**Practical No 1**

import numpy as np

import matplotlib.pyplot as plt

# Define activation functions

def linear(x):

    return x

def sigmoid(x):

    return 1 / (1 + np.exp(-x))

def tanh(x):

    return np.tanh(x)

def relu(x):

    return np.maximum(0, x)

# Generate input values

x = np.linspace(-5, 5, 400)

# Compute activation function values

y\_functions = {

    "Linear": linear(x),

    "Sigmoid": sigmoid(x),

    "Tanh": tanh(x),

    "ReLU": relu(x)

}

# Colors for plots

colors = ["blue", "red", "green", "purple"]

# Create subplots

plt.figure(figsize=(10, 6))

for i, (name, y) in enumerate(y\_functions.items(), start=1):

    plt.subplot(2, 2, i)  # Arrange in 2 rows, 2 columns

    plt.plot(x, y, label=name, color=colors[i-1], linewidth=2)

    plt.title(name, fontsize=12, fontweight="bold")

    plt.axhline(y=0, color='black', linestyle='dashed', linewidth=0.7)

    plt.axvline(x=0, color='black', linestyle='dashed', linewidth=0.7)

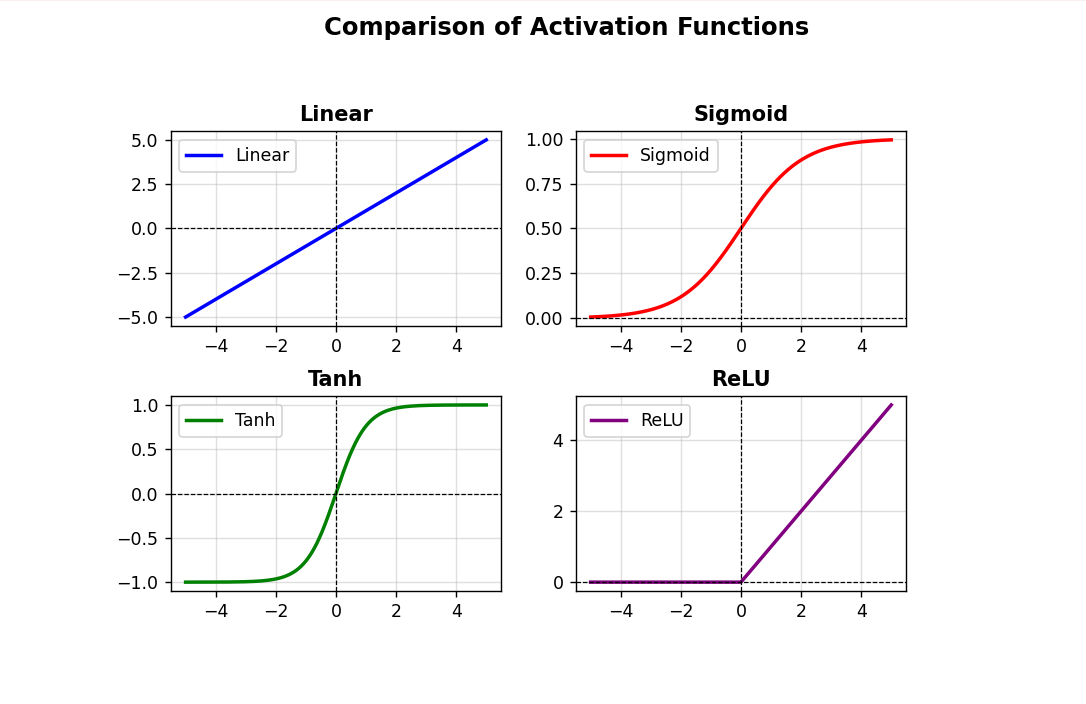
    plt.grid(alpha=0.4)

    plt.legend()

plt.suptitle("Comparison of Activation Functions", fontsize=14, fontweight="bold")

plt.tight\_layout(rect=[0, 0, 1, 0.95])  # Adjust layout

plt.show()

****

**Practical No 2**

% McCulloch-Pitts Neural Network for ANDNOT Function

% Define input vectors (x1, x2)

inputs = [0 0; 0 1; 1 0; 1 1]; % [x1, x2]

% Define weights and threshold

w = [1 -1]; % Weights for x1 and NOT x2

theta = 1;  % Threshold

% Compute net input and output

net\_input = inputs \* w';

outputs = net\_input >= theta; % Apply step activation function

% Display results

disp('Weights of Neuron:');

disp(['w1 = ' num2str(w(1))]);

disp(['w2 = ' num2str(w(2))]);

disp(['Threshold: Theta = ' num2str(theta)]);

disp(' ');

disp('Output:');

disp(['w1 = ' num2str(w(1))]);

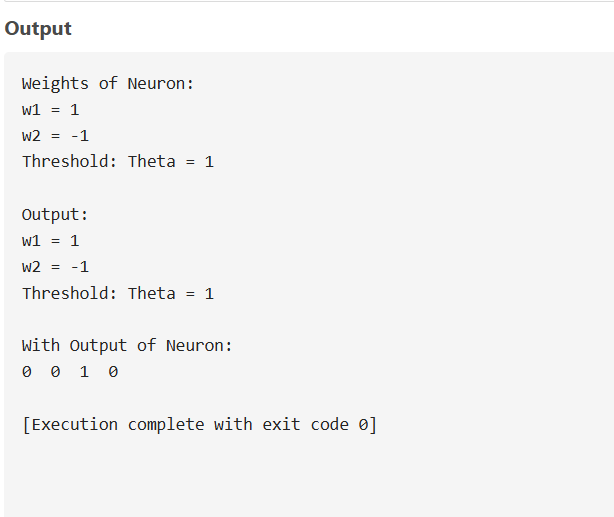
disp(['w2 = ' num2str(w(2))]);

disp(['Threshold: Theta = ' num2str(theta)]);

disp(' ');

disp('With Output of Neuron:');

disp(num2str(outputs'));



**Practical No 3**

import tkinter as tk

from tkinter import ttk

# Function to classify even (1) and odd (0)

def classify\_number(n):

    return 1 if n % 2 == 0 else 0  # Even -> 1, Odd -> 0

# Create main window

root = tk.Tk()

root.title("Perceptron: Even/Odd Classification")

root.geometry("400x350")

root.configure(bg="#2C3E50")  # Dark background

# Style for the Treeview table

style = ttk.Style()

style.theme\_use("clam")  # Use a clean style

style.configure("Treeview",

                background="#ECF0F1",

                foreground="black",

                rowheight=30,

                fieldbackground="#ECF0F1")

style.configure("Treeview.Heading",

                font=("Arial", 12, "bold"),

                background="#2980B9",

                foreground="white")

# Create table using Treeview

columns = ("Number", "ASCII Value", "Even (1) / Odd (0)")

tree = ttk.Treeview(root, columns=columns, show="headings", height=10)

tree.heading("Number", text="Number")

tree.heading("ASCII Value", text="ASCII Value")

tree.heading("Even (1) / Odd (0)", text="Even (1) / Odd (0)")

# Center columns and set width

for col in columns:

    tree.column(col, anchor="center", width=120)

# Insert data into the table

for num in range(10):  # ASCII digits 0-9

    ascii\_val = ord(str(num))

    even\_odd = classify\_number(num)

    tree.insert("", "end", values=(num, ascii\_val, even\_odd))

tree.pack(pady=20, padx=20)

# Styled Exit button

exit\_button = tk.Button(root, text="Exit", command=root.quit,

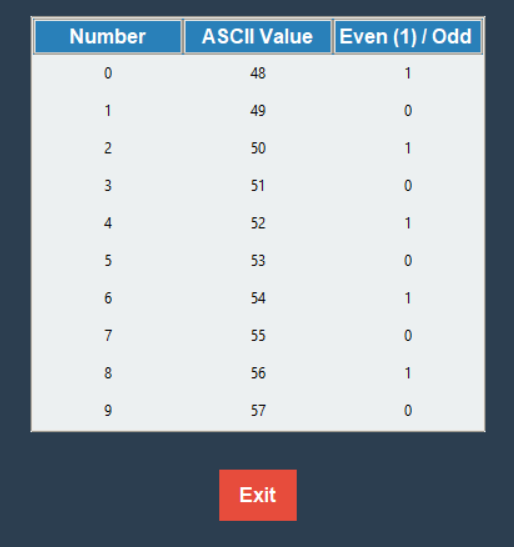
                        bg="#E74C3C", fg="white", font=("Arial", 12, "bold"),

                        padx=10, pady=5, relief="flat", activebackground="#C0392B")

exit\_button.pack(pady=10)

# Run application

root.mainloop()



**Practical No 4**

import numpy as np

import matplotlib.pyplot as plt

from sklearn.linear\_model import Perceptron

from mlxtend.plotting import plot\_decision\_regions

# Generate a simple dataset (AND logic gate)

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

y = np.array([0, 0, 0, 1])  # Labels (AND logic gate)

# Train a perceptron model

model = Perceptron(max\_iter=1000, tol=1e-3, random\_state=42)

model.fit(X, y)

# Plot decision regions

plt.figure(figsize=(6, 4))

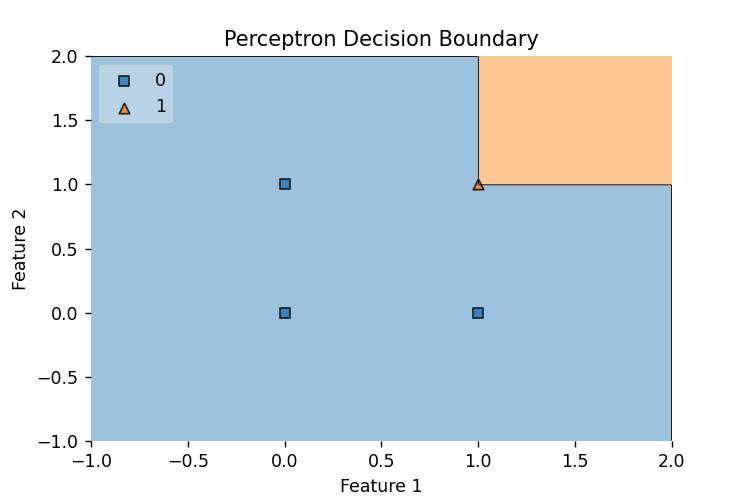
plot\_decision\_regions(X, y, clf=model, legend=2)

plt.xlabel('Feature 1')

plt.ylabel('Feature 2')

plt.title('Perceptron Decision Boundary')

plt.show()

****

**Practical No 5**

import numpy as np

def bipolar\_to\_binary(vector):

    return np.where(vector == -1, 0, 1)

def binary\_to\_bipolar(vector):

    return np.where(vector == 0, -1, 1)

def train\_bam(X, Y):

    """ Train BAM by computing the weight matrix """

    W = np.zeros((X.shape[1], Y.shape[1]))

    for i in range(X.shape[0]):

        W += np.outer(X[i], Y[i])

    return W

def recall\_bam(X, W, mode="forward"):

    """ Recall associated patterns """

    if mode == "forward":

        return np.sign(X @ W)

    elif mode == "backward":

        return np.sign(X @ W.T)

    else:

        raise ValueError("Invalid mode. Use 'forward' or 'backward'.")

# Define new bipolar input-output pairs

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

Y = np.array([[-1, 1, -1], [1, -1, 1]])  # New Associated output patterns

# Train BAM

W = train\_bam(X, Y)

# Recall from input to output (Forward direction)

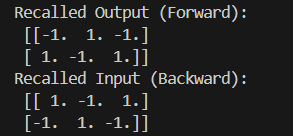
Y\_recalled = recall\_bam(X, W, mode="forward")

print("Recalled Output (Forward):\n", Y\_recalled)

# Recall from output to input (Backward direction)

X\_recalled = recall\_bam(Y, W, mode="backward")

print("Recalled Input (Backward):\n", X\_recalled)



**Practical No 6**

import numpy as np

# Sigmoid activation function and its derivative

def sigmoid(x):

    return 1 / (1 + np.exp(-x))

def sigmoid\_derivative(x):

    return x \* (1 - x)

# Sample dataset: XOR problem

X = np.array([[0, 0],

              [0, 1],

              [1, 0],

              [1, 1]])

y = np.array([[0],

              [1],

              [1],

              [0]])

# Seed for reproducibility

np.random.seed(1)

# Network architecture

input\_neurons = 2

hidden\_neurons = 3

output\_neurons = 1

# Weight and bias initialization

W1 = np.random.uniform(size=(input\_neurons, hidden\_neurons))

b1 = np.random.uniform(size=(1, hidden\_neurons))

W2 = np.random.uniform(size=(hidden\_neurons, output\_neurons))

b2 = np.random.uniform(size=(1, output\_neurons))

# Training parameters

epochs = 10000

lr = 0.1

# Training loop

for epoch in range(epochs):

    # Forward Propagation

    z1 = np.dot(X, W1) + b1

    a1 = sigmoid(z1)

    z2 = np.dot(a1, W2) + b2

    a2 = sigmoid(z2)

    # Loss computation

    loss = y - a2

    # Backpropagation

    d\_a2 = loss \* sigmoid\_derivative(a2)

    dW2 = np.dot(a1.T, d\_a2)

    db2 = np.sum(d\_a2, axis=0, keepdims=True)

    d\_a1 = np.dot(d\_a2, W2.T) \* sigmoid\_derivative(a1)

    dW1 = np.dot(X.T, d\_a1)

    db1 = np.sum(d\_a1, axis=0, keepdims=True)

    # Update weights and biases

    W2 += lr \* dW2

    b2 += lr \* db2

    W1 += lr \* dW1

    b1 += lr \* db1

    # Print the error every 1000 epochs

    if epoch % 1000 == 0:

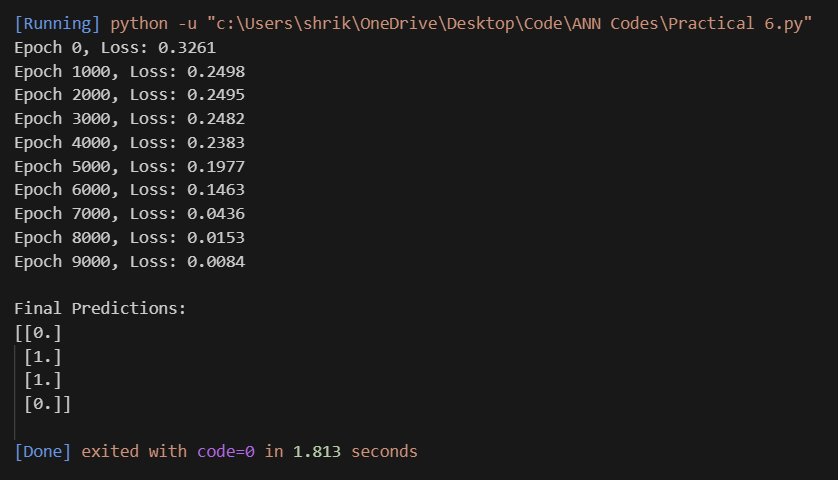
        error = np.mean(np.square(loss))

        print(f"Epoch {epoch}, Loss: {error:.4f}")

# Final output

print("\nFinal Predictions:")

print(np.round(a2))



**Practical No 7**

import numpy as np

import tkinter as tk

# Activation and its derivative

def sigmoid(x):

    return 1 / (1 + np.exp(-x))

def sigmoid\_deriv(x):

    return x \* (1 - x)

# Training the XOR model

def train\_model():

    global wh, bh, wo, bo

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

    y = np.array([[0],[1],[1],[0]])

    wh = np.random.uniform(size=(2, 2))

    bh = np.random.uniform(size=(1, 2))

    wo = np.random.uniform(size=(2, 1))

    bo = np.random.uniform(size=(1, 1))

    for \_ in range(10000):

        h\_input = np.dot(X, wh) + bh

        h\_output = sigmoid(h\_input)

        o\_input = np.dot(h\_output, wo) + bo

        output = sigmoid(o\_input)

        error = y - output

        d\_output = error \* sigmoid\_deriv(output)

        error\_hidden = d\_output.dot(wo.T)

        d\_hidden = error\_hidden \* sigmoid\_deriv(h\_output)

        wo += h\_output.T.dot(d\_output) \* 0.1

        bo += np.sum(d\_output, axis=0, keepdims=True) \* 0.1

        wh += X.T.dot(d\_hidden) \* 0.1

        bh += np.sum(d\_hidden, axis=0, keepdims=True) \* 0.1

# Predict XOR output for input

def predict():

    i1 = int(entry1.get())

    i2 = int(entry2.get())

    x = np.array([[i1, i2]])

    h = sigmoid(np.dot(x, wh) + bh)

    o = sigmoid(np.dot(h, wo) + bo)

    output\_label.config(text="Output: " + str(round(o[0][0])))

# GUI setup

root = tk.Tk()

root.title("XOR BPN")

root.geometry("250x200")

tk.Label(root, text="Input 1 (0/1):").pack()

entry1 = tk.Entry(root)

entry1.pack()

tk.Label(root, text="Input 2 (0/1):").pack()

entry2 = tk.Entry(root)

entry2.pack()

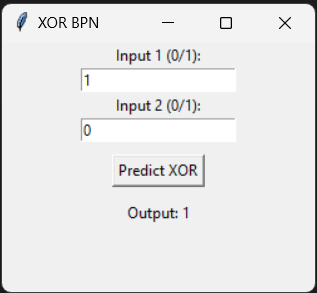
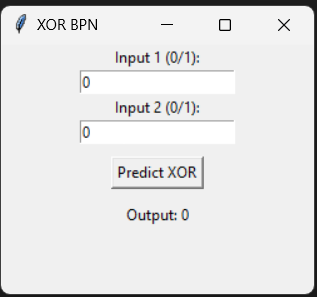
tk.Button(root, text="Predict XOR", command=predict).pack(pady=10)

output\_label = tk.Label(root, text="Output: ")

output\_label.pack()

train\_model()  # Train once at start

root.mainloop()

**Practical No 8**

import numpy as np

# Sigmoid activation and derivative

def sigmoid(x):

    return 1 / (1 + np.exp(-x))

def sigmoid\_derivative(x):

    return x \* (1 - x)

# XOR Inputs and Targets

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

T = np.array([[0], [1], [1], [0]])           # Targets

# Initialize weights and biases (Step 1)

np.random.seed(42)

wh = np.random.uniform(-1, 1, (2, 2))   # Input to Hidden

bh = np.random.uniform(-1, 1, (1, 2))   # Bias of hidden

wo = np.random.uniform(-1, 1, (2, 1))   # Hidden to Output

bo = np.random.uniform(-1, 1, (1, 1))   # Bias of output

alpha = 0.1  # Learning rate

epochs = 10000

# Training loop (Step 2 to 10)

for epoch in range(epochs):

    # Step 4-5: Feedforward

    hin = np.dot(X, wh) + bh

    hout = sigmoid(hin)

    oin = np.dot(hout, wo) + bo

    y = sigmoid(oin)

    # Step 6: Output layer error

    error\_output = (T - y) \* sigmoid\_derivative(y)   # δk

    # Step 7: Hidden layer error

    error\_hidden = error\_output.dot(wo.T) \* sigmoid\_derivative(hout)  # δj

    # Step 8: Update weights and biases

    wo += hout.T.dot(error\_output) \* alpha

    bo += np.sum(error\_output, axis=0, keepdims=True) \* alpha

    wh += X.T.dot(error\_hidden) \* alpha

    bh += np.sum(error\_hidden, axis=0, keepdims=True) \* alpha

# Final Predictions

print("Final predictions after training:")

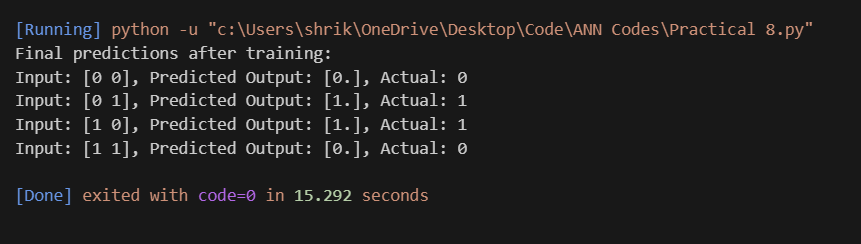
for i in range(4):

    test\_input = X[i]

    hidden = sigmoid(np.dot(test\_input, wh) + bh)

    output = sigmoid(np.dot(hidden, wo) + bo)

    print(f"Input: {test\_input}, Predicted Output: {np.round(output[0])}, Actual: {T[i][0]}")



**Practical No 9**

import numpy as np

def train\_hopfield\_network(patterns):

    # Number of patterns and number of neurons

    num\_patterns = len(patterns)

    num\_neurons = patterns[0].shape[0]

    # Initialize the weight matrix

    W = np.zeros((num\_neurons, num\_neurons))

    # Iterate over all pattern pairs

    for p in range(num\_patterns):

        for i in range(num\_neurons):

            for j in range(num\_neurons):

                if i != j:

                    # Bipolar Hebbian learning rule

                    W[i, j] += patterns[p][i] \* patterns[p][j]

    return W

def recall\_hopfield\_network(W, initial\_state, max\_iterations=100):

    num\_neurons = W.shape[0]

    state = initial\_state.copy()  # Create a copy to avoid modifying the original

    for iteration in range(max\_iterations):

        # Asynchronous update: update neurons one at a time in random order

        neuron\_order = np.random.permutation(num\_neurons)

        state\_changed = False # Flag to check if any state changed in this iteration

        for i in neuron\_order:

            # Calculate the net input to neuron i

            net\_input = np.dot(W[i, :], state) # No need to add state[i]\*W[i,i] since W[i,i] is 0

            # Update the neuron's state using the activation function (sign function)

            new\_state\_i = 1 if net\_input > 0 else -1

            if new\_state\_i != state[i]:

                state[i] = new\_state\_i

                state\_changed = True # Set the flag to True if state changed

        if not state\_changed:

            break # If no state changed, the network has converged

    return state

def test\_hopfield\_network():

    # Define the 4 patterns to be stored

    pattern1 = np.array([1, -1, 1, -1, 1, -1, 1, -1])

    pattern2 = np.array([-1, 1, -1, 1, -1, 1, -1, 1])

    pattern3 = np.array([1, 1, -1, -1, 1, 1, -1, -1])

    pattern4 = np.array([-1, -1, 1, 1, -1, -1, 1, 1])

    patterns = [pattern1, pattern2, pattern3, pattern4]

    # Train the Hopfield network

    W = train\_hopfield\_network(patterns)

    print("Weight Matrix W:\n", W)

    # Test the network with a noisy version of pattern1

    noisy\_pattern1 = np.array([1, -1, -1, -1, 1, -1, 1, -1]) # Introduce one error.

    print("\nOriginal Pattern 1:", pattern1)

    print("Noisy Pattern 1:  ", noisy\_pattern1)

    # Recall the stored pattern from the noisy input

    recalled\_pattern = recall\_hopfield\_network(W, noisy\_pattern1)

    print("Recalled Pattern 1:", recalled\_pattern)

    # Check if the network successfully recalled the original pattern

    if np.array\_equal(recalled\_pattern, pattern1):

        print("Test Passed: Network successfully recalled Pattern 1")

    else:

        print("Test Failed: Network failed to recall Pattern 1")

    # Test with a pattern that is not very close to any of the stored patterns

    test\_pattern = np.array([1, 1, 1, 1, -1, -1, -1, -1])

    print("\nTest Pattern:", test\_pattern)

    recalled\_pattern = recall\_hopfield\_network(W, test\_pattern)

    print("Recalled Pattern:", recalled\_pattern)

    print("This pattern should converge to one of the stored patterns,")

    print("although it might take more iterations.  It is not guaranteed to converge")

    print("to the closest pattern.")

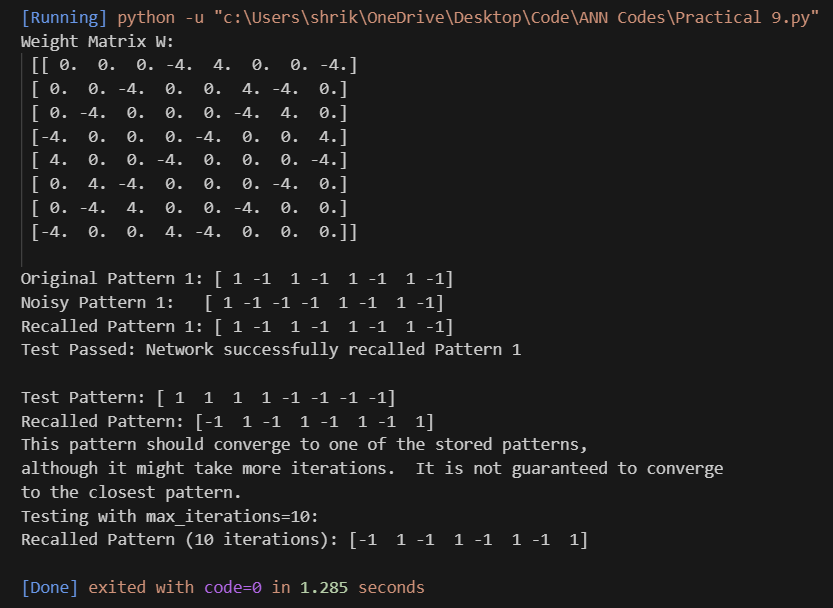
    print("Testing with max\_iterations=10:")

    recalled\_pattern\_short = recall\_hopfield\_network(W, test\_pattern, max\_iterations=10)

    print("Recalled Pattern (10 iterations):", recalled\_pattern\_short)

if \_\_name\_\_ == "\_\_main\_\_":

    test\_hopfield\_network()



**Practical No 10**

import tensorflow as tf

from tensorflow import keras

from tensorflow.keras.models import Model

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

import numpy as np

import matplotlib.pyplot as plt

import cv2 # OpenCV for image manipulation

# Build Object Detection Model

def build\_object\_detection\_model(input\_shape, num\_classes):

"""

Builds a CNN-based object detection model. This is a \*simplified\*

architecture, and in practice, you'd use a more complex one like YOLO, SSD, or Faster R-CNN.

This example assumes a single object detection and classification task.

Args:

input\_shape: Shape of the input image (e.g., (224, 224, 3)).

num\_classes: Number of object classes to detect.

Returns:

tf.keras.Model: A Keras model.

"""

# Input layer

input\_img = Input(shape=input\_shape)

# Convolutional layers

x = Conv2D(32, (3, 3), activation='relu', padding='same')(input\_img)

x = MaxPooling2D((2, 2))(x)

x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)

x = MaxPooling2D((2, 2))(x)

x = Conv2D(128, (3, 3), activation='relu', padding='same')(x)

x = MaxPooling2D((2, 2))(x)

# Flatten the features

x = Flatten()(x)

# Fully connected layers for classification and bounding box regression

# Classification branch

cls\_branch = Dense(256, activation='relu')(x)

cls\_branch = Dropout(0.5)(cls\_branch)

cls\_output = Dense(num\_classes, activation='softmax', name='class\_output')(cls\_branch)

# Bounding box regression branch

bbox\_branch = Dense(256, activation='relu')(x)

bbox\_branch = Dropout(0.5)(bbox\_branch)

bbox\_output = Dense(4, activation='linear', name='box\_output')(bbox\_branch) # 4 outputs: (x, y, width, height)

# Define the model with two outputs: class and bounding box

model = Model(inputs=input\_img, outputs=[cls\_output, bbox\_output])

return model

# Training the object detection model

def train\_object\_detector(model, train\_data, val\_data, epochs, batch\_size):

"""

Trains the object detection model.

Args:

model: A Keras model.

train\_data: Tuple of (images, [class\_labels, bounding\_boxes])

val\_data: Tuple of (images, [class\_labels, bounding\_boxes])

epochs: Number of training epochs.

batch\_size: Batch size for training.

"""

model.compile(optimizer='adam',

loss={'class\_output': 'categorical\_crossentropy', 'box\_output': 'mse'}, # Loss for both tasks

metrics={'class\_output': 'accuracy'})

# Prepare data for training

train\_images, [train\_class\_labels, train\_bbox\_labels] = train\_data

val\_images, [val\_class\_labels, val\_bbox\_labels] = val\_data

# Train the model

model.fit(train\_images,

{'class\_output': train\_class\_labels, 'box\_output': train\_bbox\_labels},

validation\_data=(val\_images, {'class\_output': val\_class\_labels, 'box\_output': val\_bbox\_labels}),

epochs=epochs,

batch\_size=batch\_size)

return model # Return the trained model

# Evaluate the object detection model

def evaluate\_object\_detector(model, test\_data):

"""

Evaluates the object detector on a test dataset.

Args:

model: A trained Keras model.

test\_data: Tuple of (images, [class\_labels, bounding\_boxes])

Returns:

dict: A dictionary of evaluation metrics.

"""

test\_images, [test\_class\_labels, test\_bbox\_labels] = test\_data

results = model.evaluate(test\_images,

{'class\_output': test\_class\_labels, 'box\_output': test\_bbox\_labels},

verbose=0)

return {'class\_accuracy': results[1], 'box\_loss': results[2]}

# Visualize the predictions

def visualize\_prediction(image, predicted\_class, predicted\_bounding\_box, class\_names):

"""

Visualizes the prediction on the image.

Args:

image: The original image (numpy array).

predicted\_class: The predicted class index.

predicted\_bounding\_box: The predicted bounding box coordinates (x, y, width, height).

class\_names: A list of class names.

"""

image = np.array(image \* 255, dtype=np.uint8) # Convert to uint8 for OpenCV compatibility

image = cv2.cvtColor(image, cv2.COLOR\_RGB2BGR)

class\_name = class\_names[predicted\_class]

x, y, w, h = predicted\_bounding\_box

x, y, w, h = int(x), int(y), int(w), int(h) # Convert to integers

cv2.rectangle(image, (x, y), (x + w, y + h), (0, 255, 0), 2) # Green rectangle

cv2.putText(image, class\_name, (x, y - 10), cv2.FONT\_HERSHEY\_SIMPLEX, 0.5, (0, 255, 0), 2)

plt.imshow(cv2.cvtColor(image, cv2.COLOR\_BGR2RGB))

plt.axis('off')

plt.show()

# Predict the object class and bounding box

def predict\_object\_detection(model, image):

"""

Predicts the class and bounding box for a given image.

Args:

model: The trained object detection model.

image: The input image to make predictions on.

Returns:

predicted\_class: The predicted class index.

predicted\_bounding\_box: The predicted bounding box coordinates (x, y, width, height).

"""

# Preprocess the input image (resize, normalize, etc.)

image\_resized = cv2.resize(image, (224, 224)) # Assuming the model expects 224x224 input size

image\_normalized = image\_resized / 255.0 # Normalize the image to [0, 1]

image\_batch = np.expand\_dims(image\_normalized, axis=0) # Add batch dimension

# Make predictions using the trained model

class\_predictions, bbox\_predictions = model.predict(image\_batch)

# Get the class with the highest probability

predicted\_class = np.argmax(class\_predictions, axis=-1)[0] # Get the index of the max value

predicted\_bounding\_box = bbox\_predictions[0] # The model predicts the bounding box

return predicted\_class, predicted\_bounding\_box

# Main function to run the training and testing process

def main():

input\_shape = (224, 224, 3) # Example image size

num\_classes = 10 # Example: 10 different object classes

class\_names = ['Class 0', 'Class 1', 'Class 2', 'Class 3', 'Class 4', 'Class 5', 'Class 6', 'Class 7', 'Class 8', 'Class 9']

# Build the model

model = build\_object\_detection\_model(input\_shape, num\_classes)

model.summary()

# Create dummy data for demonstration

num\_train\_samples = 100

num\_val\_samples = 20

num\_test\_samples = 30

train\_images = np.random.rand(num\_train\_samples, 224, 224, 3)

train\_class\_labels = np.random.randint(0, num\_classes, size=(num\_train\_samples, num\_classes)) # One-hot encoded

train\_bbox\_labels = np.random.rand(num\_train\_samples, 4) # (x, y, width, height)

val\_images = np.random.rand(num\_val\_samples, 224, 224, 3)

val\_class\_labels = np.random.randint(0, num\_classes, size=(num\_val\_samples, num\_classes))

val\_bbox\_labels = np.random.rand(num\_val\_samples, 4)

test\_images = np.random.rand(num\_test\_samples, 224, 224, 3)

test\_class\_labels = np.random.randint(0, num\_classes, size=(num\_test\_samples, num\_classes))

test\_bbox\_labels = np.random.rand(num\_test\_samples, 4)

train\_data = (train\_images, [train\_class\_labels, train\_bbox\_labels])

val\_data = (val\_images, [val\_class\_labels, val\_bbox\_labels])

test\_data = (test\_images, [test\_class\_labels, test\_bbox\_labels])

# Train the model

trained\_model = train\_object\_detector(model, train\_data, val\_data, epochs=10, batch\_size=32)

# Make a prediction

sample\_image = np.random.rand(224, 224, 3)

predicted\_class, predicted\_bounding\_box = predict\_object\_detection(trained\_model, sample\_image)

print(f"Predicted Class: {predicted\_class}, Predicted Bounding Box: {predicted\_bounding\_box}")

# Evaluate the model

evaluation\_metrics = evaluate\_object\_detector(trained\_model, test\_data)

print("Evaluation Metrics:", evaluation\_metrics)

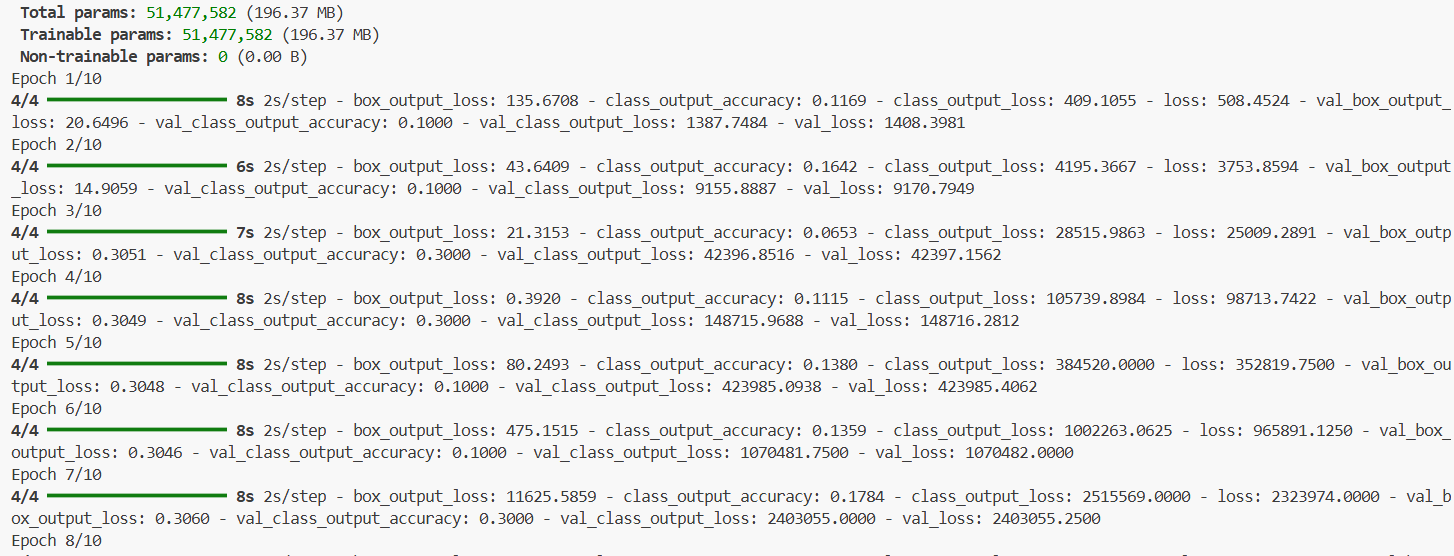
# Visualize the prediction

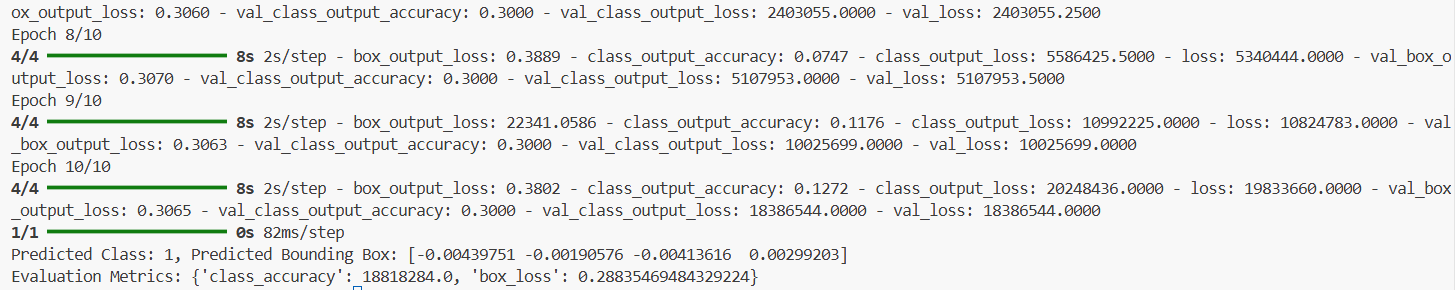
visualize\_prediction(sample\_image, predicted\_class, predicted\_bounding\_box, class\_names)

if \_\_name\_\_ == '\_\_main\_\_':

main()







**Practical No 11**

import tensorflow as tf

from tensorflow.keras import layers, models

from tensorflow.keras.datasets import mnist

from tensorflow.keras.utils import to\_categorical

# Load MNIST data

(train\_images, train\_labels), (test\_images, test\_labels) = mnist.load\_data()

# Preprocess the data

train\_images = train\_images.reshape((train\_images.shape[0], 28, 28, 1)) # Reshape to 4D tensor

test\_images = test\_images.reshape((test\_images.shape[0], 28, 28, 1)) # Reshape to 4D tensor

train\_images, test\_images = train\_images / 255.0, test\_images / 255.0 # Normalize to [0, 1]

# One-hot encode the labels

train\_labels = to\_categorical(train\_labels)

test\_labels = to\_categorical(test\_labels)

# Build the neural network model

model = models.Sequential([

layers.Conv2D(32, (3, 3), activation='relu', input\_shape=(28, 28, 1)),

layers.MaxPooling2D((2, 2)),

layers.Conv2D(64, (3, 3), activation='relu'),

layers.MaxPooling2D((2, 2)),

layers.Conv2D(64, (3, 3), activation='relu'),

layers.Flatten(),

layers.Dense(64, activation='relu'),

layers.Dense(10, activation='softmax') # 10 classes for digits 0-9

])

# Compile the model

model.compile(optimizer='adam',

loss='categorical\_crossentropy',

metrics=['accuracy'])

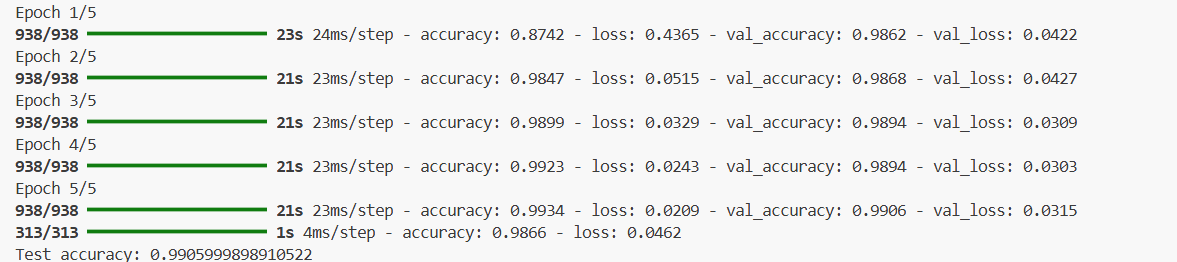
# Train the model

model.fit(train\_images, train\_labels, epochs=5, batch\_size=64, validation\_data=(test\_images, test\_labels))

# Evaluate the model

test\_loss, test\_acc = model.evaluate(test\_images, test\_labels)

print(f"Test accuracy: {test\_acc}")



**Practical No 12**

import tensorflow as tf

from tensorflow.keras import layers, models

from tensorflow.keras.datasets import mnist

# Load and preprocess data

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

x\_train = x\_train.reshape(-1, 28, 28, 1).astype("float32") / 255.0

x\_test = x\_test.reshape(-1, 28, 28, 1).astype("float32") / 255.0

# Build the CNN model

model = models.Sequential([

    layers.Conv2D(32, (3, 3), activation='relu', input\_shape=(28, 28, 1)),

    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(64, (3, 3), activation='relu'),

    layers.MaxPooling2D((2, 2)),

    layers.Flatten(),

    layers.Dense(64, activation='relu'),

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

])

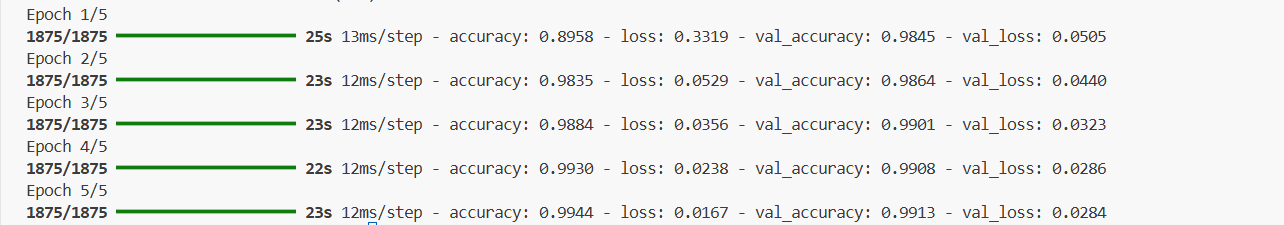
# Compile and train

model.compile(optimizer='adam', loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

model.fit(x\_train, y\_train, epochs=5, validation\_data=(x\_test, y\_test))

# Save the model in `.keras` format

model.save("cnn\_model.keras")



**Practical No 13**

import tensorflow as tf

from tensorflow.keras import layers, models

from tensorflow.keras.datasets import mnist

from tensorflow.keras.utils import to\_categorical

# Load and preprocess data

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

x\_train = x\_train.reshape(-1, 28, 28, 1).astype("float32") / 255.0

x\_test = x\_test.reshape(-1, 28, 28, 1).astype("float32") / 255.0

y\_train = to\_categorical(y\_train, 10)

y\_test = to\_categorical(y\_test, 10)

# Build the CNN model

model = models.Sequential([

    layers.Conv2D(32, (3, 3), activation='relu', input\_shape=(28, 28, 1)),

    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(64, (3, 3), activation='relu'),

    layers.MaxPooling2D((2, 2)),

    layers.Flatten(),

    layers.Dense(64, activation='relu'),

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

])

# Compile and train the model

model.compile(optimizer='adam', loss='categorical\_crossentropy', metrics=['accuracy'])

model.fit(x\_train, y\_train, epochs=5, validation\_data=(x\_test, y\_test))

# Evaluate the model

test\_loss, test\_acc = model.evaluate(x\_test, y\_test)

print(f"Test accuracy: {test\_acc \* 100:.2f}%")

