Ex-1 Understanding the Perceptron

Aim: Implementing a python program for understanding the perceptron using the Iris plants dataset.

Algorithm:

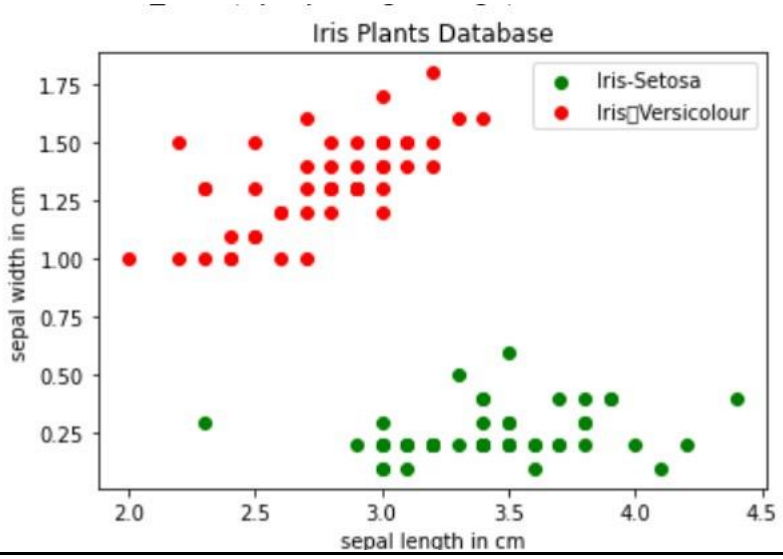
1. Import the specific libraries and also load the Iris dataset.
2. In the Iris Plants dataset we have 3 classes. We are considering 2 classes from it namely Versicolor and Setosa.
3. Next is to plot the data for two of the four variables. (Assign some colors to the data points that can be differentiated.
4. We need to split the data into training and testing so we can validate our results.
5. The next is to initialize the random weights and assign the bias value as 1.
6. Also assign or define the hyperparameters. Here hyperparameters are learning rate and epochs. Here epochs denote the iteration number over the training set.
7. Now we can start the training our perceptron with a for loop.
8. In this for loop we use simple function as If the output is greater than 0.5, we predict as 1, else 0.
9. In this we are computing MSE and we are updating the weights and bias. And we are determining the validation accuracy.
10. At last, we will plot the training loss and validation accuracy.

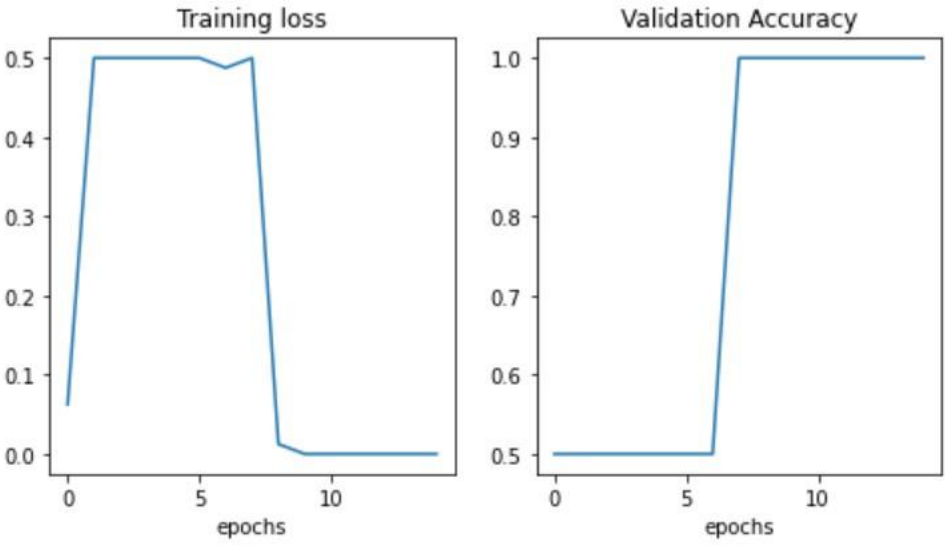
Program:

```
# Import the libraries and dataset import numpy as np

from sklearn.model_selection import

train_test_split import matplotlib.pyplot as plt

# We will be using the Iris Plants

Database from sklearn.datasets import

load_iris

SEED = 2017

# The first two classes (Iris-Setosa and Iris-Versicolour) are linear

separable iris = load_iris() idxs = np.where(iris.target<2) X =

iris.data[idxs] y = iris.target[idxs]

# Let's plot the data for two of the four variables

plt.scatter(X[y==0][:,0],X[y==0][:,2], color='green', label='Iris-Setosa')

plt.scatter(X[y==1][:,0],X[y==1][:,2], color='red', label='Iris-Versicolour')


X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=SEED)
```

```
# Next, we initialize the weights and the bias for the
perceptron weights =
np.random.normal(size=X_train.shape[1]) bias = 1
# Before training, we need to define the
hyperparameters learning_rate = 0.1 n_epochs
= 15 np.zeros(weights.shape)
# Now, we can start training our perceptron with a
for loop del_w = np.zeros(weights.shape)
hist_loss = [] hist_accuracy = [] for i in
range(n_epochs):
   # We apply a simple  function, if the output is > 0.5 we predict 1,
else 0     output = np.where((X_train.dot(weights)+bias)>0.5, 1, 0)
print(output)
    # Compute MSE     error = np.mean((y_train-
output)**2)     print("Error: ", error)     # Update weights
and bias     weights-= learning_rate * np.dot((output-
y_train), X_train)     bias += learning_rate *
np.sum(np.dot((output-y_train), X_train))
print("Weights:", weights)     print("bias:", bias)     #
Calculate MSE     loss = np.mean((output - y_train) ** 2)
hist_loss.append(loss)     output_val =
np.where(X_val.dot(weights)>0.5, 1, 0)     accuracy =
np.mean(np.where(y_val==output_val, 1, 0))
hist_accuracy.append(accuracy)
# We've saved the training loss and validation accuracy so that we can
plot them fig = plt.figure(figsize=(8, 4)) a = fig.add_subplot(1,2,1)
imgplot = plt.plot(hist_loss) plt.xlabel('epochs')
a.set_title('Training
loss')
a=fig.add_subplot(1,
2,2) imgplot =
plt.plot(hist_accuracy
) plt.xlabel('epochs')
a.set_title('Validation Accuracy')
plt.show()
```

Output:





Result:

Ex-2 Understanding the Perceptron using Diabetes Dataset

Aim: Implementing a python program for understanding the perceptron using the Diabetes dataset.

Algorithm:

1. Import the specific libraries and also load the diabetics dataset.
2. In the Iris Plants dataset we have 3 classes. We are considering 2 classes from it namely yes or no class.
3. Next is to plot the data for two of the four variables. (Assign some colors to the data points that can be differentiated.
4. We need to split the data into training and testing so we can validate our results.
5. The next is to initialize the random weights and assign the bias value as 1.
6. Also assign or define the hyperparameters. Here hyperparameters are learning rate and epochs. Here epochs denote the iteration number over the training set.
7. Now we can start the training our perceptron with a for loop.
8. In this for loop we use simple function as If the output is greater than 0.5, we predict as 1, else 0.
9. In this we are computing MSE and we are updating the weights and bias. And we are determining the validation accuracy.
10. At last, we will plot the training loss and validation accuracy.

Program:

```
import numpy as np

from sklearn import datasets

from sklearn.model_selection import train_test_split

from sklearn.linear_model import Perceptron

from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score


#  1: Load the diabetes dataset

diabetes = datasets.load_diabetes()

X, y = diabetes.data, diabetes.target


#  2: Preprocess the data

X = X[:, np.newaxis, 2]

y = (y > 140).astype(int)


#  3: Split the dataset into training and testing sets

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

```
#  4: Create and train the perceptron
perceptron = Perceptron(max_iter=1000, random_state=42)
perceptron.fit(X_train, y_train)

#  5: Evaluate the perceptron
y_pred = perceptron.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)

print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("F1 Score:", f1)
```

OUTPUT:

Accuracy: 0.6629213483146067
Precision: 0.6470588235294118
Recall: 0.55
F1 Score: 0.5945945945945946

Result:

Ex-3 Building a single-layer neural network

Aim: Implementation of a single-layer neural network using a python program.

Algorithm:

1. Import the necessary libraries, such as NumPy and Scikit-Learn.
2. Prepare the dataset for training and testing. This can include loading the data, splitting it into training and testing sets, and preprocessing the data as needed.
3. Define the architecture of the neural network. This includes the number of input and output neurons and the activation function to be used.
4. Initialize the weights and biases of the network randomly.
5. Define the forward propagation . This includes calculating the dot product of the input data and the weights, adding the biases, and passing the result through the activation function.
6. Define the backward propagation . This includes calculating the error, adjusting the weights and biases, and repeating this process for a number of epochs.
7. Use the trained network to make predictions on new data.
8. Finally, evaluate the performance of the network using metrics such as accuracy or mean squared error

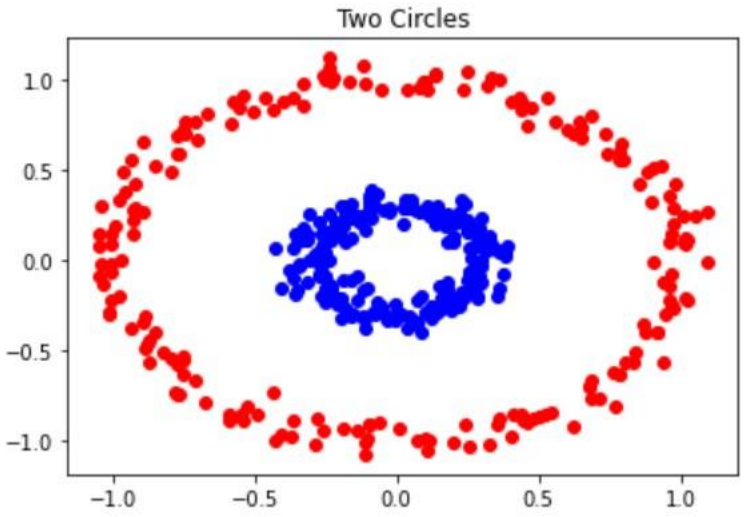
Program:

```
# Import libraries and dataset import

numpy as np  from

sklearn.model_selection import

train_test_split import matplotlib.pyplot as

plt

# We will be using make_circles from scikit-

learn from sklearn.datasets import

make_circles

SEED = 2017

X, y = make_circles(n_samples=400, factor=.3, noise=.05,

random_state=2017) outer = y == 0 inner = y plt.title("Two Circles")

plt.plot(X[outer, 0], X[outer, 1], "ro") plt.plot(X[inner, 0], X[inner, 1],

"bo") plt.show() == 1

X = X+1

X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,

random_state=SEED) def sigmoid(x):     return 1 / (1 + np.exp(-x)) n_hidden = 50 #

number of hidden units n_epochs = 1000 learning_rate = 1 # Initialise weights

weights_hidden = np.random.normal(0.0, size=(X_train.shape[1], n_hidden))

weights_output = np.random.normal(0.0, size=(n_hidden))
```

```
hist_loss = []
hist_accuracy
= []
print(weights_
hidden)
print(weights_
output)
# Run the single-layer neural network and output the statistics

for e in range(n_epochs):
del_w_hidden =
np.zeros(weights_hidden.shape)
del_w_output =
np.zeros(weights_output.shape)


   # Loop through training data in batches of 1     for
x_, y_ in zip(X_train, y_train):        # Forward
computations        hidden_input = np.dot(x_,
weights_hidden)        hidden_output =
sigmoid(hidden_input)        output =
sigmoid(np.dot(hidden_output, weights_output))


      # Backward computations        error = y_ - output        output_error = error * output *
(1 - output)        hidden_error = np.dot(output_error, weights_output) * hidden_output * (1
- hidden_output)        del_w_output += output_error * hidden_output        del_w_hidden
+= hidden_error * x_[:, None]


   # Update weights     weights_hidden += learning_rate *
del_w_hidden / X_train.shape[0]     weights_output +=
learning_rate * del_w_output / X_train.shape[0]


   # Print stats (validation loss and
accuracy)     if e % 100 == 0:
```

```
        hidden_output = sigmoid(np.dot(X_val,
weights_hidden))        out =
sigmoid(np.dot(hidden_output, weights_output))
loss = np.mean((out - y_val) ** 2)
        # Final prediction is based on a
threshold of 0.5        predictions = out >
0.5        accuracy = np.mean(predictions
== y_val)        print("Epoch: ",
'{:>4}'.format(e),
              "; Validation loss: ", '{:>6}'.format(loss.round(4)),
              "; Validation accuracy: ", '{:>6}'.format(accuracy.round(4)))
```

Output:



```
[[ 1.41581372  0.30855533 -0.8286774   0.74709373 -0.75963231  0.91704885
  -0.48495246 -0.57719441 -1.20896818 -1.70586972  0.0808446  -0.77745088
  -0.19259674  0.58443765 -0.22142047 -2.0896038  -1.53054408  1.10733381
  -0.42364537  0.49150686  0.25480185  0.75049925 -1.33832894  0.09060373
  -1.25736457 -0.38720421 -0.89152878  0.68514566  1.42506614  0.91491817
  -0.53204396 -1.46323081  1.28273973  1.70652197 -0.6210264  -0.20222603
   1.9963953   0.98509854 -0.90295226 -0.84067831 -0.78573054  0.96945578
   0.56616831 -0.38477712 -0.38741088 -0.49245588 -0.54497559 -0.76882439
  -0.8366077   0.75120829]
 [ 0.4278623   0.63510106 -1.43815054 -2.38940829  2.31206136 -1.52797071
  -0.7157763  -1.41321883  1.15097609  0.05966685 -1.39672623 -0.94216272
   0.83699277 -1.20327686  1.78814056 -1.01663469  0.67465652 -1.30155999
   0.36316766  2.7735055  -0.1750078  -0.05698066 -0.62223596 -0.20296104
   0.65097765 -0.03755962  1.04325509 -1.31261736 -1.05026247 -0.89180348
   0.41868647  0.46701954 -0.32783136 -0.87465801 -0.57567589 -0.19351514
  -0.98123078  1.56535402 -1.38543232  0.05637089  0.37151442 -0.09902364
  -0.6690398   0.73661872  0.8353953  -0.15306034  0.93446976  0.20623259
  -1.04797761 -0.38989319]]
[-1.46723602  2.07427738  0.14974924 -0.62524334 -0.22265344 -0.99273752
  0.36758329 -0.4332854   1.23580824  1.14726009 -0.99254906 -0.14468125
 -0.92648489  0.56625814  1.37419703  0.07928502 -0.66485609  0.74515938
  2.24707644 -0.84992104 -0.92020449  0.7256309  -0.00498305 -0.47895869
  0.25113205 -0.01462018 -0.53323325 -0.76871911 -0.7214322   0.25179481
  1.20416429 -1.2483621   2.0772874  -0.7569394  -0.16157457 -1.2922913
  0.5915649  -0.36964394  1.57888113  0.34552302 -0.60195869 -0.65268296
  0.38297428 -0.52851238  1.81118583 -0.26517826 -2.32389165  1.16327978
 -0.94309564 -0.10307125]
Epoch:   999 ; Validation loss:  0.2433 ; Validation accuracy:  0.5875
```

Result:

Ex-4 Building a Multi-Layer Neural Network

Aim: Implementation of a multi-layer neural network using a python program.

Algorithm:

1. Import the necessary libraries, such as NumPy and Scikit-Learn.
2. Prepare the dataset for training and testing. This can include loading the data, splitting it into training and testing sets, and preprocessing the data as needed.
3. Define the architecture of the neural network. This includes the number of input and output neurons, the number of hidden layers, and the number of neurons in each hidden layer.
4. Initialize the weights and biases of the network randomly.
5. Define the forward propagation . This includes calculating the dot product of the input data and the weights, adding the biases, and passing the result through the activation function. Then repeat the process for the next layers.
6. Define the backward propagation . This includes calculating the error, adjusting the weights and biases, and repeating this process for a number of epochs.
7. Use the trained network to make predictions on new data.
8. Finally, evaluate the performance of the network using metrics such as accuracy or mean squared error.

Program:

```
import numpy as np  import pandas as pd
from sklearn.model_selection import
train_test_split from keras.models import
Sequential from keras.layers import Dense
from keras.callbacks import EarlyStopping,
ModelCheckpoint from tensorflow.keras.optimizers
import Adam from sklearn.preprocessing import
StandardScaler
SEED = 2017
# Load the wine data set data = pd.read_csv('C:\\Users\\ifsrk\\Documents\\01 Deep
Learning\\winequality-red.csv', sep=';') y = data['quality']
X = data.drop(['quality'], axis=1)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=SEED)
# Print average quality and first rows of training set
print('Average quality training set: {:.4f}'.format(y_train.mean()))
X_train.head()
scaler = StandardScaler().fit(X_train)
X_train = pd.DataFrame(scaler.transform(X_train)) X_test =
pd.DataFrame(scaler.transform(X_test)) print('MSE:',
np.mean((y_test - ([y_train.mean()] * y_test.shape[0])) ** 2))
print('MSE:', np.mean((y_test - ([y_train.mean()] *
y_test.shape[0])) ** 2))


# Now, let's build our neural network by defining the network architecture
model = Sequential()
# First hidden layer with 100 hidden units
model.add(Dense(200, input_dim=X_train.shape[1], activation='relu'))
# Second hidden layer with 50 hidden units
model.add(Dense(25, activation='relu'))
# Output layer
model.add(Dense(1,
activation='linear'))
# Set optimizer opt = Adam() # Compile model
```

```
model.compile(loss='mse', optimizer=opt,
metrics=['accuracy']) # Define the callback for early
stopping and saving the best model callbacks = [
        EarlyStopping(monitor='val_accuracy', patience=30, verbose=2),
        ModelCheckpoint('checkpoints/multi_layer_best_model.h5', monitor='val_accuracy',
save_best_only=True, verbose=0) ]
batch_size = 64 n_epochs = 5000 model.fit(X_train.values, y_train, batch_size=64,
epochs=n_epochs, validation_split=0.2,          verbose=2,
validation_data=(X_test.values, y_test),        callbacks=callbacks)
 model.summary()
# We can now print the performance on the test set after loading the optimal weights:
best_model = model
best_model.load_weights('checkpoints/multi_layer_best_model.h5')
best_model.compile(loss='mse', optimizer='adam', metrics=['accuracy'])
# Evaluate on test set
score = best_model.evaluate(X_test.values, y_test, verbose=0)
print('Test accuracy: %.2f%%' % (score[1]*100))


# Test accuracy: 65.62%
# Benchmark accuracy on dataset 62.4%
```

Output:

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 200)               2400

 dense_1 (Dense)             (None, 25)                5025

 dense_2 (Dense)             (None, 1)                 26

=================================================================
Total params: 7,451
Trainable params: 7,451
Non-trainable params: 0
_________________________________________________________________
Test accuracy: 0.00%
```

Result:

Ex-5 Experiment with Activation Functions

Aim: Implement a python program for getting started with the activation function.

Algorithm:

1. Import the necessary libraries for your algorithms, such as NumPy and Matplotlib for data manipulation and visualization.
2. Initialize your dataset and choose an activation function to use, such as sigmoid, ReLU, or tanh.
3. Define the function for the chosen activation function using NumPy.
4. Apply the activation function to your dataset using the function you just defined.
5. Visualize the activated data using Matplotlib to observe the effect of the activation function on the dataset.
6. Repeat s 2-5 for any other activation functions you wish to test on your dataset.
7. Depending on the context, you may want to use the activation function as part of a neural network. For that you can use pre-built library such as TensorFlow or PyTorch.

Program:

```
# Import the
libraries as follows
import numpy as np
import pandas as pd
from sklearn.model_selection import
train_test_split import matplotlib.pyplot as
plt from keras.models import Sequential
from keras.layers import Dense from
tensorflow.keras.utils import
to_categorical from keras.callbacks import
Callback from keras.datasets import mnist
SEED = 2022
# Load the MNIST dataset
# Need Internet Connection to download dataset
(X_train, y_train), (X_val, y_val) = mnist.load_data()
# Show an example of each label and print the count per label
# Plot first image of each label unique_labels =
set(y_train) plt.figure(figsize=(12, 12)) i = 1 for label in
unique_labels:     image =
X_train[y_train.tolist().index(label)]     plt.subplot(10, 10,
i)     plt.axis('off')     plt.title("{0}: ({1})".format(label,
y_train.tolist().count(label)))     i += 1
    _ = plt.imshow(image,
cmap='gray') plt.show()
print(X_val) print(y_val)
# Preprocess the data
# Normalize data
X_train = X_train.astype('float32')/255.
X_val = X_val.astype('float32')/255.
X_val
# One-Hot-Encode
labels n_classes = 10
```

```
y_train = to_categorical(y_train,
n_classes) y_val =
to_categorical(y_val, n_classes)
print(y_train)
# Flatten data - we treat the image as a sequential array of values
X_train = np.reshape(X_train, (60000, 784))
X_val = np.reshape(X_val, (10000, 784))
X_train
# Define the model with the sigmoid activation function
model_sigmoid = Sequential()
model_sigmoid.add(Dense(700, input_dim=784, activation='sigmoid'))
model_sigmoid.add(Dense(700, activation='sigmoid'))
model_sigmoid.add(Dense(700, activation='sigmoid'))
model_sigmoid.add(Dense(700, activation='sigmoid'))
model_sigmoid.add(Dense(700, activation='sigmoid'))
model_sigmoid.add(Dense(350, activation='sigmoid'))
model_sigmoid.add(Dense(100, activation='sigmoid'))
model_sigmoid.add(Dense(10, activation='softmax'))
# Compile model with SGD
model_sigmoid.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['accuracy'])
# Define the model with the ReLU activation function
model_relu = Sequential()
model_relu.add(Dense(700, input_dim=784,
activation='relu')) model_relu.add(Dense(700,
activation='relu')) model_relu.add(Dense(700,
activation='relu')) model_relu.add(Dense(700,
activation='relu')) model_relu.add(Dense(700,
activation='relu'))  model_relu.add(Dense(350,
activation='relu'))  model_relu.add(Dense(100,
activation='relu'))  model_relu.add(Dense(10,
activation='softmax'))
# Compile model with SGD
model_relu.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['accuracy'])
```

```
# Create a callback function to store the loss values
per batch class history_loss(Callback):     def
on_train_begin(self, logs={}):
       self.losses = []     def
on_batch_end(self, batch,
logs={}):
       batch_loss =
logs.get('loss')
self.losses.append(batch
_loss) n_epochs = 10
batch_size = 256
validation_split = 0.2
history_sigmoid =
history_loss()
model_sigmoid.fit(X_train, y_train, epochs=n_epochs,
batch_size=batch_size,  callbacks=[history_sigmoid],
validation_split=validation_split, verbose=2) history_relu =
history_loss()
model_relu.fit(X_train, y_train, epochs=n_epochs, batch_size=batch_size,
callbacks=[history_relu],  validation_split=validation_split, verbose=2)
np.arange(len(history_sigmoid.losses)) print(history_sigmoid.losses)
plt.plot(np.arange(len(history_sigmoid.losses)), history_sigmoid.losses,
label='sigmoid') plt.plot(np.arange(len(history_relu.losses)), history_relu.losses,
label='relu') plt.title('Losses for sigmoid and ReLU model') plt.xlabel('number of
batches') plt.ylabel('loss') plt.legend(loc=1)
plt.show()
# Losses for sigmoid and ReLU model
# Extract the maximum weights of each model per layer
w_sigmoid = [] w_relu = [] for i in
range(len(model_sigmoid.layers)):
w_sigmoid.append(max(model_sigmoid.layers[i].get_weights(
)[1]))
w_relu.append(max(model_relu.layers[i].get_weights()[1]))
print(w_sigmoid) print(w_relu)
```

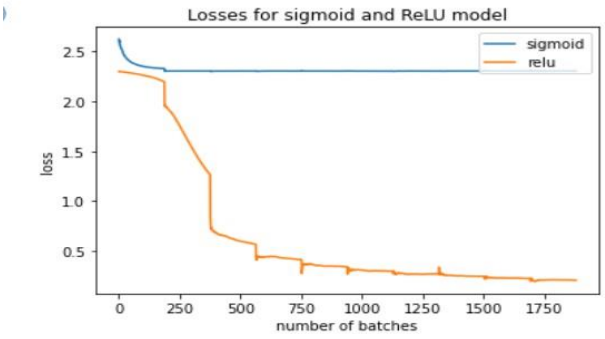
```
print(len(model_sigmoid.layers)) #
Plot the weights of both models fig,
ax = plt.subplots() index =
np.arange(len(model_sigmoid.layers
)) bar_width = 0.35
plt.bar(index, w_sigmoid, bar_width, label='sigmoid', color='b', alpha=0.4)
plt.bar(index + bar_width, w_relu, bar_width, label='relu', color='r', alpha=0.4)
plt.title('Maximum weights across layers for sigmoid and ReLU activation
functions') plt.xlabel('layer number') plt.ylabel('maximum weight')
plt.legend(loc=0)
plt.xticks(index + bar_width / 2, np.arange(8))
plt.show()
```

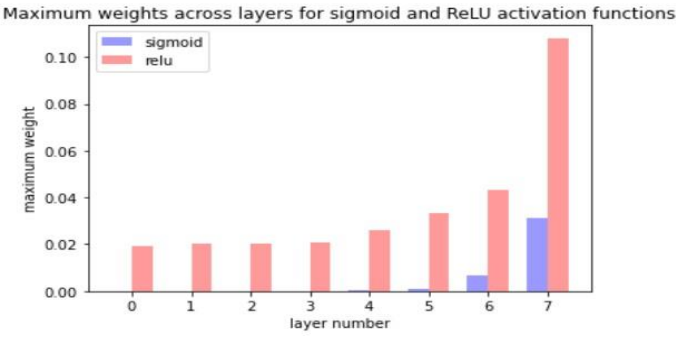
Output:

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


```
[[[0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]
  ...
  [0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]]
```

```
# Losses for sigmoid and ReLU model
```





6. Experiment with Vehicle Type Recognition

Aim: To implement Vehicle type recognition in python language.

Algorithm:

1. Gather a dataset of vehicle images, labeled with their corresponding vehicle types (e.g., car, truck, motorcycle).Split the dataset into training and testing sets.
2. Import required libraries, including TensorFlow or PyTorch for deep learning and OpenCV for image processing.
3. Resize all images to a common size (e.g., 224x224 pixels) to ensure consistent input dimensions for the CNN.
    a. Normalize pixel values to a common range (e.g., [0, 1] or [0, 255]).
    b. Optionally, apply data augmentation techniques (e.g., random rotation, flipping) to increase model robustness.
4. Create a CNN model consisting of convolutional layers (Conv2D), pooling layers (MaxPooling2D), and fully connected layers (Dense).
    a. Adjust the number of layers and filters based on the complexity of your task.
    b. Use activation functions like ReLU and appropriate kernel sizes.
    c. Add a softmax output layer with as many neurons as there are vehicle classes, and use categorical cross-entropy as the loss function.
5. Compile the CNN model by specifying the optimizer (e.g., Adam), loss function (categorical cross-entropy), and evaluation metric (accuracy).
6. Train the model using the training dataset.
    a. Specify the number of epochs and batch size.
    b. Monitor training progress and loss convergence.
7. Assess the model's performance using the test dataset.Calculate accuracy and other relevant metrics to evaluate its effectiveness.
8. Use the trained CNN model to make predictions on new vehicle images.
9. Visualize predictions, confusion matrices, and model performance metrics for better understanding.
10. Experiment with different architectures, hyperparameters, and data augmentation techniques to improve model performance.
11. If needed, deploy the trained model in a real-world application for vehicle type recognition.

