1.

```python
import numpy as np

class Perceptron:

    def __init__(self, lr=0.1, epochs=10):

        # Initialize learning rate and number of training iterations

        self.lr = lr

        self.epochs = epochs

    def fit(self, X, y):

        # Convert labels to -1 and 1

        y = np.where(y <= 0, -1, 1)

        # Initialize weights and bias

        self.w = np.zeros(X.shape[1])

        self.b = 0

        # Training loop for given number of epochs

        for _ in range(self.epochs):

            for xi, yi in zip(X, y):

                # Check if prediction is wrong (misclassified)
```

```
            if yi * (np.dot(xi, self.w) + self.b) <= 0:
                # Update weights and bias
                self.w += self.lr * yi * xi
                self.b += self.lr * yi
    def predict(self, X):
        # Predict the class labels using the sign of linear output
        return np.sign(np.dot(X, self.w) + self.b)
# Example usage
if __name__ == "__main__":
    # Input features (2D points)
    X = np.array([[1,2],[2,3],[3,4],[1,0],[0,1],[3,1]])
    # Class labels (1 or -1)
    y = np.array([1,1,1,-1,-1,-1])
    # Create Perceptron model and train it
    model = Perceptron()
    model.fit(X, y)
```

```
    # Predict on training data
    preds = model.predict(X)
    # Output predictions and actual labels
    print("Predicted labels:", preds)
    print("Actual labels:   ", y)
```

2.

```
import numpy as np
class SimpleNN:
    def __init__(self, input_size, hidden_size, output_size, lr=0.1):
        self.lr = lr
        self.w1 = np.random.randn(input_size, hidden_size) * 0.1
        self.b1 = np.zeros(hidden_size)
        self.w2 = np.random.randn(hidden_size, output_size) * 0.1
        self.b2 = np.zeros(output_size)
    def sigmoid(self, x): return 1 / (1 + np.exp(-x))
```

```
def sigmoid_deriv(self, x): return x * (1 - x)

def forward(self, X):

    self.h = self.sigmoid(np.dot(X, self.w1) + self.b1)

    return np.dot(self.h, self.w2) + self.b2  # linear output

def backward(self, X, y, out):

    err = y - out

    d_out = -2 * err

    d_hidden = np.dot(d_out, self.w2.T) * self.sigmoid_deriv(self.h)

    # update weights and biases

    self.w2 -= self.lr * np.dot(self.h.T, d_out)

    self.b2 -= self.lr * d_out.sum(axis=0)

    self.w1 -= self.lr * np.dot(X.T, d_hidden)

    self.b1 -= self.lr * d_hidden.sum(axis=0)

def fit(self, X, y, epochs):

    for i in range(epochs):

        out = self.forward(X)
```

```
            self.backward(X, y, out)
            if i % 100 == 0:
                print(f"Epoch {i}, Loss: {np.mean((y - out)**2):.5f}")
    def predict(self, X): return self.forward(X)
# Example usage
if __name__ == "__main__":
    X = np.array([[0], [1], [2], [3], [4]], dtype=float)
    y = np.array([[0], [2], [4], [6], [8]], dtype=float)
    X /= X.max(); y /= y.max()  # Normalize
    model = SimpleNN(1, 10, 1)
    model.fit(X, y, epochs=1000)
    print("Predictions:", model.predict(X))
    print("Actual:", y)
```

3.

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
```

```
from tensorflow.keras.layers import Dense, Input

from sklearn.metrics import classification_report

import matplotlib.pyplot as plt

import numpy as np

# Load MNIST dataset

(X_train, y_train), (X_test, y_test) = tf.keras.datasets.mnist.load_data()

# Show shapes

print("Train:", X_train.shape, y_train.shape)

print("Test:", X_test.shape, y_test.shape)

# Display first 10 digit images with labels

fig, axs = plt.subplots(2, 5, figsize=(10, 5))

for i, ax in enumerate(axs.flat):

    ax.imshow(X_train[i], cmap='gray')

    ax.set_title(f"Label: {y_train[i]}")

    ax.axis('off')

plt.show()
```

```
# Flatten images and normalize to [0, 1]
X_train = X_train.reshape(-1, 784).astype("float32") / 255
X_test = X_test.reshape(-1, 784).astype("float32") / 255
# Build DFF Neural Network
model = Sequential([
    Input(shape=(784,)),
    Dense(128, activation='relu'),
    Dense(64, activation='relu'),
    Dense(32, activation='relu'),
    Dense(10, activation='softmax')  # 10 classes for digits 0-9
])
# Compile model
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
# Train model
model.fit(X_train, y_train, batch_size=10, epochs=5, validation_split=0.2, verbose=1)
# Predict class labels
train_preds = np.argmax(model.predict(X_train), axis=1)
```

```
test_preds = np.argmax(model.predict(X_test), axis=1)

# Show model summary

print("\nModel Summary:")

model.summary()

# Evaluate performance

print("\nTrain Classification Report:\n", classification_report(y_train, train_preds))

print("\nTest Classification Report:\n", classification_report(y_test, test_preds))
```

4.

```
# Import required libraries

import numpy as np

import matplotlib.pyplot as plt

import tensorflow as tf

from tensorflow import keras

from tensorflow.keras import layers, regularizers

# Load and preprocess MNIST dataset
```

```
(X_train, y_train), (X_test, y_test) = keras.datasets.mnist.load_data()

X_train, X_test = X_train.reshape(-1, 28*28) / 255.0, X_test.reshape(-1, 28*28) / 255.0

y_train, y_test = keras.utils.to_categorical(y_train, 10), keras.utils.to_categorical(y_test, 10)

# Define a neural network with regularization

model = keras.Sequential([

    layers.Dense(512, activation='relu', kernel_regularizer=regularizers.l2(0.01)),

    layers.Dropout(0.5),

    layers.BatchNormalization(),

    layers.Dense(256, activation='relu', kernel_regularizer=regularizers.l1(0.01)),

    layers.Dropout(0.3),

    layers.BatchNormalization(),

    layers.Dense(10, activation='softmax')

])

# Compile the model

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

# Train with early stopping

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

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

# Plot training and validation loss

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()
```

5.

```
import tensorflow as tf

from tensorflow.keras.preprocessing.image import ImageDataGenerator, load_img, img_to_array

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

import numpy as np, os

train_dir = "D:/SJIT/DL/LAB/at/train"

img_size = (224, 224)

num_classes = len(os.listdir(train_dir))

datagen = ImageDataGenerator(rescale=1./255, validation_split=0.2)
```

```
train_gen = datagen.flow_from_directory(train_dir, target_size=img_size, batch_size=20, class_mode='categorical', subset='training')

val_gen = datagen.flow_from_directory(train_dir, target_size=img_size, batch_size=20, class_mode='categorical', subset='validation')

model = Sequential([

    Conv2D(32, (3, 3), activation='relu', input_shape=(224, 224, 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_gen, epochs=10, validation_data=val_gen)

# Prediction

img = load_img("D:/SJIT/DL/LAB/lp.jpg", target_size=img_size)
```

```
img = img_to_array(img) / 255.0
pred = model.predict(np.expand_dims(img, axis=0))
print(f"Predicted class: {np.argmax(pred)}")
```

6.

```
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, load_img, img_to_array
import numpy as np
train_dir = "D:/SJIT/DL/LAB/at/train"
val_dir = "D:/SJIT/DL/LAB/at/test"
img_size = (224, 224)
batch_size = 32
```

```
num_classes = 2

train_gen = ImageDataGenerator(rescale=1./255).flow_from_directory(train_dir, target_size=img_size, batch_size=batch_size,
class_mode='categorical')

val_gen = ImageDataGenerator(rescale=1./255).flow_from_directory(val_dir, target_size=img_size, batch_size=batch_size,
class_mode='categorical')

base_model = VGG16(weights='imagenet', include_top=False, input_shape=(224, 224, 3))

base_model.trainable = False

model = Sequential([

    base_model,

    Flatten(),

    Dense(256, activation='relu'),

    Dropout(0.5),

    Dense(num_classes, activation='softmax')

])

model.compile(optimizer=Adam(1e-4), loss='categorical_crossentropy', metrics=['accuracy'])

model.fit(train_gen, epochs=10, validation_data=val_gen)

# Fine-tuning

for layer in base_model.layers[-4:]:
```

```
    layer.trainable = True
model.compile(optimizer=Adam(1e-5), loss='categorical_crossentropy', metrics=['accuracy'])
model.fit(train_gen, epochs=10, validation_data=val_gen)
# Prediction
img = load_img("D:/SJIT/DL/LAB/lp.jpg", target_size=img_size)
img = img_to_array(img) / 255.0
pred = model.predict(np.expand_dims(img, axis=0))
print(f"Predicted class: {np.argmax(pred)}")
```

7.

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import Input, LSTM, RepeatVector
from tensorflow.keras.models import Model
from tensorflow.keras.datasets import mnist
import matplotlib.pyplot as plt
# Load MNIST dataset
```

```
(x_train, _), (x_test, _) = mnist.load_data()

# 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')

sequence_autoencoder.summary()

# Train the model
```

```
sequence_autoencoder.fit(x_train, x_train, epochs=10, batch_size=128,
                    shuffle=True, validation_data=(x_test, x_test))
# Generate reconstructed images
decoded_images = sequence_autoencoder.predict(x_test)
# Plot original and reconstructed images
n = 10
plt.figure(figsize=(20, 4))
for i in range(n):
    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)
    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()
```

8.

```
import numpy as np

import matplotlib.pyplot as plt

from tensorflow.keras.layers import Dense, Reshape, Flatten, 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)
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)
# Function to generate and plot 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')
# Generate images for several epochs
for epoch in range(1, 10):
    plot_generated_images(epoch)
```

9.

```
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

(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=num_words)

# Pad sequences

max_len = 500

x_train = sequence.pad_sequences(x_train, maxlen=max_len)

x_test = sequence.pad_sequences(x_test, maxlen=max_len)

# Build CNN model

model = models.Sequential([

    layers.Embedding(input_dim=num_words, output_dim=128, input_length=max_len),

    layers.Conv1D(32, 5, activation='relu'),

    layers.MaxPooling1D(2),

    layers.Conv1D(64, 5, activation='relu'),

    layers.MaxPooling1D(2),

    layers.Flatten(),
```

```python
    layers.Dense(64, activation='relu'),
    layers.Dense(1, activation='sigmoid')
])
# Compile model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
# Train model
history = model.fit(x_train, y_train, epochs=5, batch_size=128, validation_data=(x_test, y_test))
# Evaluate model
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f'\nTest Accuracy: {test_acc:.4f}')
# Plot accuracy
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()
```

10.

```
import tensorflow as tf

import numpy as np

# Load text

path = tf.keras.utils.get_file("shakespeare.txt", "https://storage.googleapis.com/download.tensorflow.org/data/shakespeare.txt")

text = open(path, 'rb').read().decode('utf-8')

print(f"Length of text: {len(text)} characters")

# Character mapping

vocab = sorted(set(text))

char2idx = {u: i for i, u in enumerate(vocab)}

idx2char = np.array(vocab)

text_as_int = np.array([char2idx[c] for c in text])

# Sequence settings

seq_length = 100

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):

    return chunk[:-1], chunk[1:]

dataset = sequences.map(split_input_target)

dataset = dataset.shuffle(10000).batch(64, drop_remainder=True)

# Build 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=[64, None]),

    tf.keras.layers.LSTM(rnn_units, return_sequences=True, stateful=True, recurrent_initializer='glorot_uniform'),

    tf.keras.layers.Dense(vocab_size)

])

# Loss

def loss(labels, logits):

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

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

model.fit(dataset, epochs=1)

# Generation model

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):

    input_eval = [char2idx[s] for s in start_string]

    input_eval = tf.expand_dims(input_eval, 0)

    text_generated = []

    temperature = 1.0

    model.reset_states()

    for i in range(num_generate):
```

```
        predictions = model(input_eval)

        predictions = tf.squeeze(predictions, 0)

        predictions = predictions / temperature

        predicted_id = tf.random.categorical(predictions, num_samples=1)[-1, 0].numpy()

        input_eval = tf.expand_dims([predicted_id], 0)

        text_generated.append(idx2char[predicted_id])

    return start_string + ''.join(text_generated)

# Generate sample

print("\nGenerated Text:\n")

print(generate_text(model_for_generation, start_string="ROMEO: "))
```