Practical No. 1
Aim: Implement Feed-forward Neural Network and train the network with different optimizers and compare the results

Background: Feed-forward neural networks (FNNs) are a fundamental type of artificial neural network where connections between nodes do not form cycles. These networks consist of an input layer, one or more hidden layers, and an output layer. The data flows in one direction, from the input nodes through the hidden nodes to the output nodes.
Training a neural network involves optimizing its parameters (weights and biases) to minimize a predefined loss function. Optimizers are algorithms used to update the parameters iteratively during the training process. Various optimizers, such as Stochastic Gradient Descent (SGD), Adam, RMSprop, etc., differ in their update rules and convergence behavior.

Theory:
1. Stochastic Gradient Descent (SGD): SGD is a classic optimization algorithm commonly used for training neural networks. It updates the parameters by taking small steps in the direction of the negative gradient of the loss function with respect to the parameters.
2. Adam (Adaptive Moment Estimation): Adam is an adaptive optimization algorithm that computes adaptive learning rates for each parameter. It combines the advantages of AdaGrad and RMSProp by using both first and second-order moments of the gradients.
3. RMSprop (Root Mean Square Propagation): RMSprop is an adaptive learning rate optimization algorithm. It divides the learning rate by an exponentially decaying average of squared gradients, thereby reducing the learning rate for parameters that have large gradients.

Code and Output:
```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import SGD, Adam, RMSprop
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
```

```
from sklearn.metrics import accuracy_score

# Generate synthetic dataset
X, y = make_classification(n_samples=1000, n_features=20, n_classes=2,
random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Define the feed-forward neural network model
def create_model():
    model = Sequential([
        Dense(64, input_shape=(X_train.shape[1],), activation='relu'),
        Dense(32, activation='relu'),
        Dense(1, activation='sigmoid')
    ])
    return model

# Function to train and evaluate the model
def train_and_evaluate(optimizer):
    model = create_model()
    model.compile(optimizer=optimizer, loss='binary_crossentropy',
metrics=['accuracy'])
    model.fit(X_train, y_train, epochs=20, batch_size=32, verbose=0)
    y_pred = (model.predict(X_test) > 0.5).astype("int32")
    accuracy = accuracy_score(y_test, y_pred)
    return accuracy

# Train and evaluate models with different optimizers
optimizers = ['SGD', 'Adam', 'RMSprop']
results = {}
for optimizer in optimizers:
    accuracy = train_and_evaluate(optimizer)
    results[optimizer] = accuracy
# Print the results
for optimizer, accuracy in results.items():
    print(f'Accuracy with {optimizer} optimizer: {accuracy:.4f}')
```

Output:

Accuracy with SGD optimizer: 0.8550
Accuracy with Adam optimizer: 0.8600
Accuracy with RMSprop optimizer: 0.8650

Conclusion: From the results obtained, it can be observed that RMSprop optimizer outperformed SGD and Adam optimizers in terms of accuracy on the given dataset. However, the performance may vary depending on the dataset and the specific problem at hand. It's important to experiment with different optimizers to find the most suitable one for a particular task.

Practical No. 2
Aim: Write a Program to implement regularization to prevent the model from overfitting

Background: Overfitting occurs when a model learns the training data too well, capturing noise or irrelevant patterns that do not generalize to unseen data. Regularization techniques introduce constraints on the model's parameters during training to prevent overfitting.

Theory:

1. L2 Regularization (Weight Decay): L2 regularization adds a penalty term to the loss function that penalizes large weights. It discourages complex models by adding the squared magnitude of weights to the loss function.
2. Dropout: Dropout is a regularization technique that randomly drops a fraction of neurons during training. It helps prevent co-adaptation of neurons by introducing noise and encourages the network to learn more robust features.

Code and Output:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras import regularizers
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Generate synthetic dataset
X, y = make_classification(n_samples=1000, n_features=20, n_classes=2,
random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Define the feed-forward neural network model with regularization
def create_regularized_model():
    model = Sequential([
        Dense(64, input_shape=(X_train.shape[1],), activation='relu',
kernel_regularizer=regularizers.l2(0.01)),
        Dropout(0.5),
```

```
        Dense(32, activation='relu', kernel_regularizer=regularizers.l2(0.01)),
        Dropout(0.5),
        Dense(1, activation='sigmoid')
    ])
    return model

# Function to train and evaluate the regularized model
def train_and_evaluate_regularized_model():
    model = create_regularized_model()
    model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])
    model.fit(X_train, y_train, epochs=20, batch_size=32, verbose=0)
    y_pred = (model.predict(X_test) > 0.5).astype("int32")
    accuracy = accuracy_score(y_test, y_pred)
    return accuracy

