.train(X_train,y_train)

y_pred=[model.predict(input_sequence) for input_sequence in X_test]
```

```
plt.plot(y_test,label="Actual")
plt.plot(y_pred,label="Predicted")
plt.title("Autoregressive Model")
plt.xlabel("time")
plt.ylabel("value")
plt.legend()
plt.show()
```

OUTPUT:



RESULT:

Thus the implementation of Auto-regressive (Sequence to Sequence) model is created, trained and tested for a toy example is done successfully.

DATE:

EX.NO: 14 **RECURRENT NEURAL NETWORK(RNN).**

AIM:

To implement RNN (recurrent neural network) using Tensorflow.

DESCRIPTION:

- Recurrent Neural Networks (RNNs) are a class of neural network architectures designed to process sequential data by maintaining a hidden state that captures information from previous elements in the sequence.
- RNNs are specifically designed for sequential data, such as time series, natural language, speech, and more.
- RNN can handle data of varying lengths and are well-suited for tasks where the order of elements in the sequence is important.
- The core idea of an RNN is the use of recurrent connections. At each step, the RNN takes an input and its hidden state from the previous time step to produce an output and an updated hidden state.
- The hidden state serves as a memory that retains information about previous elements in the sequence, allowing the network to capture dependencies and context.
- In an RNN, the same set of weights and biases are used at each time step. This weight sharing enables the network to apply the same operation across the entire sequence.
- A limitation of traditional RNNs is the vanishing and exploding gradient problem. During training, gradients can become too small (vanishing) or too large (exploding), which hinders the learning process for long sequences.

PROGRAM:

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

t= np.linspace(0 , 10, 1000) #time step
data = np.sin(t) + 0.1 * np.random.randn(1000)

data = (data - np.mean(data)) / np.std(data)
split = int(0.8 * len(data))
train_data, test_data = data[:split], data[split:]

def create_sequences(data, seq_length):
    sequences=[]
    labels=[]

    for i in range(len(data)- seq_length):
        sequence=data[i:i+seq_length]
        label = data[i+seq_length]
        sequences.append(sequence)
        labels.append(label)

    return np.array(sequences), np.array(labels)
```

```
seq_length=10
train_sequence, train_labels = create_sequences(train_data,seq_length)

model= tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(64,input_shape=(seq_length,1)),
    tf.keras.layers.Dense(1)
])

model.compile(loss='mean_squared_error',optimizer='adam')
model.fit(train_sequence,train_labels,epochs=100,batch_size=10)
```

OUTPUT:

```
Epoch 1/100
79/79 [==============================] - 1s 1ms/step - loss: 0.0397
Epoch 2/100
79/79 [==============================] - 0s 1ms/step - loss: 0.0295
Epoch 3/100
79/79 [==============================] - 0s 1ms/step - loss: 0.0282
Epoch 4/100
79/79 [==============================] - 0s 1ms/step - loss: 0.0290
Epoch 5/100
79/79 [==============================] - 0s 1ms/step - loss: 0.0280
Epoch 6/100
79/79 [==============================] - 0s 1ms/step - loss: 0.0277
Epoch 7/100
79/79 [==============================] - 0s 1ms/step - loss: 0.0284
Epoch 8/100
79/79 [==============================] - 0s 1ms/step - loss: 0.0295
Epoch 9/100
79/79 [==============================] - 0s 1ms/step - loss: 0.0281
Epoch 10/100
79/79 [==============================] - 0s 1ms/step - loss: 0.0284
-----------------------------------------------------------------------------------------------
Epoch 90/100
79/79 [==============================] - 0s 1ms/step - loss: 0.0259
Epoch 91/100
79/79 [==============================] - 0s 1ms/step - loss: 0.0266
Epoch 92/100
79/79 [==============================] - 0s 1ms/step - loss: 0.0270
Epoch 93/100
79/79 [==============================] - 0s 1ms/step - loss: 0.0259
Epoch 94/100
79/79 [==============================] - 0s 1ms/step - loss: 0.0261
Epoch 95/100
79/79 [==============================] - 0s 1ms/step - loss: 0.0291
Epoch 96/100
79/79 [==============================] - 0s 1ms/step - loss: 0.0275
Epoch 97/100
79/79 [==============================] - 0s 989us/step - loss: 0.0267
Epoch 98/100
79/79 [==============================] - 0s 1ms/step - loss: 0.0265
Epoch 99/100
79/79 [==============================] - 0s 1ms/step - loss: 0.0263
```

```
Epoch 100/100
79/79 [==============================] - 0s 1ms/step - loss: 0.0256

<keras.callbacks.History at 0x7fbc585a8e50>
```

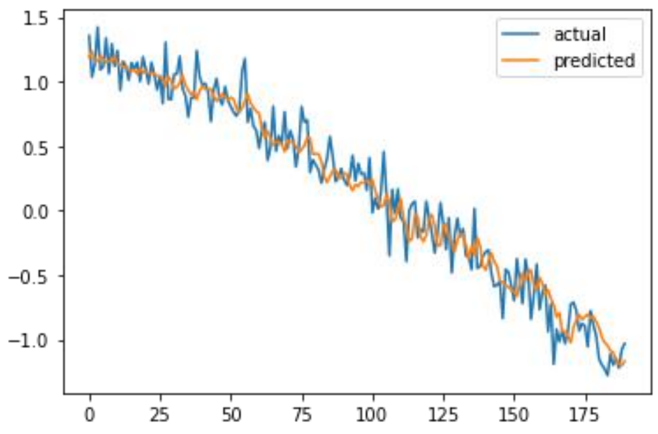
```
test_sequences,test_labels = create_sequences(test_data,seq_length)
predictions=model.predict(test_sequences)
```

OUTPUT:

```
6/6 [==============================] - 0s 903us/step
```

```
plt.plot(test_labels,label='actual')
plt.plot(predictions,label='predicted')
plt.legend()
plt.show()
```

OUTPUT:



RESULT:

Thus the implementation of Recurrent Neural Network (RNN**)** is trained and tested successfully.

DATE:

EX.NO: 15 **LONG SHORT TERM MEMORY (LSTM).**

AIM:

To implement LSTM (long short term memory) using Tensorflow.

DESCRIPTION:

- Long Short-Term Memory (LSTM) is a specialized variant of recurrent neural networks (RNNs) designed to address the vanishing gradient problem and better capture long-term dependencies in sequential data.
- LSTMs are used for processing sequential data, where the order of elements in the sequence is important, such as time series, natural language text, and speech.
- LSTMs have two primary components: the cell state and the hidden state.
- The cell state acts as a memory unit and runs along the entire sequence, allowing LSTMs to capture long-term dependencies.
- The hidden state, which is updated at each time step, serves as a working memory and can selectively pass information to the cell state.
- LSTMs utilize three types of gates to control the flow of information: the input gate, the forget gate, and the output gate.
- The input gate regulates how much new information should be added to the cell state.
- The forget gate controls which information should be discarded from the cell state.
- The output gate determines how the updated cell state should influence the hidden state.

PROGRAM:

