**Experiment - 11**

**Program:**

import numpy as np

import matplotlib.pyplot as plt

def sigmoid(x):

    return 1 / (1 + np.exp(-x))

def sigmoid\_derivative(x):

    return x \* (1 - x)

def mlp\_xor(inputs, targets, learning\_rate, epochs):

    input\_neurons = 2

    hidden\_neurons = 2

    output\_neurons = 1

    # Initialize weights with random values

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

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

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

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

    errors = []

    for epoch in range(epochs):

        total\_error = 0

        for i in range(len(inputs)):

            # Forward propagation

            hidden\_layer\_input = np.dot(inputs[i], hidden\_weights) + hidden\_bias

            hidden\_layer\_output = sigmoid(hidden\_layer\_input)

            output\_layer\_input = np.dot(hidden\_layer\_output, output\_weights) + output\_bias

            predicted\_output = sigmoid(output\_layer\_input)

            # Calculate error

            error = targets[i] - predicted\_output

            total\_error += abs(error[0])

            # Backpropagation

            delta\_output = error \* sigmoid\_derivative(predicted\_output)

            error\_hidden = delta\_output.dot(output\_weights.T)

            delta\_hidden = error\_hidden \* sigmoid\_derivative(hidden\_layer\_output)

            # Update weights and biases

            output\_weights += hidden\_layer\_output.reshape(-1,1).dot(delta\_output) \* learning\_rate

            output\_bias += delta\_output \* learning\_rate

            hidden\_weights += inputs[i].reshape(-1,1).dot(delta\_hidden) \* learning\_rate

            hidden\_bias += delta\_hidden \* learning\_rate

        errors.append(total\_error / len(inputs))

    return hidden\_weights, hidden\_bias, output\_weights, output\_bias, errors

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

#Binary

targets\_binary = np.array([[0], [1], [1], [0]])

#Bipolar

targets\_bipolar = np.array([[-1], [1], [1], [-1]])

learning\_rate = 0.1

epochs = 10000

hidden\_weights\_b, hidden\_bias\_b, output\_weights\_b, output\_bias\_b, errors\_b = mlp\_xor(inputs, targets\_binary, learning\_rate, epochs)

hidden\_weights\_bp, hidden\_bias\_bp, output\_weights\_bp, output\_bias\_bp, errors\_bp = mlp\_xor(inputs, targets\_bipolar, learning\_rate, epochs)

#Plot the error curve

plt.plot(errors\_b, label='Binary')

plt.plot(errors\_bp, label='Bipolar')

plt.xlabel('Epochs')

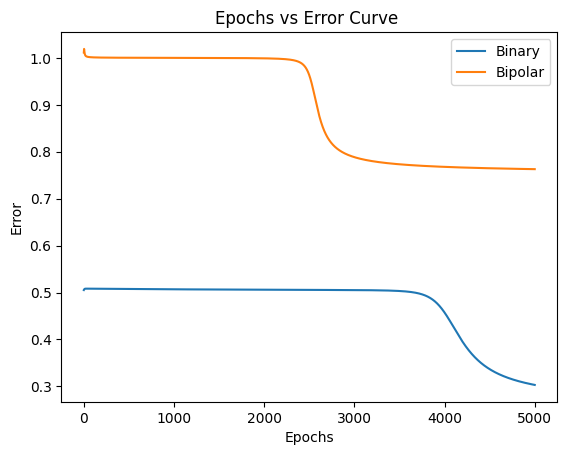
plt.ylabel('Error')

plt.title('Epochs vs Error Curve ')

plt.legend()

plt.show()

**OUTPUT:**



**Experiment - 10**

**Program:**

import numpy as np

def hebb\_rule(inputs, targets, learning\_rate, epochs):

    num\_inputs = len(inputs[0])

    weights = np.zeros(num\_inputs)

    bias = 0

    print("Initial weights:", weights)

    print("Initial bias:", bias)

    for epoch in range(epochs):

        print(f"\nEpoch {epoch + 1}:")

        for i in range(len(inputs)):

            weighted\_sum = np.dot(inputs[i], weights) + bias

            #Hebb rule weight update

            weights = weights + learning\_rate \* targets[i] \* inputs[i]

            bias = bias + learning\_rate \* targets[i]

            print(f"  Input: {inputs[i]}, Target: {targets[i]}, Weights: {weights}, Bias: {bias}")

    return weights, bias

# Example usage (AND gate)

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

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

learning\_rate = 0.2

epochs = 3  #for better output formatting

weights, bias = hebb\_rule(inputs, targets, learning\_rate, epochs)

print("\nFinal weights:", weights)

print("Final bias:", bias)

**OUTPUT:**

**AND GATE**

|  |  |
| --- | --- |
| **Epoch 1** | Input: [0 0], Target: 0, Weights: [0. 0.], Bias: 0.0  Input: [0 1], Target: 0, Weights: [0. 0.], Bias: 0.0  Input: [1 0], Target: 0, Weights: [0. 0.], Bias: 0.0  Input: [1 1], Target: 1, Weights: [0.2 0.2], Bias: 0.2 |
| **Epoch 2** | Input: [0 0], Target: 0, Weights: [0.2 0.2], Bias: 0.2  Input: [0 1], Target: 0, Weights: [0.2 0.2], Bias: 0.2  Input: [1 0], Target: 0, Weights: [0.2 0.2], Bias: 0.2  Input: [1 1], Target: 1, Weights: [0.4 0.4], Bias: 0.4 |
| **Epoch 3** | Input: [0 0], Target: 0, Weights: [0.4 0.4], Bias: 0.4  Input: [0 1], Target: 0, Weights: [0.4 0.4], Bias: 0.4  Input: [1 0], Target: 0, Weights: [0.4 0.4], Bias: 0.4  Input: [1 1], Target: 1, Weights: [0.6 0.6], Bias: 0.600000000000000 |

Final weights: [0.6 0.6]

Final bias: 0.6000000000000001

**Experiment - 13**

**Program:**

import os

import struct

import numpy as np

import pandas as pd

from zipfile import ZipFile

def extract\_zip(zip\_path, extract\_to):

    with ZipFile(zip\_path, 'r') as zip\_ref:

        zip\_ref.extractall(extract\_to)

    print(f"Extracted files to: {extract\_to}")

def read\_idx\_images(file\_path):

    with open(file\_path, 'rb') as file:

        # Read metadata

        magic\_number, num\_images, rows, cols = struct.unpack('>IIII', file.read(16))

        images = np.frombuffer(file.read(), dtype=np.uint8).reshape(num\_images, rows, cols)

    return images

def read\_idx\_labels(file\_path):

    with open(file\_path, 'rb') as file:

        magic\_number, num\_labels = struct.unpack('>II', file.read(8))

        labels = np.frombuffer(file.read(), dtype=np.uint8)

    return labels

def create\_dataframe(images, labels):

    # Flatten images into rows

    flattened\_images = images.reshape(images.shape[0], -1)

    df = pd.DataFrame(flattened\_images)

    df['label'] = labels

    return df

zip\_path = "MNIST\_ORG.zip"  # Path to the zip file

extract\_to = "mnist\_data"   # Extraction directory

extract\_zip(zip\_path, extract\_to)

train\_images\_path = os.path.join(extract\_to, "train-images.idx3-ubyte")

train\_labels\_path = os.path.join(extract\_to, "train-labels.idx1-ubyte")

test\_images\_path = os.path.join(extract\_to, "t10k-images.idx3-ubyte")

test\_labels\_path = os.path.join(extract\_to, "t10k-labels.idx1-ubyte")

train\_images = read\_idx\_images(train\_images\_path)

train\_labels = read\_idx\_labels(train\_labels\_path)

test\_images = read\_idx\_images(test\_images\_path)

test\_labels = read\_idx\_labels(test\_labels\_path)

train\_df = create\_dataframe(train\_images, train\_labels)

test\_df = create\_dataframe(test\_images, test\_labels)

print("Training DataFrame:")

print(train\_df.head())

print("Test DataFrame:")

print(test\_df.head())

from sklearn.model\_selection import train\_test\_split

X\_train = train\_df.drop(columns=['label']).values

y\_train = train\_df['label'].values

X\_test = test\_df.drop(columns=['label']).values

y\_test = test\_df['label'].values

X\_train\_images = X\_train.reshape(-1, 28, 28)

X\_test\_images = X\_test.reshape(-1, 28, 28)

import tensorflow as tf

from tensorflow.keras.models import Sequential

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

from sklearn.model\_selection import train\_test\_split

from tensorflow.keras.utils import to\_categorical

X\_train\_images = X\_train\_images / 255.0

X\_test\_images = X\_test\_images / 255.0

X\_train\_images = X\_train\_images.reshape(-1, 28, 28, 1)

X\_test\_images = X\_test\_images.reshape(-1, 28, 28, 1)

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

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

# Create a CNN model

model = Sequential()

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

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Conv2D(64, (3, 3), activation='relu'))

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Flatten())

model.add(Dense(128, activation='relu'))

model.add(Dropout(0.5))  # Dropout to prevent overfitting

model.add(Dense(10, activation='softmax'))  # Output layer for 10 classes

model.compile(optimizer='adam',

              loss='categorical\_crossentropy',

              metrics=['accuracy'])

history = model.fit(X\_train\_images, y\_train,

                    validation\_data=(X\_test\_images, y\_test),

                    epochs=10,

                    batch\_size=64)

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

print(f"Test Accuracy: {test\_acc:.2f}")

import matplotlib.pyplot as plt

# Plot training and validation accuracy

plt.plot(history.history['accuracy'], label='Train Accuracy')

plt.plot(history.history['val\_accuracy'], label='Validation Accuracy')

plt.xlabel('Epochs')

plt.ylabel('Accuracy')

plt.legend()

plt.title('Model Accuracy')

plt.show()

# Plot training and validation loss

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

plt.plot(history.history['val\_loss'], label='Validation Loss')

plt.xlabel('Epochs')

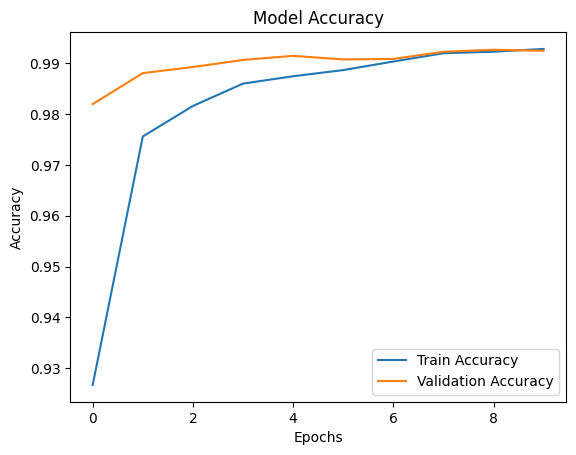
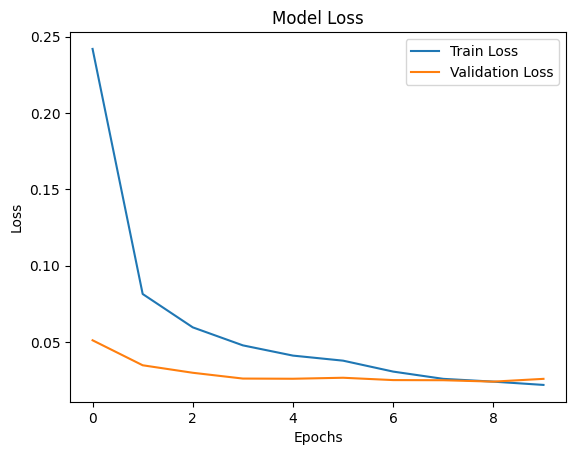
plt.ylabel('Loss')

plt.legend()

plt.title('Model Loss')

plt.show()

**OUTPUT:**



model.summary()

|  |  |  |
| --- | --- | --- |
| **Layer (type)** | **Output Shape** | **Param** |
| conv2d\_2 (Conv2D) | (None, 26, 26, 32) | 320 |
| max\_pooling2d (MaxPooling2D) | (None, 13, 13, 32) | 0 |
| conv2d\_3 (Conv2D) | (None, 11, 11, 64) | 18,496 |
| max\_pooling2d\_1 (MaxPooling2D) | (None, 5, 5, 64) | 0 |
| flatten (Flatten) | (None, 1600) | 0 |
| dense (Dense) | (None, 128) | 204,928 |
| dropout (Dropout) | (None, 128) | 0 |
| dense\_1 (Dense) | (None, 10) | 1290 |

**Experiment - 12**

**Program:**

import pandas as pd

import numpy as np

from sklearn.ensemble import RandomForestClassifier

from sklearn.model\_selection import train\_test\_split, cross\_val\_score

from sklearn.metrics import accuracy\_score

import scipy.io

import warnings

warnings.filterwarnings("ignore")

data=pd.read\_csv("/content/lymphography.data")

print(data.head())

X=data.drop(["3"],axis=1)

y=data["3"]

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

model = RandomForestClassifier(n\_estimators=100, random\_state=42)

model.fit(X\_train, y\_train)

y\_pred\_before = model.predict(X\_test)

initial\_score = accuracy\_score(y\_test, y\_pred\_before)

print(f"Initial model accuracy (before feature selection): {initial\_score:.4f}")

def forward\_selection(X, y, model, max\_features=5, scoring='accuracy', cv=5, min\_improvement=0.001):

    remaining\_features = list(X.columns)

    selected\_features = []

    best\_score = 0

    iteration = 0

    while remaining\_features and len(selected\_features) < max\_features:

        scores = []

        for feature in remaining\_features:

            current\_features = selected\_features + [feature]

            X\_selected = X[current\_features]

            score = cross\_val\_score(model, X\_selected, y, cv=cv, scoring=scoring).mean()

            scores.append((score, feature))

        best\_score\_feature = max(scores, key=lambda x: x[0])

        if best\_score\_feature[0] - best\_score < min\_improvement:

            print(f"No significant improvement; stopping at iteration {iteration}")

            break

        best\_score = best\_score\_feature[0]

        selected\_features.append(best\_score\_feature[1])

        remaining\_features.remove(best\_score\_feature[1])

        iteration += 1

        print(f"Selected feature: {best\_score\_feature[1]}, Cross-validated score: {best\_score:.4f}")

    return selected\_features

def backward\_selection(X, y, model, min\_features=5, scoring='accuracy', cv=5, min\_improvement=0.001):

    selected\_features = list(X.columns)

    best\_score = cross\_val\_score(model, X[selected\_features], y, cv=cv, scoring=scoring).mean()

    iteration = 0

    while len(selected\_features) > min\_features:

        scores = []

        for feature in selected\_features:

            current\_features = [f for f in selected\_features if f != feature]

            X\_selected = X[current\_features]

            score = cross\_val\_score(model, X\_selected, y, cv=cv, scoring=scoring).mean()

            scores.append((score, feature))

        best\_score\_feature = max(scores, key=lambda x: x[0])

        if best\_score - best\_score\_feature[0] > min\_improvement:

            print(f"Performance drop detected; stopping at iteration {iteration}")

            break

        best\_score = best\_score\_feature[0]

        selected\_features.remove(best\_score\_feature[1])

        iteration += 1

        print(f"Removed feature: {best\_score\_feature[1]}, Cross-validated score: {best\_score:.4f}")

    return selected\_features

print("\nPerforming Sequential Forward Selection (SFS):")

sfs\_selected\_features = forward\_selection(X\_train, y\_train, model, max\_features=5)

X\_train\_sfs = X\_train[sfs\_selected\_features]

X\_test\_sfs = X\_test[sfs\_selected\_features]

model.fit(X\_train\_sfs, y\_train)

y\_pred\_sfs = model.predict(X\_test\_sfs)

sfs\_score = accuracy\_score(y\_test, y\_pred\_sfs)

print(f"\nModel accuracy after SFS: {sfs\_score:.4f}")

print("\nPerforming Sequential Backward Selection (SBS):")

sbs\_selected\_features = backward\_selection(X\_train, y\_train, model, min\_features=5)

X\_train\_sbs = X\_train[sbs\_selected\_features]

X\_test\_sbs = X\_test[sbs\_selected\_features]

model.fit(X\_train\_sbs, y\_train)

y\_pred\_sbs = model.predict(X\_test\_sbs)

sbs\_score = accuracy\_score(y\_test, y\_pred\_sbs)

print(f"\nModel accuracy after SBS: {sbs\_score:.4f}")

**OUTPUT:**

Initial model accuracy (before feature selection): 0.8444

Performing Sequential Forward Selection (SFS):

Selected feature: 4.1, Cross-validated score: 0.7552

Selected feature: 2.5, Cross-validated score: 0.8048

Selected feature: 1.1, Cross-validated score: 0.8248

Selected feature: 2.6, Cross-validated score: 0.8343

No significant improvement; stopping at iteration 4

Model accuracy after SFS: 0.7556

Performing Sequential Backward Selection (SBS):

Removed feature: 2.1, Cross-validated score: 0.8624

Performance drop detected; stopping at iteration 1

Model accuracy after SBS: 0.8667