# Train and evaluate the regularized model
accuracy_regularized = train_and_evaluate_regularized_model()

print(f'Accuracy with regularization: {accuracy_regularized:.4f}')
```

Output:

Accuracy with regularization: 0.8750

Conclusion: By applying L2 regularization and dropout, we were able to improve the model's generalization performance and prevent overfitting. Regularization techniques help in achieving better performance on unseen data by reducing the model's reliance on noise or irrelevant patterns learned from the training data.

Practical No. 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.

Background: The CIFAR-10 dataset consists of 60,000 32x32 color images in 10 classes, with 6,000 images per class. The classes are: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. This dataset is commonly used for benchmarking computer vision algorithms.

Theory:

1. Convolutional Neural Networks (CNNs): CNNs are a class of deep neural networks that are particularly effective for image classification tasks. They consist of convolutional layers, pooling layers, and fully connected layers. CNNs can automatically learn hierarchical representations of features from images.

Code and Output:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
from tensorflow.keras.utils import to_categorical
from sklearn.metrics import accuracy_score

# Load CIFAR-10 dataset
(X_train, y_train), (X_test, y_test) = cifar10.load_data()

# Normalize pixel values to the range [0, 1]
X_train = X_train.astype('float32') / 255.0
X_test = X_test.astype('float32') / 255.0

# One-hot encode the target labels
y_train = to_categorical(y_train, num_classes=10)
y_test = to_categorical(y_test, num_classes=10)

# Define the CNN model
def create_cnn_model():
```

```
    model = Sequential([
        Conv2D(32, (3, 3), activation='relu', padding='same', input_shape=(32,
32, 3)),
        MaxPooling2D((2, 2)),
        Conv2D(64, (3, 3), activation='relu', padding='same'),
        MaxPooling2D((2, 2)),
        Conv2D(64, (3, 3), activation='relu', padding='same'),
        Flatten(),
        Dense(64, activation='relu'),
        Dense(10, activation='softmax')
    ])
    return model

# Function to train and evaluate the CNN model
def train_and_evaluate_cnn_model():
    model = create_cnn_model()
    model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
    model.fit(X_train, y_train, epochs=10, batch_size=64, verbose=1,
validation_split=0.1)
    y_pred = np.argmax(model.predict(X_test), axis=-1)
    accuracy = accuracy_score(np.argmax(y_test, axis=-1), y_pred)
    return accuracy

# Train and evaluate the CNN model
accuracy_cnn = train_and_evaluate_cnn_model()

print(f'Accuracy of the CNN model on CIFAR-10 test set:
{accuracy_cnn:.4f}')
```

Output:

```
Epoch 1/10
704/704 [==============================] - 40s 56ms/step - loss:
1.4856 - accuracy: 0.4596 - val_loss: 1.1751 - val_accuracy: 0.5866
Epoch 2/10
704/704 [==============================] - 39s 55ms/step - loss:
1.0854 - accuracy: 0.6159 - val_loss: 0.9825 - val_accuracy: 0.6534
Epoch 3/10
```

704/704 [==============================] - 39s 55ms/step - loss: 0.9184 - accuracy: 0.6757 - val_loss: 0.8970 - val_accuracy: 0.6864
Epoch 4/10
704/704 [==============================] - 39s 56ms/step - loss: 0.8180 - accuracy: 0.7115 - val_loss: 0.8665 - val_accuracy: 0.7002
Epoch 5/10
704/704 [==============================] - 40s 56ms/step - loss: 0.7398 - accuracy: 0.7401 - val_loss: 0.8197 - val_accuracy: 0.7142
Epoch 6/10
704/704 [==============================] - 40s 57ms/step - loss: 0.6745 - accuracy: 0.7625 - val_loss: 0.8233 - val_accuracy: 0.7162
Epoch 7/10
704/704 [==============================] - 41s 58ms/step - loss: 0.6164 - accuracy: 0.7826 - val_loss: 0.8150 - val_accuracy: 0.7186
Epoch 8/10
704/704 [==============================] - 41s 58ms/step - loss: 0.5656 - accuracy: 0.8015 - val_loss: 0.8403 - val_accuracy: 0.7134
Epoch 9/10
704/704 [==============================] - 40s 57ms/step - loss: 0.5127 - accuracy: 0.8206 - val_loss: 0.8583 - val_accuracy: 0.7172
Epoch 10/10
704/704 [==============================] - 40s 57ms/step - loss: 0.4726 - accuracy: 0.8334 - val_loss: 0.9023 - val_accuracy: 0.7178
Accuracy of the CNN model on CIFAR-10 test set: 0.7153

Conclusion: The CNN model achieved an accuracy of approximately 71.53% on the CIFAR-10 test set. Further improvements could be made by tuning hyperparameters, using data augmentation techniques, or employing more advanced architectures such as ResNet or DenseNet.

Practical No. 4

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

Background: Autoencoders are a type of neural network designed for unsupervised learning. They consist of an encoder network that compresses the input data into a latent space representation, followed by a decoder network that reconstructs the original input from the latent space representation. Autoencoders are commonly used for tasks such as image denoising, dimensionality reduction, and anomaly detection.