Program:

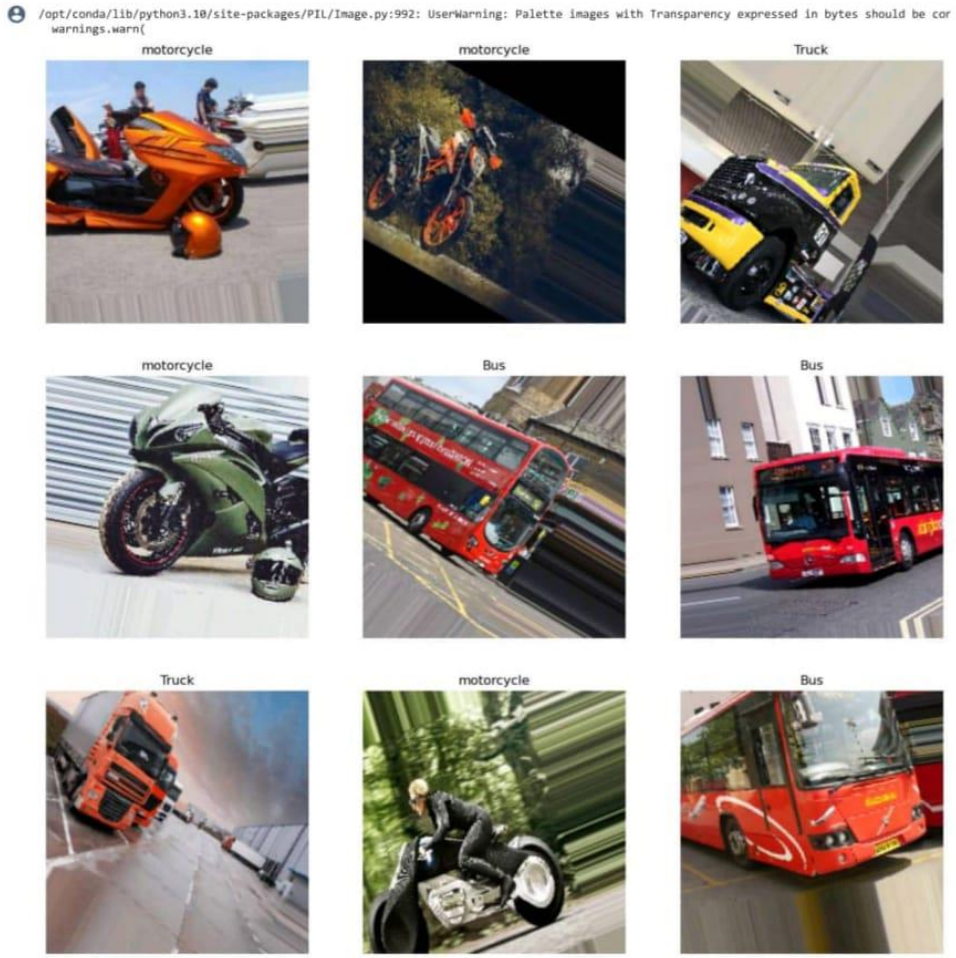
```
import os
import numpy as np
from keras.preprocessing.image import ImageDataGenerator
from keras.applications import VGG16
from keras import models, layers, optimizers

# Set the path to the dataset folder
dataset_path = '/kaggle/input/vehicle-type-recognition/Dataset'

# Set the batch size and image size
batch_size = 32
image_size = (224, 224)

# Create an instance of the ImageDataGenerator class for data augmentation
train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest')

# Create a generator for the training data
```

```
train_generator = train_datagen.flow_from_directory(
    dataset_path,
    target_size=image_size,
    batch_size=batch_size,
    class_mode='categorical')

# Load the VGG16 model with pre-trained weights
base_model = VGG16(weights='imagenet', include_top=False, input_shape=(image_size[0], image_size[1], 3))

# Freeze the layers of the base model
for layer in base_model.layers:
    layer.trainable = False

# Create a new model by adding custom layers on top of the base model
model = models.Sequential()
model.add(base_model)
model.add(layers.Flatten())
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(4, activation='softmax'))

# Compile the model
model.compile(loss='categorical_crossentropy', optimizer=optimizers.RMSprop(lr=1e-4), metrics=['acc'])

import matplotlib.pyplot as plt

# Get the class labels
class_labels = train_generator.class_indices
class_labels = dict((v, k) for k, v in class_labels.items())

# Display the images and their labels
fig, ax = plt.subplots(3, 3, figsize=(15, 15))
for i in range(9):
    x, y = train_generator.next()
    image = x[0]
    label = y[0]
```

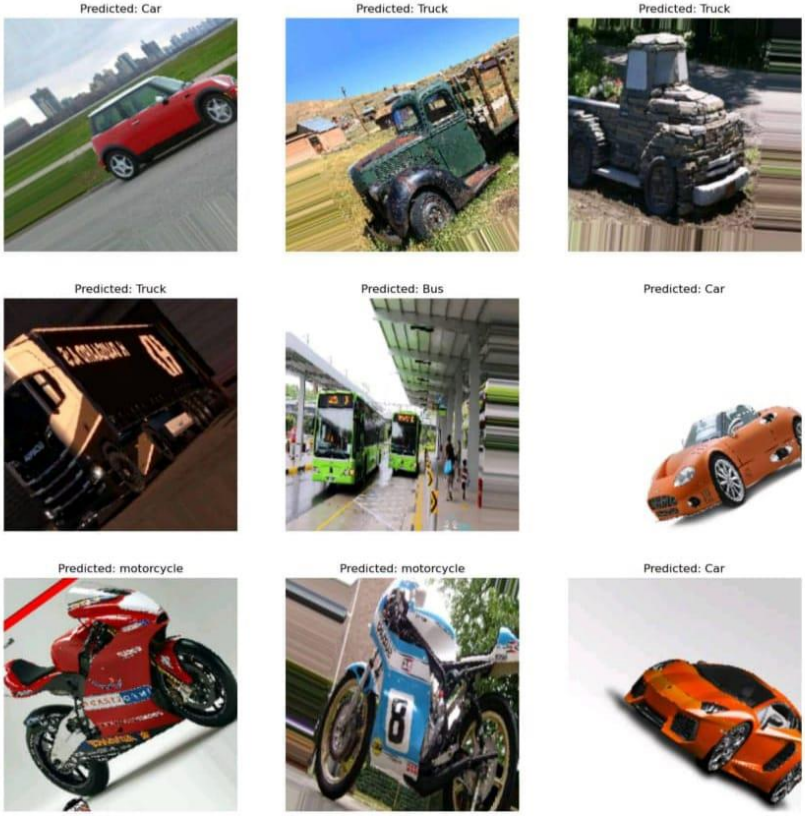
```
    label = np.argmax(label)
    label = class_labels[label]
    ax[i//3, i%3].imshow(image)
    ax[i//3, i%3].set_title(label)
    ax[i//3, i%3].axis('off')
plt.show()

# Train the model
history = model.fit_generator(train_generator, s_per_epoch=train_generator.samples // batch_size, epochs=30)
# Display the data

import matplotlib.pyplot as plt

acc = history.history['acc']
loss = history.history['loss']

epochs_range = range(len(acc))

plt.figure(figsize=(15, 15))
plt.subplot(2, 1, 1)
plt.plot(epochs_range, acc, label='Training Accuracy')
plt.legend(loc='lower right')
plt.title('Training Accuracy')

plt.subplot(2, 1, 2)
plt.plot(epochs_range, loss, label='Training Loss')
plt.legend(loc='upper right')
plt.title('Training Loss')
plt.show()
import matplotlib.pyplot as plt

# Get the class labels
class_labels = train_generator.class_indices
class_labels = dict((v, k) for k, v in class_labels.items())
```

```
# Display the images and their predicted labels
fig, ax = plt.subplots(3, 3, figsize=(15, 15))
for i in range(9):
    x, y = train_generator.next()
    image = x[0]
    label = y[0]
    # Get the predicted label
    pred = model.predict(np.expand_dims(image, axis=0))
    pred_label = np.argmax(pred)
    pred_label = class_labels[pred_label]
    ax[i//3, i%3].imshow(image)
    ax[i//3, i%3].set_title(f'Predicted: {pred_label}')
    ax[i//3, i%3].axis('off')
plt.show()
```

OUTPUT:



```
1/1 [==============================] - 0s 27ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 21ms/step
```



Result:

7. Diabetic Retinopathy

Aim: To implement Diabetic Retinopathy in python language.

Algorithm:

1. Gather a dataset of retinal images, ideally labeled with diabetic retinopathy severity levels.Split the dataset into training and testing sets.
2. Import the necessary libraries, including deep learning frameworks like TensorFlow or PyTorch, and image processing libraries like OpenCV.
3. Preprocess the retinal images to enhance their quality and prepare them for analysis.Resize images to a common size (e.g., 224x224 pixels) for consistent input dimensions.
    a. Normalize pixel values to a common range (e.g., [0, 1] or [0, 255]).
    b. Augment the training data with techniques like rotation, flipping, and brightness adjustments to increase model robustness (optional).
4. Create a CNN model tailored for image classification.
    a. Use convolutional layers (Conv2D), pooling layers (MaxPooling2D), and fully connected layers (Dense).
    b. Adjust the number of layers and filters based on the complexity of the task.Employ activation functions like ReLU and kernel sizes suitable for image analysis.
    c. Add a softmax output layer with as many neurons as there are diabetic retinopathy severity levels (e.g., 0 to 4), and use categorical cross-entropy as the loss function.
5. Compile the CNN model by specifying the optimizer (e.g., Adam), loss function (categorical cross-entropy), and evaluation metric (e.g., accuracy).
6. Train the model using the training dataset.Specify the number of epochs and batch size.Monitor training progress, including loss convergence and validation performance.
7. Assess the model's performance using the test dataset.
    a. Calculate evaluation metrics such as accuracy, precision, recall, and F1-score.
    b. Examine the confusion matrix to understand the model's strengths and weaknesses.
8. Utilize the trained CNN model to make predictions on new retinal images. Interpret the predictions to determine the severity level of diabetic retinopathy.
9. Visualize model predictions, confusion matrices, and performance metrics for better insights and communication.

Program:

```
import numpy as np # linear algebra

import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

import os

import random

import sys

import cv2

import matplotlib

from subprocess import check_output


from keras.models import Sequential

from keras.layers import Dense, Conv2D, MaxPooling2D, Dropout, Flatten

from keras.preprocessing.image import ImageDataGenerator, array_to_img, img_to_array,
load_img

from keras.optimizers import Adam

from sklearn.model_selection import train_test_split

from keras.utils import to_categorical

def classes_to_int(label):

    # label = classes.index(dir)

    label = label.strip()

    if label == "No DR":  return 0

    if label == "Mild":  return 1

    if label == "Moderate":  return 2

    if label == "Severe":  return 3

    if label == "Proliferative DR":  return 4

    print("Invalid Label", label)

    return 5


def int_to_classes(i):

    if i == 0: return "No DR"

    elif i == 1: return "Mild"
```

```
    elif i == 2: return "Moderate"
    elif i == 3: return "Severe"
    elif i == 4: return "Proliferative DR"
    print("Invalid class ", i)
    return "Invalid Class"
NUM_CLASSES = 5
# we need images of same size so we convert them into the size
WIDTH = 128
HEIGHT = 128
DEPTH = 3
inputShape = (HEIGHT, WIDTH, DEPTH)
# initialize number of epochs to train for, initial learning rate and batch size
EPOCHS = 15
INIT_LR = 1e-3
BS = 32
#global variables
ImageNameDataHash = {}
uniquePatientIDList = []
def readTrainData(trainDir):
    global ImageNameDataHash
    # loop over the input images
    images = os.listdir(trainDir)
    print("Number of files in " + trainDir + " is " + str(len(images)))
    for imageFileName in images:
        if (imageFileName == "trainLabels.csv"):
            continue
        # load the image, pre-process it, and store it in the data list
        imageFullPath = os.path.join(os.path.sep, trainDir, imageFileName)
        #print(imageFullPath)
        img = load_img(imageFullPath)
```

```
        arr = img_to_array(img)  # Numpy array with shape (233,233,3)
        dim1 = arr.shape[0]
        dim2 = arr.shape[1]
        dim3 = arr.shape[2]
        if (dim1 < HEIGHT or dim2 < WIDTH or dim3 < DEPTH):
            print("Error image dimensions are less than expected "+str(arr.shape))
        arr = cv2.resize(arr, (HEIGHT,WIDTH)) #Numpy array with shape (HEIGHT, WIDTH,3)
        #print(arr.shape) # 128,128,3
        dim1 = arr.shape[0]
        dim2 = arr.shape[1]
        dim3 = arr.shape[2]
        if (dim1 != HEIGHT or dim2 != WIDTH or dim3 != DEPTH):
            print("Error after resize, image dimensions are not equal to expected "+str(arr.shape))
        #print(type(arr))
        # scale the raw pixel intensities to the range [0, 1] - TBD TEST
        arr = np.array(arr, dtype="float") / 255.0
        imageFileName = imageFileName.replace('.jpeg','')
        ImageNameDataHash[str(imageFileName)] = np.array(arr)
    return
from datetime import datetime
print("Loading images at..."+ str(datetime.now()))
sys.stdout.flush()
readTrainData("/kaggle/working/../input/")
print("Loaded " + str(len(ImageNameDataHash)) + " images at..."+ str(datetime.now())) # 1000
import csv
def readTrainCsv():
    raw_df = pd.read_csv('/kaggle/working/../input/trainLabels.csv', sep=',')
    print(type(raw_df)) #<class 'pandas.core.frame.DataFrame'>
    row_count=raw_df.shape[0] #gives number of row count row_count=35126
    col_count=raw_df.shape[1] #gives number of col count col count=2
```

```
    print("row_count="+str(row_count)+" col count="+str(col_count))
    raw_df["PatientID"] = ''
    header_list = list(raw_df.columns)
    print(header_list) # ['image', 'level', 'PatientID']
    # double check if level of left and right are same or not
    ImageLevelHash = {}
    patientIDList = []
    for index, row in raw_df.iterrows():
        # 0 is image, 1 is level, 2 is PatientID, 3 is data
        key = row[0] + ''
        patientID = row[0] + ''
        patientID = patientID.replace('_right','')
        patientID = patientID.replace('_left','')
        #print("Adding patient ID"+ patientID)
        raw_df.at[index, 'PatientID'] = patientID
        patientIDList.append(patientID)
        ImageLevelHash[key] = str(row[1]) # level

    global uniquePatientIDList
    uniquePatientIDList = sorted(set(patientIDList))
    count=0;
    for patientID in uniquePatientIDList:
        left_level = ImageLevelHash[str(patientID+'_left')]
        right_level = ImageLevelHash[str(patientID+'_right')]
        #right_exists = str(patientID+'_right') in raw_df.values
        if (left_level != right_level):
            count = count+1
            #print("Warning for patient="+ str(patientID) + " left_level=" + left_level+ " right_level="
+right_level)
    print("count of images with both left and right eye level not matching="+str(count)) # 2240
    print("number of unique patients="+str(len(uniquePatientIDList))) # 17563
```

```
    return raw_df
random.seed(10)
print("Reading trainLabels.csv...")
df = readTrainCsv()
for i in range(0,10):
    s = df.loc[df.index[i], 'PatientID'] # get patient id of patients
    print(str(i) + " patient's patientID="+str(s))
# df has 3 columns ['image', 'level', 'PatientID']
keepImages =  list(ImageNameDataHash.keys())
df = df[df['image'].isin(keepImages)]
print(len(df)) # 1000
#convert hash to dataframe
imageNameArr = []
dataArr = []
for index, row in df.iterrows():
    key = str(row[0])
    if key in ImageNameDataHash:
        imageNameArr.append(key)
        dataArr.append(np.array(ImageNameDataHash[key])) # np.array

df2 = pd.DataFrame({'image': imageNameArr, 'data': dataArr})
df2_header_list = list(df2.columns)
print(df2_header_list) # ['image', 'data']
print(len(df2))
if len(df) != len(df2):
    print("Error length of df != df2")

for idx in range(0,len(df)):
    if (df.loc[df.index[idx], 'image'] != df2.loc[df2.index[idx], 'image']):
        print("Error " + df.loc[df.index[idx], 'image'] +"==" + df2.loc[df2.index[idx], 'image'])
```

```
print(df2.dtypes)

print(df.dtypes)

df = pd.merge(df2, df, left_on='image', right_on='image', how='outer')

df_header_list = list(df.columns)

print(df_header_list) # 'image', 'data', level', 'PatientID'

print(len(df)) # 1000

print(df.sample())

sample0 = df.loc[df.index[0], 'data']

print(sample0)

print(type(sample0)) # <class 'numpy.ndarray'>

print(sample0.shape) # 128,128,3

from matplotlib import pyplot as plt

plt.imshow(sample0, interpolation='nearest')

plt.show()

print("Sample Image")

X = df['data']

Y = df['level']

# scale the raw pixel intensities to the range [0, 1]

#print(type(X)) # 'pandas.core.series.Series'

#X = np.array(X, dtype="float") / 255.0 -- TBD moved to top

Y = np.array(Y)

# convert the labels from integers to vectors

Y =  to_categorical(Y, num_classes=NUM_CLASSES)

# partition the data into training and testing splits using 75% training and 25% for validation

print("Parttition data into 75:25...")

sys.stdout.flush()

print("Unique patients in dataframe df=" + str(df.PatientID.nunique())) # 500

unique_ids = df.PatientID.unique()

print('unique_ids shape='+ str(len(unique_ids))) #500
```

```
# Refer https://www.kaggle.com/kmader/tf-data-tutorial-with-retina-and-keras

train_ids, valid_ids = train_test_split(unique_ids, test_size = 0.25, random_state = 10) #stratify =
rr_df['level'])

trainid_list = train_ids.tolist()

print('trainid_list shape=', str(len(trainid_list))) # 375


traindf = df[df.PatientID.isin(trainid_list)]

valSet = df[~df.PatientID.isin(trainid_list)]

print(traindf.head())

print(valSet.head())


traindf = traindf.reset_index(drop=True)

valSet = valSet.reset_index(drop=True)


print(traindf.head())

print(valSet.head())

trainX = traindf['data']

trainY = traindf['level']


valX = valSet['data']

valY = valSet['level']


#(trainX, valX, trainY, valY) = train_test_split(X,Y,test_size=0.25, random_state=10)

print('trainX shape=', trainX.shape[0], 'valX shape=', valX.shape[0]) # 750, 250

trainY =  to_categorical(trainY, num_classes=NUM_CLASSES)

valY =  to_categorical(valY, num_classes=NUM_CLASSES)

#construct the image generator for data augmentation

print("Generating images...")

sys.stdout.flush()

aug = ImageDataGenerator(rotation_range=30, width_shift_range=0.1, \
```

```
    height_shift_range=0.1, shear_range=0.2, zoom_range=0.2,\
    horizontal_flip=True, fill_mode="nearest")
def createModel():
    model = Sequential()
    # first set of CONV => RELU => MAX POOL layers
    model.add(Conv2D(32, (3, 3), padding='same', activation='relu', input_shape=inputShape))
    model.add(Conv2D(32, (3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.25))

    model.add(Conv2D(64, (3, 3), padding='same', activation='relu'))
    model.add(Conv2D(64, (3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.25))

    model.add(Conv2D(64, (3, 3), padding='same', activation='relu'))
    model.add(Conv2D(64, (3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.25))

    model.add(Flatten())
    model.add(Dense(512, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(output_dim=NUM_CLASSES, activation='softmax'))
    # returns our fully constructed deep learning + Keras image classifier
    opt = Adam(lr=INIT_LR, decay=INIT_LR / EPOCHS)
    # use binary_crossentropy if there are two classes
    model.compile(loss="categorical_crossentropy", optimizer=opt, metrics=["accuracy"])
    return model
print("Reshaping trainX at..."+ str(datetime.now()))
```

```
#print(trainX.sample())
print(type(trainX)) # <class 'pandas.core.series.Series'>
print(trainX.shape) # (750,)
from numpy import zeros
Xtrain = np.zeros([trainX.shape[0],HEIGHT, WIDTH, DEPTH])
for i in range(trainX.shape[0]): # 0 to traindf Size -1
    Xtrain[i] = trainX[i]
print(Xtrain.shape) # (750,128,128,3)
print("Reshaped trainX at..."+ str(datetime.now()))
print("Reshaping valX at..."+ str(datetime.now()))
print(type(valX)) # <class 'pandas.core.series.Series'>
print(valX.shape) # (250,)
from numpy import zeros
Xval = np.zeros([valX.shape[0],HEIGHT, WIDTH, DEPTH])
for i in range(valX.shape[0]): # 0 to traindf Size -1
    Xval[i] = valX[i]
print(Xval.shape) # (250,128,128,3)
print("Reshaped valX at..."+ str(datetime.now()))
# initialize the model
print("compiling model...")
sys.stdout.flush()
model = createModel()
# print the summary of model
from keras.utils import print_summary
print_summary(model, line_length=None, positions=None, print_fn=None)
# add some visualization
from IPython.display import SVG
from keras.utils.vis_utils import model_to_dot
SVG(model_to_dot(model).create(prog='dot', format='svg'))
# train the network
```

```
print("training network...")
sys.stdout.flush()
#class_mode ='categorical', # 2D one-hot encoded labels
H = model.fit_generator(aug.flow(Xtrain, trainY, batch_size=BS), \
    validation_data=(Xval, valY), \
    s_per_epoch=len(trainX) // BS, \
    epochs=EPOCHS, verbose=1)

# save the model to disk
print("Saving model to disk")
sys.stdout.flush()
model.save("/tmp/mymodel")
# set the matplotlib backend so figures can be saved in the background
# plot the training loss and accuracy
print("Generating plots...")
sys.stdout.flush()
matplotlib.use("Agg")
matplotlib.pyplot.style.use("ggplot")
matplotlib.pyplot.figure()
N = EPOCHS
matplotlib.pyplot.plot(np.arange(0, N), H.history["loss"], label="train_loss")
matplotlib.pyplot.plot(np.arange(0, N), H.history["val_loss"], label="val_loss")
matplotlib.pyplot.plot(np.arange(0, N), H.history["acc"], label="train_acc")
matplotlib.pyplot.plot(np.arange(0, N), H.history["val_acc"], label="val_acc")
matplotlib.pyplot.title("Training Loss and Accuracy on diabetic retinopathy detection")
matplotlib.pyplot.xlabel("Epoch #")
matplotlib.pyplot.ylabel("Loss/Accuracy")
matplotlib.pyplot.legend(loc="lower left")
matplotlib.pyplot.savefig("plot.png")
```