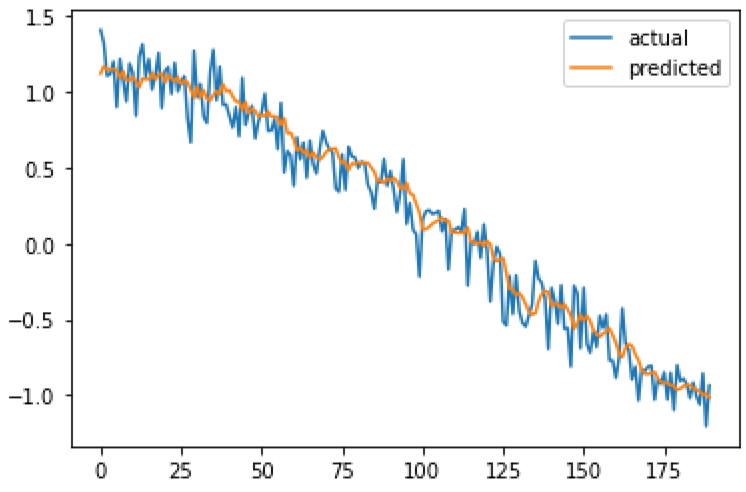
```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

t= np.linspace(0 , 10, 1000) #time step
data = np.sin(t) + 0.1 * np.random.randn(1000)

data = (data - np.mean(data)) / np.std(data)
split = int(0.8 * len(data))
train_data, test_data = data[:split], data[split:]

def create_sequences(data, seq_length):
    sequences=[]
    labels=[]

    for i in range(len(data)- seq_length):
        sequence=data[i:i+seq_length]
        label = data[i+seq_length]
        sequences.append(sequence)
        labels.append(label)
    return np.array(sequences), np.array(labels)

seq_length=10
train_sequence, train_labels = create_sequences(train_data,seq_length)

model= tf.keras.Sequential([
    tf.keras.layers.LSTM(64, return_sequences=True, input_shape=(seq_length,1)),
    tf.keras.layers.LSTM(64),
    tf.keras.layers.Dense(1),
])

model.compile(loss='mean_squared_error',optimizer='adam')
model.fit(train_sequence,train_labels,epochs=100,batch_size=10)
```

OUTPUT:

```
Epoch 1/100
79/79 [==============================] - 2s 3ms/step - loss: 0.1407
Epoch 2/100
79/79 [==============================] - 0s 3ms/step - loss: 0.0318
Epoch 3/100
79/79 [==============================] - 0s 3ms/step - loss: 0.0295
Epoch 4/100
79/79 [==============================] - 0s 3ms/step - loss: 0.0298
Epoch 5/100
79/79 [==============================] - 0s 3ms/step - loss: 0.0288
Epoch 6/100
79/79 [==============================] - 0s 3ms/step - loss: 0.0292
Epoch 7/100
79/79 [==============================] - 0s 3ms/step - loss: 0.0286
Epoch 8/100
79/79 [==============================] - 0s 3ms/step - loss: 0.0295
Epoch 9/100
79/79 [==============================] - 0s 3ms/step - loss: 0.0312
Epoch 10/100
79/79 [==============================] - 0s 3ms/step - loss: 0.0290
------------------------------------------------------------------------------------------------
Epoch 91/100
79/79 [==============================] - 0s 3ms/step - loss: 0.0269
Epoch 92/100
79/79 [==============================] - 0s 3ms/step - loss: 0.0277
Epoch 93/100
79/79 [==============================] - 0s 3ms/step - loss: 0.0276
Epoch 94/100
79/79 [==============================] - 0s 3ms/step - loss: 0.0275
Epoch 95/100
79/79 [==============================] - 0s 3ms/step - loss: 0.0276
Epoch 96/100
79/79 [==============================] - 0s 3ms/step - loss: 0.0278
Epoch 97/100
79/79 [==============================] - 0s 3ms/step - loss: 0.0273
Epoch 98/100
79/79 [==============================] - 0s 3ms/step - loss: 0.0280
Epoch 99/100
```

```
79/79 [==============================] - 0s 3ms/step - loss: 0.0283
Epoch 100/100
79/79 [==============================] - 0s 3ms/step - loss: 0.0278

<keras.callbacks.History at 0x7ff4503680a0>

test_sequences,test_labels = create_sequences(test_data,seq_length)
predictions=model.predict(test_sequences)

plt.plot(test_labels,label='actual')
plt.plot(predictions,label='predicted')
plt.legend()
plt.show()
```

OUTPUT:



RESULT:

Thus the implementation of long short term memory (LSTM) model is is trained and tested successfully.