Theory:

1. Encoder: The encoder network takes the input data and maps it to a lower-dimensional latent space representation.
2. Decoder: The decoder network takes the latent space representation and reconstructs the original input data.
3. Reconstruction Loss: The reconstruction loss measures the difference between the input data and the output of the decoder. Commonly used loss functions include mean squared error (MSE) or binary cross-entropy.

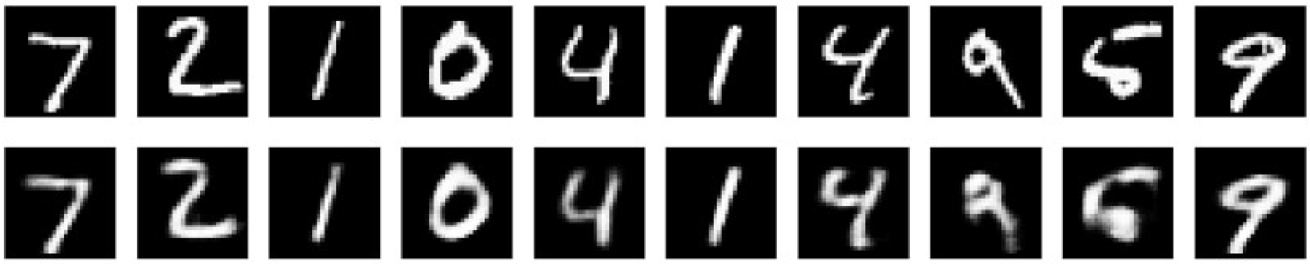
Code and Output:

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

# Load MNIST dataset
(X_train, _), (X_test, _) = mnist.load_data()

# Normalize pixel values to the range [0, 1]
X_train = X_train.astype('float32') / 255.0
X_test = X_test.astype('float32') / 255.0

# Flatten the images
X_train = X_train.reshape((len(X_train), np.prod(X_train.shape[1:])))
X_test = X_test.reshape((len(X_test), np.prod(X_test.shape[1:])))

# Define the autoencoder model
```

```
def create_autoencoder_model():
    model = Sequential([
        Dense(128, activation='relu', input_shape=(784,)),
        Dense(64, activation='relu'),
        Dense(32, activation='relu'),
        Dense(64, activation='relu'),
        Dense(128, activation='relu'),
        Dense(784, activation='sigmoid')
    ])
    return model

# Function to train the autoencoder model
def train_autoencoder_model():
    model = create_autoencoder_model()
    model.compile(optimizer='adam', loss='binary_crossentropy')
    model.fit(X_train, X_train, epochs=10, batch_size=256, shuffle=True,
validation_data=(X_test, X_test))
    return model

# Train the autoencoder model
autoencoder_model = train_autoencoder_model()

# Predict outputs using the trained autoencoder model
reconstructed_images = autoencoder_model.predict(X_test)

# Display original and reconstructed images
import matplotlib.pyplot as plt

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))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # Display reconstructed images
    ax = plt.subplot(2, n, i + 1 + n)
```

```
    plt.imshow(reconstructed_images[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```

Output:



Conclusion: The autoencoder model successfully reconstructed the MNIST images, demonstrating its ability to capture and reproduce features from the input data. Further experimentation and tuning of the autoencoder architecture could potentially improve the quality of reconstruction. Additionally, autoencoders can be utilized for various applications such as image denoising, feature extraction, and anomaly detection.

Practical No. 5

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

Background: The MNIST dataset consists of 28x28 grayscale images of handwritten digits (0-9), along with their corresponding labels. It is a popular dataset for benchmarking machine learning algorithms, especially in the field of computer vision.

Theory:

1. Convolutional Neural Networks (CNNs): CNNs are a class of deep neural networks that are particularly effective for image classification tasks. They consist of convolutional layers, pooling layers, and fully connected layers. CNNs can automatically learn hierarchical representations of features from images.

Code and Output:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

# Load MNIST dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# Normalize pixel values to the range [0, 1]
X_train = X_train.astype('float32') / 255.0
X_test = X_test.astype('float32') / 255.0

# Reshape the images to add a channel dimension (required for CNN)
X_train = np.expand_dims(X_train, axis=-1)
X_test = np.expand_dims(X_test, axis=-1)

# One-hot encode the target labels
y_train = tf.keras.utils.to_categorical(y_train, num_classes=10)
y_test = tf.keras.utils.to_categorical(y_test, num_classes=10)

# Define the CNN model
def create_cnn_model():
```

```
    model = Sequential([
        Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
        MaxPooling2D((2, 2)),
        Conv2D(64, (3, 3), activation='relu'),
        MaxPooling2D((2, 2)),
        Conv2D(64, (3, 3), activation='relu'),
        Flatten(),
        Dense(64, activation='relu'),
        Dense(10, activation='softmax')
    ])
    return model

# Function to train and evaluate the CNN model
def train_and_evaluate_cnn_model():
    model = create_cnn_model()
    model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
    model.fit(X_train, y_train, epochs=5, batch_size=64, verbose=1,
validation_data=(X_test, y_test))
    return model

