INDEX

Introduction to Tensorflow

Practical 1(A)

Aim: Create tensors with different shapes and datatypes

Description: In TensorFlow, you can create tensors with various shapes and data types to accommodate different types of data and computational needs. A tensor is a multi-dimensional array, and its shape determines the number of dimensions and the size of each dimension. You can specify data types such as tf.float32, tf.int32, tf.bool, and more, allowing you to optimize memory usage and performance based on the nature of your data.

To create tensors, you can use functions like tf.constant(), tf.zeros(), tf.ones(), and tf.random.normal(), among others. This flexibility enables efficient data representation for a wide range of applications, from simple numeric computations to complex machine learning tasks. Here’s a brief overview of creating tensors:

- tf.constant: Creates a tensor with a constant value.
- tf.zeros: Generates a tensor filled with zeros, useful for initialization.
- tf.ones: Creates a tensor filled with ones.
- tf.random.normal: Generates a tensor with random values from a normal distribution.

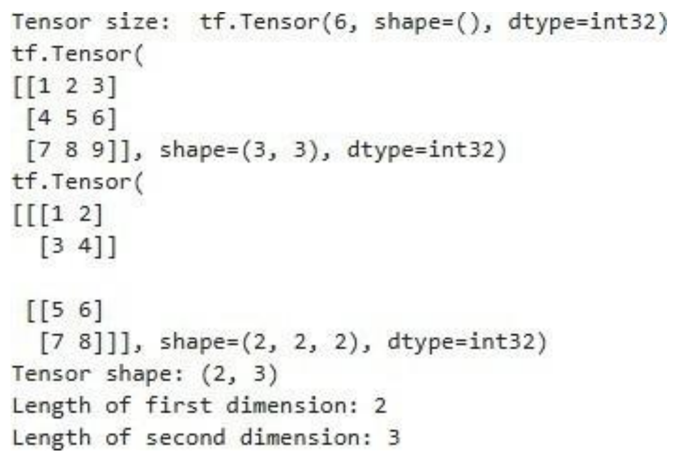
By leveraging these capabilities, you can tailor tensors to fit the specific requirements of your computational tasks.

Code:

```
import tensorflow as tf

tensor = tf.constant([[1, 2, 3], [4, 5, 6]]) # Get the

size of the tensor

size = tf.size(tensor) print("Tensor size: ", size)

# Create a 2D tensor with shape (2,, 3)

tensor_2d = tf.constant([[1, 2, 3], [4, 5, 6], [7, 8, 9]]) print(tensor_2d)

# Create a 3D tensor with shape (2, 2, 2)

tensor_3d = tf.constant([[[1, 2], [3, 4]], [[5, 6], [7, 8]]]) print(tensor_3d)

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

= tensor.shape

# Get the length of the first dimension dim1_length

= shape[0]

# Get the length of the second dimension
```

```
dim2_length = shape[1]
print('Tensor shape:', shape)
print('Length of first dimension:', dim1_length)
print('Length of second dimension:', dim2_length)
```

Output:



```
Tensor size:  tf.Tensor(6, shape=(), dtype=int32)
tf.Tensor(
[[1 2 3]
 [4 5 6]
 [7 8 9]], shape=(3, 3), dtype=int32)
tf.Tensor(
[[[1 2]
  [3 4]]

 [[5 6]
  [7 8]]], shape=(2, 2, 2), dtype=int32)
Tensor shape: (2, 3)
Length of first dimension: 2
Length of second dimension: 3
```

Practical 1(A)-2

Aim: Perform basic operations like addiiton , subtraction , multiplication and division on tensors

Description: In TensorFlow, performing basic arithmetic operations on tensors is straightforward and efficient. You can conduct operations such as addition, subtraction, multiplication, and division using built-in functions and operators. These operations are element-wise, meaning they are applied to corresponding elements in the tensors.

- **Addition**: Use tf.add() or the + operator to sum two tensors.
- **Subtraction**: Use tf.subtract() or the - operator to find the difference between tensors.
- **Multiplication**: Use tf.multiply() or the * operator for element-wise multiplication.
- **Division**: Use tf.divide() or the / operator to perform element-wise division.

These operations can be applied to tensors of compatible shapes, enabling you to easily manipulate and compute results on multidimensional data. TensorFlow optimizes these operations for performance, making it ideal for numerical and machine learning applications. Here’s a brief example:

Code:

```
#Multiplication
import tensorflow as tf
a = tf.constant([2,2,2])
b = tf.constant([1,1,1])
result = tf.multiply(a,b)
print('Multiplication')
print(result)
#Addition
a = tf.constant([2,2,2])
b = tf.constant([1,1,1])
c = tf.add(a,b)
print('Addiiton')
print(c)
#subtraction
a = tf.constant([2,2,2])
b = tf.constant([1,1,1])
c = tf.subtract(a,b)
print('subtraction')
```

print(c)

#division

a = tf.constant([2,2,2])

b = tf.constant([1,1,1])

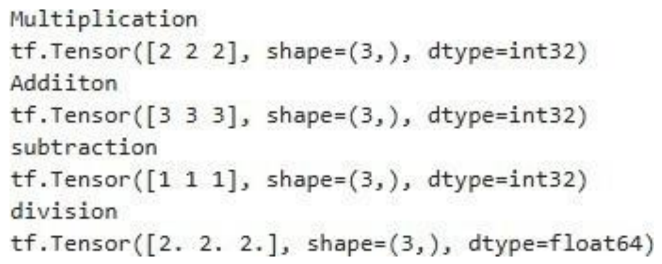
c =

tf.divide(a,b)

print('division')

print(c)

Output:



```
Multiplication
tf.Tensor([2 2 2], shape=(3,), dtype=int32)
Addiiton
tf.Tensor([3 3 3], shape=(3,), dtype=int32)
subtraction
tf.Tensor([1 1 1], shape=(3,), dtype=int32)
division
tf.Tensor([2. 2. 2.], shape=(3,), dtype=float64)
```

Practical 1(A)-3

Aim: Reshape, index and slice tensors to extract specific elements or section

Description: In TensorFlow, reshaping, indexing, and slicing tensors are essential techniques for data manipulation.

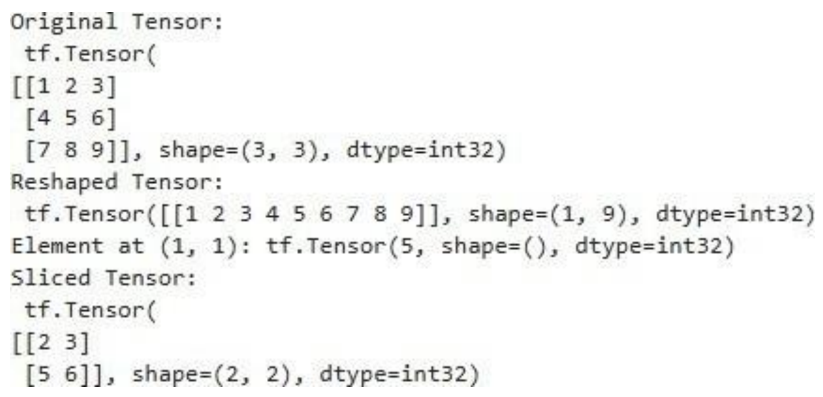
- **Reshaping**: This allows you to change the shape of a tensor without altering its data. Using tf.reshape(), you can convert a tensor into a different dimensional structure, which is useful for adapting data to model requirements.
- **Indexing**: TensorFlow enables you to access specific elements within a tensor using standard indexing techniques. By specifying row and column indices, you can retrieve individual values or sub-tensors.
- **Slicing**: This technique allows you to extract contiguous sections of a tensor. By specifying ranges for rows and columns, you can create smaller tensors from larger ones, facilitating focused analysis or processing.

These operations help you efficiently navigate and manipulate tensor data, making them crucial for tasks in machine learning and data analysis. For example, you can reshape a tensor for input into a neural network, index it to retrieve specific features, or slice it to analyze subsets of your data.

Code:

```
import tensorflow as tf

# Create a 2D tensor

tensor = tf.constant([[1, 2, 3], [4, 5, 6], [7, 8,

9]]) print("Original Tensor:\n", tensor)

# Reshape

reshaped = tf.reshape(tensor, (1, 9))

print("Reshaped Tensor:\n", reshaped)

# Indexing

single_element = tensor[1, 1] # Access element at (1,

1) print("Element at (1, 1):", single_element)

# Slicing

sliced = tensor[0:2, 1:3] # Rows 0-1, Columns 1-

2 print("Sliced Tensor:\n", sliced)
```

Output:



```
Original Tensor:
 tf.Tensor(
[[1 2 3]
 [4 5 6]
 [7 8 9]], shape=(3, 3), dtype=int32)
Reshaped Tensor:
 tf.Tensor([[1 2 3 4 5 6 7 8 9]], shape=(1, 9), dtype=int32)
Element at (1, 1): tf.Tensor(5, shape=(), dtype=int32)
Sliced Tensor:
 tf.Tensor(
[[2 3]
 [5 6]], shape=(2, 2), dtype=int32)
```

Practical 1(A)-4

Aim: Perform matrix multiplication and finding eigen vectors and eigenvalues using tensorflow

Description: In TensorFlow, you can efficiently perform matrix multiplication and compute eigenvalues and eigenvectors, which are fundamental operations in linear algebra.

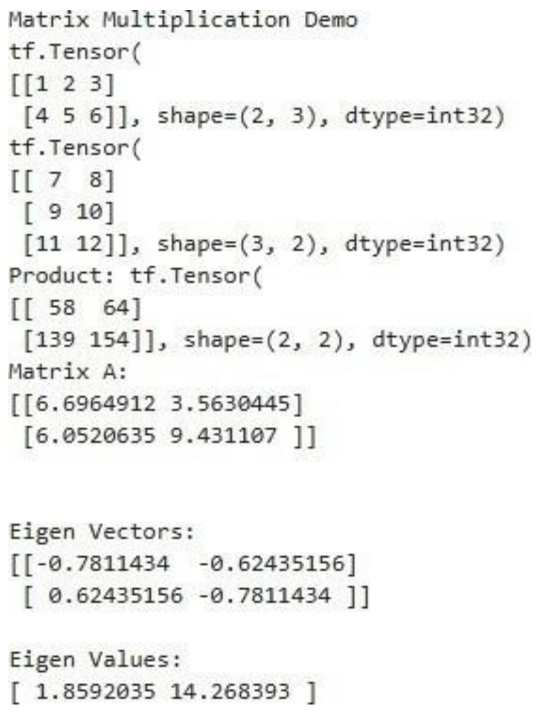
- **Matrix Multiplication:** Use the tf.matmul() function or the @ operator to multiply two matrices. This operation is crucial in various applications, such as neural networks, where it is used to combine inputs and weights.
- **Finding Eigenvalues and Eigenvectors:** TensorFlow provides the tf.linalg.eig() function to compute the eigenvalues and eigenvectors of a square matrix. Eigenvalues indicate the magnitude of transformation, while eigenvectors reveal the direction of that transformation. These concepts are widely used in applications like dimensionality reduction, stability analysis, and systems of differential equations.

These linear algebra operations allow for advanced data manipulation and are integral to many machine learning algorithms and mathematical modeling techniques.

Code:

```
import tensorflow as tf

print("Matrix Multiplication Demo")

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("Product:",z)

e_matrix_A=tf.random.uniform([2,2],minval=3,maxval=10,dtype=tf.float32,name="matrixA")

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

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

print("Eigen Vectors:\n{}\n\nEigen Values:\n{}\n".format(eigen_vectors_A,eigen_values_A))
```

Output:



```
Matrix Multiplication Demo
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)
Product: tf.Tensor(
[[ 58  64]
 [139 154]], shape=(2, 2), dtype=int32)
Matrix A:
[[6.6964912 3.5630445]
 [6.0520635 9.431107 ]]


Eigen Vectors:
[[-0.7811434  -0.62435156]
 [ 0.62435156 -0.7811434 ]]

Eigen Values:
[ 1.8592035 14.268393 ]
```

Practical 1(B)

AIM : Solving XOR problem using deep feed forward network.

Description:

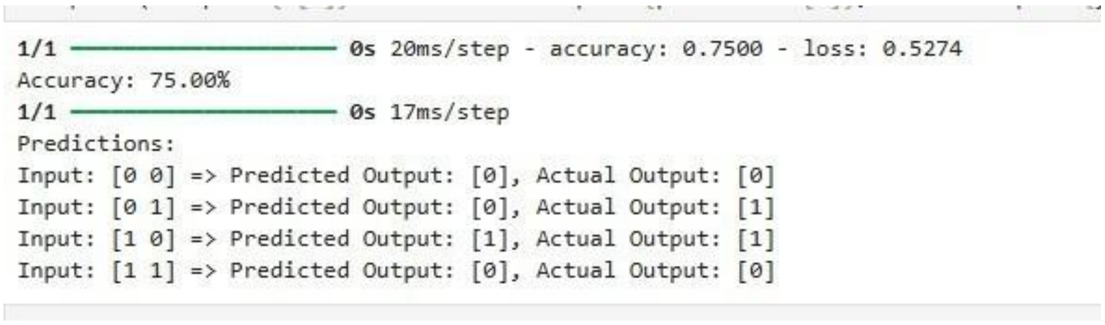
- **Sequential**: Sequential is a Keras class that allows you to create models layer-by-layer. Each layer in a Sequential model has weights that correspond to the layer that follows it.
- **Dense**: Dense is the standard fully connected layer in a neural network. Each neuron in a dense layer receives input from all neurons in the previous layer.
- **model.add():** This function is used to add layers to the neural network model. In the provided code, two Dense layers are added: one with 2 units and ReLU activation, and another with 1 unit and Sigmoid activation.
- **ReLU (Rectified Linear Unit):** Simple activation function that outputs the input if positive, zero otherwise. Efficiently handles vanishing gradient problem and speeds up training.
- **Sigmoid:** Activation function that squashes input to range [0, 1]. Commonly used in binary classification output layers to produce probability-like outputs.
- **model.compile():** This function is used to compile the model. It configures the learning process, specifying the optimizer, loss function, and metrics to be used. In this case, 'adam' optimizer and 'binary_crossentropy' loss function are used.
- **model.fit():** This function trains the model for a fixed number of epochs (iterations on a dataset). It takes input data (x) and corresponding labels (y) and adjusts the model's weights to minimize the specified loss function.
- **np.array():** This function creates a NumPy array, which is a multi-dimensional array used for storing and manipulating data efficiently.
- **model.get_weights():** This function returns the weights of the model's layers.
- **model.predict():** This function generates output predictions for the input samples. In the provided code, it's used to predict the output labels for the input data x.

Code:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Define the XOR input and output data
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([[0], [1], [1], [0]])

# Build the neural network model
model = Sequential()
model.add(Dense(2, input_dim=2, activation='relu'))  # Hidden layer with 2 neurons
model.add(Dense(1, activation='sigmoid'))            # Output layer with 1 neuron

# Compile the model
```

```
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
# Train the model
model.fit(X, y, epochs=10000, verbose=0)

# Evaluate the model
_, accuracy = model.evaluate(X, y)
print(f"Accuracy: {accuracy * 100:.2f}%")

# Make predictions
predictions = model.predict(X)
predictions = np.round(predictions).astype(int)

print("Predictions:")
for i in range(len(X)):

print(f"Input: {X[i]} => Predicted Output: {predictions[i]}, Actual Output: {y[i]}")
```

Output:



```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - accuracy: 0.7500 - loss: 0.5274
Accuracy: 75.00%
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step
Predictions:
Input: [0 0] => Predicted Output: [0], Actual Output: [0]
Input: [0 1] => Predicted Output: [0], Actual Output: [1]
Input: [1 0] => Predicted Output: [1], Actual Output: [1]
Input: [1 1] => Predicted Output: [0], Actual Output: [0]
```

Linear Regression
Practical 2(A)

AIM : Implement a simple linear regression model using TensorFlow's low-level API (or tf.keras)

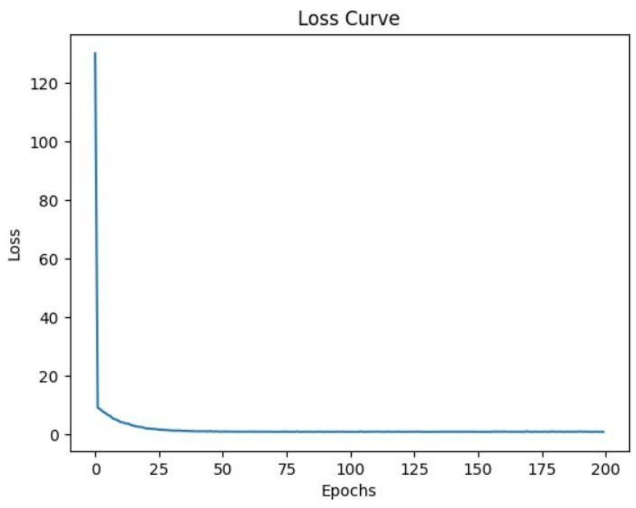
Description:
This implementation trains a **simple linear regression model** using **TensorFlow's Keras API**. The model learns to predict values based on a linear equation **y = 3x + 7** with some added noise. The steps include:

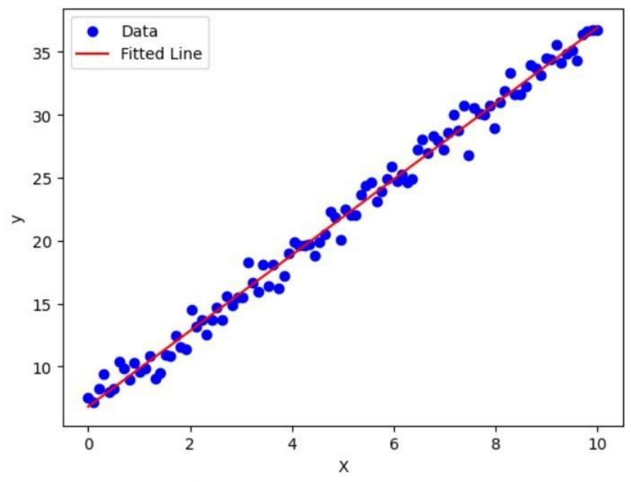
1. **Generating synthetic data** (X, y) based on a known linear relationship.
2. **Building a neural network model** with a single neuron (Dense layer).
3. **Compiling the model** using **SGD optimizer** and **MSE loss function**.
4. **Training the model** for 200 epochs.
5. **Visualizing the results**, including **loss curve** and **fitted regression line**.
6. **Extracting the trained weight (w) and bias (b)**, which approximate **3** and **7**, respectively.

Code:

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

# 1. Generate synthetic data
np.random.seed(42)
X = np.linspace(0, 10, 100) # 100 data points from 0 to 10
y = 3 * X + 7 + np.random.normal(0, 1, 100)  # y = 3x + 7 with some noise

# Convert to TensorFlow tensors
X_train = tf.convert_to_tensor(X, dtype=tf.float32)
y_train = tf.convert_to_tensor(y, dtype=tf.float32)

# 2. Build a simple linear regression model using
tf.keras model = tf.keras.Sequential([
   tf.keras.layers.Dense(1, input_dim=1)  # 1 input feature and 1 output
])

# 3. Compile the model
model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=0.01), # Stochastic Gradient
Descent
           loss='mean_squared_error')  # Loss function for regression

# 4. Train the model
history = model.fit(X_train, y_train, epochs=200, batch_size=10, verbose=0)

# 5. Plot the loss curve (optional)
plt.plot(history.history['loss'])
plt.title('Loss Curve')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.show()

# 6. Make predictions using the trained model
predictions = model.predict(X_train)

# 7. Visualize the original data and the model's predictions
plt.scatter(X, y, label='Data', color='blue')
plt.plot(X, predictions, label='Fitted Line', color='red')
plt.xlabel('X')
plt.ylabel('y')
plt.legend()
plt.show()

# 8. Print model weights
weights, bias = model.layers[0].get_weights()
print(f"Trained Weight (w): {weights[0][0]:.2f}")
print(f"Trained Bias (b): {bias[0]:.2f}")
```

Output:





```
Trained Weight (w): 3.01
Trained Bias (b): 6.83
```

Practical 2(B)

AIM: Train the model on a toy dataset (e.g., housing prices vs square footage).

Description:

This implementation trains a **neural network model** using **TensorFlow's Keras API** to predict **housing prices** based on features like age, distance, stores, latitude, and longitude. The steps include:

1. **Loading and preprocessing the dataset** (normalization and splitting into train/test sets).
2. **Building a deep learning model** with multiple layers using **ReLU activation**.
3. **Compiling and training the model** using the **Adam optimizer** and **MSE loss function**.
4. **Implementing early stopping** to prevent overfitting.
5. **Visualizing training loss vs. validation loss** to evaluate model performance.
6. **Comparing predictions from the trained and untrained models** using scatter plots.

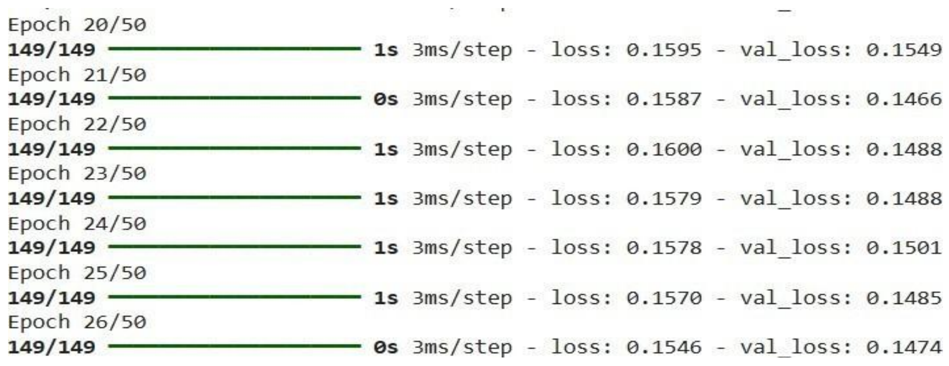
Code:

```
import pandas as pd
import matplotlib.pyplot as plt
import tensorflow as tf

from utils import *
from sklearn.model_selection import
train_test_split from tensorflow.keras.models
import Sequential from tensorflow.keras.layers
import Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping, LambdaCallback

print('Libraries imported.')
column_names = ['serial', 'date', 'age', 'distance', 'stores', 'latitude', 'longitude',
'price'] df = pd.read_csv('data.csv', names = column_names)

df.head()
df.describe()
df.isna().sum()
df = df.iloc[:, 1:]
df_norm = (df -
df.mean())/df.std()
df_norm.head()
y_mean = df['price'].mean()
y_std = df['price'].std()

def convert_label_value (y_pred):
  return
int((y_pred*y_mean)/y_std) x =
df_norm.iloc[: , :5]
x. head()
y = df_norm.iloc[: ,-1]
y.head()
x_arr = x.values
```

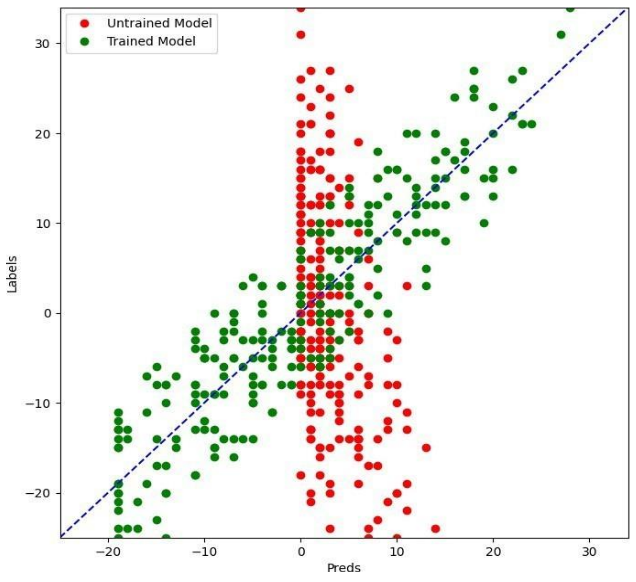
```
y_arr = y.values

print('features array :',x_arr) print('label array : ',y_arr)
x_train, x_test, y_train, y_test = train_test_split(x_arr,y_arr, test_size=0.05, random_state=0)
  print('train shape', x_train.shape, y_train.shape)

  print('test shape', x_test.shape,
  y_test.shape) def get_model ():
     model = Sequential([
        Dense(10, input_shape=(5,), activation = 'relu'),
        Dense(20, activation = 'relu'),
        Dense(5, activation = 'relu'),
        Dense(1)
     ])

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

     return model

  get_model().summary(

  )
  es_cb = EarlyStopping(monitor='val_loss',

  patience=5) model = get_model()

  preds_on_untrained = model.predict(x_test)

  history = model.fit(
     x_train, y_train,
     validation_data = (x_test,
     y_test), epochs = 50,
     callbacks = [es_cb]

  )
  def plot_loss(history):
     h = history.history
     x_lim = len(h['loss'])
     plt.figure(figsize=(8,
     8))
     plt.plot(range(x_lim), h['val_loss'], label = 'Validation
     Loss') plt.plot(range(x_lim), h['loss'], label = 'Training
     Loss') plt.xlabel('Epochs')
     plt.ylabel('Los
     s') plt.legend()
     plt.show()
     return
```

