Practical 1

Aim: Implement Feed-forward Neural Network and train the network with different optimizers and compare the results.

Theory: This program implements a feed-forward neural network and trains it using various optimizers. By comparing the results obtained from different optimizers, it allows for an evaluation of their effectiveness in optimizing the network's parameters and improving training performance.

Implementation:

```
#Import necessary libraries
import tensorflow as tf
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelBinarizer

# Load Iris dataset
iris = load_iris()

# Get features and output
X = iris.data
y = iris.target

# One-hot encode labels
lb = LabelBinarizer()
y = lb.fit_transform(y)

# Split data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Define model architecture
model = tf.keras.Sequential([
tf.keras.layers.Dense(16, input_shape=(4,), activation='relu'),
tf.keras.layers.Dense(8, activation='relu'),
tf.keras.layers.Dense(3, activation='softmax')])

# Define a list of optimizers to use
optimizers = ['sgd', 'adam', 'rmsprop']

# Loop over each optimizer and compile, train, and evaluate the model
for optimizer in optimizers:
  model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['accuracy'])

  # Train the model
  history = model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=50, verbose=0)

  # Evaluate the model on the test set
  loss, accuracy = model.evaluate(X_test, y_test, verbose=0)
```

```
    # Print the optimizer, test loss, and test accuracy
    print('Optimizer:', optimizer)
    print('Test loss:', loss)
    print('Test accuracy:', accuracy)
```

```
Optimizer: sgd
Test loss: 0.5112755298614502
Test accuracy: 0.8333333134651184
Optimizer: adam
Test loss: 0.32541367411613464
Test accuracy: 0.9666666388511658
Optimizer: rmsprop
Test loss: 0.21086803078651428
Test accuracy: 0.9666666388511658
```

```
#Import necessary libraries
import tensorflow as tf
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelBinarizer

# Load Iris dataset
iris = load_iris()
X = iris.data
y = iris.target

# One-hot encode labels
lb = LabelBinarizer()
y = lb.fit_transform(y)

# Split data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Define model architecture
model = tf.keras.Sequential([
    tf.keras.layers.Dense(16, input_shape=(4,), activation='relu'),
    tf.keras.layers.Dense(8, activation='relu'),
    tf.keras.layers.Dense(3, activation='softmax')])

# Compile model with different optimizers
optimizers = ['sgd', 'adam', 'rmsprop']
for optimizer in optimizers:
    model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['accuracy'])
    history = model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=50, verbose=0)

    # Evaluate model
    loss, accuracy = model.evaluate(X_test, y_test, verbose=0)
    print('Optimizer:', optimizer)
    print('Test loss:', loss)
    print('Test accuracy:', accuracy)

# Allow user to input values for the flower attributes
print('\nInput values for the flower attributes:')
sepal_length = float(input('Sepal length (cm): '))
sepal_width = float(input('Sepal width (cm): '))
petal_length = float(input('Petal length (cm): '))
petal_width = float(input('Petal width (cm): '))
```

```
# Predict class of flower based on input values
input_values = np.array([[sepal_length, sepal_width, petal_length, petal_width]])
prediction = model.predict(input_values)
predicted_class = np.argmax(prediction)
class_names = iris.target_names
print('\nPredicted class: ', class_names[predicted_class])
```

```
Optimizer: sgd
Test loss: 0.604692280292511
Test accuracy: 0.8333333134651184
Optimizer: adam
Test loss: 0.3206736743450165
Test accuracy: 0.9666666388511658
Optimizer: rmsprop
Test loss: 0.20026826858520508
Test accuracy: 0.9666666388511658

Input values for the flower attributes:
Sepal length (cm): 10
Sepal width (cm): 40
Petal length (cm): 30
Petal width (cm): 8

Predicted class:  versicolor
```

```
#Import necessary libraries
import tensorflow as tf
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelBinarizer

# Load Iris dataset
iris = load_iris()
X = iris.data
y = iris.target

# One-hot encode labels
lb = LabelBinarizer()
y = lb.fit_transform(y)

# Split data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Define model architecture
model = tf.keras.Sequential([
    tf.keras.layers.Dense(16, input_shape=(4,), activation='relu'),
    tf.keras.layers.Dense(8, activation='relu'),
    tf.keras.layers.Dense(3, activation='softmax')
])

# Define dictionary of optimizers
optimizers = {
    'sgd': tf.keras.optimizers.SGD(),
    'adam': tf.keras.optimizers.Adam(),
    'rmsprop': tf.keras.optimizers.RMSprop()
}
```

```
# Compile model with different optimizers
for optimizer_name, optimizer in optimizers.items():
    model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['accuracy'])

    # Train model
    history = model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=50, verbose=0)


    # Evaluate model
    loss, accuracy = model.evaluate(X_test, y_test, verbose=0)
    print('Optimizer:', optimizer_name)
    print('Test loss:', loss)
    print('Test accuracy:', accuracy)


    # Estimate memory requirement
    size_in_bytes = model.count_params() * 4  # each parameter is a 32-bit float
    size_in_mb = size_in_bytes / (1024 * 1024)
    print(f'Memory requirement: {size_in_mb:.2f} MB')
```

```
Optimizer: sgd
Test loss: 0.4390714466571808
Test accuracy: 0.9333333373069763
Memory requirement: 0.00 MB
Optimizer: adam
Test loss: 0.2147025465965271
Test accuracy: 0.9666666388511658
Memory requirement: 0.00 MB
Optimizer: rmsprop
Test loss: 0.13673563301563263
Test accuracy: 0.9666666388511658
Memory requirement: 0.00 MB
```

Practical 2

Aim: Write a Program to implement regularization to prevent the model from overfitting.

Theory: This program implements regularization techniques to mitigate overfitting in a machine learning model. Regularization methods, such as L1 or L2 regularization, are applied to the model's parameters to prevent it from becoming too complex and overly specialized to the training data, thereby improving generalization to unseen data.

Implementation:

```
#Import necessary libraries
import tensorflow as tf

# Load the data
(train_data, train_labels), (test_data, test_labels) = tf.keras.datasets.mnist.load_data()

# Preprocess the data
train_data = train_data.reshape((60000, 784)) / 255.0
test_data = test_data.reshape((10000, 784)) / 255.0
train_labels = tf.keras.utils.to_categorical(train_labels)
test_labels = tf.keras.utils.to_categorical(test_labels)

# Define the model architecture
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(128, activation='relu', input_shape=(784,),
kernel_regularizer=tf.keras.regularizers.l2(0.01)),
    tf.keras.layers.Dense(64, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(0.01)),
    tf.keras.layers.Dense(10, activation='softmax')
])

# Compile the model
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Train the model
history = model.fit(train_data, train_labels,
                    epochs=10,
                    batch_size=128,
                    validation_data=(test_data, test_labels))
```

```
Epoch 1/10
469/469 [==============================] - 5s 7ms/step - loss: 1.1288 - accuracy: 0.8836 - val_loss: 0.6104 - val_accuracy: 0.9
198
Epoch 2/10
469/469 [==============================] - 3s 6ms/step - loss: 0.5622 - accuracy: 0.9208 - val_loss: 0.5081 - val_accuracy: 0.9
255
Epoch 3/10
469/469 [==============================] - 3s 6ms/step - loss: 0.4927 - accuracy: 0.9295 - val_loss: 0.4749 - val_accuracy: 0.9
322
Epoch 4/10
469/469 [==============================] - 3s 6ms/step - loss: 0.4571 - accuracy: 0.9351 - val_loss: 0.4272 - val_accuracy: 0.9
387
Epoch 5/10
469/469 [==============================] - 3s 6ms/step - loss: 0.4265 - accuracy: 0.9402 - val_loss: 0.4026 - val_accuracy: 0.9
459
Epoch 6/10
469/469 [==============================] - 3s 6ms/step - loss: 0.4074 - accuracy: 0.9426 - val_loss: 0.3906 - val_accuracy: 0.9
397
Epoch 7/10
469/469 [==============================] - 3s 6ms/step - loss: 0.3873 - accuracy: 0.9459 - val_loss: 0.3599 - val_accuracy: 0.9
514
Epoch 8/10
469/469 [==============================] - 3s 5ms/step - loss: 0.3689 - accuracy: 0.9488 - val_loss: 0.3557 - val_accuracy: 0.9
512
Epoch 9/10
469/469 [==============================] - 3s 5ms/step - loss: 0.3564 - accuracy: 0.9507 - val_loss: 0.3396 - val_accuracy: 0.9
532
Epoch 10/10
469/469 [==============================] - 3s 5ms/step - loss: 0.3443 - accuracy: 0.9520 - val_loss: 0.3432 - val_accuracy: 0.9
495
```

```
#Import necessary libraries
import tensorflow as tf

# Load the data
(train_data, train_labels), (test_data, test_labels) = tf.keras.datasets.mnist.load_data()
# The MNIST dataset contains 70,000 images of handwritten digits that are split into 60,000 training images
and 10,000 testing images.

# Preprocess the data
train_data = train_data.reshape((60000, 784)) / 255.0   # Reshape and normalize training data
test_data = test_data.reshape((10000, 784)) / 255.0     # Reshape and normalize testing data
train_labels = tf.keras.utils.to_categorical(train_labels)   # Convert training labels to one-hot encoding
test_labels = tf.keras.utils.to_categorical(test_labels)     # Convert testing labels to one-hot encoding

# Define the model architecture
model = tf.keras.models.Sequential([

    tf.keras.layers.Dense(128, activation='relu', input_shape=(784,),
kernel_regularizer=tf.keras.regularizers.l2(0.01)),
    tf.keras.layers.Dense(64, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(0.01)),
    tf.keras.layers.Dense(10, activation='softmax')
])

# Compile the model
model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),   # Use Adam optimizer with learning rate 0.001
    loss='categorical_crossentropy',    # Use categorical cross-entropy loss function
    metrics=['accuracy']    # Monitor accuracy during training
)

# In this case, the Adam optimizer is used with a learning rate of 0.001, categorical cross-entropy is used as the
loss function and accuracy is monitored during training.

# Train the model
history = model.fit(
    train_data,
    train_labels,
    epochs=10,
    batch_size=128,
    validation_data=(test_data, test_labels)
)

# The purpose of this program is to demonstrate how to implement a neural network model for image classification using
TensorFlow/Keras. The model uses regularization techniques to prevent overfitting and achieves high accuracy on the
MNIST dataset.
```

```
Epoch 1/10
469/469 [==============================] - 5s 9ms/step - loss: 1.1294 - accuracy: 0.8822 - val_loss: 0.6255 - val_accuracy: 0.9
163
Epoch 2/10
469/469 [==============================] - 4s 8ms/step - loss: 0.5564 - accuracy: 0.9220 - val_loss: 0.4978 - val_accuracy: 0.9
286
Epoch 3/10
469/469 [==============================] - 3s 7ms/step - loss: 0.4919 - accuracy: 0.9280 - val_loss: 0.4494 - val_accuracy: 0.9
352
Epoch 4/10
469/469 [==============================] - 3s 7ms/step - loss: 0.4527 - accuracy: 0.9349 - val_loss: 0.4111 - val_accuracy: 0.9
438
Epoch 5/10
469/469 [==============================] - 3s 6ms/step - loss: 0.4237 - accuracy: 0.9402 - val_loss: 0.4007 - val_accuracy: 0.9
440
Epoch 6/10
469/469 [==============================] - 3s 6ms/step - loss: 0.4016 - accuracy: 0.9430 - val_loss: 0.3787 - val_accuracy: 0.9
472
Epoch 7/10
469/469 [==============================] - 3s 6ms/step - loss: 0.3823 - accuracy: 0.9456 - val_loss: 0.3605 - val_accuracy: 0.9
507
Epoch 8/10
469/469 [==============================] - 3s 7ms/step - loss: 0.3664 - accuracy: 0.9484 - val_loss: 0.3422 - val_accuracy: 0.9
541
Epoch 9/10
469/469 [==============================] - 3s 7ms/step - loss: 0.3532 - accuracy: 0.9498 - val_loss: 0.3425 - val_accuracy: 0.9
503
Epoch 10/10
469/469 [==============================] - 3s 5ms/step - loss: 0.3422 - accuracy: 0.9516 - val_loss: 0.3256 - val_accuracy: 0.9
543
```

Practical 3

Aim: Implement deep learning for recognizing classes for datasets like CIFAR-10 images for previously unseen images and assign them to one of the 10 classes.

Theory: This program implements deep learning algorithms, such as convolutional neural networks (CNNs), to classify previously unseen images from datasets like CIFAR-10 into one of the ten predefined classes. By training on labeled data, the model learns to extract meaningful features and make accurate predictions, enabling effective image classification.

Implementation:

```
#Import necessary libraries
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import numpy as np
from PIL import Image

# Load the CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()

# Preprocess the data by scaling pixel values to the range [0, 1]
x_train = x_train.astype("float32") / 255.0
x_test = x_test.astype("float32") / 255.0

# Convert the labels to one-hot encoded vectors
y_train = keras.utils.to_categorical(y_train, 10)
y_test = keras.utils.to_categorical(y_test, 10)

# Define the model architecture
model = keras.Sequential(
    [
        keras.Input(shape=(32, 32, 3)),
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Flatten(),
        layers.Dropout(0.5),
        layers.Dense(10, activation="softmax"),
    ]
)

# Compile the model with categorical cross-entropy loss and the Adam optimizer
model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])

# Train the model on the CIFAR-10 dataset
model.fit(x_train, y_train, batch_size=64, epochs=10, validation_data=(x_test, y_test))

# Save the model to a file
model.save("cifar10_model.h5")


# Load the saved model
model = keras.models.load_model("cifar10_model.h5")
```

```
# Load and preprocess the test image
img = Image.open("two.png")
img = img.resize((32, 32))
img_array = np.array(img)
img_array = img_array.astype("float32") / 255.0
img_array = np.expand_dims(img_array, axis=0)

# Make predictions on the test image
predictions = model.predict(img_array)

# Get the predicted class label
class_label = np.argmax(predictions)

# Print the predicted class label
print("Predicted class label:", class_label)
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170500096/170498071 [==============================] - 1109s 7us/step
170508288/170498071 [==============================] - 1109s 7us/step
Epoch 1/10
782/782 [==============================] - 89s 112ms/step - loss: 1.6155 - accuracy: 0.4162 - val_loss: 1.3592 - val_accuracy:
0.5245
Epoch 2/10
782/782 [==============================] - 86s 110ms/step - loss: 1.3119 - accuracy: 0.5348 - val_loss: 1.1603 - val_accuracy:
0.5993
Epoch 3/10
782/782 [==============================] - 79s 102ms/step - loss: 1.1933 - accuracy: 0.5802 - val_loss: 1.0791 - val_accuracy:
0.6307
Epoch 4/10
782/782 [==============================] - 84s 107ms/step - loss: 1.1242 - accuracy: 0.6062 - val_loss: 1.0426 - val_accuracy:
0.6434
Epoch 5/10
782/782 [==============================] - 77s 99ms/step - loss: 1.0796 - accuracy: 0.6245 - val_loss: 0.9905 - val_accuracy:
0.6615
Epoch 6/10
782/782 [==============================] - 80s 102ms/step - loss: 1.0402 - accuracy: 0.6376 - val_loss: 0.9876 - val_accuracy:
0.6544
Epoch 7/10
782/782 [==============================] - 81s 104ms/step - loss: 1.0102 - accuracy: 0.6492 - val_loss: 0.9556 - val_accuracy:
0.6722
Epoch 8/10
782/782 [==============================] - 81s 104ms/step - loss: 0.9927 - accuracy: 0.6570 - val_loss: 0.9164 - val_accuracy:
0.6932
Epoch 9/10
782/782 [==============================] - 100s 128ms/step - loss: 0.9651 - accuracy: 0.6645 - val_loss: 0.9007 - val_accuracy
0.6932
Epoch 10/10
782/782 [==============================] - 100s 127ms/step - loss: 0.9417 - accuracy: 0.6730 - val_loss: 0.9026 - val_accuracy
0.6921
Predicted class label: 0
```

```
#Import necessary libraries
import tensorflow as tf
from tensorflow import keras
import numpy as np
from PIL import Image

# Load the CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()

# Normalize the pixel values to be between 0 and 1
x_train = x_train.astype("float32") / 255.0
x_test = x_test.astype("float32") / 255.0

# Convert the labels to one-hot encoded vectors
y_train = keras.utils.to_categorical(y_train, num_classes=10)
y_test = keras.utils.to_categorical(y_test, num_classes=10)

# Define the model architecture
model = keras.models.Sequential([
    keras.layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),
    keras.layers.MaxPooling2D((2, 2)),
    keras.layers.Conv2D(64, (3, 3), activation='relu'),
    keras.layers.MaxPooling2D((2, 2)),
```

```
    keras.layers.Conv2D(64, (3, 3), activation='relu'),
    keras.layers.Flatten(),
    keras.layers.Dense(64, activation='relu'),
    keras.layers.Dense(10, activation='softmax')
])

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

# Train the model
history = model.fit(x_train, y_train, epochs=10, batch_size=64, validation_data=(x_test, y_test))

# Save the trained model to a file
model.save("cifar10_model.h5")

# Load the saved model
model = keras.models.load_model("cifar10_model.h5")

# Load and preprocess the test image
img = Image.open("two.png")
img = img.resize((32, 32))
img_array = np.array(img)
img_array = img_array.astype("float32") / 255.0
img_array = np.expand_dims(img_array, axis=0)

# Make predictions on the test image
predictions = model.predict(img_array)

# Get the predicted class label
class_label = np.argmax(predictions)

# Print the predicted class label
print("Predicted class label:", class_label)
```

```
Epoch 1/10
782/782 [==============================] - 134s 169ms/step - loss: 1.6046 - accuracy: 0.4103 - val_loss: 1.3309 - val_accuracy:
0.5205
Epoch 2/10
782/782 [==============================] - 115s 147ms/step - loss: 1.2525 - accuracy: 0.5541 - val_loss: 1.1820 - val_accuracy:
0.5786
Epoch 3/10
782/782 [==============================] - 106s 135ms/step - loss: 1.1099 - accuracy: 0.6079 - val_loss: 1.0913 - val_accuracy:
0.6136
Epoch 4/10
782/782 [==============================] - 105s 134ms/step - loss: 1.0141 - accuracy: 0.6444 - val_loss: 1.0678 - val_accuracy:
0.6210
Epoch 5/10
782/782 [==============================] - 114s 146ms/step - loss: 0.9434 - accuracy: 0.6699 - val_loss: 0.9590 - val_accuracy:
0.6680
Epoch 6/10
782/782 [==============================] - 114s 146ms/step - loss: 0.8897 - accuracy: 0.6880 - val_loss: 1.0224 - val_accuracy:
0.6455
Epoch 7/10
782/782 [==============================] - 96s 123ms/step - loss: 0.8405 - accuracy: 0.7045 - val_loss: 0.9590 - val_accuracy:
0.6697
Epoch 8/10
782/782 [==============================] - 101s 130ms/step - loss: 0.7953 - accuracy: 0.7194 - val_loss: 0.8865 - val_accuracy:
0.6934
Epoch 9/10
782/782 [==============================] - 96s 123ms/step - loss: 0.7605 - accuracy: 0.7334 - val_loss: 0.8663 - val_accuracy:
0.7023
Epoch 10/10
782/782 [==============================] - 98s 125ms/step - loss: 0.7262 - accuracy: 0.7448 - val_loss: 0.8965 - val_accuracy:
0.6994
Predicted class label: 2
```

Practical 4

Aim: Implement deep learning for the Prediction of the autoencoder from the test data (e.g., MNIST data set).

Theory: This program implements deep learning techniques, specifically an autoencoder, to predict and reconstruct the input data from the test dataset, such as the MNIST dataset. By training the autoencoder, it learns a compact representation of the data and generates predictions that aim to closely resemble the original input.

Implementation:

```
#Import necessary libraries
import tensorflow as tf
from tensorflow import keras
import numpy as np
import matplotlib.pyplot as plt

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

# Normalize the pixel values to be between 0 and 1
x_train = x_train.astype("float32") / 255.0
x_test = x_test.astype("float32") / 255.0

# Define the encoder architecture
encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(128, activation="relu"),
    keras.layers.Dense(64, activation="relu"),
    keras.layers.Dense(32, activation="relu"),
])

# Define the decoder architecture
decoder = keras.models.Sequential([
    keras.layers.Dense(64, activation="relu", input_shape=[32]),
    keras.layers.Dense(128, activation="relu"),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28]),
])

# Combine the encoder and decoder into an autoencoder model
autoencoder = keras.models.Sequential([encoder, decoder])

# Compile the autoencoder model
autoencoder.compile(loss="binary_crossentropy", optimizer=keras.optimizers.Adam(learning_rate=0.001))

# Train the autoencoder model
history = autoencoder.fit(x_train, x_train, epochs=10, batch_size=128, validation_data=(x_test, x_test))

# Use the trained autoencoder to predict the reconstructed images for the test data
decoded_imgs = autoencoder.predict(x_test)
```

```
# Plot some of the original test images and their reconstructed counterparts
# Number of images to display
n = 10
plt.figure(figsize=(20, 4))

for i in range(n):
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i])
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # Display reconstructed images
    ax = plt.subplot(2, n, i + n + 1)
    plt.imshow(decoded_imgs[i])
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```

```
Epoch 1/10
469/469 [==============================] - 11s 20ms/step - loss: 0.2035 - val_loss: 0.1434
Epoch 2/10
469/469 [==============================] - 8s 16ms/step - loss: 0.1324 - val_loss: 0.1216
Epoch 3/10
469/469 [==============================] - 6s 13ms/step - loss: 0.1177 - val_loss: 0.1123
Epoch 4/10
469/469 [==============================] - 7s 15ms/step - loss: 0.1101 - val_loss: 0.1058
Epoch 5/10
469/469 [==============================] - 8s 18ms/step - loss: 0.1050 - val_loss: 0.1018
Epoch 6/10
469/469 [==============================] - 8s 18ms/step - loss: 0.1019 - val_loss: 0.0997
Epoch 7/10
469/469 [==============================] - 9s 19ms/step - loss: 0.0999 - val_loss: 0.0976
Epoch 8/10
469/469 [==============================] - 14s 29ms/step - loss: 0.0982 - val_loss: 0.0960
Epoch 9/10
469/469 [==============================] - 11s 23ms/step - loss: 0.0964 - val_loss: 0.0944
Epoch 10/10
469/469 [==============================] - 9s 20ms/step - loss: 0.0948 - val_loss: 0.0935
```

Practical 5

Aim: Implement Convolutional Neural Network for Digit Recognition on the MNIST Dataset.

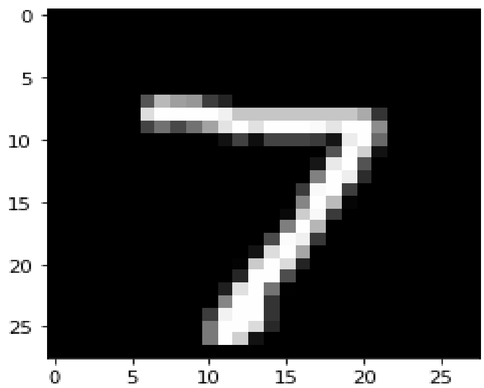
Theory: This program implements a Convolutional Neural Network (CNN) for digit recognition on the MNIST dataset. It utilizes specialized layers like convolutional and pooling layers to effectively extract features from the images and achieve accurate classification of handwritten digits.

Implementation:

```
#Import necessary libraries
import tensorflow as tf
from tensorflow import keras
import numpy as np
import matplotlib.pyplot as plt

# Load the MNIST dataset
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

# Preprocess the data
x_train = x_train.astype("float32") / 255.0
x_test = x_test.astype("float32") / 255.0
x_train = np.expand_dims(x_train, -1)
x_test = np.expand_dims(x_test, -1)

# Define the CNN architecture
model = keras.models.Sequential([
    keras.layers.Conv2D(32, (3, 3), activation="relu", input_shape=(28, 28, 1)),
    keras.layers.MaxPooling2D((2, 2)),
    keras.layers.Conv2D(64, (3, 3), activation="relu"),
    keras.layers.MaxPooling2D((2, 2)),
    keras.layers.Flatten(),
    keras.layers.Dense(64, activation="relu"),
    keras.layers.Dense(10, activation="softmax")
])

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

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

# Evaluate the model on the test data
test_loss, test_acc = model.evaluate(x_test, y_test)
print("Test accuracy:", test_acc)
```

```
# Show predictions for a sample input image
sample_img = x_test[0]
sample_label = y_test[0]
sample_img = np.expand_dims(sample_img, 0)
pred = model.predict(sample_img)
pred_label = np.argmax(pred)
print("Sample image true label:", sample_label)
print("Sample image predicted label:", pred_label)

# Display the sample image
plt.imshow(sample_img.squeeze(), cmap='gray')
plt.show()
```

```
Epoch 1/10
469/469 [==============================] - 82s 170ms/step - loss: 0.2289 - accuracy: 0.9330 - val_loss: 0.0650 - val_accuracy:
0.9799
Epoch 2/10
469/469 [==============================] - 72s 153ms/step - loss: 0.0663 - accuracy: 0.9796 - val_loss: 0.0452 - val_accuracy:
0.9856
Epoch 3/10
469/469 [==============================] - 74s 157ms/step - loss: 0.0445 - accuracy: 0.9864 - val_loss: 0.0357 - val_accuracy:
0.9890
Epoch 4/10
469/469 [==============================] - 82s 175ms/step - loss: 0.0344 - accuracy: 0.9895 - val_loss: 0.0360 - val_accuracy:
0.9886
Epoch 5/10
469/469 [==============================] - 71s 152ms/step - loss: 0.0278 - accuracy: 0.9916 - val_loss: 0.0288 - val_accuracy:
0.9899
Epoch 6/10
469/469 [==============================] - 71s 151ms/step - loss: 0.0226 - accuracy: 0.9931 - val_loss: 0.0320 - val_accuracy:
0.9899
Epoch 7/10
469/469 [==============================] - 71s 152ms/step - loss: 0.0196 - accuracy: 0.9937 - val_loss: 0.0291 - val_accuracy:
0.9907
Epoch 8/10
469/469 [==============================] - 70s 149ms/step - loss: 0.0140 - accuracy: 0.9955 - val_loss: 0.0275 - val_accuracy:
0.9910
Epoch 9/10
469/469 [==============================] - 69s 146ms/step - loss: 0.0135 - accuracy: 0.9958 - val_loss: 0.0360 - val_accuracy:
0.9898
Epoch 10/10
469/469 [==============================] - 65s 139ms/step - loss: 0.0107 - accuracy: 0.9962 - val_loss: 0.0347 - val_accuracy:
0.9903
313/313 [==============================] - 5s 16ms/step - loss: 0.0347 - accuracy: 0.9903
Test accuracy: 0.9902999997138977
Sample image true label: 7
Sample image predicted label: 7
```



Practical 6

Aim: Write a program to implement Transfer Learning on the suitable dataset (e.g., classify the cats versus dog's dataset from Kaggle).

Theory: This program applies transfer learning on a suitable dataset, such as the cats versus dog's dataset from Kaggle, to leverage pre-trained models and fine-tune them for accurate classification. It utilizes the knowledge learned from a large dataset to solve a similar task, achieving efficient and effective classification of cats and dogs.

Implementation:

```
#Import necessary libraries
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import os
import zipfile
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.applications import VGG16

# Download and extract dataset
url = "https://storage.googleapis.com/mledu-datasets/cats_and_dogs_filtered.zip"
filename = os.path.join(os.getcwd(), "cats_and_dogs_filtered.zip")
tf.keras.utils.get_file(filename, url)
with zipfile.ZipFile("cats_and_dogs_filtered.zip", "r") as zip_ref:
    zip_ref.extractall()

# Define data generators
train_dir = os.path.join(os.getcwd(), "cats_and_dogs_filtered", "train")
validation_dir = os.path.join(os.getcwd(), "cats_and_dogs_filtered","validation")
train_datagen = ImageDataGenerator(rescale=1./255,rotation_range=20,width_shift_range=0.2,
height_shift_range=0.2,shear_range=0.2,zoom_range=0.2,horizontal_flip=True)
validation_datagen = ImageDataGenerator(rescale=1./255)
train_generator = train_datagen.flow_from_directory(train_dir,target_size=(150,150),batch_size=20,
class_mode="binary")
validation_generator = validation_datagen.flow_from_directory(validation_dir,target_size=(150,150),
batch_size=20,class_mode="binary")
```

```
Found 2000 images belonging to 2 classes.
Found 1000 images belonging to 2 classes.
```

```
# Load pre-trained VGG16 model
conv_base = VGG16(weights="imagenet",include_top=False,input_shape=(150, 150, 3))

# Freeze convolutional base layers
conv_base.trainable = False

# Build model on top of the convolutional base
model = tf.keras.models.Sequential()
model.add(conv_base)
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(256, activation="relu"))
model.add(tf.keras.layers.Dropout(0.5))
model.add(tf.keras.layers.Dense(1, activation="sigmoid"))
```

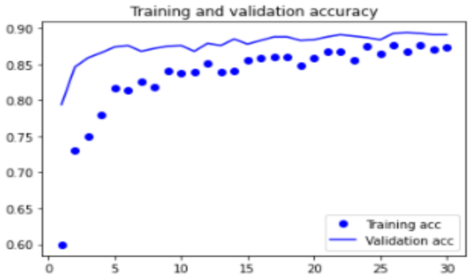
Compile model
model.compile(loss="binary_crossentropy",
optimizer=tf.keras.optimizers.RMSprop(learning_rate=2e-5),metrics=["accuracy"])

# *Train model*
history = model.fit(train_generator,steps_per_epoch=100,epochs=30,validation_data=validation_generator,
validation_steps=50)

```
Epoch 1/30
100/100 [==============================] - 1076s 11s/step - loss: 0.6879 - accuracy: 0.5995 - val_loss: 0.4976 - val_accuracy:
0.7940
Epoch 2/30
100/100 [==============================] - 941s 9s/step - loss: 0.5463 - accuracy: 0.7305 - val_loss: 0.4099 - val_accuracy: 0.
8460
Epoch 3/30
100/100 [==============================] - 761s 8s/step - loss: 0.4932 - accuracy: 0.7505 - val_loss: 0.3661 - val_accuracy: 0.
8590
Epoch 4/30
100/100 [==============================] - 828s 8s/step - loss: 0.4573 - accuracy: 0.7800 - val_loss: 0.3403 - val_accuracy: 0.
8660
Epoch 5/30
100/100 [==============================] - 851s 9s/step - loss: 0.4208 - accuracy: 0.8175 - val_loss: 0.3220 - val_accuracy: 0.
8740
Epoch 6/30
100/100 [==============================] - 839s 8s/step - loss: 0.4000 - accuracy: 0.8145 - val_loss: 0.3098 - val_accuracy: 0.
8760
Epoch 7/30
100/100 [==============================] - 832s 8s/step - loss: 0.3845 - accuracy: 0.8265 - val_loss: 0.3031 - val_accuracy: 0.
8680
Epoch 8/30
100/100 [==============================] - 786s 8s/step - loss: 0.3976 - accuracy: 0.8190 - val_loss: 0.2957 - val_accuracy: 0.
8720
Epoch 9/30
100/100 [==============================] - 808s 8s/step - loss: 0.3658 - accuracy: 0.8400 - val_loss: 0.2895 - val_accuracy: 0.
8750
Epoch 10/30
100/100 [==============================] - 811s 8s/step - loss: 0.3569 - accuracy: 0.8375 - val_loss: 0.2866 - val_accuracy: 0.
8760
Epoch 11/30
100/100 [==============================] - 771s 8s/step - loss: 0.3582 - accuracy: 0.8390 - val_loss: 0.2950 - val_accuracy: 0.
8680
Epoch 12/30
100/100 [==============================] - 773s 8s/step - loss: 0.3456 - accuracy: 0.8505 - val_loss: 0.2853 - val_accuracy: 0.
8790
Epoch 13/30
100/100 [==============================] - 1080s 11s/step - loss: 0.3429 - accuracy: 0.8395 - val_loss: 0.2743 - val_accuracy:
0.8760
Epoch 14/30
100/100 [==============================] - 1117s 11s/step - loss: 0.3490 - accuracy: 0.8400 - val_loss: 0.2746 - val_accuracy:
0.8850
Epoch 15/30
100/100 [==============================] - 919s 9s/step - loss: 0.3236 - accuracy: 0.8555 - val_loss: 0.2824 - val_accuracy: 0.
8780
Epoch 16/30
100/100 [==============================] - 798s 8s/step - loss: 0.3266 - accuracy: 0.8590 - val_loss: 0.2679 - val_accuracy: 0.
8830
Epoch 17/30
100/100 [==============================] - 781s 8s/step - loss: 0.3261 - accuracy: 0.8605 - val_loss: 0.2679 - val_accuracy: 0.
8880
Epoch 18/30
100/100 [==============================] - 774s 8s/step - loss: 0.3236 - accuracy: 0.8595 - val_loss: 0.2650 - val_accuracy: 0.
8880
Epoch 19/30
100/100 [==============================] - 822s 8s/step - loss: 0.3255 - accuracy: 0.8485 - val_loss: 0.2704 - val_accuracy: 0.
8830
Epoch 20/30
100/100 [==============================] - 814s 8s/step - loss: 0.3166 - accuracy: 0.8590 - val_loss: 0.2662 - val_accuracy: 0.
8840
Epoch 21/30
100/100 [==============================] - 844s 8s/step - loss: 0.3141 - accuracy: 0.8680 - val_loss: 0.2590 - val_accuracy: 0.
8880
Epoch 22/30
100/100 [==============================] - 876s 9s/step - loss: 0.2981 - accuracy: 0.8680 - val_loss: 0.2680 - val_accuracy: 0.
8910
Epoch 23/30
100/100 [==============================] - 792s 8s/step - loss: 0.3047 - accuracy: 0.8555 - val_loss: 0.2549 - val_accuracy: 0.
8890
Epoch 24/30
100/100 [==============================] - 781s 8s/step - loss: 0.2975 - accuracy: 0.8745 - val_loss: 0.2597 - val_accuracy: 0.
8870
Epoch 25/30
100/100 [==============================] - 782s 8s/step - loss: 0.3119 - accuracy: 0.8640 - val_loss: 0.2826 - val_accuracy: 0.
8840
Epoch 26/30
100/100 [==============================] - 820s 8s/step - loss: 0.2896 - accuracy: 0.8760 - val_loss: 0.2565 - val_accuracy: 0.
8930
Epoch 27/30
100/100 [==============================] - 796s 8s/step - loss: 0.3009 - accuracy: 0.8675 - val_loss: 0.2554 - val_accuracy: 0.
8940
Epoch 28/30
100/100 [==============================] - 794s 8s/step - loss: 0.2863 - accuracy: 0.8770 - val_loss: 0.2547 - val_accuracy: 0.
8930
Epoch 29/30
100/100 [==============================] - 784s 8s/step - loss: 0.2982 - accuracy: 0.8705 - val_loss: 0.2537 - val_accuracy: 0.
8910
Epoch 30/30
100/100 [==============================] - 771s 8s/step - loss: 0.2904 - accuracy: 0.8730 - val_loss: 0.2558 - val_accuracy: 0.
8910
```

```
# Show sample input and its predicted class
x, y_true = next(validation_generator)
y_pred = model.predict(x)
class_names = ['cat', 'dog']
for i in range(len(x)):
   plt.imshow(x[i])
plt.title(f'Predicted class: {class_names[int(round(y_pred[i][0]))]}, True␣class: {class_names[int(y_true[i])]}')
plt.show()
```


```
# Plot accuracy and loss over time
acc = history.history["accuracy"]
val_acc = history.history["val_accuracy"]
loss = history.history["loss"]
val_loss = history.history["val_loss"]
epochs = range(1, len(acc) + 1)
plt.plot(epochs, acc, "bo", label="Training acc")
plt.plot(epochs, val_acc, "b", label="Validation acc")
plt.title("Training and validation accuracy")
plt.legend()
plt.figure()
plt.plot(epochs, loss, "bo", label="Training loss")
plt.plot(epochs, val_loss, "b", label="Validation loss")
plt.title("Training and validation loss")
plt.legend()
plt.show()
```



Practical 7

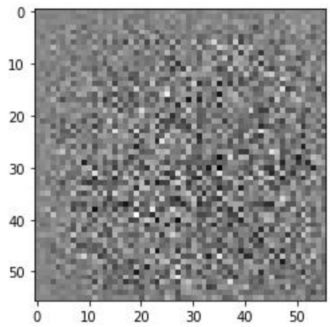
Aim: Write a program for the Implementation of a Generative Adversarial Network for generating synthetic shapes (like digits).

Theory: This program implements a Generative Adversarial Network (GAN) to generate synthetic shapes, such as digits. The GAN consists of a generator and a discriminator network that work together in a competitive manner to produce realistic and diverse synthetic shapes based on a given dataset.

Implementation:

```
#Import necessary libraries
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

# Load the MNIST dataset
(train_images, _), (_, _) = tf.keras.datasets.mnist.load_data()
train_images = train_images.reshape(train_images.shape[0], 28, 28, 1).astype('float32')
train_images = (train_images - 127.5) / 127.5 # Normalize the images to [-1, 1]

# Define the generator model
generator = tf.keras.Sequential([
tf.keras.layers.Dense(7*7*256, use_bias=False, input_shape=(100,)),
tf.keras.layers.BatchNormalization(),
tf.keras.layers.LeakyReLU(),
tf.keras.layers.Reshape((7, 7, 256)),
tf.keras.layers.Conv2DTranspose(128, (5, 5), strides=(1, 1),padding='same', use_bias=False),
tf.keras.layers.BatchNormalization(),
tf.keras.layers.LeakyReLU(),
tf.keras.layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same',use_bias=False),
tf.keras.layers.BatchNormalization(),
tf.keras.layers.LeakyReLU(),
tf.keras.layers.Conv2DTranspose(32, (5, 5), strides=(2, 2), padding='same',use_bias=False),
tf.keras.layers.BatchNormalization(),
tf.keras.layers.LeakyReLU(),
tf.keras.layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same',use_bias=False, activation='tanh')
])

# Define the discriminator model
discriminator = tf.keras.Sequential([
tf.keras.layers.Conv2D(32, (5, 5), strides=(2, 2), padding='same',input_shape=[28, 28, 1]),
tf.keras.layers.LeakyReLU(),
tf.keras.layers.Dropout(0.3),
tf.keras.layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same'),
tf.keras.layers.LeakyReLU(),
tf.keras.layers.Dropout(0.3),
tf.keras.layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'),
tf.keras.layers.LeakyReLU(),
tf.keras.layers.Dropout(0.3),
tf.keras.layers.Flatten(),
tf.keras.layers.Dense(1)
])
```

```
# Define the loss functions and optimizers
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)
def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    total_loss = real_loss + fake_loss
    return total_loss
def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)
generator_optimizer = tf.keras.optimizers.Adam(1e-4)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)

# Define the training loop
EPOCHS = 50
noise_dim = 100
num_examples_to_generate = 16
seed = tf.random.normal([num_examples_to_generate, noise_dim])
@tf.function
def train_step(images):
    noise = tf.random.normal([BATCH_SIZE, noise_dim])
    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        generated_images = generator(noise, training=True)
        real_output = discriminator(images, training=True)
        fake_output = discriminator(generated_images, training=True)
        gen_loss = generator_loss(fake_output)
        disc_loss = discriminator_loss(real_output, fake_output)
        gradients_of_generator = gen_tape.gradient(gen_loss, generator.trainable_variables)
        gradients_of_discriminator = disc_tape.gradient(disc_loss, discriminator.trainable_variables)
        generator_optimizer.apply_gradients(zip(gradients_of_generator, generator.trainable_variables))
# Apply gradients to the discriminator variables
        discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator,discriminator.trainable_variables))
# Train the generator
    with tf.GradientTape() as gen_tape:
# Generate fake images using the generator
        generated_images = generator(noise, training=True)
# Get discriminator's prediction of the generated images
        gen_preds = discriminator(generated_images, training=False)
# Calculate generator's loss
        gen_loss = generator_loss(gen_preds)
# Get gradients of the generator loss with respect to the generator  variables
        gradients_of_generator = gen_tape.gradient(gen_loss, generator.trainable_variables)
# Apply gradients to the generator variables
        generator_optimizer.apply_gradients(zip(gradients_of_generator, generator.trainable_variables))
# Print the losses
        print("Discriminator loss:", disc_loss.numpy(), "Generator loss:", gen_loss.numpy())
# Save checkpoint
        ckpt_manager.save()
# Generate and save 10 random images from the generator after training
        NOISE_DIM = 100

for i in range(10):
    noise = tf.random.normal([1, noise_dim])
    generated_images = generator(noise, training=False)
    img = tf.squeeze(generated_images[0])
    plt.imshow(img, cmap='gray')
    plt.savefig(f'generated_image_{i}.png')
```




```
#Import Necessary libraries
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

# Check if TensorFlow is able to detect a GPU
print(tf.config.list_physical_devices('GPU'))

# Set the GPU device to use
device_name = '/device:GPU:0'
mnist = tf.keras.datasets.mnist
(train_images, train_labels), (_, _) = mnist.load_data()

# Normalize the images to [-1, 1]
train_images = (train_images.astype('float32') - 127.5) / 127.5

# Reshape the images to (28, 28, 1) and add a channel dimension
train_images = np.expand_dims(train_images, axis=-1)

# Batch and shuffle the data
BUFFER_SIZE = 60000
BATCH_SIZE = 256
train_dataset = tf.data.Dataset.from_tensor_slices(train_images).shuffle(BUFFER_SIZE).batch(BATCH_SIZE)

def make_generator_model():
    model = tf.keras.Sequential()
    model.add(tf.keras.layers.Dense(7*7*256, use_bias=False, input_shape=(100,)))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.LeakyReLU())
    model.add(tf.keras.layers.Reshape((7, 7, 256)))
    assert model.output_shape == (None, 7, 7, 256)
    model.add(tf.keras.layers.Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same', use_bias=False))
    assert model.output_shape == (None, 7, 7, 128)
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.LeakyReLU())
    model.add(tf.keras.layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same', use_bias=False))
    assert model.output_shape == (None, 14, 14, 64)
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.LeakyReLU())
    model.add(tf.keras.layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same', use_bias=False,
activation='tanh'))
    assert model.output_shape == (None, 28, 28, 1)
    return model
```

```
def make_discriminator_model():
    model = tf.keras.Sequential()
    model.add(tf.keras.layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same', input_shape=[28, 28, 1]))
    model.add(tf.keras.layers.LeakyReLU())
    model.add(tf.keras.layers.Dropout(0.3))
    model.add(tf.keras.layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
    model.add(tf.keras.layers.LeakyReLU())
    model.add(tf.keras.layers.Dropout(0.3))
    model.add(tf.keras.layers.Flatten())
    model.add(tf.keras.layers.Dense(1))
    return model
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)

def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    total_loss = real_loss + fake_loss
    return total_loss

def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)

# Define the models
generator = make_generator_model()
discriminator = make_discriminator_model()

# Define the optimizers
generator_optimizer = tf.keras.optimizers.Adam(1e-4)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)

# Define the training loop
EPOCHS = 100
noise_dim = 100
num_examples_to_generate = 16

@tf.function
def train_step(images):
    noise = tf.random.normal([BATCH_SIZE, noise_dim])

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        generated_images = generator(noise, training=True)

        # Evaluate discriminator on real and fake images
        real_output = discriminator(images, training=True)
        fake_output = discriminator(generated_images, training=True)

        # Calculate the losses
        gen_loss = generator_loss(fake_output)
        disc_loss = discriminator_loss(real_output, fake_output)

        # Calculate the gradients and apply them
        gradients_of_generator = gen_tape.gradient(gen_loss, generator.trainable_variables)
        gradients_of_discriminator = disc_tape.gradient(disc_loss, discriminator.trainable_variables)
        generator_optimizer.apply_gradients(zip(gradients_of_generator, generator.trainable_variables))
        discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator,discriminator.trainable_variables))

@tf.function
def generate_and_save_images(model, epoch, test_input):
```

```
    predictions = model(test_input, training=False)

    # Rescale to [0, 1]
    predictions = (predictions + 1) / 2.0

    # Plot the images
    fig = plt.figure(figsize=(4, 4))
    for i in range(predictions.shape[0]):
        plt.subplot(4, 4, i+1)
        plt.imshow(predictions[i, :, :, 0].numpy(), cmap='gray')
        plt.axis('off')

    # Save the figure
    plt.savefig('image_at_epoch_{:04d}.png'.format(epoch))
    plt.show()

    # Generate a fixed set of noise for evaluating the model during training
fixed_noise = tf.random.normal([num_examples_to_generate, noise_dim])

# Train the model
for epoch in range(EPOCHS):
    for image_batch in train_dataset:
        train_step(image_batch)

    # Generate and save images every 10 epochs
    if (epoch + 1) % 10 == 0:
        generate_and_save_images(generator, epoch + 1, fixed_noise)

    # Print progress every epoch
    print('Epoch {} completed'.format(epoch + 1))
```

```
[]
Epoch 1 completed
Epoch 2 completed
Epoch 3 completed
Epoch 4 completed
Epoch 5 completed
Epoch 6 completed
Epoch 7 completed
Epoch 8 completed
Epoch 9 completed
```

```
Epoch 10 completed
Epoch 11 completed
Epoch 12 completed
Epoch 13 completed
Epoch 14 completed
Epoch 15 completed
Epoch 16 completed
Epoch 17 completed
Epoch 18 completed
Epoch 19 completed
```

Epoch 20 completed
Epoch 21 completed
Epoch 22 completed
Epoch 23 completed
Epoch 24 completed
Epoch 25 completed
Epoch 26 completed
Epoch 27 completed
Epoch 28 completed
Epoch 29 completed

Epoch 30 completed
Epoch 31 completed
Epoch 32 completed
Epoch 33 completed
Epoch 34 completed
Epoch 35 completed
Epoch 36 completed
Epoch 37 completed
Epoch 38 completed
Epoch 39 completed

Epoch 40 completed
Epoch 41 completed
Epoch 42 completed
Epoch 43 completed
Epoch 44 completed
Epoch 45 completed
Epoch 46 completed
Epoch 47 completed
Epoch 48 completed
Epoch 49 completed

Epoch 50 completed
Epoch 51 completed
Epoch 52 completed
Epoch 53 completed
Epoch 54 completed
Epoch 55 completed
Epoch 56 completed
Epoch 57 completed
Epoch 58 completed
Epoch 59 completed

Epoch 60 completed
Epoch 61 completed
Epoch 62 completed
Epoch 63 completed
Epoch 64 completed
Epoch 65 completed
Epoch 66 completed
Epoch 67 completed
Epoch 68 completed
Epoch 69 completed

Epoch 70 completed
Epoch 71 completed
Epoch 72 completed
Epoch 73 completed
Epoch 74 completed
Epoch 75 completed
Epoch 76 completed
Epoch 77 completed
Epoch 78 completed
Epoch 79 completed

Epoch 80 completed
Epoch 81 completed
Epoch 82 completed
Epoch 83 completed
Epoch 84 completed
Epoch 85 completed
Epoch 86 completed
Epoch 87 completed
Epoch 88 completed
Epoch 89 completed

Epoch 90 completed
Epoch 91 completed
Epoch 92 completed
Epoch 93 completed
Epoch 94 completed
Epoch 95 completed
Epoch 96 completed
Epoch 97 completed
Epoch 98 completed
Epoch 99 completed

Epoch 100 completed

Practical 8

Aim: Write a program to implement a simple form of a recurrent neural network.

a. E.g. (4-to-1 RNN) to show that the quantity of rain on a certain day also depends on the values of the previous day.

b. LSTM for sentiment analysis on datasets like UMICH SI650 for similar

Theory: The program implements a simple form of a recurrent neural network (RNN), such as a 4-to-1 RNN, to demonstrate the dependency of rainfall quantity on the previous day's values. Additionally, it utilizes LSTM (Long Short-Term Memory) for sentiment analysis on datasets like UMICH SI650, aiming to analyze and classify sentiment in textual data.

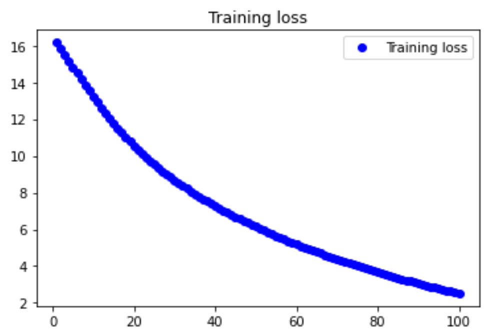
Implementation:

A. (4-to-1 RNN) to show that the quantity of rain on a certain day also depends on the values of the previous day.

```
#Import necessary libraries
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

# Define sequence of 50 days of rain data
rain_data = np.array([2.3, 1.5, 3.1, 2.0, 2.5, 1.7, 2.9, 3.5, 3.0, 2.1,
2.5, 2.2, 2.8, 3.2, 1.8, 2.7, 1.9, 3.1, 3.3, 2.0,
2.5, 2.2, 2.4, 3.0, 2.1, 2.5, 3.2, 3.1, 1.9, 2.7,
2.2, 2.8, 3.1, 2.0, 2.5, 1.7, 2.9, 3.5, 3.0, 2.1,
2.5, 2.2, 2.8, 3.2, 1.8, 2.7, 1.9, 3.1, 3.3, 2.0])

# Create input and output sequences for training
def create_sequences(values, time_steps):
   x = []
   y = []
   for i in range(len(values)-time_steps):
      x.append(values[i:i+time_steps])
      y.append(values[i+time_steps])
   return np.array(x), np.array(y)
time_steps = 4
x_train, y_train = create_sequences(rain_data, time_steps)

# Define RNN model
model = tf.keras.models.Sequential([
   tf.keras.layers.SimpleRNN(8, input_shape=(time_steps, 1)),
   tf.keras.layers.Dense(1)
])

# Compile model
model.compile(optimizer="adam", loss="mse")

# Train model
history = model.fit(x_train.reshape(-1, time_steps, 1), y_train, epochs=100)

# Plot loss over time
loss = history.history["loss"]
epochs = range(1, len(loss) + 1)
```

```
plt.plot(epochs, loss, "bo", label="Training loss")
plt.title("Training loss")
plt.legend()
plt.show()

# Test model on new sequence
test_sequence = np.array([2.5, 2.2, 2.8, 3.2])
x_test = np.array([test_sequence])
y_test = model.predict(x_test.reshape(-1, time_steps, 1))

# Print input, output, and prediction
print("Previous days' rain data:", test_sequence)
print("Expected rain amount for next day:", y_test[0][0])
prediction = model.predict(np.array([test_sequence]).reshape(1, time_steps, 1))
print("Prediction:", prediction[0][0])
```

```
Epoch 1/100
2/2 [==============================] - 1s 8ms/step - loss: 16.1703
Epoch 2/100
2/2 [==============================] - 0s 10ms/step - loss: 15.8426
Epoch 3/100
2/2 [==============================] - 0s 9ms/step - loss: 15.5047
Epoch 4/100
2/2 [==============================] - 0s 9ms/step - loss: 15.1716
Epoch 5/100
2/2 [==============================] - 0s 5ms/step - loss: 14.8375

Epoch 95/100
2/2 [==============================] - 0s 4ms/step - loss: 2.7807
Epoch 96/100
2/2 [==============================] - 0s 4ms/step - loss: 2.7283
Epoch 97/100
2/2 [==============================] - 0s 5ms/step - loss: 2.6745
Epoch 98/100
2/2 [==============================] - 0s 4ms/step - loss: 2.6230
Epoch 99/100
2/2 [==============================] - 0s 5ms/step - loss: 2.5718
Epoch 100/100
2/2 [==============================] - 0s 6ms/step - loss: 2.5197
```




```
Previous days' rain data: [2.5 2.2 2.8 3.2]
Expected rain amount for next day: 1.0503196
Prediction: 1.0503196
```

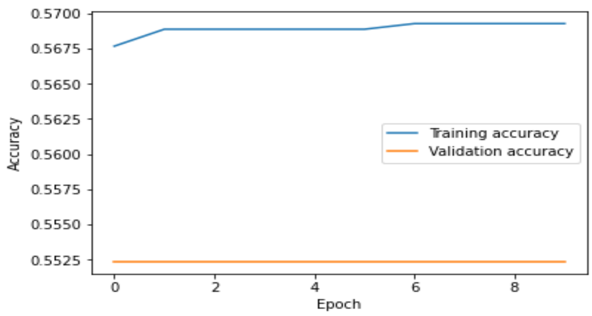
#The output of this program will show the loss of the training data over time, as well as the expected rain amount for the next day given the previous 4 days' rain data, and the model's prediction of the next day's rain amount. Note that the expected rain amount is simply the true value for the next day in

B. LSTM for sentiment analysis on datasets like UMICH SI650 for similar

```
#Import necessary libraries
import pandas as pd
import numpy as np
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

# Load data
data = pd.read_csv("training.txt", delimiter="\t", names=["label", "text"])

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(data["text"],data["label"], test_size=0.2, random_state=42)

# Tokenize words
tokenizer = Tokenizer(num_words=5000, oov_token="<OOV>")
tokenizer.fit_on_texts(X_train)

# Convert words to sequences
X_train_seq = tokenizer.texts_to_sequences(X_train)
X_test_seq = tokenizer.texts_to_sequences(X_test)

# Pad sequences to have same length
max_length = 100
X_train_pad = pad_sequences(X_train_seq, maxlen=max_length, padding="post",truncating="post")
X_test_pad = pad_sequences(X_test_seq, maxlen=max_length, padding="post",truncating="post")

# Build LSTM model
model = tf.keras.models.Sequential([
tf.keras.layers.Embedding(input_dim=5000, output_dim=32,input_length=max_length),
tf.keras.layers.LSTM(units=64, dropout=0.2, recurrent_dropout=0.2),
tf.keras.layers.Dense(1, activation="sigmoid")
])

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

# Train model
history = model.fit(X_train_pad, y_train, epochs=10, batch_size=32,validation_split=0.1)

# Evaluate model on test data
loss, accuracy = model.evaluate(X_test_pad, y_test)
print("Test loss:", loss)
print("Test accuracy:", accuracy)
```

```
# Plot training and validation accuracy over time
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.show()


# Make predictions on test data
predictions = model.predict(X_test_pad)

# Print input, output, and prediction for random example
index = np.random.randint(0, len(X_test_pad))
text = tokenizer.sequences_to_texts([X_test_pad[index]])[0]
label = y_test.values[index]
prediction = predictions[index][0]
print("Text:", text)
print("Actual label:", label)
print("Predicted label:", round(prediction))
```

```
Epoch 1/10
156/156 [==============================] - 31s 143ms/step - loss: 0.6852 - accuracy: 0.5677 - val_loss: 0.6877 - val_accuracy:
0.5523
Epoch 2/10
156/156 [==============================] - 21s 138ms/step - loss: 0.6839 - accuracy: 0.5689 - val_loss: 0.6885 - val_accuracy:
0.5523
Epoch 3/10
156/156 [==============================] - 22s 140ms/step - loss: 0.6841 - accuracy: 0.5689 - val_loss: 0.6882 - val_accuracy:
0.5523
Epoch 4/10
156/156 [==============================] - 23s 146ms/step - loss: 0.6841 - accuracy: 0.5689 - val_loss: 0.6880 - val_accuracy:
0.5523
Epoch 5/10
156/156 [==============================] - 22s 139ms/step - loss: 0.6845 - accuracy: 0.5689 - val_loss: 0.6877 - val_accuracy:
0.5523
Epoch 6/10
156/156 [==============================] - 22s 143ms/step - loss: 0.6838 - accuracy: 0.5689 - val_loss: 0.6877 - val_accuracy:
0.5523
Epoch 7/10
156/156 [==============================] - 22s 138ms/step - loss: 0.6839 - accuracy: 0.5693 - val_loss: 0.6880 - val_accuracy:
0.5523
Epoch 8/10
156/156 [==============================] - 26s 165ms/step - loss: 0.6841 - accuracy: 0.5693 - val_loss: 0.6881 - val_accuracy:
0.5523
Epoch 9/10
156/156 [==============================] - 22s 144ms/step - loss: 0.6840 - accuracy: 0.5693 - val_loss: 0.6877 - val_accuracy:
0.5523
Epoch 10/10
156/156 [==============================] - 20s 127ms/step - loss: 0.6840 - accuracy: 0.5693 - val_loss: 0.6886 - val_accuracy:
0.5523
44/44 [==============================] - 1s 30ms/step - loss: 0.6801 - accuracy: 0.5809
Test loss: 0.6801254749298096
Test accuracy: 0.5809248685836792
```




```
Text: these harry potter movies really suck <OOV> <OOV> <OOV> <OOV
<OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV>
OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <
OV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <O
V> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OOV> <OO
Actual label: 0
Predicted label: 1
```

Practical 9

Aim: Write a program for object detection from the image/video.

Theory: This program enables object detection from images or videos by utilizing computer vision techniques and algorithms. It analyzes the visual content to identify and locate objects of interest within the given input.

Implementation:

```
#Import necessary libraries
import numpy as np
import tensorflow as tf
from tensorflow.keras.applications.vgg16 import VGG16, preprocess_input,decode_predictions
from tensorflow.keras.preprocessing.image import load_img, img_to_array

# Load the VGG16 model
model = VGG16()

# Load the image to detect objects in
img = load_img('objectdetectimage.jpg', target_size=(224, 224))
img_arr = img_to_array(img)
img_arr = np.expand_dims(img_arr, axis=0)
img_arr = preprocess_input(img_arr)

# Predict the objects in the image
preds = model.predict(img_arr)
decoded_preds = decode_predictions(preds, top=5)[0]

# Print the predicted objects and their probabilities
for pred in decoded_preds:
  print(f"{pred[1]}: {pred[2]*100:.2f}%")
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16_weights_tf_dim_ordering_tf_kernels.h5
553467904/553467096 [==============================] - 108s 0us/step
553476096/553467096 [==============================] - 108s 0us/step
Downloading data from https://storage.googleapis.com/download.tensorflow.org/data/imagenet_class_index.json
40960/35363 [==================================] - 0s 0us/step
49152/35363 [=========================================] - 0s 0us/step
necklace: 99.62%
chain: 0.26%
starfish: 0.03%
chain_mail: 0.02%
hook: 0.01%
```

Practical 10

Aim: Write a program for object detection using pre-trained models to use object detection.

Theory: The program utilizes pre-trained models to perform object detection, enabling the identification and localization of objects within images or videos. It streamlines the process by leveraging existing model weights and architectures, allowing for efficient and accurate object recognition.

Implementation:

```
#Import necessary libraries
import tensorflow as tf
import numpy as np
from tensorflow.keras.preprocessing.image import load_img
from tensorflow.keras.preprocessing.image import img_to_array
from tensorflow.keras.applications.vgg16 import VGG16
from tensorflow.keras.applications.vgg16 import preprocess_input
from tensorflow.keras.applications.vgg16 import decode_predictions

# Load the VGG16 model with pre-trained weights
model = VGG16()

# Load the image
image = load_img('objectdetectimage2.jpg', target_size=(224, 224))

# Convert the image to a numpy array
image = img_to_array(image)

# Reshape the image data for VGG
image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))

# Preprocess the image
image = preprocess_input(image)

# Make predictions on the image using the VGG model
predictions = model.predict(image)

# Decode the predictions
decoded_predictions = decode_predictions(predictions, top=2)

# Print the predictions with their probabilities
for i, prediction in enumerate(decoded_predictions[0]):
    print("Object ", i+1, ": ", prediction[1], ", Probability: ", prediction[2])
```

```
Object  1 :  birdhouse , Probability:  0.10978619
Object  2 :  soccer_ball , Probability:  0.09997672
```