OUTPUT:

0 patient's patientID=10

1 patient's patientID=10

2 patient's patientID=13

3 patient's patientID=13

4 patient's patientID=15

5 patient's patientID=15

6 patient's patientID=16

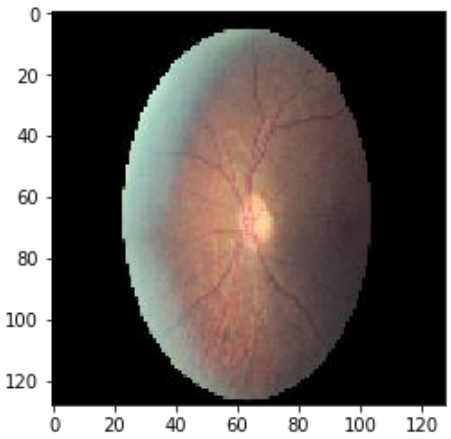
7 patient's patientID=16

8 patient's patientID=17

9 patient's patientID=17

```
[[[0. 0. 0.]
  [0. 0. 0.]
  [0. 0. 0.]
  ...
  [0. 0. 0.]
  [0. 0. 0.]
  [0. 0. 0.]]

 [[0. 0. 0.]
  [0. 0. 0.]
  [0. 0. 0.]
  ...
  [0. 0. 0.]
  [0. 0. 0.]
  [0. 0. 0.]]
[[0. 0. 0.]
  [0. 0. 0.]
  [0. 0. 0.]
  ...
  [0. 0. 0.]
  [0. 0. 0.]
  [0. 0. 0.]]
```

```
 ...


 [[0. 0. 0.]

  [0. 0. 0.]

  [0. 0. 0.]

  ...

  [0. 0. 0.]

  [0. 0. 0.]

  [0. 0. 0.]]


 [[0. 0. 0.]

  [0. 0. 0.]

  [0. 0. 0.]

  ...

[0. 0. 0.]

  [0. 0. 0.]

  [0. 0. 0.]]


 [[0. 0. 0.]

  [0. 0. 0.]

  [0. 0. 0.]

  ...

  [0. 0. 0.]

  [0. 0. 0.]

  [0. 0. 0.]]]
<class 'numpy.ndarray'>
(128, 128, 3)
```



Parttition data into 75:25...

Unique patients in dataframe df=500

unique_ids shape=500

trainid_list shape= 375

image ... PatientID

0 10_left ... 10

1 10_right ... 10

4 15_left ... 15

5 15_right ... 15

6 16_left ... 16

[5 rows x 4 columns]

image ... PatientID

2 13_left ... 13

3 13_right ... 13

12 20_left ... 20

13 20_right ... 20

14 21_left ... 21

[5 rows x 4 columns]

image ... PatientID

0 10_left ... 10

1 10_right ... 10

2 15_left ... 15

3 15_right ... 15

4 16_left ... 16

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 128, 128, 32)	896
conv2d_2 (Conv2D)	(None, 126, 126, 32)	9248
max_pooling2d_1 (MaxPooling2	(None, 63, 63, 32)	0
dropout_1 (Dropout)	(None, 63, 63, 32)	0
conv2d_3 (Conv2D)	(None, 63, 63, 64)	18496
conv2d_4 (Conv2D)	(None, 61, 61, 64)	36928
max_pooling2d_2 (MaxPooling2	(None, 30, 30, 64)	0
dropout_2 (Dropout)	(None, 30, 30, 64)	0
conv2d_5 (Conv2D)	(None, 30, 30, 64)	36928
conv2d_6 (Conv2D)	(None, 28, 28, 64)	36928
max_pooling2d_3 (MaxPooling2	(None, 14, 14, 64)	0
dropout_3 (Dropout)	(None, 14, 14, 64)	0
flatten_1 (Flatten)	(None, 12544)	0
dense_1 (Dense)	(None, 512)	6423040
dropout_4 (Dropout)	(None, 512)	0
dense_2 (Dense)	(None, 5)	2565

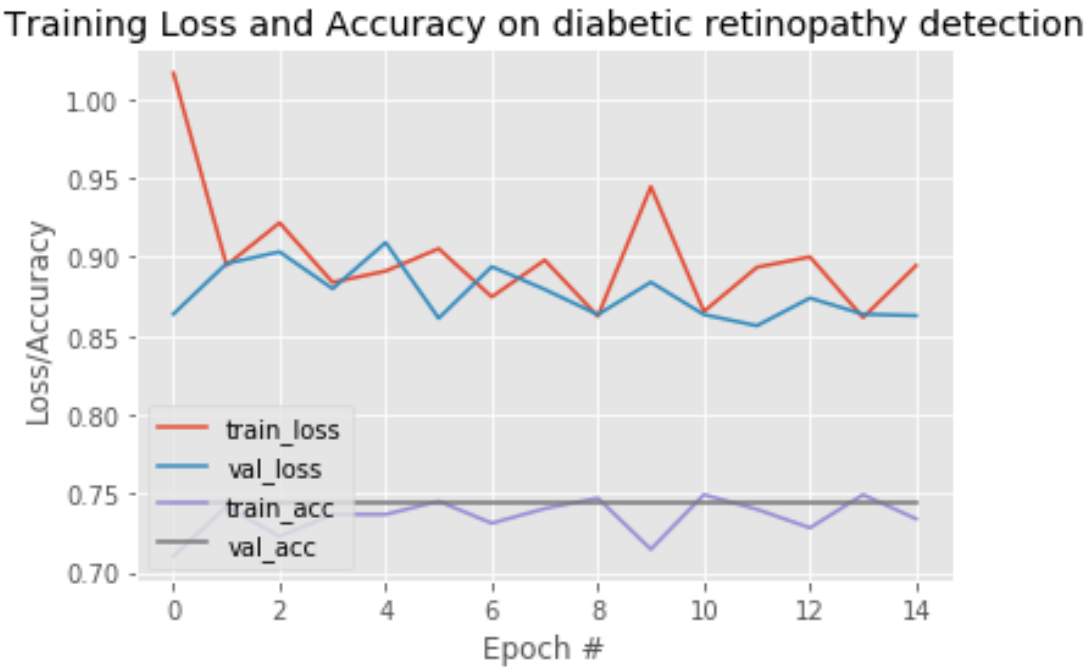
==========

Total params: 6,565,029

Trainable params: 6,565,029

Non-trainable params: 0

_____



Result:

Ex-8 Experimenting with different optimizers

Aim: Implement a Python program for Experimenting with different optimizers Compare the results of the training for each optimizer and determine which optimizer performed the best.

Algorithm:

1: Import the necessary libraries, such as numpy, keras, and matplotlib.

2: Load the dataset to be used for training the model.

3: Define the model architecture.

4: Compile the model by specifying the loss function, metrics, and optimizer.

5: Create a list of optimizers to be tested, such as SGD, RMSprop, Adam, etc.

6: Create a loop that iterates over the list of optimizers. For each iteration: (i) set the optimizer for the model using the model.compile method (ii) fit the model on the dataset using the fit method (iii) store the results of the training, such as accuracy or loss.

7: Plot the results of the training for each optimizer, such as accuracy or loss, over the number of epochs.

8: Compare the results of the training for each optimizer and determine which optimizer performed .

Program:

```
import numpy

as np import

pandas as pd

from sklearn.model_selection import

train_test_split from keras.models import

Sequential from keras.layers import

Dense, Dropout

from keras.callbacks import EarlyStopping, ModelCheckpoint

from tensorflow.keras.optimizers import SGD, Adadelta, Adam, RMSprop, Adagrad, Nadam,
Adamax


SEED = 2022

# Data can be downloaded at https://archive.ics.uci.edu/ml/machine-learning-

databases/winequality/winequality-red.csv data =

pd.read_csv('C:\\Users\\ifsrk\\Documents\\01 Deep Learning\\winequality-red.csv', sep=';') y

= data['quality']
```

```
X = data.drop(['quality'], axis=1)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=SEED)

X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.2,
random_state=SEED)

SEED

print(np.any(np.isnan(X_test)))

print(np.any(np.isinf(X_test)))

print(np.any(np.isnan(X_train)))

print(np.any(np.isinf(X_train)))

print(np.any(np.isnan(y_test)))

print(np.any(np.isinf(y_test)))

print(np.any(np.isnan(y_train)))

print(np.any(np.isinf(y_train)))


def create_model(opt): model =

Sequential() model.add(Dense(100,

input_dim=X_train.shape[1],

activation='relu')) model.add(Dense(50,

activation='relu')) model.add(Dense(25,

activation='relu')) model.add(Dense(10,

activation='relu')) model.add(Dense(1,

activation='linear')) return model def

create_callbacks(opt):

  callbacks = [

  EarlyStopping(monitor='accuracy', patience=50, verbose=2),

  ModelCheckpoint('checkpoints/optimizers_best_' + opt + '.h5', monitor='accuracy',
save_best_only=True, verbose=1)

  ]

  return

callbacks

opts =

dict({

  'sgd': SGD(),

   'sgd-0001': SGD(learning_rate=0.0001, decay=0.00001),
```

```
    'adam': Adam(),
    'adadelta': Adadelta(),
    'rmsprop': RMSprop(),
    'rmsprop-0001': RMSprop(learning_rate=0.0001),
    'nadam': Nadam(),
    'adamax': Adamax()




}
)
X_train.values batch_size = 128 n_epochs = 1000

results = []
# Loop through the
optimizers for opt in
opts:
   model =
create_model(opt)
callbacks =
create_callbacks(opt)
model.compile(loss='mse',
optimizer=opts[opt],
metrics=['accuracy'])

#   model.compile(loss='mse', optimizer=opts[opt], metrics=['mean_squared_error'])
   hist = model.fit(X_train.values, y_train, batch_size=batch_size,
epochs=n_epochs, validation_data=(X_val.values, y_val), verbose=1,
callbacks=callbacks)     print(hist.history)     best_epoch =
np.argmax(hist.history['accuracy'])     print(best_epoch)     best_acc =
hist.history['accuracy'][best_epoch]     print(best_acc)     best_model =
create_model(opt)     best_model.summary()
   # Load the model weights with the highest validation accuracy
best_model.load_weights('checkpoints/optimizers_best_' + opt + '.h5')
```

```
best_model.compile(loss='mse', optimizer=opts[opt],
metrics=['accuracy'])     score = best_model.evaluate(X_test.values,
y_test, verbose=0)     results.append([opt, best_epoch, best_acc,
score[1]]) res = pd.DataFrame(results) res


res.columns = ['optimizer', 'epochs', 'val_accuracy',
'test_accuracy'] res
```

Output:

	0	1	2	3
0	sgd	0	0.0	0.0
1	sgd-0001	0	0.0	0.0
2	adam	0	0.0	0.0
3	adadelta	0	0.0	0.0
4	rmsprop	0	0.0	0.0
5	rmsprop-0001	0	0.0	0.0
6	nadam	0	0.0	0.0
7	adamax	0	0.0	0.0

	optimizer	epochs	val_accuracy	test_accuracy
0	rmsprop	216	0.574219	0.571875
1	adamax	251	0.585938	0.603125
2	sgd-0001	167	0.562500	0.571875
3	nadam	133	0.582031	0.553125
4	adam	139	0.578125	0.581250
5	sgd	0	0.000000	0.000000
6	rmsprop-0001	62	0.550781	0.565625
7	adadelta	208	0.578125	0.575000

Result:

Ex-9 Improving Generalization with Regularization

Aim: Implement a python program for Improving generalizations with regularization.

Algorithm:

1. Define a neural network architecture with a set of parameters that need to be learned through training.
2. Split the dataset into training and validation sets.
3. Choose a suitable regularization technique, such as L1, L2, or Dropout regularization
4. Initialize the weights and biases of the network randomly.
5. Set the number of epochs and the learning rate for training the model.
6. For each epoch, perform the following s:
    a. Feed the training data through the network and compute the loss using a suitable loss function.
    b. Add the regularization term to the loss function.
    c. Use backpropagation to calculate the gradients of the loss with respect to the weights.
    d. Update the weights using the gradients and the learning rate.
    e. Evaluate the performance of the model on the validation set.
7. If the performance on the validation set does not improve for a certain number of epochs, stop the training and return the current model.
8. Otherwise, continue training until the desired level of performance is achieved.
9. Once the training is complete, use the trained model to make predictions on new data

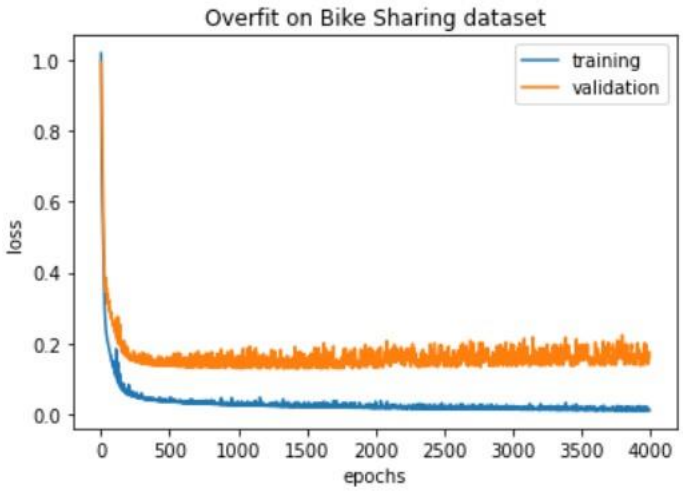
Program:

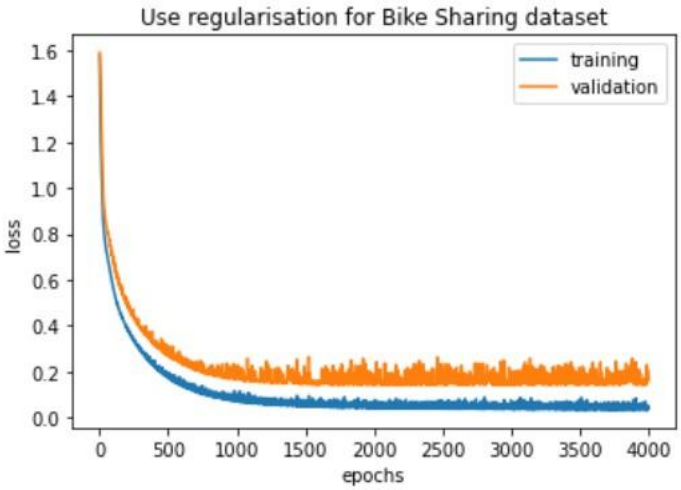
```
import numpy as np import pandas as pd
from matplotlib import pyplot as plt

from keras.models import
Sequential from keras.layers
import Dense, Dropout from
keras import regularizers
# Dataset can be downloaded at https://archive.ics.uci.edu/ml/machine-learning-databases/00275/
data = pd.read_csv('C:\\Users\\ifsrk\\Documents\\01 Deep Learning\\001 Handson\\hour.csv')
# Feature engineering
ohe_features = ['season', 'weathersit', 'mnth', 'hr', 'weekday'] for feature
in ohe_features:     dummies = pd.get_dummies(data[feature],
prefix=feature, drop_first=False)     data = pd.concat([data, dummies],
axis=1) drop_features = ['instant', 'dteday', 'season', 'weathersit',
                'weekday', 'atemp', 'mnth', 'workingday', 'hr', 'casual',
'registered'] data = data.drop(drop_features, axis=1) norm_features =
['cnt', 'temp', 'hum', 'windspeed'] scaled_features = {} for feature in
norm_features:     mean, std = data[feature].mean(),
data[feature].std()     scaled_features[feature] = [mean, std]
data.loc[:, feature] = (data[feature] - mean)/std data[-31*24:]
# Save the final month for testing
# 744 rows test_data = data[-31*24:] data = data[:-31*24] # Extract the target field target_fields =
['cnt'] features, targets = data.drop(target_fields, axis=1), data[target_fields]

test_features, test_targets = test_data.drop(target_fields, axis=1), test_data[target_fields]
# Create a validation set (based on the last )
X_train, y_train = features[:-30*24], targets[:-30*24]
X_val, y_val = features[-30*24:], targets[-30*24:]

model = Sequential()
```

```
model.add(Dense(250, input_dim=X_train.shape[1],
activation='relu')) model.add(Dense(150, activation='relu'))
model.add(Dense(50, activation='relu')) model.add(Dense(25,
activation='relu')) model.add(Dense(1, activation='linear'))

# Compile model model.compile(loss='mse',
optimizer='sgd', metrics=['mse'])

n_epochs =
4000 batch_size
= 1024

history = model.fit(X_train.values, y_train['cnt'],
validation_data=(X_val.values, y_val['cnt']),
batch_size=batch_size, epochs=n_epochs, verbose=0
                )

plt.plot(np.arange(len(history.history['loss'])), history.history['loss'], label='training')
plt.plot(np.arange(len(history.history['val_loss'])), history.history['val_loss'],
label='validation') plt.title('Overfit on Bike Sharing dataset') plt.xlabel('epochs')
plt.ylabel('loss') plt.legend(loc=0) plt.show()

print('Minimum loss: ', min(history.history['val_loss']),
      '\nAfter ', np.argmin(history.history['val_loss']), ' epochs')

# Minimum loss:  0.140975862741
# After  730  epochs

model_reg = Sequential()
model_reg.add(Dense(250, input_dim=X_train.shape[1],
activation='relu',
kernel_regularizer=regularizers.l2(0.005)))
model_reg.add(Dense(150, activation='relu'))
model_reg.add(Dense(50, activation='relu'))
```

```
model_reg.add(Dense(25, activation='relu',
kernel_regularizer=regularizers.l2(0.005)))
model_reg.add(Dense(1, activation='linear'))

# Compile model model_reg.compile(loss='mse',
optimizer='sgd', metrics=['mse'])

history_reg = model_reg.fit(X_train.values, y_train['cnt'],
validation_data=(X_val.values, y_val['cnt']),
batch_size=batch_size, epochs=n_epochs, verbose=1
            )

plt.plot(np.arange(len(history_reg.history['loss'])), history_reg.history['loss'], label='training')
plt.plot(np.arange(len(history_reg.history['val_loss'])), history_reg.history['val_loss'],
label='validation')
plt.title('Use regularisation for Bike
Sharing dataset') plt.xlabel('epochs')
plt.ylabel('loss') plt.legend(loc=0)
plt.show()
print('Minimum loss: ', min(history_reg.history['val_loss']),
    '\nAfter ', np.argmin(history_reg.history['val_loss']), ' epochs')
# Minimum loss:  0.13514482975
# After  3647  epochs
```

Output:




Use regularisation for Bike Sharing dataset
training
validation
loss
epochs
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
500
1000
1500
2000
2500
3000
3500
4000

Result:

Ex-10 Adding dropout to prevent overfitting

Aim: To implement improving generalisation with regularisation.

Algorithm:

1. Initialize your neural network structure.

2. Choose a dropout rate (e.g., 0.2 to 0.5), which represents the fraction of neurons to "turn off" during training.

3. During the training process:

4. For each layer where dropout is applied:

5. Randomly set a fraction of the neurons to zero (dropout) by creating a binary mask.

6. Multiply the input to that layer by this mask to deactivate some neurons.

7. Continue with forward and backward propagation as usual, taking into account the dropped neurons.

8. During evaluation (not training), don't use dropout. Instead, scale the neuron activations by (1 - dropout_rate) to maintain expected values.

9. Repeat the training process for multiple epochs while adjusting other training parameters as needed.

Program:

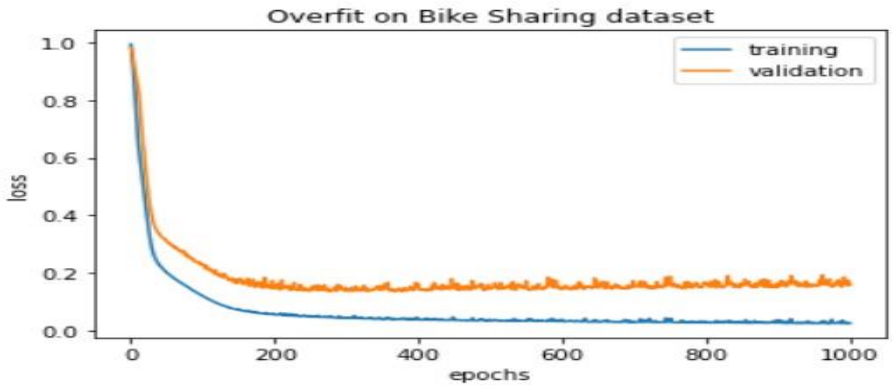
```
import numpy
as np  import
pandas as pd
from matplotlib import pyplot as plt


from keras.models import
Sequential from keras.layers
import Dense, Dropout


import numpy as np
from matplotlib import pyplot as plt


# Dataset can be downloaded at https://archive.ics.uci.edu/ml/machine-learning-databases/00275/


data = pd.read_csv('C:\\Users\\ifsrk\\Documents\\01 Deep Learning\\001 Handson\\hour.csv')
data


# Feature engineering
ohe_features = ['season', 'weathersit', 'mnth', 'hr', 'weekday'] for feature
in ohe_features:     dummies = pd.get_dummies(data[feature],
prefix=feature, drop_first=False)     data = pd.concat([data, dummies],
axis=1)


data


drop_features = ['instant', 'dteday', 'season', 'weathersit', 'weekday', 'atemp', 'mnth', 'workingday',
'hr',
'casual', 'registered']
data = data.drop(drop_features,
axis=1) data


norm_features = ['cnt', 'temp', 'hum', 'windspeed']
```

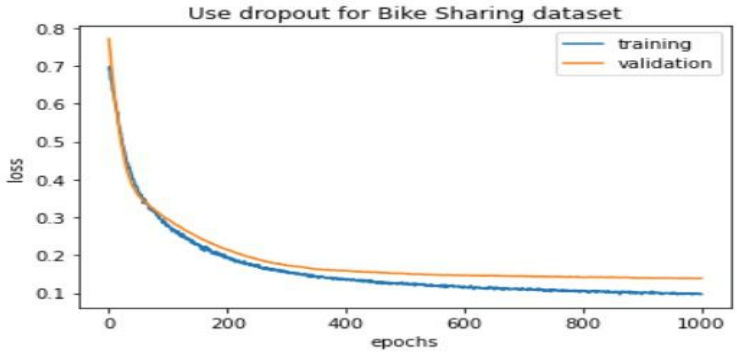
```
scaled_features = {} for feature in
norm_features:     mean, std =
data[feature].mean(), data[feature].std()
scaled_features[feature] = [mean, std]
data.loc[:, feature] = (data[feature] -
mean)/std

scaled_features

# Save the final
month for testing
test_data = data[-
31*24:] data = data[:-
31*24]

# Extract the target field
target_fields = ['cnt']
features, targets = data.drop(target_fields, axis=1), data[target_fields]
test_features, test_targets = test_data.drop(target_fields, axis=1),
test_data[target_fields]

# Create a validation set (based on the last )
X_train, y_train = features[:-30*24], targets[:-30*24]
X_val, y_val = features[-30*24:], targets[-30*24:]

model = Sequential()
model.add(Dense(250, input_dim=X_train.shape[1],
activation='relu')) model.add(Dense(150, activation='relu'))
model.add(Dense(50, activation='relu')) model.add(Dense(25,
activation='relu')) model.add(Dense(1, activation='linear'))

# Compile model
model.compile(loss='mse', optimizer='sgd', metrics=['mse'])
model.summary()
```

```
!pip install pydot
# Visualize network architecture

import pydot import pydotplus import graphviz

from IPython.display import SVG

#from tensorflow.keras.utils.vis_utils import

model_to_dot #from

tensorflow.keras.utils.vis_utils import plot_model

from tensorflow.keras.utils import model_to_dot

from tensorflow.keras.utils import plot_model

SVG(model_to_dot(model, show_shapes=True).create(prog="dot", format="svg"))

# Save the visualization as a file plot_model(model, show_shapes=True,

to_file="dropout_network_model.png")

n_epochs = 1000 batch_size = 1024

history = model.fit(X_train.values, y_train['cnt'],

validation_data=(X_val.values, y_val['cnt']),

batch_size=batch_size, epochs=n_epochs, verbose=1

    )

plt.plot(np.arange(len(history.history['loss'])), history.history['loss'], label='training')

plt.plot(np.arange(len(history.history['val_loss'])), history.history['val_loss'],

label='validation') plt.title('Overfit on Bike Sharing dataset') plt.xlabel('epochs')

plt.ylabel('loss') plt.legend(loc=0) plt.show()

print('Minimum loss: ', min(history.history['val_loss']),

 '\nAfter ', np.argmin(history.history['val_loss']), ' epochs')

# Minimum loss:  0.129907280207
```

```
# After  980  epochs
model_drop =
Sequential()
model_drop.add(Dense(250, input_dim=X_train.shape[1],
activation='relu')) model_drop.add(Dropout(0.20))
model_drop.add(Dense(150, activation='relu'))
model_drop.add(Dropout(0.20)) model_drop.add(Dense(50,
activation='relu')) model_drop.add(Dropout(0.20))
model_drop.add(Dense(25, activation='relu'))
model_drop.add(Dropout(0.20)) model_drop.add(Dense(1,
activation='linear'))
# Compile model model_drop.compile(loss='mse',
optimizer='sgd', metrics=['mse']) model_drop.summary()
history_drop = model_drop.fit(X_train.values, y_train['cnt'],
validation_data=(X_val.values, y_val['cnt']),
batch_size=batch_size, epochs=n_epochs, verbose=1
    )
plt.plot(np.arange(len(history_drop.history['loss'])), history_drop.history['loss'], label='training')
plt.plot(np.arange(len(history_drop.history['val_loss'])), history_drop.history['val_loss'],
label='validation')
plt.title('Use dropout for Bike
Sharing dataset') plt.xlabel('epochs')
plt.ylabel('loss') plt.legend(loc=0)
plt.show()
print('Minimum loss: ', min(history_drop.history['val_loss']),
 '\nAfter ', np.argmin(history_drop.history['val_loss']), ' epochs')
# Minimum loss:  0.126063346863
```

Output:


Overfit on Bike Sharing dataset
training
validation
loss
epochs
1.0
0.8
0.6
0.4
0.2
0.0
0
200
400
600
800
1000


Use dropout for Bike Sharing dataset
training
validation
loss
epochs
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
200
400
600
800
1000

Result:

EX-11 IMAGE AGUMENTATIONT

Aim: To implement Image Agumentation in python language.

