1. Implementation of Logic gates using Artificial Neural Network

1. Import the necessary libraries, including NumPy and Matplotlib.
2. Define the sigmoid activation function.
3. Define the function to initialize the parameters of the neural network (weights and biases).
4. Define the forward propagation function, which computes the output of the neural network for a given input and set of parameters.
5. Define the backward propagation function, which computes the gradients of the parameters with respect to the loss function.
6. Define the function to update the parameters using the computed gradients and a given learning rate.
7. Set the input and output data, the number of neurons in the hidden layer, the number of input features, and the number of output features.
8. Initialize the parameters of the neural network.
9. Set the number of epochs and the learning rate.
10. Initialize an array to store the loss values during training.
11. Loop over the specified number of epochs, computing the loss and gradients for each epoch and updating the parameters.
12. Plot the loss values over the course of training.
13. Test the neural network on a different set of inputs, computing the output and comparing it to a threshold to make a prediction.
14. Print the prediction.

3. Implementation of Back Propagation Algorithm

1. Import the necessary libraries: numpy and matplotlib.pyplot
2. Define sigmoid activation function
3. Define the function to initialize the parameters of the neural network (weights and biases)
4. Define the function to perform forward propagation and compute the cost function
5. Define the function to perform backward propagation and compute the gradients of the cost function with respect to the parameters
6. Define the function to update the parameters using the computed gradients and a given learning rate
7. Define the input data (X) and output data (Y) for the XOR problem

8. Initialize the parameters of the neural network using the input and output dimensions
9. Set the number of epochs and learning rate
10. Create an array to store the losses for each epoch
11. For each epoch, perform forward propagation, backward propagation, and parameter updates using the training data
12. Plot the loss over the epochs
13. Use the trained model to make predictions on new data (in this case, the same input data used for training)
14. Print the predictions

2. Implementation of Perception Algorithm

1. Define the unit step function that returns 1 if the input is greater than or equal to zero, and 0 otherwise.
2. Define the perceptron model function that takes in an input vector, weight vector, and bias value. The function calculates the weighted sum of the input vector and bias, and then passes it through the unit step function to obtain the output.
3. Define the OR logic function that uses the perceptron model with weights [1,1] and bias 0.5 to determine the OR of two inputs.
4. Define four input vectors to test the OR logic function.
5. Print the results of the OR logic function for each test input vector.

4. Implementation of Self Organizing Maps

1. Define a class named SOM.
2. Implement a method named winner that takes two arguments - weights and sample - and returns the index of the neuron that has the smallest Euclidean distance to the sample.
3. Implement a method named update that takes four arguments - weights, sample, J, and alpha - and updates the weights of the neuron at index J based on the given sample and learning rate.
4. Define a main function.
5. Create a training dataset named T, initialize the number of rows (m) and number of columns (n), and create an initial set of weights.

6. Create an instance of the SOM class and set the number of epochs and learning rate.
7. Train the model for the given number of epochs by iterating over the training dataset and updating the weights for each sample.
8. Test the model by providing a sample s and determining which cluster it belongs to.
9. Print the cluster index and the trained weights.

5. Implementation of Radial Basis Function Network

1. Define the RBFNN class with the following methods:
    a. **init**() to initialize the number of kernels, centers, beta, learning rate, epochs, weights, biases, errors, and gradients.
    b. rbf_activation() to calculate the radial basis function activation.
    c. linear_activation() to calculate the linear activation.
    d. least_square_error() to calculate the least square error.
    e. _forward_propagation() to perform the forward propagation of the input data.
    f. _backward_propagation() to perform the backward propagation of the error.
    g. fit() to train the network using input and output data.
    h. predict() to predict the output for a given input.
2. Define the main() function to create the input and output data, create an instance of the RBFNN class, train the network, and test the network using XOR gate inputs.
3. Create an instance of the RBFNN class with the desired number of kernels, centers, beta, learning rate, and epochs.
4. Train the network using the fit() method and the input and output data.
5. Use the predict() method to predict the output for each of the XOR gate inputs.

6. Print the RBF weights, RBF bias, and the predicted output for each XOR gate input.

6. Implementation of De-Morgan‘s Law

1. Start the function **demorgan_law** with a single parameter **statement**.
2. Check if the string 'not (' is in the statement and ' and ' is in the statement.
3. If step 2 is True, split the statement using the string 'not (' as the delimiter and retrieve the A and B values using the string ' and ' as the delimiter.
4. Return a formatted string with the retrieved A and B values and 'not' keyword.
5. If step 2 is False, check if the string 'not (' is in the statement and ' or ' is in the statement.
6. If step 5 is True, split the statement using the string 'not (' as the delimiter and retrieve the A and B values using the string ' or ' as the delimiter.
7. Return a formatted string with the retrieved A and B values and 'not' keyword.
8. If step 2 and step 6 are both False, return the original statement.

8. Implementation of Simple genetic algorithm

1. Define onemax(x) function that returns the negative sum of bits in binary string x.
2. Define selection(pop, scores, k=3) function that randomly selects one individual from the population pop and k-1 additional individuals with replacement, and returns the individual with the lowest score.
3. Define crossover(p1, p2, r_cross) function that takes two parent individuals p1 and p2 and a crossover rate r_cross, and returns a list containing two children created by combining the bits before and after a random crossover point.

4. Define mutation(bitstring, r_mut) function that takes a binary string bitstring and a mutation rate r_mut, and flips each bit with probability r_mut.
5. Define genetic_algorithm(objective, n_bits, n_iter, n_pop, r_cross, r_mut) function that initializes the population, evaluates fitness scores, selects parents, creates children using crossover, mutates the children, and replaces the old population with the new population of children for n_iter generations, returning the best individual and its score.

9. Implementation of fuzzy based Logical operations

1. Define the first fuzzy set A and the second fuzzy set B with their respective membership values.
2. Create an empty dictionary Y to store the result of the union, intersection or complement of the fuzzy sets.
3. To compute the union of A and B, iterate through the keys of A and B simultaneously, and for each key, compare the membership values of A and B. Store the maximum value in Y for that key.
4. To compute the intersection of A and B, iterate through the keys of A and B simultaneously, and for each key, compare the membership values of A and B. Store the minimum value in Y for that key.
5. To compute the complement of A, iterate through the keys of A and for each key, subtract its membership value from 1 and store the result in Y for that key.

10. Implementation of fuzzy based arithmetic operations

1. Define four functions: fuzzy_addition, fuzzy_subtraction, fuzzy_multiplication, fuzzy_division.
2. Each function takes two fuzzy sets a and b as input.
3. Initialize an empty dictionary called result.
4. Use two nested for loops to iterate over each element in both sets.

5. If the keys match, apply the corresponding operation (max, subtraction, multiplication, division) to the values and store the result in the result dictionary.
6. Break out of the inner loop and continue to the next key in set a.
7. Return the result dictionary.
8. Define two fuzzy sets A and B using dictionaries.
9. Call the four functions on the sets and store the results in C, D, E, and F respectively.
10. Print the resulting fuzzy sets.