1.

Task 1

Implement the OR Boolean logic gate using perceptron Neural Network. Inputs = x1, x2 and bias, weights should be fed into the perceptron with single Output = y. Display final weights and bias of each perceptron.

```
# TASK 1: OR Gate using Perceptron Neural Network

import numpy as np


# Input data (x1, x2)

X = np.array([

    [0, 0],

    [0, 1],

    [1, 0],

    [1, 1]

])


# OR gate output

y = np.array([0, 1, 1, 1])


# Initialize weights and bias

weights = np.zeros(2)

bias = 0

learning_rate = 0.1

epochs = 10


# Step activation function

def step_function(z):

    return 1 if z >= 0 else 0


# Training the perceptron

for _ in range(epochs):

    for i in range(len(X)):
```

```
        z = np.dot(X[i], weights) + bias
        y_pred = step_function(z)
        error = y[i] - y_pred
        weights += learning_rate * error * X[i]
        bias += learning_rate * error

# Display final weights and bias
print("Final Weights:", weights)
print("Final Bias:", bias)

# Testing
print("\nOR Gate Output")
for x in X:
    print(x, "->", step_function(np.dot(x, weights) + bias))
```

Task 2

- **Use the iris dataset Encode the input and show the new representation**
- **Decode the lossy representation for the output**

Map the input to reconstruction and visualize

```
# TASK 2: Encoding and Decoding using IRIS Dataset

import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.preprocessing import MinMaxScaler
from sklearn.decomposition import PCA
# import pandas as pd   # Uncomment if CSV is given

# ---------- Load Dataset ----------
# If IRIS CSV is given:
# df = pd.read_csv("iris.csv")
# X = df.iloc[:, :-1].values
```

```
# y = df.iloc[:, -1].values

# If CSV is NOT given:
iris = load_iris()
X = iris.data
y = iris.target

# ---------- Normalize ----------
scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X)

# ---------- Encode (Lossy Representation) ----------
pca = PCA(n_components=2)
X_encoded = pca.fit_transform(X_scaled)

# ---------- Decode (Reconstruction) ----------
X_decoded = pca.inverse_transform(X_encoded)

# ---------- Visualization ----------
plt.figure(figsize=(10,4))

plt.subplot(1,2,1)
plt.scatter(X_encoded[:,0], X_encoded[:,1], c=y)
plt.title("Encoded Representation")

plt.subplot(1,2,2)
plt.scatter(X_decoded[:,0], X_decoded[:,1], c=y)
plt.title("Reconstructed Output")

plt.show()
```

2.
Task 2

- Use the heart disease Dataset
- Create an Auto Encoder and fit it with our data using 3 neurons in the dense layer
- Display new reduced dimension values

Plot loss for different Auto encoders

```
# TASK 2: Autoencoder on Heart Disease Dataset


import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense


# ---------- Load Dataset ----------
# If CSV is given:
# df = pd.read_csv("heart.csv")


# Sample load (works if CSV not provided)
df = pd.read_csv("heart.csv")   # examiner usually gives this


X = df.values   # all features


# ---------- Normalize ----------
scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X)


# ---------- Autoencoder with 3 Neurons ----------
input_dim = X_scaled.shape[1]
```

```
input_layer = Input(shape=(input_dim,))
encoded = Dense(3, activation='relu')(input_layer)
decoded = Dense(input_dim, activation='sigmoid')(encoded)

autoencoder = Model(input_layer, decoded)
autoencoder.compile(optimizer='adam', loss='mse')

history_3 = autoencoder.fit(
    X_scaled, X_scaled,
    epochs=50,
    batch_size=16,
    verbose=0
)

# ---------- Reduced Dimension Output ----------
encoder = Model(input_layer, encoded)
X_reduced = encoder.predict(X_scaled)

print("Reduced Dimension Values (first 5 rows):")
print(X_reduced[:5])

# ---------- Autoencoders with Different Neurons ----------
losses = {}

for neurons in [2, 3, 5]:
    inp = Input(shape=(input_dim,))
    enc = Dense(neurons, activation='relu')(inp)
    dec = Dense(input_dim, activation='sigmoid')(enc)
    model = Model(inp, dec)
    model.compile(optimizer='adam', loss='mse')
    history = model.fit(X_scaled, X_scaled, epochs=50, batch_size=16, verbose=0)
```

```
    losses[neurons] = history.history['loss']


# ---------- Plot Loss ----------
plt.figure(figsize=(8,5))
for n, loss in losses.items():
    plt.plot(loss, label=f"{n} Neurons")

plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Autoencoder Loss Comparison")
plt.legend()
plt.show()
```

Task 2

- **Load the Intel Image dataset**
- **Train and test the dataset**
- **Create a model using CNN**
- **Evaluate the model using confusion matrix.**

```
# TASK 2: Intel Image Dataset using CNN

import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

# ---------- Dataset Paths ----------
train_dir = "intel_image/train"
test_dir = "intel_image/test"

# ---------- Image Preprocessing ----------
datagen = ImageDataGenerator(rescale=1./255)

train_data = datagen.flow_from_directory(
```

```
    train_dir,
    target_size=(150, 150),
    batch_size=32,
    class_mode='categorical'
)

test_data = datagen.flow_from_directory(
    test_dir,
    target_size=(150, 150),
    batch_size=32,
    class_mode='categorical',
    shuffle=False
)

# ---------- CNN Model ----------
model = Sequential([
    Conv2D(32, (3,3), activation='relu',
input_shape=(150,150,3)),
    MaxPooling2D(2,2),

    Conv2D(64, (3,3), activation='relu'),
    MaxPooling2D(2,2),

    Flatten(),
    Dense(128, activation='relu'),
    Dense(train_data.num_classes, activation='softmax')
])

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

# ---------- Train the Model ----------
history = model.fit(
    train_data,
    epochs=5,
    validation_data=test_data
)

# ---------- Test the Model ----------
y_pred = model.predict(test_data)
y_pred_classes = np.argmax(y_pred, axis=1)
y_true = test_data.classes

# ---------- Confusion Matrix ----------
cm = confusion_matrix(y_true, y_pred_classes)
```

```
disp = ConfusionMatrixDisplay(cm,
display_labels=train_data.class_indices.keys())
disp.plot(cmap="Blues")
plt.title("Confusion Matrix - Intel Image Dataset")
plt.show()

```

Task 2

- **Implement autoencoder**
- **Use the Iris Dataset**
- **Create an autoencoder and fit it with our data using 2 neurons in the dense layer**
- **Plot loss w.r.t. epoch**

Calculate reconstruction error using Mean Squared Error (MSE).

TASK 2: Autoencoder using IRIS Dataset

import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import load_iris

from sklearn.preprocessing import MinMaxScaler

from sklearn.metrics import mean_squared_error

from tensorflow.keras.models import Model

from tensorflow.keras.layers import Input, Dense

---------- Load IRIS Dataset ----------

iris = load_iris()

X = iris.data # input features

---------- Normalize Data ----------

scaler = MinMaxScaler()

X_scaled = scaler.fit_transform(X)

---------- Autoencoder Architecture ----------

input_dim = X_scaled.shape[1]

```
input_layer = Input(shape=(input_dim,))
encoded = Dense(2, activation='relu')(input_layer)   # 2 neurons (compressed)
decoded = Dense(input_dim, activation='sigmoid')(encoded)

autoencoder = Model(input_layer, decoded)
autoencoder.compile(optimizer='adam', loss='mse')

# ---------- Train Autoencoder ----------
history = autoencoder.fit(
    X_scaled, X_scaled,
    epochs=50,
    batch_size=16,
    verbose=0
)

# ---------- Plot Loss vs Epoch ----------
plt.plot(history.history['loss'])
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Autoencoder Training Loss")
plt.show()

# ---------- Reconstruction ----------
X_reconstructed = autoencoder.predict(X_scaled)

# ---------- Reconstruction Error (MSE) ----------
mse = mean_squared_error(X_scaled, X_reconstructed)
print("Reconstruction Error (MSE):", mse)
```

Task 1

Implement the NOT Boolean logic gate using perceptron Neural Network. Inputs = x1, x2 and bias, weights should be fed into the perceptron with single Output = y. Display final weights and bias of each perceptron.

Task 2

1. Use the Iris Dataset

2.Create an Auto Encoder and fit it with our data using 3 neurons in the dense layer

3. Display new reduced dimension values

4.Plot loss for different encoders

```
# TASK 1: NOT Gate using Perceptron Neural Network


import numpy as np


# Input data (x1 only)

X = np.array([[0],

              [1]])


# NOT gate output

y = np.array([1, 0])


# Initialize weight and bias

weight = 0.0

bias = 0.0

learning_rate = 0.1

epochs = 10


# Step activation function

def step(z):

    return 1 if z >= 0 else 0


# Training perceptron
```

```
for _ in range(epochs):
    for i in range(len(X)):
        z = X[i] * weight + bias
        y_pred = step(z)
        error = y[i] - y_pred
        weight += learning_rate * error * X[i]
        bias += learning_rate * error

# Display final parameters
print("Final Weight:", weight)
print("Final Bias:", bias)

# Testing NOT gate
print("\nNOT Gate Output")
for x in X:
    output = step(x * weight + bias)
    print(f"{x[0]} -> {output}")

# TASK 2: Autoencoder using IRIS Dataset

import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense

# ---------- Load IRIS Dataset ----------
iris = load_iris()
X = iris.data
```

```
# ---------- Normalize ----------
scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X)

# ---------- Autoencoder Model ----------
input_dim = X_scaled.shape[1]

input_layer = Input(shape=(input_dim,))
encoded = Dense(3, activation='relu')(input_layer)   # 3 neurons
decoded = Dense(input_dim, activation='sigmoid')(encoded)

autoencoder = Model(input_layer, decoded)
autoencoder.compile(optimizer='adam', loss='mse')

# ---------- Train ----------
history_3 = autoencoder.fit(
    X_scaled, X_scaled,
    epochs=50,
    batch_size=16,
    verbose=0
)

# ---------- Reduced Dimension Output ----------
encoder = Model(input_layer, encoded)
X_reduced = encoder.predict(X_scaled)

print("Reduced Dimension Values (first 5 rows):")
print(X_reduced[:5])

# ---------- Train Different Encoders ----------
losses = {}
```

```
for neurons in [2, 3, 5]:
    inp = Input(shape=(input_dim,))
    enc = Dense(neurons, activation='relu')(inp)
    dec = Dense(input_dim, activation='sigmoid')(enc)
    model = Model(inp, dec)
    model.compile(optimizer='adam', loss='mse')
    hist = model.fit(X_scaled, X_scaled, epochs=50, batch_size=16, verbose=0)
    losses[neurons] = hist.history['loss']


# ---------- Plot Loss ----------
plt.figure(figsize=(8,5))
for n, loss in losses.items():
    plt.plot(loss, label=f"{n} Neurons")


plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Loss Comparison for Different Autoencoders")
plt.legend()
plt.show()
```

Task 1

Implement the OR Boolean logic gate using perceptron Neural Network. Inputs = x1, x2 and bias, weights should be fed into the perceptron with single Output = y. Display final weights and bias of each perceptron.

Task 2
Use the heart disease dataset and do the following

- **Use the Dataset**
- **Create an autoencoder and fit it with our data using 2 neurons in the dense layer**
- **Plot loss w.r.t. epochs**
- **Calculate reconstruction error using Mean Squared Error (MSE).**

```
# TASK 1: OR Gate using Perceptron Neural Network

import numpy as np

# Input data (x1, x2)
X = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
])

# OR gate output
y = np.array([0, 1, 1, 1])

# Initialize weights and bias
weights = np.zeros(2)
bias = 0
learning_rate = 0.1
epochs = 10

# Step activation function
def step(z):
    return 1 if z >= 0 else 0

# Training the perceptron
for _ in range(epochs):
    for i in range(len(X)):
        z = np.dot(X[i], weights) + bias
        y_pred = step(z)
        error = y[i] - y_pred
```

```
        weights += learning_rate * error * X[i]
        bias += learning_rate * error

# Display final weights and bias
print("Final Weights:", weights)
print("Final Bias:", bias)

# Test OR gate
print("\nOR Gate Output")
for x in X:
    print(x, "->", step(np.dot(x, weights) + bias))



# TASK 2: Autoencoder using Heart Disease Dataset

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense

# ---------- Load Dataset ----------
df = pd.read_csv("heart.csv")   # dataset given in lab
X = df.values                # use all features

# ---------- Normalize Data ----------
scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X)
```

```
# ---------- Autoencoder Model ----------
input_dim = X_scaled.shape[1]

input_layer = Input(shape=(input_dim,))
encoded = Dense(2, activation='relu')(input_layer)   # 2 neurons
decoded = Dense(input_dim, activation='sigmoid')(encoded)

autoencoder = Model(input_layer, decoded)
autoencoder.compile(optimizer='adam', loss='mse')

# ---------- Train Autoencoder ----------
history = autoencoder.fit(
    X_scaled, X_scaled,
    epochs=50,
    batch_size=16,
    verbose=0
)

# ---------- Plot Loss vs Epoch ----------
plt.plot(history.history['loss'])
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Autoencoder Loss vs Epochs")
plt.show()

# ---------- Reconstruction & MSE ----------
X_reconstructed = autoencoder.predict(X_scaled)
mse = mean_squared_error(X_scaled, X_reconstructed)

print("Reconstruction Error (MSE):", mse)
```

- **Load California Housing dataset and select 2 features (e.g., Median Income, House Age) and 1 target (Median House Value).**
- **Normalize inputs and initialize a single-layer NN with random weights and bias.**
- **Perform forward propagation, calculate prediction error, Squared Error, and MSE.**
- **Update weights and bias using gradient descent.**

Plot Loss vs Weight, Loss vs Bias, and Error Surface

```
# Single Layer Neural Network on California Housing Dataset
# (Manual implementation using NumPy)

import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from sklearn.datasets import fetch_california_housing

# ------------------------------------------------
# 1. Load Dataset
# ------------------------------------------------

# CASE 1: If dataset is GIVEN as CSV file
# import pandas as pd
# df = pd.read_csv("california_housing.csv")
# X = df[["MedInc", "HouseAge"]].values     # select 2 features
# y = df["MedHouseVal"].values              # target column

# CASE 2: If dataset is NOT given (use sklearn)
data = fetch_california_housing()
X = data.data[:, [0, 1]]    # MedInc, HouseAge
y = data.target             # Median House Value

# ------------------------------------------------
```

```
# 2. Normalize Inputs
# ------------------------------------------------
scaler = MinMaxScaler()
X = scaler.fit_transform(X)
y = y.reshape(-1, 1)

# ------------------------------------------------
# 3. Initialize Weights and Bias
# ------------------------------------------------
np.random.seed(42)
w = np.random.rand(2, 1)
b = np.random.rand(1)
lr = 0.1
epochs = 50

losses = []
weights_history = []
bias_history = []

# ------------------------------------------------
# 4. Training (Forward + Gradient Descent)
# ------------------------------------------------
for _ in range(epochs):

    # Forward propagation
    y_pred = np.dot(X, w) + b

    # Error
    error = y_pred - y

    # Squared Error and MSE
```

```
    mse = np.mean(error ** 2)

    losses.append(mse)
    weights_history.append(w.copy())
    bias_history.append(b.copy())

    # Gradients
    dw = (2 / len(X)) * np.dot(X.T, error)
    db = (2 / len(X)) * np.sum(error)

    # Update
    w = w - lr * dw
    b = b - lr * db

# ------------------------------------------------
# 5. Plot Loss vs Weight
# ------------------------------------------------
weights_history = np.array(weights_history)

plt.figure()
plt.plot(weights_history[:, 0], losses)
plt.xlabel("Weight w1")
plt.ylabel("Loss (MSE)")
plt.title("Loss vs Weight")
plt.show()

# ------------------------------------------------
# 6. Plot Loss vs Bias
# ------------------------------------------------
plt.figure()
plt.plot(bias_history, losses)
```

```
plt.xlabel("Bias")
plt.ylabel("Loss (MSE)")
plt.title("Loss vs Bias")
plt.show()

# ------------------------------------------------
# 7. Error Surface Plot
# ------------------------------------------------
from mpl_toolkits.mplot3d import Axes3D

w1_vals = np.linspace(0, 1, 30)
w2_vals = np.linspace(0, 1, 30)
W1, W2 = np.meshgrid(w1_vals, w2_vals)
Z = np.zeros(W1.shape)

for i in range(W1.shape[0]):
    for j in range(W1.shape[1]):
        y_temp = X[:,0]*W1[i,j] + X[:,1]*W2[i,j] + b
        Z[i,j] = np.mean((y_temp.reshape(-1,1) - y) ** 2)

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(W1, W2, Z)
ax.set_xlabel("Weight w1")
ax.set_ylabel("Weight w2")
ax.set_zlabel("Loss (MSE)")
ax.set_title("Error Surface")
plt.show()
```

Take the dataset of Life expectancy

Initialize a neural network with random weights.

a) **Calculate output of Neural Network**

b) **Calculate squared error loss**

c) **Plot the mean squared error for each iteration in stochastic Gradient Descent.**

```
# Neural Network on Life Expectancy Dataset (Manual Implementation)

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler

# ------------------------------------------------
# 1. Load Dataset
# ------------------------------------------------

# If Life Expectancy dataset is GIVEN as CSV
# Example file: life_expectancy.csv
# df = pd.read_csv("life_expectancy.csv")
# X = df[["Adult Mortality", "BMI"]].values    # choose any 2 features
# y = df["Life expectancy"].values

# If CSV is NOT given (for reference only – needs internet)
# from sklearn.datasets import fetch_openml
# data = fetch_openml(name="life_expectancy", as_frame=True)
# df = data.frame
# X = df.iloc[:, :-1].values
# y = df.iloc[:, -1].values

# ---- LAB SAFE ASSUMPTION ----
```

```
df = pd.read_csv("life_expectancy.csv")
X = df[["Adult Mortality", "BMI"]].values
y = df["Life expectancy"].values

# ------------------------------------------------
# 2. Normalize Inputs
# ------------------------------------------------
scaler = MinMaxScaler()
X = scaler.fit_transform(X)
y = y.reshape(-1, 1)

# ------------------------------------------------
# 3. Initialize Neural Network (Single Neuron)
# ------------------------------------------------
np.random.seed(42)
w = np.random.rand(2, 1)     # random weights
b = np.random.rand(1)        # random bias
lr = 0.1
epochs = 30

mse_list = []

# ------------------------------------------------
# 4. Stochastic Gradient Descent Training
# ------------------------------------------------
for epoch in range(epochs):

    squared_errors = []

    for i in range(len(X)):  # SGD → one sample at a time
```

```
        # a) Forward pass (output)
        y_pred = np.dot(X[i], w) + b

        # b) Squared Error Loss
        error = y_pred - y[i]
        sq_error = error ** 2
        squared_errors.append(sq_error)

        # Gradient calculation
        dw = 2 * error * X[i].reshape(-1, 1)
        db = 2 * error

        # Update weights and bias
        w = w - lr * dw
        b = b - lr * db

    # c) Mean Squared Error per epoch
    mse = np.mean(squared_errors)
    mse_list.append(mse)

# ------------------------------------------------
# 5. Plot MSE vs Iterations
# ------------------------------------------------
plt.plot(mse_list)
plt.xlabel("Epochs")
plt.ylabel("Mean Squared Error")
plt.title("MSE vs Epochs (Stochastic Gradient Descent)")
plt.show()
```

Task 1

Implement the OR Boolean logic gate using perceptron Neural Network. Inputs = x1, x2 and bias, weights should be fed into the perceptron with single Output = y. Display final weights and bias of each perceptron.

Task 2

Implement autoencoder

- **Use the Wine Dataset**
- **Create an autoencoder and fit it with our data using 3 neurons in the dense layer**
- **Calculate loss w.r.t to different epochs and plot using line graph.**

TASK 1: OR Gate using Perceptron Neural Network

```
import numpy as np

# Input data (x1, x2)
X = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
])

# OR gate output
y = np.array([0, 1, 1, 1])

# Initialize weights and bias
weights = np.zeros(2)
bias = 0
learning_rate = 0.1
epochs = 10
```

```
# Step activation function
def step(z):
    return 1 if z >= 0 else 0

# Train perceptron
for _ in range(epochs):
    for i in range(len(X)):
        z = np.dot(X[i], weights) + bias
        y_pred = step(z)
        error = y[i] - y_pred
        weights += learning_rate * error * X[i]
        bias += learning_rate * error

# Display final weights and bias
print("Final Weights:", weights)
print("Final Bias:", bias)

# Test OR gate
print("\nOR Gate Output")
for x in X:
    print(x, "->", step(np.dot(x, weights) + bias))

# TASK 2: Autoencoder using Wine Dataset

import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_wine
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense
```

```
# ---------- Load Wine Dataset ----------
wine = load_wine()
X = wine.data   # input features

# ---------- Normalize Data ----------
scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X)

# ---------- Autoencoder Architecture ----------
input_dim = X_scaled.shape[1]

input_layer = Input(shape=(input_dim,))
encoded = Dense(3, activation='relu')(input_layer)   # 3 neurons
decoded = Dense(input_dim, activation='sigmoid')(encoded)

autoencoder = Model(input_layer, decoded)
autoencoder.compile(optimizer='adam', loss='mse')

# ---------- Train Autoencoder ----------
history = autoencoder.fit(
    X_scaled, X_scaled,
    epochs=50,
    batch_size=16,
    verbose=0
)

# ---------- Plot Loss vs Epoch ----------
plt.plot(history.history['loss'])
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Autoencoder Loss vs Epochs (Wine Dataset)")
```

plt.show()

Implement Self Organizing Map for anomaly Detection

- **Use Credit Card Applications Dataset:**
- **Detect fraud customers in the dataset using SOM and perform hyperparameter tuning**
- **Show map and use markers to distinguish frauds.**

```
# Self Organizing Map (SOM) for Fraud Detection
# Credit Card Applications Dataset

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from minisom import MiniSom

# ------------------------------------------------
# 1. Load Dataset
# ------------------------------------------------

# Dataset given by examiner
df = pd.read_csv("credit_card_applications.csv")

X = df.iloc[:, :-1].values   # customer features
y = df.iloc[:, -1].values    # approval (0/1)

# ------------------------------------------------
# 2. Normalize Data
# ------------------------------------------------
scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X)
```

```
# ------------------------------------------------
# 3. Hyperparameter Tuning (Map Size)
# ------------------------------------------------
map_sizes = [(8,8), (10,10), (12,12)]
best_som = None
best_avg_distance = -1

for size in map_sizes:
    som = MiniSom(
        x=size[0],
        y=size[1],
        input_len=X_scaled.shape[1],
        sigma=1.0,
        learning_rate=0.5
    )
    som.random_weights_init(X_scaled)
    som.train_random(X_scaled, num_iteration=100)

    avg_distance = np.mean(som.distance_map())

    if avg_distance > best_avg_distance:
        best_avg_distance = avg_distance
        best_som = som
        best_size = size

print("Best SOM size:", best_size)

# ------------------------------------------------
# 4. Fraud Detection using Distance Map
# ------------------------------------------------
```

```
distance_map = best_som.distance_map()

threshold = np.percentile(distance_map, 90)  # top 10% anomalies

frauds = []

for i, x in enumerate(X_scaled):
    w = best_som.winner(x)
    if distance_map[w] > threshold:
        frauds.append(i)

print("Number of fraud customers detected:", len(frauds))

# ------------------------------------------------
# 5. Visualization with Markers
# ------------------------------------------------
plt.figure(figsize=(7,7))
plt.title("Self Organizing Map - Fraud Detection")

plt.imshow(distance_map, cmap='coolwarm')
plt.colorbar()

markers = ['o', 's']
colors = ['g', 'r']

for i, x in enumerate(X_scaled):
    w = best_som.winner(x)

    if i in frauds:
        plt.plot(w[0]+0.5, w[1]+0.5, 'rx', markersize=12)
    else:
```

```
    plt.plot(w[0]+0.5, w[1]+0.5, 'go', markersize=3)

plt.show()
```

Train a small neural network (dataset - MNIST classification)

Compare the optimizers:

1. **SGD**
2. **SGD + Momentum**
3. **Adam**

Plot:

1. **Training loss vs epochs**
2. **Accuracy vs epochs**

```
# MNIST Classification – Optimizer Comparison
# SGD vs SGD + Momentum vs Adam

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.optimizers import SGD, Adam
from tensorflow.keras.utils import to_categorical

# ------------------------------------------------
# 1. Load Dataset
# ------------------------------------------------

# CASE 1: If MNIST dataset is GIVEN as CSV files (LAB SAFE)
# Files usually: mnist_train.csv, mnist_test.csv
# First column = label, remaining 784 columns = pixels
```

```
# train_df = pd.read_csv("mnist_train.csv")
# test_df = pd.read_csv("mnist_test.csv")

# X_train = train_df.iloc[:, 1:].values.reshape(-1, 28, 28) / 255.0
# y_train = train_df.iloc[:, 0].values

# X_test = test_df.iloc[:, 1:].values.reshape(-1, 28, 28) / 255.0
# y_test = test_df.iloc[:, 0].values



# CASE 2: If dataset is NOT given (needs internet)
from tensorflow.keras.datasets import mnist
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train = X_train / 255.0
X_test = X_test / 255.0

# One-hot encoding
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)

# ------------------------------------------------
# 2. Model Definition
# ------------------------------------------------
def create_model():
    model = Sequential([
        Flatten(input_shape=(28,28)),
        Dense(128, activation='relu'),
        Dense(10, activation='softmax')
    ])
    return model
```

```
epochs = 10
histories = {}

# ------------------------------------------------
# 3. SGD Optimizer
# ------------------------------------------------
model_sgd = create_model()
model_sgd.compile(
    optimizer=SGD(learning_rate=0.01),
    loss='categorical_crossentropy',
    metrics=['accuracy']
)
histories['SGD'] = model_sgd.fit(
    X_train, y_train,
    epochs=epochs,
    verbose=0
)

# ------------------------------------------------
# 4. SGD + Momentum
# ------------------------------------------------
model_momentum = create_model()
model_momentum.compile(
    optimizer=SGD(learning_rate=0.01, momentum=0.9),
    loss='categorical_crossentropy',
    metrics=['accuracy']
)
histories['SGD + Momentum'] = model_momentum.fit(
    X_train, y_train,
    epochs=epochs,
    verbose=0
```

```
)

# ------------------------------------------------
# 5. Adam Optimizer
# ------------------------------------------------
model_adam = create_model()
model_adam.compile(
    optimizer=Adam(learning_rate=0.001),
    loss='categorical_crossentropy',
    metrics=['accuracy']
)
histories['Adam'] = model_adam.fit(
    X_train, y_train,
    epochs=epochs,
    verbose=0
)

# ------------------------------------------------
# 6. Plot Training Loss vs Epochs
# ------------------------------------------------
plt.figure()
for name, history in histories.items():
    plt.plot(history.history['loss'], label=name)

plt.xlabel("Epochs")
plt.ylabel("Training Loss")
plt.title("Training Loss vs Epochs")
plt.legend()
plt.show()

# ------------------------------------------------
```

```
# 7. Plot Accuracy vs Epochs
# ------------------------------------------------
plt.figure()
for name, history in histories.items():
    plt.plot(history.history['accuracy'], label=name)

plt.xlabel("Epochs")
plt.ylabel("Training Accuracy")
plt.title("Accuracy vs Epochs")
plt.legend()
plt.show()
```

Train a small neural network (dataset - MNIST classification)

Compare the optimizers:

4. **SGD**
5. **SGD + Momentum**
6. **Adam**

Plot:

3. **Training loss vs epochs**
4. **Accuracy vs epochs**

```
# MNIST Classification – Optimizer Comparison
# SGD vs SGD + Momentum vs Adam

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.optimizers import SGD, Adam
from tensorflow.keras.utils import to_categorical

# ------------------------------------------------
```

```
# 1. Load Dataset
# ------------------------------------------------

# CASE 1: If MNIST dataset is GIVEN as CSV files (LAB SAFE)
# Files usually: mnist_train.csv, mnist_test.csv
# First column = label, remaining 784 columns = pixels

# train_df = pd.read_csv("mnist_train.csv")
# test_df = pd.read_csv("mnist_test.csv")

# X_train = train_df.iloc[:, 1:].values.reshape(-1, 28, 28) / 255.0
# y_train = train_df.iloc[:, 0].values

# X_test = test_df.iloc[:, 1:].values.reshape(-1, 28, 28) / 255.0
# y_test = test_df.iloc[:, 0].values



# CASE 2: If dataset is NOT given (needs internet)
from tensorflow.keras.datasets import mnist
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train = X_train / 255.0
X_test = X_test / 255.0

# One-hot encoding
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)

# ------------------------------------------------
# 2. Model Definition
# ------------------------------------------------
def create_model():
```

```
    model = Sequential([
        Flatten(input_shape=(28,28)),
        Dense(128, activation='relu'),
        Dense(10, activation='softmax')
    ])
    return model

epochs = 10
histories = {}

# ---------------------------------------------
# 3. SGD Optimizer
# ---------------------------------------------
model_sgd = create_model()
model_sgd.compile(
    optimizer=SGD(learning_rate=0.01),
    loss='categorical_crossentropy',
    metrics=['accuracy']
)
histories['SGD'] = model_sgd.fit(
    X_train, y_train,
    epochs=epochs,
    verbose=0
)

# ---------------------------------------------
# 4. SGD + Momentum
# ---------------------------------------------
model_momentum = create_model()
model_momentum.compile(
    optimizer=SGD(learning_rate=0.01, momentum=0.9),
```

```
    loss='categorical_crossentropy',
    metrics=['accuracy']
)
histories['SGD + Momentum'] = model_momentum.fit(
    X_train, y_train,
    epochs=epochs,
    verbose=0
)

# ------------------------------------------------
# 5. Adam Optimizer
# ------------------------------------------------
model_adam = create_model()
model_adam.compile(
    optimizer=Adam(learning_rate=0.001),
    loss='categorical_crossentropy',
    metrics=['accuracy']
)
histories['Adam'] = model_adam.fit(
    X_train, y_train,
    epochs=epochs,
    verbose=0
)

# ------------------------------------------------
# 6. Plot Training Loss vs Epochs
# ------------------------------------------------
plt.figure()
for name, history in histories.items():
    plt.plot(history.history['loss'], label=name)
```

```
plt.xlabel("Epochs")
plt.ylabel("Training Loss")
plt.title("Training Loss vs Epochs")
plt.legend()
plt.show()

# ------------------------------------------------
# 7. Plot Accuracy vs Epochs
# ------------------------------------------------
plt.figure()
for name, history in histories.items():
    plt.plot(history.history['accuracy'], label=name)

plt.xlabel("Epochs")
plt.ylabel("Training Accuracy")
plt.title("Accuracy vs Epochs")
plt.legend()
plt.show()
```

1. **Use MNIST or IRIS/ Cifar-10 Dataset**
2. **Train a model with and without data augmentation (horizontal flip, rotation, noise).**
3. **Compare generalization performance on the validation set. (Accuracy & Error)**
4. **Evaluate the model using confusion matrix, precision, recall**

```
# Data Augmentation vs No Augmentation on MNIST (ONE FLOW)

import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.utils import to_categorical
```

```
from sklearn.metrics import confusion_matrix, precision_score, recall_score

# ------------------------------
# Load Dataset
# ------------------------------

# If dataset is GIVEN as CSV (commented for lab use)
# import pandas as pd
# train_df = pd.read_csv("mnist_train.csv")
# test_df = pd.read_csv("mnist_test.csv")
# X_train = train_df.iloc[:,1:].values.reshape(-1,28,28)/255.0
# y_train = train_df.iloc[:,0].values
# X_test = test_df.iloc[:,1:].values.reshape(-1,28,28)/255.0
# y_test = test_df.iloc[:,0].values

# If dataset is NOT given
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train = X_train / 255.0
X_test = X_test / 255.0

y_train_cat = to_categorical(y_train)
y_test_cat = to_categorical(y_test)

# ------------------------------
# Model Definition
# ------------------------------
def create_model():
    model = Sequential([
        Flatten(input_shape=(28,28)),
        Dense(128, activation='relu'),
        Dense(10, activation='softmax')
```

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

# ----------------------------
# Training WITHOUT Augmentation
# ----------------------------
model_no_aug = create_model()
history_no_aug = model_no_aug.fit(
    X_train, y_train_cat,
    epochs=5,
    validation_data=(X_test, y_test_cat),
    verbose=0
)

# ----------------------------
# Training WITH Augmentation
# ----------------------------
datagen = ImageDataGenerator(
    rotation_range=15,
    horizontal_flip=True,
    zoom_range=0.1,
    width_shift_range=0.1,
    height_shift_range=0.1
)

model_aug = create_model()
history_aug = model_aug.fit(
    datagen.flow(X_train, y_train_cat, batch_size=64),
```

```
    epochs=5,
    validation_data=(X_test, y_test_cat),
    verbose=0
)

# ----------------------------
# Compare Accuracy & Error
# ----------------------------
plt.figure(figsize=(10,4))

plt.subplot(1,2,1)
plt.plot(history_no_aug.history['val_accuracy'], label='No Augmentation')
plt.plot(history_aug.history['val_accuracy'], label='With Augmentation')
plt.xlabel("Epochs")
plt.ylabel("Validation Accuracy")
plt.legend()
plt.title("Accuracy Comparison")

plt.subplot(1,2,2)
plt.plot(history_no_aug.history['val_loss'], label='No Augmentation')
plt.plot(history_aug.history['val_loss'], label='With Augmentation')
plt.xlabel("Epochs")
plt.ylabel("Validation Error")
plt.legend()
plt.title("Error Comparison")

plt.show()

# ----------------------------
# Evaluation Metrics
# ----------------------------
```

```
y_pred = model_aug.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)

cm = confusion_matrix(y_test, y_pred_classes)
precision = precision_score(y_test, y_pred_classes, average='macro')
recall = recall_score(y_test, y_pred_classes, average='macro')

print("Confusion Matrix:\n", cm)
print("Precision:", precision)
print("Recall:", recall)
```

1. **Implement a tiny SimCLR framework using a small dataset (e.g., CIFAR-10 subset).**
2. **Use data augmentations.**
3. **Implement the NT-Xent loss function to compute similarity between pairs.**
4. **Compare Data with and without Augmentation.**

```
# ============================================================
# TINY SimCLR IMPLEMENTATION (FULL ONE-BLOCK SOLUTION)
# Dataset: CIFAR-10 (or image files if dataset is given)
# ============================================================

import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras import layers, models
from tensorflow.keras.optimizers import Adam

# ------------------------------------------------------------
# 1. LOAD DATASET
# ------------------------------------------------------------

# CASE 1: If DATASET IS GIVEN as IMAGE FILES (OFFLINE / LAB)
```

```
# Folder structure example:
# dataset/
#  ├── class1/
#  ├── class2/
#  ├── class3/
#
# Use this code:
#
# train_ds = tf.keras.utils.image_dataset_from_directory(
#     "dataset/",
#     image_size=(32, 32),
#     batch_size=256,
#     shuffle=True
# )
#
# x_train = tf.concat([x for x, y in train_ds], axis=0)
# x_train = x_train / 255.0
#
# Labels are NOT used in SimCLR (self-supervised)

# CASE 2: If DATASET IS NOT GIVEN (use built-in CIFAR-10)
from tensorflow.keras.datasets import cifar10
(x_train, _), _ = cifar10.load_data()
x_train = x_train[:5000] / 255.0   # small subset for fast training

# ------------------------------------------------------------
# 2. DATA AUGMENTATION (SimCLR style)
# ------------------------------------------------------------

data_augmentation = tf.keras.Sequential([
    layers.RandomFlip("horizontal"),
```

```
    layers.RandomRotation(0.1),
    layers.RandomZoom(0.1)
])

def augment(images):
    return data_augmentation(images, training=True)

# ------------------------------------------------------------
# 3. ENCODER NETWORK (Small CNN)
# ------------------------------------------------------------

def get_encoder():
    model = models.Sequential([
        layers.Conv2D(32, 3, activation='relu', input_shape=(32,32,3)),
        layers.MaxPooling2D(),
        layers.Conv2D(64, 3, activation='relu'),
        layers.MaxPooling2D(),
        layers.Flatten(),
        layers.Dense(128, activation='relu')
    ])
    return model

encoder = get_encoder()

# ------------------------------------------------------------
# 4. PROJECTION HEAD
# ------------------------------------------------------------

projection_head = models.Sequential([
    layers.Dense(64, activation='relu'),
    layers.Dense(32)
```

```
])

# ------------------------------------------------------------
# 5. NT-XENT LOSS FUNCTION
# ------------------------------------------------------------

def nt_xent_loss(z1, z2, temperature=0.5):
    z1 = tf.math.l2_normalize(z1, axis=1)
    z2 = tf.math.l2_normalize(z2, axis=1)

    representations = tf.concat([z1, z2], axis=0)
    similarity_matrix = tf.matmul(
        representations, representations, transpose_b=True
    )

    batch_size = tf.shape(z1)[0]
    labels = tf.concat([
        tf.range(batch_size, 2 * batch_size),
        tf.range(0, batch_size)
    ], axis=0)

    logits = similarity_matrix / temperature
    loss = tf.keras.losses.sparse_categorical_crossentropy(
        labels, logits, from_logits=True
    )
    return tf.reduce_mean(loss)

# ------------------------------------------------------------
# 6. TRAINING STEP
# ------------------------------------------------------------
```

```
optimizer = Adam(0.001)


@tf.function
def train_step(images, use_augmentation=True):
    with tf.GradientTape() as tape:

        if use_augmentation:
            x1 = augment(images)
            x2 = augment(images)
        else:
            x1 = images
            x2 = images

        h1 = encoder(x1)
        h2 = encoder(x2)

        z1 = projection_head(h1)
        z2 = projection_head(h2)

        loss = nt_xent_loss(z1, z2)

    grads = tape.gradient(
        loss,
        encoder.trainable_variables + projection_head.trainable_variables
    )
    optimizer.apply_gradients(
        zip(grads, encoder.trainable_variables + projection_head.trainable_variables)
    )

    return loss
```

```
# ------------------------------------------------------------
# 7. TRAIN WITH & WITHOUT AUGMENTATION
# ------------------------------------------------------------

epochs = 10
loss_with_aug = []
loss_without_aug = []

for epoch in range(epochs):
    l_aug = train_step(x_train, use_augmentation=True)
    l_no_aug = train_step(x_train, use_augmentation=False)

    loss_with_aug.append(l_aug.numpy())
    loss_without_aug.append(l_no_aug.numpy())

    print(f"Epoch {epoch+1} | With Aug: {l_aug.numpy():.4f} | Without Aug: {l_no_aug.numpy():.4f}")

# ------------------------------------------------------------
# 8. PLOT COMPARISON
# ------------------------------------------------------------

plt.plot(loss_with_aug, label="With Augmentation")
plt.plot(loss_without_aug, label="Without Augmentation")
plt.xlabel("Epochs")
plt.ylabel("NT-Xent Loss")
plt.title("SimCLR: Augmentation vs No Augmentation")
plt.legend()
plt.show()
```

1. **Use a pretrained model (e.g., ResNet-50 or MobileNet).**
2. **Freeze its encoder.**
3. **Train a classifier head on a different dataset (e.g., Flowers dataset).**

Compare accuracy with fine tuning

```
# ============================================================
# TRANSFER LEARNING WITH PRETRAINED MODEL + FINE TUNING
# Model: MobileNetV2
# Dataset: Flowers (or any given image dataset)
# ============================================================

import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.keras import layers, models, optimizers

# ------------------------------------------------------------
# 1. LOAD DATASET
# ------------------------------------------------------------

# CASE 1: If DATASET IS GIVEN as IMAGE FILES (LAB / OFFLINE)
# Folder structure example:
# flowers/
#  ├── daisy/
#  ├── dandelion/
#  ├── roses/
#  ├── sunflowers/
#  └── tulips/
#
# train_ds = tf.keras.utils.image_dataset_from_directory(
#     "flowers/",
#     image_size=(224, 224),
```

```
#     batch_size=32,
#     validation_split=0.2,
#     subset="training",
#     seed=123
# )
#
# val_ds = tf.keras.utils.image_dataset_from_directory(
#     "flowers/",
#     image_size=(224, 224),
#     batch_size=32,
#     validation_split=0.2,
#     subset="validation",
#     seed=123
# )
#
# class_names = train_ds.class_names

# CASE 2: If DATASET IS NOT GIVEN (use TensorFlow Flowers dataset)
data_dir = tf.keras.utils.get_file(
    "flower_photos",
    "https://storage.googleapis.com/download.tensorflow.org/example_images/flower_photos.tgz",
    untar=True
)

train_ds = tf.keras.utils.image_dataset_from_directory(
    data_dir,
    image_size=(224, 224),
    batch_size=32,
    validation_split=0.2,
    subset="training",
    seed=123
```

```
)

val_ds = tf.keras.utils.image_dataset_from_directory(
    data_dir,
    image_size=(224, 224),
    batch_size=32,
    validation_split=0.2,
    subset="validation",
    seed=123
)

class_names = train_ds.class_names
num_classes = len(class_names)

# Normalize images
normalization_layer = layers.Rescaling(1./255)
train_ds = train_ds.map(lambda x, y: (normalization_layer(x), y))
val_ds = val_ds.map(lambda x, y: (normalization_layer(x), y))

# ------------------------------------------------------------
# 2. LOAD PRETRAINED MODEL (ENCODER)
# ------------------------------------------------------------

base_model = tf.keras.applications.MobileNetV2(
    input_shape=(224, 224, 3),
    include_top=False,
    weights="imagenet"
)

# ------------------------------------------------------------
# 3. FREEZE ENCODER
```

```
# ------------------------------------------------------------
base_model.trainable = False

# ------------------------------------------------------------
# 4. CLASSIFIER HEAD
# ------------------------------------------------------------

model = models.Sequential([
    base_model,
    layers.GlobalAveragePooling2D(),
    layers.Dense(128, activation="relu"),
    layers.Dense(num_classes, activation="softmax")
])

model.compile(
    optimizer=optimizers.Adam(learning_rate=0.001),
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"]
)

# ------------------------------------------------------------
# 5. TRAIN CLASSIFIER HEAD ONLY
# ------------------------------------------------------------
history_frozen = model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=5
)

# ------------------------------------------------------------
# 6. FINE TUNING (UNFREEZE TOP LAYERS)
```

```
# ------------------------------------------------------------

base_model.trainable = True

# Freeze lower layers, fine-tune higher layers
for layer in base_model.layers[:-50]:
    layer.trainable = False

model.compile(
    optimizer=optimizers.Adam(learning_rate=1e-5),
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"]
)

history_finetune = model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=5
)

# ------------------------------------------------------------
# 7. COMPARE ACCURACY
# ------------------------------------------------------------

plt.plot(history_frozen.history['val_accuracy'], label="Frozen Encoder")
plt.plot(history_finetune.history['val_accuracy'], label="Fine Tuned")
plt.xlabel("Epochs")
plt.ylabel("Validation Accuracy")
plt.title("Accuracy Comparison: Frozen vs Fine Tuning")
plt.legend()
plt.show()
```

Choose two datasets with different distributions (dogs &cats , cars).

1. **Resize images to the required input size of the chosen pre-trained model.**
2. **Load Pre-trained Model (LeNet-5 or VGG-16)**
3. **Compare the performances of all the models and visualize**
4. **Write down your observations and conclusions**

```
# ============================================================
# COMPARISON OF PRETRAINED MODELS ON TWO DATASETS
# Datasets: Dogs vs Cats  |  Cars
# Model: VGG-16
# ============================================================

import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.keras import layers, models, optimizers

# ------------------------------------------------------------
# 1. LOAD & RESIZE DATASETS
# ------------------------------------------------------------

# CASE: DATASETS GIVEN AS IMAGE FOLDERS (LAB / EXAM CASE)

# Dataset 1: Dogs vs Cats
# Folder structure:
# dogs_cats/
#  ├── dogs/
# └── cats/

dogs_cats_train = tf.keras.utils.image_dataset_from_directory(
    "dogs_cats/",
    image_size=(224, 224),   # VGG-16 input size
    batch_size=32,
```

```
    validation_split=0.2,
    subset="training",
    seed=123
)

dogs_cats_val = tf.keras.utils.image_dataset_from_directory(
    "dogs_cats/",
    image_size=(224, 224),
    batch_size=32,
    validation_split=0.2,
    subset="validation",
    seed=123
)

# Dataset 2: Cars
# Folder structure:
# cars/
# ├── class1/
# ├── class2/
# └── ...

cars_train = tf.keras.utils.image_dataset_from_directory(
    "cars/",
    image_size=(224, 224),
    batch_size=32,
    validation_split=0.2,
    subset="training",
    seed=123
)

cars_val = tf.keras.utils.image_dataset_from_directory(
```

```
    "cars/",
    image_size=(224, 224),
    batch_size=32,
    validation_split=0.2,
    subset="validation",
    seed=123
)

# Normalize images
normalizer = layers.Rescaling(1./255)
dogs_cats_train = dogs_cats_train.map(lambda x, y: (normalizer(x), y))
dogs_cats_val = dogs_cats_val.map(lambda x, y: (normalizer(x), y))
cars_train = cars_train.map(lambda x, y: (normalizer(x), y))
cars_val = cars_val.map(lambda x, y: (normalizer(x), y))

# ------------------------------------------------------------
# 2. LOAD PRETRAINED MODEL (VGG-16)
# ------------------------------------------------------------

base_model = tf.keras.applications.VGG16(
    input_shape=(224, 224, 3),
    include_top=False,
    weights="imagenet"
)

base_model.trainable = False   # freeze encoder

# ------------------------------------------------------------
# 3. MODEL CREATION FUNCTION
# ------------------------------------------------------------
```

```
def build_model(num_classes):
    model = models.Sequential([
        base_model,
        layers.GlobalAveragePooling2D(),
        layers.Dense(256, activation='relu'),
        layers.Dense(num_classes, activation='softmax')
    ])
    model.compile(
        optimizer=optimizers.Adam(0.001),
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )
    return model

# ------------------------------------------------------------
# 4. TRAIN ON DOGS vs CATS
# ------------------------------------------------------------

dogs_cats_classes = len(dogs_cats_train.class_names)
model_dogs_cats = build_model(dogs_cats_classes)

history_dogs_cats = model_dogs_cats.fit(
    dogs_cats_train,
    validation_data=dogs_cats_val,
    epochs=5
)

# ------------------------------------------------------------
# 5. TRAIN ON CARS DATASET
# ------------------------------------------------------------
```

```
cars_classes = len(cars_train.class_names)
model_cars = build_model(cars_classes)

history_cars = model_cars.fit(
    cars_train,
    validation_data=cars_val,
    epochs=5
)

# ------------------------------------------------------------
# 6. PERFORMANCE COMPARISON (VISUALIZATION)
# ------------------------------------------------------------

plt.figure(figsize=(10,4))

plt.subplot(1,2,1)
plt.plot(history_dogs_cats.history['val_accuracy'], label='Dogs vs Cats')
plt.plot(history_cars.history['val_accuracy'], label='Cars')
plt.xlabel("Epochs")
plt.ylabel("Validation Accuracy")
plt.title("Accuracy Comparison")
plt.legend()

plt.subplot(1,2,2)
plt.plot(history_dogs_cats.history['val_loss'], label='Dogs vs Cats')
plt.plot(history_cars.history['val_loss'], label='Cars')
plt.xlabel("Epochs")
plt.ylabel("Validation Loss")
plt.title("Loss Comparison")
plt.legend()
```

plt.show()

Choose two datasets with different distributions (dogs &cats , cars).

1. **Resize images to the required input size of the chosen pre-trained model.**
2. **Load Pre-trained Model (AlexNet or ResNet-50)**
3. **Compare the performances of all the models and visualize**
4. **Write down your observations and conclusions**

```
# ============================================================
# Comparison of Pretrained ResNet-50 on Two Datasets
# Datasets: Dogs & Cats  |  Cars
# ============================================================

import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.keras import layers, models, optimizers

# ------------------------------------------------------------
# 1. LOAD & RESIZE DATASETS
# ------------------------------------------------------------

# DATASET 1: Dogs vs Cats
# Folder structure:
# dogs_cats/
#  ├── dogs/
# └── cats/

dogs_cats_train = tf.keras.utils.image_dataset_from_directory(
    "dogs_cats/",
    image_size=(224, 224),     # ResNet-50 input size
    batch_size=32,
    validation_split=0.2,
```

```
    subset="training",
    seed=42
)

dogs_cats_val = tf.keras.utils.image_dataset_from_directory(
    "dogs_cats/",
    image_size=(224, 224),
    batch_size=32,
    validation_split=0.2,
    subset="validation",
    seed=42
)

# DATASET 2: Cars
# Folder structure:
# cars/
# ├── class1/
# ├── class2/
# └── ...

cars_train = tf.keras.utils.image_dataset_from_directory(
    "cars/",
    image_size=(224, 224),
    batch_size=32,
    validation_split=0.2,
    subset="training",
    seed=42
)

cars_val = tf.keras.utils.image_dataset_from_directory(
    "cars/",
```

```
    image_size=(224, 224),
    batch_size=32,
    validation_split=0.2,
    subset="validation",
    seed=42
)

# Normalize images
rescale = layers.Rescaling(1./255)
dogs_cats_train = dogs_cats_train.map(lambda x, y: (rescale(x), y))
dogs_cats_val = dogs_cats_val.map(lambda x, y: (rescale(x), y))
cars_train = cars_train.map(lambda x, y: (rescale(x), y))
cars_val = cars_val.map(lambda x, y: (rescale(x), y))

# ------------------------------------------------------------
# 2. LOAD PRETRAINED MODEL (RESNET-50)
# ------------------------------------------------------------

base_model = tf.keras.applications.ResNet50(
    input_shape=(224, 224, 3),
    include_top=False,
    weights="imagenet"
)

base_model.trainable = False   # Freeze encoder

# ------------------------------------------------------------
# 3. MODEL CREATION FUNCTION
# ------------------------------------------------------------

def build_model(num_classes):
```

```
    model = models.Sequential([
        base_model,
        layers.GlobalAveragePooling2D(),
        layers.Dense(256, activation='relu'),
        layers.Dense(num_classes, activation='softmax')
    ])

    model.compile(
        optimizer=optimizers.Adam(0.001),
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )
    return model

# ------------------------------------------------------------
# 4. TRAIN ON DOGS vs CATS
# ------------------------------------------------------------

dogs_cats_classes = len(dogs_cats_train.class_names)
model_dogs_cats = build_model(dogs_cats_classes)

history_dogs_cats = model_dogs_cats.fit(
    dogs_cats_train,
    validation_data=dogs_cats_val,
    epochs=5
)

# ------------------------------------------------------------
# 5. TRAIN ON CARS DATASET
# ------------------------------------------------------------
```

```
cars_classes = len(cars_train.class_names)
model_cars = build_model(cars_classes)

history_cars = model_cars.fit(
    cars_train,
    validation_data=cars_val,
    epochs=5
)

# ------------------------------------------------------------
# 6. PERFORMANCE COMPARISON & VISUALIZATION
# ------------------------------------------------------------

plt.figure(figsize=(10,4))

plt.subplot(1,2,1)
plt.plot(history_dogs_cats.history['val_accuracy'], label="Dogs vs Cats")
plt.plot(history_cars.history['val_accuracy'], label="Cars")
plt.xlabel("Epochs")
plt.ylabel("Validation Accuracy")
plt.title("Accuracy Comparison")
plt.legend()

plt.subplot(1,2,2)
plt.plot(history_dogs_cats.history['val_loss'], label="Dogs vs Cats")
plt.plot(history_cars.history['val_loss'], label="Cars")
plt.xlabel("Epochs")
plt.ylabel("Validation Loss")
plt.title("Loss Comparison")
plt.legend()
```

plt.show()

1. **Choose two datasets with different distributions (dogs &cats , cars).**
2. **Resize images to the required input size of the chosen pre-trained model.**
3. **Load Pre-trained Model (ResNet-50)**
4. **Compare the performances of all the models and visualize**

Write down your observations and conclusions

```
# ============================================================
# ResNet-50 Performance Comparison
# Datasets: Dogs & Cats vs Cars
# ============================================================

import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.keras import layers, models, optimizers

# ------------------------------------------------------------
# 1. LOAD & RESIZE DATASETS
# ------------------------------------------------------------

# Dogs & Cats dataset
dogs_cats_train = tf.keras.utils.image_dataset_from_directory(
    "dogs_cats/",
    image_size=(224, 224),       # ResNet-50 input size
    batch_size=32,
    validation_split=0.2,
    subset="training",
    seed=42
)

dogs_cats_val = tf.keras.utils.image_dataset_from_directory(
    "dogs_cats/",
```

```
    image_size=(224, 224),
    batch_size=32,
    validation_split=0.2,
    subset="validation",
    seed=42
)

# Cars dataset
cars_train = tf.keras.utils.image_dataset_from_directory(
    "cars/",
    image_size=(224, 224),
    batch_size=32,
    validation_split=0.2,
    subset="training",
    seed=42
)

cars_val = tf.keras.utils.image_dataset_from_directory(
    "cars/",
    image_size=(224, 224),
    batch_size=32,
    validation_split=0.2,
    subset="validation",
    seed=42
)

# Normalize images
rescale = layers.Rescaling(1./255)
dogs_cats_train = dogs_cats_train.map(lambda x, y: (rescale(x), y))
dogs_cats_val = dogs_cats_val.map(lambda x, y: (rescale(x), y))
cars_train = cars_train.map(lambda x, y: (rescale(x), y))
```

```
cars_val = cars_val.map(lambda x, y: (rescale(x), y))

# ------------------------------------------------------------
# 2. LOAD PRETRAINED RESNET-50
# ------------------------------------------------------------

base_model = tf.keras.applications.ResNet50(
    input_shape=(224, 224, 3),
    include_top=False,
    weights="imagenet"
)

base_model.trainable = False   # freeze encoder

# ------------------------------------------------------------
# 3. MODEL CREATION FUNCTION
# ------------------------------------------------------------

def build_model(num_classes):
    model = models.Sequential([
        base_model,
        layers.GlobalAveragePooling2D(),
        layers.Dense(256, activation='relu'),
        layers.Dense(num_classes, activation='softmax')
    ])

    model.compile(
        optimizer=optimizers.Adam(0.001),
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )
```

```
    return model

# ------------------------------------------------------------
# 4. TRAIN ON DOGS & CATS
# ------------------------------------------------------------

dogs_cats_classes = len(dogs_cats_train.class_names)
model_dogs_cats = build_model(dogs_cats_classes)

history_dogs_cats = model_dogs_cats.fit(
    dogs_cats_train,
    validation_data=dogs_cats_val,
    epochs=5
)

# ------------------------------------------------------------
# 5. TRAIN ON CARS DATASET
# ------------------------------------------------------------

cars_classes = len(cars_train.class_names)
model_cars = build_model(cars_classes)

history_cars = model_cars.fit(
    cars_train,
    validation_data=cars_val,
    epochs=5
)

# ------------------------------------------------------------
# 6. PERFORMANCE COMPARISON & VISUALIZATION
# ------------------------------------------------------------
```

```
plt.figure(figsize=(10,4))

plt.subplot(1,2,1)
plt.plot(history_dogs_cats.history['val_accuracy'], label="Dogs & Cats")
plt.plot(history_cars.history['val_accuracy'], label="Cars")
plt.xlabel("Epochs")
plt.ylabel("Validation Accuracy")
plt.title("Accuracy Comparison (ResNet-50)")
plt.legend()

plt.subplot(1,2,2)
plt.plot(history_dogs_cats.history['val_loss'], label="Dogs & Cats")
plt.plot(history_cars.history['val_loss'], label="Cars")
plt.xlabel("Epochs")
plt.ylabel("Validation Loss")
plt.title("Loss Comparison (ResNet-50)")
plt.legend()

plt.show()
```

- **Take the dataset of Breast cancer**
- **Initialize a neural network with random weights.**
- **Calculate output of Neural Network:**
- **Calculate MSE**
- **Plot error surface using loss function verses weight, bias**
- **Perform this cycle in step c for every input output pair**
- **Perform 5 epochs of step d.**
- **Update weights accordingly using stochastic gradient descend.**
- **Plot the mean squared error for each iteration in stochastic Gradient Descent.**
- **Similarly plot accuracy for iteration and note the results**

```
# ============================================================
# Neural Network using Breast Cancer Dataset (SGD)
# ============================================================

import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_breast_cancer
from sklearn.preprocessing import MinMaxScaler

# ------------------------------------------------------------
# 1. Load Dataset
# ------------------------------------------------------------

# If dataset is GIVEN as CSV
# import pandas as pd
# df = pd.read_csv("breast_cancer.csv")
# X = df.iloc[:, :-1].values
# y = df.iloc[:, -1].values

# If dataset is NOT given (standard sklearn)
data = load_breast_cancer()
X = data.data
y = data.target

# ------------------------------------------------------------
# 2. Normalize Input
# ------------------------------------------------------------
scaler = MinMaxScaler()
X = scaler.fit_transform(X)
y = y.reshape(-1, 1)
```

```
# ------------------------------------------------------------
# 3. Initialize Neural Network (Single Neuron)
# ------------------------------------------------------------
np.random.seed(42)
w = np.random.rand(X.shape[1], 1)    # random weights
b = np.random.rand(1)                # random bias
lr = 0.1
epochs = 5

mse_history = []
accuracy_history = []

# ------------------------------------------------------------
# 4. Activation Function
# ------------------------------------------------------------
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# ------------------------------------------------------------
# 5. Training using Stochastic Gradient Descent
# ------------------------------------------------------------
for epoch in range(epochs):

    squared_errors = []
    correct = 0

    for i in range(len(X)):  # SGD → one sample at a time

        # a) Forward Pass (Output)
        z = np.dot(X[i], w) + b
        y_pred = sigmoid(z)
```

```
        # b) Squared Error
        error = y_pred - y[i]
        sq_error = error ** 2
        squared_errors.append(sq_error)

        # c) Gradient Calculation
        dw = 2 * error * X[i].reshape(-1, 1)
        db = 2 * error

        # d) Weight Update (SGD)
        w = w - lr * dw
        b = b - lr * db

        # Accuracy
        predicted_class = 1 if y_pred >= 0.5 else 0
        if predicted_class == y[i]:
            correct += 1

    # e) Mean Squared Error
    mse = np.mean(squared_errors)
    mse_history.append(mse)

    accuracy = correct / len(X)
    accuracy_history.append(accuracy)

    print(f"Epoch {epoch+1} | MSE: {mse:.4f} | Accuracy: {accuracy:.4f}")

# ------------------------------------------------------------
# 6. Plot MSE vs Iterations
# ------------------------------------------------------------
```

```
plt.figure()
plt.plot(mse_history)
plt.xlabel("Epochs")
plt.ylabel("Mean Squared Error")
plt.title("MSE vs Epochs (SGD)")
plt.show()

# ------------------------------------------------------------
# 7. Plot Accuracy vs Iterations
# ------------------------------------------------------------
plt.figure()
plt.plot(accuracy_history)
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.title("Accuracy vs Epochs (SGD)")
plt.show()

# ------------------------------------------------------------
# 8. Error Surface (Loss vs Weight & Bias)
# ------------------------------------------------------------
w_vals = np.linspace(0, 1, 30)
b_vals = np.linspace(0, 1, 30)
W, B = np.meshgrid(w_vals, b_vals)
Z = np.zeros(W.shape)

# Using only first feature for visualization
X1 = X[:, 0].reshape(-1, 1)

for i in range(W.shape[0]):
    for j in range(W.shape[1]):
        y_temp = sigmoid(X1 * W[i,j] + B[i,j])
```

```
        Z[i,j] = np.mean((y_temp - y) ** 2)


from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(W, B, Z)
ax.set_xlabel("Weight")
ax.set_ylabel("Bias")
ax.set_zlabel("Loss (MSE)")
ax.set_title("Error Surface")
plt.show()
```

- **Take the dataset of Iris.**
- **Initialize a neural network with random weights.**
- **Calculate output of Neural Network:**
- **Calculate MSE**
- **Plot error surface using loss function verses weight, bias**
- **Perform this cycle in step c for every input output pair**
- **Perform 10 epochs of step d.**
- **Update weights accordingly using stochastic gradient descend.**
- **Plot the mean squared error for each iteration in stochastic Gradient Descent.**
- **Similarly plot accuracy for iteration and note the results**

```
    # ============================================================
    # Neural Network using IRIS Dataset (SGD – Full Implementation)
    # ============================================================

    import numpy as np
    import matplotlib.pyplot as plt
    from sklearn.datasets import load_iris
    from sklearn.preprocessing import MinMaxScaler
```

```
# ------------------------------------------------------------
# 1. Load IRIS Dataset
# ------------------------------------------------------------

# If dataset is GIVEN as CSV:
# import pandas as pd
# df = pd.read_csv("iris.csv")
# X = df.iloc[:, :-1].values
# y = df.iloc[:, -1].values

# If dataset is NOT given (standard sklearn)
iris = load_iris()
X = iris.data
y = iris.target

# Convert to binary classification (class 0 vs others)
y = (y == 0).astype(int)

# ------------------------------------------------------------
# 2. Normalize Input
# ------------------------------------------------------------
scaler = MinMaxScaler()
X = scaler.fit_transform(X)
y = y.reshape(-1, 1)

# ------------------------------------------------------------
# 3. Initialize Neural Network (Single Neuron)
# ------------------------------------------------------------
np.random.seed(42)
w = np.random.rand(X.shape[1], 1)   # random weights
b = np.random.rand(1)               # random bias
```

```
lr = 0.1
epochs = 10

mse_history = []
accuracy_history = []

# ---------------------------------------------------------
# 4. Activation Function
# ---------------------------------------------------------
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# ---------------------------------------------------------
# 5. Training using Stochastic Gradient Descent
# ---------------------------------------------------------
for epoch in range(epochs):

    squared_errors = []
    correct = 0

    for i in range(len(X)):   # SGD → one sample at a time

        # a) Forward pass (Output of Neural Network)
        z = np.dot(X[i], w) + b
        y_pred = sigmoid(z)

        # b) Squared Error
        error = y_pred - y[i]
        sq_error = error ** 2
        squared_errors.append(sq_error)
```

```
        # c) Gradient calculation
        dw = 2 * error * X[i].reshape(-1, 1)
        db = 2 * error

        # d) Update weights and bias (SGD)
        w = w - lr * dw
        b = b - lr * db

        # Accuracy calculation
        predicted_class = 1 if y_pred >= 0.5 else 0
        if predicted_class == y[i]:
            correct += 1

    # Mean Squared Error for epoch
    mse = np.mean(squared_errors)
    mse_history.append(mse)

    accuracy = correct / len(X)
    accuracy_history.append(accuracy)

    print(f"Epoch {epoch+1} | MSE: {mse:.4f} | Accuracy: {accuracy:.4f}")

# ------------------------------------------------------------
# 6. Plot MSE vs Iterations (Epochs)
# ------------------------------------------------------------
plt.figure()
plt.plot(mse_history)
plt.xlabel("Epochs")
plt.ylabel("Mean Squared Error")
plt.title("MSE vs Epochs (SGD)")
plt.show()
```

```
# ------------------------------------------------------------
# 7. Plot Accuracy vs Iterations
# ------------------------------------------------------------
plt.figure()
plt.plot(accuracy_history)
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.title("Accuracy vs Epochs (SGD)")
plt.show()

# ------------------------------------------------------------
# 8. Error Surface (Loss vs Weight & Bias)
# ------------------------------------------------------------

# For visualization, use only the first feature
X1 = X[:, 0].reshape(-1, 1)

w_vals = np.linspace(0, 1, 30)
b_vals = np.linspace(0, 1, 30)
W, B = np.meshgrid(w_vals, b_vals)
Z = np.zeros(W.shape)

for i in range(W.shape[0]):
    for j in range(W.shape[1]):
        y_temp = sigmoid(X1 * W[i, j] + B[i, j])
        Z[i, j] = np.mean((y_temp - y) ** 2)

from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
```

```
ax.plot_surface(W, B, Z)
ax.set_xlabel("Weight")
ax.set_ylabel("Bias")
ax.set_zlabel("Loss (MSE)")
ax.set_title("Error Surface (Iris Dataset)")
plt.show()
```

- **Implement batch gradient descent optimizer function**

Take the dataset of Titanic

- **Initialize a neural network with random weights.**
- **Calculate output of Neural Network:**
- **Calculate squared error loss**
- **Update network parameter using batch gradient descent optimizer function Implementation.**
- **Display updated weight and bias values**
- **Plot loss w.r.t. Iterations**

```
# ============================================================
# Batch Gradient Descent Optimizer - Titanic Dataset
# ============================================================

import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler

# ------------------------------------------------------------
# 1. Load Titanic Dataset
# ------------------------------------------------------------

# CASE 1: If dataset is GIVEN as CSV (LAB / EXAM)
import pandas as pd
df = pd.read_csv("titanic.csv")
```

```
# Select numeric features only (simple for exams)
# Example features: Pclass, Age, Fare
df = df[['Pclass', 'Age', 'Fare', 'Survived']]
df = df.dropna()   # remove missing values

X = df[['Pclass', 'Age', 'Fare']].values
y = df['Survived'].values.reshape(-1, 1)

# ------------------------------------------------------------
# 2. Normalize Input
# ------------------------------------------------------------
scaler = MinMaxScaler()
X = scaler.fit_transform(X)

# ------------------------------------------------------------
# 3. Initialize Neural Network (Single Neuron)
# ------------------------------------------------------------
np.random.seed(42)
w = np.random.rand(X.shape[1], 1)   # random weights
b = np.random.rand(1)               # random bias
lr = 0.1
epochs = 50

loss_history = []

# ------------------------------------------------------------
# 4. Activation Function
# ------------------------------------------------------------
def sigmoid(z):
    return 1 / (1 + np.exp(-z))
```

```
# ------------------------------------------------------------
# 5. Batch Gradient Descent Training
# ------------------------------------------------------------
for epoch in range(epochs):

    # a) Forward pass (output of neural network)
    z = np.dot(X, w) + b
    y_pred = sigmoid(z)

    # b) Squared Error Loss
    error = y_pred - y
    loss = np.mean(error ** 2)
    loss_history.append(loss)

    # c) Gradients (BATCH – use full dataset)
    dw = (2 / len(X)) * np.dot(X.T, error)
    db = (2 / len(X)) * np.sum(error)

    # d) Update weights and bias
    w = w - lr * dw
    b = b - lr * db

    # Display updated parameters
    print(f"Epoch {epoch+1}")
    print("Loss:", loss)
    print("Updated Weights:", w.flatten())
    print("Updated Bias:", b)
    print("-" * 40)

# ------------------------------------------------------------
# 6. Plot Loss vs Iterations
```

```
# ------------------------------------------------------------
plt.plot(loss_history)
plt.xlabel("Iterations (Epochs)")
plt.ylabel("Squared Error Loss")
plt.title("Loss vs Iterations (Batch Gradient Descent)")
plt.show()
```

1. **Implement the NOR Boolean logic gate using perceptron Neural Network. Inputs = x1, x2 and bias, weights should be fed into the perceptron with single Output = y. Display final weights and bias of each perceptron.**
2. **Take the dataset of Diabetes 2**

b) Initialize a neural network with random weights.

c)Calculate output of Neural Network:

i. Calculate squared error loss

ii. Update network parameter using batch Mini Batch gradient descent optimizer function Implementation.

Iii. Display updated weight and bias values

iv. Plot loss w.r.t. bias values

```
# ============================================================
# TASK 1: NOR Gate using Perceptron Neural Network
# ============================================================

import numpy as np

# Input data (x1, x2)
X = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
])

# NOR gate output
```

```
y = np.array([1, 0, 0, 0])

# Initialize weights and bias
weights = np.zeros(2)
bias = 0
learning_rate = 0.1
epochs = 10

# Step activation function
def step(z):
    return 1 if z >= 0 else 0

# Training perceptron
for _ in range(epochs):
    for i in range(len(X)):
        z = np.dot(X[i], weights) + bias
        y_pred = step(z)
        error = y[i] - y_pred
        weights += learning_rate * error * X[i]
        bias += learning_rate * error

# Display final weights and bias
print("TASK 1 OUTPUT")
print("Final Weights:", weights)
print("Final Bias:", bias)

print("\nNOR Gate Results")
for x in X:
    print(x, "->", step(np.dot(x, weights) + bias))
```

```
# ============================================================
# TASK 2: Mini-Batch Gradient Descent - Diabetes Dataset
# ============================================================

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler

# ------------------------------------------------------------
# 1. Load Dataset
# ------------------------------------------------------------

# Dataset GIVEN as CSV
df = pd.read_csv("diabetes.csv")

X = df.iloc[:, :-1].values   # input features
y = df.iloc[:, -1].values.reshape(-1, 1)   # target

# ------------------------------------------------------------
# 2. Normalize Input
# ------------------------------------------------------------
scaler = MinMaxScaler()
X = scaler.fit_transform(X)

# ------------------------------------------------------------
# 3. Initialize Neural Network (Single Neuron)
# ------------------------------------------------------------
np.random.seed(42)
w = np.random.rand(X.shape[1], 1)
b = np.random.rand(1)
```

```
lr = 0.1
epochs = 20
batch_size = 32

bias_history = []
loss_history = []

# ---------------------------------------------------------------
# 4. Activation Function
# ---------------------------------------------------------------
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# ---------------------------------------------------------------
# 5. Mini-Batch Gradient Descent
# ---------------------------------------------------------------
for epoch in range(epochs):

    for i in range(0, len(X), batch_size):
        X_batch = X[i:i+batch_size]
        y_batch = y[i:i+batch_size]

        # a) Forward pass
        z = np.dot(X_batch, w) + b
        y_pred = sigmoid(z)

        # b) Squared Error Loss
        error = y_pred - y_batch
        loss = np.mean(error ** 2)

        # c) Gradients (Mini-Batch)
```

```
        dw = (2 / len(X_batch)) * np.dot(X_batch.T, error)
        db = (2 / len(X_batch)) * np.sum(error)

        # d) Update weights and bias
        w = w - lr * dw
        b = b - lr * db

    loss_history.append(loss)
    bias_history.append(b[0])

    print(f"Epoch {epoch+1}")
    print("Loss:", loss)
    print("Updated Weights:", w.flatten())
    print("Updated Bias:", b)
    print("-" * 40)

# ------------------------------------------------------------
# 6. Plot Loss vs Bias
# ------------------------------------------------------------
plt.plot(bias_history, loss_history)
plt.xlabel("Bias")
plt.ylabel("Squared Error Loss")
plt.title("Loss vs Bias (Mini-Batch Gradient Descent)")
plt.show()
```

- **Take the dataset of Diabetes**
- **Initialize a neural network with random weights.**
    - **c)Calculate output of Neural Network:**
    - **Calculate Mean squared error loss**
    - **Update network parameter using batch momentum based gradient descent optimizer function Implementation.**
    - **Display updated weight and bias values**

- **Plot loss w.r.t. iterations**

```
# ============================================================
# Batch Gradient Descent with Momentum - Diabetes Dataset
# ============================================================

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler

# ------------------------------------------------------------
# 1. Load Dataset
# ------------------------------------------------------------

# If dataset is GIVEN as CSV (LAB / EXAM CASE)
df = pd.read_csv("diabetes.csv")

X = df.iloc[:, :-1].values          # input features
y = df.iloc[:, -1].values.reshape(-1,1) # target

# ------------------------------------------------------------
# 2. Normalize Input
# ------------------------------------------------------------
scaler = MinMaxScaler()
X = scaler.fit_transform(X)

# ------------------------------------------------------------
# 3. Initialize Neural Network (Single Neuron)
# ------------------------------------------------------------
np.random.seed(42)
```

```
w = np.random.rand(X.shape[1], 1)   # random weights
b = np.random.rand(1)               # random bias

lr = 0.1
epochs = 50
momentum = 0.9

# Velocity terms (momentum)
v_w = np.zeros_like(w)
v_b = 0

loss_history = []

# ------------------------------------------------------------
# 4. Activation Function
# ------------------------------------------------------------
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# ------------------------------------------------------------
# 5. Batch Gradient Descent with Momentum
# ------------------------------------------------------------
for epoch in range(epochs):

    # a) Forward pass (Output of Neural Network)
    z = np.dot(X, w) + b
    y_pred = sigmoid(z)

    # b) Mean Squared Error Loss
    error = y_pred - y
    loss = np.mean(error ** 2)
```

```
    loss_history.append(loss)

    # c) Gradients (BATCH)
    dw = (2 / len(X)) * np.dot(X.T, error)
    db = (2 / len(X)) * np.sum(error)

    # d) Momentum update
    v_w = momentum * v_w + lr * dw
    v_b = momentum * v_b + lr * db

    # e) Update weights and bias
    w = w - v_w
    b = b - v_b

    # Display updated parameters
    print(f"Epoch {epoch+1}")
    print("Loss:", loss)
    print("Updated Weights:", w.flatten())
    print("Updated Bias:", b)
    print("-" * 40)

# ------------------------------------------------------------
# 6. Plot Loss vs Iterations
# ------------------------------------------------------------
plt.plot(loss_history)
plt.xlabel("Iterations (Epochs)")
plt.ylabel("Mean Squared Error")
plt.title("Loss vs Iterations (Batch GD with Momentum)")
plt.show()
```

1. **Implement the XOR Boolean logic gate using perceptron Neural Network. Inputs = x1, x2 and bias, weights should be fed into the perceptron with single Output = y. Display final weights and bias of each perceptron.**

2. **Take the dataset of Penguin**

3. **b) Initialize a neural network with random weights.**

c)Calculate output of Neural Network:

i. Calculate squared error loss

ii. Update network parameter using batch Adaptive delta gradient descent optimizer function Implementation.

iii. Display updated weight and bias values

iv. Plot accuracy w.r.t. epoch values

```
# ============================================================

# TASK 1: XOR Gate using Multi-Layer Perceptron

# ============================================================


import numpy as np


# XOR input and output

X = np.array([

    [0, 0],

    [0, 1],

    [1, 0],

    [1, 1]

])


y = np.array([[0], [1], [1], [0]])


# Initialize weights and biases (random)

np.random.seed(42)

W1 = np.random.randn(2, 2)   # hidden layer weights
```

```
b1 = np.random.randn(1, 2)

W2 = np.random.randn(2, 1)   # output layer weights
b2 = np.random.randn(1, 1)

lr = 0.5
epochs = 5000

# Activation functions
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def sigmoid_derivative(z):
    return z * (1 - z)

# Training
for _ in range(epochs):
    # Forward pass
    h = sigmoid(np.dot(X, W1) + b1)
    y_pred = sigmoid(np.dot(h, W2) + b2)

    # Backpropagation
    error = y - y_pred
    d_output = error * sigmoid_derivative(y_pred)

    d_hidden = d_output.dot(W2.T) * sigmoid_derivative(h)

    # Update weights and biases
    W2 += lr * h.T.dot(d_output)
    b2 += lr * np.sum(d_output, axis=0, keepdims=True)
```

```
    W1 += lr * X.T.dot(d_hidden)
    b1 += lr * np.sum(d_hidden, axis=0, keepdims=True)

# Final output
print("TASK 1 OUTPUT")
print("Hidden Layer Weights:\n", W1)
print("Hidden Layer Bias:\n", b1)
print("Output Layer Weights:\n", W2)
print("Output Layer Bias:\n", b2)

print("\nXOR Gate Result:")
print(np.round(y_pred))

# ============================================================
# TASK 2: Adadelta Optimizer – Penguin Dataset
# ============================================================

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler

# ------------------------------------------------------------
# 1. Load Penguin Dataset
# ------------------------------------------------------------

# Dataset GIVEN as CSV (lab case)
df = pd.read_csv("penguins.csv")

# Select numeric features and target
df = df.dropna()
```

```
X = df[['bill_length_mm', 'bill_depth_mm', 'flipper_length_mm', 'body_mass_g']].values
y = (df['species'] == 'Adelie').astype(int).values.reshape(-1, 1)

# ------------------------------------------------------------
# 2. Normalize Inputs
# ------------------------------------------------------------
scaler = MinMaxScaler()
X = scaler.fit_transform(X)

# ------------------------------------------------------------
# 3. Initialize Neural Network (Single Neuron)
# ------------------------------------------------------------
np.random.seed(42)
w = np.random.randn(X.shape[1], 1)
b = np.random.randn(1)

epochs = 50
rho = 0.95
eps = 1e-6

# Adadelta accumulators
Eg_w = np.zeros_like(w)
Eg_b = 0
Ed_w = np.zeros_like(w)
Ed_b = 0

accuracy_history = []

# ------------------------------------------------------------
# 4. Activation Function
# ------------------------------------------------------------
```

```
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# ------------------------------------------------------------
# 5. Batch Adadelta Gradient Descent
# ------------------------------------------------------------
for epoch in range(epochs):

    # Forward pass
    z = np.dot(X, w) + b
    y_pred = sigmoid(z)

    # Squared error loss
    error = y_pred - y
    loss = np.mean(error ** 2)

    # Gradients (batch)
    dw = (2 / len(X)) * np.dot(X.T, error)
    db = (2 / len(X)) * np.sum(error)

    # Accumulate gradients
    Eg_w = rho * Eg_w + (1 - rho) * (dw ** 2)
    Eg_b = rho * Eg_b + (1 - rho) * (db ** 2)

    # Compute updates
    delta_w = - np.sqrt(Ed_w + eps) / np.sqrt(Eg_w + eps) * dw
    delta_b = - np.sqrt(Ed_b + eps) / np.sqrt(Eg_b + eps) * db

    # Accumulate updates
    Ed_w = rho * Ed_w + (1 - rho) * (delta_w ** 2)
    Ed_b = rho * Ed_b + (1 - rho) * (delta_b ** 2)
```

```
    # Update parameters
    w += delta_w
    b += delta_b

    # Accuracy
    predictions = (y_pred >= 0.5).astype(int)
    accuracy = np.mean(predictions == y)
    accuracy_history.append(accuracy)

    print(f"Epoch {epoch+1}")
    print("Loss:", loss)
    print("Updated Weights:", w.flatten())
    print("Updated Bias:", b)
    print("-" * 40)

# ------------------------------------------------------------
# 6. Plot Accuracy vs Epochs
# ------------------------------------------------------------
plt.plot(accuracy_history)
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.title("Accuracy vs Epochs (Adadelta Optimizer)")
plt.show()
```

1. **Implement backpropagation algorithm from scratch.**

a) **Take Iris Dataset**

b) **Initialize a neural network with random weights.**

c) **Calculate Squared Error (SE)**

d) **Perform multiple iterations.**

e) **Update weights accordingly.**

f) **Plot accuracy for iterations and note the results.**

```
# ============================================================
# Backpropagation Algorithm from Scratch - IRIS Dataset
# ============================================================

import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.preprocessing import MinMaxScaler

# ------------------------------------------------------------
# a) Load IRIS Dataset
# ------------------------------------------------------------

# If dataset is GIVEN as CSV:
# import pandas as pd
# df = pd.read_csv("iris.csv")
# X = df.iloc[:, :-1].values
# y = df.iloc[:, -1].values

# Standard IRIS dataset
iris = load_iris()
X = iris.data
y = iris.target

# Convert to binary classification (Setosa vs Others)
y = (y == 0).astype(int).reshape(-1, 1)

# ------------------------------------------------------------
# Normalize Input
# ------------------------------------------------------------
scaler = MinMaxScaler()
```

```
X = scaler.fit_transform(X)

# ------------------------------------------------------------
# b) Initialize Neural Network with Random Weights
# ------------------------------------------------------------
np.random.seed(42)

input_neurons = X.shape[1]
hidden_neurons = 5
output_neurons = 1

W1 = np.random.rand(input_neurons, hidden_neurons)
b1 = np.random.rand(1, hidden_neurons)

W2 = np.random.rand(hidden_neurons, output_neurons)
b2 = np.random.rand(1, output_neurons)

lr = 0.5
epochs = 100

accuracy_history = []

# ------------------------------------------------------------
# Activation Functions
# ------------------------------------------------------------
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def sigmoid_derivative(a):
    return a * (1 - a)
```

```
# ------------------------------------------------------------
# c–e) Backpropagation Training
# ------------------------------------------------------------
for epoch in range(epochs):

    # ---------- Forward Pass ----------
    z1 = np.dot(X, W1) + b1
    a1 = sigmoid(z1)

    z2 = np.dot(a1, W2) + b2
    y_pred = sigmoid(z2)

    # ---------- c) Squared Error ----------
    error = y_pred - y
    SE = error ** 2

    # ---------- Backward Pass ----------
    d_output = error * sigmoid_derivative(y_pred)
    d_hidden = d_output.dot(W2.T) * sigmoid_derivative(a1)

    # ---------- e) Update Weights ----------
    W2 -= lr * a1.T.dot(d_output)
    b2 -= lr * np.sum(d_output, axis=0, keepdims=True)

    W1 -= lr * X.T.dot(d_hidden)
    b1 -= lr * np.sum(d_hidden, axis=0, keepdims=True)

    # ---------- Accuracy ----------
    predictions = (y_pred >= 0.5).astype(int)
    accuracy = np.mean(predictions == y)
    accuracy_history.append(accuracy)
```

```
    if (epoch + 1) % 10 == 0:
        print(f"Epoch {epoch+1} | Accuracy: {accuracy:.4f}")


# ------------------------------------------------------------
# f) Plot Accuracy vs Iterations
# ------------------------------------------------------------
plt.plot(accuracy_history)
plt.xlabel("Iterations (Epochs)")
plt.ylabel("Accuracy")
plt.title("Accuracy vs Iterations (Backpropagation)")
plt.show()
```

Build a multiclass image categorization CNN network which correctly classifies different categories of images in the dataset.(handwritten digits from Mnist digit dataset

- **Split original dataset to train and test set**
- **Build CNN Model**
- **Generate the accuracy of the built model using Adam Optimizer and Adagrad Optimizer.**
- **Compare performance of different optimizer on Digit categorization.**
- **Plot training vs validation accuracy**
- **Evaluate the model using confusion matrix, precision, recall.**

```
# ============================================================
# Multiclass CNN for MNIST Digit Classification
# Optimizer Comparison: Adam vs Adagrad
# ============================================================

import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist
```

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.optimizers import Adam, Adagrad
from sklearn.metrics import confusion_matrix, precision_score, recall_score

# ------------------------------------------------------------
# 1. Load and Split Dataset
# ------------------------------------------------------------

# If dataset is GIVEN as CSV (commented for exam)
# import pandas as pd
# train_df = pd.read_csv("mnist_train.csv")
# test_df = pd.read_csv("mnist_test.csv")
# X_train = train_df.iloc[:,1:].values.reshape(-1,28,28,1)/255.0
# y_train = train_df.iloc[:,0].values
# X_test = test_df.iloc[:,1:].values.reshape(-1,28,28,1)/255.0
# y_test = test_df.iloc[:,0].values

# Standard MNIST loading
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# Normalize & reshape
X_train = X_train.reshape(-1,28,28,1) / 255.0
X_test = X_test.reshape(-1,28,28,1) / 255.0

# One-hot encoding
y_train_cat = to_categorical(y_train, 10)
y_test_cat = to_categorical(y_test, 10)

# ------------------------------------------------------------
```

```
# 2. CNN Model Definition
# ------------------------------------------------------------

def create_cnn(optimizer):
    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)),
        Flatten(),
        Dense(128, activation='relu'),
        Dense(10, activation='softmax')
    ])

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

# ------------------------------------------------------------
# 3. Train with ADAM Optimizer
# ------------------------------------------------------------

model_adam = create_cnn(Adam())
history_adam = model_adam.fit(
    X_train, y_train_cat,
    epochs=5,
    validation_data=(X_test, y_test_cat),
    verbose=0
```

```
)

# ------------------------------------------------------------
# 4. Train with ADAGRAD Optimizer
# ------------------------------------------------------------

model_adagrad = create_cnn(Adagrad())
history_adagrad = model_adagrad.fit(
    X_train, y_train_cat,
    epochs=5,
    validation_data=(X_test, y_test_cat),
    verbose=0
)

# ------------------------------------------------------------
# 5. Plot Training vs Validation Accuracy
# ------------------------------------------------------------

plt.figure(figsize=(10,4))

plt.subplot(1,2,1)
plt.plot(history_adam.history['accuracy'], label='Train (Adam)')
plt.plot(history_adam.history['val_accuracy'], label='Val (Adam)')
plt.title("Adam Optimizer")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend()

plt.subplot(1,2,2)
plt.plot(history_adagrad.history['accuracy'], label='Train (Adagrad)')
plt.plot(history_adagrad.history['val_accuracy'], label='Val (Adagrad)')
```

```
plt.title("Adagrad Optimizer")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend()

plt.show()

# ------------------------------------------------------------
# 6. Evaluation: Confusion Matrix, Precision, Recall (Adam)
# ------------------------------------------------------------

y_pred = model_adam.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)

cm = confusion_matrix(y_test, y_pred_classes)
precision = precision_score(y_test, y_pred_classes, average='macro')
recall = recall_score(y_test, y_pred_classes, average='macro')

print("Confusion Matrix:\n", cm)
print("Precision:", precision)
print("Recall:", recall)
```

Build a multiclass image categorization CNN network which correctly classifies different categories of images in the dataset.(Fashion Mnist dataset.)

- **Split original dataset to train and test set**
- **Build CNN Model**
- **Generate the accuracy of the built model using RMSProp and SGDOptimizer.**
- **Perform hyperparameter tuning to increase the accuracy of the CNN.**
- **Compare performance of different optimizer on Cloth categorization.**
- **Plot training vs validation loss**
- **Evaluate the model using confusion matrix, precision.**

```
# ============================================================
```

```
# Multiclass CNN - Fashion MNIST
# Optimizer Comparison: RMSProp vs SGD
# ============================================================

import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import fashion_mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.optimizers import RMSprop, SGD
from sklearn.metrics import confusion_matrix, precision_score

# ------------------------------------------------------------
# 1. Load & Split Dataset
# ------------------------------------------------------------

# If dataset is GIVEN as CSV (commented for exam)
# import pandas as pd
# train_df = pd.read_csv("fashion_mnist_train.csv")
# test_df = pd.read_csv("fashion_mnist_test.csv")
# X_train = train_df.iloc[:,1:].values.reshape(-1,28,28,1)/255.0
# y_train = train_df.iloc[:,0].values
# X_test = test_df.iloc[:,1:].values.reshape(-1,28,28,1)/255.0
# y_test = test_df.iloc[:,0].values

# Standard Fashion-MNIST loading
(X_train, y_train), (X_test, y_test) = fashion_mnist.load_data()

# Normalize and reshape
X_train = X_train.reshape(-1,28,28,1) / 255.0
```

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

# One-hot encode labels
y_train_cat = to_categorical(y_train, 10)
y_test_cat = to_categorical(y_test, 10)

# ------------------------------------------------------------
# 2. CNN Model with Hyperparameter Tuning
# ------------------------------------------------------------

def create_cnn(optimizer):
    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)),

        Flatten(),
        Dense(128, activation='relu'),
        Dropout(0.3),              # Hyperparameter tuning
        Dense(10, activation='softmax')
    ])

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

```
# ------------------------------------------------------------
# 3. Train using RMSProp Optimizer
# ------------------------------------------------------------

model_rms = create_cnn(RMSprop(learning_rate=0.001))
history_rms = model_rms.fit(
    X_train, y_train_cat,
    epochs=5,
    validation_data=(X_test, y_test_cat),
    verbose=0
)

# ------------------------------------------------------------
# 4. Train using SGD Optimizer
# ------------------------------------------------------------

model_sgd = create_cnn(SGD(learning_rate=0.01, momentum=0.9))
history_sgd = model_sgd.fit(
    X_train, y_train_cat,
    epochs=5,
    validation_data=(X_test, y_test_cat),
    verbose=0
)

# ------------------------------------------------------------
# 5. Plot Training vs Validation Loss
# ------------------------------------------------------------

plt.figure(figsize=(10,4))

plt.subplot(1,2,1)
```

```
plt.plot(history_rms.history['loss'], label='Train Loss (RMSProp)')
plt.plot(history_rms.history['val_loss'], label='Val Loss (RMSProp)')
plt.title("RMSProp Loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()

plt.subplot(1,2,2)
plt.plot(history_sgd.history['loss'], label='Train Loss (SGD)')
plt.plot(history_sgd.history['val_loss'], label='Val Loss (SGD)')
plt.title("SGD Loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()

plt.show()

# ------------------------------------------------------------
# 6. Evaluation using Confusion Matrix & Precision (RMSProp)
# ------------------------------------------------------------

y_pred = model_rms.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)

cm = confusion_matrix(y_test, y_pred_classes)
precision = precision_score(y_test, y_pred_classes, average='macro')

print("Confusion Matrix:\n", cm)
print("Precision:", precision)
```

1. Implement CNN and compare its performance using different optimizers

Take the MNIST dataset

b) Initialize a neural network basic layers with random weights.

c) Perform practical analysis of optimizers on MNIST dataset keeping batch size, and epochs same but with different optimizers.

d) Compare the results by choosing 5 different optimizers [SGD, Adadelta, Adagrad, Adam, RMSprop] on a simple neural network

```
# ============================================================

# CNN on MNIST - Optimizer Comparison

# Optimizers: SGD, Adadelta, Adagrad, Adam, RMSprop

# ============================================================


import numpy as np

import matplotlib.pyplot as plt

from tensorflow.keras.datasets import mnist

from tensorflow.keras.models import Sequential

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

from tensorflow.keras.utils import to_categorical

from tensorflow.keras.optimizers import SGD, Adadelta, Adagrad, Adam, RMSprop


# ------------------------------------------------------------

# a) Load and Split MNIST Dataset

# ------------------------------------------------------------


# If dataset is GIVEN as CSV (exam case)

# import pandas as pd

# train_df = pd.read_csv("mnist_train.csv")

# test_df = pd.read_csv("mnist_test.csv")

# X_train = train_df.iloc[:,1:].values.reshape(-1,28,28,1)/255.0

# y_train = train_df.iloc[:,0].values

# X_test = test_df.iloc[:,1:].values.reshape(-1,28,28,1)/255.0

# y_test = test_df.iloc[:,0].values
```

```
# Standard MNIST loading
(X_train, y_train), (X_test, y_test) = mnist.load_data()

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

y_train_cat = to_categorical(y_train, 10)
y_test_cat = to_categorical(y_test, 10)

# ------------------------------------------------------------
# b) CNN Model (Random Weight Initialization by default)
# ------------------------------------------------------------

def create_cnn(optimizer):
    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)),
        Flatten(),
        Dense(128, activation='relu'),
        Dense(10, activation='softmax')
    ])

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

```
# ------------------------------------------------------------
# c) Fixed Hyperparameters for Fair Comparison
# ------------------------------------------------------------

batch_size = 128
epochs = 5

optimizers = {
    "SGD": SGD(learning_rate=0.01),
    "Adadelta": Adadelta(),
    "Adagrad": Adagrad(),
    "Adam": Adam(),
    "RMSprop": RMSprop()
}

histories = {}

# ------------------------------------------------------------
# d) Train CNN with Different Optimizers
# ------------------------------------------------------------

for name, opt in optimizers.items():
    print(f"\nTraining with {name} optimizer")
    model = create_cnn(opt)
    history = model.fit(
        X_train, y_train_cat,
        batch_size=batch_size,
        epochs=epochs,
        validation_data=(X_test, y_test_cat),
        verbose=0
```

```
    )
    histories[name] = history

# ------------------------------------------------------------
# e) Compare Validation Accuracy
# ------------------------------------------------------------

plt.figure(figsize=(10,5))
for name, history in histories.items():
    plt.plot(history.history['val_accuracy'], label=name)

plt.xlabel("Epochs")
plt.ylabel("Validation Accuracy")
plt.title("Optimizer Comparison on MNIST (CNN)")
plt.legend()
plt.show()
```

- **Load the CIFAR-10 image dataset.**
- **Split the dataset into training and testing sets.**
- **Design a CNN model for object classification.**
- **Train and evaluate the model.**
- **Plot accuracy and loss curves.**

```
# ============================================================
# CNN for CIFAR-10 Image Classification
# ============================================================

import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from tensorflow.keras.utils import to_categorical
```

```
from tensorflow.keras.optimizers import Adam

# ------------------------------------------------------------
# 1. Load CIFAR-10 Dataset
# ------------------------------------------------------------

# If dataset is GIVEN as image files (commented for exam)
# train_ds = tf.keras.utils.image_dataset_from_directory(
#     "cifar10_train/",
#     image_size=(32,32),
#     batch_size=64
# )
# test_ds = tf.keras.utils.image_dataset_from_directory(
#     "cifar10_test/",
#     image_size=(32,32),
#     batch_size=64
# )

# Standard CIFAR-10 loading
(X_train, y_train), (X_test, y_test) = cifar10.load_data()

# ------------------------------------------------------------
# 2. Preprocess Data
# ------------------------------------------------------------

# Normalize images
X_train = X_train / 255.0
X_test = X_test / 255.0

# One-hot encode labels
y_train_cat = to_categorical(y_train, 10)
```

```
y_test_cat = to_categorical(y_test, 10)

# ------------------------------------------------------------
# 3. Design CNN Model
# ------------------------------------------------------------

model = Sequential([
    Conv2D(32, (3,3), activation='relu', input_shape=(32,32,3)),
    MaxPooling2D((2,2)),

    Conv2D(64, (3,3), activation='relu'),
    MaxPooling2D((2,2)),

    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.3),
    Dense(10, activation='softmax')
])

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

# ------------------------------------------------------------
# 4. Train the Model
# ------------------------------------------------------------

history = model.fit(
    X_train, y_train_cat,
```

```
    epochs=10,
    batch_size=64,
    validation_data=(X_test, y_test_cat)
)

# ------------------------------------------------------------
# 5. Evaluate the Model
# ------------------------------------------------------------

test_loss, test_accuracy = model.evaluate(X_test, y_test_cat)
print("Test Accuracy:", test_accuracy)
print("Test Loss:", test_loss)

# ------------------------------------------------------------
# 6. Plot Accuracy and Loss Curves
# ------------------------------------------------------------

plt.figure(figsize=(10,4))

# Accuracy plot
plt.subplot(1,2,1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.title("Accuracy Curve")
plt.legend()

# Loss plot
plt.subplot(1,2,2)
plt.plot(history.history['loss'], label='Training Loss')
```

```
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Loss Curve")
plt.legend()

plt.show()
```

- **Use alpaca dataset**
- **CNN must include : Convolution layer, Pooling layer, Flatten layer,Dense layer**

Plot:

- **Accuracy vs Epochs**
- **Loss (Error) vs Epochs**

```
# ============================================================
# CNN for Alpaca Dataset Classification
# ============================================================

import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
from tensorflow.keras.optimizers import Adam

# ------------------------------------------------------------
# 1. Load Alpaca Dataset
# ------------------------------------------------------------

train_ds = tf.keras.utils.image_dataset_from_directory(
    "alpaca_dataset/",
    image_size=(64, 64),
    batch_size=32,
```

```
    validation_split=0.2,
    subset="training",
    seed=42
)

val_ds = tf.keras.utils.image_dataset_from_directory(
    "alpaca_dataset/",
    image_size=(64, 64),
    batch_size=32,
    validation_split=0.2,
    subset="validation",
    seed=42
)

# Normalize images
normalizer = tf.keras.layers.Rescaling(1./255)
train_ds = train_ds.map(lambda x, y: (normalizer(x), y))
val_ds = val_ds.map(lambda x, y: (normalizer(x), y))

num_classes = len(train_ds.class_names)

# ------------------------------------------------------------
# 2. Build CNN Model
# (Conv → Pool → Flatten → Dense)
# ------------------------------------------------------------

model = Sequential([
    Conv2D(32, (3,3), activation='relu', input_shape=(64,64,3)),
    MaxPooling2D((2,2)),

    Conv2D(64, (3,3), activation='relu'),
```

```
    MaxPooling2D((2,2)),

    Flatten(),
    Dense(128, activation='relu'),
    Dense(num_classes, activation='softmax')
])

model.compile(
    optimizer=Adam(),
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

# ------------------------------------------------------------
# 3. Train the Model
# ------------------------------------------------------------

history = model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=10
)

# ------------------------------------------------------------
# 4. Plot Accuracy vs Epochs
# ------------------------------------------------------------

plt.figure(figsize=(10,4))

plt.subplot(1,2,1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
```

```
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.title("Accuracy vs Epochs")
plt.legend()

# ------------------------------------------------------------
# 5. Plot Loss vs Epochs
# ------------------------------------------------------------

plt.subplot(1,2,2)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Loss vs Epochs")
plt.legend()

plt.show()
```

1. **Load the Corn 3-Classes image dataset.**
2. **Preprocess the images:**
    a. **Resize images to a fixed size (e.g., 224×224)**
    b. **Normalize pixel values.**
3. **Split the dataset into training and testing sets.**
4. **Create a CNN model using:**
    a. **Convolution layer**
    b. **Max Pooling layer**
    c. **Flatten layer**
    d. **Dense layer**
5. **Train the CNN model for multi-class classification.**
6. **Test the model on unseen images.**

7. Plot graphs:

a. Training vs Validation Accuracy

b. Training vs Validation Loss (Error)

```
# ============================================================
# CNN for Corn 3-Class Image Classification
# ============================================================

import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

# ------------------------------------------------------------
# 1. Load Corn Dataset & Split into Train/Test
# ------------------------------------------------------------

train_ds = tf.keras.utils.image_dataset_from_directory(
    "corn_dataset/",
    image_size=(224, 224),        # a) Resize images
    batch_size=32,
    validation_split=0.2,         # train-test split
    subset="training",
    seed=42
)

test_ds = tf.keras.utils.image_dataset_from_directory(
    "corn_dataset/",
    image_size=(224, 224),
    batch_size=32,
    validation_split=0.2,
    subset="validation",
```

```
    seed=42
)

num_classes = len(train_ds.class_names)

# ------------------------------------------------------------
# 2. Preprocess Images (Normalization)
# ------------------------------------------------------------

normalizer = tf.keras.layers.Rescaling(1./255)

train_ds = train_ds.map(lambda x, y: (normalizer(x), y))
test_ds = test_ds.map(lambda x, y: (normalizer(x), y))

# ------------------------------------------------------------
# 3. Create CNN Model
# (Conv → Pool → Flatten → Dense)
# ------------------------------------------------------------

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)),

    Flatten(),
    Dense(128, activation='relu'),
    Dense(num_classes, activation='softmax')
])
```

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

# ------------------------------------------------------------
# 4. Train CNN Model
# ------------------------------------------------------------

history = model.fit(
    train_ds,
    validation_data=test_ds,
    epochs=10
)

# ------------------------------------------------------------
# 5. Test Model on Unseen Images
# ------------------------------------------------------------

test_loss, test_accuracy = model.evaluate(test_ds)
print("Test Accuracy:", test_accuracy)
print("Test Loss:", test_loss)

# ------------------------------------------------------------
# 6. Plot Training vs Validation Accuracy
# ------------------------------------------------------------

plt.figure(figsize=(10,4))

plt.subplot(1,2,1)
```

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

# ------------------------------------------------------------
# 7. Plot Training vs Validation Loss
# ------------------------------------------------------------

plt.subplot(1,2,2)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Training vs Validation Loss")
plt.legend()

plt.show()
```

Implement Self Organizing Map for anomaly Detection

1. **Use Credit Card Applications Dataset:**
2. **Detect fraud customers in the dataset using SOM and perform hyperparameter tuning**
3. **Show map and use markers to distinguish frauds.**

```
# ============================================================
# Self Organizing Map (SOM) for Fraud Detection
# Credit Card Applications Dataset
# ============================================================

import numpy as np
```

```
import pandas as pd
import matplotlib.pyplot as plt
from minisom import MiniSom
from sklearn.preprocessing import MinMaxScaler

# ------------------------------------------------------------
# 1. Load Credit Card Applications Dataset
# ------------------------------------------------------------

# Dataset GIVEN as CSV
# Columns usually include customer details + last column = approval (0/1)
dataset = pd.read_csv("Credit_Card_Applications.csv")

X = dataset.iloc[:, :-1].values   # customer features
y = dataset.iloc[:, -1].values    # approval (used only for visualization)

# ------------------------------------------------------------
# 2. Normalize Data
# ------------------------------------------------------------

scaler = MinMaxScaler(feature_range=(0,1))
X_scaled = scaler.fit_transform(X)

# ------------------------------------------------------------
# 3. Hyperparameter Tuning (SOM Size Selection)
# ------------------------------------------------------------

# Best practical values for anomaly detection
som_x = 10
som_y = 10
sigma = 1.0
```

```
learning_rate = 0.5

som = MiniSom(
    x=som_x,
    y=som_y,
    input_len=X_scaled.shape[1],
    sigma=sigma,
    learning_rate=learning_rate,
    random_seed=42
)

som.random_weights_init(X_scaled)
som.train_random(X_scaled, num_iteration=100)

# ---------------------------------------------------------
# 4. Distance Map (Anomaly Detection)
# ---------------------------------------------------------

distance_map = som.distance_map()

# ---------------------------------------------------------
# 5. Visualization with Fraud Markers
# ---------------------------------------------------------

plt.figure(figsize=(7,7))
plt.title("Self Organizing Map – Fraud Detection")

plt.imshow(distance_map, cmap='coolwarm')
plt.colorbar(label="Distance (Anomaly Score)")

markers = ['o', 's']
```

```
colors = ['g', 'r']

for i, x in enumerate(X_scaled):
    w = som.winner(x)
    plt.plot(
        w[0] + 0.5,
        w[1] + 0.5,
        markers[y[i]],
        markeredgecolor=colors[y[i]],
        markerfacecolor='None',
        markersize=10,
        markeredgewidth=2
    )

plt.show()

# ------------------------------------------------------------
# 6. Extract Fraud Customers (Outliers)
# ------------------------------------------------------------

threshold = np.percentile(distance_map, 90)

frauds = []

for i, x in enumerate(X_scaled):
    w = som.winner(x)
    if distance_map[w[0], w[1]] > threshold:
        frauds.append(dataset.iloc[i, 0])  # customer ID

print("Detected Fraud Customers:")
print(frauds)
```

Task 1

Implement the NAND Boolean Logic Gate using a Perceptron Neural Network.

- **Inputs: x1, x2, bias**
- **Train using perceptron learning rule**
- **Output: y**
- **Display final weights and bias**
- **Verify truth table results**

Task 2

Use the Iris Dataset

- **Normalize the input features**
- **Perform Min–Max scaling**
- **Visualize original vs normalized features**

```
# ==========================================
# TASK 1: NAND Gate using Perceptron
# ==========================================

import numpy as np

# Input dataset (x1, x2)
X = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
])

# NAND output
y = np.array([1, 1, 1, 0])

# Initialize weights and bias
```

```
weights = np.zeros(2)
bias = 0
learning_rate = 0.1
epochs = 10

# Step activation function
def step(z):
    return 1 if z >= 0 else 0

# Training using perceptron learning rule
for _ in range(epochs):
    for i in range(len(X)):
        z = np.dot(X[i], weights) + bias
        y_pred = step(z)
        error = y[i] - y_pred
        weights += learning_rate * error * X[i]
        bias += learning_rate * error

# Display final weights and bias
print("Final Weights:", weights)
print("Final Bias:", bias)

# Verify NAND truth table
print("\nNAND Gate Truth Table Verification")
for x in X:
    output = step(np.dot(x, weights) + bias)
    print(f"{x} -> {output}")
```

```
# ==========================================
# TASK 2: Min-Max Normalization on Iris Dataset
# ==========================================

import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.preprocessing import MinMaxScaler

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

# Min-Max Scaling
scaler = MinMaxScaler()
X_normalized = scaler.fit_transform(X)

# Visualization: Original vs Normalized
plt.figure(figsize=(10,4))

# Original Data
plt.subplot(1,2,1)
plt.boxplot(X)
plt.title("Original Features")
plt.xticks(range(1,5), feature_names, rotation=45)

# Normalized Data
plt.subplot(1,2,2)
plt.boxplot(X_normalized)
plt.title("Min-Max Normalized Features")
```

```
plt.xticks(range(1,5), feature_names, rotation=45)


plt.tight_layout()
plt.show()
```

Task 1

Implement Multi-output Perceptron for

- **AND gate**
- **OR gate**
- **Display weight matrix and bias vector**

Task 2

Load Flowers Dataset

- **Train CNN model with 3 kernals.**
- **Plot training and validation accuracy using graph.**

```
# ==========================================
# TASK 1: Multi-Output Perceptron (AND, OR)
# ==========================================

import numpy as np

# Inputs
X = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
])

# Targets: [AND, OR]
Y = np.array([
    [0, 0],
```

```
    [0, 1],
    [0, 1],
    [1, 1]
])

# Initialize weights (2 inputs → 2 outputs) and bias
weights = np.zeros((2, 2))
bias = np.zeros(2)

learning_rate = 0.1
epochs = 10

# Step activation
def step(z):
    return np.where(z >= 0, 1, 0)

# Training
for _ in range(epochs):
    for i in range(len(X)):
        z = np.dot(X[i], weights) + bias
        y_pred = step(z)
        error = Y[i] - y_pred
        weights += learning_rate * np.outer(X[i], error)
        bias += learning_rate * error

# Display results
print("Weight Matrix:\n", weights)
print("Bias Vector:\n", bias)

print("\nTruth Table Verification [AND, OR]")
for x in X:
```

```
    out = step(np.dot(x, weights) + bias)
    print(x, "→", out)

# ==========================================
# TASK 2: CNN on Flowers Dataset
# ==========================================

import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

# Load dataset
train_ds = tf.keras.utils.image_dataset_from_directory(
    "flowers/",
    image_size=(64, 64),
    batch_size=32,
    validation_split=0.2,
    subset="training",
    seed=42
)

val_ds = tf.keras.utils.image_dataset_from_directory(
    "flowers/",
    image_size=(64, 64),
    batch_size=32,
    validation_split=0.2,
    subset="validation",
    seed=42
)
```

```
# Normalize
normalizer = tf.keras.layers.Rescaling(1./255)
train_ds = train_ds.map(lambda x, y: (normalizer(x), y))
val_ds = val_ds.map(lambda x, y: (normalizer(x), y))

num_classes = len(train_ds.class_names)

# CNN Model with 3 kernels
model = Sequential([
    Conv2D(3, (3,3), activation='relu', input_shape=(64,64,3)),  # 3 kernels
    MaxPooling2D((2,2)),
    Flatten(),
    Dense(64, activation='relu'),
    Dense(num_classes, activation='softmax')
])

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

# Train
history = model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=10
)

# 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.title("Training vs Validation Accuracy")
plt.legend()
plt.show()
```

Implement perceptron

- **Train on AND,OR gate**
- **Compare convergence with normal perceptron**

Task 2

Use Iris Dataset

- **Encode using Autoencoder (3 neurons)**
- **Decode and reconstruct**
- **Plot original vs reconstructed data**

```
# ==========================================
# TASK 1: Perceptron for AND & OR Gates
# ==========================================

import numpy as np

# Input patterns
X = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
])

# Targets
y_and = np.array([0, 0, 0, 1])
y_or  = np.array([0, 1, 1, 1])
```

```
# Perceptron training function
def train_perceptron(X, y, lr=0.1, max_epochs=20):
    w = np.zeros(X.shape[1])
    b = 0
    epochs = 0

    for epoch in range(max_epochs):
        error_count = 0
        for i in range(len(X)):
            z = np.dot(X[i], w) + b
            y_pred = 1 if z >= 0 else 0
            error = y[i] - y_pred
            if error != 0:
                w += lr * error * X[i]
                b += lr * error
                error_count += 1
        epochs += 1
        if error_count == 0:
            break

    return w, b, epochs

# Train AND gate
w_and, b_and, epochs_and = train_perceptron(X, y_and)

# Train OR gate
w_or, b_or, epochs_or = train_perceptron(X, y_or)

print("AND Gate")
print("Weights:", w_and, "Bias:", b_and, "Epochs to Converge:", epochs_and)
```

```
print("\nOR Gate")
print("Weights:", w_or, "Bias:", b_or, "Epochs to Converge:", epochs_or)

# ==========================================
# TASK 2: Autoencoder on Iris Dataset
# ==========================================

import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense

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

# Normalize data
scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X)

# Autoencoder architecture
input_layer = Input(shape=(4,))
encoded = Dense(3, activation='relu')(input_layer)   # Encoder (3 neurons)
decoded = Dense(4, activation='sigmoid')(encoded)    # Decoder

autoencoder = Model(input_layer, decoded)
autoencoder.compile(optimizer='adam', loss='mse')
```

```
# Train autoencoder
autoencoder.fit(
    X_scaled, X_scaled,
    epochs=100,
    batch_size=16,
    verbose=0
)

# Reconstruct data
X_reconstructed = autoencoder.predict(X_scaled)

# ------------------------------------------------
# Plot Original vs Reconstructed Data
# ------------------------------------------------
plt.figure(figsize=(10,4))

plt.subplot(1,2,1)
plt.plot(X_scaled[:10])
plt.title("Original Iris Data")

plt.subplot(1,2,2)
plt.plot(X_reconstructed[:10])
plt.title("Reconstructed Iris Data")

plt.show()
```

Use dataset with initial values X = [1.0, 2.0], Y = [0.5, 1.5]

- **Initialize neural network with random weights**
- **Compute output using linear activation**
- **Calculate MAE and MSE**
- **Plot loss surface (weight vs loss)**

```
# ============================================================
```

```
# Neural Network with Linear Activation
# ============================================================

import numpy as np
import matplotlib.pyplot as plt

# ------------------------------------------------------------
# 1. Given Dataset
# ------------------------------------------------------------
X = np.array([1.0, 2.0])
Y = np.array([0.5, 1.5])

# ------------------------------------------------------------
# 2. Initialize Neural Network with Random Weights
# ------------------------------------------------------------
np.random.seed(42)
w = np.random.rand()    # random weight
b = np.random.rand()    # random bias

print("Initial Weight:", w)
print("Initial Bias:", b)

# ------------------------------------------------------------
# 3. Compute Output (Linear Activation)
# y = w*x + b
# ------------------------------------------------------------
Y_pred = w * X + b
print("\nPredicted Output:", Y_pred)

# ------------------------------------------------------------
# 4. Calculate MAE and MSE
```

```
# ------------------------------------------------------------
MAE = np.mean(np.abs(Y - Y_pred))
MSE = np.mean((Y - Y_pred) ** 2)

print("\nMean Absolute Error (MAE):", MAE)
print("Mean Squared Error (MSE):", MSE)

# ------------------------------------------------------------
# 5. Plot Loss Surface (Weight vs Loss)
# ------------------------------------------------------------

w_values = np.linspace(-2, 2, 100)
loss_values = []

for w_temp in w_values:
    y_temp = w_temp * X + b
    loss = np.mean((Y - y_temp) ** 2)
    loss_values.append(loss)

plt.plot(w_values, loss_values)
plt.xlabel("Weight")
plt.ylabel("Mean Squared Error")
plt.title("Loss Surface (Weight vs MSE)")
plt.show()
```

Use CIFAR-10 Dataset

- **Train CNN with and without data augmentation**
- **Augmentations: rotate, zoom, distort images**
- **Compare validation accuracy and loss and plot using Graph**

```
# ------------------------------------------------------------
# Load Dataset (IF DATASET IS GIVEN AS IMAGE FOLDERS – EXAM CASE)
# ------------------------------------------------------------
```

```
# Expected folder structure:
# cifar10_dataset/
# ├── airplane/
# ├── automobile/
# ├── bird/
# ├── cat/
# ├── deer/
# ├── dog/
# ├── frog/
# ├── horse/
# ├── ship/
# └── truck/

import tensorflow as tf

train_ds = tf.keras.utils.image_dataset_from_directory(
    "cifar10_dataset/",
    image_size=(32, 32),         # CIFAR-10 image size
    batch_size=64,
    validation_split=0.2,        # 80% training
    subset="training",
    seed=42
)

test_ds = tf.keras.utils.image_dataset_from_directory(
    "cifar10_dataset/",
    image_size=(32, 32),
    batch_size=64,
    validation_split=0.2,        # 20% testing
    subset="validation",
```

```
    seed=42
)

# Normalize images to [0,1]
normalizer = tf.keras.layers.Rescaling(1./255)

train_ds = train_ds.map(lambda x, y: (normalizer(x), y))
test_ds = test_ds.map(lambda x, y: (normalizer(x), y))
```

Implement XOR gate using 2-layer Neural Network

- **Use Adadelta optimizer**
- **Plot accuracy vs epoch**

Fashion-MNIST Classification

- **CNN with RMSProp & Adam**
- **Compare confusion matrices**

```
# ==========================================
# TASK 1: XOR Gate using 2-Layer Neural Network
# ==========================================

import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adadelta

# XOR dataset
X = np.array([
    [0,0],
    [0,1],
    [1,0],
    [1,1]
])
```

```
y = np.array([0,1,1,0])

# Build 2-layer neural network
model_xor = Sequential([
    Dense(4, activation='relu', input_shape=(2,)),  # hidden layer
    Dense(1, activation='sigmoid')                  # output layer
])

model_xor.compile(
    optimizer=Adadelta(),
    loss='binary_crossentropy',
    metrics=['accuracy']
)

# Train model
history_xor = model_xor.fit(
    X, y,
    epochs=100,
    verbose=0
)

# Plot Accuracy vs Epoch
plt.plot(history_xor.history['accuracy'])
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.title("XOR Gate – Accuracy vs Epoch (Adadelta)")
plt.show()

# ============================================================
# TASK 2: Fashion-MNIST CNN (Dataset GIVEN)
# Optimizer Comparison: RMSProp vs Adam
```

```
# ============================================================

import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
from tensorflow.keras.optimizers import RMSprop, Adam
from sklearn.metrics import confusion_matrix

# ------------------------------------------------------------
# 1. LOAD DATASET (GIVEN AS IMAGE FOLDERS)
# ------------------------------------------------------------

train_ds = tf.keras.utils.image_dataset_from_directory(
    "fashion_mnist/",
    image_size=(28, 28),
    batch_size=64,
    validation_split=0.2,
    subset="training",
    seed=42
)

test_ds = tf.keras.utils.image_dataset_from_directory(
    "fashion_mnist/",
    image_size=(28, 28),
    batch_size=64,
    validation_split=0.2,
    subset="validation",
    seed=42
)
```

```
class_names = train_ds.class_names
num_classes = len(class_names)

# ------------------------------------------------------------
# 2. PREPROCESSING (NORMALIZATION)
# ------------------------------------------------------------

normalizer = tf.keras.layers.Rescaling(1./255)

train_ds = train_ds.map(lambda x, y: (normalizer(x), y))
test_ds = test_ds.map(lambda x, y: (normalizer(x), y))

# ------------------------------------------------------------
# 3. CNN MODEL DEFINITION
# ------------------------------------------------------------

def create_cnn(optimizer):
    model = Sequential([
        Conv2D(32, (3,3), activation='relu', input_shape=(28,28,3)),
        MaxPooling2D((2,2)),
        Flatten(),
        Dense(128, activation='relu'),
        Dense(num_classes, activation='softmax')
    ])

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

```
    return model

# ------------------------------------------------------------
# 4. TRAIN CNN WITH RMSProp
# ------------------------------------------------------------

model_rms = create_cnn(RMSprop())
model_rms.fit(
    train_ds,
    epochs=5,
    verbose=0
)

# ------------------------------------------------------------
# 5. TRAIN CNN WITH Adam
# ------------------------------------------------------------

model_adam = create_cnn(Adam())
model_adam.fit(
    train_ds,
    epochs=5,
    verbose=0
)

# ------------------------------------------------------------
# 6. CONFUSION MATRIX COMPARISON
# ------------------------------------------------------------

# Get true labels
y_true = np.concatenate([y.numpy() for x, y in test_ds])
```

```
# Predictions RMSProp
y_pred_rms = np.argmax(
    np.vstack([model_rms.predict(x) for x, y in test_ds]),
    axis=1
)

# Predictions Adam
y_pred_adam = np.argmax(
    np.vstack([model_adam.predict(x) for x, y in test_ds]),
    axis=1
)

# Confusion Matrices
cm_rms = confusion_matrix(y_true, y_pred_rms)
cm_adam = confusion_matrix(y_true, y_pred_adam)

print("Confusion Matrix – RMSProp:\n", cm_rms)
print("\nConfusion Matrix – Adam:\n", cm_adam)
```

//option 2

```
# ==========================================
# TASK 2: Fashion-MNIST CNN (RMSProp vs Adam)
# ==========================================

import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import fashion_mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.optimizers import RMSprop, Adam
from sklearn.metrics import confusion_matrix
```

```
# Load dataset
(X_train, y_train), (X_test, y_test) = fashion_mnist.load_data()

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

y_train_cat = to_categorical(y_train, 10)
y_test_cat = to_categorical(y_test, 10)

# CNN model function
def create_cnn(optimizer):
    model = Sequential([
        Conv2D(32,(3,3),activation='relu',input_shape=(28,28,1)),
        MaxPooling2D((2,2)),
        Flatten(),
        Dense(128,activation='relu'),
        Dense(10,activation='softmax')
    ])
    model.compile(
        optimizer=optimizer,
        loss='categorical_crossentropy',
        metrics=['accuracy']
    )
    return model

# Train with RMSProp
model_rms = create_cnn(RMSprop())
model_rms.fit(X_train, y_train_cat, epochs=5, verbose=0)
```

```
# Train with Adam
model_adam = create_cnn(Adam())
model_adam.fit(X_train, y_train_cat, epochs=5, verbose=0)

# Predictions
pred_rms = np.argmax(model_rms.predict(X_test), axis=1)
pred_adam = np.argmax(model_adam.predict(X_test), axis=1)

# Confusion Matrices
cm_rms = confusion_matrix(y_test, pred_rms)
cm_adam = confusion_matrix(y_test, pred_adam)

print("Confusion Matrix – RMSProp:\n", cm_rms)
print("\nConfusion Matrix – Adam:\n", cm_adam)
```

Implement the backpropagation algorithm.

1. **Take Iris Dataset**
2. **Initialize a neural network with random weights.**
3. **Calculate error**
4. **Perform multiple iterations of NN**
5. **Update weights accordingly.**
6. **Plot accuracy for iterations and note the results.**

```
# ============================================================
# Backpropagation Algorithm from Scratch - Iris Dataset
# ============================================================

import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.preprocessing import MinMaxScaler
```

```
# ------------------------------------------------------------
# 1. Load Iris Dataset
# ------------------------------------------------------------

# If dataset is GIVEN as CSV (exam case)
# import pandas as pd
# df = pd.read_csv("iris.csv")
# X = df.iloc[:, :-1].values
# y = df.iloc[:, -1].values

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

# Convert to binary classification (Setosa vs Others)
y = (y == 0).astype(int).reshape(-1, 1)

# ------------------------------------------------------------
# 2. Normalize Input Features
# ------------------------------------------------------------
scaler = MinMaxScaler()
X = scaler.fit_transform(X)

# ------------------------------------------------------------
# 3. Initialize Neural Network with Random Weights
# ------------------------------------------------------------
np.random.seed(42)

input_neurons = X.shape[1]
hidden_neurons = 5
```

```
output_neurons = 1

W1 = np.random.rand(input_neurons, hidden_neurons)
b1 = np.random.rand(1, hidden_neurons)

W2 = np.random.rand(hidden_neurons, output_neurons)
b2 = np.random.rand(1, output_neurons)

learning_rate = 0.5
epochs = 100

accuracy_history = []

# ---------------------------------------------------------
# Activation Functions
# ---------------------------------------------------------
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def sigmoid_derivative(a):
    return a * (1 - a)

# ---------------------------------------------------------
# 4 & 5. Backpropagation Training Loop
# ---------------------------------------------------------
for epoch in range(epochs):

    # ---------- Forward Pass ----------
    z1 = np.dot(X, W1) + b1
    a1 = sigmoid(z1)
```

```
    z2 = np.dot(a1, W2) + b2
    y_pred = sigmoid(z2)

    # ---------- 3. Calculate Error (Squared Error) ----------
    error = y_pred - y
    squared_error = error ** 2

    # ---------- Backward Pass ----------
    d_output = error * sigmoid_derivative(y_pred)
    d_hidden = d_output.dot(W2.T) * sigmoid_derivative(a1)

    # ---------- Update Weights & Bias ----------
    W2 -= learning_rate * np.dot(a1.T, d_output)
    b2 -= learning_rate * np.sum(d_output, axis=0, keepdims=True)

    W1 -= learning_rate * np.dot(X.T, d_hidden)
    b1 -= learning_rate * np.sum(d_hidden, axis=0, keepdims=True)

    # ---------- 6. Accuracy ----------
    predictions = (y_pred >= 0.5).astype(int)
    accuracy = np.mean(predictions == y)
    accuracy_history.append(accuracy)

    if (epoch + 1) % 10 == 0:
        print(f"Epoch {epoch+1} | Accuracy: {accuracy:.4f}")

# ------------------------------------------------------------
# 7. Plot Accuracy vs Iterations
# ------------------------------------------------------------
plt.plot(accuracy_history)
plt.xlabel("Iterations (Epochs)")
```

```
plt.ylabel("Accuracy")
plt.title("Accuracy vs Iterations (Backpropagation on Iris)")
plt.show()
```

Build a multiclass image categorization of CNN network which correctly classifies different categories of images in the dataset.

1. Take Flower dataset

2. Split original dataset to train and test set

3. Build CNN Model

4. Generate the accuracy of the built model using any optimizer.

5. Compare performance of different optimizers on Flower categorization.

```
# ============================================================
# Multiclass CNN for Flower Dataset
# Optimizer Comparison
# ============================================================

import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
from tensorflow.keras.optimizers import Adam, SGD, RMSprop

# ------------------------------------------------------------
# 1. Load Flower Dataset & Split into Train/Test
# ------------------------------------------------------------

train_ds = tf.keras.utils.image_dataset_from_directory(
    "flowers/",
    image_size=(64, 64),
    batch_size=32,
    validation_split=0.2,
```

```
    subset="training",
    seed=42
)

test_ds = tf.keras.utils.image_dataset_from_directory(
    "flowers/",
    image_size=(64, 64),
    batch_size=32,
    validation_split=0.2,
    subset="validation",
    seed=42
)

class_names = train_ds.class_names
num_classes = len(class_names)

# ------------------------------------------------------------
# 2. Normalize Images
# ------------------------------------------------------------

normalizer = tf.keras.layers.Rescaling(1./255)

train_ds = train_ds.map(lambda x, y: (normalizer(x), y))
test_ds = test_ds.map(lambda x, y: (normalizer(x), y))

# ------------------------------------------------------------
# 3. CNN Model Definition
# ------------------------------------------------------------

def create_cnn(optimizer):
    model = Sequential([
```

```
        Conv2D(32, (3,3), activation='relu', input_shape=(64,64,3)),
        MaxPooling2D((2,2)),
        Conv2D(64, (3,3), activation='relu'),
        MaxPooling2D((2,2)),
        Flatten(),
        Dense(128, activation='relu'),
        Dense(num_classes, activation='softmax')
    ])

    model.compile(
        optimizer=optimizer,
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )
    return model

# ------------------------------------------------------------
# 4. Train CNN using Different Optimizers
# ------------------------------------------------------------

optimizers = {
    "Adam": Adam(),
    "SGD": SGD(learning_rate=0.01),
    "RMSprop": RMSprop()
}

results = {}

for name, opt in optimizers.items():
    print(f"\nTraining with {name} optimizer")
    model = create_cnn(opt)
```

```
    history = model.fit(
        train_ds,
        validation_data=test_ds,
        epochs=5,
        verbose=0
    )
    results[name] = history.history['val_accuracy'][-1]

# ------------------------------------------------------------
# 5. Compare Optimizer Performance
# ------------------------------------------------------------

print("\nValidation Accuracy Comparison:")
for opt, acc in results.items():
    print(f"{opt}: {acc:.4f}")
```

Train a small neural network (dataset – Cifar- 100 Classification)

Compare the optimizers:

- **Adagrad**
- **SGD**
- **Adam**

Plot:

- **Training loss vs epochs**
- **Accuracy vs epochs**

// load

```
# ============================================================
# CIFAR-100 Classification - Optimizer Comparison
# Adagrad vs SGD vs Adam
# ============================================================

import numpy as np
import matplotlib.pyplot as plt
```

```
from tensorflow.keras.datasets import cifar100
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.optimizers import Adagrad, SGD, Adam

# ------------------------------------------------------------
# 1. Load CIFAR-100 Dataset
# ------------------------------------------------------------

# If dataset is GIVEN as image folders (exam case)
# train_ds = tf.keras.utils.image_dataset_from_directory(...)
# test_ds = tf.keras.utils.image_dataset_from_directory(...)

(X_train, y_train), (X_test, y_test) = cifar100.load_data(label_mode='fine')

# Normalize images
X_train = X_train / 255.0
X_test = X_test / 255.0

# One-hot encode labels (100 classes)
y_train_cat = to_categorical(y_train, 100)
y_test_cat = to_categorical(y_test, 100)

# ------------------------------------------------------------
# 2. Define a Small CNN Model
# ------------------------------------------------------------

def create_cnn(optimizer):
    model = Sequential([
        Conv2D(32, (3,3), activation='relu', input_shape=(32,32,3)),
```

```
        MaxPooling2D((2,2)),
        Conv2D(64, (3,3), activation='relu'),
        MaxPooling2D((2,2)),
        Flatten(),
        Dense(256, activation='relu'),
        Dense(100, activation='softmax')
    ])

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

# ------------------------------------------------------------
# 3. Training Settings
# ------------------------------------------------------------

epochs = 10
batch_size = 64

optimizers = {
    "Adagrad": Adagrad(),
    "SGD": SGD(learning_rate=0.01),
    "Adam": Adam()
}

histories = {}

# ------------------------------------------------------------
```

```
# 4. Train Model with Different Optimizers
# ------------------------------------------------------------

for name, opt in optimizers.items():
    print(f"\nTraining with {name} optimizer")
    model = create_cnn(opt)
    history = model.fit(
        X_train, y_train_cat,
        epochs=epochs,
        batch_size=batch_size,
        validation_data=(X_test, y_test_cat),
        verbose=0
    )
    histories[name] = history

# ------------------------------------------------------------
# 5. Plot Training Loss vs Epochs
# ------------------------------------------------------------

plt.figure(figsize=(10,4))

plt.subplot(1,2,1)
for name, history in histories.items():
    plt.plot(history.history['loss'], label=name)

plt.xlabel("Epochs")
plt.ylabel("Training Loss")
plt.title("Training Loss vs Epochs")
plt.legend()

# ------------------------------------------------------------
```

```
# 6. Plot Accuracy vs Epochs
# ------------------------------------------------------------

plt.subplot(1,2,2)
for name, history in histories.items():
    plt.plot(history.history['accuracy'], label=name)

plt.xlabel("Epochs")
plt.ylabel("Training Accuracy")
plt.title("Accuracy vs Epochs")
plt.legend()

plt.show()
```

// ig dir

```
# ============================================================
# CIFAR-100 Classification - Optimizer Comparison
# Adagrad vs SGD vs Adam
# ============================================================

import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
from tensorflow.keras.optimizers import Adagrad, SGD, Adam
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.datasets import cifar100

# ------------------------------------------------------------
# 1. LOAD DATASET
# ------------------------------------------------------------
```

```
# ---------- CASE 1: DATASET NOT GIVEN (Use Keras CIFAR-100) ----------
"""
(X_train, y_train), (X_test, y_test) = cifar100.load_data(label_mode='fine')

X_train = X_train / 255.0
X_test = X_test / 255.0

y_train = to_categorical(y_train, 100)
y_test = to_categorical(y_test, 100)
"""

# ---------- CASE 2: DATASET GIVEN AS IMAGE FOLDERS (EXAM CASE) ----------

train_ds = tf.keras.utils.image_dataset_from_directory(
    "cifar100/",
    image_size=(32, 32),
    batch_size=64,
    validation_split=0.2,
    subset="training",
    seed=42
)

test_ds = tf.keras.utils.image_dataset_from_directory(
    "cifar100/",
    image_size=(32, 32),
    batch_size=64,
    validation_split=0.2,
    subset="validation",
    seed=42
)
```

```
# Normalize images
normalizer = tf.keras.layers.Rescaling(1./255)
train_ds = train_ds.map(lambda x, y: (normalizer(x), y))
test_ds = test_ds.map(lambda x, y: (normalizer(x), y))

num_classes = 100

# ------------------------------------------------------------
# 2. SMALL CNN MODEL
# ------------------------------------------------------------

def create_cnn(optimizer):
    model = Sequential([
        Conv2D(32, (3,3), activation='relu', input_shape=(32,32,3)),
        MaxPooling2D((2,2)),
        Conv2D(64, (3,3), activation='relu'),
        MaxPooling2D((2,2)),
        Flatten(),
        Dense(256, activation='relu'),
        Dense(num_classes, activation='softmax')
    ])

    model.compile(
        optimizer=optimizer,
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )
    return model

# ------------------------------------------------------------
```

```
# 3. TRAINING PARAMETERS
# ------------------------------------------------------------

epochs = 10

optimizers = {
    "Adagrad": Adagrad(),
    "SGD": SGD(learning_rate=0.01),
    "Adam": Adam()
}

histories = {}

# ------------------------------------------------------------
# 4. TRAIN WITH DIFFERENT OPTIMIZERS
# ------------------------------------------------------------

for name, opt in optimizers.items():
    print(f"\nTraining with {name}")
    model = create_cnn(opt)
    history = model.fit(
        train_ds,
        validation_data=test_ds,
        epochs=epochs,
        verbose=0
    )
    histories[name] = history

# ------------------------------------------------------------
# 5. PLOT TRAINING LOSS vs EPOCHS
# ------------------------------------------------------------
```

```
plt.figure(figsize=(10,4))

plt.subplot(1,2,1)
for name, history in histories.items():
   plt.plot(history.history['loss'], label=name)

plt.xlabel("Epochs")
plt.ylabel("Training Loss")
plt.title("Training Loss vs Epochs")
plt.legend()

# ----------------------------------------------------------
# 6. PLOT ACCURACY vs EPOCHS
# ----------------------------------------------------------

plt.subplot(1,2,2)
for name, history in histories.items():
   plt.plot(history.history['accuracy'], label=name)

plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.title("Accuracy vs Epochs")
plt.legend()

plt.show()
```

- **Use IRIS Dataset**
- **Train a model with and without data augmentation (horizontal flip, rotation, noise).**
- **Compare generalization performance on the validation set. (Accuracy & Error)**
- **Plot accuracy vs epochs**
- **Plot loss vs epochs**

```
# ============================================================
# IRIS Dataset - With vs Without Data Augmentation
# ============================================================

import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam

# ------------------------------------------------------------
# 1. Load IRIS Dataset
# ------------------------------------------------------------

# If dataset is GIVEN as CSV (exam case)
# import pandas as pd
# df = pd.read_csv("iris.csv")
# X = df.iloc[:, :-1].values
# y = df.iloc[:, -1].values

iris = load_iris()
X = iris.data
y = iris.target

# ------------------------------------------------------------
# 2. Normalize Features
# ------------------------------------------------------------
```

```
scaler = MinMaxScaler()
X = scaler.fit_transform(X)

# ------------------------------------------------------------
# 3. Train–Validation Split
# ------------------------------------------------------------

X_train, X_val, y_train, y_val = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# ------------------------------------------------------------
# 4. DATA AUGMENTATION (TABULAR)
# ------------------------------------------------------------

def augment_data(X, noise_factor=0.05):
    noise = noise_factor * np.random.normal(size=X.shape)
    X_noisy = X + noise
    return X_noisy

X_train_aug = augment_data(X_train)

# ------------------------------------------------------------
# 5. Neural Network Model
# ------------------------------------------------------------

def create_model():
    model = Sequential([
        Dense(16, activation='relu', input_shape=(4,)),
        Dense(16, activation='relu'),
        Dense(3, activation='softmax')
```

```
    ])
    model.compile(
        optimizer=Adam(),
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )
    return model

# ------------------------------------------------------------
# 6. Train WITHOUT Data Augmentation
# ------------------------------------------------------------

model_no_aug = create_model()

history_no_aug = model_no_aug.fit(
    X_train, y_train,
    validation_data=(X_val, y_val),
    epochs=50,
    verbose=0
)

# ------------------------------------------------------------
# 7. Train WITH Data Augmentation
# ------------------------------------------------------------

model_aug = create_model()

history_aug = model_aug.fit(
    X_train_aug, y_train,
    validation_data=(X_val, y_val),
    epochs=50,
```

```
    verbose=0
)

# ----------------------------------------------------------
# 8. Plot Accuracy vs Epochs
# ----------------------------------------------------------

plt.figure(figsize=(10,4))

plt.subplot(1,2,1)
plt.plot(history_no_aug.history['val_accuracy'], label='Without Augmentation')
plt.plot(history_aug.history['val_accuracy'], label='With Augmentation')
plt.xlabel("Epochs")
plt.ylabel("Validation Accuracy")
plt.title("Accuracy vs Epochs")
plt.legend()

# ----------------------------------------------------------
# 9. Plot Loss vs Epochs
# ----------------------------------------------------------

plt.subplot(1,2,2)
plt.plot(history_no_aug.history['val_loss'], label='Without Augmentation')
plt.plot(history_aug.history['val_loss'], label='With Augmentation')
plt.xlabel("Epochs")
plt.ylabel("Validation Loss")
plt.title("Loss vs Epochs")
plt.legend()

plt.show()
```

Task 1: Build a small CNN for MNIST digits dataset

- **Split dataset into train/test**
- **Use 2 convolution layers + pooling + dense**
- **Metrics / Plots: Accuracy, Confusion Matrix, Precision & Recall, plot training vs validation accuracy and loss**

Task 2: Compare Adam vs SGD optimizer

Metrics / Plots: Plot training loss & accuracy for each optimizer

```
# ============================================================
# TASK 1: CNN for MNIST Digits
# ============================================================
# ------------------------------------------------------------
# LOAD MNIST DATASET (USED FOR TASK 1 AND TASK 2)
# ------------------------------------------------------------

from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical

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

# Reshape & normalize
X_train = X_train.reshape(-1, 28, 28, 1) / 255.0
X_test = X_test.reshape(-1, 28, 28, 1) / 255.0

# One-hot encoding
y_train_cat = to_categorical(y_train, 10)
y_test_cat = to_categorical(y_test, 10)
////////////////////////////////////////////////////
import numpy as np
import matplotlib.pyplot as plt
```

```
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.optimizers import Adam
from sklearn.metrics import confusion_matrix, precision_score, recall_score

# ------------------------------------------------------------
# 1. Load & Split Dataset
# ------------------------------------------------------------

# If dataset is GIVEN as CSV (exam case)
# import pandas as pd
# train_df = pd.read_csv("mnist_train.csv")
# test_df = pd.read_csv("mnist_test.csv")
# X_train = train_df.iloc[:,1:].values.reshape(-1,28,28,1)/255.0
# y_train = train_df.iloc[:,0].values
# X_test = test_df.iloc[:,1:].values.reshape(-1,28,28,1)/255.0
# y_test = test_df.iloc[:,0].values

(X_train, y_train), (X_test, y_test) = mnist.load_data()

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

y_train_cat = to_categorical(y_train, 10)
y_test_cat = to_categorical(y_test, 10)

# ------------------------------------------------------------
# 2. Build CNN Model (2 Conv Layers)
# ------------------------------------------------------------
```

```
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)),
    Flatten(),
    Dense(128, activation='relu'),
    Dense(10, activation='softmax')
])

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

# ------------------------------------------------------------
# 3. Train Model
# ------------------------------------------------------------

history = model.fit(
    X_train, y_train_cat,
    validation_data=(X_test, y_test_cat),
    epochs=5,
    batch_size=64
)

# ------------------------------------------------------------
# 4. Evaluation Metrics
# ------------------------------------------------------------
```

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

cm = confusion_matrix(y_test, y_pred)
precision = precision_score(y_test, y_pred, average='macro')
recall = recall_score(y_test, y_pred, average='macro')

print("Confusion Matrix:\n", cm)
print("Precision:", precision)
print("Recall:", recall)

# ------------------------------------------------------------
# 5. Plot Training vs Validation Accuracy & Loss
# ------------------------------------------------------------

plt.figure(figsize=(10,4))

plt.subplot(1,2,1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Val Accuracy')
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.title("Training vs Validation Accuracy")
plt.legend()

plt.subplot(1,2,2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Val Loss')
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Training vs Validation Loss")
```

```
plt.legend()


plt.show()
```

- **Calculate and plot all activation functions (Sigmoid and tanh) for input ranging in (-10, +10)**
- **Calculate and plot Derivative of given Activation function and plot also observe the behaviour of curves.**
- **Consider a target vector Y and prediction vector Ŷ. Calculate MSE and MAE.**

```
# ============================================================
# Activation Functions, Derivatives, and Error Metrics
# ============================================================

import numpy as np
import matplotlib.pyplot as plt

# ------------------------------------------------------------
# 1. Input Range
# ------------------------------------------------------------

x = np.linspace(-10, 10, 400)

# ------------------------------------------------------------
# 2. Activation Functions
# ------------------------------------------------------------

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def tanh(x):
    return np.tanh(x)

sigmoid_y = sigmoid(x)
```

```
tanh_y = tanh(x)

# ------------------------------------------------------------
# 3. Derivatives of Activation Functions
# ------------------------------------------------------------

def sigmoid_derivative(x):
    s = sigmoid(x)
    return s * (1 - s)

def tanh_derivative(x):
    return 1 - np.tanh(x)**2

sigmoid_deriv = sigmoid_derivative(x)
tanh_deriv = tanh_derivative(x)

# ------------------------------------------------------------
# 4. Plot Activation Functions
# ------------------------------------------------------------

plt.figure(figsize=(10,4))

plt.subplot(1,2,1)
plt.plot(x, sigmoid_y, label="Sigmoid")
plt.plot(x, tanh_y, label="Tanh")
plt.xlabel("Input")
plt.ylabel("Output")
plt.title("Activation Functions")
plt.legend()

# ------------------------------------------------------------
```

```
# 5. Plot Derivatives
# ------------------------------------------------------------

plt.subplot(1,2,2)
plt.plot(x, sigmoid_deriv, label="Sigmoid Derivative")
plt.plot(x, tanh_deriv, label="Tanh Derivative")
plt.xlabel("Input")
plt.ylabel("Derivative")
plt.title("Derivatives of Activation Functions")
plt.legend()

plt.show()

# ------------------------------------------------------------
# 6. Error Calculation (MSE & MAE)
# ------------------------------------------------------------

# Given target and prediction vectors
Y = np.array([1.0, 0.0, 1.0, 1.0])
Y_hat = np.array([0.8, 0.2, 0.6, 0.9])

# Mean Squared Error
MSE = np.mean((Y - Y_hat) ** 2)

# Mean Absolute Error
MAE = np.mean(np.abs(Y - Y_hat))

print("Target Vector (Y):", Y)
print("Prediction Vector (Y_hat):", Y_hat)
print("Mean Squared Error (MSE):", MSE)
print("Mean Absolute Error (MAE):", MAE)
```

- **Calculate and plot all activation functions (Tanh and Relu) for input ranging in (-5, +5)**
- **Calculate and plot Derivative of given Activation function and plot also observe the behaviour of curves.**
- **Consider a target vector Y and prediction vector Ŷ. Calculate MSE and MAE.**

```
# ============================================================
# Tanh & ReLU Activation Functions, Derivatives, MSE & MAE
# ============================================================

import numpy as np
import matplotlib.pyplot as plt

# ------------------------------------------------------------
# 1. Input Range (-5, +5)
# ------------------------------------------------------------
x = np.linspace(-5, 5, 400)

# ------------------------------------------------------------
# 2. Activation Functions
# ------------------------------------------------------------
def tanh(x):
    return np.tanh(x)

def relu(x):
    return np.maximum(0, x)

tanh_y = tanh(x)
relu_y = relu(x)

# ------------------------------------------------------------
# 3. Derivatives of Activation Functions
# ------------------------------------------------------------
```

```
def tanh_derivative(x):
    return 1 - np.tanh(x)**2

def relu_derivative(x):
    return np.where(x > 0, 1, 0)

tanh_d = tanh_derivative(x)
relu_d = relu_derivative(x)

# ------------------------------------------------------------
# 4. Plot Activation Functions
# ------------------------------------------------------------
plt.figure(figsize=(10,4))

plt.subplot(1,2,1)
plt.plot(x, tanh_y, label="Tanh")
plt.plot(x, relu_y, label="ReLU")
plt.xlabel("Input")
plt.ylabel("Output")
plt.title("Activation Functions")
plt.legend()

# ------------------------------------------------------------
# 5. Plot Derivatives
# ------------------------------------------------------------
plt.subplot(1,2,2)
plt.plot(x, tanh_d, label="Tanh Derivative")
plt.plot(x, relu_d, label="ReLU Derivative")
plt.xlabel("Input")
plt.ylabel("Derivative")
plt.title("Derivatives of Activation Functions")
```

```
plt.legend()


plt.show()


# ------------------------------------------------------------
# 6. Error Calculation (MSE & MAE)
# ------------------------------------------------------------


# Given target and predicted values
Y = np.array([1.0, 0.0, 1.0, 1.0])
Y_hat = np.array([0.9, 0.2, 0.6, 0.8])


# Mean Squared Error
MSE = np.mean((Y - Y_hat) ** 2)


# Mean Absolute Error
MAE = np.mean(np.abs(Y - Y_hat))


print("Target Vector (Y):", Y)
print("Prediction Vector (Ŷ):", Y_hat)
print("Mean Squared Error (MSE):", MSE)
print("Mean Absolute Error (MAE):", MAE)
```

Task 1

Implement the AND Boolean logic gate using perceptron Neural Network. Inputs = x1, x2 and bias, weights should be fed into the perceptron with single Output = y. Display final weights and bias of each perceptron.

Task 2

1. Use the titanic Dataset

2.Create an Auto Encoder and fit it with our data using 3 neurons in the dense layer

3. Display new reduced dimension values

4.Plot loss for different encoders [Sparse Autoencoder, Noise Autoencoder]

```
# ============================================================
# TASK 1: AND Gate using Perceptron Neural Network
# ============================================================

import numpy as np

# Input data
X = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
])

# AND gate output
y = np.array([0, 0, 0, 1])

# Initialize weights and bias
weights = np.zeros(2)
bias = 0.0
learning_rate = 0.1
epochs = 10

# Step activation function
def step(z):
    return 1 if z >= 0 else 0

# Training perceptron
for _ in range(epochs):
    for i in range(len(X)):
        z = np.dot(X[i], weights) + bias
```

```
        y_pred = step(z)
        error = y[i] - y_pred
        weights += learning_rate * error * X[i]
        bias += learning_rate * error

# Display final parameters
print("Final Weights:", weights)
print("Final Bias:", bias)

# Verify AND gate
print("\nAND Gate Verification")
for x in X:
    output = step(np.dot(x, weights) + bias)
    print(f"{x} -> {output}")

# ============================================================
# TASK 2: Autoencoders on Titanic Dataset
# ============================================================

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense, GaussianNoise
from tensorflow.keras.regularizers import l1
from tensorflow.keras.optimizers import Adam

# ------------------------------------------------------------
# 1. Load Titanic Dataset
# ------------------------------------------------------------
```

```
# IF DATASET IS GIVEN AS CSV (EXAM CASE)
df = pd.read_csv("titanic.csv")

# Select numeric features only
df = df[['Age', 'Fare', 'Pclass']]
df = df.fillna(df.mean())

X = df.values

# Normalize data
scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X)

# ------------------------------------------------------------
# 2. Sparse Autoencoder (3 Neurons)
# ------------------------------------------------------------

input_dim = X_scaled.shape[1]
input_layer = Input(shape=(input_dim,))

encoded_sparse = Dense(
    3, activation='relu',
    activity_regularizer=l1(1e-3)
)(input_layer)

decoded_sparse = Dense(input_dim, activation='sigmoid')(encoded_sparse)

sparse_autoencoder = Model(input_layer, decoded_sparse)
sparse_autoencoder.compile(
    optimizer=Adam(),
```

```
    loss='mse'
)

history_sparse = sparse_autoencoder.fit(
    X_scaled, X_scaled,
    epochs=50,
    batch_size=16,
    verbose=0
)

# Get reduced dimensions
encoder_sparse = Model(input_layer, encoded_sparse)
reduced_sparse = encoder_sparse.predict(X_scaled)

print("\nReduced Dimensions (Sparse Autoencoder):")
print(reduced_sparse[:5])

# ------------------------------------------------------------
# 3. Noise (Denoising) Autoencoder
# ------------------------------------------------------------

noisy_input = GaussianNoise(0.1)(input_layer)
encoded_noise = Dense(3, activation='relu')(noisy_input)
decoded_noise = Dense(input_dim, activation='sigmoid')(encoded_noise)

noise_autoencoder = Model(input_layer, decoded_noise)
noise_autoencoder.compile(
    optimizer=Adam(),
    loss='mse'
)
```

```
history_noise = noise_autoencoder.fit(
    X_scaled, X_scaled,
    epochs=50,
    batch_size=16,
    verbose=0
)

# ------------------------------------------------------------
# 4. Plot Loss Comparison
# ------------------------------------------------------------

plt.plot(history_sparse.history['loss'], label='Sparse Autoencoder')
plt.plot(history_noise.history['loss'], label='Noise Autoencoder')
plt.xlabel("Epochs")
plt.ylabel("Loss (MSE)")
plt.title("Loss Comparison of Autoencoders")
plt.legend()
plt.show()
```

Use dataset – MNIST digit classification

Create a neural network and apply following optimizers

- **SGD**
- **SGD + Momentum**
- **Adam**

Plot the comparison using ROC curve

CASE 1: DATASET NOT GIVEN (Built-in MNIST)

```
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical

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

```
# Normalize
X_train = X_train / 255.0
X_test = X_test / 255.0

# One-hot encoding
y_train_cat = to_categorical(y_train, 10)
y_test_cat = to_categorical(y_test, 10)
```

CASE 2: DATASET GIVEN (CSV FILES)

```
import pandas as pd
from tensorflow.keras.utils import to_categorical

train_df = pd.read_csv("mnist_train.csv")
test_df = pd.read_csv("mnist_test.csv")

X_train = train_df.iloc[:, 1:].values.reshape(-1, 28, 28) / 255.0
y_train = train_df.iloc[:, 0].values

X_test = test_df.iloc[:, 1:].values.reshape(-1, 28, 28) / 255.0
y_test = test_df.iloc[:, 0].values

y_train_cat = to_categorical(y_train, 10)
y_test_cat = to_categorical(y_test, 10)
# ============================================================
# MNIST Digit Classification - ROC Curve Comparison
# Optimizers: SGD, SGD + Momentum, Adam
# ============================================================

import numpy as np
import matplotlib.pyplot as plt
```

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.optimizers import SGD, Adam
from tensorflow.keras.utils import to_categorical
from sklearn.metrics import roc_curve, auc
from sklearn.preprocessing import label_binarize

# ------------------------------------------------------------
# DATASET LOADING (WRITE ANY ONE BASED ON QUESTION)
# ------------------------------------------------------------

# -------- CASE 1: DATASET NOT GIVEN (BUILT-IN MNIST) --------
from tensorflow.keras.datasets import mnist
(X_train, y_train), (X_test, y_test) = mnist.load_data()

X_train = X_train / 255.0
X_test  = X_test / 255.0

# -------- CASE 2: DATASET GIVEN AS CSV --------
"""
import pandas as pd
train_df = pd.read_csv("mnist_train.csv")
test_df = pd.read_csv("mnist_test.csv")

X_train = train_df.iloc[:,1:].values.reshape(-1,28,28)/255.0
y_train = train_df.iloc[:,0].values
X_test = test_df.iloc[:,1:].values.reshape(-1,28,28)/255.0
y_test = test_df.iloc[:,0].values
"""

# One-hot encoding
```

```
y_train_cat = to_categorical(y_train, 10)
y_test_cat  = to_categorical(y_test, 10)

# For ROC (One-vs-Rest)
y_test_bin = label_binarize(y_test, classes=np.arange(10))

# ------------------------------------------------------------
# NEURAL NETWORK MODEL
# ------------------------------------------------------------

def create_model(optimizer):
    model = Sequential([
        Flatten(input_shape=(28,28)),
        Dense(128, activation='relu'),
        Dense(10, activation='softmax')
    ])
    model.compile(
        optimizer=optimizer,
        loss='categorical_crossentropy',
        metrics=['accuracy']
    )
    return model

# ------------------------------------------------------------
# TRAIN WITH DIFFERENT OPTIMIZERS
# ------------------------------------------------------------

optimizers = {
    "SGD": SGD(learning_rate=0.01),
    "SGD + Momentum": SGD(learning_rate=0.01, momentum=0.9),
    "Adam": Adam()
```

```
}

roc_results = {}

for name, opt in optimizers.items():
    model = create_model(opt)
    model.fit(X_train, y_train_cat, epochs=5, batch_size=128, verbose=0)
    y_scores = model.predict(X_test)
    roc_results[name] = y_scores

# ------------------------------------------------------------
# ROC CURVE COMPARISON
# ------------------------------------------------------------

plt.figure(figsize=(8,6))

for name, scores in roc_results.items():
    fpr, tpr, _ = roc_curve(y_test_bin.ravel(), scores.ravel())
    roc_auc = auc(fpr, tpr)
    plt.plot(fpr, tpr, label=f"{name} (AUC = {roc_auc:.3f})")

plt.plot([0,1], [0,1], 'k--')
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve Comparison – MNIST Optimizers")
plt.legend()
plt.show()
// folder
# ============================================================
# MNIST Digit Classification (Dataset Given as Image Folders)
# Optimizer Comparison using ROC Curve
```

```
# ============================================================

import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.optimizers import SGD, Adam
from sklearn.metrics import roc_curve, auc
from sklearn.preprocessing import label_binarize

# ------------------------------------------------------------
# 1. LOAD MNIST DATASET FROM FOLDER
# ------------------------------------------------------------

train_ds = tf.keras.utils.image_dataset_from_directory(
    "mnist/",
    image_size=(28, 28),
    batch_size=128,
    validation_split=0.2,
    subset="training",
    seed=42,
    color_mode="grayscale"
)

test_ds = tf.keras.utils.image_dataset_from_directory(
    "mnist/",
    image_size=(28, 28),
    batch_size=128,
    validation_split=0.2,
    subset="validation",
```

```
    seed=42,
    color_mode="grayscale"
)

# Normalize images
normalizer = tf.keras.layers.Rescaling(1./255)
train_ds = train_ds.map(lambda x, y: (normalizer(x), y))
test_ds  = test_ds.map(lambda x, y: (normalizer(x), y))

# Extract test labels for ROC
y_test = np.concatenate([y.numpy() for x, y in test_ds])
y_test_bin = label_binarize(y_test, classes=np.arange(10))

# ------------------------------------------------------------
# 2. NEURAL NETWORK MODEL
# ------------------------------------------------------------

def create_model(optimizer):
    model = Sequential([
        Flatten(input_shape=(28,28,1)),
        Dense(128, activation='relu'),
        Dense(10, activation='softmax')
    ])
    model.compile(
        optimizer=optimizer,
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )
    return model

# ------------------------------------------------------------
```

```
# 3. TRAIN WITH DIFFERENT OPTIMIZERS
# ------------------------------------------------------------

optimizers = {
    "SGD": SGD(learning_rate=0.01),
    "SGD + Momentum": SGD(learning_rate=0.01, momentum=0.9),
    "Adam": Adam()
}

roc_results = {}

for name, opt in optimizers.items():
    print(f"Training with {name}")
    model = create_model(opt)
    model.fit(train_ds, epochs=5, verbose=0)

    # Collect prediction scores
    y_scores = np.vstack([model.predict(x) for x, y in test_ds])
    roc_results[name] = y_scores

# ------------------------------------------------------------
# 4. ROC CURVE COMPARISON
# ------------------------------------------------------------

plt.figure(figsize=(8,6))

for name, scores in roc_results.items():
    fpr, tpr, _ = roc_curve(y_test_bin.ravel(), scores.ravel())
    roc_auc = auc(fpr, tpr)
    plt.plot(fpr, tpr, label=f"{name} (AUC = {roc_auc:.3f})")
```

```
plt.plot([0,1], [0,1], 'k--')
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve Comparison – MNIST Optimizers")
plt.legend()
plt.show()
```

1. **Use MNIST or IRIS/ Cifar-10 Dataset**
2. **Train a model with and without data augmentation (horizontal flip, rotation, noise).**
3. **Compare generalization performance on the validation set. (Accuracy & Error)**
4. **Observe improvements and plot the graph.**

```
# ============================================================
# MNIST - With vs Without Data Augmentation
# ============================================================

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
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.optimizers import Adam

# ------------------------------------------------------------
# 1. Load MNIST Dataset
# ------------------------------------------------------------

(X_train, y_train), (X_test, y_test) = mnist.load_data()

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

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

y_train_cat = to_categorical(y_train, 10)
y_test_cat  = to_categorical(y_test, 10)

# ------------------------------------------------------------
# 2. CNN Model Definition
# ------------------------------------------------------------

def create_cnn():
    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)),
        Flatten(),
        Dense(128, activation='relu'),
        Dropout(0.3),
        Dense(10, activation='softmax')
    ])
    model.compile(
        optimizer=Adam(),
        loss='categorical_crossentropy',
        metrics=['accuracy']
    )
    return model

# ------------------------------------------------------------
# 3. TRAINING WITHOUT DATA AUGMENTATION
# ------------------------------------------------------------
```

```
model_no_aug = create_cnn()

history_no_aug = model_no_aug.fit(
    X_train, y_train_cat,
    epochs=10,
    batch_size=64,
    validation_data=(X_test, y_test_cat),
    verbose=0
)

# ------------------------------------------------------------
# 4. DATA AUGMENTATION SETUP
# ------------------------------------------------------------

datagen = ImageDataGenerator(
    rotation_range=15,      # rotation
    width_shift_range=0.1,  # distortion
    height_shift_range=0.1,
    zoom_range=0.1,
    horizontal_flip=True   # horizontal flip
)

datagen.fit(X_train)

# ------------------------------------------------------------
# 5. TRAINING WITH DATA AUGMENTATION
# ------------------------------------------------------------

model_aug = create_cnn()

history_aug = model_aug.fit(
```

```
    datagen.flow(X_train, y_train_cat, batch_size=64),
    epochs=10,
    validation_data=(X_test, y_test_cat),
    verbose=0
)

# ------------------------------------------------------------
# 6. PLOT VALIDATION ACCURACY
# ------------------------------------------------------------

plt.figure(figsize=(10,4))

plt.subplot(1,2,1)
plt.plot(history_no_aug.history['val_accuracy'], label='Without Augmentation')
plt.plot(history_aug.history['val_accuracy'], label='With Augmentation')
plt.xlabel("Epochs")
plt.ylabel("Validation Accuracy")
plt.title("Validation Accuracy Comparison")
plt.legend()

# ------------------------------------------------------------
# 7. PLOT VALIDATION LOSS (ERROR)
# ------------------------------------------------------------

plt.subplot(1,2,2)
plt.plot(history_no_aug.history['val_loss'], label='Without Augmentation')
plt.plot(history_aug.history['val_loss'], label='With Augmentation')
plt.xlabel("Epochs")
plt.ylabel("Validation Loss")
plt.title("Validation Loss Comparison")
plt.legend()
```

```
plt.show()
# ============================================================
# Dataset Given in Folder: With vs Without Data Augmentation
# ============================================================

import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.optimizers import Adam

# ------------------------------------------------------------
# 1. LOAD DATASET FROM FOLDER
# ------------------------------------------------------------

train_ds = tf.keras.utils.image_dataset_from_directory(
    "dataset/",
    image_size=(32, 32),
    batch_size=64,
    validation_split=0.2,
    subset="training",
    seed=42
)

val_ds = tf.keras.utils.image_dataset_from_directory(
    "dataset/",
    image_size=(32, 32),
    batch_size=64,
    validation_split=0.2,
```

```
    subset="validation",
    seed=42
)

num_classes = len(train_ds.class_names)

# Normalize images
normalizer = tf.keras.layers.Rescaling(1./255)
train_ds = train_ds.map(lambda x, y: (normalizer(x), y))
val_ds   = val_ds.map(lambda x, y: (normalizer(x), y))

# ------------------------------------------------------------
# 2. CNN MODEL
# ------------------------------------------------------------

def create_cnn():
    model = Sequential([
        Conv2D(32, (3,3), activation='relu', input_shape=(32,32,3)),
        MaxPooling2D((2,2)),
        Conv2D(64, (3,3), activation='relu'),
        MaxPooling2D((2,2)),
        Flatten(),
        Dense(128, activation='relu'),
        Dropout(0.3),
        Dense(num_classes, activation='softmax')
    ])
    model.compile(
        optimizer=Adam(),
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )
```

```
    return model

# ------------------------------------------------------------
# 3. TRAIN WITHOUT DATA AUGMENTATION
# ------------------------------------------------------------

model_no_aug = create_cnn()

history_no_aug = model_no_aug.fit(
    train_ds,
    validation_data=val_ds,
    epochs=10,
    verbose=0
)

# ------------------------------------------------------------
# 4. DATA AUGMENTATION SETUP
# ------------------------------------------------------------

datagen = ImageDataGenerator(
    rotation_range=15,
    horizontal_flip=True,
    zoom_range=0.1,
    width_shift_range=0.1,
    height_shift_range=0.1
)

# Convert dataset to numpy for augmentation
X_train = []
y_train = []
```

```
for images, labels in train_ds:
    X_train.append(images.numpy())
    y_train.append(labels.numpy())

X_train = tf.concat(X_train, axis=0)
y_train = tf.concat(y_train, axis=0)

# ------------------------------------------------------------
# 5. TRAIN WITH DATA AUGMENTATION
# ------------------------------------------------------------

model_aug = create_cnn()

history_aug = model_aug.fit(
    datagen.flow(X_train, y_train, batch_size=64),
    epochs=10,
    validation_data=val_ds,
    verbose=0
)

# ------------------------------------------------------------
# 6. PLOT VALIDATION ACCURACY & LOSS
# ------------------------------------------------------------

plt.figure(figsize=(10,4))

plt.subplot(1,2,1)
plt.plot(history_no_aug.history['val_accuracy'], label='Without Augmentation')
plt.plot(history_aug.history['val_accuracy'], label='With Augmentation')
plt.xlabel("Epochs")
plt.ylabel("Validation Accuracy")
```

```
plt.title("Accuracy vs Epochs")
plt.legend()

plt.subplot(1,2,2)
plt.plot(history_no_aug.history['val_loss'], label='Without Augmentation')
plt.plot(history_aug.history['val_loss'], label='With Augmentation')
plt.xlabel("Epochs")
plt.ylabel("Validation Loss")
plt.title("Loss vs Epochs")
plt.legend()

plt.show()
```

- **Load California Housing dataset, select 2 features (e.g., Median Income, House Age) and 1 target (Median House Value).**
- **Normalize inputs and initialize a single-layer neural network with random weights and bias.**
- **Perform forward propagation and calculate prediction error, Squared Error, and MSE.**
- **Update weights and bias using gradient descent.**
- **Visualize or analyze how loss changes with weight and bias and observe the error surface.**

```
# ============================================================
# California Housing - Single Layer Neural Network
# ============================================================

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# ------------------------------------------------------------
# 1. LOAD DATASET (GIVEN AS CSV)
# ------------------------------------------------------------
```

```
df = pd.read_csv("california_housing.csv")

# Select 2 features and 1 target
X = df[['MedInc', 'HouseAge']].values
y = df['MedHouseVal'].values

# ------------------------------------------------------------
# 2. NORMALIZE INPUTS
# ------------------------------------------------------------

X = (X - X.mean(axis=0)) / X.std(axis=0)
y = (y - y.mean()) / y.std()

# ------------------------------------------------------------
# 3. INITIALIZE NETWORK (SINGLE LAYER)
# ------------------------------------------------------------

np.random.seed(42)
weights = np.random.randn(2)
bias = np.random.randn()
learning_rate = 0.01
epochs = 100

# ------------------------------------------------------------
# 4. FORWARD PROPAGATION
# ------------------------------------------------------------

def forward(X, w, b):
    return np.dot(X, w) + b
```

```
# ------------------------------------------------------------
# 5. ERROR & MSE
# ------------------------------------------------------------

def mse(y_true, y_pred):
    return np.mean((y_true - y_pred) ** 2)

mse_history = []

# ------------------------------------------------------------
# 6. GRADIENT DESCENT TRAINING
# ------------------------------------------------------------

for _ in range(epochs):
    y_pred = forward(X, weights, bias)

    error = y_pred - y
    loss = mse(y, y_pred)
    mse_history.append(loss)

    dw = (2/len(y)) * np.dot(X.T, error)
    db = (2/len(y)) * np.sum(error)

    weights -= learning_rate * dw
    bias -= learning_rate * db

# ------------------------------------------------------------
# 7. PLOT MSE vs ITERATIONS
# ------------------------------------------------------------

plt.plot(mse_history)
```

```
plt.xlabel("Epochs")
plt.ylabel("Mean Squared Error")
plt.title("MSE vs Epochs")
plt.show()

# ------------------------------------------------------------
# 8. ERROR SURFACE (WEIGHT vs BIAS)
# ------------------------------------------------------------

w_range = np.linspace(-2, 2, 40)
b_range = np.linspace(-2, 2, 40)

W, B = np.meshgrid(w_range, b_range)
Loss = np.zeros(W.shape)

for i in range(W.shape[0]):
    for j in range(W.shape[1]):
        y_hat = X[:,0] * W[i,j] + B[i,j]
        Loss[i,j] = mse(y, y_hat)

# ------------------------------------------------------------
# 9. VISUALIZE ERROR SURFACE
# ------------------------------------------------------------

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(W, B, Loss, cmap='viridis')
ax.set_xlabel("Weight")
ax.set_ylabel("Bias")
ax.set_zlabel("Loss (MSE)")
ax.set_title("Error Surface")
```

plt.show()

- **Load Iris dataset**
- **Normalize inputs and create an autoencoder with 2–3 neurons in the dense layer.**
- **Encode and decode the inputs.**
- **Calculate reconstruction error (MSE) and analyze the results.**

Observe loss progression over epochs and compare with different encoder configurations

```
# ============================================================
# Autoencoder on IRIS Dataset (2 vs 3 Neurons)
# ============================================================

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.optimizers import Adam
from sklearn.datasets import load_iris

# ------------------------------------------------------------
# 1. LOAD IRIS DATASET
# ------------------------------------------------------------

# If dataset is GIVEN as CSV
# df = pd.read_csv("iris.csv")
# X = df.iloc[:, :-1].values

# Standard IRIS dataset
iris = load_iris()
X = iris.data
```

```
# ------------------------------------------------------------
# 2. NORMALIZE INPUT FEATURES
# ------------------------------------------------------------

scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X)

# ------------------------------------------------------------
# 3. AUTOENCODER FUNCTION
# ------------------------------------------------------------

def build_autoencoder(encoding_dim):
    input_dim = X_scaled.shape[1]

    input_layer = Input(shape=(input_dim,))
    encoded = Dense(encoding_dim, activation='relu')(input_layer)
    decoded = Dense(input_dim, activation='sigmoid')(encoded)

    autoencoder = Model(input_layer, decoded)
    autoencoder.compile(
        optimizer=Adam(),
        loss='mse'
    )

    encoder = Model(input_layer, encoded)
    return autoencoder, encoder

# ------------------------------------------------------------
# 4. TRAIN AUTOENCODER WITH 2 NEURONS
# ------------------------------------------------------------
```

```
ae_2, encoder_2 = build_autoencoder(2)

history_2 = ae_2.fit(
    X_scaled, X_scaled,
    epochs=50,
    batch_size=16,
    verbose=0
)

encoded_2 = encoder_2.predict(X_scaled)
recon_2 = ae_2.predict(X_scaled)

mse_2 = np.mean((X_scaled - recon_2) ** 2)

# ------------------------------------------------------------
# 5. TRAIN AUTOENCODER WITH 3 NEURONS
# ------------------------------------------------------------

ae_3, encoder_3 = build_autoencoder(3)

history_3 = ae_3.fit(
    X_scaled, X_scaled,
    epochs=50,
    batch_size=16,
    verbose=0
)

encoded_3 = encoder_3.predict(X_scaled)
recon_3 = ae_3.predict(X_scaled)

mse_3 = np.mean((X_scaled - recon_3) ** 2)
```

```
# ------------------------------------------------------------
# 6. DISPLAY RESULTS
# ------------------------------------------------------------

print("Reconstruction MSE with 2 neurons:", mse_2)
print("Reconstruction MSE with 3 neurons:", mse_3)

print("\nEncoded values (2 neurons):\n", encoded_2[:5])
print("\nEncoded values (3 neurons):\n", encoded_3[:5])

# ------------------------------------------------------------
# 7. PLOT LOSS vs EPOCHS
# ------------------------------------------------------------

plt.plot(history_2.history['loss'], label='2 Neurons')
plt.plot(history_3.history['loss'], label='3 Neurons')
plt.xlabel("Epochs")
plt.ylabel("Reconstruction Loss (MSE)")
plt.title("Loss vs Epochs for Different Encoder Sizes")
plt.legend()
plt.show()
```

- **Load a small image dataset (Intel Image or CIFAR-10 subset).**
- **Preprocess images and split into training and testing sets.**
- **Build and train a CNN model.**
- **Evaluate the model using confusion matrix, precision, recall, and accuracy.**
- **Analyze training vs validation metrics over epochs to check generalization.**

```
# ============================================================
# Small Image Dataset Classification using CNN
# ============================================================
```

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from tensorflow.keras.optimizers import Adam
from sklearn.metrics import confusion_matrix, precision_score, recall_score, accuracy_score
import seaborn as sns

# ------------------------------------------------------------
# 1. LOAD DATASET FROM FOLDERS & SPLIT
# ------------------------------------------------------------

train_ds = tf.keras.utils.image_dataset_from_directory(
    "dataset/",
    image_size=(64, 64),
    batch_size=32,
    validation_split=0.2,
    subset="training",
    seed=42
)

val_ds = tf.keras.utils.image_dataset_from_directory(
    "dataset/",
    image_size=(64, 64),
    batch_size=32,
    validation_split=0.2,
    subset="validation",
    seed=42
)
```

```
class_names = train_ds.class_names
num_classes = len(class_names)

# ------------------------------------------------------------
# 2. PREPROCESSING (NORMALIZATION)
# ------------------------------------------------------------

normalizer = tf.keras.layers.Rescaling(1./255)

train_ds = train_ds.map(lambda x, y: (normalizer(x), y))
val_ds   = val_ds.map(lambda x, y: (normalizer(x), y))

# ------------------------------------------------------------
# 3. BUILD CNN MODEL
# ------------------------------------------------------------

model = Sequential([
    Conv2D(32, (3,3), activation='relu', input_shape=(64,64,3)),
    MaxPooling2D((2,2)),
    Conv2D(64, (3,3), activation='relu'),
    MaxPooling2D((2,2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.3),
    Dense(num_classes, activation='softmax')
])

model.compile(
    optimizer=Adam(),
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
```

```
)

# ------------------------------------------------------------
# 4. TRAIN CNN MODEL
# ------------------------------------------------------------

history = model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=10
)

# ------------------------------------------------------------
# 5. MODEL EVALUATION METRICS
# ------------------------------------------------------------

# Collect true labels and predictions
y_true = np.concatenate([y.numpy() for x, y in val_ds])
y_pred_prob = np.concatenate([model.predict(x) for x, y in val_ds])
y_pred = np.argmax(y_pred_prob, axis=1)

# Metrics
acc = accuracy_score(y_true, y_pred)
precision = precision_score(y_true, y_pred, average='macro')
recall = recall_score(y_true, y_pred, average='macro')

print("Accuracy:", acc)
print("Precision:", precision)
print("Recall:", recall)

# ------------------------------------------------------------
```

```
# 6. CONFUSION MATRIX
# ------------------------------------------------------------

cm = confusion_matrix(y_true, y_pred)

plt.figure(figsize=(6,5))
sns.heatmap(cm, annot=True, fmt='d',
            xticklabels=class_names,
            yticklabels=class_names,
            cmap='Blues')
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix")
plt.show()

# ------------------------------------------------------------
# 7. TRAINING vs VALIDATION METRICS (GENERALIZATION)
# ------------------------------------------------------------

plt.figure(figsize=(10,4))

# Accuracy plot
plt.subplot(1,2,1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.title("Accuracy vs Epochs")
plt.legend()

# Loss plot
```

```
plt.subplot(1,2,2)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Loss vs Epochs")
plt.legend()

plt.show()
```

- **Load MNIST or Fashion-MNIST dataset.**
- **Train a small neural network with different optimizers: SGD, SGD+Momentum, Adam.**
- **Evaluate accuracy and ROC curve for each optimizer.**
- **Compare training loss, validation loss, and overall performance across optimizers.**

```
# ============================================================
# MNIST / Fashion-MNIST - Optimizer Comparison
# SGD vs SGD+Momentum vs Adam
# ============================================================

import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist   # change to fashion_mnist if needed
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.optimizers import SGD, Adam
from tensorflow.keras.utils import to_categorical
from sklearn.preprocessing import label_binarize
from sklearn.metrics import roc_curve, auc

# ------------------------------------------------------------
# 1. LOAD DATASET
# ------------------------------------------------------------
```

```
# For Fashion-MNIST, replace with:
# from tensorflow.keras.datasets import fashion_mnist
# (X_train, y_train), (X_test, y_test) = fashion_mnist.load_data()

(X_train, y_train), (X_test, y_test) = mnist.load_data()

X_train = X_train / 255.0
X_test  = X_test / 255.0

y_train_cat = to_categorical(y_train, 10)
y_test_cat  = to_categorical(y_test, 10)

# For ROC (One-vs-Rest)
y_test_bin = label_binarize(y_test, classes=np.arange(10))

# ------------------------------------------------------------
# 2. MODEL DEFINITION
# ------------------------------------------------------------

def create_model(optimizer):
    model = Sequential([
        Flatten(input_shape=(28,28)),
        Dense(128, activation='relu'),
        Dense(10, activation='softmax')
    ])
    model.compile(
        optimizer=optimizer,
        loss='categorical_crossentropy',
        metrics=['accuracy']
    )
```

```
    return model

# ------------------------------------------------------------
# 3. TRAIN WITH DIFFERENT OPTIMIZERS
# ------------------------------------------------------------

optimizers = {
    "SGD": SGD(learning_rate=0.01),
    "SGD + Momentum": SGD(learning_rate=0.01, momentum=0.9),
    "Adam": Adam()
}

histories = {}
roc_scores = {}
final_accuracy = {}

for name, opt in optimizers.items():
    print(f"Training with {name}")
    model = create_model(opt)

    history = model.fit(
        X_train, y_train_cat,
        epochs=5,
        batch_size=128,
        validation_data=(X_test, y_test_cat),
        verbose=0
    )

    histories[name] = history
    final_accuracy[name] = history.history['val_accuracy'][-1]
```

```
    y_score = model.predict(X_test)
    roc_scores[name] = y_score

# ------------------------------------------------------------
# 4. PLOT TRAINING vs VALIDATION LOSS
# ------------------------------------------------------------

plt.figure(figsize=(10,4))

plt.subplot(1,2,1)
for name, hist in histories.items():
    plt.plot(hist.history['loss'], label=f"{name} Train")
    plt.plot(hist.history['val_loss'], linestyle='--', label=f"{name} Val")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Training & Validation Loss")
plt.legend()

# ------------------------------------------------------------
# 5. PLOT TRAINING vs VALIDATION ACCURACY
# ------------------------------------------------------------

plt.subplot(1,2,2)
for name, hist in histories.items():
    plt.plot(hist.history['accuracy'], label=f"{name} Train")
    plt.plot(hist.history['val_accuracy'], linestyle='--', label=f"{name} Val")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.title("Training & Validation Accuracy")
plt.legend()
```

```
plt.show()

# ------------------------------------------------------------
# 6. ROC CURVE COMPARISON (MACRO / OvR)
# ------------------------------------------------------------

plt.figure(figsize=(8,6))

for name, scores in roc_scores.items():
    fpr, tpr, _ = roc_curve(y_test_bin.ravel(), scores.ravel())
    roc_auc = auc(fpr, tpr)
    plt.plot(fpr, tpr, label=f"{name} (AUC={roc_auc:.3f})")

plt.plot([0,1], [0,1], 'k--')
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve Comparison – MNIST")
plt.legend()
plt.show()

# ------------------------------------------------------------
# 7. PRINT FINAL ACCURACY COMPARISON
# ------------------------------------------------------------

print("\nFinal Validation Accuracy:")
for name, acc in final_accuracy.items():
    print(f"{name}: {acc:.4f}")
```

//opt 2

```
# ============================================================
# MNIST / Fashion-MNIST (Dataset Given) - Optimizer Comparison
# SGD vs SGD+Momentum vs Adam with ROC Curve
```

```
# ============================================================

import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.optimizers import SGD, Adam
from sklearn.preprocessing import label_binarize
from sklearn.metrics import roc_curve, auc

# ------------------------------------------------------------
# 1. LOAD DATASET FROM GIVEN FOLDER
# ------------------------------------------------------------

train_ds = tf.keras.utils.image_dataset_from_directory(
    "dataset/",
    image_size=(28, 28),
    color_mode="grayscale",
    batch_size=128,
    validation_split=0.2,
    subset="training",
    seed=42
)

val_ds = tf.keras.utils.image_dataset_from_directory(
    "dataset/",
    image_size=(28, 28),
    color_mode="grayscale",
    batch_size=128,
    validation_split=0.2,
```

```
    subset="validation",
    seed=42
)

# Normalize
normalizer = tf.keras.layers.Rescaling(1./255)
train_ds = train_ds.map(lambda x, y: (normalizer(x), y))
val_ds   = val_ds.map(lambda x, y: (normalizer(x), y))

# Extract validation labels for ROC
y_true = np.concatenate([y.numpy() for x, y in val_ds])
y_true_bin = label_binarize(y_true, classes=np.arange(10))

# ------------------------------------------------------------
# 2. MODEL DEFINITION
# ------------------------------------------------------------

def create_model(optimizer):
    model = Sequential([
        Flatten(input_shape=(28,28,1)),
        Dense(128, activation='relu'),
        Dense(10, activation='softmax')
    ])
    model.compile(
        optimizer=optimizer,
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )
    return model

# ------------------------------------------------------------
```

```
# 3. TRAIN WITH DIFFERENT OPTIMIZERS
# ------------------------------------------------------------

optimizers = {
    "SGD": SGD(learning_rate=0.01),
    "SGD + Momentum": SGD(learning_rate=0.01, momentum=0.9),
    "Adam": Adam()
}

histories = {}
roc_scores = {}

for name, opt in optimizers.items():
    print(f"Training with {name}")
    model = create_model(opt)

    history = model.fit(
        train_ds,
        validation_data=val_ds,
        epochs=5,
        verbose=0
    )

    histories[name] = history

    # Predictions for ROC
    y_scores = np.vstack([model.predict(x) for x, y in val_ds])
    roc_scores[name] = y_scores

# ------------------------------------------------------------
# 4. PLOT TRAINING vs VALIDATION LOSS
```

```
# ------------------------------------------------------------

plt.figure(figsize=(10,4))

plt.subplot(1,2,1)
for name, hist in histories.items():
    plt.plot(hist.history['loss'], label=f"{name} Train")
    plt.plot(hist.history['val_loss'], linestyle='--', label=f"{name} Val")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Training & Validation Loss")
plt.legend()

# ------------------------------------------------------------
# 5. PLOT TRAINING vs VALIDATION ACCURACY
# ------------------------------------------------------------

plt.subplot(1,2,2)
for name, hist in histories.items():
    plt.plot(hist.history['accuracy'], label=f"{name} Train")
    plt.plot(hist.history['val_accuracy'], linestyle='--', label=f"{name} Val")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.title("Training & Validation Accuracy")
plt.legend()

plt.show()

# ------------------------------------------------------------
# 6. ROC CURVE COMPARISON
# ------------------------------------------------------------
```

```
plt.figure(figsize=(8,6))


for name, scores in roc_scores.items():
    fpr, tpr, _ = roc_curve(y_true_bin.ravel(), scores.ravel())
    roc_auc = auc(fpr, tpr)
    plt.plot(fpr, tpr, label=f"{name} (AUC={roc_auc:.3f})")


plt.plot([0,1], [0,1], 'k--')
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve Comparison (Dataset Given)")
plt.legend()
plt.show()
```

- **Load California Housing dataset and select 2 features (e.g., Median Income, House Age) and 1 target (Median House Value).**
- **Normalize inputs and initialize a single-layer neural network with random weights and bias.**
- **Perform forward propagation and calculate prediction error, Squared Error, and MSE.**
- **Update weights and bias using gradient descent.**
- **Visualize or analyze how loss changes with weight and bias and observe the error surface.**

```
# ============================================================
# California Housing - Single Layer Neural Network
# ============================================================


import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D


# ------------------------------------------------------------
```

```
# 1. LOAD DATASET (DATASET IS GIVEN AS CSV)
# ------------------------------------------------------------

df = pd.read_csv("california_housing.csv")

# Select 2 features and 1 target
X = df[['MedInc', 'HouseAge']].values      # Inputs
y = df['MedHouseVal'].values               # Target

# ------------------------------------------------------------
# 2. NORMALIZE INPUTS
# ------------------------------------------------------------

X = (X - X.mean(axis=0)) / X.std(axis=0)
y = (y - y.mean()) / y.std()

# ------------------------------------------------------------
# 3. INITIALIZE SINGLE-LAYER NETWORK
# ------------------------------------------------------------

np.random.seed(42)
weights = np.random.randn(2)   # w1, w2
bias = np.random.randn()       # b

learning_rate = 0.01
epochs = 100

# ------------------------------------------------------------
# 4. FORWARD PROPAGATION
# ------------------------------------------------------------
```

```
def forward(X, w, b):
    return np.dot(X, w) + b

# ------------------------------------------------------------
# 5. ERROR & MSE
# ------------------------------------------------------------

def mse(y_true, y_pred):
    return np.mean((y_true - y_pred) ** 2)

mse_history = []

# ------------------------------------------------------------
# 6. GRADIENT DESCENT TRAINING
# ------------------------------------------------------------

for _ in range(epochs):
    y_pred = forward(X, weights, bias)

    error = y_pred - y                # Prediction error
    squared_error = error ** 2
    loss = mse(y, y_pred)             # Mean Squared Error
    mse_history.append(loss)

    # Gradients
    dw = (2 / len(y)) * np.dot(X.T, error)
    db = (2 / len(y)) * np.sum(error)

    # Update weights and bias
    weights -= learning_rate * dw
    bias -= learning_rate * db
```

```
# ------------------------------------------------------------
# 7. PLOT MSE vs EPOCHS
# ------------------------------------------------------------

plt.plot(mse_history)
plt.xlabel("Epochs")
plt.ylabel("Mean Squared Error")
plt.title("MSE vs Epochs")
plt.show()

# ------------------------------------------------------------
# 8. ERROR SURFACE (WEIGHT vs BIAS)
# ------------------------------------------------------------

w_range = np.linspace(-2, 2, 40)
b_range = np.linspace(-2, 2, 40)

W, B = np.meshgrid(w_range, b_range)
Loss = np.zeros(W.shape)

for i in range(W.shape[0]):
    for j in range(W.shape[1]):
        y_hat = X[:, 0] * W[i, j] + B[i, j]
        Loss[i, j] = mse(y, y_hat)

# ------------------------------------------------------------
# 9. VISUALIZE ERROR SURFACE
# ------------------------------------------------------------

fig = plt.figure()
```

```
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(W, B, Loss, cmap='viridis')
ax.set_xlabel("Weight")
ax.set_ylabel("Bias")
ax.set_zlabel("Loss (MSE)")
ax.set_title("Error Surface (Weight vs Bias)")
plt.show()
```

Task 1: Implement a small CNN for CIFAR-10 subset

- **Preprocess & normalize images**
- **Convolution + Pooling + Flatten + Dense layers**
- **Metrics / Plots: Accuracy, Confusion Matrix, Precision & Recall, plot training vs validation accuracy/loss**

Task 2: Apply data augmentation (flip, rotate, noise)

- **Compare performance with original data**
- **Metrics / Plots: Plot accuracy vs epochs and loss vs epochs**

```
# ============================================================
# TASK 1 & 2: CIFAR-10 Subset CNN (Dataset Given)
# ============================================================

import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.optimizers import Adam
from sklearn.metrics import confusion_matrix, precision_score, recall_score, accuracy_score
import seaborn as sns

# ------------------------------------------------------------
# 1. LOAD DATASET FROM GIVEN FOLDER
```

```
# ------------------------------------------------------------

train_ds = tf.keras.utils.image_dataset_from_directory(
    "dataset/",
    image_size=(32, 32),
    batch_size=32,
    validation_split=0.2,
    subset="training",
    seed=42
)

val_ds = tf.keras.utils.image_dataset_from_directory(
    "dataset/",
    image_size=(32, 32),
    batch_size=32,
    validation_split=0.2,
    subset="validation",
    seed=42
)

class_names = train_ds.class_names
num_classes = len(class_names)

# ------------------------------------------------------------
# 2. PREPROCESSING (NORMALIZATION)
# ------------------------------------------------------------

normalizer = tf.keras.layers.Rescaling(1./255)
train_ds = train_ds.map(lambda x, y: (normalizer(x), y))
val_ds   = val_ds.map(lambda x, y: (normalizer(x), y))
```

```
# ------------------------------------------------------------
# 3. CNN MODEL DEFINITION
# ------------------------------------------------------------

def create_cnn():
    model = Sequential([
        Conv2D(32, (3,3), activation='relu', input_shape=(32,32,3)),
        MaxPooling2D((2,2)),
        Conv2D(64, (3,3), activation='relu'),
        MaxPooling2D((2,2)),
        Flatten(),
        Dense(128, activation='relu'),
        Dropout(0.3),
        Dense(num_classes, activation='softmax')
    ])

    model.compile(
        optimizer=Adam(),
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )
    return model

# ------------------------------------------------------------
# 4. TRAIN MODEL (WITHOUT AUGMENTATION)
# ------------------------------------------------------------

model_no_aug = create_cnn()

history_no_aug = model_no_aug.fit(
    train_ds,
```

```
    validation_data=val_ds,
    epochs=10
)

# ------------------------------------------------------------
# 5. EVALUATION METRICS (TASK 1)
# ------------------------------------------------------------

y_true = np.concatenate([y.numpy() for x, y in val_ds])
y_pred_prob = np.concatenate([model_no_aug.predict(x) for x, y in val_ds])
y_pred = np.argmax(y_pred_prob, axis=1)

accuracy = accuracy_score(y_true, y_pred)
precision = precision_score(y_true, y_pred, average='macro')
recall = recall_score(y_true, y_pred, average='macro')

print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)

# Confusion Matrix
cm = confusion_matrix(y_true, y_pred)

plt.figure(figsize=(6,5))
sns.heatmap(cm, annot=True, fmt='d',
            xticklabels=class_names,
            yticklabels=class_names,
            cmap='Blues')
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix")
```

```
plt.show()

# ------------------------------------------------------------
# 6. TRAINING vs VALIDATION PLOTS (TASK 1)
# ------------------------------------------------------------

plt.figure(figsize=(10,4))

plt.subplot(1,2,1)
plt.plot(history_no_aug.history['accuracy'], label='Train Accuracy')
plt.plot(history_no_aug.history['val_accuracy'], label='Val Accuracy')
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.title("Accuracy vs Epochs (No Augmentation)")
plt.legend()

plt.subplot(1,2,2)
plt.plot(history_no_aug.history['loss'], label='Train Loss')
plt.plot(history_no_aug.history['val_loss'], label='Val Loss')
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Loss vs Epochs (No Augmentation)")
plt.legend()

plt.show()
# ------------------------------------------------------------
# 7. DATA AUGMENTATION SETUP
# ------------------------------------------------------------

datagen = ImageDataGenerator(
    rotation_range=15,
```

```
    horizontal_flip=True,
    width_shift_range=0.1,
    height_shift_range=0.1,
    zoom_range=0.1
)

# Convert training dataset to NumPy (required for ImageDataGenerator)
X_train = []
y_train = []

for images, labels in train_ds:
    X_train.append(images.numpy())
    y_train.append(labels.numpy())

X_train = np.concatenate(X_train, axis=0)
y_train = np.concatenate(y_train, axis=0)

# ------------------------------------------------------------
# 8. TRAIN MODEL WITH DATA AUGMENTATION
# ------------------------------------------------------------

model_aug = create_cnn()

history_aug = model_aug.fit(
    datagen.flow(X_train, y_train, batch_size=32),
    epochs=10,
    validation_data=val_ds
)

# ------------------------------------------------------------
# 9. COMPARE ACCURACY & LOSS (TASK 2)
```

```
# ------------------------------------------------------------


plt.figure(figsize=(10,4))

plt.subplot(1,2,1)
plt.plot(history_no_aug.history['val_accuracy'], label='No Augmentation')
plt.plot(history_aug.history['val_accuracy'], label='With Augmentation')
plt.xlabel("Epochs")
plt.ylabel("Validation Accuracy")
plt.title("Accuracy Comparison")
plt.legend()

plt.subplot(1,2,2)
plt.plot(history_no_aug.history['val_loss'], label='No Augmentation')
plt.plot(history_aug.history['val_loss'], label='With Augmentation')
plt.xlabel("Epochs")
plt.ylabel("Validation Loss")
plt.title("Loss Comparison")
plt.legend()

plt.show()

train_ds = tf.keras.utils.image_dataset_from_directory(
    r"C:\Users\Jenith\Desktop\dataset",
    image_size=(32, 32),
    batch_size=32,
    validation_split=0.2,
    subset="training",
    seed=42
)
```