Algorithm:

1. Import the necessary libraries, including OpenCV (cv2), NumPy, and Matplotlib for visualization (optional).
2. Load the original image that you want to augment using OpenCV's cv2.imread().
3. To create augmented versions of the image, you can apply various transformations such as rotation, flipping, scaling, and brightness adjustments. Here are some common augmentations
4. Use Matplotlib or any other suitable library to display the augmented images for visual inspection and verification.
5. If you want to generate multiple augmented images, you can repeat s 3 and 4 within a loop, adjusting augmentation parameters as needed.
6. If you want to save the augmented images to disk for later use, use OpenCV's cv2.imwrite() function.
7. Use the augmented images along with the original images in your deep learning model's training dataset to increase diversity and improve model performance.
8. If you have multiple images to augment, repeat the above s for each image.
9. Experiment with different augmentation techniques and parameters to find the most effective augmentations for your specific problem.

PROGRAM

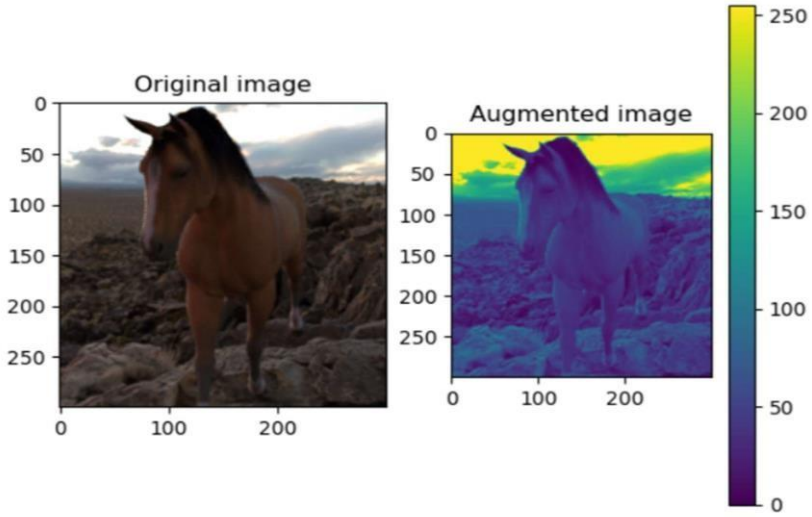
```
import matplotlib.pyplot as
plt import numpy as np
import tensorflow as tf
import
tensorflow_datasets as tfds
from keras import layers
import keras

 #We will use iterators to extract only four random
images w ith labels from the training set and display
them using the matplotlib `.imshow()` function.
get_label_name =
metadata.features['label'].int2str
train_iter = iter(train_ds) fig =
plt.figure(figsize=(8, 10)) for x in range(6):
image, label = next(train_iter)
fig.add_subplot(1, 6, x+1)
plt.imshow(image)
plt.axis('off')
plt.title(get_label_name(label));
#resize
IMG_SIZE1 = 180 IMG_SIZE2 = 180 resize_and_rescale =
keras.Sequential([layers.Resizing(IMG_
SIZE,IMG_SIZE,interpolation="lanczos3"),layers.Rescaling
(1./255)]) result =
resize_and_rescale(image)
```

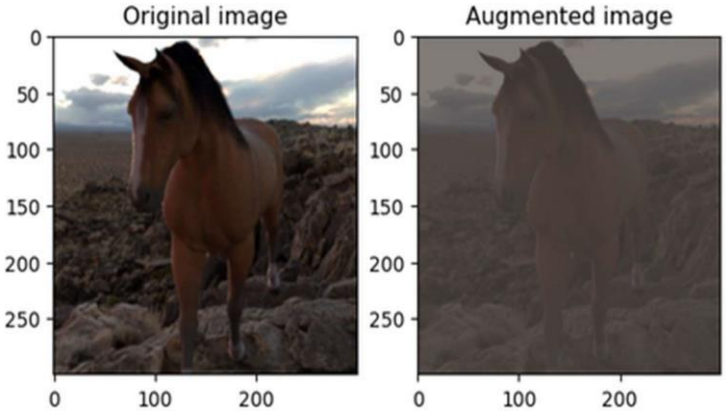
```
plt.axis('off')
plt.imshow(result);
#rotate
data_augmentation
=
keras.Sequential([lay
ers.RandomFlip("ho
rizontal_and_vertica
l"),layers.RandomRo
tation(0.4),])
plt.figure(figsize=(8,
7)) for i in range(6):
augmented_image = data_augmentation(image)
ax = plt.subplot(2, 3, i + 1)
plt.imshow(augmented_image.numpy()/255)
plt.axis("off")

def
random_invert_img(x,
p=0.5): if
tf.random.uniform([]) <
p:
x = (255-
x) else: x
return x def
random_invert(
factor=0.5):

return layers.Lambda(lambda x: random_invert_img(x,
facto  r))
random_invert = random_invert()
```

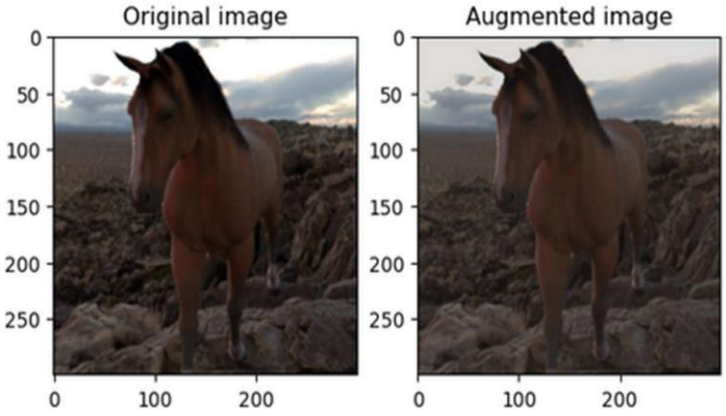
```
plt.figure(figsize=(8, 7))

for i in range(9):
augmented_image = random_invert(image)
ax = plt.subplot(3, 3, i + 1)
plt.imshow(augmented_image.numpy().astype("uint8")) plt.axis("off")

def visualize(original, augmented):

fig = plt.figure() plt.subplot(1,2,1)
plt.title('Original image')
plt.imshow(original)
plt.subplot(1,2,2)
plt.title('Augmented image')
plt.imshow(augmented) flipped =
tf.image.flip_left_right(image)
visualize(image, flipped) grayscaled
= tf.image.rgb_to_grayscale(image)
visualize(image,
tf.squeeze(grayscaled))
_ = plt.colorbar()

saturated =
tf.image.adjust_saturation(image, 4)
visualize(image, saturated)  bright =
tf.image.adjust_brightness(image, 0.4)
visualize(image, bright) for i in
range(3):
seed = (i, 0) # tuple of size (2,)
stateless_random_brightness =
tf.image.stateless_random_b rightness( image,
max_delta=0.95, seed=seed) visualize(image,
```

```
stateless_random_brightness) cropped =
tf.image.central_crop(image, central_fraction=0.5)
visualize(image, cropped)  for i in range(3):
seed = (i, 0) # tuple of size (2,)
stateless_random_crop =
tf.image.stateless_random_crop( image,
size=[210, 300, 3], seed=seed) visualize(image,
stateless_random_crop) rotated =
tf.image.rot90(image) visualize(image, rotated)
for i in range(3):
seed = (i, 0) # tuple of size (2,)
stateless_random_contrast =
tf.image.stateless_random_con trast( image,
lower=0.1, upper=0.9, seed=seed) visualize(image,
stateless_random_contrast)
```

OUTPUT:


Original image
Augmented image
0
50
100
150
200
250
0
100
200
250
200
150
100
50
0


Original image
Augmented image
0
50
100
150
200
250
0
100
200


Original image
Augmented image
0
50
100
150
200
250
0
100
200

EX-12 Imagenet-LeNet

Aim: To implement Imagenet-LeNet using python programming.

Algorithm:

1. Import deep learning libraries like TensorFlow or PyTorch and other necessary libraries.
2. Download and preprocess the ImageNet dataset, which includes resizing images to a manageable size and normalizing pixel values.
3. Create a modified version of LeNet architecture suitable for ImageNet by increasing the number of layers and neurons.
4. Consider using convolutional layers (Conv2D), pooling layers (MaxPooling2D), and fully connected layers (Dense).Use activation functions like ReLU, and consider adding batch normalization layers.
5. Compile the LeNet-based model by specifying the optimizer (e.g., Adam, SGD), loss function (e.g., categorical cross-entropy), and evaluation metric (e.g., top-1 accuracy).
6. Apply data augmentation techniques to increase the diversity of training examples, such as random cropping, flipping, and rotation.
7. Train the model on the ImageNet training dataset.Specify the number of epochs, batch size, and other training parameters.Monitor training progress, including loss and accuracy.
8. Assess the model's performance on the ImageNet validation dataset.

   Calculate top-1 and top-5 accuracy, among other relevant metrics.

9. Fine-tune the model by adjusting hyperparameters, architecture, or regularization techniques to achieve better performance.
10. If needed, deploy the trained model in real-world applications for image classification tasks.
11. Experiment with various architectures beyond LeNet, such as deeper convolutional networks (e.g., VGG, ResNet, Inception) that have proven effective on ImageNet.

PROGRAM

```
import
tensorflow as tf
from tensorflow
import keras
import numpy as
np
(train_x, train_y), (test_x, test_y) =
keras.datasets.mnist.load_data() train_x = train_x / 255.0
test_x = test_x / 255.0
train_x =
tf.expand_dims(train_x,
3) test_x =
tf.expand_dims(test_x,
3) val_x = train_x[:5000]
val_y = train_y[:5000]


lenet_5_model = keras.models.Sequential([
    keras.layers.Conv2D(6, kernel_size=5, strides=1,  activation='tanh',
input_shape=train_x[0].shape, padding='same'), #C1     keras.layers.AveragePooling2D(), #S2
keras.layers.Conv2D(16, kernel_size=5, strides=1, activation='tanh', padding='valid'), #C3
keras.layers.AveragePooling2D(), #S4     keras.layers.Conv2D(120, kernel_size=5, strides=1,
activation='tanh', padding='valid'), #C5     keras.layers.Flatten(), #Flatten
```

keras.layers.Dense(84, activation='tanh'), #F6 keras.layers.Dense(10, activation='softmax')

#Output layer

])

lenet_5_model.compile(optimizer='adam', loss=keras.losses.sparse_categorical_crossentropy, metrics=['accuracy'])

lenet_5_model.fit(train_x, train_y, epochs=5, validation_data=(val_x, val_y))

lenet_5_model.evaluate(test_x, test_y)

OUTPUT

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


Epoch 1/5
1875/1875 [==============================] - 49s 25ms/step - loss: 0.2258 - accuracy: 0.9321 - val_loss: 0.0837 - val_accuracy: 0.9750
Epoch 2/5
1875/1875 [==============================] - 45s 24ms/step - loss: 0.0824 - accuracy: 0.9746 - val_loss: 0.0707 - val_accuracy: 0.9790
Epoch 3/5
1875/1875 [==============================] - 46s 25ms/step - loss: 0.0579 - accuracy: 0.9818 - val_loss: 0.0377 - val_accuracy: 0.9898
Epoch 4/5
1875/1875 [==============================] - 45s 24ms/step - loss: 0.0447 - accuracy: 0.9862 - val_loss: 0.0291 - val_accuracy: 0.9914
Epoch 5/5
1875/1875 [==============================] - 46s 25ms/step - loss: 0.0341 - accuracy: 0.9891 - val_loss: 0.0317 - val_accuracy: 0.9892
<keras.callbacks.History at 0x7ff8fb7fdac0>
```

```
313/313 [==============================] - 4s 13ms/step - loss: 0.0519 - accuracy: 0.9848
[0.05189066007733345, 0.9847999811172485]
```

Result:

EX-12 Imagenet- AlexNet

Aim: To implement Imagenet- AlexNet in python language.

Algorithm

1. Import the necessary deep learning libraries (e.g., TensorFlow or PyTorch) and other supporting libraries.
2. Download and preprocess the ImageNet dataset or a subset of it.Normalize the images.Split the dataset into training, validation, and test sets.
3. Create a neural network model with the following layers:Convolutional layers with appropriate filter sizes, strides, and padding.Max-pooling layers, Fully connected (dense) layers, dropout layers to prevent overfitting. Define appropriate activation functions (e.g., ReLU) and batch normalization as needed.
4. Specify the loss function (e.g., categorical cross-entropy) and optimizer (e.g., SGD or Adam).Choose evaluation metrics (e.g., accuracy).
5. Apply data augmentation techniques such as random cropping, flipping, and rotation to increase the diversity of training examples.
6. Train the model on the training dataset using the compiled model, specifying the number of epochs, batch size, and other training parameters.Monitor training and validation performance to detect overfitting.
7. Fine-tune the model by adjusting hyperparameters or using learning rate schedules if necessary.
8. Evaluate the trained model on the test dataset to measure its performance in terms of accuracy or other relevant metrics.
9. Use the trained model to make predictions on new, unseen images.
10. Visualize model predictions and performance metrics.

PROGRAM