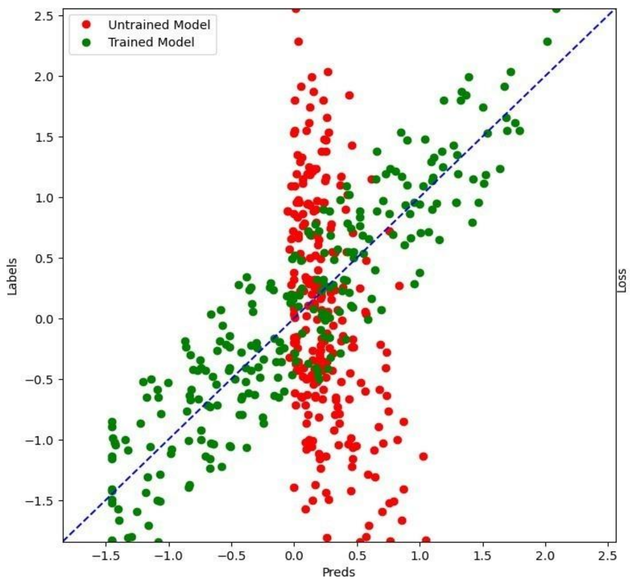
```
    plot_loss(history)

    preds_on_trained = model.predict(x_test)

    def compare_predictions(preds1, preds2, y_test):
    plt.figure(figsize=(8, 8))

    plt.plot(preds1, y_test, 'ro', label='Untrained
    Model') plt.plot(preds2, y_test, 'go',
    label='Trained Model') plt.xlabel('Preds')
    plt.ylabel('Labels')

    y_min = min(min(y_test), min(preds1),
    min(preds2)) y_max = max(max(y_test),
    max(preds1), max(preds2))

    plt.xlim([y_min, y_max])
    plt.ylim([y_min, y_max])
    plt.plot([y_min, y_max], [y_min, y_max], 'b--')
    plt.legend()
    plt.show()
    return

compare_predictions(preds_on_untrained, preds_on_trained, y_test)
price_untrained = [convert_label_value(y) for y in
preds_on_untrained] price_trained = [convert_label_value(y) for y in
preds_on_trained] prince_test = [convert_label_value(y) for y in
y_test]

def compare_predictions(preds1, preds2, y_test):
    plt.figure(figsize=(8, 8))
    plt.plot(preds1, y_test, 'ro', label='Untrained
    Model') plt.plot(preds2, y_test, 'go',
    label='Trained Model') plt.xlabel('Preds')
    plt.ylabel('Labels')

    y_min = min(min(y_test), min(preds1),
    min(preds2)) y_max = max(max(y_test),
    max(preds1), max(preds2))

    plt.xlim([y_min, y_max])
    plt.ylim([y_min, y_max])
    plt.plot([y_min, y_max], [y_min, y_max], 'b--')
    plt.legend()
    plt.show()
    return

compare_predictions(price_untrained, price_trained, prince_test)
```

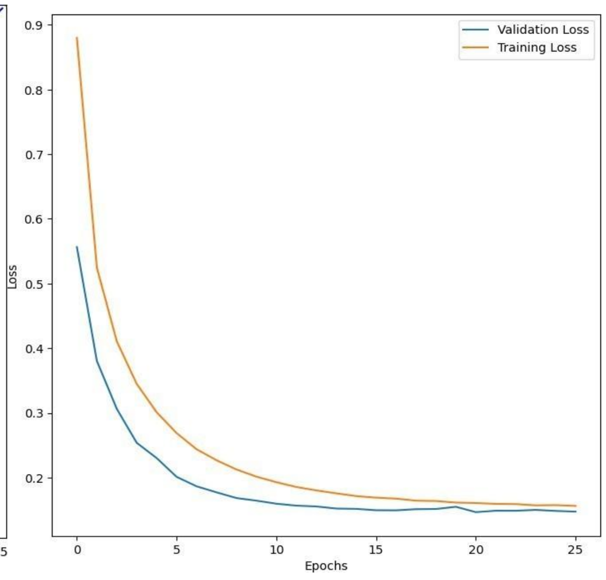
Output:



```
Epoch 20/50
149/149 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.1595 - val_loss: 0.1549
Epoch 21/50
149/149 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.1587 - val_loss: 0.1466
Epoch 22/50
149/149 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.1600 - val_loss: 0.1488
Epoch 23/50
149/149 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.1579 - val_loss: 0.1488
Epoch 24/50
149/149 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.1578 - val_loss: 0.1501
Epoch 25/50
149/149 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.1570 - val_loss: 0.1485
Epoch 26/50
149/149 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.1546 - val_loss: 0.1474
```







Practical 2(C)

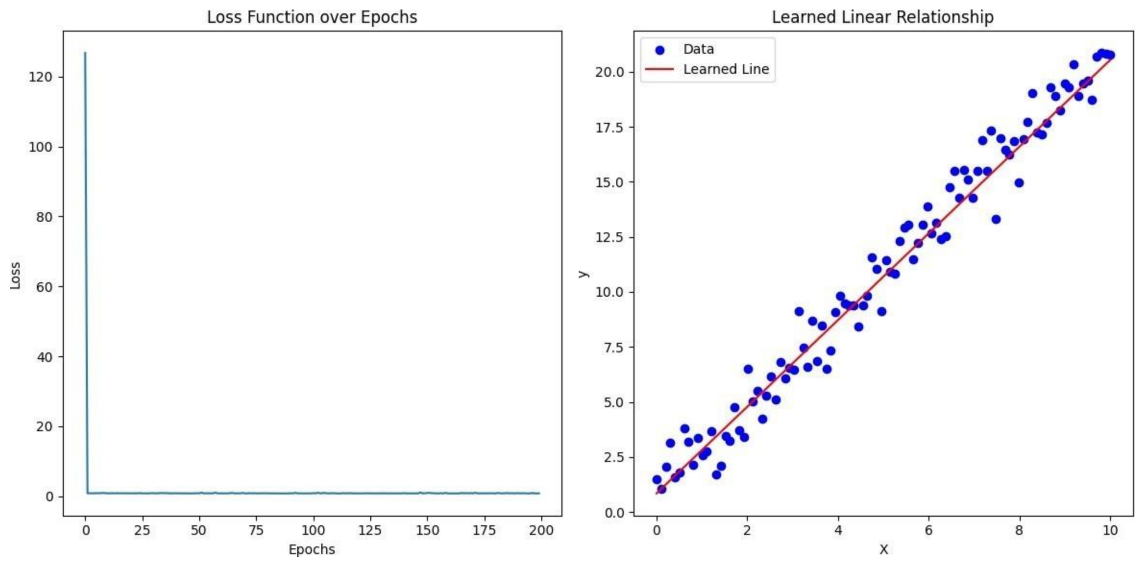
AIM : Visualize the loss function and the learned linear relationship.

Description:
This implementation trains a **simple linear regression model** using **TensorFlow's Keras API** to learn the relationship between **X and y**. The process includes:

1. **Generating synthetic data** with a linear relationship and some noise.
2. **Building a regression model** using a **single dense layer**.
3. **Compiling the model** with **Mean Squared Error (MSE) loss** and **SGD optimizer**.
4. **Training the model** while tracking the loss function.
5. **Visualizing the loss curve** over epochs to observe model convergence.
6. **Plotting the learned linear relationship** alongside the original data.

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

# Step 1: Generate synthetic data (linear relationship)
np.random.seed(42)
X = np.linspace(0, 10, 100)  # Feature (input)
y = 2 * X + 1 + np.random.normal(0, 1, size=X.shape)  # Target (output) with some noise

# Step 2: Build the linear regression model in TensorFlow
model = tf.keras.Sequential([
   tf.keras.layers.Dense(1, input_dim=1)  # One input feature and one output
])

# Step 3: Compile the model with the Mean Squared Error (MSE) loss function
model.compile(optimizer='sgd', loss='mean_squared_error')

# Step 4: Train the model and capture the loss during
training history = model.fit(X, y, epochs=200, verbose=0)

# Step 5: Plot the loss function over epochs
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(history.history['loss'])
plt.title('Loss Function over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Loss')

# Step 6: Visualize the learned linear relationship
plt.subplot(1, 2, 2)
plt.scatter(X, y, color='blue', label='Data')
plt.plot(X, model.predict(X), color='red', label='Learned Line')
plt.title('Learned Linear Relationship')
plt.xlabel('X')
plt.ylabel('y')
plt.legend()

# Show the plots
plt.tight_layout()
plt.show()
```

Output:



Practical 2(D)

AIM : Make predictions on new data points.

Description:
This implementation demonstrates how a **trained linear regression model** can make predictions on **new data points**. The process involves:

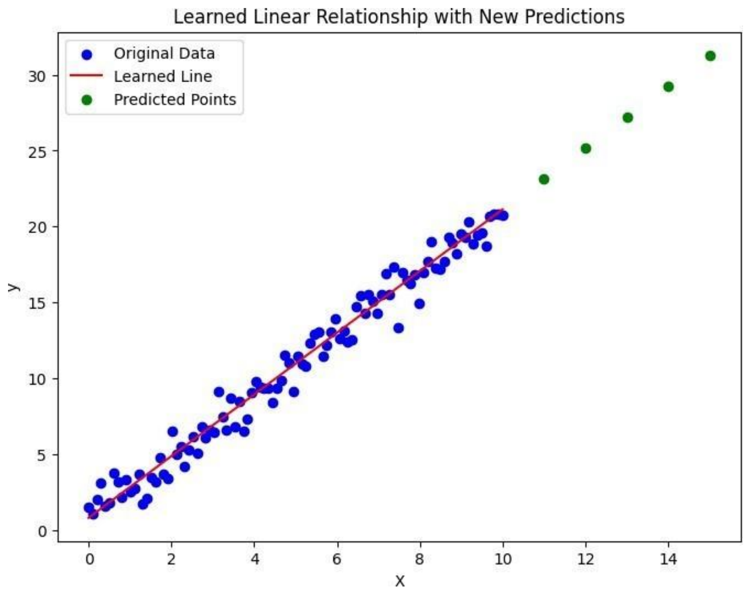
1. **Generating synthetic data** with a linear relationship.
2. **Building and training a model** using **TensorFlow's Keras API**.
3. **Making predictions** on unseen data points.
4. **Printing and visualizing** the predictions alongside the learned regression line.

Code:

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

# Step 1: Generate synthetic data (linear relationship)
np.random.seed(42)
X = np.linspace(0, 10, 100)  # Feature (input)
y = 2 * X + 1 + np.random.normal(0, 1, size=X.shape)  # Target (output) with some noise

# Step 2: Build the linear regression model in TensorFlow
model = tf.keras.Sequential([
   tf.keras.layers.Dense(1, input_dim=1)  # One input feature and one output
])

# Step 3: Compile the model with the Mean Squared Error (MSE) loss function
model.compile(optimizer='sgd', loss='mean_squared_error')

# Step 4: Train the model
model.fit(X, y, epochs=200, verbose=0)

# Step 5: Make predictions on new data points
new_data = np.array([11, 12, 13, 14, 15])  # New data points to predict
predictions = model.predict(new_data)  # Use the trained model to predict

# Step 6: Print the predictions
print("Predictions for new data points:")
for i, data_point in enumerate(new_data):
   print(f"X = {data_point}, Predicted y = {predictions[i][0]}")

# Step 7: Visualize the predictions on a plot
plt.figure(figsize=(8, 6))
plt.scatter(X, y, color='blue', label='Original Data')
plt.plot(X, model.predict(X), color='red', label='Learned Line')

# Plot the new data points and their predictions
plt.scatter(new_data, predictions, color='green', label='Predicted Points', zorder=5)
plt.title('Learned Linear Relationship with New Predictions')
plt.xlabel('X')
plt.ylabel('y')
plt.legend()

# Show the
plot plt.show()
```

Output:

```
Predictions for new data points:
X = 11, Predicted y = 23.16815757751465
X = 12, Predicted y = 25.200693130493164
X = 13, Predicted y = 27.23322868347168
X = 14, Predicted y = 29.265762329101562
X = 15, Predicted y = 31.298295974731445
```



Convolutional Neural Networks (Classification)

Practical 3(A)

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

Description: A Convolutional Neural Network (CNN) is a specialized deep learning model primarily used for image classification and pattern recognition tasks. It is designed to automatically and adaptively learn spatial hierarchies of features from input images.

