Question 1 Answer

For a boolean function on 5 variables, the Karnaugh map has 32 cells, corresponding to the $2^5 = 32$ possible input combinations. Each cell can be either 0 or 1, so there are 2^{32} possible boolean functions on 5 variables.

The maximum number of minterms in the Karnaugh map of a boolean function on 5 variables is 16. This is because if a boolean function on 5 variables has more than 16 minterms (i.e., more than 16 ones in its Karnaugh map), then at least one pair of adjacent cells can be grouped together to form a larger group of multiple cells that can be covered by a single gate. Therefore, the maximum number of gates required in a single hidden layer to implement a boolean function with 5 inputs is 16.

However, it's worth noting that constructing a network with 16 gates that can implement a complex function can be difficult in practice, and a more efficient network may require fewer gates. Increasing the number of layers in the network can allow for a more efficient implementation of the function. By adding additional layers to the network, the gates in each layer can be used to combine the output of gates in the previous layer, creating a hierarchical structure that can represent more complex functions with fewer gates. Any boolean function on n variables can be expressed as a composition of boolean functions on two variables using the Boolean Algebra. We can construct the network like a binary tree, that would result in a tree with total 12 $(3 * (n-1))$ nodes and $2 * \log_2 5 \approx 5$ layers.

Depending on the boolean function, there may be a case where the K-map reduces the whole function to a single gate.

Question 2 Answer

Here is the implementation code

```
import numpy as np

def sig(x):
 return 1/(1 + np.exp(-x))

def calculate_label(XY, a, b, r):
    if (XY[0] - a)**2 + (XY[1] - b)**2 < r**2:
        return 1
    else:
        return 0


def generate_data(D, N, a, b, r):
    temp = np.random.default_rng().uniform(size=(D, N))
    Points = np.ones((D, N+1))
    Points[:, :-1] = temp
    Labels = np.array(list(map(lambda XY: calculate_label(XY, a, b,
     r), Points))).reshape((D, 1))
    return Points, Labels

```

```
D = 100
N = 2
eta = 0.5
Threshold = 0.5
epochs = 1000

# Initialize weights from uniform distribution [0, 1)
W_i = np.random.default_rng().uniform(0, 1, (11, 1))
W_ij = np.random.default_rng().uniform(0, 1, (10, N+1))

a = 0.5
b = 0.6
r = 0.4
# Generate the data
X_train, Y_train = generate_data(D, N, a, b, r)
X_test, Y_test = generate_data(D, N, a, b, r)

for _ in range(epochs):
    E = 0
    # SGD with batch size 1
    for i in range(D):
        x = X_train[i].reshape((N+1, 1))
        t = Y_train[i].item()

        # Output of hidden layer
        temp = np.dot(W_ij,x).reshape((10, 1))
        temp = np.array(list(map(sig, temp)), ndmin=2)
        H_i = np.append(temp, [[1]], axis=0)

        # Output of out layer with one node
        o = sig(np.dot(W_i.T, H_i).item())

        # Error
        E += ((t - o)**2)/2

        # Back prop
        delta_W_i = eta * (t - o) * o * (1 - o) * H_i
        delta_W_ij = eta * (t - o) * o * (1 - o) * np.dot((W_i *
    H_i * (1-H_i))[:-1, :], x.T)
        W_i += delta_W_i
        W_ij += delta_W_ij
    # print("Avg Error after epoch: ", E/D)


# Forward prop for testing
temp = np.dot(X_test, W_ij.T)
temp = np.array(list(map(sig, temp)), ndmin=2)
H_di = np.ones((D, 11))
H_di[:, :-1] = temp

# Output converted to label 0/1 depending on the threshold set in
    Threshold variable (0.5)
O = np.array(list(map(lambda x: 1 if x >= Threshold else 0, sig(np.
    dot(H_di, W_i))))).reshape((D, 1))

# Counting number of mislabelings (if sum of true+predicted label
    is 1 then its a mislabelling and if it's 2 or 0 then its a
```

```
    correct labelling) and calculating the accuracy
Accuracy = (1 - (np.count_nonzero((0 + Y_test) == 1)/D)) * 100
print(Accuracy)
```

As we increase the number of epochs, the accuracy of the model increases. For example, for 10 epochs the accuracy is around 65%, for 100 epochs the accuracy is around 80%, and for 1000 epochs the accuracy is around 95%. Learning rate and the threshold at which we convert the output to label also changes the accuracy of the model. I have found that initializing the weights with values from the uniform distribution $[-1, 1]$ works well.

Question 3 Answer

The output can be expressed as the following function,

$$f(x_1, \ldots, x_n) = \begin{cases} 1, & \text{if } \sum x_i < \frac{n}{2}, i = \{1 \ldots n\} \\ 0, & \text{otherwise} \end{cases}$$

The perceptron models the function,

$$\hat{f}(x_1, \ldots, x_n) = threshold(\omega^T X + b)$$

where,

$$threshold(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases}$$

and,

$$X = \begin{bmatrix} x_1 \\ . \\ . \\ x_n \end{bmatrix}, \omega = \begin{bmatrix} \omega_1 \\ . \\ . \\ \omega_n \end{bmatrix} b = \text{bias}$$

If we initialize all the weights $(\omega_1, \ldots, \omega_n)$ as -1 and bias as $\frac{n}{2}$ then the $\hat{f}$ would return 1 if more of x_i have value 0 than value 1, otherwise returns 0.

Question 4 Answer

Here we have our error defined as mean squared error,

$$E = \frac{1}{n} \sum (o_k - y_k)^2 \tag{1}$$

where,

$$y_k = \text{real valued output for } k^{th} \text{ datapoint}$$

$$o_k = \text{artificial neuron output for } k^{th} \text{ datapoint} = e^{net_k} \tag{2}$$

$$net_k = wx_k, \text{where } w = \text{weight}, x = k^{th} \text{ input} \tag{3}$$

and the weight update would be given by the equation

$$w_i \leftarrow w_i + \Delta w$$

$$\Delta w = -\eta \frac{\partial E}{\partial w}$$

and we can rewrite $\frac{\partial E}{\partial w}$ as

$$\frac{\partial E}{\partial w} = \frac{\partial E}{\partial o_k} \frac{\partial o_k}{\partial net_k} \frac{net_k}{\partial w} \tag{4}$$

from (1) we get,

$$\frac{\partial E}{\partial o_k} = \frac{2}{n}(o_k - y_k) \tag{5}$$

from (2) we get,

$$\frac{\partial o_k}{\partial net_k} = o_k \tag{6}$$

from (3) we get,

$$\frac{\partial net_k}{\partial w} = x_k \tag{7}$$

need so sum up for all the x_k,so plugging in these values to (5) we get,

$$\frac{\partial E}{\partial w} = \frac{2}{n} \sum x_k o_k (o_k - y_k)$$

the weight update equation would be,

$$w \leftarrow w - \eta \frac{2}{n} \sum x_k o_k (o_k - y_k)$$

where w = weight, η is the learning rate, n is total number of samples, x_k is the k^{th} input, y_k is the k^{th} real valued output, o_k is the k^{th} output from the artificial neuron.

I am not including the bias weight since the question asks to model an artificial neuron with a single weight.