```
import tensorflow
as tf from
tensorflow import
keras import
matplotlib.pyplot
as plt import os
import time

(train_images, train_labels), (test_images, test_labels) =
keras.datasets.cifar10.load_data()

CLASS_NAMES= ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship',
'truck']

validation_images, validation_labels = train_images[:5000], train_labels[:5000]
train_images, train_labels = train_images[5000:], train_labels[5000:]

train_ds = tf.data.Dataset.from_tensor_slices((train_images, train_labels)) test_ds =
tf.data.Dataset.from_tensor_slices((test_images, test_labels)) validation_ds =
tf.data.Dataset.from_tensor_slices((validation_images, validation_labels))
```

```
plt.figure(figsize=(20,20)) for i, (image,
label) in enumerate(train_ds.take(5)):
   ax = plt.subplot(5,5,i+1)
plt.imshow(image)
plt.title(CLASS_NAMES[label.nump
y()[0]])
   plt.axis('off')

def process_images(image, label):
   # Normalize images to have a mean of 0 and standard deviation of 1
image = tf.image.per_image_standardization(image)
   # Resize images from 32x32 to
277x277     image =
tf.image.resize(image, (227,227))
return image, label

train_ds_size = tf.data.experimental.cardinality(train_ds).numpy()
test_ds_size = tf.data.experimental.cardinality(test_ds).numpy()
validation_ds_size =
tf.data.experimental.cardinality(validation_ds).numpy()
print("Training data size:", train_ds_size) print("Test data size:",
test_ds_size) print("Validation data size:", validation_ds_size)

train_ds = (train_ds
            .map(process_images)
            .shuffle(buffer_size=train_ds_size)
            .batch(batch_size=32, drop_remainder=True))

test_ds = (test_ds
            .map(process_images)
            .shuffle(buffer_size=train_ds_size)
            .batch(batch_size=32, drop_remainder=True))
```

```
validation_ds = (validation_ds
        .map(process_images)
        .shuffle(buffer_size=train_ds_size)
        .batch(batch_size=32, drop_remainder=True))

model = keras.models.Sequential([
  keras.layers.Conv2D(filters=96, kernel_size=(11,11), strides=(4,4),
activation='relu', input_shape=(227,227,3)),
keras.layers.BatchNormalization(),  keras.layers.MaxPool2D(pool_size=(3,3),
strides=(2,2)),  keras.layers.Conv2D(filters=256, kernel_size=(5,5),
strides=(1,1), activation='relu', padding="same"),
  keras.layers.BatchNormalization(),
keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),
  keras.layers.Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), activation='relu',
padding="same"),
  keras.layers.BatchNormalization(),
  keras.layers.Conv2D(filters=384, kernel_size=(3,3), strides=(1,1),
activation='relu', padding="same"),  keras.layers.BatchNormalization(),
  keras.layers.Conv2D(filters=256, kernel_size=(3,3), strides=(1,1), activation='relu',
padding="same"),
  keras.layers.BatchNormalization(),
keras.layers.MaxPool2D(pool_size=(3,3),
strides=(2,2)),  keras.layers.Flatten(),
keras.layers.Dense(4096, activation='relu'),
keras.layers.Dropout(0.5),
keras.layers.Dense(4096, activation='relu'),
keras.layers.Dropout(0.5),
keras.layers.Dense(10, activation='softmax')
])

model.compile(loss='sparse_categorical_crossentropy',
optimizer=tf.optimizers.SGD(lr=0.001), metrics=['accuracy']) model.summary()

root_logdir = os.path.join(os.curdir, "logs\\fit\\")
```

```
def get_run_logdir():
    run_id = time.strftime("run_%Y_%m_%d-%H_%M_%S")
    return os.path.join(root_logdir, run_id)

run_logdir = get_run_logdir()
tensorboard_cb = keras.callbacks.TensorBoard(run_logdir)

model.compile(loss='sparse_categorical_crossentropy',
optimizer=tf.optimizers.SGD(lr=0.001), metrics=['accuracy'])
model.summary()

model.fit(train_ds,
        epochs=5,
validation_data=validation_ds,
validation_freq=1,
callbacks=[tensorboard_cb])

tensorboard --logdir logs

model.evaluate(test_ds)
```

OUTPUT

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 55, 55, 96)	34944
batch_normalization (BatchNormalization)	(None, 55, 55, 96)	384
max_pooling2d (MaxPooling2D)	(None, 27, 27, 96)	0
conv2d_1 (Conv2D)	(None, 27, 27, 256)	614656
batch_normalization_1 (BatchNormalization)	(None, 27, 27, 256)	1024
max_pooling2d_1 (MaxPooling2D)	(None, 13, 13, 256)	0
conv2d_2 (Conv2D)	(None, 13, 13, 384)	885120
batch_normalization_2 (BatchNormalization)	(None, 13, 13, 384)	1536
conv2d_3 (Conv2D)	(None, 13, 13, 384)	1327488
batch_normalization_3 (Batc	(None, 13, 13, 384)	1536

```
batch_normalization_3 (Batc  (None, 13, 13, 384)      1536
hNormalization)

conv2d_4 (Conv2D)           (None, 13, 13, 256)       884992

batch_normalization_4 (Batc  (None, 13, 13, 256)      1024
hNormalization)

max_pooling2d_2 (MaxPooling  (None, 6, 6, 256)        0
2D)

flatten (Flatten)           (None, 9216)              0

dense (Dense)               (None, 4096)              37752832

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: 58,327,818
Trainable params: 58,325,066
Non-trainable params: 2,752
_________________________________________________________________
```

Result:

EX-13 RNN

Aim: To implement RNN in python language.

Algorithm:

1: Import the deep learning framework you plan to use (e.g., TensorFlow or PyTorch) and other necessary libraries.

2: Load and preprocess your sequential data. RNNs are commonly used for tasks like sequence prediction or sequence classification.

Preprocess the data, which may include tokenization, one-hot encoding, or embedding, depending on your task.

3: reate an RNN model by defining the layers and their configurations.

Specify the number of hidden units or cells in the RNN layer.

Choose an appropriate activation function (e.g., tanh or ReLU) for the RNN cells.

Optionally, stack multiple RNN layers if needed.

4: Compile the RNN model by specifying the loss function and optimizer.

Choose appropriate metrics for evaluation (e.g., accuracy or mean squared error).

5: Train the RNN model using your preprocessed data.

Specify the number of epochs and batch size.

Monitor training progress and adjust hyperparameters as needed.

6: After training, evaluate the RNN model's performance on a validation or test dataset using relevant metrics.

7:Use the trained RNN model to make predictions on new sequences or data points.

8: Visualize the model's predictions and performance metrics.

PROGRAM

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, LSTM#, CuDNNLSTM


mnist = tf.keras.datasets.mnist  # mnist is a dataset of 28x28 images of handwritten digits and
their labels (x_train, y_train),(x_test, y_test) = mnist.load_data()  # unpacks images to
x_train/x_test and labels to y_tr ain/y_test
```

```
x_train = x_train/255.0
x_test = x_test/255.0

print(x_train.shape)
print(x_train[0].shape)

model = Sequential()

# IF you are running with a GPU, try out the CuDNNLSTM layer type instead (don't pass an
activation, tan h is required)
model.add(LSTM(128, input_shape=(x_train.shape[1:]), activation='relu',
return_sequences=True)) model.add(Dropout(0.2))

model.add(LSTM(128, activation='relu'))
model.add(Dropout(0.1))

model.add(Dense(32, activation='relu'))
model.add(Dropout(0.2))

model.add(Dense(10, activation='softmax'))

opt = tf.keras.optimizers.Adam(lr=0.001, decay=1e-6)

# Compile model
model.compile(
   loss='sparse_categorical_crossentropy',
optimizer=opt,
   metrics=['accuracy'],
)

model.fit(
x_train,
y_train,
epochs=3,
        validation_data=(x_test,y_test))
```

OUTPUT

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 0s 0us/step
(60000, 28, 28)
(28, 28)
```

```
Epoch 1/5
1875/1875 [==============================] - 58s 30ms/step - loss: 0.5438 - accuracy: 0.8256 - val_loss: 0.1829 - val_accuracy:
0.9436
Epoch 2/5
1875/1875 [==============================] - 55s 29ms/step - loss: 0.1551 - accuracy: 0.9585 - val_loss: 0.0985 - val_accuracy:
0.9725
Epoch 3/5
1875/1875 [==============================] - 56s 30ms/step - loss: 0.1086 - accuracy: 0.9705 - val_loss: 0.0922 - val_accuracy:
0.9730
Epoch 4/5
1875/1875 [==============================] - 61s 33ms/step - loss: 0.0852 - accuracy: 0.9766 - val_loss: 0.0568 - val_accuracy:
0.9823
Epoch 5/5
1875/1875 [==============================] - 57s 30ms/step - loss: 0.0665 - accuracy: 0.9823 - val_loss: 0.0559 - val_accuracy:
0.9850
```

Out[9]: <keras.callbacks.History at 0x284cd933910>

Result:

EX-13 LSTM

Aim: To implement LSTM in python language.

Algorithm:

1. Import the deep learning framework you plan to use (e.g., TensorFlow or PyTorch) and other necessary libraries.
2. Load and preprocess your sequential data. LSTMs are commonly used for tasks like sequence prediction, text generation, or sentiment analysis.Preprocess the data, which may include tokenization, one-hot encoding, or embedding, depending on your task.
3. Create an LSTM model by defining the layers and their configurations.
    a. Specify the number of LSTM units (neurons) in each LSTM layer.
    b. Choose an appropriate activation function (usually 'tanh') for the LSTM cells.
    c. Optionally, stack multiple LSTM layers if needed.
    d. You can also add dropout layers to prevent overfitting.
4. Compile the LSTM model by specifying the loss function and optimizer.

Choose appropriate metrics for evaluation (e.g., accuracy or mean squared error).

5. Train the LSTM model using your preprocessed data. Specify the number of epochs and batch size. Monitor training progress and adjust hyperparameters as needed.
6. After training, evaluate the LSTM model's performance on a validation or test dataset using relevant metrics.
7. Use the trained LSTM model to make predictions on new sequences or data points.
8. Visualize the model's predictions and performance metrics.

PROGRAM

```
import pandas import matplotlib.pyplot as plt dataset =
pandas.read_csv('AirPassengers.csv', usecols=[1],
engine='python') plt.plot(dataset) plt.show()


import numpy as
np import
matplotlib.pyplot
as plt import
pandas as pd
import
tensorflow as tf
from tensorflow.keras.models
import Sequential from
tensorflow.keras.layers import
Dense from tensorflow.keras.layers
import LSTM from
sklearn.preprocessing import
MinMaxScaler from sklearn.metrics
import mean_squared_error


# fix random seed for reproducibility
tf.random.set_seed(7)
```

```
# load the dataset dataframe =
pd.read_csv('AirPassengers.csv', usecols=[1],
engine='python') dataset = dataframe.values dataset =
dataset.astype('float32')

# normalize the dataset
scaler =
MinMaxScaler(feature_range=(0, 1))
dataset = scaler.fit_transform(dataset)

# split into train and test sets
train_size = int(len(dataset) *
0.67) test_size = len(dataset) -
train_size
train, test = dataset[0:train_size,:], dataset[train_size:len(dataset),:]
print(len(train), len(test))

# convert an array of values into a
dataset matrix def
create_dataset(dataset,
look_back=1):   dataX, dataY = [],
[]          for i in range(len(dataset)-
look_back-1):                a =
dataset[i:(i+look_back), 0]
            dataX.append(a)
                        dataY.append(dataset[i +
look_back, 0])   return np.array(dataX),
np.array(dataY)

# reshape into X=t and Y=t+1
look_back = 1 trainX, trainY =
create_dataset(train, look_back)
```

```
testX, testY = create_dataset(test,
look_back)

# reshape input to be [samples, time s, features]
trainX = np.reshape(trainX, (trainX.shape[0], 1,
trainX.shape[1])) testX = np.reshape(testX,
(testX.shape[0], 1, testX.shape[1]))

# create and fit the LSTM
network model =
Sequential()
model.add(LSTM(4, input_shape=(1, look_back)))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(trainX, trainY, epochs=100, batch_size=1, verbose=2)

# make predictions
trainPredict =
model.predict(trainX)
testPredict =
model.predict(testX)
# invert predictions
trainPredict =
scaler.inverse_transform(trainPredict)
trainY =
scaler.inverse_transform([trainY])
testPredict =
scaler.inverse_transform(testPredict)
testY =
scaler.inverse_transform([testY]) #
calculate root mean squared error
```

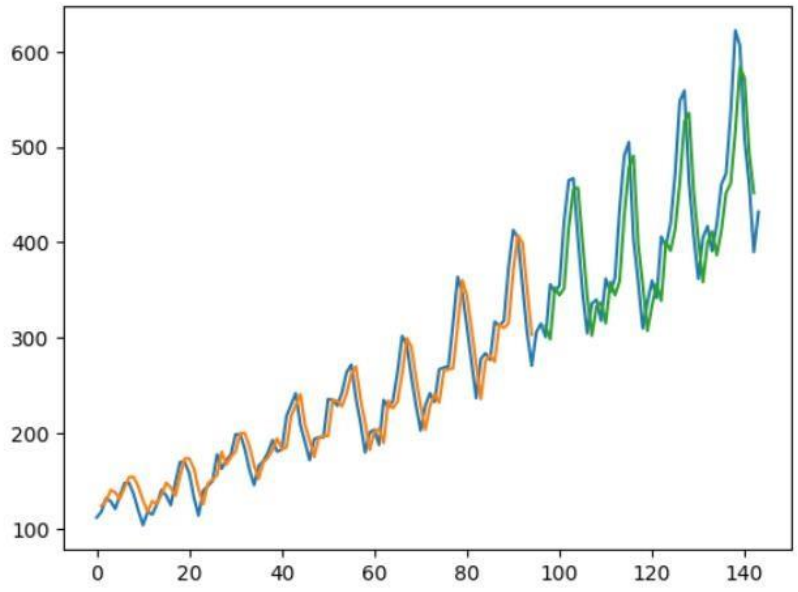
```
trainScore = np.sqrt(mean_squared_error(trainY[0],
trainPredict[:,0])) print('Train Score: %.2f RMSE' %
(trainScore)) testScore =
np.sqrt(mean_squared_error(testY[0], testPredict[:,0]))
print('Test Score: %.2f RMSE' % (testScore))


# shift train predictions for plotting
trainPredictPlot =
np.empty_like(dataset)
trainPredictPlot[:, :] = np.nan
trainPredictPlot[look_back:len(trainPredict)+look_back, :] = trainPredict
# shift test predictions for plotting testPredictPlot =
np.empty_like(dataset) testPredictPlot[:, :] = np.nan
testPredictPlot[len(trainPredict)+(look_back*2)+1:len(dataset)-1, :]
= testPredict
# plot baseline and
predictions
plt.plot(scaler.inverse_transf
orm(dataset))
plt.plot(trainPredictPlot)
plt.plot(testPredictPlot)
plt.show()
# LSTM for international airline passengers problem with
regression framing import numpy as np import
matplotlib.pyplot as plt from pandas import read_csv import
math import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import
Dense from tensorflow.keras.layers
import LSTM from
sklearn.preprocessing import
MinMaxScaler from sklearn.metrics
```

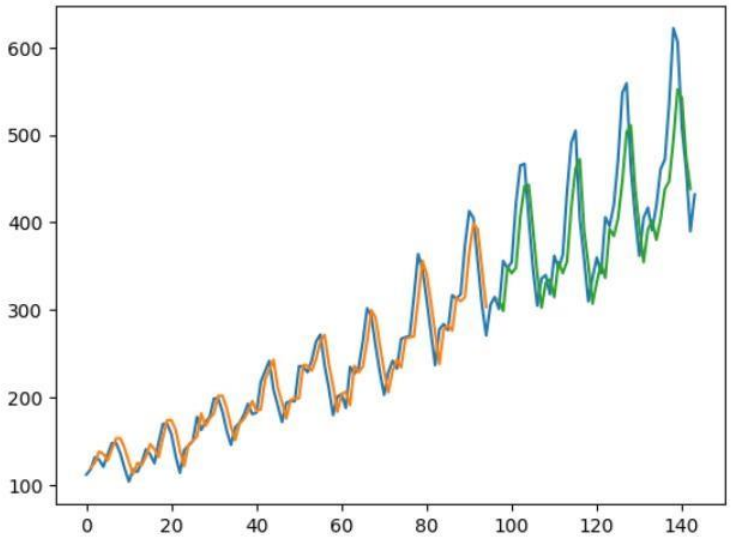
```
import mean_squared_error #
convert an array of values into a
dataset matrix def
create_dataset(dataset,
look_back=1):
        dataX, dataY = [], []
        for i in range(len(dataset)-
look_back-1):              a =
dataset[i:(i+look_back), 0]
        dataX.append(a)
                dataY.append(dataset[i +
look_back, 0])   return np.array(dataX),
np.array(dataY) # fix random seed for
reproducibility tf.random.set_seed(7) #
load the dataset
dataframe = read_csv('AirPassengers.csv', usecols=[1],
engine='python') dataset = dataframe.values dataset =
dataset.astype('float32') # normalize the dataset scaler =
MinMaxScaler(feature_range=(0, 1)) dataset =
scaler.fit_transform(dataset) # split into train and test sets
train_size = int(len(dataset) * 0.67) test_size = len(dataset)
- train_size train, test = dataset[0:train_size,:],
dataset[train_size:len(dataset),:]
# reshape into X=t and
Y=t+1 look_back = 1
trainX, trainY = create_dataset(train, look_back)
testX, testY = create_dataset(test, look_back) #
reshape input to be [samples, time s, features]
trainX = np.reshape(trainX, (trainX.shape[0], 1,
trainX.shape[1])) testX = np.reshape(testX,
(testX.shape[0], 1, testX.shape[1]))
```

```
# create and fit the LSTM
network model =
Sequential()
model.add(LSTM(4, input_shape=(1, look_back)))
model.add(Dense(1))
model.compile(loss='mean_squared_error',
optimizer='adam') model.fit(trainX, trainY,
epochs=100, batch_size=1, verbose=2)
# make predictions
trainPredict =
model.predict(trainX)
testPredict =
model.predict(testX)
# invert predictions
trainPredict =
scaler.inverse_transform(trainPredict)
trainY =
scaler.inverse_transform([trainY])
testPredict =
scaler.inverse_transform(testPredict)
testY =
scaler.inverse_transform([testY]) #
calculate root mean squared error
trainScore = np.sqrt(mean_squared_error(trainY[0],
trainPredict[:,0])) print('Train Score: %.2f RMSE' % (trainScore))
testScore = np.sqrt(mean_squared_error(testY[0],
testPredict[:,0])) print('Test Score: %.2f RMSE' %
(testScore)) # shift train predictions for plotting
trainPredictPlot = np.empty_like(dataset)
trainPredictPlot[:, :] = np.nan
trainPredictPlot[look_back:len(trainPredict)+look_back, :] = trainPredict
```

```
# shift test predictions for plotting testPredictPlot =
np.empty_like(dataset) testPredictPlot[:, :] = np.nan
testPredictPlot[len(trainPredict)+(look_back*2)+1:len(dataset)-1, :]
= testPredict
# plot baseline and
predictions
plt.plot(scaler.inverse_transf
orm(dataset))
plt.plot(trainPredictPlot)
plt.plot(testPredictPlot)
plt.show()
```

OUTPUT





Result:

EX-14

GRU

Aim: To implement GRU in python Languages.

Algorithm:

1:Import the deep learning framework you plan to use (e.g., TensorFlow or PyTorch) and other necessary libraries.

2:Load and preprocess your sequential data. GRUs are commonly used for tasks like sequence prediction, sentiment analysis, and language modeling.

Preprocess the data, which may include tokenization, one-hot encoding, or embedding, depending on your task.

3:Create a GRU model by defining the layers and their configurations.

Specify the number of GRU units (neurons) in each GRU layer.

Choose an appropriate activation function (usually 'tanh') for the GRU cells.

Optionally, stack multiple GRU layers if needed.

You can also add dropout layers to prevent overfitting.

4:Compile the GRU model by specifying the loss function and optimizer.

Choose appropriate metrics for evaluation (e.g., accuracy or mean squared error).

5:Train the GRU model using your preprocessed data.Specify the number of epochs and batch size. Monitor training progress and adjust hyperparameters as needed.

6:After training, evaluate the GRU model's performance on a validation or test dataset using relevant metrics.

7:Use the trained GRU model to make predictions on new sequences or data points.

8:Visualize the model's predictions and performance metrics.

PROGRAM

```
# Importing the
libraries import
numpy as np
import
matplotlib.pyplo
t as plt
plt.style.use('fiv
ethirtyeight')
import pandas
as pd
from sklearn.preprocessing import
MinMaxScaler from keras.models import
Sequential
from keras.layers import Dense, LSTM, Dropout,
GRU, Bidirectional from keras.optimizers import SGD
import math
from sklearn.metrics import mean_squared_error
# Some functions to
help out with def
plot_predictions(test,pr
edicted):
   plt.plot(test, color='red',label='Real IBM Stock
Price')     plt.plot(predicted,
color='blue',label='Predicted IBM Stock Price')
plt.title('IBM Stock Price Prediction')
plt.xlabel('Time')     plt.ylabel('IBM Stock Price')
   plt.legend()
   plt.show()

def return_rmse(test,predicted):
   rmse = math.sqrt(mean_squared_error(test,
predicted))     print("The root mean squared error is
{}.".format(rmse))
   # First, we get the data
dataset = pd.read_csv('../input/IBM_2006-01-01_to_2018-01-
```

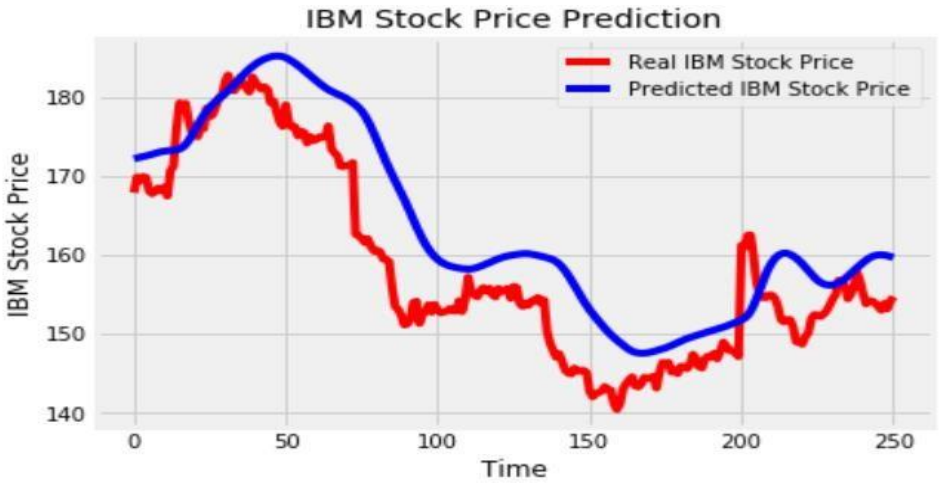
```
01.csv', index_col='Date', parse_dates=['Date'])
dataset.head()
# Checking for missing values
training_set =
dataset[:'2016'].iloc[:,1:2].values
test_set =
dataset['2017':].iloc[:,1:2].values
# We have chosen 'High' attribute for prices. Let's see
what it looks like
dataset["High"][:'2016'].plot(figsize=(16,4),legend=True)
dataset["High"]['2017':].plot(figsize=(16,4),legend=True)
plt.legend(['Training set (Before 2017)','Test set (2017
and beyond)']) plt.title('IBM stock price') plt.show()
# Scaling the training set
sc = MinMaxScaler(feature_range=(0,1))
training_set_scaled =
sc.fit_transform(training_set)
# Since LSTMs store long term memory state, we create a data structure with 60 times and 1
output
# So for each element of training set, we have 60 previous training set elements
X_train =
[] y_train
= [] for i
in
range(60,
2769):
    X_train.append(training_set_scaled[i-60:i,0])
y_train.append(training_set_scaled[i,0])
X_train, y_train = np.array(X_train), np.array(y_train)
# Reshaping X_train for efficient modelling

X_train = np.reshape(X_train, (X_train.shape[0],X_train.shape[1],1))
# The LSTM
architecture
regressor =
Sequential()
# First LSTM layer with Dropout regularisation
regressor.add(LSTM(units=50, return_sequences=True,
input_shape=(X_train.shape[1],1))) regressor.add(Dropout(0.2)) # Second
LSTM layer
regressor.add(LSTM(units=50,
return_sequences=True))
regressor.add(Dropout(0.2)) # Third
LSTM layer
regressor.add(LSTM(units=50,
return_sequences=True))
regressor.add(Dropout(0.2)) # Fourth
LSTM layer
regressor.add(LSTM(units=50))
```

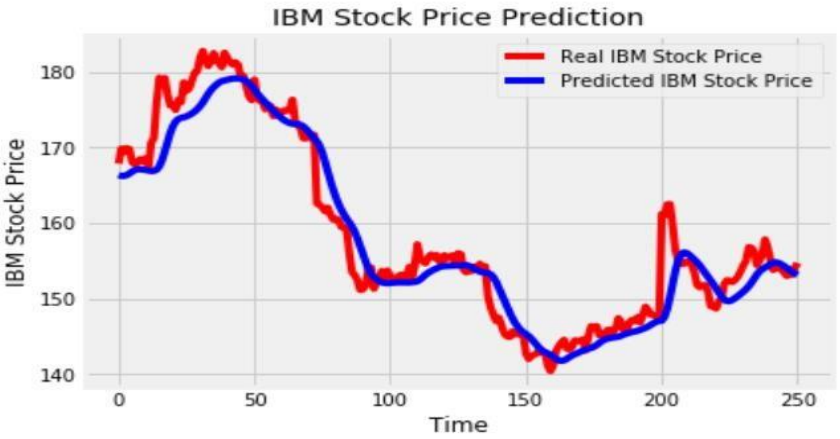
```
regressor.add(Dropout(0.2)) # The output
layer
regressor.add(Dense(units=1))

# Compiling the RNN
regressor.compile(optimizer='rmsprop',loss='mean_squared_error')
# Fitting to the training set
regressor.fit(X_train,y_train,epochs=5,batch_size=32)
# Now to get the test set ready in a similar way as the training set.
# The following has been done so forst 60 entires of test set have 60 previous values which is
impossible to  get unless we take the whole  # 'High' attribute data for processing
dataset_total                                                                    =
pd.concat((dataset["High"][:'2016'],dataset["High"]['2017':]),axis=0)
inputs  =  dataset_total[len(dataset_total)-len(test_set)  -  60:].values
inputs = inputs.reshape(-1,1) inputs  = sc.transform(inputs)
# Preparing X_test and
predicting the prices X_test =
[] for i in range(60,311):
   X_test.append(inputs[i-60:i,0])
X_test = np.array(X_test)
X_test = np.reshape(X_test, (X_test.shape[0],X_test.shape[1],1))
predicted_stock_price = regressor.predict(X_test)
predicted_stock_price = sc.inverse_transform(predicted_stock_price)
# Visualizing the results for LSTM
plot_predictions(test_set,predicted_stock_price) # Evaluating our
model
return_rmse(test_set,predicted_stock_price)
# The GRU architecture
regressorGRU =
Sequential()
# First GRU layer with Dropout regularisation
regressorGRU.add(GRU(units=50, return_sequences=True,
input_shape=(X_train.shape[1],1), activation=' tanh'))
regressorGRU.add(Dropout(0.2))
# Second GRU layer
regressorGRU.add(GRU(units=50, return_sequences=True,
input_shape=(X_train.shape[1],1), activation=' tanh'))
regressorGRU.add(Dropout(0.2))
# Third GRU layer
regressorGRU.add(GRU(units=50, return_sequences=True,
input_shape=(X_train.shape[1],1), activation=' tanh'))
regressorGRU.add(Dropout(0.2))
# Fourth GRU layer
regressorGRU.add(GRU(units=50, activation='tanh'))
regressorGRU.add(Dropout(0.2))
# The output layer
regressorGRU.add(Dense(units=1))
# Compiling the RNN
regressorGRU.compile(optimizer=SGD(lr=0.01,
```

```
decay=1e7, momentum=0.9,
nesterov=False),loss='mean_squared_error')
# Fitting to the training set
regressorGRU.fit(X_train,y_train,epochs=5,batch_size=150)
# Preparing X_test and
predicting the prices X_test =
[] for i in range(60,311):
    X_test.append(inputs[i-60:i,0])
X_test = np.array(X_test)
X_test = np.reshape(X_test, (X_test.shape[0],X_test.shape[1],1))
GRU_predicted_stock_price = regressorGRU.predict(X_test)
GRU_predicted_stock_price = sc.inverse_transform(GRU_predicted_stock_price)
# Visualizing the results for GRU
plot_predictions(test_set,GRU_predicted_stock_price)
# Evaluating GRU
return_rmse(test_set,GRU_predicted_stock_price)
```

OUTPUT

```
Epoch 1/5
2709/2709 [==============================] - 23s 8ms/step - loss: 0.0219
Epoch 2/5
2709/2709 [==============================] - 19s 7ms/step - loss: 0.0099
Epoch 3/5
2709/2709 [==============================] - 19s 7ms/step - loss: 0.0084
Epoch 4/5
2709/2709 [==============================] - 19s 7ms/step - loss: 0.0072
Epoch 5/5
2709/2709 [==============================] - 19s 7ms/step - loss: 0.0063
```





Result: