Practical 1

Aim: - Performing matrix multiplication and finding eigen vectors and eigen values using TensorFlow.

Description:

Matrix:

Matrix is a 2-D rectangular array consisting of numbers represents 2nd order tensor. Matrices should be written in uppercase, bold and italics.

For Example: X. If p and q are positive integers, that is p, q ∈ ℕ then the p×q matrix contains p*q numbers, with p rows and q columns.

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1q} \\ a_{21} & a_{22} & \cdots & a_{2q} \\ \vdots & \vdots & \ddots & \vdots \\ a_{p1} & a_{p2} & \cdots & a_{pq} \end{bmatrix}$$

Vector:

A vector is an ordered array of single numbers represent 1st order tensor. Vectors should be written in lowercase, bold, and italics. For Example: x

$$x = [x_1\ x_2\ x_3\ \ldots\ x_n] \qquad \text{or} \qquad x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

Eigen vector:

Eigenvectors for square matrices are defined as non-zero vector values which when multiplied by the square matrices give the scaler multiple of the vector, i.e. we define an eigenvector for matrix A to be “v” if it specifies the condition, Av = λv

Eigen values:

Eigenvalues are the scalar values associated with the eigenvectors in linear transformation. The word ‘Eigen’ is of German Origin which means ‘characteristic’.

Hence, these are the characteristic value that indicates the factor by which eigenvectors are stretched in their direction. It doesn’t involve the change in the direction of the vector except when the eigenvalue is negative.

When the eigenvalue is negative the direction is just reversed. The equation for eigenvalue is given by

$$Av = \lambda v$$

Tensor:
Sometimes we need an array with more than 2 axis; a tensor is an array of numbers arranged on a grid. It wraps scalar, vector and matrix. We can represent tensor name "A" as A. we can access element of A at coordinates (i, j, k) by writing $A_{i,j,k}$
Matrix multiplication is an operation that takes two matrices as input and produces another matrix as output. In order to perform matrix multiplication, the number of columns in the first matrix must be equal to the number of rows in the second matrix.

Code:

```
import tensorflow as tf
print ("Matrix Multiplication")
x = tf.constant([1,2,3,4,5,6], shape=[2,3])
print(x)
y = tf.constant([7,8,9,10,11,12], shape=[3,2])
print(y)
z = tf.matmul(x,y)
print(z)
matrix_A = tf.random.uniform([2,2], minval=1, maxval=10, dtype=tf.float32, name="matrixA")
print("Matrix A :\n{}\n".format(matrix_A))
eigen_values_A,eigen_vectors_A = tf.linalg.eigh(matrix_A)
print("Eigen Value:\n{}\n\n".format(eigen_values_A))
print("Eigen Vector:\n{}\n\n".format(eigen_vectors_A))
```

Output:

```
Matrix Multiplication
tf.Tensor(
[[1 2 3]
 [4 5 6]], shape=(2, 3), dtype=int32)
tf.Tensor(
[[ 7  8]
 [ 9 10]
 [11 12]], shape=(3, 2), dtype=int32)
tf.Tensor(
[[ 58  64]
 [139 154]], shape=(2, 2), dtype=int32)
Matrix A :
[[9.710442  3.9360168]
 [4.6311445 5.3724747]]

Eigen Value:
[ 2.4275565 12.655359 ]


Eigen Vector:
[[-0.53659356  0.84384084]
 [ 0.84384084  0.53659356]]
```

Code:

```
import tensorflow as tf

print ("Matrix Multiplication")

x = tf.constant([1,2,3,4,5,6], shape=[3,2])

print(x)

y = tf.constant([7,8,9,10,11,12], shape=[2,3])

print(y)

z = tf.matmul(x,y)

print(z)

matrix_A = tf.random.uniform([2,2], minval=1, maxval=10, dtype=tf.float32, name="matrixA")

print("Matrix A :\n{}\n".format(matrix_A))

eigen_values_A,eigen_vectors_A = tf.linalg.eigh(matrix_A)

print("Eigen Value:\n{}\n\n".format(eigen_values_A))

print("Eigen Vector:\n{}\n\n".format(eigen_vectors_A))
```

Output:

```
Matrix Multiplication
tf.Tensor(
[[1 2]
 [3 4]
 [5 6]], shape=(3, 2), dtype=int32)
tf.Tensor(
[[ 7  8  9]
 [10 11 12]], shape=(2, 3), dtype=int32)
tf.Tensor(
[[ 27  30  33]
 [ 61  68  75]
 [ 95 106 117]], shape=(3, 3), dtype=int32)
Matrix A :
[[6.107383  9.800496 ]
 [2.5632954 5.2292867]]

Eigen Value:
[3.0677109 8.26896  ]


Eigen Vector:
[[-0.6446611   0.76446855]
 [ 0.76446855  0.6446611 ]]
```

Learning:

In this practical exercise, TensorFlow is used to execute matrix multiplication and compute eigenvalues and eigenvectors.

Matrix multiplication is presented with the tf.matmul() function. The eigenvalue and eigenvector computations are performed using the tf.linalg.eigh() function.

Initially, random matrices are generated using tf.random.uniform(), and then their eigenvalues and eigenvectors are calculated.

This exercise improves awareness of TensorFlow's matrix operations and numerical computation capabilities, such as eigen analysis, which are important in a variety of domains including physics, engineering, and data analysis.

Practical 2

Aim: - Solving XOR problem using deep feed forward network.

Description:

XOR:

XOR, which stands for "exclusive or," is a logical operation that outputs true only when its two inputs differ (one is true and the other is false). In other words, if both inputs are the same (either both true or both false), the XOR operation will yield false. It's commonly represented by the symbol $\oplus$ or XOR.

Deep feedforward network:

A deep feedforward network, also known as a multilayer perceptron (MLP), consists of multiple layers of sigmoid neurons. It's adept at handling nonlinear data by using hidden layers to capture complex relationships between input and output. The primary aim is to approximate a function, such as mapping input x to output y. Through parameter optimization, the network learns the best function approximation. Information flows in a single direction, from input to output, distinguishing it from recurrent neural networks which incorporate feedback connections. These networks are foundational in various commercial applications and are vital for machine learning practitioners.

Cross Entropy:

Cross entropy is a concept from information theory and machine learning that measures the difference between two probability distributions.

In the context of machine learning and classification tasks, it quantifies the dissimilarity between the predicted probabilities and the actual labels.

Mathematically, the cross entropy between two probability distributions (p) and (q) over the same set of events is defined as:

$$H(p, q) = -\sum_x p(x) \log(q(x))$$

Where:

- (p(x)) is the true probability of event (x)
- (q(x)) is the predicted probability of event (x)

Optimizer:

An optimizer in the context of machine learning and neural networks is an algorithm or method used to adjust the model's parameters during training in order to minimize the loss function. The goal is to find the optimal set of parameters that best fit the training data and generalize well to unseen data.

ReLU:

ReLU, which stands for Rectified Linear Unit, is an activation function used in artificial neural networks. It's defined mathematically as:

f(x) = max(0, x)

In simpler terms, the ReLU function returns the input (x) if (x) is positive, and zero otherwise. This activation function introduces non-linearity to the network, allowing it to learn from the error and make adjustments to the weights during training.
ReLU has become popular in deep learning due to its simplicity and effectiveness in training deep neural networks. It helps mitigate the vanishing gradient problem, which can occur during the training of deep networks with other activation functions like sigmoid or tanh.
Sigmoid:
The sigmoid function is a popular activation function used in neural networks, especially in the output layer for binary classification tasks. It squashes the output of a neuron to a value between 0 and 1, which can be interpreted as a probability.
Mathematically, the sigmoid function is defined as:

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

Here's what the sigmoid function does:
- For large positive values of (x), (f(x)) approaches 1.
- For large negative values of (x), (f(x)) approaches 0.
- For (x = 0), (f(x)) is exactly 0.5.

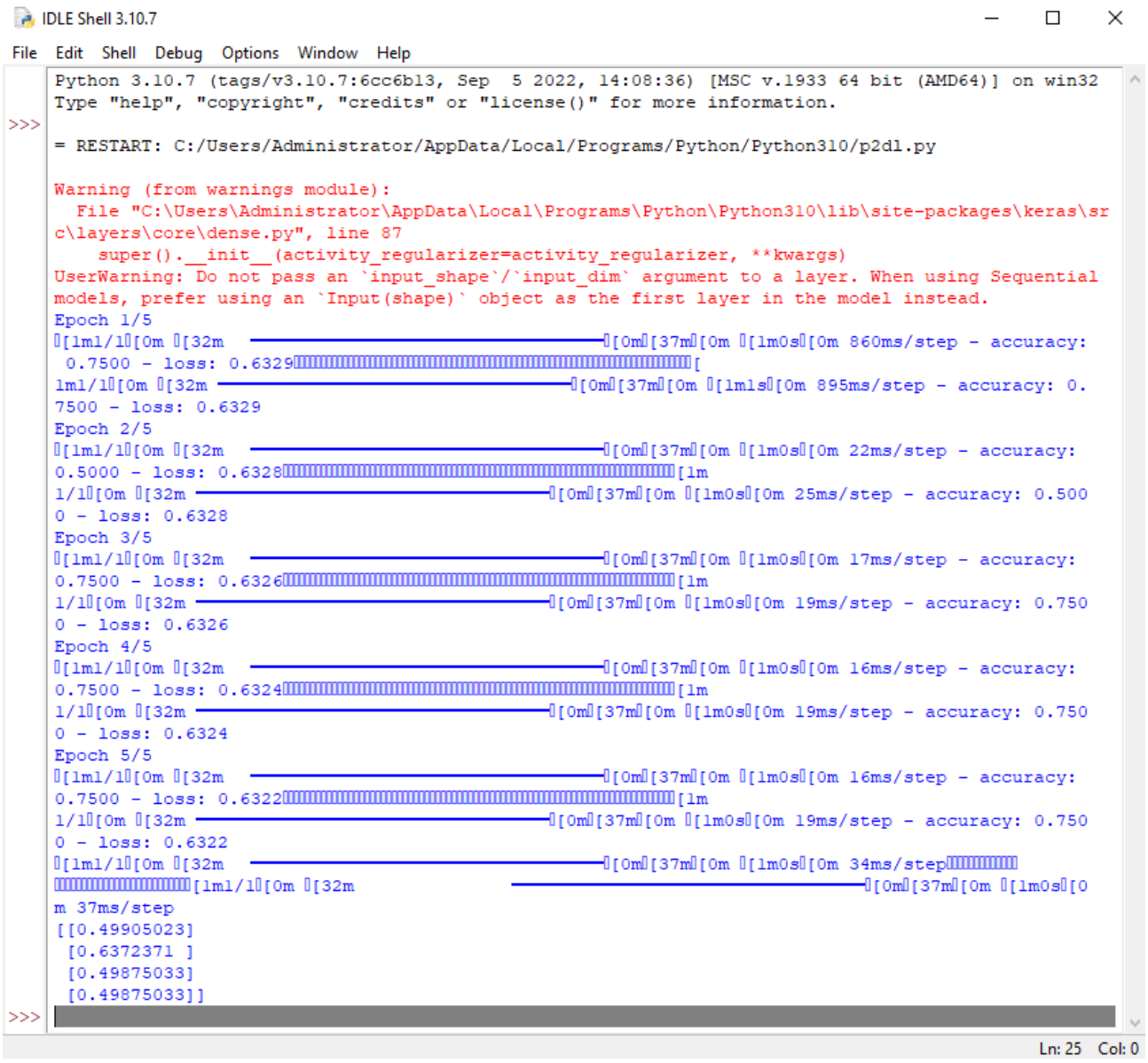
The sigmoid function introduces non-linearity to the network, allowing it to learn from the error and make adjustments to the weights during training. However, it's worth noting that sigmoid can suffer from the vanishing gradient problem, especially in deeper networks, which is why other activation functions like ReLU are often preferred for hidden layers.

Code:

```
import numpy as np

from keras.models import Sequential

from keras.layers import Dense

model = Sequential()

model.add(Dense(units=2, activation='relu', input_dim=2))

model.add(Dense(units=1, activation='sigmoid'))

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

X = np.array([[0.,0.],[0.,1.],[1.,0.],[1.,1.]])

y = np.array([0.,1.,1.,0.])

model.fit(X, y, epochs=5)
```

```
predictions = model.predict(X)
print(predictions)
```

Output:



Learning:

In this session, we built a neural network with Keras to answer the XOR classification issue. The network was made up of three layers: an input layer with two units, a hidden layer with ReLU activation, and an output layer with sigmoid activation. We used binary crossentropy as a loss function and the Adam optimizer for training. We trained the model on XOR input-output pairs for 1000 epochs and printed the learned weights to see how the parameters changed. Predictions on the input data proved the model's ability to approximate the XOR function while accurately classifying the inputs. This experiment demonstrated the necessity of selecting the appropriate model architecture, activation functions, loss functions, and optimizers for efficient classification jobs. Furthermore, it emphasised the need of knowing and interpreting model weights and predictions to evaluate performance and reliability.

Practical 3

Aim : - Implementing deep neural network for performing classification task.

Description:
Deep neural network

A deep neural network (DNN) is a type of artificial neural network with multiple layers between the input and output layers. It's designed to recognize patterns and make decisions by mimicking the human brain's structure and function through interconnected nodes (neurons). The depth of a neural network refers to the number of hidden layers it contains, allowing it to learn hierarchical representations of data, capturing complex patterns and relationships.

Structure:

1. Input Layer: The first layer that receives input features from the dataset.
2. Hidden Layers: Intermediate layers between the input and output layers where the complex computations occur. Each hidden layer consists of multiple neurons that apply weighted sums of inputs and activation functions (e.g., ReLU, sigmoid).
3. Output Layer: The final layer that produces the predictions or classifications based on the input data. The number of neurons in this layer depends on the problem type (e.g., regression, classification).

Activation Functions:

- ReLU (Rectified Linear Unit): Commonly used in hidden layers to introduce non-linearity.
- Sigmoid: Often used in the output layer for binary classification.
- Tanh: Another activation function used to introduce non-linearity, similar to sigmoid but ranging from -1 to 1.

Training:

- Backpropagation: An optimization algorithm used to update the weights of the neural network based on the error between the predicted and actual outputs. It computes the gradient of the loss function with respect to each weight and adjusts the weights to minimize the loss.

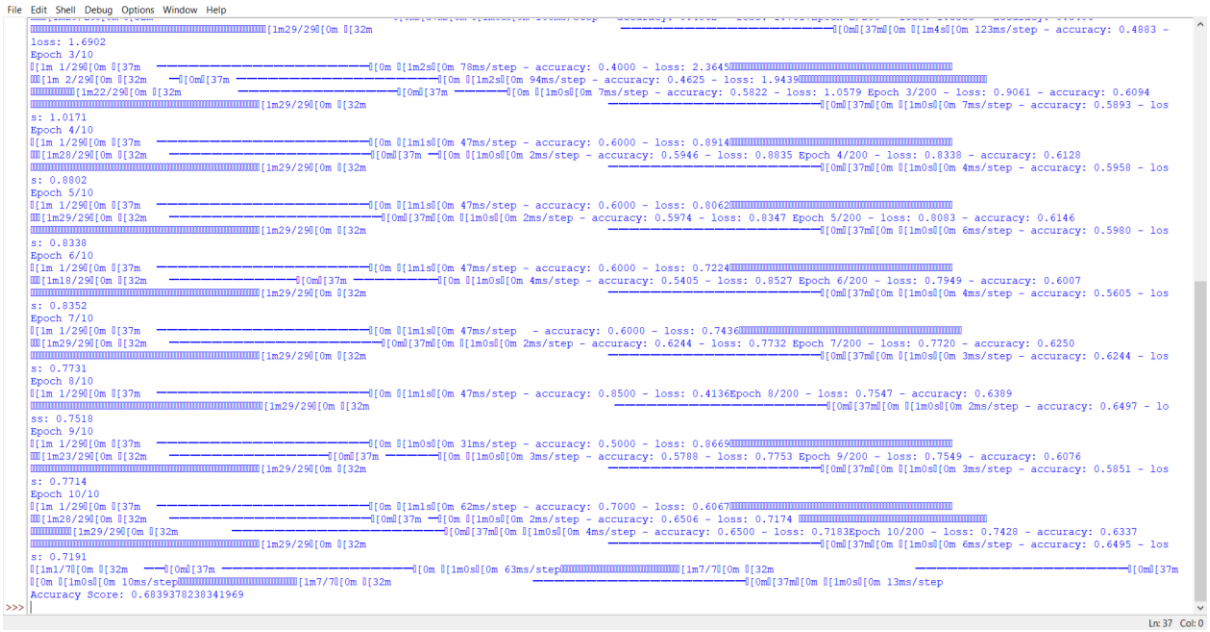
Applications:
- Image Recognition: DNNs have achieved state-of-the-art performance in tasks like image classification, object detection, and facial recognition.
-Natural Language Processing (NLP): Used in machine translation, sentiment analysis, chatbots, and text generation.
- Speech Recognition: DNNs power voice assistants and speech-to-text applications.

Code:

```
# diabetes_classifier.py

from keras.models import Sequential

from keras.layers import Dense, Input

import pandas as pd

from sklearn.model_selection import train_test_split

from sklearn.metrics import accuracy_score

from keras.callbacks import LambdaCallback

# Define column names

names = ["No. of pregnancies", "Glucose level", "Blood Pressure", "Skin thickness", "Insulin",
"BMI", "Diabetes pedigree", "Age", "Class"]
# Load dataset
df = pd.read_csv("diabetes.csv", names=names)

# Ensure all data is numeric

df = df.apply(pd.to_numeric, errors='coerce').fillna(0)

# Split features and target

X = df.iloc[:, :-1]

y = df.iloc[:, -1]

# Split the dataset into training and testing sets

xtrain, xtest, ytrain, ytest = train_test_split(X, y, test_size=0.25, random_state=1)

# Define the model
binaryc = Sequential()

binaryc.add(Input(shape=(8,)))

binaryc.add(Dense(units=10, activation='relu'))

binaryc.add(Dense(units=8, activation='relu'))

binaryc.add(Dense(units=1, activation='sigmoid'))

# Compile the model
binaryc.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

# Custom callback to print each epoch

print_epoch_callback = LambdaCallback(
```

```
    on_epoch_end=lambda epoch, logs: print(f'Epoch {epoch + 1}/{200} - loss: {logs["loss"]:.4f} - accuracy: {logs["accuracy"]:.4f}')

)

# Train the model

binaryc.fit(xtrain, ytrain, epochs=10, batch_size=20, callbacks=[print_epoch_callback])

# Make predictions

predictions = binaryc.predict(xtest)

class_labels = [1 if i > 0.5 else 0 for i in predictions]

# Print accuracy score

print('Accuracy Score:', accuracy_score(ytest, class_labels))
```

Output:



Learning:

This hands-on activity involved creating and training a deep neural network for a binary classification challenge using TensorFlow. We imported the "pima-indians-diabetes.csv" dataset, separated it into features and target variables, and standardised the features for better model convergence. The neural network design was made up of thick layers with different unit sizes and activation functions. The model was built using the Adam optimizer and a categorical crossentropy loss function.The model was trained using training data, with a validation split to track performance. Finally, we ran the model on test data and obtained a test accuracy score. This exercise illustrated critical phases in data preparation, model development, training, and assessment, giving you a solid grasp of neural network workflows for classification problems.

Practical 4

Aim : - Using Feed Forward Network with multiple hidden layers for performing multiclass classification and predicting the class.

Description:
Multiclass classification is a type of supervised learning task where the goal is to classify instances into one of three or more classes. Unlike binary classification, which has only two classes, multiclass classification involves distinguishing between multiple classes or categories.

Algorithms for Multiclass Classification

Several algorithms can handle multiclass classification tasks, including:

- One-vs-All (OvA): This strategy involves training a single classifier per class, treating all other classes as negative. The class with the highest confidence score is then assigned to the instance.

- One-vs-One (OvO): In this strategy, a binary classifier is trained for each pair of classes. For N classes, N*(N-1)/2 classifiers are trained. The class that wins the most duels is assigned to the instance.

- Multinomial Logistic Regression: It generalizes logistic regression to multiclass problems, where the target variable can take on more than two classes.

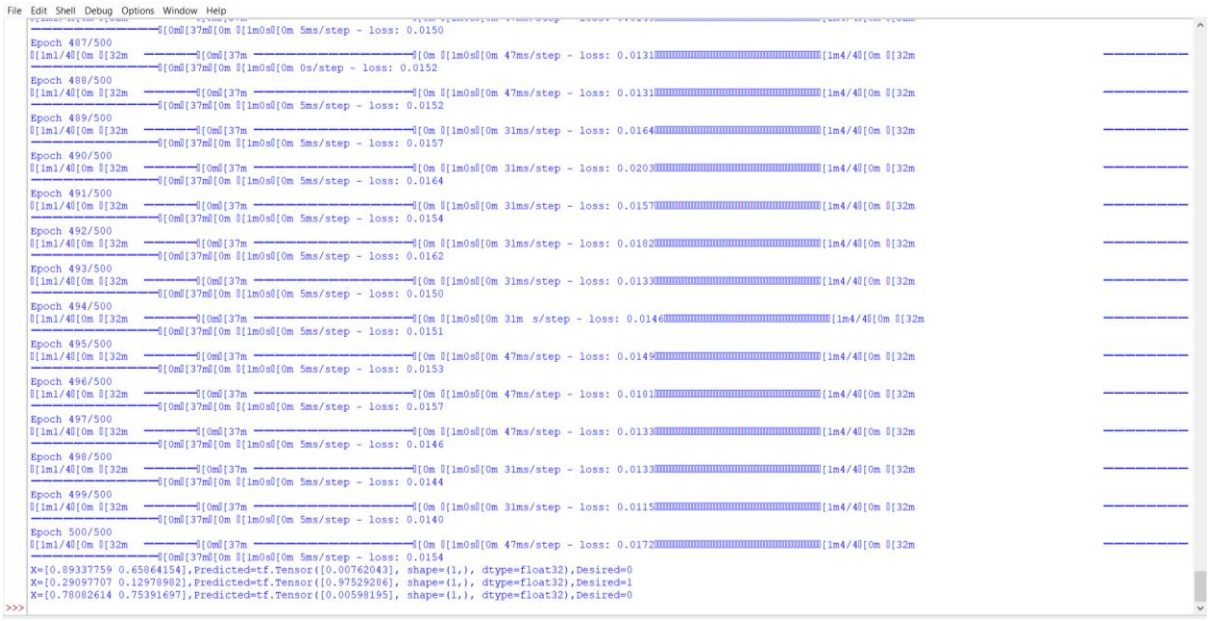
- Decision Trees, Random Forests, and Gradient Boosted Trees: These tree-based algorithms can be adapted for multiclass classification by modifying the splitting criteria and aggregation methods.

Code:

#4a(Aim: Using deep feed forward network with two hidden layers for performing classification and predicting the class)

```
from keras.models import Sequential

from keras.layers import Dense

from sklearn.datasets import make_blobs

from sklearn.preprocessing import MinMaxScaler

X,Y=make_blobs(n_samples=100,centers=2,n_features=2,random_state=1)

scalar=MinMaxScaler()

scalar.fit(X)

X=scalar.transform(X)

model=Sequential()

model.add(Dense(4,input_dim=2,activation='relu'))

model.add(Dense(4,activation='relu'))

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

model.compile(loss='binary_crossentropy',optimizer='adam')

model.fit(X,Y,epochs=500)

Xnew,Yreal=make_blobs(n_samples=3,centers=2,n_features=2,random_state=1)

Xnew=scalar.transform(Xnew)

Ynew=model.predict_step(Xnew)

for i in range(len(Xnew)):

print("X=%s,Predicted=%s,Desired=%s"%(Xnew[i],Ynew[i],Yreal[i]))
```

Output:



b) Using a deep feed forward network with two hidden layers for performing classification and predicting the probability of class.

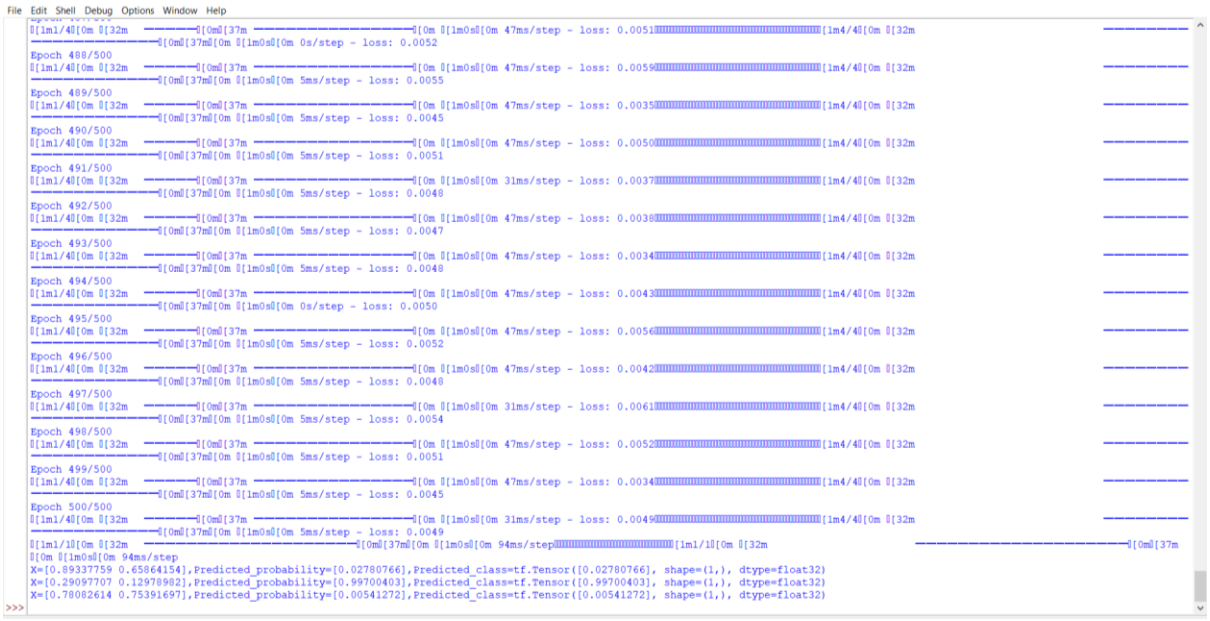
Code:

#4b(b) Using a deep feed forward network with two hidden layers for performing classification and predicting the probability of class.)

```
from keras.models import Sequential

from keras.layers import Dense

from sklearn.datasets import make_blobs

from sklearn.preprocessing import MinMaxScaler

X,Y=make_blobs(n_samples=100,centers=2,n_features=2,random_state=1)

scalar=MinMaxScaler()

scalar.fit(X)

X=scalar.transform(X)

model=Sequential()

model.add(Dense(4,input_dim=2,activation='relu'))

model.add(Dense(4,activation='relu'))
```

```
model.add(Dense(1,activation='sigmoid'))
model.compile(loss='binary_crossentropy',optimizer='adam')
model.fit(X,Y,epochs=500)
Xnew,Yreal=make_blobs(n_samples=3,centers=2,n_features=2,random_state=1)
Xnew=scalar.transform(Xnew)
Yclass=model.predict_step(Xnew)
Ynew=model.predict(Xnew)
for i in range(len(Xnew)):
 print("X=%s,Predicted_probability=%s,Predicted_class=%s"%(Xnew[i],Ynew[i],Yclass[i]))
```

Output:

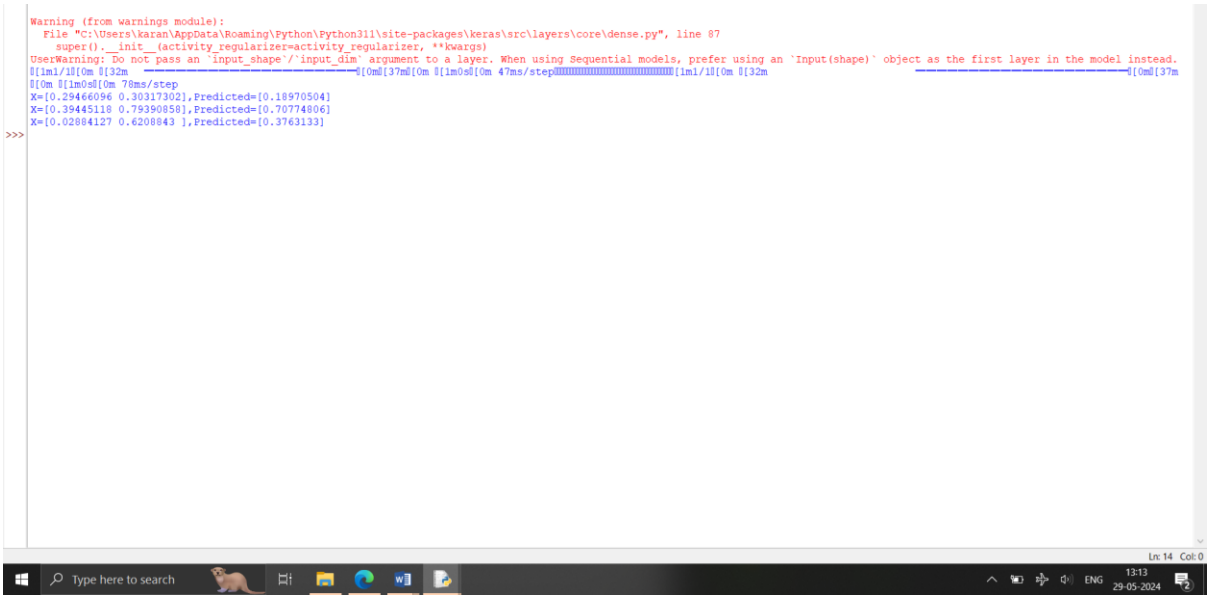


c) Using a deep field forward network with two hidden layers for performing linear regression and predicting values.

Code:

```
#4c(c) Using a deep field forward network with two hidden layers for performing linear regression and predicting values.)
from keras.models import Sequential
from keras.layers import Dense
from sklearn.datasets import make_regression
from sklearn.preprocessing import MinMaxScaler
X,Y=make_regression(n_samples=100,n_features=2,noise=0.1,random_state=1)
scalarX,scalarY=MinMaxScaler(),MinMaxScaler()
scalarX.fit(X)
scalarY.fit(Y.reshape(100,1))
X=scalarX.transform(X)
Y=scalarY.transform(Y.reshape(100,1))
model=Sequential()
model.add(Dense(4,input_dim=2,activation='relu'))
model.add(Dense(4,activation='relu'))
model.add(Dense(1,activation='sigmoid'))
model.compile(loss='mse',optimizer='adam')
model.fit(X,Y,epochs=500,verbose=0)
Xnew,a=make_regression(n_samples=3,n_features=2,noise=0.1,random_state=1)
Xnew=scalarX.transform(Xnew)
Ynew=model.predict(Xnew)
for i in range(len(Xnew)):
 print("X=%s,Predicted=%s"%(Xnew[i],Ynew[i]))
```

Output:



Learning:

This practical demonstrated important elements for creating a Feed Forward Neural Network for multiclass categorization. Beginning with data preparation, we read and divided the dataset into features and target variables.The target variable was encoded with 'LabelEncoder' and then transformed to one-hot encoding with 'np_utils.to_categorical'. A neural network model was created using Keras' 'Sequential', which included numerous hidden layers with'relu' activation functions and a'softmax' activation in the output layer to calculate class probabilities.The model was built using the 'categorical_crossentropy' loss and the 'adam' optimizer, then trained on real data. Predictions were produced and compared to real labels using 'np.argmax()', with the findings saved in a DataFrame for further study. This practical gave hands-on experience in data preparation, neural network architecture design, model compilation, training, and assessment for multiclass classification tasks.

Practical 5

Aim : -
a) Evaluating feed forward deep network for regression using KFold Cross Validation

Description:

Feed Forward Deep Network using KFold Cross Validation

A Feedforward Deep Network, a fundamental architecture in deep learning, consists of multiple layers where information flows in one direction, from input to output. KFold Cross Validation is a technique used to assess the performance and robustness of a model. In this method, the dataset is partitioned into k equally sized folds. The model is trained k times, each time using k-1 folds for training and the remaining fold for validation. This process ensures that each data point is used for validation exactly once. By averaging the performance metrics across all folds, KFold Cross Validation provides a reliable estimate of the model's performance, crucial for model selection and hyperparameter tuning.

Functions:

pd.read_csv(): This function from the pandas library is used to read data from a CSV file into a DataFrame, which is a two-dimensional labeled data structure.

Sequential(): This function from Keras initializes a linear stack of layers. It's a simple way to build a neural network model layer by layer.

Dense(): This function from Keras adds a fully connected layer to the neural network model. The parameters specify the number of neurons in the layer, the initialization method for the layer weights, and the activation function to be used.

StandardScaler(): This function from scikit-learn is used to standardize features by removing the mean and scaling to unit variance. It's an essential preprocessing step to ensure that features are on a similar scale, which can improve the performance of the neural network.

KerasRegressor(): This function from scikeras.wrappers wraps a Keras model (in this case, a regression model) into a scikit-learn compatible estimator. It allows using Keras models seamlessly within scikit-learn pipelines and cross-validation.

Pipeline(): This function from scikit-learn creates a pipeline by chaining together multiple preprocessing and modeling steps. In this code, it's used to combine standardization of features and the KerasRegressor model into a single entity.

KFold(): This function from scikit-learn is used to perform K-Fold cross-validation. It splits the dataset into K consecutive folds, where each fold is then used once as a validation while the K - 1 remaining folds form the training set.

cross_val_score(): This function from scikit-learn is used to evaluate a model by cross-validation. It computes the score (mean squared error in this case) of the model for each cross-validation fold and returns the list of scores.

Code:

```
#5a(Evaluating feed forward deep network for regression using KFold cross validation.)

import pandas as pd

from keras.models import Sequential

from keras.layers import Dense

# from keras.wrappers.scikit_learn import KerasRegressor

from scikeras.wrappers import KerasRegressor

from sklearn.model_selection import cross_val_score, KFold

from sklearn.preprocessing import StandardScaler

from sklearn.pipeline import Pipeline

from sklearn.neural_network import MLPRegressor

#dataframe = pd.read_csv("housing.csv")

dataframe = pd.read_csv(r"housing.csv")

dataset = dataframe.values

# Print the shape of dataset to verify the number of features and samples

print("Shape of dataset:", dataset.shape)

X = dataset[:, :-1]  # Select all columns except the last one as features

Y = dataset[:, -1]   # Select the last column as target variable

def wider_model():

    model = Sequential()

    model.add(Dense(15, input_dim=13, kernel_initializer='normal', activation='relu'))

    # model.add(Dense(20, input_dim=13, kernel_initializer='normal', activation='relu'))

    model.add(Dense(13, kernel_initializer='normal', activation='relu'))

    model.add(Dense(1, kernel_initializer='normal'))

    model.compile(loss='mean_squared_error', optimizer='adam')

    return model
```

```
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasRegressor(build_fn=wider_model, epochs=10, batch_size=5)))
pipeline = Pipeline(estimators)
kfold = KFold(n_splits=10)
results = cross_val_score(pipeline, X, Y, cv=kfold)
print("Wider: %.2f (%.2f) MSE" % (results.mean(), results.std()))
```

Output:

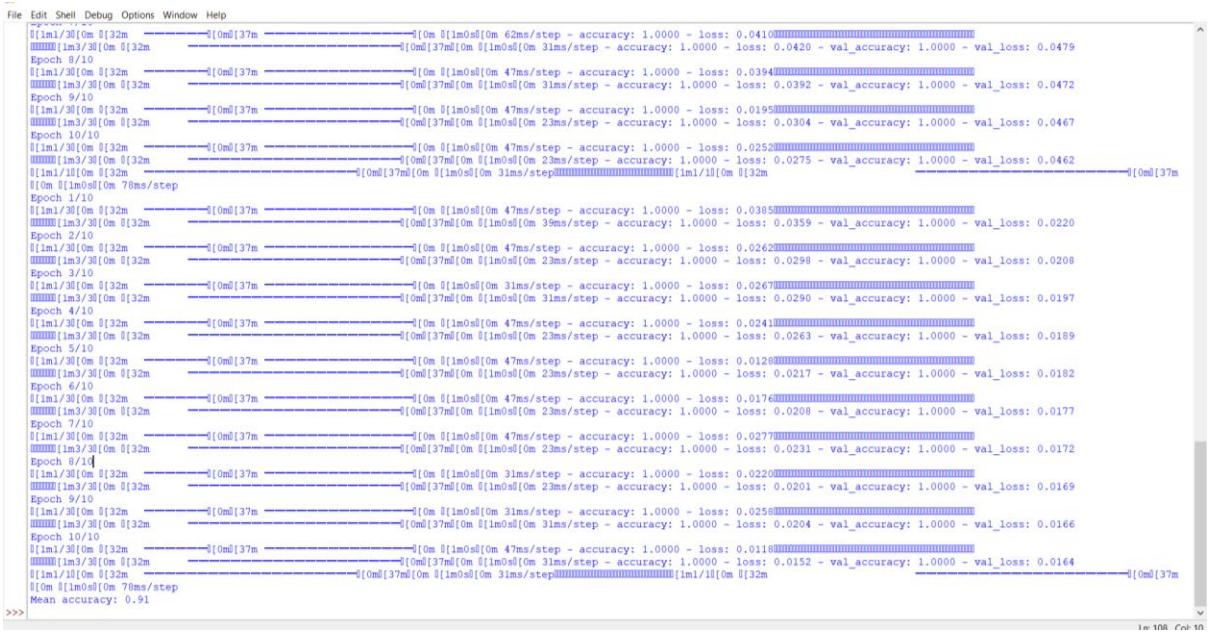
b) Evaluating feed forward deep network for multiclass classification using KFold cross-validation.

Code:

```
#5b(b) Evaluating feed forward deep network for multiclass Classification using KFold cross-validation.)
from sklearn.datasets import make_classification
from sklearn.model_selection import KFold
from keras.models import Sequential
from keras.layers import Dense
from keras.utils import to_categorical
from sklearn.metrics import accuracy_score

X, y = make_classification(n_samples=100,
                           n_features=20,
                           n_informative=2,
                           n_redundant=0,
                           n_classes=2,
                           n_clusters_per_class=2,
                           random_state=42)
# Convert the target variable to categorical format
y = to_categorical(y)

# Define the k-fold cross-validator
n_splits = 5
kf = KFold(n_splits=n_splits, shuffle=True, random_state=42)

# Define the feed-forward deep network model
model = Sequential()
```

```
model.add(Dense(64, activation='relu', input_shape=(X.shape[1],)))
model.add(Dense(64, activation='relu'))
model.add(Dense(y.shape[1], activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
# Perform k-fold cross-validation
fold_accuracies = []
for train_index, val_index in kf.split(X):
    X_train, X_val = X[train_index], X[val_index]
    y_train, y_val = y[train_index], y[val_index]
    model.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_val, y_val))
    y_pred_prob = model.predict(X_val)
    y_pred = y_pred_prob.argmax(axis=1)  # Get the predicted class index with highest probability
    accuracy = accuracy_score(y_val.argmax(axis=1), y_pred)
    fold_accuracies.append(accuracy)

# Calculate the mean accuracy across all folds
mean_accuracy = sum(fold_accuracies) / len(fold_accuracies)
print(f'Mean accuracy: {mean_accuracy:.2f}')
```

Output:



Learning:

This practical illustrates how to build and evaluate a Feedforward Deep Network with KFold Cross Validation.After installing the required libraries, the housing data is loaded and prepared. Model architecture is characterised by a larger network layout. KerasRegressor from scikeras.wrappers combines the Keras model and scikit-learn's cross-validation framework. A pipeline is built to include data standardisation and the specified Keras model. KFold Cross Validation with Ten Folds evaluates the model's prediction ability. The mean squared error (MSE) findings indicate the model's accuracy. This method provides a reliable evaluation of the model's generalisation ability, which aids in model selection and hyperparameter tweaking.

Practical 6

Aim : -

a) Implementing regularization to avoid overfitting in binary classification.

Description:

Regularization:-

Regularization is a critical technique in machine learning used to prevent overfitting by penalizing overly complex models. It involves adding a regularization term to the loss function during training, which discourages large weights or overly complex model architectures. Common regularization techniques include L1 regularization, which adds the absolute values of the weights to the loss function, and L2 regularization, which adds the squared magnitudes of the weights.By incorporating regularization, models are encouraged to learn simpler representations of the data, leading to improved generalization performance on unseen data and mitigating the risk of overfitting to the training set.

Overfitting:-

Overfitting occurs when a machine learning model learns to capture noise and idiosyncrasies in the training data, rather than generalizing well to unseen data. This results in a model that performs exceptionally well on the training set but fails to generalize to new, unseen examples. Overfitting often happens when the model is overly complex relative to the amount of training data available, leading to the memorization of specific data points rather than learning underlying patterns. Regularization techniques are commonly employed to mitigate overfitting.

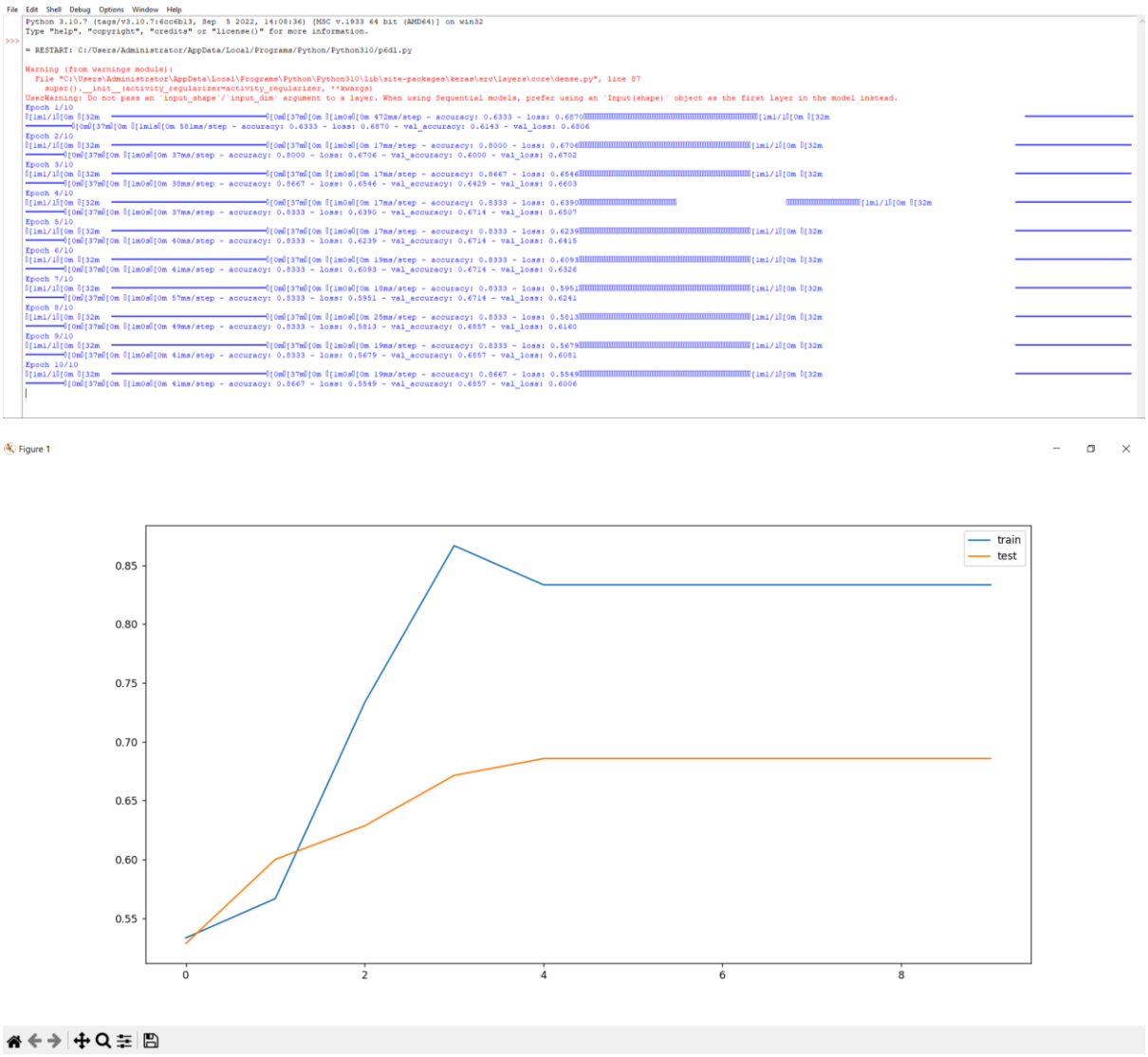
Binary classification:-

Binary classification is a machine learning task where the objective is to classify data into one of two categories. It involves training a model on labeled examples to learn patterns that distinguish between the two classes. Examples include determining whether an email is spam or not, predicting whether a patient has a particular disease, or classifying customer churn. Binary classifiers typically output probabilities or discrete predictions indicating the likelihood of belonging to each class. Evaluation metrics such as accuracy, precision, recall, and F1-score assess the model's performance.

Code:

```
from matplotlib import pyplot
from sklearn.datasets import make_moons
from keras.models import Sequential
from keras.layers import Dense
X,Y=make_moons(n_samples=100,noise=0.2,random_state=1)
n_train=30
trainX,testX=X[:n_train,:],X[n_train:]
trainY,testY=Y[:n_train],Y[n_train:]
#print(trainX)
#print(trainY)
#print(testX)
#print(testY)
model=Sequential()
model.add(Dense(500,input_dim=2,activation='relu'))
model.add(Dense(1,activation='sigmoid'))
model.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accuracy'])
history=model.fit(trainX,trainY,validation_data=(testX,testY),epochs=10)
pyplot.plot(history.history['accuracy'],label='train')
pyplot.plot(history.history['val_accuracy'],label='test')
pyplot.legend()
pyplot.show()
```

Output:



b) Implement L2 regularization with Alpha=0.001

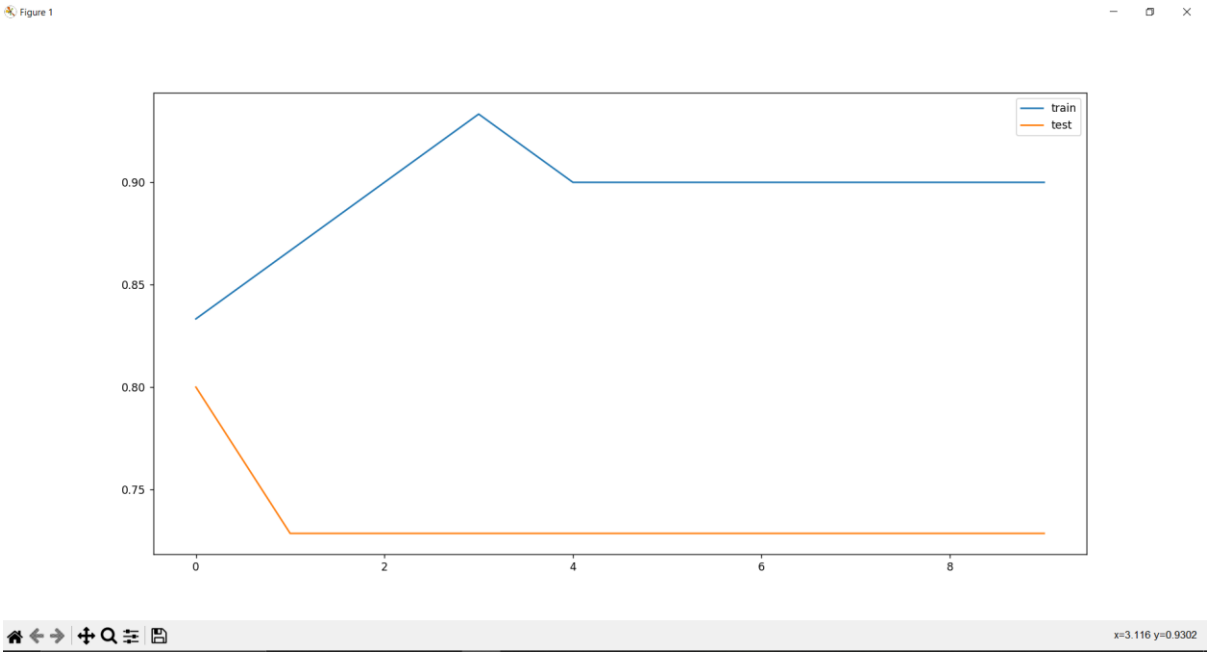
Code:

```
from matplotlib import pyplot
from sklearn.datasets import make_moons
from keras.models import Sequential
from keras.layers import Dense
from keras.regularizers import l2
X,Y=make_moons(n_samples=100,noise=0.2,random_state=1)
```

```
n_train=30
trainX,testX=X[:n_train,:],X[n_train:]
trainY,testY=Y[:n_train],Y[n_train:]
#print(trainX)
#print(trainY)
#print(testX)
#print(testY)
model=Sequential()
model.add(Dense(500,input_dim=2,activation='relu',kernel_regularizer=l2(0.001)))
model.add(Dense(1,activation='sigmoid'))
model.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accuracy'])
history=model.fit(trainX,trainY,validation_data=(testX,testY),epochs=10)
pyplot.plot(history.history['accuracy'],label='train')
pyplot.plot(history.history['val_accuracy'],label='test')
pyplot.legend()
pyplot.show()
```

Output:

```
========== RESTART: C:/Users/Administrator/AppData/Local/Programs/Python/Python310/p62d1.py ==========
|
Warning (from warnings module):
  File "C:\Users\Administrator\AppData\Local\Programs\Python\Python310\lib\site-packages\keras\src\layers\core\dense.py", line 87
    super().__init__(activity_regularizer=activity_regularizer, **kwargs)
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
Epoch 1/10
1/1 [==============================] - 0s 507ms/step - accuracy: 0.4333 - loss: 0.7047
1/1 [==============================] - 0s 642ms/step - accuracy: 0.4333 - loss: 0.7047 - val_accuracy: 0.7571 - val_loss: 0.6893
Epoch 2/10
1/1 [==============================] - 0s 18ms/step - accuracy: 0.6333 - loss: 0.6890
1/1 [==============================] - 0s 38ms/step - accuracy: 0.6333 - loss: 0.6890 - val_accuracy: 0.8143 - val_loss: 0.6791
Epoch 3/10
1/1 [==============================] - 0s 17ms/step - accuracy: 0.8667 - loss: 0.6736
1/1 [==============================] - 0s 38ms/step - accuracy: 0.8667 - loss: 0.6736 - val_accuracy: 0.7429 - val_loss: 0.6692
Epoch 4/10
1/1 [==============================] - 0s 17ms/step - accuracy: 0.9000 - loss: 0.6587
1/1 [==============================] - 0s 39ms/step - accuracy: 0.9000 - loss: 0.6587 - val_accuracy: 0.7286 - val_loss: 0.6596
Epoch 5/10
1/1 [==============================] - 0s 18ms/step - accuracy: 0.9333 - loss: 0.6441
1/1 [==============================] - 0s 38ms/step - accuracy: 0.9333 - loss: 0.6441 - val_accuracy: 0.7286 - val_loss: 0.6503
Epoch 6/10
1/1 [==============================] - 0s 20ms/step - accuracy: 0.9333 - loss: 0.6300
1/1 [==============================] - 0s 41ms/step - accuracy: 0.9333 - loss: 0.6300 - val_accuracy: 0.7286 - val_loss: 0.6413
Epoch 7/10
1/1 [==============================] - 0s 21ms/step - accuracy: 0.9000 - loss: 0.6161
1/1 [==============================] - 0s 41ms/step - accuracy: 0.9000 - loss: 0.6161 - val_accuracy: 0.7286 - val_loss: 0.6326
Epoch 8/10
1/1 [==============================] - 0s 20ms/step - accuracy: 0.9000 - loss: 0.6027
1/1 [==============================] - 0s 41ms/step - accuracy: 0.9000 - loss: 0.6027 - val_accuracy: 0.7286 - val_loss: 0.6242
Epoch 9/10
1/1 [==============================] - 0s 24ms/step - accuracy: 0.9000 - loss: 0.5895
1/1 [==============================] - 0s 44ms/step - accuracy: 0.9000 - loss: 0.5895 - val_accuracy: 0.7286 - val_loss: 0.6160
Epoch 10/10
1/1 [==============================] - 0s 21ms/step - accuracy: 0.9000 - loss: 0.5767
1/1 [==============================] - 0s 41ms/step - accuracy: 0.9000 - loss: 0.5767 - val_accuracy: 0.7286 - val_loss: 0.6081
```



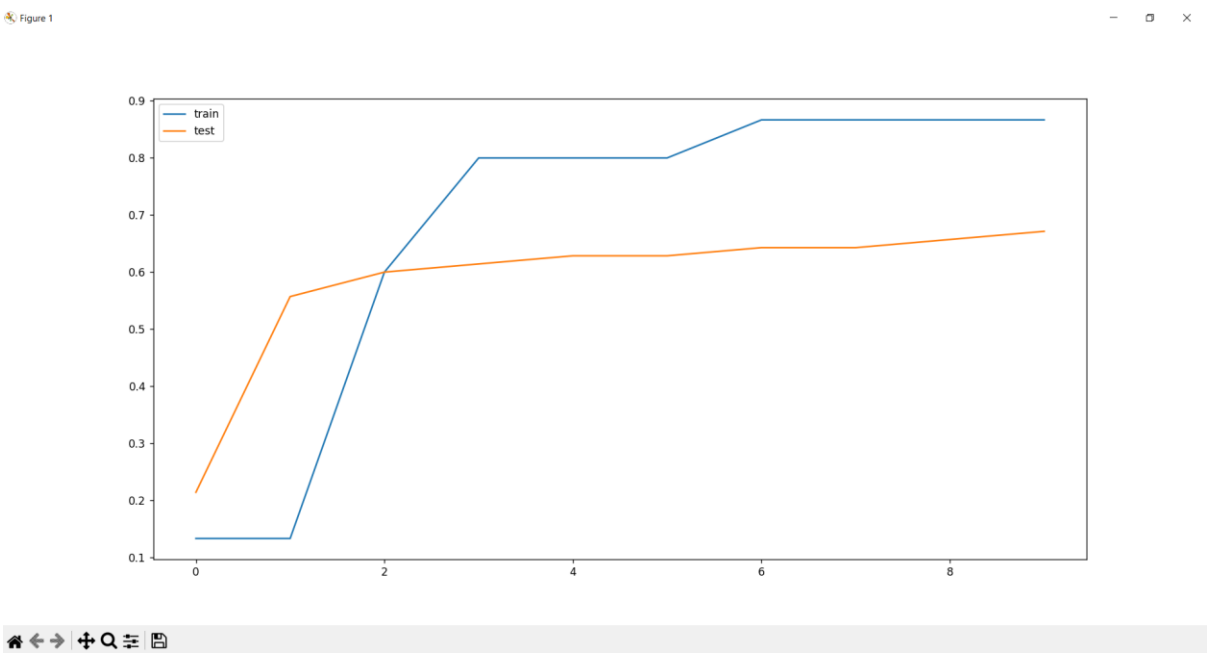
c) Replace L2 regularization with L1 regularization

Code:

```
from matplotlib import pyplot
from sklearn.datasets import make_moons
from keras.models import Sequential
from keras.layers import Dense
from keras.regularizers import l1_l2
X,Y=make_moons(n_samples=100,noise=0.2,random_state=1)
n_train=30
trainX,testX=X[:n_train,:],X[n_train:]
trainY,testY=Y[:n_train],Y[n_train:]
#print(trainX)
#print(trainY)
#print(testX)
#print(testY)
model=Sequential()
model.add(Dense(500,input_dim=2,activation='relu',kernel_regularizer=l1_l2(l1=0.001,l2=0.001
)))
model.add(Dense(1,activation='sigmoid'))
model.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accuracy'])
history=model.fit(trainX,trainY,validation_data=(testX,testY),epochs=10)
pyplot.plot(history.history['accuracy'],label='train')
pyplot.plot(history.history['val_accuracy'],label='test')
pyplot.legend()
pyplot.show()
```

Output:

```
Warning (from warnings module):
  File "C:\Users\karan\AppData\Roaming\Python\Python311\site-packages\keras\src\layers\core\dense.py", line 87
    super().__init__(activity_regularizer=activity_regularizer, **kwargs)
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
Epoch 1/10
[1m1/1[0m [32m━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━[0m[37m[0m [1m0s[0m [1m1s/step - accuracy: 0.1333 - loss: 0.7833
[1m1/1[0m [32m━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━[0m[37m[0m [1m1s[0m [1m1s/step - accuracy: 0.1333 - loss: 0.7833 - val_accuracy: 0.2143 - val_loss: 0.7659
Epoch 2/10
[1m1/1[0m [32m━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━[0m[37m[0m [1m0s[0m [1m31ms/step - accuracy: 0.1333 - loss: 0.7663
[1m1/1[0m [32m━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━[0m[37m[0m [1m0s[0m [1m78ms/step - accuracy: 0.1333 - loss: 0.7663 - val_accuracy: 0.5571 - val_loss: 0.7550
Epoch 3/10
[1m1/1[0m [32m━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━[0m[37m[0m [1m0s[0m [1m31ms/step - accuracy: 0.6000 - loss: 0.7498
[1m1/1[0m [32m━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━[0m[37m[0m [1m0s[0m [1m78ms/step - accuracy: 0.6000 - loss: 0.7498 - val_accuracy: 0.6000 - val_loss: 0.7445
Epoch 4/10
[1m1/1[0m [32m━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━[0m[37m[0m [1m0s[0m [1m31ms/step - accuracy: 0.8000 - loss: 0.7337
[1m1/1[0m [32m━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━[0m[37m[0m [1m0s[0m [1m78ms/step - accuracy: 0.8000 - loss: 0.7337 - val_accuracy: 0.6143 - val_loss: 0.7345
Epoch 5/10
[1m1/1[0m [32m━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━[0m[37m[0m [1m0s[0m [1m31ms/step - accuracy: 0.8000 - loss: 0.7181
[1m1/1[0m [32m━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━[0m[37m[0m [1m0s[0m [1m94ms/step - accuracy: 0.8000 - loss: 0.7181 - val_accuracy: 0.6286 - val_loss: 0.7248
Epoch 6/10
[1m1/1[0m [32m━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━[0m[37m[0m [1m0s[0m [1m47ms/step - accuracy: 0.8000 - loss: 0.7029
[1m1/1[0m [32m━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━[0m[37m[0m [1m0s[0m [1m109ms/step - accuracy: 0.8000 - loss: 0.7029 - val_accuracy: 0.6286 - val_loss: 0.7155
Epoch 7/10
[1m1/1[0m [32m━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━[0m[37m[0m [1m0s[0m [1m47ms/step - accuracy: 0.8667 - loss: 0.6882
[1m1/1[0m [32m━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━[0m[37m[0m [1m0s[0m [1m109ms/step - accuracy: 0.8667 - loss: 0.6882 - val_accuracy: 0.6429 - val_loss: 0.7066
Epoch 8/10
[1m1/1[0m [32m━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━[0m[37m[0m [1m0s[0m [1m47ms/step - accuracy: 0.8667 - loss: 0.6740
[1m1/1[0m [32m━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━[0m[37m[0m [1m0s[0m [1m109ms/step - accuracy: 0.8667 - loss: 0.6740 - val_accuracy: 0.6429 - val_loss: 0.6981
Epoch 9/10
[1m1/1[0m [32m━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━[0m[37m[0m [1m0s[0m [1m47ms/step - accuracy: 0.8667 - loss: 0.6601
[1m1/1[0m [32m━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━[0m[37m[0m [1m0s[0m [1m109ms/step - accuracy: 0.8667 - loss: 0.6601 - val_accuracy: 0.6571 - val_loss: 0.6898
Epoch 10/10
[1m1/1[0m [32m━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━[0m[37m[0m [1m0s[0m [1m47ms/step - accuracy: 0.8667 - loss: 0.6467
[1m1/1[0m [32m━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━[0m[37m[0m [1m0s[0m [1m94ms/step - accuracy: 0.8667 - loss: 0.6467 - val_accuracy: 0.6714 - val_loss: 0.6820
>>>
```



Learning:

In this practical, we used regularisation approaches to reduce overfitting in binary classification with a neural network trained on the make_moons dataset. I first used L1 regularisation with alpha=0.001 before switching to L2 regularisation. I experimented with regularisation to see how it affected the model's training and validation accuracy across 4000 epochs. Regularisation reduced overfitting by penalising big weights in the model. Furthermore, I discovered that L2 regularisation produces smoother weight updates than L1 regularisation, which leads to better generalisation performance on unseen data. Regularisation methods are critical for increasing model resilience in binary classification applications.

Practical 7

Aim : - Demonstrate recurrent neural network that learns to perform sequence analysis for stock.

Description:

Recurrent Neural Network

A Recurrent Neural Network (RNN) is a type of artificial neural network designed to handle sequential data by maintaining a memory of past inputs. Unlike feedforward networks, RNNs have connections that loop back on themselves, allowing them to exhibit temporal dynamics. This architecture enables RNNs to capture patterns and dependencies within sequential data, making them well-suited for tasks such as speech recognition, natural language processing, time series analysis, and more. RNNs process input data step-by-step, updating their internal state with each new input, thus retaining information from previous steps to influence future predictions or classifications.

Functions:

pd.read_csv(): This function from the pandas library is used to read data from a CSV file into a DataFrame, which is a two-dimensional labeled data structure.

MinMaxScaler(): This function from the scikit-learn library is used for feature scaling. It scales the input data to a specified range, which is typically between 0 and 1.

Sequential(): This function from Keras initializes a linear stack of layers. It's a simple way to build a neural network model layer by layer.

LSTM(): This function from Keras adds a Long Short-Term Memory (LSTM) layer to the neural network model. LSTM is a type of recurrent neural network (RNN) architecture, particularly well-suited for sequence prediction problems.

Dropout(): This function from Keras adds a dropout layer to the neural network model. Dropout is a regularization technique used to prevent overfitting by randomly setting a fraction of input units to zero during training.

compile(): This method is used to compile the neural network model. It specifies the optimizer and the loss function to be used during training.

fit(): This method trains the neural network model on the training data. It takes input features (X_train) and target values (Y_train) and specifies the number of epochs (iterations over the entire training dataset) and the batch size (number of samples per gradient update).

predict(): This method predicts the output values based on the input features (X_test). In the provided code, it's used to predict the stock prices using the trained LSTM model.

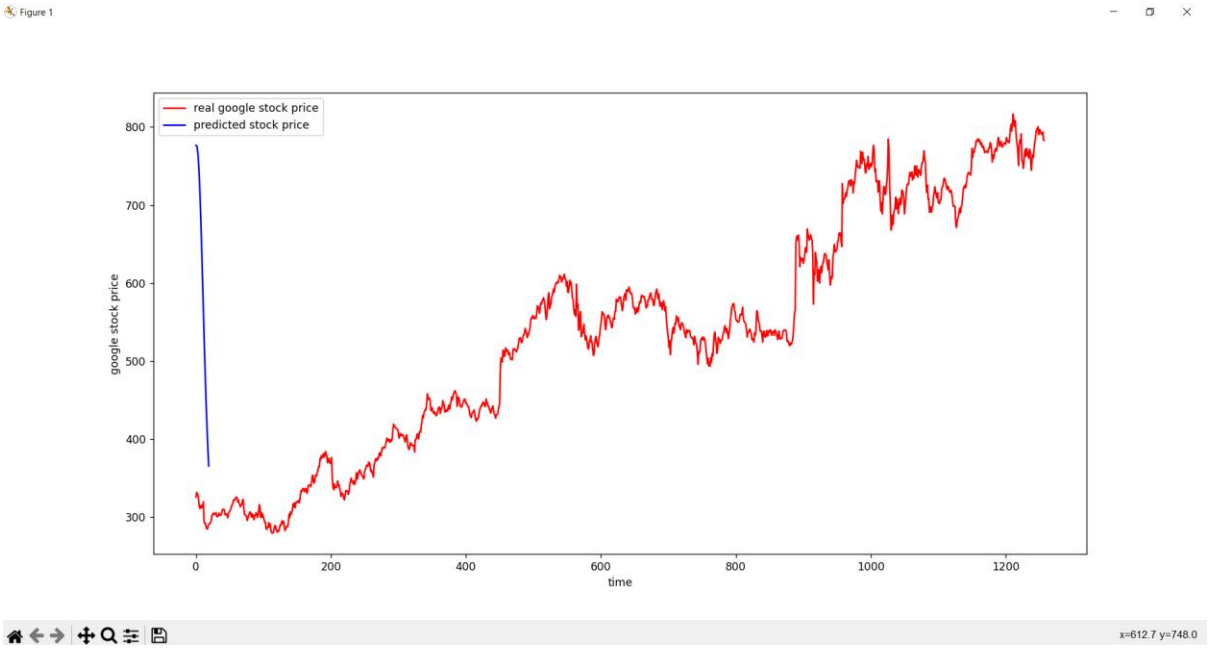
Code:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Dropout
from sklearn.preprocessing import MinMaxScaler

dataset_train=pd.read_csv(r'Google_Stock_price_Train.csv')
#print(dataset_train)
training_set=dataset_train.iloc[:,1:2].values
#print(training_set)
sc=MinMaxScaler(feature_range=(0,1))
training_set_scaled=sc.fit_transform(training_set)
#print(training_set_scaled)
X_train=[]
Y_train=[]
for i in range(60,1258):
    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)
print(X_train)
print('*******************************************')
print(Y_train)
X_train=np.reshape(X_train,(X_train.shape[0],X_train.shape[1],1))
print('*******************************************')
print(X_train)
```

```
regressor=Sequential()
regressor.add(LSTM(units=50,return_sequences=True,input_shape=(X_train.shape[1],1)))
regressor.add(Dropout(0.2))
regressor.add(LSTM(units=50,return_sequences=True))
regressor.add(Dropout(0.2))
regressor.add(LSTM(units=50,return_sequences=True))
regressor.add(Dropout(0.2))
regressor.add(LSTM(units=50))
regressor.add(Dropout(0.2))
regressor.add(Dense(units=1))
regressor.compile(optimizer='adam',loss='mean_squared_error')
regressor.fit(X_train,Y_train,epochs=10,batch_size=32)
dataset_test=pd.read_csv(r'Google_Stock_price_Train.csv')
real_stock_price=dataset_test.iloc[:,1:2].values
dataset_total=pd.concat((dataset_train['Open'],dataset_test['Open']),axis=0)
inputs=dataset_total[len(dataset_total)-len(dataset_test)-60:].values
inputs=inputs.reshape(-1,1)
inputs=sc.transform(inputs)
X_test=[]
for i in range(60,80):
    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)
plt.plot(real_stock_price,color='red',label='real google stock price')
plt.plot(predicted_stock_price,color='blue',label='predicted stock price')
plt.xlabel('time')
```

```
plt.ylabel('google stock price')
plt.legend()
plt.show()
```

Output:



Learning:

This practical demonstrates the use of LSTM neural networks to anticipate Google stock values. It starts with data preparation, which involves scaling the training dataset with MinMaxScaler for consistency. Sequential data management is used to organise the dataset into input sequences and outputs. To prevent overfitting, the LSTM model design uses many layers with dropout regularisation. Following compilation using the Adam optimizer and the mean squared error loss function, the model is trained on the training dataset. It then forecasts future stock prices, which are inversely scaled for comparison to actual prices. Visualisation with matplotlib allows for the evaluation of model performance, demonstrating LSTM's usefulness in time series prediction.

Practical 8

Aim : - Performing encoding and decoding of images using deep autoencoder

Description:

Autoencoder architecture

The autoencoder architecture consists of an encoder and a decoder neural network. The encoder compresses the input data into a lower-dimensional representation, while the decoder reconstructs the original data from this compressed representation.By learning an efficient data representation, the autoencoder can effectively remove noise from corrupted input while retaining essential features.This architecture is widely used in tasks like image denoising, dimensionality reduction, and unsupervised feature learning.

Training Parameters:

Adjusting training parameters such as the number of epochs, batch size, and validation data is crucial for optimizing model performance. The number of epochs determines the number of times the entire training dataset is passed through the model, while batch size affects the efficiency of parameter updates. Validation data helps monitor model generalization and prevent overfitting.

Evaluation Metrics:

Selecting relevant evaluation metrics, such as reconstruction error or accuracy, guides model assessment and performance evaluation. These metrics quantify the model's ability to accurately reconstruct input data and provide insights into its effectiveness for the intended task.

Applications of performing encoding and decoding of images using deep autoencoders include:

1. Image Compression: Efficiently encode images for storage or transmission.

2. Denoising: Remove noise from corrupted images for improved quality.

3. Feature Extraction: Extract meaningful features for downstream tasks like classification.

4. Anomaly Detection: Detect outliers or anomalies in image data.

5. Image Generation: Generate new images by decoding latent representations.

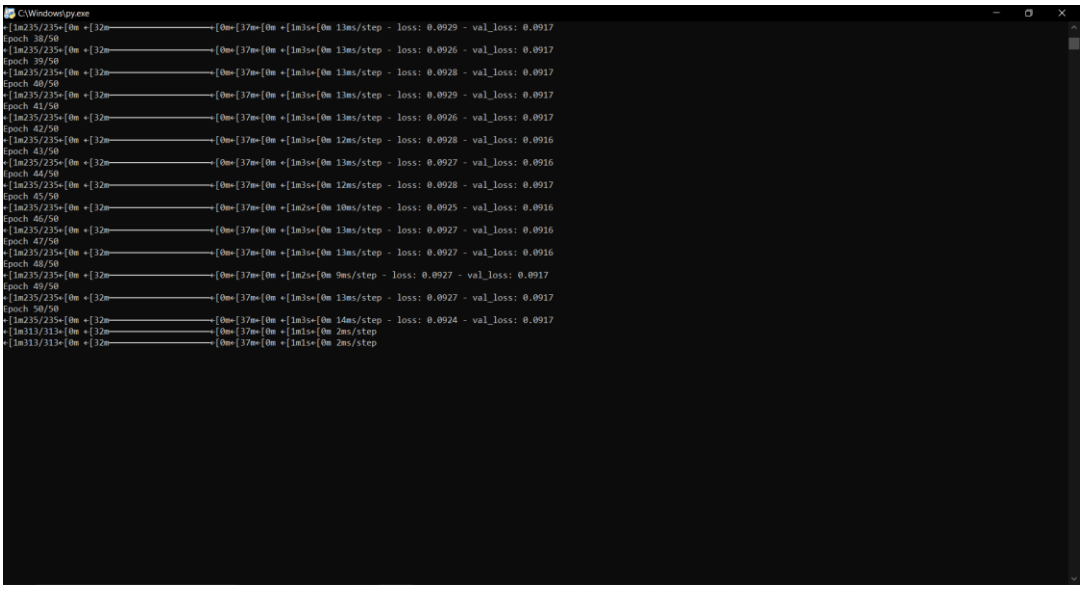
6. Dimensionality Reduction: Reduce the dimensionality of image data while preserving essential features.

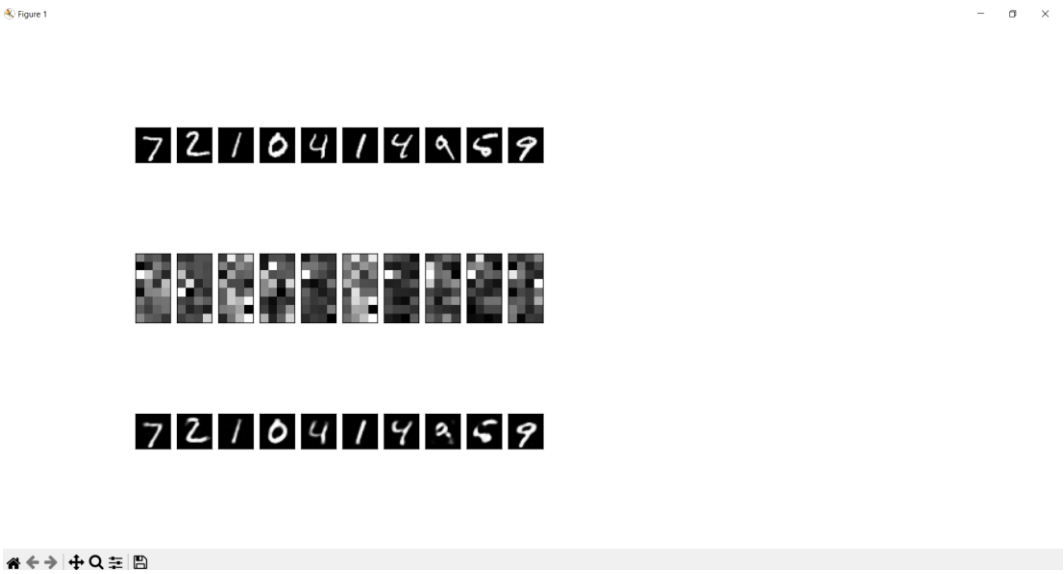
Code:

```
#8(Aim: Performing encoding and decoding of images using deep autoencoder.)
import keras
from keras import layers
from keras.datasets import mnist
import numpy as np
encoding_dim=32
#this is our input image
input_img=keras.Input(shape=(784,))
#"encoded" is the encoded representation of the input
encoded=layers.Dense(encoding_dim, activation='relu')(input_img)
#"decoded" is the lossy reconstruction of the input
decoded=layers.Dense(784, activation='sigmoid')(encoded)
#creating autoencoder model
autoencoder=keras.Model(input_img,decoded)
#create the encoder model
encoder=keras.Model(input_img,encoded)
encoded_input=keras.Input(shape=(encoding_dim,))
#Retrive the last layer of the autoencoder model
decoder_layer=autoencoder.layers[-1]
#create the decoder model
decoder=keras.Model(encoded_input,decoder_layer(encoded_input))
autoencoder.compile(optimizer='adam',loss='binary_crossentropy')
#scale and make train and test dataset
(X_train,_),(X_test,_)=mnist.load_data()
X_train=X_train.astype('float32')/255.
X_test=X_test.astype('float32')/255.
X_train=X_train.reshape((len(X_train),np.prod(X_train.shape[1:])))
X_test=X_test.reshape((len(X_test),np.prod(X_test.shape[1:])))
print(X_train.shape)
print(X_test.shape)
#train autoencoder with training dataset
autoencoder.fit(X_train,X_train,
 epochs=50,
 batch_size=256,
 shuffle=True,
 validation_data=(X_test,X_test))
encoded_imgs=encoder.predict(X_test)
decoded_imgs=decoder.predict(encoded_imgs)
import matplotlib.pyplot as plt
n = 10 # How many digits we will display
plt.figure(figsize=(40, 4))
for i in range(10):
```

```
 # display original
 ax = plt.subplot(3, 20, i + 1)
 plt.imshow(X_test[i].reshape(28, 28))
 plt.gray()
 ax.get_xaxis().set_visible(False)
 ax.get_yaxis().set_visible(False)
 # display encoded image
 ax = plt.subplot(3, 20, i + 1 + 20)
 plt.imshow(encoded_imgs[i].reshape(8,4))
 plt.gray()
 ax.get_xaxis().set_visible(False)
 ax.get_yaxis().set_visible(False)
 # display reconstruction
 ax = plt.subplot(3, 20, 2*20 +i+ 1)
 plt.imshow(decoded_imgs[i].reshape(28, 28))
 plt.gray()
 ax.get_xaxis().set_visible(False)
 ax.get_yaxis().set_visible(False)
plt.show()
```

Output:





Learning :

Through this practical, we learned the fundamentals of implementing autoencoders with the Keras framework. Understanding the design of the autoencoder, namely the encoding and decoding levels, was a key lesson. Preprocessing activities such as scaling and reshaping data were required to ensure model compliance.Compiling the model entailed defining optimizer and loss function parameters that are critical for training. Understanding the training approach, which included epochs, batch size, and validation data, was critical for improving model performance. Evaluating the training model by recreating test pictures and visualising both encoded representations and rebuilt images revealed information about the model's efficacy.

Practical 9

Aim : - Implementation of convolutional neural network to predict numbers from number images.

Description:

Convolutional Neural Network (CNN) Architecture

The convolutional neural network (CNN) architecture comprises convolutional layers followed by pooling layers to extract hierarchical features from input images.Additional convolutional layers deepen feature extraction, while pooling layers reduce spatial dimensions. Fully connected layers at the end of the network perform classification based on the learned features. CNNs excel in image recognition tasks due to their ability to automatically learn spatial hierarchies of features directly from pixel data.

Hyperparameter Tuning:

Tuning hyperparameters such as learning rate, batch size, dropout rate, and optimizer choice (e.g., Adam, SGD) affects the model's convergence speed and performance. Experimentation with different hyperparameter values is necessary to find the optimal configuration for the task.

Regularization:

Regularization methods such as dropout and L2 regularization mitigate overfitting by discouraging the network from relying too heavily on particular features during training. This ensures a more generalized model, capable of making accurate predictions on unseen data. Dropout randomly deactivates neurons during training, while L2 regularization penalizes large weight magnitudes, collectively aiding in the creation of more robust convolutional neural networks.

Data Augmentation:

Data augmentation, through methods such as rotation, translation, and flipping, enhances training dataset diversity, reducing overfitting and improving the model's capability to handle input image variations.By exposing the model to a wider range of data variations, it becomes more adept at generalizing to unseen examples, resulting in improved performance and robustness.

Transfer Learning:

Transfer learning involves utilizing pre-trained convolutional neural network (CNN) models like VGG or ResNet and fine-tuning them on a specific dataset. This approach accelerates training and enhances performance, particularly beneficial when training data is scarce.

Code:

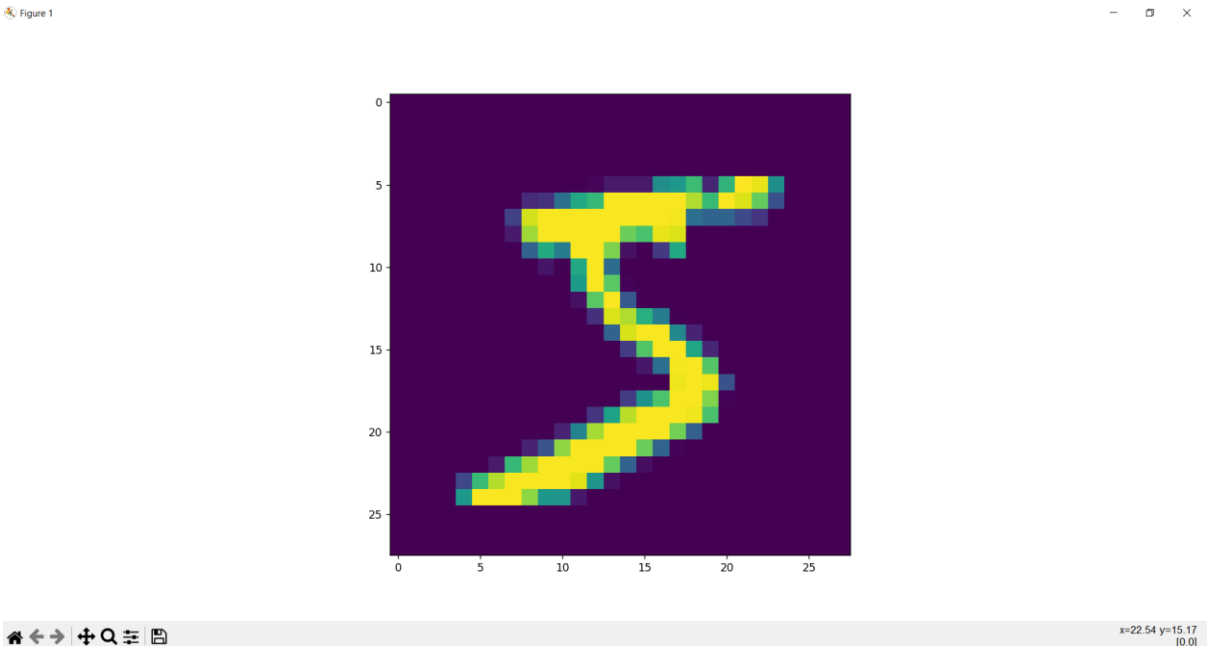
```
from keras.datasets import mnist
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Dense,Conv2D,Flatten
import matplotlib.pyplot as plt
#download mnist data and split into train and test sets
(X_train,Y_train),(X_test,Y_test)=mnist.load_data()
#plot the first image in the dataset
plt.imshow(X_train[0])
plt.show()
print(X_train[0].shape)
X_train=X_train.reshape(60000,28,28,1)
X_test=X_test.reshape(10000,28,28,1)
Y_train=to_categorical(Y_train)
Y_test=to_categorical(Y_test)
Y_train[0]
print(Y_train[0])
model=Sequential()
#add model layers
#learn image features
model.add(Conv2D(64,kernel_size=3,activation='relu',input_shape=(28,28,1)))
model.add(Conv2D(32,kernel_size=3,activation='relu'))
model.add(Flatten())
model.add(Dense(10,activation='softmax'))
model.compile(optimizer='adam',loss='categorical_crossentropy',metrics=['accuracy'])
#train
model.fit(X_train,Y_train,validation_data=(X_test,Y_test),epochs=3)
print(model.predict(X_test[:4]))
```

#actual results for 1st 4 images in the test set

print(Y_test[:4])

Output:

```
C:\Windows\py.exe
2024-05-29 20:09:51.530601: I tensorflow/core/util/port.cc:113] oneDNN custom operations are on. You may see slightly different numerical results due to floating-point round-off errors from different computation orders. To turn them off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.
2024-05-29 20:09:52.788295: I tensorflow/core/util/port.cc:113] oneDNN custom operations are on. You may see slightly different numerical results due to floating-point round-off errors from different computation orders. To turn them off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.
(28, 28)
[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
C:\Users\karan\AppData\Roaming\Python\Python311\site-packages\keras\src\layers\convolutional\base_conv.py:107: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
2024-05-29 20:09:59.798824: I tensorflow/core/platform/cpu_feature_guard.cc:210] This TensorFlow binary is optimized to use available CPU instructions in performance-critical operations.
To enable the following instructions: AVX2 FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags.
Epoch 1/3
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 65s 34ms/step - accuracy: 0.9037 - loss: 1.0679 - val_accuracy: 0.9658 - val_loss: 0.1170
Epoch 2/3
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 67s 36ms/step - accuracy: 0.9755 - loss: 0.0795 - val_accuracy: 0.9714 - val_loss: 0.0947
Epoch 3/3
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 60s 32ms/step - accuracy: 0.9852 - loss: 0.0477 - val_accuracy: 0.9774 - val_loss: 0.0782
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 128ms/step
[[1.04327400e-08 4.61779459e-14 5.88426474e-07 1.80732575e-04
  2.79159681e-14 6.43224623e-12 7.52140266e-13 9.99818742e-01
  2.04562056e-09 6.96279878e-09]
 [4.40743900e-11 8.01574085e-12 1.00000000e+00 5.07062795e-11
  6.20900223e-15 2.43227002e-14 1.04639089e-08 1.21319255e-17
  6.83567716e-11 1.23095923e-16]
 [4.76886862e-06 9.99507308e-01 9.50852427e-06 1.48767640e-07
  1.78192480e-04 1.13609485e-06 4.15147355e-07 1.99554933e-07
  2.97737366e-04 5.02836826e-07]
 [9.99987006e-01 8.30873348e-13 1.50637888e-07 1.40747580e-10
  2.11608693e-07 1.24453683e-08 1.57632530e-06 1.91007245e-11
  5.03883157e-08 1.11104573e-05]]
[[0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
```



Learning:

Through this practical, we gained a thorough grasp of how to create convolutional neural networks (CNNs) for number prediction from pictures.CNN architecture, data preparation, hyperparameter optimisation, and regularisation approaches like as dropout were among the most important lessons learned.Furthermore, we investigated the benefits of transfer learning using pre-trained models like VGG or ResNet.This hands-on experience taught me how to properly construct, train, and fine-tune CNN models, allowing for further study in image categorization and related applications. Overall, the practical offered useful insights into the complexities of deep learning systems and their applicability in image recognition problems.

Practical 10

Aim : - Denoising of images using autoencoder

Description:

Autoencoder architecture

The autoencoder architecture consists of an encoder and a decoder neural network. The encoder compresses the input data into a lower-dimensional representation, while the decoder reconstructs the original data from this compressed representation.By learning an efficient data representation, the autoencoder can effectively remove noise from corrupted input while retaining essential features.This architecture is widely used in tasks like image denoising, dimensionality reduction, and unsupervised feature learning.

Corrupted image

Corrupted image generation involves introducing noise or distortions into clean images to create a dataset for training denoising models like autoencoders.Common methods include adding Gaussian noise, salt-and-pepper noise, blurring, or occluding parts of the image.The level and type of corruption depend on the specific task and desired robustness of the model. Generating diverse corrupted images helps the model learn to remove various types of noise and enhance its ability to generalize to unseen data.

Applications:

Applications of corrupted image generation:

1. Medical Imaging: Enhances denoising for clearer MRI and X-ray scans.
2. Surveillance: Improves image quality for better object detection.
3. Satellite Imaging: Enables noise removal for accurate interpretation.
4. Photography: Enhances photos by removing noise and artifacts.
5. Autonomous Vehicles: Improves perception in varied environmental conditions.
6. Art Restoration: Assists in restoring damaged artworks and documents.

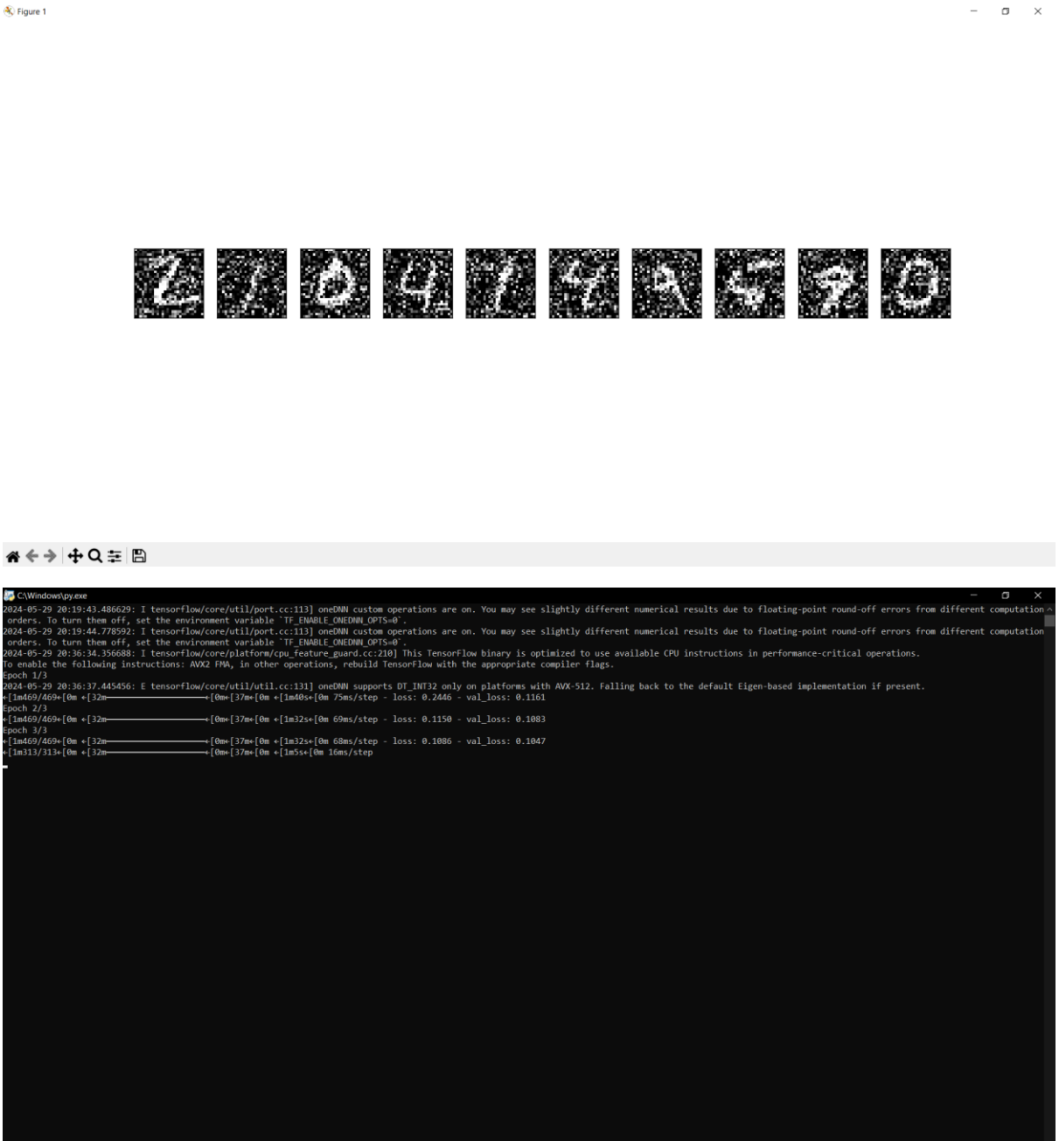
Code:

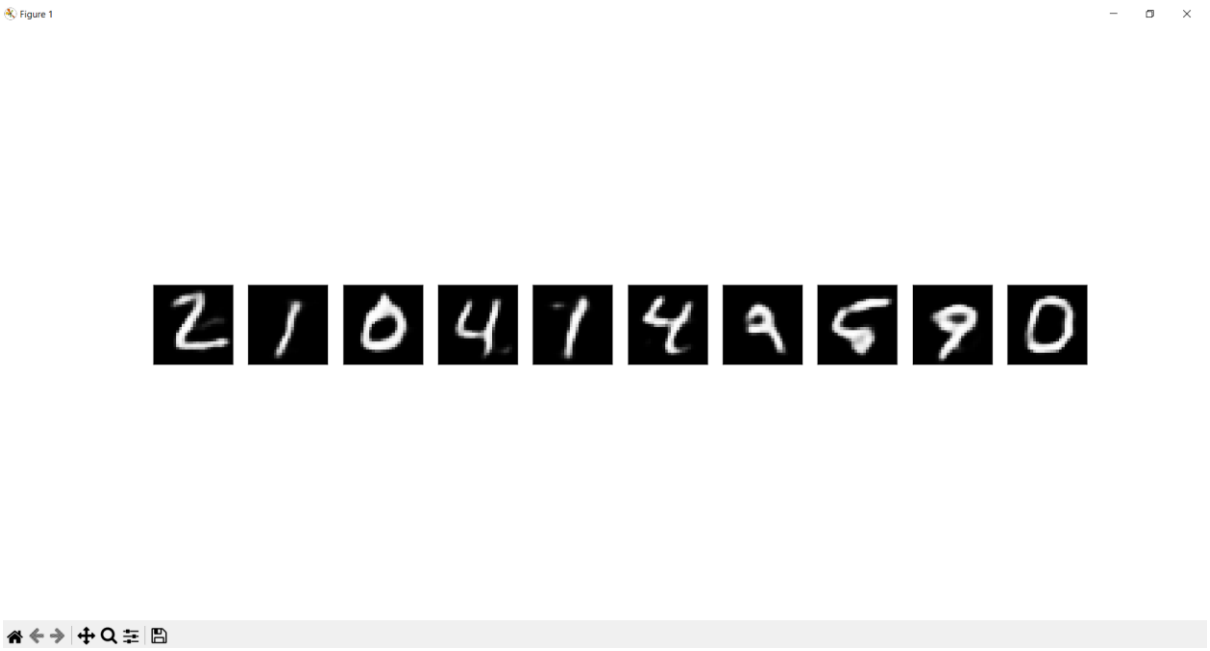
```
from keras.datasets import mnist
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Dense,Conv2D,Flatten
import matplotlib.pyplot as plt
#download mnist data and split into train and test sets
(X_train,Y_train),(X_test,Y_test)=mnist.load_data()
#plot the first image in the dataset
plt.imshow(X_train[0])
plt.show()
print(X_train[0].shape)
X_train=X_train.reshape(60000,28,28,1)
X_test=X_test.reshape(10000,28,28,1)
Y_train=to_categorical(Y_train)
Y_test=to_categorical(Y_test)
Y_train[0]
print(Y_train[0])
model=Sequential()
#add model layers
#learn image features
model.add(Conv2D(64,kernel_size=3,activation='relu',input_shape=(28,28,1)))
model.add(Conv2D(32,kernel_size=3,activation='relu'))
model.add(Flatten())
model.add(Dense(10,activation='softmax'))
model.compile(optimizer='adam',loss='categorical_crossentropy',metrics=['accuracy'])
#train
model.fit(X_train,Y_train,validation_data=(X_test,Y_test),epochs=3)
print(model.predict(X_test[:4]))
```

#actual results for 1st 4 images in the test set

print(Y_test[:4])

Output:





Learning:

This practical emphasises the need of creating a variety of damaged pictures in order to properly train denoising models. We improve model robustness and generalisation by incorporating different types and degrees of noise into the data. Customising corruption types for unique applications promotes relevance and efficacy. We learn how to strike the right balance between data quality and quantity for successful model training. Understanding assessment metrics helps quantify model performance appropriately. Finally, this practical demonstrates the importance of damaged picture production in building effective denoising methods for real-world applications.