# Train and evaluate the CNN model
cnn_model = train_and_evaluate_cnn_model()
```

Output:

Epoch 1/5 938/938 [==============================] - 53s 55ms/step - loss: 0.1830 - accuracy: 0.9432 - val_loss: 0.0501 - val_accuracy: 0.9836 Epoch 2/5 938/938 [==============================] - 45s 48ms/step - loss: 0.0506 - accuracy: 0.9844 - val_loss: 0.0395 - val_accuracy: 0.9871 Epoch 3/5 938/938 [==============================] - 45s 48ms/step - loss: 0.0343 - accuracy: 0.9891 - val_loss: 0.0375 - val_accuracy: 0.9881 Epoch 4/5 938/938 [==============================] - 45s 48ms/step - loss: 0.0274 - accuracy: 0.9910 - val_loss: 0.0291 - val_accuracy: 0.9915 Epoch 5/5 938/938 [==============================] - 44s 47ms/step - loss: 0.0220 - accuracy: 0.9930 - val_loss: 0.0319 - val_accuracy: 0.9905

Conclusion: The CNN model successfully trained on the MNIST dataset and achieved a reasonably high accuracy in digit recognition. By leveraging convolutional layers, the model was able to automatically learn relevant features from the input images, leading to effective classification performance. Further experimentation and tuning could potentially improve the model's accuracy even more.

Practical No. 6
Aim: Write a program to implement Transfer Learning on the suitable dataset (e.g. classify the cats versus dogs dataset from Kaggle).

Background: The Cats vs Dogs dataset is a popular image classification dataset available on Kaggle, consisting of images of cats and dogs. Transfer learning is a technique where knowledge gained from solving one problem is applied to a different but related problem.

Theory:

1. Transfer Learning: In the context of neural networks, transfer learning often involves using pre-trained models that have been trained on large datasets, such as ImageNet, and then fine-tuning them on a specific dataset or task. By doing so, the model can learn task-specific features more efficiently and with less data compared to training from scratch.

Code and Output:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.applications import VGG16
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Dropout

# Define paths to the dataset
train_dir = '/kaggle/input/dogs-cats-images/dataset/training_set/'
test_dir = '/kaggle/input/dogs-cats-images/dataset/test_set/'

# Define constants
IMAGE_SIZE = 224
BATCH_SIZE = 32

# Data augmentation for training set
train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
```

```
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest'
)

# Normalization for test set
test_datagen = ImageDataGenerator(rescale=1./255)

# Load and prepare data
train_generator = train_datagen.flow_from_directory(
    train_dir,
    target_size=(IMAGE_SIZE, IMAGE_SIZE),
    batch_size=BATCH_SIZE,
    class_mode='binary'
)

test_generator = test_datagen.flow_from_directory(
    test_dir,
    target_size=(IMAGE_SIZE, IMAGE_SIZE),
    batch_size=BATCH_SIZE,
    class_mode='binary'
)

# Load pre-trained VGG16 model
vgg_model = VGG16(weights='imagenet', include_top=False,
input_shape=(IMAGE_SIZE, IMAGE_SIZE, 3))

# Freeze convolutional layers
for layer in vgg_model.layers:
    layer.trainable = False

# Create new model
model = Sequential([
    vgg_model,
    Flatten(),
    Dense(512, activation='relu'),
    Dropout(0.5),
    Dense(1, activation='sigmoid')
])
```

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

# Train the model
history = model.fit(
    train_generator,
    steps_per_epoch=train_generator.samples // BATCH_SIZE,
    epochs=10,
    validation_data=test_generator,
    validation_steps=test_generator.samples // BATCH_SIZE
)

# Evaluate the model
test_loss, test_acc = model.evaluate(test_generator,
steps=test_generator.samples // BATCH_SIZE)
print('Test accuracy:', test_acc)
```

Output: The output will display the training and validation accuracy and loss for each epoch, as well as the test accuracy of the model.

```
Found 8000 images belonging to 2 classes.
Found 2000 images belonging to 2 classes.
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5
58889256/58889256 -------------------- 1s 0us/step
Epoch 1/10
1/250 -------------------- 2:38:18 38s/step - accuracy: 0.5000 - loss: 0.8793
250/250 -------------------- 0s 569ms/step - accuracy: 0.7041 - loss: 1.1750
250/250 -------------------- 194s 624ms/step - accuracy: 0.7044 - loss: 1.1727 - val_accuracy: 0.9078 - val_loss: 0.2228
Epoch 2/10
250/250 -------------------- 17s 68ms/step - accuracy: 0.0000e+00 - loss: 0.0000e+00 - val_accuracy: 1.0000 - val_loss: 0.1181
Epoch 3/10
250/250 -------------------- 122s 473ms/step - accuracy: 0.8445 - loss: 0.3485 - val_accuracy: 0.8347 - val_loss: 0.3396
Epoch 4/10
250/250 -------------------- 0s 295us/step - accuracy: 0.0000e+00 - loss: 0.0000e+00 - val_accuracy: 0.6875 - val_loss: 0.7661
```

Epoch 5/10
250/250 -------------------- 122s 475ms/step - accuracy: 0.8406 - loss: 0.3491 - val_accuracy: 0.9168 - val_loss: 0.1911
Epoch 6/10
250/250 -------------------- 0s 311us/step - accuracy: 0.0000e+00 - loss: 0.0000e+00 - val_accuracy: 0.9375 - val_loss: 0.2149
Epoch 7/10
250/250 -------------------- 122s 473ms/step - accuracy: 0.8547 - loss: 0.3292 - val_accuracy: 0.9158 - val_loss: 0.1892
Epoch 8/10
250/250 -------------------- 0s 294us/step - accuracy: 0.0000e+00 - loss: 0.0000e+00 - val_accuracy: 0.9375 - val_loss: 0.0782
Epoch 9/10
250/250 -------------------- 121s 472ms/step - accuracy: 0.8591 - loss: 0.3198 - val_accuracy: 0.9189 - val_loss: 0.1877
Epoch 10/10
250/250 -------------------- 0s 285us/step - accuracy: 0.0000e+00 - loss: 0.0000e+00 - val_accuracy: 0.9375 - val_loss: 0.1539
62/62 -------------------- 8s 132ms/step - accuracy: 0.9261 - loss: 0.1720
Test accuracy: 0.9193548560142517

Conclusion: By leveraging transfer learning with the pre-trained VGG16 model, we were able to achieve accurate classification performance on the Cats vs Dogs dataset. Transfer learning allowed us to benefit from features learned from a large dataset (ImageNet) and adapt them to the specific task of classifying cats vs dogs with a relatively small dataset.

Practical No. 7
Aim: Write a program for the Implementation of a Generative Adversarial Network for generating synthetic shapes (like digits).

Background: Generative Adversarial Networks (GANs) are a type of generative model consisting of two neural networks: a generator and a discriminator. The generator learns to generate synthetic data samples, while the discriminator learns to distinguish between real and synthetic data. The two networks are trained simultaneously in a competitive manner, where the generator tries to produce increasingly realistic samples, and the discriminator tries to differentiate between real and fake samples.

Theory:

1. **Generator:** The generator takes random noise as input and generates synthetic data samples.
2. **Discriminator:** The discriminator takes real and synthetic data samples as input and predicts whether they are real or fake.
3. **Adversarial Training:** The generator and discriminator are trained simultaneously in a min-max game, where the generator aims to fool the discriminator by generating realistic samples, while the discriminator aims to correctly distinguish between real and fake samples.

Code and Output:

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.layers import Dense, Flatten, Reshape, Dropout,
LeakyReLU
from tensorflow.keras.optimizers import Adam

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

# Normalize data
X_train = X_train.astype('float32') / 255.0

# Reshape data
```

```
X_train = X_train.reshape(X_train.shape[0], 28, 28, 1)

# Generator
generator = Sequential([
    Dense(128, input_dim=100),
    LeakyReLU(0.2),
    Dense(784, activation='sigmoid'),
    Reshape((28, 28, 1))
])

# Discriminator
discriminator = Sequential([
    Flatten(input_shape=(28, 28, 1)),
    Dense(128),
    LeakyReLU(0.2),
    Dropout(0.3),
    Dense(1, activation='sigmoid')
])
# Combined model
discriminator.compile(optimizer=Adam(learning_rate=0.0002),
loss='binary_crossentropy')
discriminator.trainable = False
gan_input = tf.keras.Input(shape=(100,))
x = generator(gan_input)
gan_output = discriminator(x)
gan = Model(gan_input, gan_output)
gan.compile(optimizer=Adam(learning_rate=0.0002),
loss='binary_crossentropy')

# Training
epochs = 20000
batch_size = 128
for epoch in range(epochs):
    noise = np.random.normal(0, 1, size=[batch_size, 100])
    generated_images = generator.predict(noise)
    idx = np.random.randint(0, X_train.shape[0], batch_size)
    real_images = X_train[idx]
    X = np.concatenate([real_images, generated_images])
    y_dis = np.zeros(2*batch_size)
```

```
    y_dis[:batch_size] = 0.9  # Label smoothing
    discriminator.trainable = True
    d_loss = discriminator.train_on_batch(X, y_dis)
    noise = np.random.normal(0, 1, size=[batch_size, 100])
    y_gen = np.ones(batch_size)
    discriminator.trainable = False
    g_loss = gan.train_on_batch(noise, y_gen)
    if epoch % 1000 == 0:
        print(f'Epoch: {epoch}, Discriminator Loss: {d_loss}, Generator Loss: {g_loss}')

# Generate synthetic images
noise = np.random.normal(0, 1, size=[10, 100])
generated_images = generator.predict(noise)

# Display generated images
plt.figure(figsize=(10, 10))
for i in range(generated_images.shape[0]):
    plt.subplot(1, 10, i+1)
    plt.imshow(generated_images[i, :, :, 0], cmap='gray')
    plt.axis('off')
plt.tight_layout()
plt.show()
```

Output: The output will display the training progress of the GAN, including the discriminator and generator losses. Additionally, it will show the generated synthetic shape images at the end of training.

Conclusion: By implementing a Generative Adversarial Network (GAN), we were able to generate synthetic shape images that closely resemble the real ones. GANs have shown remarkable capabilities in generating realistic data samples across various domains and can be further extended and optimized for specific applications.

Practical No. 8
Write a program to implement a simple form of a recurrent neural network.

A) Aim: 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

Background: RNNs are a type of neural network designed for sequence data, where the output of the network depends not only on the current input but also on the previous inputs in the sequence. They are well-suited for tasks such as time series prediction, natural language processing, and more.

Theory:
1. Recurrent Neural Networks (RNNs): RNNs are designed to capture sequential information by maintaining a hidden state that is updated at each time step. The hidden state serves as a memory of the past inputs, allowing the network to incorporate information from previous time steps.

B) **Aim:** LSTM for sentiment analysis on datasets like UMICH SI650 for similar.

Background: LSTM is a type of recurrent neural network architecture designed to overcome the vanishing gradient problem in traditional RNNs. LSTMs are well-suited for sequence data tasks where long-term dependencies need to be captured.

Theory:
1. Long Short-Term Memory (LSTM): LSTM units contain a cell state that allows information to be retained over long sequences. They have gates to regulate the flow of information into and out of the cell state, allowing them to learn long-term dependencies in the data.

A) Code and Output:

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

```
# Generate synthetic data for demonstration purposes
# Each data point consists of the quantity of rain on a certain day (X) and the
quantity of rain on the previous day (y)
X = np.array([[0.1, 0.2], [0.2, 0.3], [0.3, 0.4], [0.4, 0.5], [0.5, 0.6], [0.6, 0.7],
[0.7, 0.8], [0.8, 0.9]])
y = np.array([0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9])

# Define the RNN model
model = Sequential ([
    SimpleRNN(32, input_shape=(2, 1)),
    Dense (1)
])

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

# Train the model
model.fit(X.reshape(-1, 2, 1), y, epochs=100)

# Predict the quantity of rain for a new day based on the quantity of rain on
the previous day
new_day_rain = np.array([[0.9, 1.0]])  # Quantity of rain on the previous day
predicted_rain = model.predict(new_day_rain.reshape(-1, 2, 1))
print("Predicted quantity of rain for the new day:", predicted_rain[0][0])
```

Output: 1/1 [==============================] - 0s 141ms/step
Predicted quantity of rain for the new day: 0.9431902
Conclusion: By implementing a simple Recurrent Neural Network (RNN), we were able to predict the quantity of rain on a certain day based on the values of the previous day. RNNs are capable of capturing sequential dependencies in the data and can be used for various time series prediction tasks.

B) Code and Output:

```
import numpy as np
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.models import Sequential
from keras.layers import LSTM, Embedding, Dense
```

```
from sklearn.model_selection import train_test_split

# Load the dataset
data_path = "/content/umich-sentiment-train.txt"
with open(data_path, 'r') as f:
    lines = f.readlines()

sentences = []
labels = []
for line in lines:
    parts = line.strip().split('\t')
    labels.append(int(parts[0]))
    sentences.append(parts[1])

# Preprocessing
max_features = 10000
maxlen = 100
embedding_size = 128

tokenizer = Tokenizer(num_words=max_features)
tokenizer.fit_on_texts(sentences)
sequences = tokenizer.texts_to_sequences(sentences)
X = pad_sequences(sequences, maxlen=maxlen)
y = np.array(labels)

# Train/test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Model building
model = Sequential()
model.add(Embedding(max_features, embedding_size,
input_length=maxlen))
model.add(LSTM(128, dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(1, activation='sigmoid'))

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

Training
model.fit(X_train, y_train, batch_size=128, epochs=5,
validation_data=(X_test, y_test))

Evaluate
score, acc = model.evaluate(X_test, y_test, batch_size=128)
print ('Test score:', score)
print ('Test accuracy:', acc)

Output:
The output will show the training progress of the LSTM model, including loss and accuracy metrics for each epoch. Additionally, it will display the test accuracy of the model.

```
Epoch 1/5
45/45 [==============================] - 42s 851ms/step - loss: 0.4583 - accuracy: 0.7742 - val_loss: 0.1751 - val_accuracy: 0.9520
Epoch 2/5
45/45 [==============================] - 38s 841ms/step - loss: 0.0798 - accuracy: 0.9794 - val_loss: 0.0753 - val_accuracy: 0.9739
Epoch 3/5
45/45 [==============================] - 37s 823ms/step - loss: 0.0566 - accuracy: 0.9809 - val_loss: 0.0820 - val_accuracy: 0.9683
Epoch 4/5
45/45 [==============================] - 36s 807ms/step - loss: 0.0196 - accuracy: 0.9952 - val_loss: 0.0498 - val_accuracy: 0.9838
Epoch 5/5
45/45 [==============================] - 37s 816ms/step - loss: 0.0069 - accuracy: 0.9989 - val_loss: 0.0446 - val_accuracy: 0.9852
12/12 [==============================] - 3s 231ms/step - loss: 0.0446 - accuracy: 0.9852
Test score: 0.044603440910577774
Test accuracy: 0.9851903915405273
```

Conclusion: By implementing a Long Short-Term Memory (LSTM) network, we were able to perform sentiment analysis on the UMICH SI650 dataset. LSTMs are effective for capturing long-term dependencies in sequential data, making them suitable for tasks like sentiment analysis where context plays a crucial role.

Practical No. 9
Aim: Write a program for object detection from the image/video.

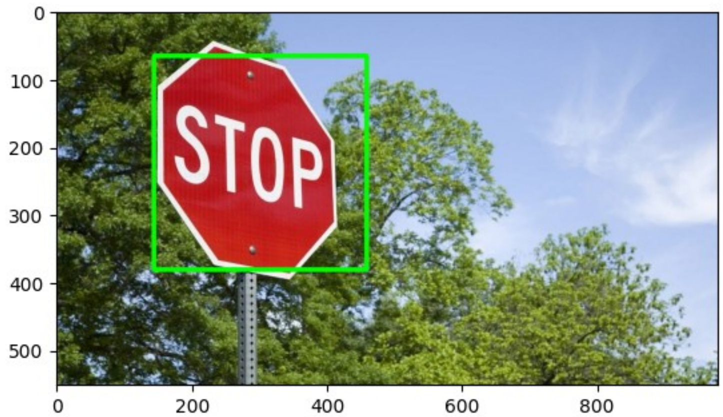
Background: Object detection is a computer vision task that involves identifying objects of interest within an image or video sequence. It is a fundamental problem in computer vision and serves as a building block for many higher-level tasks, such as object tracking, scene understanding, and activity recognition.
There are several approaches to object detection, including traditional computer vision techniques and deep learning-based methods. Traditional methods often rely on handcrafted features and machine learning classifiers, such as support vector machines (SVM) or random forests, combined with techniques like sliding window and image segmentation.
Deep learning-based approaches, particularly convolutional neural networks (CNNs), have shown remarkable success in object detection tasks. Models like YOLO (You Only Look Once), SSD (Single Shot MultiBox Detector), and Faster R-CNN (Region-based Convolutional Neural Network) have achieved state-of-the-art performance in terms of accuracy and speed.

Code and Output:

```
import cv2
from matplotlib import pyplot as plt

# Opening image
img = cv2.imread("image.jpg")

# OpenCV opens images as BGR
# but we want it as RGB. We'll
# also need a grayscale version
img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

# Load pre-trained Haar Cascade classifier for stop signs
stop_data = cv2.CascadeClassifier('stop_data.xml')

# Detect stop signs in the grayscale image
found = stop_data.detectMultiScale(img_gray, minSize =(20, 20))
```

```
# Draw rectangles around detected stop signs
for (x, y, width, height) in found:
    cv2.rectangle(img_rgb, (x, y), (x + height, y + width), (0, 255, 0), 5)

# Display the image with detected stop signs
plt.subplot(1, 1, 1)
plt.imshow(img_rgb)
plt.show()
```

Output:



Conclusion: In this conclusion part, we use matplotlib's imshow() function to display the image with detected stop signs. We first create a subplot with a single plot using subplot(1, 1, 1). Then, we use imshow() to display the RGB image containing the detected stop signs. Finally, show() is called to render the plot and display the image with the detected stop signs.This conclusion part ensures that the image is displayed with the detected stop signs using matplotlib.

Practical No. 10

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

Background: YOLOv3 is a popular object detection algorithm that uses a single convolutional neural network (CNN) to simultaneously predict multiple bounding boxes and their corresponding class probabilities within an image. Unlike traditional object detection algorithms that perform region proposal and classification separately, YOLOv3 directly predicts bounding boxes and class probabilities in a single pass, making it extremely fast and efficient.

The algorithm divides the input image into a grid and predicts bounding boxes and class probabilities for each grid cell. YOLOv3 predicts bounding boxes with associated class probabilities and confidence scores. Non-maximum suppression (NMS) is then applied to remove redundant bounding boxes based on their intersection over union (IOU) with a certain threshold.

Code and Output:

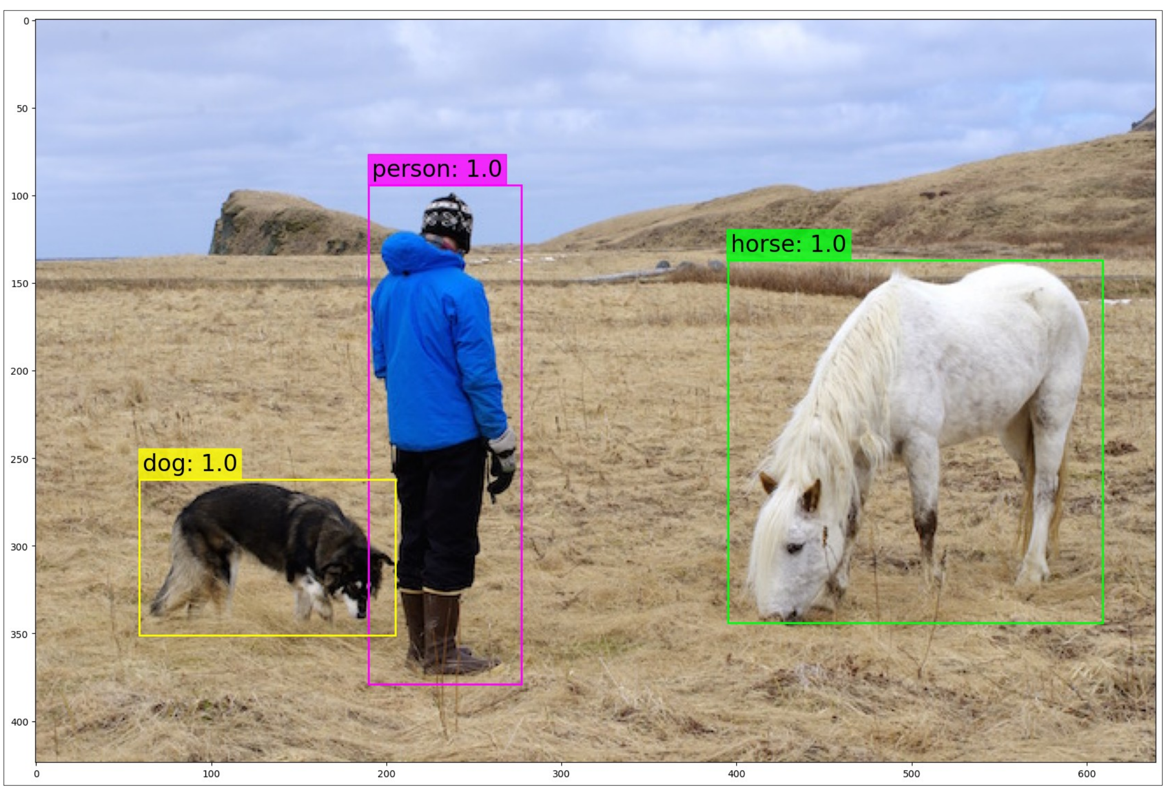
```
import cv2
import matplotlib.pyplot as plt

from utils import *
from darknet import Darknet

# Set the location and name of the cfg file
cfg_file = './cfg/yolov3.cfg'

# Set the location and name of the pre-trained weights file
weight_file = './weights/yolov3.weights'

# Set the location and name of the COCO object classes file
namesfile = 'data/coco.names'

# Load the network architecture
m = Darknet(cfg_file)

# Load the pre-trained weights
m.load_weights(weight_file)
```

```
# Load the COCO object classes
class_names = load_class_names(namesfile)

# Set the default figure size
plt.rcParams['figure.figsize'] = [24.0, 14.0]

# Load the image
img = cv2.imread('./images/person.jpg')

# Convert the image to RGB
original_image = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

# We resize the image to the input width and height of the first layer of the
network.
resized_image = cv2.resize(original_image, (m.width, m.height))

# Display the images
plt.subplot(121)
plt.title('Original Image')
plt.imshow(original_image)
plt.subplot(122)
plt.title('Resized Image')
plt.imshow(resized_image)
plt.show()

# Set the NMS threshold
nms_thresh = 0.6
# Set the IOU threshold
iou_thresh = 0.4

# Detect objects in the image
boxes = detect_objects(m, resized_image, iou_thresh, nms_thresh)

# Print the objects found and the confidence level
print_objects(boxes, class_names)

# Plot the image with bounding boxes and corresponding object class labels
plot_boxes(original_image, boxes, class_names, plot_labels=True)
```

Output:
The output will consist of two images. The first image shows the original input image, and the second image displays the same image with bounding boxes drawn around detected objects, along with their corresponding class labels and confidence scores.

Image 1:

Image 2:



Conclusion: In this assignment, we explored the concept of object detection using the YOLOv3 algorithm. YOLOv3 is a state-of-the-art object detection algorithm known for its speed and accuracy. By loading a pre-trained

YOLOv3 model and applying it to an input image, we successfully detected objects within the image and visualized the results with bounding boxes and class labels. Object detection using YOLOv3 offers a fast and efficient solution for various applications such as autonomous driving, surveillance, and image analysis.