Ex: 3 **Implementation of Artificial Neural Networks**

AIM: The aim of Artificial Neural Networks is to simulate the way a human brain processes information

Algorithm:

Step 1: **Initialize weights and biases** (usually small random numbers)

Step 2: **Forward Propagation**:

- Compute the weighted sum: Z = W·X + b
- Apply activation function: A = activation(Z)

Step 3: **Loss Calculation**:

- Compute the loss (e.g., Mean Squared Error or Cross-Entropy)

Step 4: **Backward Propagation**:

- Compute gradients of loss w.r.t weights and biases using chain rule

Step 5: **Update Weights**:

- Update parameters using gradient descent:
  - W = W - learning_rate * dW
  - b = b - learning_rate * db

Step 5: **Repeat steps 2–5 for multiple epochs**

IMPLEMENTATION

```
import numpy as np

# Activation Function (Sigmoid)

def sigmoid(z):

    return 1 / (1 + np.exp(-z))

# Derivative of Sigmoid

def sigmoid_derivative(a):

    return a * (1 - a)

# Loss Function (Binary Cross-Entropy)

def compute_loss(y_true, y_pred):

    m = y_true.shape[0]

    return -np.mean(y_true * np.log(y_pred) + (1 - y_true) * np.log(1 - y_pred))

# Initialize parameters

def initialize_parameters(input_dim, hidden_dim, output_dim):

    np.random.seed(1)
```

```
    W1 = np.random.randn(input_dim, hidden_dim)
    b1 = np.zeros((1, hidden_dim))
    W2 = np.random.randn(hidden_dim, output_dim)
    b2 = np.zeros((1, output_dim))
    return W1, b1, W2, b2
# Training the Neural Network
def train(X, Y, hidden_dim=4, epochs=10000, lr=0.01):
    input_dim, output_dim = X.shape[1], 1
    W1, b1, W2, b2 = initialize_parameters(input_dim, hidden_dim, output_dim)
    for epoch in range(epochs):
        # Forward Propagation
        Z1 = X @ W1 + b1
        A1 = sigmoid(Z1)
        Z2 = A1 @ W2 + b2
        A2 = sigmoid(Z2)
        # Compute Loss
        loss = compute_loss(Y, A2)
        # Backward Propagation
        dZ2 = A2 - Y
        dW2 = A1.T @ dZ2
        db2 = np.sum(dZ2, axis=0, keepdims=True)
        dZ1 = dZ2 @ W2.T * sigmoid_derivative(A1)
        dW1 = X.T @ dZ1
        db1 = np.sum(dZ1, axis=0)
        # Update weights
        W1 -= lr * dW1
        b1 -= lr * db1
        W2 -= lr * dW2
        b2 -= lr * db2
        if epoch % 1000 == 0:
            print(f"Epoch {epoch} - Loss: {loss:.4f}")
    return W1, b1, W2, b2
```

```
# Predict function
def predict(X, W1, b1, W2, b2):
    A1 = sigmoid(X @ W1 + b1)
    A2 = sigmoid(A1 @ W2 + b2)
    return (A2 > 0.5).astype(int)
```

Output:

```
Epoch 0 - Loss: 0.7139
Epoch 1000 - Loss: 0.2804
Epoch 2000 - Loss: 0.2406
Epoch 3000 - Loss: 0.2160
Epoch 4000 - Loss: 0.2000
```

EX 4 Implementation of Fuzzy Sets

AIM: The aim of Fuzzy Set Theory is to model uncertainty

ALGORITHM:

Step 1: Define the universe of discourse

- A list of elements over which the fuzzy set is defined.

Step 2: Define membership functions

- A function $\mu(x)$: $U \rightarrow [0, 1]$ that assigns a degree of membership to each element.

Step 3: Create fuzzy sets

- Use the universe and membership function to define fuzzy sets (e.g., "Hot", "Cold").

Step 4: Apply fuzzy operations:

- Union: $\mu A \cup B(x) = \max(\mu A(x), \mu B(x))$
- Intersection: $\mu A \cap B(x) = \min(\mu A(x), \mu B(x))$
- Complement: $\mu A'(x) = 1 - \mu A(x)$

Step 5: Display results

- Show degrees of membership for elements after operations.

IMPLENTATION

```
import numpy as np
universe = np.linspace(0, 100, 11)  # [0, 10, 20, ..., 100]
def cold(x):
    return max(0, min(1, (30 - x) / 30))
def warm(x):
    return max(0, min((x - 20) / 30, (80 - x) / 30))
def hot(x):
    return max(0, min(1, (x - 60) / 40))
cold_set = {x: cold(x) for x in universe}
warm_set = {x: warm(x) for x in universe}
hot_set  = {x: hot(x)  for x in universe}
def fuzzy_union(setA, setB):
    return {x: max(setA[x], setB[x]) for x in setA}
def fuzzy_intersection(setA, setB):
    return {x: min(setA[x], setB[x]) for x in setA}
def fuzzy_complement(fuzzy_set):
    return {x: 1 - fuzzy_set[x] for x in fuzzy_set}
print("Fuzzy Set - Cold:")
print(cold_set)
print("\nFuzzy Set - Warm:")
print(warm_set)
print("\nFuzzy Set - Hot:")
print(hot_set)
# Union of Cold and Warm
union_cold_warm = fuzzy_union(cold_set, warm_set)
print("\nUnion (Cold ∪ Warm):")
print(union_cold_warm)
# Intersection of Warm and Hot
intersection_warm_hot = fuzzy_intersection(warm_set, hot_set)
print("\nIntersection (Warm ∩ Hot):")
print(intersection_warm_hot)
```

Complement of Hot

complement_hot = fuzzy_complement(hot_set)

print("\nComplement of Hot:")

print(complement_hot)

OUTPUT:

Fuzzy Set - Cold:

{0.0: 1, 10.0: 0.666..., 20.0: 0.333..., 30.0: 0.0, ..., 100.0: 0}

Fuzzy Set - Warm:

{0.0: 0, 10.0: 0, 20.0: 0.0, 30.0: 0.333..., ..., 70.0: 0.333..., 100.0: 0.0}

Fuzzy Set - Hot:

{0.0: 0, ..., 60.0: 0.0, 70.0: 0.25, 80.0: 0.5, 90.0: 0.75, 100.0: 1.0}

Union (Cold ∪ Warm):

{0.0: 1, 10.0: 0.666..., 20.0: 0.333..., 30.0: 0.333..., ..., 100.0: 0.0}

Intersection (Warm ∩ Hot):

{0.0: 0, ..., 70.0: 0.25, 80.0: 0.0, ..., 100.0: 0.0}

Complement of Hot:

{0.0: 1, ..., 100.0: 0.0}

EX: 5 Implementation of Covariance

AIM:

To implement the covariance between two variables

ALGORITHM

Step 1: Check the input data Ensure both lists/arrays X and Y have the same number of elements: if len(X) != len(Y): raise error

Step 2: Calculate the mean of X and Y

Step 3: Subtract the mean from each element

For each index i∈[1,n]i \in [1, n]i∈[1,n]:

Step 4: Multiply corresponding deviations For each index i∈[1,n]

Step 5: Divide by (n - 1) (for sample covariance) Cov(X,Y) = sum_product/ n−1

IMPLENTATION

```
def calculate_covariance(X, Y):
    if len(X) != len(Y):
        raise ValueError("X and Y must be the same length.")
    n = len(X)
    mean_x = sum(X) / n
    mean_y = sum(Y) / n
    covariance_sum = sum((X[i] - mean_x) * (Y[i] - mean_y) for i in range(n))
    covariance = covariance_sum / (n - 1)  # Sample covariance
    return covariance
# Sample Data
X = [2, 4, 6, 8, 10]
Y = [1, 3, 5, 7, 9]
# Calculate and print covariance
cov = calculate_covariance(X, Y)
print("Covariance between X and Y:", cov)
```

OUTPUT:

X = [2, 4, 6, 8, 10]

Y = [1, 3, 5, 7, 9]