**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)

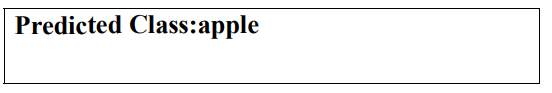
1/1 [==============================] - 0s 140ms/step

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})")



**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

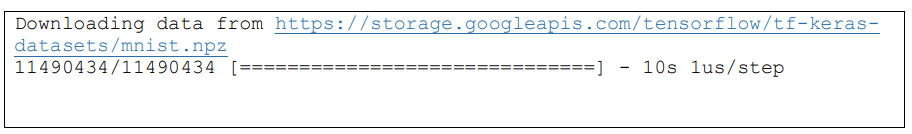
from tensorflow.keras.datasets import mnist

from tensorflow.keras.utils import plot\_model

import matplotlib.pyplot as plt

# Load MNIST dataset

(x\_train, \_), (x\_test, \_) = mnist.load\_data()



**# 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)**

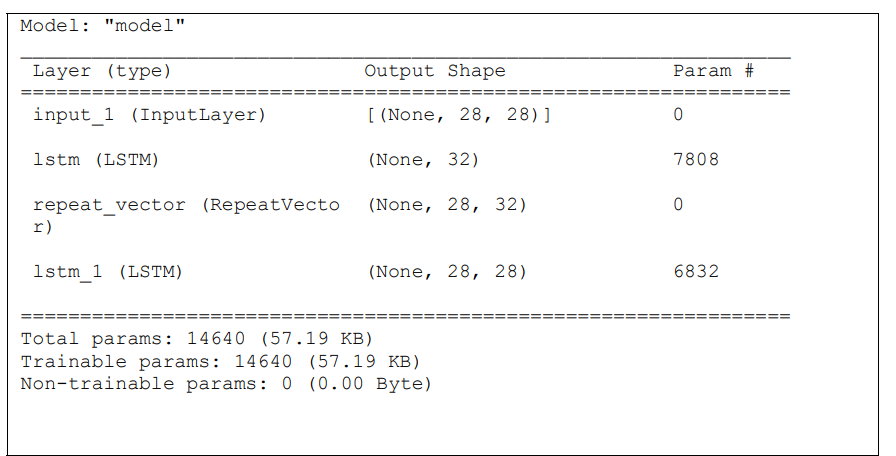
**sequence\_autoencoder = Model(inputs, decoded)**

**# Compile the model**

**sequence\_autoencoder.compile(optimizer='adam', loss='mean\_squared\_error')**

**# Print the model summary**

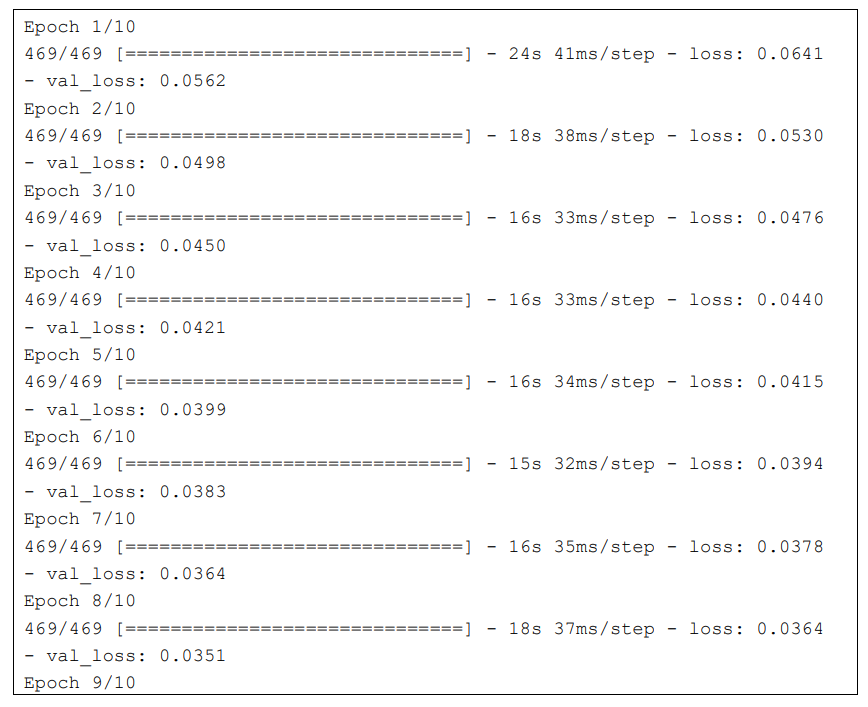
**sequence\_autoencoder.summary()**

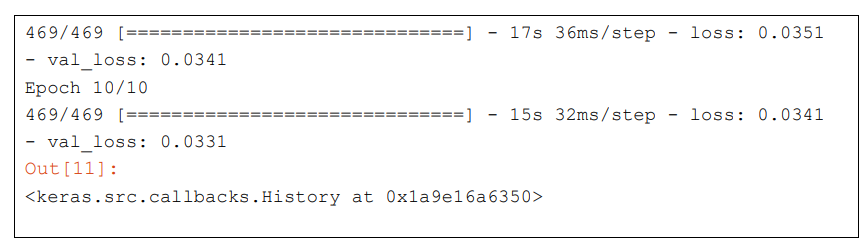


**# Train the model**

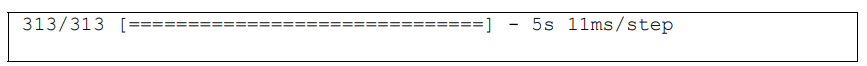
**sequence\_autoencoder.fit(x\_train, x\_train, epochs=10, batch\_size=128,**

**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 # Number of images to display

plt.figure(figsize=(20, 4))

for i in range(n):

# Original images

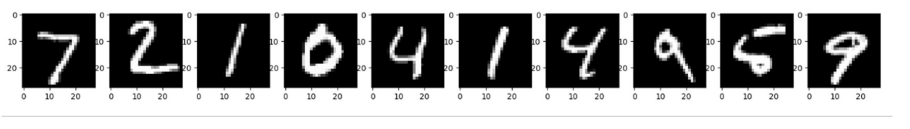
ax = plt.subplot(2, n, i + 1)

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))

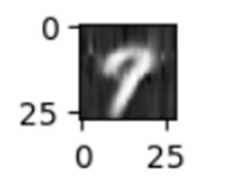
plt.gray()

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()

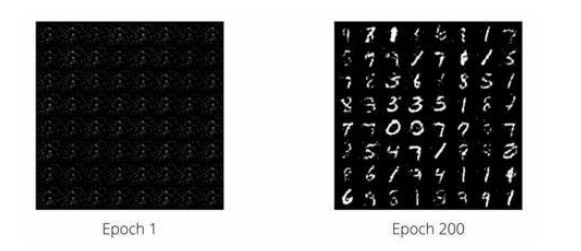
plt.savefig(f'gan\_generated\_image\_epoch\_{epoch}.png')

# Plot generated images for a few epochs

for epoch in range(1, 10):

plot\_generated\_images(epoch)

**OUTPUT:**



**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

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:**