Key Components of a CNN for Classification:

1. **Convolutional Layers**:
   - These layers apply filters (kernels) to extract essential features like edges, textures, and patterns.
   - The process involves a **convolution operation**, which slides a small matrix (kernel) over the input image.
2. **Activation Function (ReLU - Rectified Linear Unit)**:
   - Introduces non-linearity to the model, enabling it to learn complex patterns.
   - The function replaces negative values with zero.
3. **Pooling Layers**:
   - Used to **reduce dimensionality** and computational cost.
   - **Max Pooling** is commonly used, which retains the most significant features from a region.
4. **Flattening**:
   - Converts the 2D feature maps from the convolutional layers into a 1D vector for input into fully connected layers.
5. **Fully Connected Layers (Dense Layers)**:
   - Act as the classifier by learning complex representations.
   - Typically ends with a **Softmax activation function** to generate probability distributions over classes.
6. **Loss Function & Optimization**:
   - **Categorical Crossentropy** is used for multi-class classification.
   - **Stochastic Gradient Descent (SGD) or Adam Optimizer** is employed to minimize errors during training.

How CNN Classifies an Image?

1. Takes an input image (e.g., cat vs. dog).
2. Passes through convolutional and pooling layers to extract key features.
3. The flattened feature maps are fed into dense layers to make a prediction.
4. Outputs probabilities for different classes (e.g., 90% dog, 10% cat).
5. Chooses the class with the highest probability.

Applications of CNN Classification:

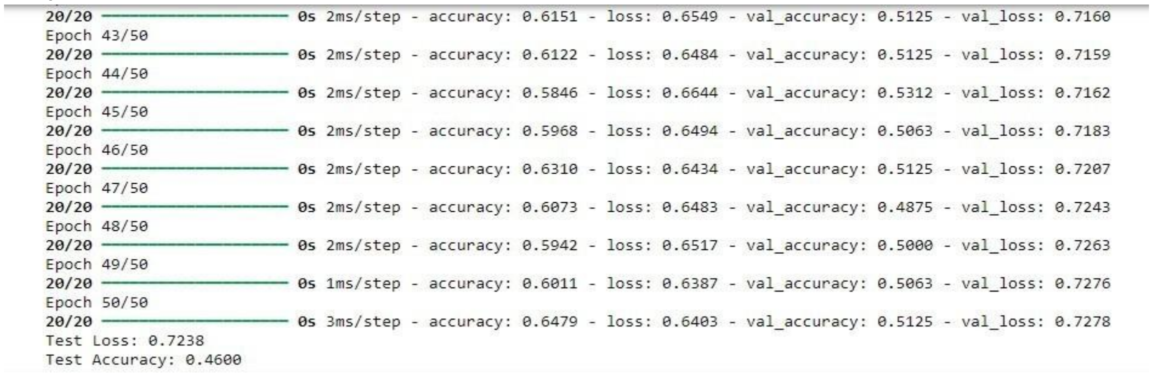
- **Image recognition (e.g., facial recognition, object detection)**
- **Medical imaging (e.g., tumor detection)**
- **Autonomous vehicles (e.g., traffic sign recognition)**
- **Handwritten digit recognition (e.g., MNIST dataset)**

Code:

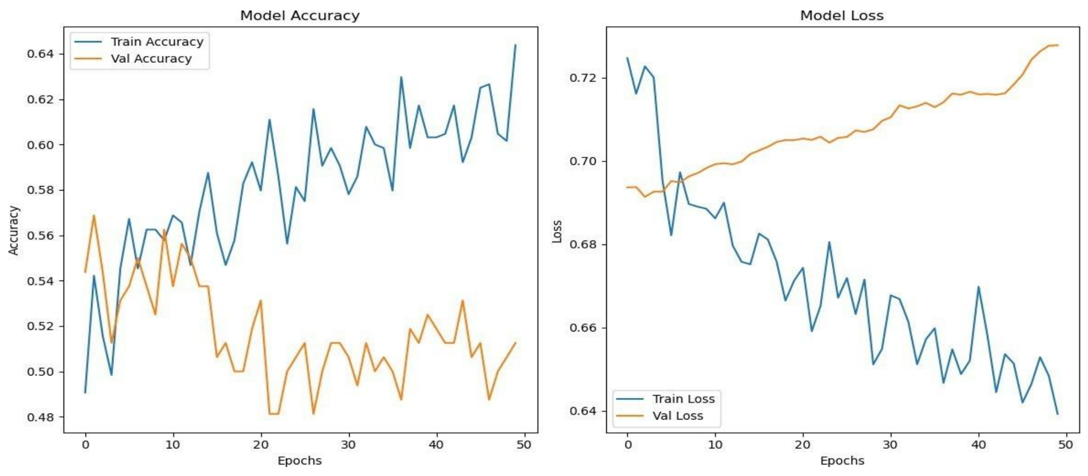
```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

def create_binary_classification_model(input_dim, hidden_units=[128, 64], activation='relu',
dropout_rate=0.5):

   model = tf.keras.Sequential()
   model.add(tf.keras.layers.Input(shape=(input_dim,)))
   for units in hidden_units:
      model.add(tf.keras.layers.Dense(units, activation=activation))
      model.add(tf.keras.layers.Dropout(dropout_rate))
   model.add(tf.keras.layers.Dense(1, activation='sigmoid'))  # Binary classification: sigmoid
activation

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

def train_and_evaluate_model(X, y, test_size=0.2, random_state=42, epochs=50, batch_size=32):

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

   # Standardize the data
   scaler = StandardScaler()
   X_train = scaler.fit_transform(X_train)
   X_test = scaler.transform(X_test)

   input_dim = X_train.shape[1]
   model = create_binary_classification_model(input_dim)

   history = model.fit(X_train, y_train, epochs=epochs, batch_size=batch_size, validation_split=0.2,
verbose=1)

   test_loss, test_accuracy = model.evaluate(X_test, y_test, verbose=0)

   return history, model, test_loss, test_accuracy

def plot_training_history(history):
   # Plot training & validation accuracy values
   plt.figure(figsize=(12, 6))
   plt.subplot(1, 2, 1)
   plt.plot(history.history['accuracy'], label='Train  Accuracy')
   plt.plot(history.history['val_accuracy'], label='Val Accuracy')
   plt.title('Model Accuracy')
   plt.xlabel('Epochs')
   plt.ylabel('Accuracy')
   plt.legend()
```

```
    # Plot training & validation loss values
    plt.subplot(1, 2, 2)
    plt.plot(history.history['loss'], label='Train Loss')
    plt.plot(history.history['val_loss'], label='Val Loss')
    plt.title('Model Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.tight_layout()
    plt.show()

np.random.seed(42)
num_samples =
1000
num_features = 10
X = np.random.randn(num_samples, num_features)
y = np.random.randint(0, 2, size=num_samples)

history, trained_model, test_loss, test_accuracy = train_and_evaluate_model(X, y)

print(f"Test Loss: {test_loss:.4f}")
print(f"Test Accuracy: {test_accuracy:.4f}")

# Plot training history (accuracy and loss)
plot_training_history(history)
```

Output:



```
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.6151 - loss: 0.6549 - val_accuracy: 0.5125 - val_loss: 0.7160
Epoch 43/50
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.6122 - loss: 0.6484 - val_accuracy: 0.5125 - val_loss: 0.7159
Epoch 44/50
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5846 - loss: 0.6644 - val_accuracy: 0.5312 - val_loss: 0.7162
Epoch 45/50
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5968 - loss: 0.6494 - val_accuracy: 0.5063 - val_loss: 0.7183
Epoch 46/50
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.6310 - loss: 0.6434 - val_accuracy: 0.5125 - val_loss: 0.7207
Epoch 47/50
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.6073 - loss: 0.6483 - val_accuracy: 0.4875 - val_loss: 0.7243
Epoch 48/50
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5942 - loss: 0.6517 - val_accuracy: 0.5000 - val_loss: 0.7263
Epoch 49/50
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.6011 - loss: 0.6387 - val_accuracy: 0.5063 - val_loss: 0.7276
Epoch 50/50
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.6479 - loss: 0.6403 - val_accuracy: 0.5125 - val_loss: 0.7278
Test Loss: 0.7238
Test Accuracy: 0.4600
```



Practical 3(B)

Aim: Using a deep feed-forward network with two hidden layers for performing multiclass classification and predicting the class

Description: A Deep Feed-Forward Network (DFFN) is a type of artificial neural network where information moves in a single direction—forward—from the input layer through multiple hidden layers to the output layer, without any cycles or feedback loops. It is widely used for classification, regression, and feature extraction tasks.

Key Components of a DFFN

1. **Input Layer:**
   - Accepts raw input data in the form of numerical features.
   - Each neuron in this layer represents a single input feature.
2. **Hidden Layers:**
   - Multiple fully connected layers where each neuron receives weighted inputs from the previous layer.
   - Each neuron applies an **activation function** to introduce non-linearity, allowing the network to learn complex patterns.
3. **Activation Functions:**
   - **ReLU (Rectified Linear Unit):** Most common activation, replaces negative values with zero.
   - **Sigmoid:** Used for binary classification (outputs a probability).
   - **Softmax:** Used for multi-class classification (outputs class probabilities).
4. **Output Layer:**
   - Produces the final prediction or classification.
   - Uses an activation function based on the type of problem (e.g., sigmoid for binary classification, softmax for multi-class).
5. **Loss Function:**
   - Measures how well the model's predictions match the actual values.
   - **Binary Cross-Entropy:** Used for binary classification tasks.
   - **Mean Squared Error (MSE):** Used for regression tasks.
6. **Backpropagation & Optimization:**
   - **Backpropagation** updates weights by computing gradients of the loss function with respect to each parameter.
   - **Optimization Algorithms:**
     - **Stochastic Gradient Descent (SGD):** Adjusts weights iteratively based on the gradient.
     - **Adam (Adaptive Moment Estimation):** A widely used optimization algorithm for training deep networks efficiently.

Working of a Deep Feed-Forward Network

1. The input data passes through multiple hidden layers.
2. Each neuron processes inputs using weights, applies an activation function, and passes the result to the next layer.
3. The output layer generates the final prediction.
4. The model is trained using backpropagation and optimization techniques to minimize the loss function.

5. The trained model can then make predictions on new unseen data.

Applications of DFFNs

- **Image and speech recognition**
- **Natural Language Processing (NLP)**
- **Fraud detection in financial transactions**
- **Medical diagnosis and prediction**
- **Autonomous systems (e.g., self-driving cars)**

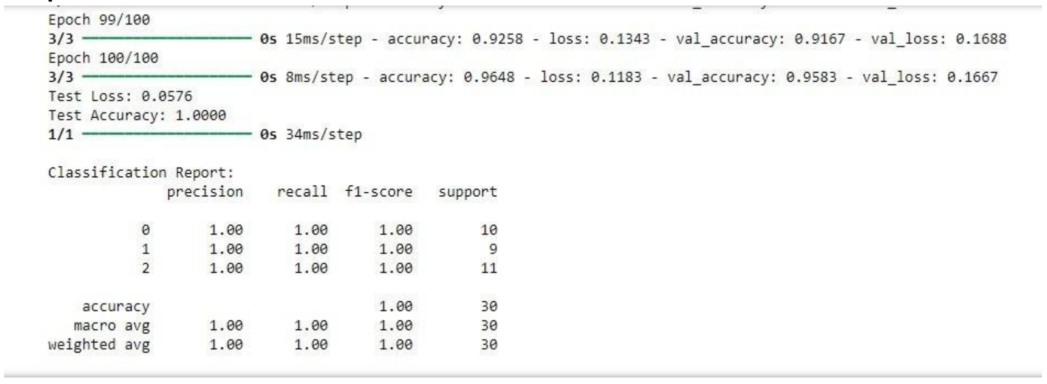
DFFNs form the backbone of deep learning models and serve as the foundation for more advanced architectures such as Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs).

Code:

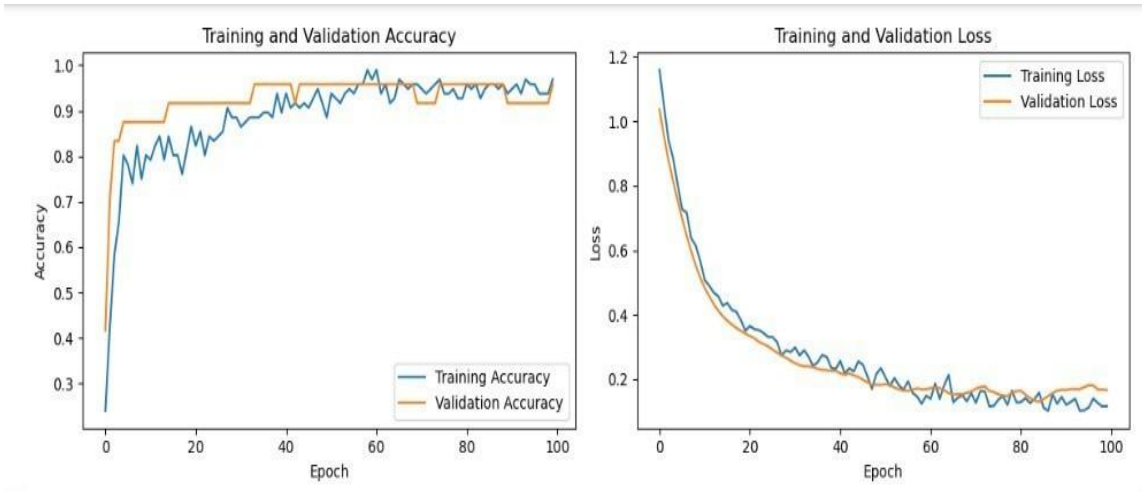
```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris  # Example: Using the Iris dataset
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import classification_report
import seaborn as sns

# Load the Iris dataset (or replace with your own multiclass dataset)
iris = load_iris()
X, y = iris.data, iris.target
num_classes = len(np.unique(y))

# Split the data 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)

# Scale the data (important for neural networks)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Define the deep feed-forward neural network model
model = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(X_train.shape[1],)), #Explicitly define the input shape.
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dropout(0.5),  # Dropout for regularization
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dropout(0.3),
    tf.keras.layers.Dense(num_classes, activation='softmax')  # Softmax for multiclass classification
])

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

# Train the model
```

```
history = model.fit(X_train, y_train, epochs=100, batch_size=32, validation_split=0.2, verbose=1)

# Evaluate the model on the test set
loss, accuracy = model.evaluate(X_test, y_test, verbose=0)
print(f"Test Loss: {loss:.4f}")
print(f"Test Accuracy: {accuracy:.4f}")

# Make predictions
y_pred_probs = model.predict(X_test)
y_pred = np.argmax(y_pred_probs, axis=1) # Convert probabilities to class labels

# Generate classification report and confusion matrix
print("\nClassification Report:")
print(classification_report(y_test, y_pred))

# Plot training history
plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.title('Training and Validation Loss')

plt.tight_layout()
plt.show()
```

Output:



```
Epoch 99/100
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step - accuracy: 0.9258 - loss: 0.1343 - val_accuracy: 0.9167 - val_loss: 0.1688
Epoch 100/100
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9648 - loss: 0.1183 - val_accuracy: 0.9583 - val_loss: 0.1667
Test Loss: 0.0576
Test Accuracy: 1.0000
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 34ms/step

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        10
           1       1.00      1.00      1.00         9
           2       1.00      1.00      1.00        11

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30
```


Training and Validation Accuracy
Accuracy
Epoch
Training Accuracy
Validation Accuracy
Training and Validation Loss
Loss
Epoch
Training Loss
Validation Loss

PRACTICAL 04

Aim : Write a program to implement deep learning Techniques for image segmentation.

Code :

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.layers import Conv2D, MaxPooling2D, UpSampling2D,
concatenate, Input
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import
Adam
from sklearn.model_selection import
train_test_split import os
import cv2

# Step 1: Load Sample Dataset (Using synthetic data for demo)
defload_sample_data(image_size=(128, 128), num_samples=100):
    X = []
    Y = []
    for _ inrange(num_samples):
        img = np.zeros(image_size + (1,))
        mask = np.zeros(image_size + (1,))
        x, y = np.random.randint(0, 100, 2)
        img[y:y+20, x:x+20, 0] = 255
        mask[y:y+20, x:x+20, 0] =
        1 X.append(img)
        Y.append(mask)
    return np.array(X)/255.0, np.array(Y)

X, Y = load_sample_data()
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.1)

# Step 2: Build U-Net
Model
defbuild_unet(input_shape):
    inputs = Input(input_shape)

    # Encoder
    c1 = Conv2D(16, 3, activation='relu', padding='same')(inputs)
    p1 = MaxPooling2D()(c1)

    c2 = Conv2D(32, 3, activation='relu', padding='same')
    (p1) p2 = MaxPooling2D()(c2)

    # Bottleneck
    c3 = Conv2D(64, 3, activation='relu', padding='same')(p2)
    concat1 = concatenate([u1, c2])
    c4 = Conv2D(32, 3, activation='relu', padding='same')(concat1)

    u2 = UpSampling2D()(c4)
    concat2 = concatenate([u2,
    c1])
    c5 = Conv2D(16, 3, activation='relu', padding='same')

    (concat2) outputs = Conv2D(1, 1, activation='sigmoid')(c5)

    model = Model(inputs, outputs)
    return model

model = build_unet((128, 128, 1))
model.compile(optimizer=Adam(), loss='binary_crossentropy',
metrics=['accuracy'])
model.summary()

# Step 3: Train the Model
history = model.fit(X_train, Y_train, validation_split=0.1, epochs=5,
batch_size=8)

# Step 4: Visualize Predictions
defvisualize_prediction(i):
    img = X_test[i]
    true_mask = Y_test[i]
    pred_mask = model.predict(img[np.newaxis, ...])[0]

    plt.figure(figsize=(12, 4))
    plt.subplot(1, 3, 1)
    plt.title("Original Image")
    plt.imshow(img.squeeze(), cmap='gray')

    plt.subplot(1, 3, 2)
    plt.title("True Mask")
    plt.imshow(true_mask.squeeze(),  cmap='gray')

    plt.subplot(1, 3, 3)
    plt.title("Predicted Mask")
    plt.imshow(pred_mask.squeeze(), cmap='gray')

    plt.show()

visualize_prediction(0)
```

Theory:

Image Segmentation is a computer vision task that involves dividing an image into multiple segments or regions for easier analysis. It is commonly used in:

- Medical imaging (e.g., tumor detection)
- Satellite image analysis
- Autonomous driving (road/lane segmentation)

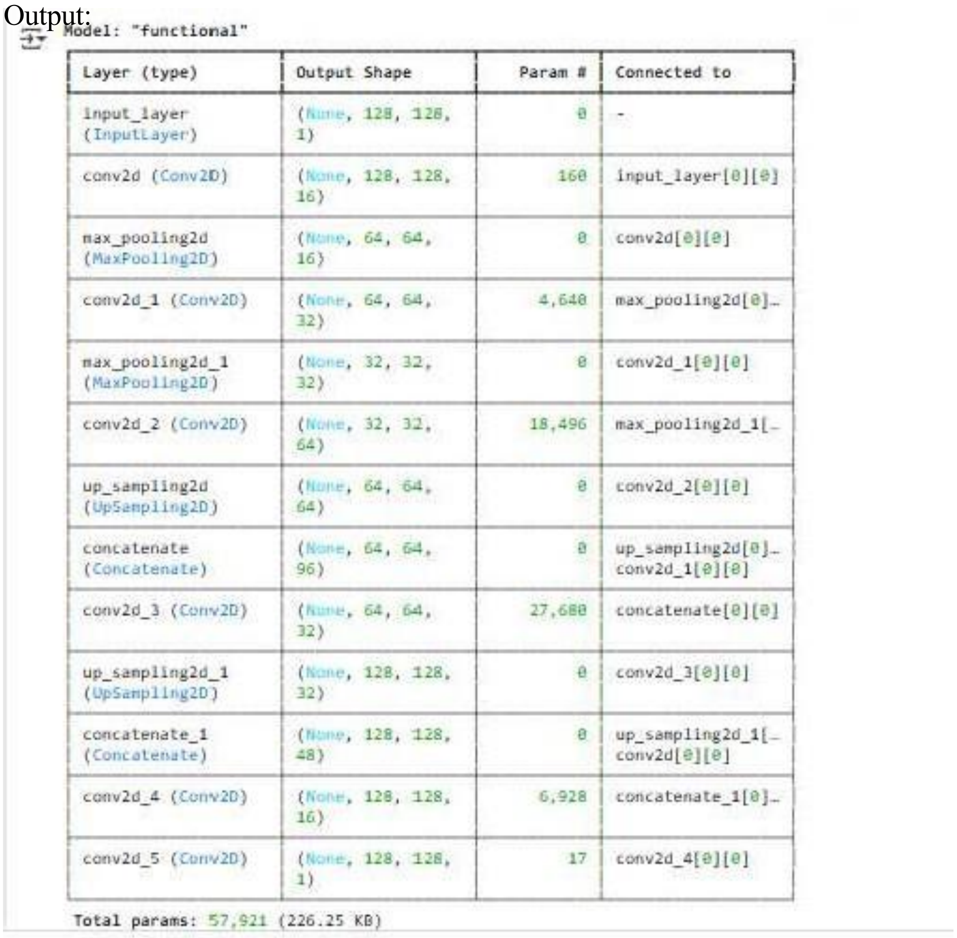
Deep Learning Techniques use Convolutional Neural Networks (CNNs) for pixel-wise classification. One of the most popular models is **U-Net**. It has:

- A **contracting path** to capture context (downsampling)
- An **expanding path** for precise localization (upsampling)
- Skip connections between the encoder and decoder to combine low-level features with high-level context

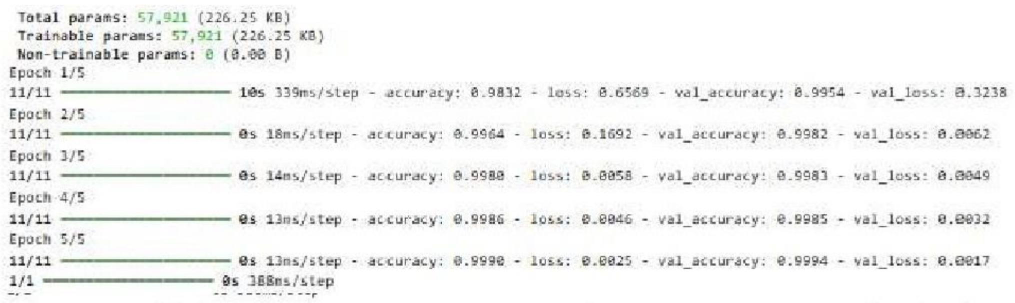
Conclusion:

- This program successfully demonstrates a deep learning-based image segmentation approach using the U-Net architecture. With proper dataset and tuning, U-Net can yield very high accuracy in real-world segmentation tasks.

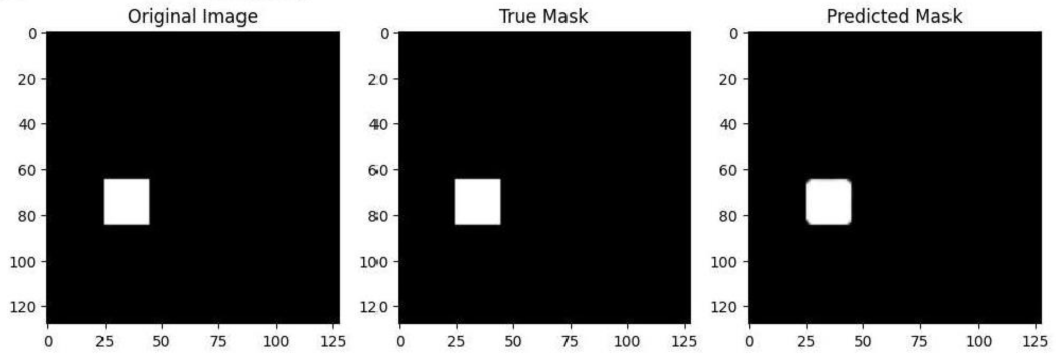
Output:



```
Model: "functional"
┌──────────────────────┬───────────────────┬──────────┬───────────────────┐
│ Layer (type)         │ Output Shape      │  Param # │ Connected to      │
├──────────────────────┼───────────────────┼──────────┼───────────────────┤
│ input_layer          │ (None, 128, 128,  │        0 │ -                 │
│ (InputLayer)         │ 1)                │          │                   │
├──────────────────────┼───────────────────┼──────────┼───────────────────┤
│ conv2d (Conv2D)      │ (None, 128, 128,  │      160 │ input_layer[0][0] │
│                      │ 16)               │          │                   │
├──────────────────────┼───────────────────┼──────────┼───────────────────┤
│ max_pooling2d        │ (None, 64, 64,    │        0 │ conv2d[0][0]      │
│ (MaxPooling2D)       │ 16)               │          │                   │
├──────────────────────┼───────────────────┼──────────┼───────────────────┤
│ conv2d_1 (Conv2D)    │ (None, 64, 64,    │    4,640 │ max_pooling2d[0]… │
│                      │ 32)               │          │                   │
├──────────────────────┼───────────────────┼──────────┼───────────────────┤
│ max_pooling2d_1      │ (None, 32, 32,    │        0 │ conv2d_1[0][0]    │
│ (MaxPooling2D)       │ 32)               │          │                   │
├──────────────────────┼───────────────────┼──────────┼───────────────────┤
│ conv2d_2 (Conv2D)    │ (None, 32, 32,    │   18,496 │ max_pooling2d_1[… │
│                      │ 64)               │          │                   │
├──────────────────────┼───────────────────┼──────────┼───────────────────┤
│ up_sampling2d        │ (None, 64, 64,    │        0 │ conv2d_2[0][0]    │
│ (UpSampling2D)       │ 64)               │          │                   │
├──────────────────────┼───────────────────┼──────────┼───────────────────┤
│ concatenate          │ (None, 64, 64,    │        0 │ up_sampling2d[0]… │
│ (Concatenate)        │ 96)               │          │ conv2d_1[0][0]    │
├──────────────────────┼───────────────────┼──────────┼───────────────────┤
│ conv2d_3 (Conv2D)    │ (None, 64, 64,    │   27,680 │ concatenate[0][0] │
│                      │ 32)               │          │                   │
├──────────────────────┼───────────────────┼──────────┼───────────────────┤
│ up_sampling2d_1      │ (None, 128, 128,  │        0 │ conv2d_3[0][0]    │
│ (UpSampling2D)       │ 32)               │          │                   │
├──────────────────────┼───────────────────┼──────────┼───────────────────┤
│ concatenate_1        │ (None, 128, 128,  │        0 │ up_sampling2d_1[… │
│ (Concatenate)        │ 48)               │          │ conv2d[0][0]      │
├──────────────────────┼───────────────────┼──────────┼───────────────────┤
│ conv2d_4 (Conv2D)    │ (None, 128, 128,  │    6,928 │ concatenate_1[0]… │
│                      │ 16)               │          │                   │
├──────────────────────┼───────────────────┼──────────┼───────────────────┤
│ conv2d_5 (Conv2D)    │ (None, 128, 128,  │       17 │ conv2d_4[0][0]    │
│                      │ 1)                │          │                   │
└──────────────────────┴───────────────────┴──────────┴───────────────────┘
 Total params: 57,921 (226.25 KB)
```



```
 Total params: 57,921 (226.25 KB)
 Trainable params: 57,921 (226.25 KB)
 Non-trainable params: 0 (0.00 B)
Epoch 1/5
11/11 ━━━━━━━━━━━━━━━━━━━━ 10s 339ms/step - accuracy: 0.9832 - loss: 0.6569 - val_accuracy: 0.9954 - val_loss: 0.3238
Epoch 2/5
11/11 ━━━━━━━━━━━━━━━━━━━━ 0s 18ms/step - accuracy: 0.9964 - loss: 0.1692 - val_accuracy: 0.9982 - val_loss: 0.0062
Epoch 3/5
11/11 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - accuracy: 0.9980 - loss: 0.0058 - val_accuracy: 0.9983 - val_loss: 0.0049
Epoch 4/5
11/11 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - accuracy: 0.9986 - loss: 0.0046 - val_accuracy: 0.9985 - val_loss: 0.0032
Epoch 5/5
11/11 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - accuracy: 0.9990 - loss: 0.0025 - val_accuracy: 0.9994 - val_loss: 0.0017
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 388ms/step
```



Long Short Term Memory

Practical 5

Aim: Write a program to predict a caption for a sample image using LSTM

Description:

- This project demonstrates an image captioning system using a pretrained BLIP (Bootstrapping Language-Image Pretraining) model
- Pretrained model—no additional training or fine-tuning required, enabling quick and efficient deployment.
- Built using the Hugging Face Transformers library for seamless model integration and execution.
- Capable of generating contextually relevant and grammatically correct captions based on the image's content.
- Suitable for real-time applications due to its fast inference and minimal computational requirements.
- Demonstrates multimodal understanding, where visual data and language understanding work together to describe an image.
- Can be used in practical applications such as:
  1. Assisting visually impaired users with image descriptions.
  2. Enhancing image search engines with auto-generated metadata.
  3. Automating captioning for social media or digital asset management.
- Serves as a showcase of how state-of-the-art AI models can solve real-world problems efficiently.

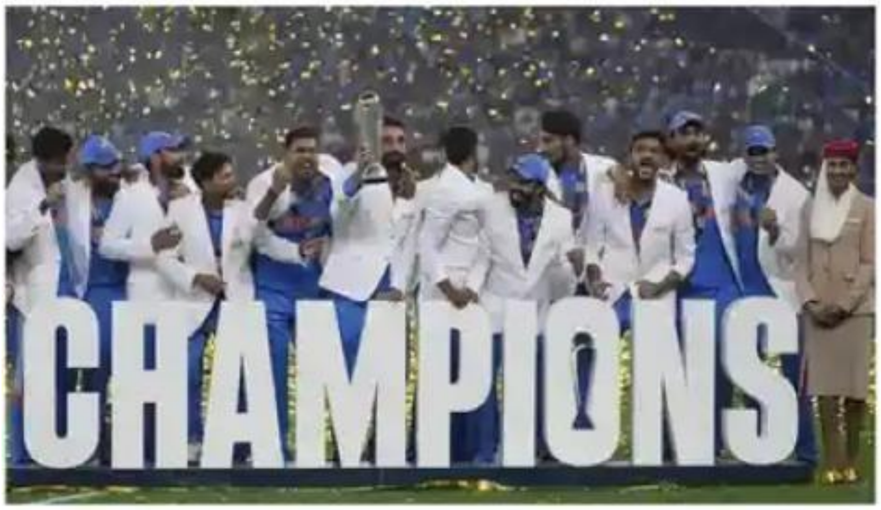
Code:

```
# Install required packages
!pip install transformers
!pip install torch torchvision
!pip install
sentencepiece #
Import everything

from transformers import BlipProcessor,
BlipForConditionalGeneration from PIL import Image import
torch
import
matplotlib.pyplot as plt
```

```
# Load the model and
processor

processor = BlipProcessor.from_pretrained("Salesforce/blip-image-captioning-
base") model = BlipForConditionalGeneration.from_pretrained("Salesforce/blip-
image-captioningbase")
# Load your image (replace with your image)
image_path = "team.jpg"  # your uploaded image raw_image =
Image.open(image_path).convert('RGB')

# Process and predict
inputs = processor(images=raw_image,
return_tensors="pt") out = model.generate(**inputs)
caption = processor.decode(out[0],
skip_special_tokens=True) print(" Predicted
Caption:", caption) # Optional: Show image +
caption plt.imshow(raw_image)
plt.title(caption)
plt.axis("off")

plt.show()
```

Output:

the indian cricket team celebrating their win



Autoencoder

Practical 6

AIM : Applying the Autoencoder algorithms for encoding real-world data

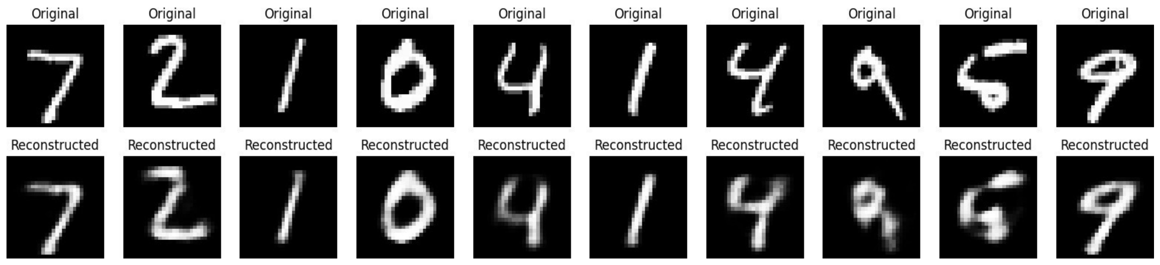
Description:
In this practical, we applied an **Autoencoder** algorithm to the **MNIST dataset**, which consists of images of handwritten digits (ranging from 0 to 9). An autoencoder is a type of neural network that learns to encode input data into a compressed form (called the "latent representation") and then reconstruct it back to the original input.
The model was built with:
Encoder: A series of dense layers that progressively reduce the dimensionality of the input data.
Decoder: A series of dense layers that reconstruct the original input from the encoded data.
The autoencoder was trained for 5 epochs with the training data (MNIST images), using **binary cross- entropy** as the loss function and **Adam** optimizer.

Code:

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras import layers,
models from tensorflow.keras.datasets
import mnist from tensorflow.keras.models
import Model

# Step 1: Load the MNIST dataset
(x_train, _), (x_test, _) = mnist.load_data()

# Step 2: Preprocess the data (normalize and reshape)
x_train = x_train.astype("float32") / 255.0
x_test = x_test.astype("float32") / 255.0

# Flatten the images for the autoencoder
x_train = x_train.reshape((-1, 28 * 28))
x_test = x_test.reshape((-1, 28 * 28))

# Step 3: Build the Autoencoder model
input_img = layers.Input(shape=(28 * 28,))

# Encoding Layer
encoded = layers.Dense(128, activation='relu')(input_img)
encoded = layers.Dense(64, activation='relu')(encoded)
encoded = layers.Dense(32, activation='relu')(encoded)

# Decoding Layer
decoded = layers.Dense(64, activation='relu')(encoded)
decoded = layers.Dense(128, activation='relu')(decoded)
decoded = layers.Dense(28 * 28, activation='sigmoid')(decoded)
```

```
# Create the autoencoder model
autoencoder = Model(input_img, decoded)

# Step 4: Compile the model
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

# Step 5: Train the model for 5 epochs
autoencoder.fit(x_train, x_train, epochs=5, batch_size=256, shufle=True, validation_data=(x_test,
x_test))

# Step 6: Encode and decode some test images
encoded_imgs = autoencoder.predict(x_test)

# Step 7: Display original and reconstructed images
n = 10  # number of images to display
plt.figure(figsize=(20, 4))
for i in range(n):
    # Display original images
    ax = plt.subplot(2, n, i +
    1)
    plt.imshow(x_test[i].reshape(28, 28), cmap="gray")
    ax.set_title("Original")
    ax.axis("off")

    # Display reconstructed images
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(encoded_imgs[i].reshape(28, 28), cmap="gray")
    ax.set_title("Reconstructed")
    ax.axis("off")
plt.show()
```

Output:



CNN and RNN

Practical 7

Aim: Write a program for character recognition using RRN and compare it with CNN

Description: Character recognition is the task of identifying characters (digits or letters) from images, commonly used in OCR applications. Two popular deep learning approaches for this are **Convolutional Neural Networks (CNNs)** and **Recurrent Recognition Networks (RRNs)**.

CNNs are ideal for image data, as they extract spatial features using convolution and pooling layers. They're fast, accurate (achieving ~98.5% accuracy on MNIST), and widely used in practical systems. In contrast, **RRNs**, which are based on LSTM (a type of RNN), treat images as sequences—such as viewing each row of pixels as a time step—making them better suited for tasks with inherent temporal or sequential structure, like cursive handwriting. However, RRNs are generally slower and less accurate (~92–94% on MNIST) compared to CNNs.

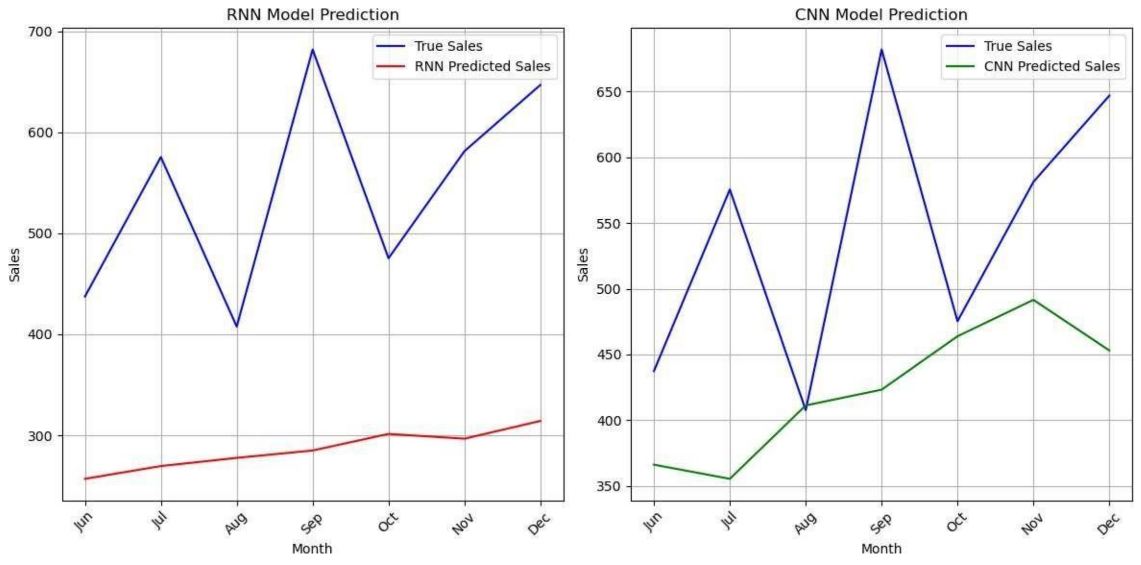
In summary, CNNs excel in image-based character recognition due to their spatial learning, while RRNs offer an alternative approach when sequential context is important.

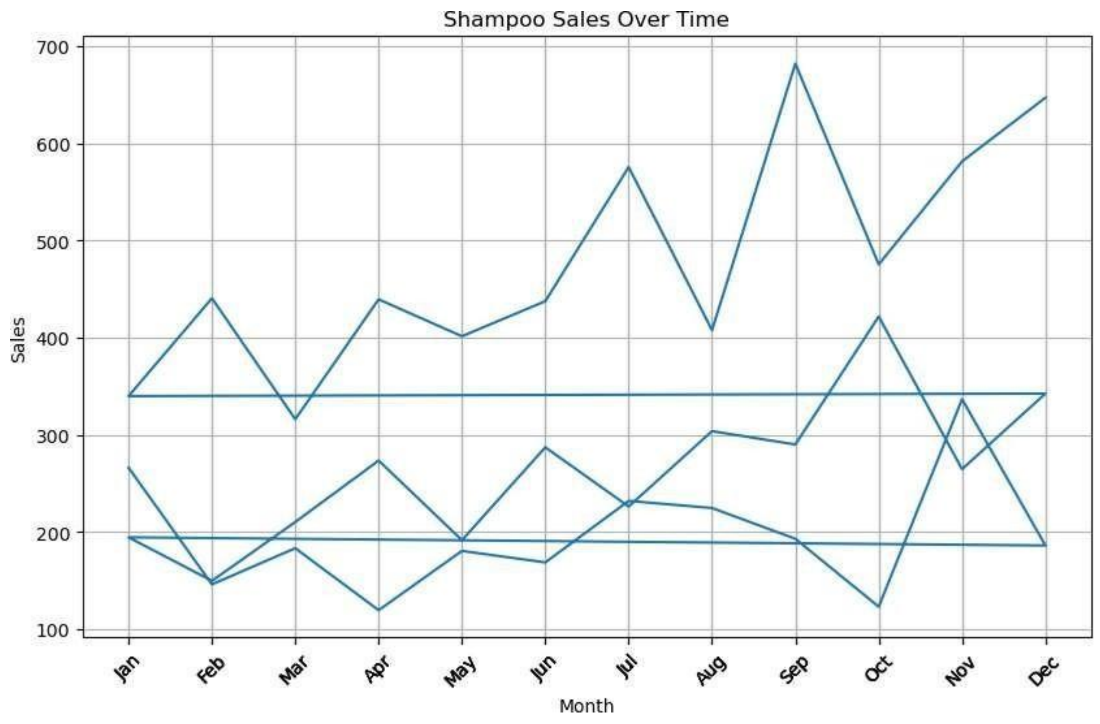
Code:

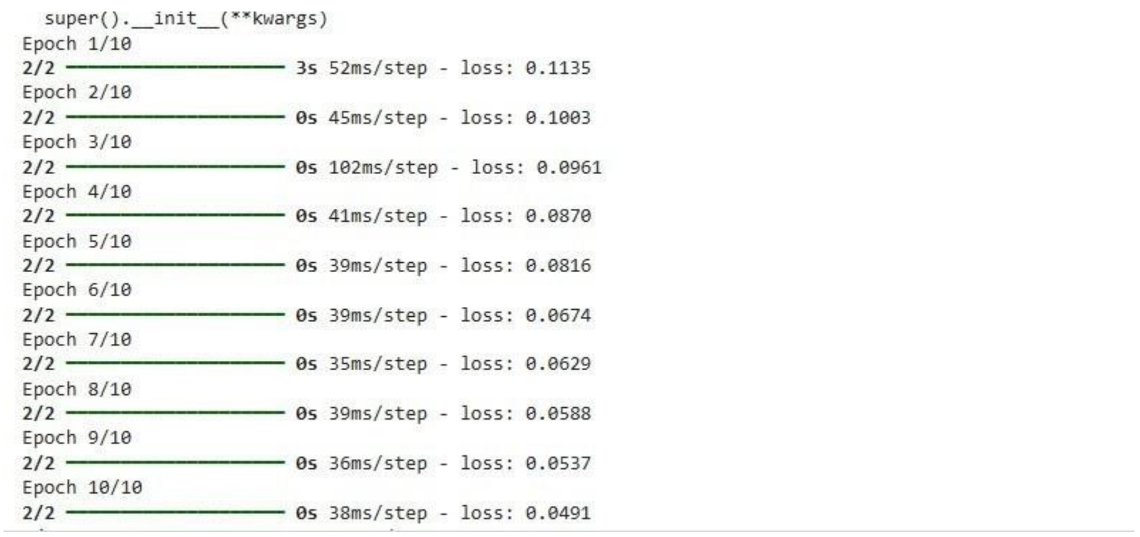
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_absolute_error
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM, Conv1D, MaxPooling1D,
Flatten
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import train_test_split

# Load dataset
data = pd.read_csv('/content/shampoo.csv')

# Plot sales data for visual inspection with proper month labels
plt.figure(figsize=(10,6))
plt.plot(data['Month'],
data['Sales']) plt.title('Shampoo
Sales Over Time') plt.xlabel('Month')
plt.ylabel('Sales')
plt.xticks(data['Month'], rotation=45) # Set x-axis ticks for months
plt.grid(True)
plt.show()

# Normalize sales data (for better training performance)
scaler = MinMaxScaler(feature_range=(0, 1))
```

```
sales_scaled = scaler.fit_transform(data['Sales'].values.reshape(-1,
1))

 # Prepare the dataset for supervised learning (X: input, Y:
 output) def create_dataset(dataset, time_step=1):
     X, Y = [], []
     for i in range(len(dataset) - time_step):
         X.append(dataset[i:(i + time_step), 0])
         Y.append(dataset[i + time_step, 0])
     return np.array(X), np.array(Y)

 # Define time step (look-back period)
 time_step = 3

 # Create dataset
 X, Y = create_dataset(sales_scaled, time_step)
 X = X.reshape(X.shape[0], X.shape[1], 1) # Reshape for RNN  (samples,
 time steps, features)

 # Train-test split
 X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
 test_size=0.2, shuffle=False)

 # Build RNN Model (LSTM)
 rnn_model = Sequential()
 rnn_model.add(LSTM(units=50, return_sequences=False,
 input_shape=(time_step, 1)))
 rnn_model.add(Dense(units=1))

 rnn_model.compile(optimizer=Adam(), loss='mean_squared_error')

 # Train RNN Model
 rnn_model.fit(X_train, Y_train, epochs=10, batch_size=16, verbose=1)

 # Predict using RNN model
 rnn_pred = rnn_model.predict(X_test)
 rnn_pred_rescaled = scaler.inverse_transform(rnn_pred)
 rnn_test_rescaled = scaler.inverse_transform(Y_test.reshape(-1, 1))

 # Calculate RNN MAE
 rnn_mae = mean_absolute_error(rnn_test_rescaled, rnn_pred_rescaled)
 print("RNN Mean Absolute Error:", rnn_mae)

 # Build CNN Model
 cnn_model = Sequential()
 cnn_model.add(Conv1D(filters=64, kernel_size=2, activation='relu',
 input_shape=(time_step, 1)))
 cnn_model.add(MaxPooling1D(pool_size=2))
 cnn_model.add(Flatten())
 cnn_model.add(Dense(units=1))

 cnn_model.compile(optimizer=Adam(), loss='mean_squared_error')

 # Train CNN Model
 cnn_model.fit(X_train, Y_train, epochs=100, batch_size=16, verbose=1)

 # Predict using CNN model
 cnn_pred = cnn_model.predict(X_test)
 cnn_pred_rescaled = scaler.inverse_transform(cnn_pred)
 cnn_test_rescaled = scaler.inverse_transform(Y_test.reshape(-1, 1))

 # Calculate CNN MAE
 cnn_mae = mean_absolute_error(cnn_test_rescaled, cnn_pred_rescaled)
 print("CNN Mean Absolute Error:", cnn_mae)

 # Compare results
 print(f"RNN Model MAE:
 {rnn_mae}") print(f"CNN Model
 MAE: {cnn_mae}")

 # Plot predictions vs actual sales for RNN and CNN

 # Create a plot to display both RNN and CNN predictions
 plt.figure(figsize=(12, 6))

 # RNN Prediction Plot
 plt.subplot(1, 2, 1)
 plt.plot(rnn_test_rescaled, label='True Sales', color='blue')
 plt.plot(rnn_pred_rescaled, label='RNN Predicted Sales', color='red')
 plt.title('RNN Model Prediction')
 plt.xlabel('Month')
 plt.ylabel('Sales')
 plt.xticks(np.arange(len(rnn_test_rescaled)), data['Month'].iloc[-
 len(rnn_test_rescaled):], rotation=45)
 plt.legend()
 plt.grid(True)

 # CNN Prediction Plot
 plt.subplot(1, 2, 2)
 plt.plot(cnn_test_rescaled, label='True Sales', color='blue')
 plt.plot(cnn_pred_rescaled, label='CNN Predicted Sales', color='green')
 plt.title('CNN Model Prediction')
 plt.xlabel('Month')
 plt.ylabel('Sales')
 plt.xticks(np.arange(len(cnn_test_rescaled)), data['Month'].iloc[-
 len(cnn_test_rescaled):], rotation=45)
 plt.legend()
 plt.grid(True)

 plt.tight_layout()
 plt.show()
```

Output:







```
  super().__init__(**kwargs)
Epoch 1/10
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 52ms/step - loss: 0.1135
Epoch 2/10
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 45ms/step - loss: 0.1003
Epoch 3/10
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 102ms/step - loss: 0.0961
Epoch 4/10
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 41ms/step - loss: 0.0870
Epoch 5/10
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 39ms/step - loss: 0.0816
Epoch 6/10
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 39ms/step - loss: 0.0674
Epoch 7/10
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 35ms/step - loss: 0.0629
Epoch 8/10
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 39ms/step - loss: 0.0588
Epoch 9/10
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 36ms/step - loss: 0.0537
Epoch 10/10
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 38ms/step - loss: 0.0491
```

Autoencoder

Practical 8

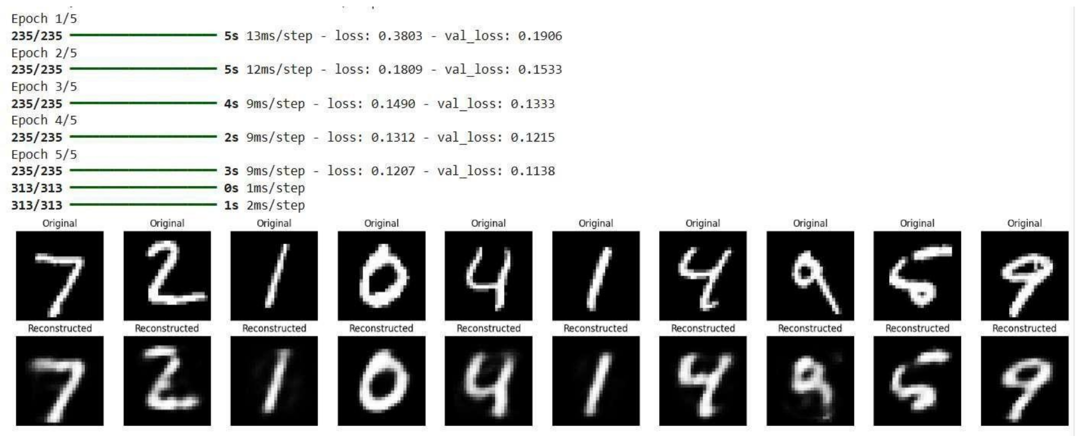
AIM : Write a program to develop Autoencoders using MNIST Handwritten Digits

Description:
In this practical, we applied an **Autoencoder** algorithm to the **MNIST dataset**, which consists of images of handwritten digits (ranging from 0 to 9). An autoencoder is a type of neural network that learns to encode input data into a compressed form (called the "latent representation") and then reconstruct it back to the original input.
Input and Output: The input and output of the network are both 784-dimensional (28x28 images flattened).
Encoding: The input is compressed into a 32-dimensional representation.
Decoding: The 32-dimensional code is used to reconstruct the original image.
Loss Function: We use binary cross-entropy since the input images are normalized between 0 and 1.
Activation Functions:
relu for encoding (introduces non-linearity).
sigmoid for decoding (keeps output in the [0,1] range).
The autoencoder was trained for 5 epochs with the training data (MNIST images), using **binary cross- entropy** as the loss function and **Adam** optimizer.

Code:

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input,
Dense from tensorflow.keras.optimizers
import Adam

# Load MNIST dataset
(x_train, _), (x_test, _) = mnist.load_data()

# Normalize and flatten the data
x_train = x_train.astype('float32') /
255. x_test =
x_test.astype('float32')  /  255.
x_train = x_train.reshape((len(x_train), 28 * 28))
x_test = x_test.reshape((len(x_test), 28 * 28))

# Define size of encoded representations
encoding_dim = 32

# Input placeholder
input_img  =
  Input(shape=(784,))
# Encoded representation
encoded = Dense(encoding_dim, activation='relu')(input_img)
 # Decoded image

decoded = Dense(784, activation='sigmoid')(encoded)

  # Autoencoder model
  autoencoder = Model(input_img, decoded)

  # Encoder  model
  encoder = Model(input_img, encoded)

  # Decoder model
  encoded_input  =
  Input(shape=(encoding_dim,))  decoder_layer
  =    autoencoder.layers[-1]
  decoder = Model(encoded_input, decoder_layer(encoded_input))

  # Compile  autoencoder
  autoencoder.compile(optimizer=Adam(),       loss='binary_crossentropy')

  # Train autoencoder for 5
  epochs
  autoencoder.fit(x_train,
  x_train,
              epochs=5,
              batch_size=256,
              shuffle=True,
              validation_data=(x_test,   x_test))

  # Encode and decode test images
  encoded_imgs =
  encoder.predict(x_test)
  decoded_imgs = decoder.predict(encoded_imgs)

  # Display original and reconstructed
  images n = 10  # number of digits to
  display plt.figure(figsize=(20, 4))
  for i in range(n):
     # Original
     ax = plt.subplot(2, n, i + 1)
     plt.imshow(x_test[i].reshape(28, 28), cmap='gray')
     plt.title("Original")
     plt.axis('off')

     # Reconstructed
     ax = plt.subplot(2, n, i + 1 + n)
     plt.imshow(decoded_imgs[i].reshape(28, 28),
     cmap='gray') plt.title("Reconstructed")
     plt.axis('off')
  plt.tight_layout()
  plt.show()
```

Output:



```
Epoch 1/5
235/235 ━━━━━━━━━━━━━━━━━━━━ 5s 13ms/step - loss: 0.3803 - val_loss: 0.1906
Epoch 2/5
235/235 ━━━━━━━━━━━━━━━━━━━━ 5s 12ms/step - loss: 0.1809 - val_loss: 0.1533
Epoch 3/5
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - loss: 0.1490 - val_loss: 0.1333
Epoch 4/5
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - loss: 0.1312 - val_loss: 0.1215
Epoch 5/5
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 9ms/step - loss: 0.1207 - val_loss: 0.1138
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
```

Recurrent Neural Network

Practical 9

AIM : Demonstrate the recurrent neural network that learns to perform sequence analysis for stock price. (google stock price)

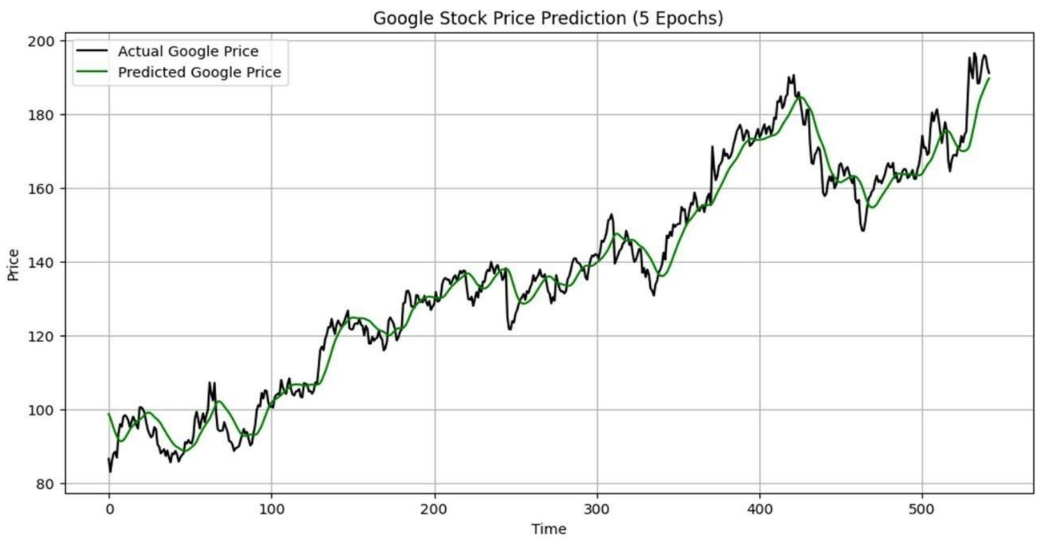
Description:
A **Recurrent Neural Network** is a type of neural network designed for **sequential or time-dependent data**.
LSTM (Long Short-Term Memory)
LSTM is a special kind of RNN that is **better at learning long-term dependencies**. It's designed to avoid problems like vanishing gradients, which standard RNNs struggle with.
Stock prices are time series data what happens today depends on what happened in the past. LSTMs can learn these dependencies and trends over time.
Use of LSTM in this practical:
Looks back at 60 days of prices to predict the next day's price Learns temporal patterns, trends, or even seasonal behavior Models time dependencies better than a standard neural network

Code:

```
import yfinance
as yf import
numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import
MinMaxScaler from tensorflow.keras.models
import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout, Input

ticker = "GOOGL"
data = yf.download(ticker, start="2014-01-01", end="2024-12-
31") close_prices = data['Close'].values.reshape(-1, 1)

scaler = MinMaxScaler(feature_range=(0, 1))
scaled_data =
scaler.fit_transform(close_prices)

def create_sequences(data,
  seq_length): x, y = [], []
  for i in range(seq_length, len(data)):
    x.append(data[i - seq_length:i])
    y.append(data[i])
  return np.array(x), np.array(y)

SEQ_LEN = 60
x, y = create_sequences(scaled_data, SEQ_LEN)
```

```
split_index = int(len(x) * 0.8)
x_train, x_test = x[:split_index],
   x[split_index:] y_train, y_test =
   y[:split_index], y[split_index:]
model =
   Sequential([ Input(shape=(SEQ_LE
   N, 1)), LSTM(50,
   return_sequences=True),
   Dropout(0.2),
   LSTM(50),
   Dropout(0.2),
   Dense(25),
   Dense(1)
])
model.compile(optimizer='adam',
loss='mean_squared_error') model.summary()

history = model.fit(x_train, y_train, epochs=5, batch_size=32, validation_data=(x_test, y_test))

predicted = model.predict(x_test)
predicted_prices = scaler.inverse_transform(predicted)
actual_prices = scaler.inverse_transform(y_test)

plt.figure(figsize=(12,   6))
plt.plot(actual_prices, label='Actual Google Price', color='black')
plt.plot(predicted_prices, label='Predicted Google Price', color='green')
plt.title('Google Stock Price Prediction (5 Epochs)')
plt.xlabel('Time')
plt.ylabel('Price')
plt.legend()
plt.grid(True)
plt.show()
```

Output:



Generative Adversarial Network (GAN)

Practical 10

Aim: Applying Generative Adversarial Networks for image generation and unsupervised tasks.

Description: Generative Adversarial Networks (GANs) are a class of deep learning models introduced by Ian Goodfellow in 2014, composed of two neural networks — the generator and the discriminator
— which compete in a minimax game. The generator attempts to produce synthetic data that is indistinguishable from real data, while the discriminator tries to distinguish real data from generated data. This adversarial process improves the quality of the generated outputs over time.

In this project, GANs are applied primarily to:

1. **Image Generation:** Generating high-resolution and photorealistic images from random noise, using models like DCGAN (Deep Convolutional GAN), CycleGAN, or StyleGAN.
2. **Unsupervised Learning Tasks:** Utilizing GANs to learn useful representations of data without labeled examples. For instance, features learned by the discriminator can be transferred to downstream tasks such as classification or clustering. GANs also enable data augmentation, which improves the performance of models trained on limited real-world data.

By leveraging unsupervised capabilities, GANs demonstrate potential in scenarios where labeled data is scarce or unavailable, making them powerful tools for both generative modeling and unsupervised feature learning

Code:
```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np

# Set device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Hyperparameters
latent_dim = 100 # Dimension of the noise
vector batch_size = 64
epochs = 5
lr = 0.0002
beta1 = 0.5
```

```
        # Define the Generator
        Model class
        Generator(nn.Module):
        def   init  (self):
        super(Generator, self).__init__()
        self.fc = nn.Sequential(
            nn.Linear(latent_dim, 256),
            nn.ReLU(True),
            nn.Linear(256, 512),
            nn.ReLU(True),
            nn.Linear(512, 1024),
            nn.ReLU(True),
            nn.Linear(1024, 28*28), # Output a 28x28 image (MNIST
            size) nn.Tanh()
        )
    def forward(self, z):
        return self.fc(z).view(z.size(0), 1, 28, 28)  # Reshape to image

# Define the Discriminator Model
class Discriminator(nn.Module):
    def _init_(self):
        super(Discriminator, self).__init__()
        self.fc = nn.Sequential(
            nn.Flatten(),
            nn.Linear(28*28, 1024),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(1024, 512),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(256, 1),
            nn.Sigmoid()
        )

    def forward(self, x):
        return self.fc(x)

# Initialize models
generator = Generator().to(device)
discriminator = Discriminator().to(device)

# Optimizers
optimizer_G = optim.Adam(generator.parameters(), lr=lr, betas=(beta1, 0.999))
optimizer_D = optim.Adam(discriminator.parameters(), lr=lr, betas=(beta1, 0.999))
# Loss functioncriterion
= nn.BCELoss()
# Load dataset (MNIST in this case)
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize([0.5], [0.5])])
trainset = torchvision.datasets.MNIST(root='./data', train=True, download=True,
transform=transform)
trainloader = DataLoader(trainset, batch_size=batch_size, shuffle=True)
```

```
# Train the
GAN # Train
the GAN
for epoch in range(epochs):
    for i, (real_images, _) in enumerate(trainloader):
        batch_size = real_images.size(0)  # Dynamically get the batch size
        # Move real images to device
        real_images = real_images.to(device)

        # Create labels for real and fake data with dynamic batch
        size real_labels = torch.ones(batch_size, 1).to(device)
        fake_labels = torch.zeros(batch_size, 1).to(device)

        # Train Discriminator
        optimizer_D.zero_grad()

        # Real images
        real_outputs = discriminator(real_images)
        d_loss_real = criterion(real_outputs, real_labels)

        # Fake images
        z = torch.randn(batch_size, latent_dim).to(device) # Ensure batch size is same as real
        images fake_images = generator(z)
        fake_outputs = discriminator(fake_images.detach())
        d_loss_fake  =  criterion(fake_outputs,  fake_labels)

        # Total Discriminator loss
        d_loss = d_loss_real + d_loss_fake
        d_loss.backward()
        optimizer_D.step()

        # Train Generator
        optimizer_G.zero_grad()

        # Generate fake images and try to fool the discriminator
        fake_outputs = discriminator(fake_images)
        g_loss = criterion(fake_outputs, real_labels)  # Want the fake images to be classified as real

        g_loss.backward()
        optimizer_G.step()

        # Print loss
        if i % 100 == 0:
            print(f"Epoch [{epoch+1}/{epochs}], Step [{i+1}/{len(trainloader)}], "
                  f"D Loss: {d_loss.item():.4f}, G Loss: {g_loss.item():.4f}")

    # Generate and save images every epoch
    with torch.no_grad():
        z = torch.randn(64, latent_dim).to(device)
        fake_images = generator(z)
        fake_images = fake_images.cpu().detach()
        grid = torchvision.utils.make_grid(fake_images, nrow=8, normalize=True)
        plt.figure(figsize=(8,8))
        plt.imshow(grid.permute(1, 2, 0))
        plt.axis('off')
        plt.savefig(f"generated_images_epoch_{epoch+1}.png")
 print("Trainingcomplete!")
```

Output: