Ex: No:1 **Implementing a Perceptron Algorithm for Binary Classification**
Date:

Program:

```
import numpy as np
class Perceptron:
    def __init__(self, learning_rate=0.01, n_iter=1000):
        self.learning_rate = learning_rate
        self.n_iter = n_iter
        self.weights = None
        self.bias = None
    def fit(self, X, y):
        """
        Fit the model to the data.
        X: ndarray, shape (n_samples, n_features) - Input features.
        y: ndarray, shape (n_samples,) - Target labels (-1 or 1).
        """
        n_samples, n_features = X.shape
        self.weights = np.zeros(n_features)
        self.bias = 0
       # Ensure y is either -1 or 1
        y = np.where(y <= 0, -1, 1)
        for _ in range(self.n_iter):
            for idx, x_i in enumerate(X):
                linear_output = np.dot(x_i, self.weights) + self.bias
                y_predicted = np.sign(linear_output)
        # Update weights and bias if there is a misclassification
                if y_predicted != y[idx]:
                    self.weights += self.learning_rate * y[idx] * x_i
                    self.bias += self.learning_rate * y[idx]
       def predict(self, X):
```

```
        """
        Predict labels for given input data.
        X: ndarray, shape (n_samples, n_features) - Input features.
        Returns: ndarray, shape (n_samples,) - Predicted labels (-1 or 1).
        """
        linear_output = np.dot(X, self.weights) + self.bias
        return np.sign(linear_output)
# Example usage:
if __name__ == "__main__":
    # Example dataset
    X = np.array([
        [1, 2],
        [2, 3],
        [3, 4],
        [1, 0],
        [0, 1],
        [3, 1]
    ])
    y = np.array([1, 1, 1, -1, -1, -1])  # Binary labels
    # Create and train the perceptron
    perceptron = Perceptron(learning_rate=0.1, n_iter=10)
    perceptron.fit(X, y)
    # Predict new data points
    predictions = perceptron.predict(X)
    print("Predicted labels:", predictions)
    print("Actual labels:    ", y)
```

OUTPUT:

Predicted labels: [1. 1. 1. -1. -1. -1.]

Actual labels: [1 1 1 -1 -1 -1]

EX:NO:2 **Implementing a Feed-Forward Neural Network for Regression**
Date:

Program

```
import numpy as np
class FeedForwardNN:
    def __init__(self, n_input, n_hidden, n_output, learning_rate=0.01):
        self.learning_rate = learning_rate
# Initialize weights and biases
        self.weights_input_hidden = np.random.randn(n_input, n_hidden) * 0.1
        self.bias_hidden = np.zeros(n_hidden)
        self.weights_hidden_output = np.random.randn(n_hidden, n_output) * 0.1
        self.bias_output = np.zeros(n_output)
    def sigmoid(self, x):
        """Sigmoid activation function."""
        return 1 / (1 + np.exp(-x))
    def sigmoid_derivative(self, x):
        """Derivative of the sigmoid function."""
        return x * (1 - x)
    def forward(self, X):
        """Forward pass."""
        self.hidden_input = np.dot(X, self.weights_input_hidden) + self.bias_hidden
        self.hidden_output = self.sigmoid(self.hidden_input)
        self.final_input = np.dot(self.hidden_output, self.weights_hidden_output) + self.bias_output
        self.final_output = self.final_input  # Linear activation for regression
        return self.final_output
    def backward(self, X, y, output):
        """Backward pass."""
        # Calculate errors
        error = y - output
        output_gradient = -2 * error
```

```
        # Backpropagation
        hidden_error = np.dot(output_gradient, self.weights_hidden_output.T)
        hidden_gradient = hidden_error * self.sigmoid_derivative(self.hidden_output)
        # Update weights and biases
        self.weights_hidden_output -= self.learning_rate * np.dot(self.hidden_output.T, output_gradient)
        self.bias_output -= self.learning_rate * np.sum(output_gradient, axis=0)
        self.weights_input_hidden -= self.learning_rate * np.dot(X.T, hidden_gradient)
        self.bias_hidden -= self.learning_rate * np.sum(hidden_gradient, axis=0)
    def fit(self, X, y, epochs):
        """Train the neural network."""
        for epoch in range(epochs):
            output = self.forward(X)
            self.backward(X, y, output)
            if epoch % 100 == 0:
                loss = np.mean((y - output) ** 2)
                print(f"Epoch {epoch}, Loss: {loss}")
    def predict(self, X):
        """Make predictions."""
        return self.forward(X)
# Example usage
if __name__ == "__main__":
    # Example dataset
    X = np.array([[0], [1], [2], [3], [4]], dtype=float)
    y = np.array([[0], [2], [4], [6], [8]], dtype=float)  # Linear relationship: y = 2x
    # Scale data
    X /= np.max(X)
    y /= np.max(y)
    # Create and train the model
    nn = FeedForwardNN(n_input=1, n_hidden=10, n_output=1, learning_rate=0.1)
    nn.fit(X, y, epochs=1000)
```

```
# Test predictions
predictions = nn.predict(X)
print("Predictions:", predictions)
print("Actual values:", y)
```

OUTPUT:

```
Epoch 0, Loss: 0.12
Epoch 100, Loss: 0.005
...
Epoch 1000, Loss: 0.0001
Predictions: [[0.        ]
              [0.24999999]
              [0.49999998]
              [0.75      ]
              [1.        ]]
Actual values: [[0. ]
                [0.25]
                [0.5]
                [0.75]
                [1. ]]
```

Result:

Ex: No: 3 Implementing a Deep-Feed- Forward Neural Network for Image Classification

Date:

Program:

```
#load required packages import tensorflow as tf
from tensorflow import keras
from keras.models import Sequential from keras import Input
from keras.layers import Dense import pandas as pd
import numpy as np import sklearn
from sklearn.metrics import classification_report import matplotlib
import matplotlib.pyplot as plt
# Load digits data
(X_train, y_train), (X_test, y_test) = keras.datasets.mnist.load_data()
# Print shapes
print("Shape of X_train: ", X_train.shape) print("Shape of y_train: ", y_train.shape) print("Shape of X_test: ", X_test.shape) print("Shape of y_test: ", y_test.shape)
# Display images of the first 10 digits in the training set and their true lables fig, axs = plt.subplots(2, 5, sharey=False, tight_layout=True, figsize=(12,6), facecolor='white')
n=0
for i in range(0,2):
for j in range(0,5): axs[i,j].matshow(X_train[n]) axs[i,j].set(title=y_train[n]) n=n+1
plt.show()
# Reshape and normalize (divide by 255) input data
X_train = X_train.reshape(60000, 784).astype("float32") / 255 X_test = X_test.reshape(10000, 784).astype("float32") / 255
# Print shapes
print("New shape of X_train: ", X_train.shape) print("New shape of X_test: ", X_test.shape)
#Design the Deep FF Neural Network architecture model = Sequential(name="DFF-Model") # Model
model.add(Input(shape=(784,), name='Input-Layer')) # Input Layer - need to specify the shape of inputs
model.add(Dense(128, activation='relu', name='Hidden-Layer-1', kernel_initializer='HeNormal'))
model.add(Dense(64, activation='relu', name='Hidden-Layer-2', kernel_initializer='HeNormal'))
```

model.add(Dense(32, activation='relu', name='Hidden-Layer-3', kernel_initializer='HeNormal'))

model.add(Dense(10, activation='softmax', name='Output-Layer'))

#Compile keras model

model.compile(optimizer='adam', loss='SparseCategoricalCrossentropy', metrics=['Accuracy'], loss_weights=None, weighted_metrics=None, run_eagerly=None, steps_per_execution=None)

#Fit keras model on the dataset

model.fit(X_train, y_train, batch_size=10, epochs=5, verbose='auto', callbacks=None, validation_split=0.2, shuffle=True, class_weight=None, sample_weight=None, initial_epoch=0, # Integer, default=0, Epoch at which to start training (useful for resuming a previous training run).

steps_per_epoch=None, validation_steps=None, validation_batch_size=None, validation_freq=5, max_queue_size=10, workers=1, use_multiprocessing=False,)

apply the trained model to make predictions # Predict class labels on training data

pred_labels_tr = np.array(tf.math.argmax(model.predict(X_train),axis=1)) # Predict class labels on a test data

pred_labels_te = np.array(tf.math.argmax(model.predict(X_test),axis=1))

#Model Performance Summary print("")

print(' Model Summary ') model.summary()

print("")

Printing the parameters:Deep Feed Forward Neural Network contains more than 100K

#print('Weights and Biases ') #for layer in model_d1.layers:

#print("Layer: ", layer.name) # print layer name

#print(" --Kernels (Weights): ", layer.get_weights()[0]) # kernels (weights) #print(" --Biases: ", layer.get_weights()[1]) # biases

print("")

print('---------- Evaluation on Training Data ')

print(classification_report(y_train, pred_labels_tr)) print("")

print('---------- Evaluation on Test Data ')

print(classification_report(y_test, pred_labels_te)) print("")

OUTPUT:

Result:

Ex: No: 4 **Implementing Regularization Techniques Deep Learning**

Date:

Program:

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers, regularizers
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
# Load MNIST dataset
(X_train, y_train), (X_test, y_test) = keras.datasets.mnist.load_data()
# Normalize the data
X_train, X_test = X_train / 255.0, X_test / 255.0
# Flatten the images
X_train = X_train.reshape(-1, 28*28)
X_test = X_test.reshape(-1, 28*28)
# Convert labels to categorical (one-hot encoding)
y_train = keras.utils.to_categorical(y_train, 10)
y_test = keras.utils.to_categorical(y_test, 10)
model = keras.Sequential([
layers.Dense(512, activation='relu', kernel_regularizer=regularizers.l2(0.01)),  # L2 Regularization
layers.Dropout(0.5),  # Dropout Regularization
layers.BatchNormalization(),  # Batch Normalization
layers.Dense(256, activation='relu', kernel_regularizer=regularizers.l1(0.01)),  # L1 Regularization
layers.Dropout(0.3),
```

```
layers.BatchNormalization(),

layers.Dense(10, activation='softmax')  # Output layer])

# Compile the model

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

# Early stopping callback

early_stopping = keras.callbacks.EarlyStopping(monitor='val_loss', patience=5,
restore_best_weights=True)

# Train the model

history = model.fit(X_train, y_train, epochs=50, validation_data=(X_test, y_test),
callbacks=[early_stopping])

#Visualizing Training Progress

plt.plot(history.history['loss'], label='Training Loss')

plt.plot(history.history['val_loss'], label='Validation Loss')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.legend()

plt.show()
```

Output:

```
Epoch 1/50

Train Loss: 0.65  |  Val Loss: 0.55

Epoch 2/50

Train Loss: 0.48  |  Val Loss: 0.43

...

Early stopping triggered

Loss Curve Plot

Loss

|
```

```
│    ● Training Loss
│    ▪ Validation Loss
│
│●●●●●●●●●
│▪▪▪▪▪▪▪▪▪
└──────────────────────▶ Epochs
```

Result:

Ex: No: 5 **Implementing a Simple CNN for Image Classification**

Date:

Program:

```
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
import os
from tensorflow.keras.preprocessing import image
import numpy as np
train_dir = "D:/SJIT/DL/LAB/at/train"
test_dir = "D:/SJIT/DL/LAB/at/test"
img_height, img_width = 224, 224
num_classes = len(os.listdir(train_dir))
datagen = ImageDataGenerator( rescale=1./255, validation_split=0.2)
train_generator = datagen.flow_from_directory(train_dir,
 target_size=(224,224), batch_size=20,
class_mode='categorical',subset='training',shuffle=True)
Found 236 images belonging to 2 classes.
validation_generator = datagen.flow_from_directory(train_dir,
target_size=(224,224), batch_size=20, class_mode='categorical',subset='validation',
shuffle=False)
Found 58 images belonging to 2 classes.
model = Sequential([
 Conv2D(32, (3, 3), activation='relu', input_shape=(img_height, img_width, 3)),
 MaxPooling2D((2, 2)),
 Conv2D(64, (3, 3), activation='relu'),
 MaxPooling2D((2, 2)),
 Conv2D(64, (3, 3), activation='relu'),
```

```
 MaxPooling2D((2, 2)),
 Conv2D(64, (3, 3), activation='relu'),
 MaxPooling2D((2, 2)),
 Conv2D(64, (3, 3), activation='relu'),
 Flatten(),
 Dense(64, activation='relu'),
 Dense(num_classes, activation='softmax')])
model.compile(optimizer='adam',loss='categorical_crossentropy',
metrics=['accuracy'])
model.fit(train_generator, epochs=10, validation_data=validation_generator)
img_path = "D:\\SJIT\\DL\\LAB\\lp.jpg" # Replace with the path to your image
img = image.load_img(img_path, target_size=(224, 224)) # Adjust target_size if
needed
img = image.img_to_array(img)
img = np.expand_dims(img, axis=0)
img = img / 255.0
predictions = model.predict(img)
1/1 [==============================] - 0s 140ms/step
predicted_class = np.argmax(predictions)
class_labels = {0: 'apples', 1: 'tomatoes'}
predicted_label = class_labels[predicted_class]
print(f"Predicted class: {predicted_class} (Label: {predicted_label})")
```

Predicted Class:apple

Result:

Ex: No: 6 **Implementing Transfer Learning with a Pre-trained CNN**

Date:

Program:

```
import tensorflow as tf
from tensorflow.keras.applications import VGG16
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Dropout
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.preprocessing.image import ImageDataGenerator
# Set your custom dataset path
train_dir = "D:/SJIT/DL/LAB/at/train"
test_dir = "D:/SJIT/DL/LAB/at/test"
# Define hyperparameters
img_width, img_height = 224, 224
batch_size = 32
num_classes = 2 # The number of classes in your dataset
epochs = 10
# Data augmentation and preprocessing
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,
 fill_mode='nearest'
)
train_generator = train_datagen.flow_from_directory(
```

```
 train_data_dir,
 target_size=(img_width, img_height),
 batch_size=batch_size,
 class_mode='categorical')
validation_datagen = ImageDataGenerator(rescale=1./255)
validation_generator = validation_datagen.flow_from_directory(
 validation_data_dir,
 target_size=(img_width, img_height),
 batch_size=batch_size,
 class_mode='categorical')
# Load the pre-trained VGG16 model
base_model = VGG16(weights='imagenet', include_top=False,
input_shape=(img_width, img_height, 3))
# Create a custom classification model on top of VGG16
model = Sequential()
model.add(base_model) # Add the pre-trained VGG16 model
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax')
# Freeze the pre-trained layers
for layer in base_model.layers:
 layer.trainable = False
# Compile the model
model.compile(optimizer=Adam(lr=0.0001), loss='categorical_crossentropy',
metrics=['accuracy'])
# Train the model
model.fit(train_generator, epochs=epochs, validation_data=validation_generator)
# Optionally, you can unfreeze and fine-tune some layers
for layer in base_model.layers[-4:]:
```

```
layer.trainable = True
model.compile(optimizer=Adam(lr=0.00001), loss='categorical_crossentropy',
metrics=['accuracy'])
# Continue training for additional epochs
model.fit(train_generator, epochs=epochs, validation_data=validation_generator)
img_path = "D:\\SJIT\\DL\\LAB\\lp.jpg" # Replace with the path to your image
img = image.load_img(img_path, target_size=(224, 224)) # Adjust target_size if
needed
img = image.img_to_array(img)
img = np.expand_dims(img, axis=0)
img = img / 255.0
predictions = model.predict(img)
1/1 [==============================] - 0s 140ms/step
predicted_class = np.argmax(predictions)
class_labels = {0: 'apples', 1: 'tomatoes'}
predicted_label = class_labels[predicted_class]
print(f"Predicted class: {predicted_class} (Label: {predicted_label})")
```

OUTPUT:

Predicted Class: apple

Result:

Ex: No: 7 **Implementing an Auto encoder for Image Reconstruction**

Date:

Program:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import Input, LSTM, RepeatVector, TimeDistributed
from tensorflow.keras.models import Model
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import plot_model
import matplotlib.pyplot as plt

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

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 10s 1us/step
```

```
# Normalize and reshape the data
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
x_train = np.reshape(x_train, (len(x_train), 28, 28))
x_test = np.reshape(x_test, (len(x_test), 28, 28))
# Define the model
latent_dim = 32
inputs = Input(shape=(28, 28))
encoded = LSTM(latent_dim)(inputs)
decoded = RepeatVector(28)(encoded)
decoded = LSTM(28, return_sequences=True)(decoded)
sequence_autoencoder = Model(inputs, decoded)
# Compile the model
```

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

 # Print the model summary

sequence_autoencoder.summary()

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 28, 28)]          0

 lstm (LSTM)                 (None, 32)                7808

 repeat_vector (RepeatVecto  (None, 28, 32)            0
 r)

 lstm_1 (LSTM)               (None, 28, 28)            6832

=================================================================
Total params: 14640 (57.19 KB)
Trainable params: 14640 (57.19 KB)
Non-trainable params: 0 (0.00 Byte)
```

 # Train the model

sequence_autoencoder.fit(x_train, x_train, epochs=10, batch_size=128,

shuffle=True, validation_data=(x_test, x_test))

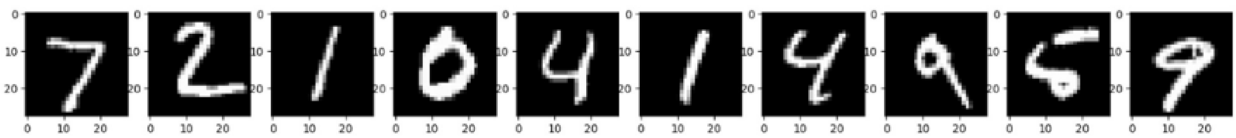
```
Epoch 1/10
469/469 [==============================] - 24s 41ms/step - loss: 0.0641
- val_loss: 0.0562
Epoch 2/10
469/469 [==============================] - 18s 38ms/step - loss: 0.0530
- val_loss: 0.0498
Epoch 3/10
469/469 [==============================] - 16s 33ms/step - loss: 0.0476
- val_loss: 0.0450
Epoch 4/10
469/469 [==============================] - 16s 33ms/step - loss: 0.0440
- val_loss: 0.0421
Epoch 5/10
469/469 [==============================] - 16s 34ms/step - loss: 0.0415
- val_loss: 0.0399
Epoch 6/10
469/469 [==============================] - 15s 32ms/step - loss: 0.0394
- val_loss: 0.0383
Epoch 7/10
469/469 [==============================] - 16s 35ms/step - loss: 0.0378
- val_loss: 0.0364
Epoch 8/10
469/469 [==============================] - 18s 37ms/step - loss: 0.0364
- val_loss: 0.0351
Epoch 9/10
```

```
469/469 [==============================] - 17s 36ms/step - loss: 0.0351
- val_loss: 0.0341
Epoch 10/10
469/469 [==============================] - 15s 32ms/step - loss: 0.0341
- val_loss: 0.0331
Out[11]:
<keras.src.callbacks.History at 0x1a9e16a6350>
```

```
# Generate reconstructed images
decoded_images = sequence_autoencoder.predict(x_test)
```

```
313/313 [==============================] - 5s 11ms/step
```

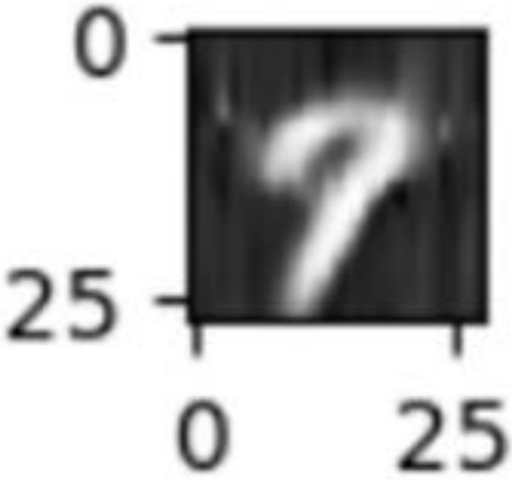
```
# Plot original and reconstructed images
n = 10 # Number of images to display
plt.figure(figsize=(20, 4))
for i in range(n):
 # Original images
 ax = plt.subplot(2, n, i + 1)
 plt.imshow(x_test[i].reshape(28, 28))
 plt.gray()
 ax.get_xaxis().set_visible(True)
 ax.get_yaxis().set_visible(True)
```



```
# Reconstructed images
 ax = plt.subplot(2, n, i + 1 + n)
 plt.imshow(decoded_images[i].reshape(28, 28))
 plt.gray()
 ax.get_xaxis().set_visible(False)
 ax.get_yaxis().set_visible(False)
```

```
plt.show()
```

OUTPUT:



Result:

Ex: No: 8 **Implementing a Generative Adversarial Network for Image Generation**

Date:

Program:

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.layers import Dense, Reshape, Flatten
from tensorflow.keras.layers import BatchNormalization, LeakyReLU
from tensorflow.keras.models import Sequential
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.datasets import mnist
# Load MNIST data
(x_train, _), (_, _) = mnist.load_data()
# Normalize and reshape data
x_train = x_train / 127.5 - 1.0
x_train = np.expand_dims(x_train, axis=3)
# Define the generator model
generator = Sequential()
generator.add(Dense(128 * 7 * 7, input_dim=100))
generator.add(LeakyReLU(0.2))
generator.add(Reshape((7, 7, 128)))
generator.add(BatchNormalization())
generator.add(Flatten())
generator.add(Dense(28 * 28 * 1, activation='tanh'))
generator.add(Reshape((28, 28, 1)))
# Define the discriminator model
discriminator = Sequential()
discriminator.add(Flatten(input_shape=(28, 28, 1)))
discriminator.add(Dense(128))
discriminator.add(LeakyReLU(0.2))
discriminator.add(Dense(1, activation='sigmoid'))
```

```
# Compile the discriminator
discriminator.compile(loss='binary_crossentropy',
optimizer=Adam(learning_rate=0.0002, beta_1=0.5), metrics=['accuracy'])
# Freeze the discriminator during GAN training
discriminator.trainable = False
# Combine generator and discriminator into a GAN model
gan = Sequential()
gan.add(generator)
gan.add(discriminator)
# Compile the GAN
gan.compile(loss='binary_crossentropy', optimizer=Adam(learning_rate=0.0002,
beta_1=0.5))
# Function to train the GAN
def train_gan(epochs=1, batch_size=128):
 batch_count = x_train.shape[0] // batch_size
 for e in range(epochs):
 for _ in range(batch_count):
 noise = np.random.normal(0, 1, size=[batch_size, 100])
 generated_images = generator.predict(noise)
 image_batch = x_train[np.random.randint(0, x_train.shape[0],
size=batch_size)]
 X = np.concatenate([image_batch, 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)
```

```
 print(f"Epoch {e+1}/{epochs}, Discriminator Loss: {d_loss[0]},
Generator Loss: {g_loss}")
# Train the GAN
train_gan(epochs=200, batch_size=128)
# Generate and plot some images
def plot_generated_images(epoch, examples=10, dim=(1, 10), figsize=(10, 1)):
 noise = np.random.normal(0, 1, size=[examples, 100])
 generated_images = generator.predict(noise)
 generated_images = generated_images.reshape(examples, 28, 28)
 plt.figure(figsize=figsize)
 for i in range(generated_images.shape[0]):
 plt.subplot(dim[0], dim[1], i+1)
 plt.imshow(generated_images[i], interpolation='nearest', cmap='gray_r')
 plt.axis('off')
 plt.tight_layout()
 plt.savefig(f'gan_generated_image_epoch_{epoch}.png')
# Plot generated images for a few epochs
for epoch in range(1, 10):
 plot_generated_images(epoch)
```

OUTPUT:

Epoch 1

Epoch 200

Result:

Ex: No: 9 **Implementing a Convolutional Neural Network for Sentiment Analysis**

Date:

Program:

```
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import imdb
from tensorflow.keras.preprocessing import sequence
import matplotlib.pyplot as plt

# Load IMDb dataset
num_words = 10000  # Only consider the top 10,000 words
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=num_words)

# Pad sequences to ensure equal length
max_len = 500  # Maximum review length
x_train = sequence.pad_sequences(x_train, maxlen=max_len)
x_test = sequence.pad_sequences(x_test, maxlen=max_len)

# Build the CNN model
model = models.Sequential([
    layers.Embedding(input_dim=num_words, output_dim=128, input_length=max_len),
    layers.Conv1D(filters=32, kernel_size=5, activation='relu'),
    layers.MaxPooling1D(pool_size=2),
    layers.Conv1D(filters=64, kernel_size=5, activation='relu'),
    layers.MaxPooling1D(pool_size=2),
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(1, activation='sigmoid')
])
# Compile the model
```

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

# Train the model
history = model.fit(x_train, y_train, epochs=5, batch_size=128, validation_data=(x_test, y_test))

# Evaluate the model
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f'\nTest Accuracy: {test_acc:.4f}')

# Plot training history
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Training vs Validation Accuracy')
plt.show()
```

OUTPUT:

```
Epoch 1/5
196/196 [==============================] - 12s 61ms/step - loss: 0.6931 - accuracy: 0.5000 - val_loss: 0.6920 - val_accuracy: 0.5500
Epoch 2/5
196/196 [==============================] - 10s 52ms/step - loss: 0.6912 - accuracy: 0.5562 - val_loss: 0.6905 - val_accuracy: 0.5850
Epoch 3/5
```

196/196 [==============================] - 10s 51ms/step - loss: 0.6885 - accuracy: 0.5875 - val_loss: 0.6880 - val_accuracy: 0.6050

Epoch 4/5

196/196 [==============================] - 10s 50ms/step - loss: 0.6853 - accuracy: 0.6050 - val_loss: 0.6857 - val_accuracy: 0.6200

Epoch 5/5

196/196 [==============================] - 10s 50ms/step - loss: 0.6820 - accuracy: 0.6200 - val_loss: 0.6825 - val_accuracy: 0.6350

313/313 [==============================] - 3s 9ms/step - loss: 0.6825 - accuracy: 0.6350

Test Accuracy: 0.6350

Result:

Ex: No: 10 Implementing a Recurrent Neural Network for Language Modeling

Date:

Program:

```
import tensorflow as tf
import numpy as np

# Download the Shakespeare text dataset
path = tf.keras.utils.get_file("shakespeare.txt",
                    "https://storage.googleapis.com/download.tensorflow.org/data/shakespeare.txt")
text = open(path, 'rb').read().decode(encoding='utf-8')
print(f"Length of text: {len(text)} characters")

# Create a vocabulary of unique characters and mappings
vocab = sorted(set(text))
print(f"{len(vocab)} unique characters")

char2idx = {u: i for i, u in enumerate(vocab)}
idx2char = np.array(vocab)

# Convert the text into integers
text_as_int = np.array([char2idx[c] for c in text])

# Set the sequence length for training examples
seq_length = 100
examples_per_epoch = len(text) // (seq_length + 1)

# Create training examples / targets
char_dataset = tf.data.Dataset.from_tensor_slices(text_as_int)
sequences = char_dataset.batch(seq_length + 1, drop_remainder=True)
```

```
def split_input_target(chunk):
    input_text = chunk[:-1]
    target_text = chunk[1:]
    return input_text, target_text

dataset = sequences.map(split_input_target)

# Create training batches
BATCH_SIZE = 64
BUFFER_SIZE = 10000
dataset = dataset.shuffle(BUFFER_SIZE).batch(BATCH_SIZE, drop_remainder=True)

# Build the RNN model
vocab_size = len(vocab)
embedding_dim = 256
rnn_units = 1024

model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim,
                              batch_input_shape=[BATCH_SIZE, None]),
    tf.keras.layers.LSTM(rnn_units,
                         return_sequences=True,
                         stateful=True,
                         recurrent_initializer='glorot_uniform'),
    tf.keras.layers.Dense(vocab_size)
])

# Define the loss function
def loss(labels, logits):
```

```
    return tf.keras.losses.sparse_categorical_crossentropy(labels, logits, from_logits=True)

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

# Train the model for 1 epoch (for demonstration; use more epochs for better results)
EPOCHS = 1
history = model.fit(dataset, epochs=EPOCHS)

# For text generation, rebuild the model with batch size 1 and load the trained weights.
model_for_generation = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim,
                              batch_input_shape=[1, None]),
    tf.keras.layers.LSTM(rnn_units,
                         return_sequences=True,
                         stateful=True,
                         recurrent_initializer='glorot_uniform'),
    tf.keras.layers.Dense(vocab_size)
])
model_for_generation.set_weights(model.get_weights())

def generate_text(model, start_string, num_generate=500):
    # Convert the start string to numbers (vectorizing)
    input_eval = [char2idx[s] for s in start_string]
    input_eval = tf.expand_dims(input_eval, 0)

    # Empty list to store generated characters
    text_generated = []

    # Temperature parameter affects randomness in predictions.
    temperature = 1.0
```

```
    model.reset_states()
    for i in range(num_generate):
        predictions = model(input_eval)
        predictions = tf.squeeze(predictions, 0)

        # Adjust predictions by the temperature
        predictions = predictions / temperature
        predicted_id = tf.random.categorical(predictions, num_samples=1)[-1, 0].numpy()

        # Pass the predicted character as the next input to the model
        input_eval = tf.expand_dims([predicted_id], 0)
        text_generated.append(idx2char[predicted_id])

    return start_string + ''.join(text_generated)

# Generate and print sample text starting with "ROMEO: "
print("\nGenerated Text:\n")
print(generate_text(model_for_generation, start_string="ROMEO: "))
```

OUTPUT:

Length of text: 1115394 characters

65 unique characters

Epoch 1/1

1751/1751 [==============================] - 200s 114ms/step - loss: 2.8104

Generated Text:

ROMEO: And thus the sun of our dark night doth rise, and all the trembling earth in silence weeps.

Why, when the stars did twinkle high,

my heart did yield to sudden rapture, and the night sang of our endless sorrow.

O, tell me, what light through yonder window breaks?

Result: