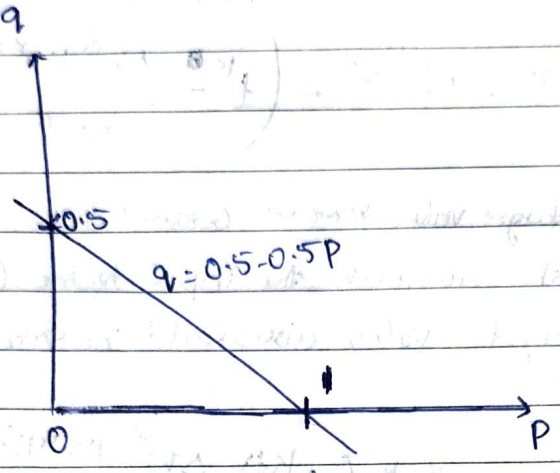
DOMS | Page No.
Date / /

① Given



Given function: $q = 0.5 - 0.5P$

$0.5P + q = 0.5$

From the equation $w_1x + w_2y = b$

Here $w_1 = 0.5$, $w_2 = 1$, $b = 0.5$

These weights can be considered as initial weight

Let Consider three points to Check whether the taken weights or not

→ Consider threshold as 0

$P_1 = (2,3)$, $P_2 = (-1,0)$ $P_3 = (1,-2)$ $P_4 = (1,1)$

If value >= 0, output = 1 (positive class)

If value < 0, output = 0 (negative class)

$$\text{Signum} = \begin{cases} 1 & \text{if } x \geq 0 \\ -1 & \text{if } x < 0 \end{cases}$$

→ $P_1 = (2,3)$

The given function is $0.5p + q - 0.5 = 0$

→ Signum $(0.5p + q - 0.5)$

→ Signum $(0.5(2) + 3 - 0.5)$

→ Signum $(4 - 0.5) = 1$

→ $p_4 = (1, 1)$

The given function is $0.5p + q - 0.5 = 0$

$\Rightarrow$ Signum$(0.5p + q - 0.5)$

$=$ Signum$(0.5(1) + 1 - 0.5)$

$=$ Signum$(1) = 1$

For all values giving the output from the activation belongs to Class 0

→ $p_2 = (-1, 0)$

The given function is $0.5p + q - 0.5 = 0$

$\Rightarrow$ Signum$(0.5p + q - 0.5)$

$\Rightarrow$ Signum$(0.5(-1) + 0 - 0.5)$

$=$ Signum$(-1) = -1$

→ $p_3 = (1, -2)$

The given function is $0.5 + q - 0.5 = 0$

$\Rightarrow$ Signum$(0.5(1) - 2 - 0.5)$

$=$ Signum$(-2) = -1$

For all values it is giving the output from the activation so it belongs to Class 1.

The weight of the perceptron are $w_1 = 0.5$, $w_2 = 1$ and the threshold is 0.5.

(2) To Calculate the gradient Descent for the given Sum of Squared error value $\mathcal{E} = \sum (q_i^o - p_i^o w)^2$ for the input mapping, We need to find the partial derivative of $\mathcal{E}$ with respect to w and update w using gradient descent rule $w \leftarrow w - \eta \, d\mathcal{E}/dw$

$$d\mathcal{E}/dw = \frac{d}{dw}\left[\sum_{i=1}^{D} (q_i^o - p_i^{o\,w})^2\right]$$

$$\Rightarrow \sum_{i=1}^{D} 2\left(q_i^o - p_i^{o\,w}\right) \frac{d}{dw} \cdot p_i^w$$

Let $p_i^w = t$

adding log

$\log p_i^w = \log t$

$w \log p_i^o = \log t$

$\log p_i^o = \frac{1}{t} \frac{dt}{dw}$

$p_i^w \log p_i^o = \frac{dt}{dw}$

$$\Rightarrow \sum_{i=1}^{D} -2\left(q_i^o - p_i^{o\,w}\right) p_i^w \log p_i^o \frac{dt}{dw}$$

$$\Rightarrow \sum_{i=1}^{D} -2\left(q_i^o - p_i^{o\,w}\right) p_i^w \log p_i^o$$

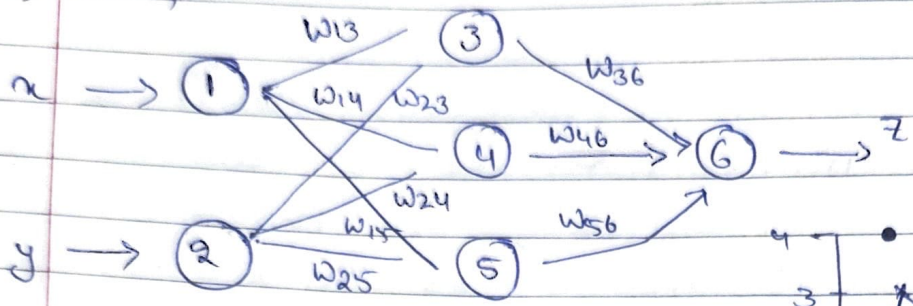
Now let's update w using the gradient descent update

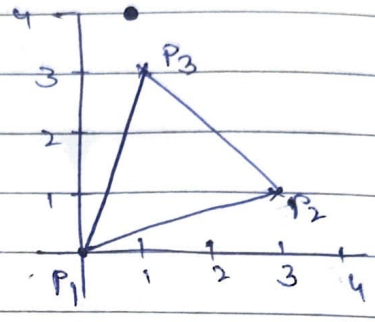
$w \leftarrow w - \eta \, d\mathcal{E}/dw$ where η is learning rate

$$w \leftarrow w + \eta\left[\sum_{i=1}^{D} 2\left(q_i^o - p_i^w\right) p_i^w \log p_i^o\right]$$

This equation allows us to update the weights w based on the given input-output mapping.

Q3) Given,





The points given, $P_1 = (0,0)$

$P_2 = (1,3)$

$P_3 = (3,1)$

Finding the equation of line from the given points,

Point: P_1 and $P_2 \rightarrow (0,0)$, $(1,3)$

$$\text{Slope} = \frac{y_2 - y_1}{x_2 - x_1} \Rightarrow \frac{3-0}{1-0} = 3$$

Equation of line $= y - y_1 = m(x - x_1)$

$$y - 0 = 3(x - 0)$$

$$y = 3x \Rightarrow 3x - y = 0 \rightarrow (1)$$

$\therefore$ Comparing with the equation of weight

$$w_{13}x + w_{23}y + \theta = 0$$

$$w_{13} = 3, \; w_{23} = -1, \; b = 0$$

point P_2 and $P_3 \rightarrow (1,3)$ and $(3,1)$

$$\text{Slope} = \frac{y_2 - y_1}{x_2 - x_1} \Rightarrow \frac{1-3}{3-1} = -1$$

Equation of line $= y - y_1 = m(x - x_1)$

$$y - 3 = -1(x - 1)$$

$$y - 3 = -x + 1$$

$$x + y - 4 = 0 \rightarrow (2)$$

$\therefore$ Comparing with the equation:

$$w_{14}x + w_{24}y + b = 0$$

$$w_{14} = 1, \; w_{24} = 1, \; b = +4$$

DOMS | Page No.
Date / /

point 3 and point1 (3,1) and (0,0) Slope = $\frac{y_2-y_1}{x_2-x_1} \Rightarrow \frac{0-1}{0-3} = \frac{1}{3}$

Equation of line : $y-y_1 = m(x-x_1)$

$\Rightarrow y-1 = \frac{1}{3}(x-3)$

$3y-3 = x-3 \Rightarrow x-3y=0$ - (3)

Comparing with the equation $w_{15}=1, w_{25}=-3, b=0$

Initial weights:

$w_{13}=3, w_{23}=-1, w_{14}=1, w_{24}=1, w_{15}=1, w_{25}=-3$

Let's Consider if the point is inside the triangle as -1 and outsides as +1

target Class = -1 (Inside)

target Class = +1 (outside)

∴ If the point is above the line the equation gives us a positive value we can use this to get the target Class

$$\text{Signum} = \begin{cases} 1 & \text{if } x \geq 0 \rightarrow \text{above the line} \\ -1 & \text{if } x < 0 \rightarrow \text{below line} \end{cases}$$

Let Consider a point (1,4)

Eq (1) → Signum(3(1)-4-0) = Signum(-1) = -1

Eq (2) → Signum(1+4-4) = Signum(1) = 1

Eq (3) → Signum(1-3(4)-0) = Signum(-11) = -1

We know that (1,4) is outside the triangle so target Class should be +1 but for equation (1) and (3) we got -1 so in order to correct the prediction we can change the signs of the weights

modified weights:

$w_{13}=-3, w_{23}=1, w_{14}=1, w_{24}=1, w_{15}=-1, w_{25}=3$

By this weights, the above point will correctly

Outside

Now take another point (1,2)

① → Sign(−3(1) + 2(−1)) = Signum(−3−2) = −1

Signum(1(1) + 2(1) − 4) = Signum(2−4) = −1

Signum(−1(1) + 2(3) = Signum(−1+6) = +1

Here we need to change the polarity again so we cannot do this for every point to overcom this we need to use a bias/threshold at the end.

The weight

$w_{13} = -3, w_{23} = 1, w_{14} = 1, w_{24} = 1, w_{15} = 1, w_{25} = -3$

Assuming the weights

$w_{36} = 1, w_{46} = 1, w_{56} = 1, Tb = -2$

(3,3), (4,1), (2,5) — +1 ; (2,2), (1.3, 1.3), — −1

① (3,3) →

① → Sign(−3x + y) = Signum(−3(3)+3) = −1

② Signum(x+y−4) = Signum(6−4) = +1

③ Signum(x−3y) = Signum(3−9) = −1

Z = Signum(1/(W36) + 1)

Z = Signum(①(W36) + ②(W46) + ③(W56) + 2

= Signum(−1(1) + 1(+1) + −1(1)) + 2) = Signum((−1+1−1)+2) = +1

② (4,1)

① → Sign(−3(4) + 1) = −1

② = Sign(4+1−4) = +1

③ = Signum(4 − 3(1) = +1

= Sign((−1 + 1 + 1) + 2) = +1

(3) (2,5)

① → Signum(-6+5) = -1

② → Signum(7-4) = +1

③ → Signum(2-3(5) = -1

z = Signum((-1+1-1)+2) = +1

For target Class = +1

(2,2)

① → Signum(-6+2) = -1

② → Signum(0) = +1

③ → Signum(2-6) = -1

z = Signum((-1+1-1)+2) = +1

(1.3, 1.3)

① → Signum(-3(1.3)+(1.3)) => -1

→ Signum(1.3+1.3-4) => -1

→ Signum(1.3-3(1.3) => -1

z = Signum((-1-1-1)+2) = -1

These weight will satisfy the conditions.

(4) Given,

$$\Delta w^{(k)} = \eta (t^{(k)} - w^k x^k) \frac{x^k}{\|x^k\|^2} \rightarrow (1)$$

where,

x^k = input vector

t^k = target at iteration k

η = positive number

we also know that,

$$\Delta w^k = w^{k+1} - w^k \rightarrow (2)$$

The difference in the weight after updating the weight is Δw^k

$$\frac{\Delta w^k \cdot \|x^k\|^2}{x^k} = \eta (t^k - w^k x^k)$$

$$\frac{\Delta w^k \cdot \|x^k\|^2}{\eta x^k} = t^k - w^k x^k$$

$$w^k - x^k = t^k - \frac{\Delta w^k \cdot \|x^k\|^2}{\eta x^k}$$

At the (k+1) iteration substituting that equation (1)

$$\Delta w^{k+1} = \eta (t^{k+1} - w^{k+1} x^{k+1}) \frac{x^{k+1}}{\|x^{k+1}\|^2}$$

In the question as mentioned,

Same input vector x^k is presented at iteration (k+1)

$$x^k = x^{k+1}$$

$$\Delta w^{k+1} = \eta (t^{k+1} - w^{k+1} x^k) \frac{x^k}{\|x^k\|^2}$$

$$\Delta w^{k+1} = \eta (t^{k+1} - (\Delta w^k + w^k) x^k) \frac{x^k}{\|x^k\|^2}$$

$$\Delta w^{k+1} = \eta \left(t^{k+1} - \Delta w^k x^k w^k x^k \right) \frac{x^k}{\|x^k\|^2}$$

$$\Delta w^{k+1} = \eta \left(t^{k+1} - \Delta w^k x^k - \left(t^k - \frac{\Delta w^k \cdot \|x^k\|^2}{\eta x^k} \right) \right) \frac{x^k}{\|x^k\|^2}$$

we can assume that target value remains constant

as mentioned in the question the input to the (k+1) is same as k so that target value also should be same

$$t^{k+1} = t^k$$

$$\Delta w^{k+1} = \eta \left(t^{k+1} - \Delta w^k x^k - \left(t^{k+1} - \frac{\Delta w^k \cdot \|x^k\|^2}{\eta x^k} \right) \right) \frac{x^k}{\|x^k\|^2}$$

$$= \eta \left(t^{k+1} - \Delta w^k x^k - t^{k+1} + \frac{\Delta w^k \cdot \|x^k\|^2}{\eta x^k} \right) \frac{x^k}{\|x^k\|^2}$$

$$= \eta \left(\Delta w^k \cdot \frac{\|x^k\|^2}{\eta x^k} - \Delta w^k x^k \right) \frac{x^k}{\|x^k\|^2}$$

$\therefore \|x\|^2 = \|x\| \cdot \|x\|$

$x x^T = x \cdot x^T$

$$= \eta \left(\frac{\Delta w^k}{\eta} - \frac{\Delta w^k x^k \cdot x^k}{\|x^k\|^2} \right)$$

$$\Delta w^{k+1} = \eta \left[\frac{\Delta w^k}{\eta} - \Delta w^k \right]$$

$$\Delta w^{k+1} = \Delta w^k - \eta \Delta w^k$$

$$\boxed{\Delta w^{k+1} = (1-\eta) \Delta w^k}$$

This is a convergence theorem for the perceptron algorithm. It states that if the data is linearly separable, then the perceptron algorithm will converge in a finite number of steps. The theorem was first proved by A. Novikoff in 1962 .

Statement: The Novikoff convergence theorem states that if we have a linearly separable dataset with two classes, one denoted by "+1" and the other by "-1," and there exists a positive lower bound on the margin (the distance from any training instance to the separating decision boundary) the perceptron learning algorithm with error correction will converge in a finite number of steps. The algorithm will iteratively adjust the weights to fine-tune the decision boundary, aiming to correctly classify all the training instances.

Procedure (Novikoff's algorithm):

1. Start by randomly initializing the weights, resulting in an arbitrary weight vector (denoted as V0).
2. Iterate through the dataset and update the weights for misclassified data points.
3. For each iteration:
    a. If a data point is classified correctly, the weight remains unchanged.
    b. If a data is misclassified, Increase the weight vector by adding a step size i.e., margin multiplied by the features of the misclassified instance.
4. Repeat the iterations until all data points are correctly.
5. If the dataset is linearly separable and satisfies a positive lower bound on the margin, the algorithm will converge in a finite number of iterations.
6. The final weight vector obtained after convergence defines the separating hyperplane that correctly classifies the data points.

Mathematical Proof:

For each data point $\boldsymbol{w_1, w_2, w_3 \ldots\ldots w_N}$ in some Euclidean space, if and only if these points are linearly separable, there exists a hyperplane denoted by $\boldsymbol{y}$ which can satisfactorily separate the two classes. The separation is achieved by satisfying the below equation.

$(\boldsymbol{w_i, y}) > \boldsymbol{\theta} > 0$ $\boldsymbol{i=1,2, \ldots\ldots N}$ _____ Equation (1)

where $\boldsymbol{\theta}$ is a threshold value

The equation 1 signifies that the dot product between the data point and the hyperplane should be greater than the threshold value $\boldsymbol{\theta}$. This condition ensures that the data point falls on the correct side of the hyperplane, enabling effective separation of the two classes.

To achieve a perfect hyperplane using the error correction rule in the context of the perceptron learning algorithm. Let's Start with an initial random vector v_o (arbitrary), which serves as the initial separation for the data points. The vector is modified based on the error correction rule, which is as follows: For correctly classified data points, the previous weight vector ($vn\text{-}1$) is carried forward as the new weight vector (v_n). And for incorrectly classified data points, the new weight vector (v_n) is obtained by adding the weight vector value corresponding to the data point (w_n) to the previous weight vector (v_{n-1}). This error correction process is iterated over all available data points, ranging from 1 to N (the total number of data points).

$$\boldsymbol{v_n} = \{ \boldsymbol{v_{n-1}} \quad \boldsymbol{if}\ (\boldsymbol{w_{in}} \,.\, \boldsymbol{v_{n-1}}) > \boldsymbol{\theta}$$
$$\boldsymbol{v_n} = \{\boldsymbol{v_{n-1}} + \boldsymbol{w_{in}}\ \boldsymbol{if}\ (\boldsymbol{w_{in}} \,.\, \boldsymbol{v_{n-1}}) \leq \boldsymbol{\theta}$$ ____Equation (2)

The condition $(w_n \cdot v_{n-1}) > \boldsymbol{\theta}$ determines whether a data point is correctly classified or not, where $\boldsymbol{\theta}$ is the threshold value. If the dot product is greater than $\boldsymbol{\theta}$, the data point is classified correctly, and the previous weight vector is carried forward.

Otherwise, if the dot product is less than or equal to $\boldsymbol{\theta}$, the data point is misclassified, and the weight vector corresponding to that data point is added to the previous weight vector.

Further the proof states that if we remove few terms from the training sequence for which the condition and ($w_{in}.V_{n-1)}$ ≤ $\boldsymbol{\theta}$ is not satisfied, we are left with a modified training sequence. The terms that don't fulfil the condition are considered inessential for the proof. By removing these inessential terms, we obtain a modified training sequence where the correction or adjustment takes place at every step. This means that at each iteration or term in the modified sequence, adjustments can be made to improve the performance or accuracy of the algorithm.

$\boldsymbol{v_n = v_{n-1} + w_{in}} \quad (\boldsymbol{w_{in}} . \boldsymbol{v_{n-1}}) \leq \boldsymbol{\theta\ for\ each\ n}$____Equation (3)

The variable "n" represents the number of corrections made up to the n-th step. It suggests that the values of "n" can only belong to a finite set of integers, meaning there is a maximum value of "n" beyond which the corrections cannot continue.

So the v_n can be implied as

$\| \boldsymbol{v_n} \|^2 = \boldsymbol{C}n^2$ ____Equation (4)

Where C is a positive constant such that n is sufficiently large

Equation (3) implies that the integer-argument function V_n satisfies a specific difference inequality. The equation can be written as

$$\| \boldsymbol{v_n} \|^2 = \| \boldsymbol{v_{n-1}} \|^2 + 2\ (\boldsymbol{w_{in}} . \boldsymbol{v_{n-1}}) + (\boldsymbol{w_{in}} . \boldsymbol{w_{in}}) \leq 2\boldsymbol{\theta} + \boldsymbol{M}$$

The equation can be rewritten as

$$\| \boldsymbol{v_n} \|^2 - \| \boldsymbol{v_{n-1}} \|^2 = 2\ (\boldsymbol{w_{in}} . \boldsymbol{v_{n-1}}) + (\boldsymbol{w_{in}} . \boldsymbol{w_{in}}) \leq 2\boldsymbol{\theta} + \boldsymbol{M}$$

Where M is $\boldsymbol{max}\ (\boldsymbol{w_{in}} . \boldsymbol{w_{in}})$ for $\boldsymbol{i} = \boldsymbol{1}, \ldots \ldots, \boldsymbol{N}$

The statement suggests that an inequality can be derived based on the fact that the dot product between the vectors $\boldsymbol{v}_{n-1}$ (previous vector) and $\boldsymbol{w}_{in}$ (weight vector) is less than zero ($2\ (\boldsymbol{w_{in}} . \boldsymbol{v_{n-1}}) < \boldsymbol{0}$). This indicates that a misclassification occurred, and an update had to be made to the weight vector.

From this observation, it can be concluded that $\boldsymbol{0 \leq 2\theta + M \leq 1}$, where $\boldsymbol{\theta}$ represents a threshold value and $\boldsymbol{M}$ is the maximum value of ($\boldsymbol{w_{in}} . \boldsymbol{w_{in}}$) among all possible values of i and n.

In [63]:
```
import numpy as np
import pandas as pd
from math import exp

#class
class NeuralNetwork:
    def __init__(self, hidden_nodes, num_classes, learning_rate, num_epochs):
        #initializing the paramaters
        self.hidden_nodes = hidden_nodes
        self.num_classes = num_classes
        self.learning_rate = learning_rate
        self.num_epochs = num_epochs
        self.W1 = None
        self.b1 = None
        self.W2 = None
        self.b2 = None
    #relu function which is used as activation function
    @staticmethod
    def relu_func(x):
        return np.maximum(0, x)
    #derivative of relu function
    @staticmethod
    def relu_drv(x):
        return np.where(x > 0, 1, 0)
    #softmax function used to convert the output of NN to categorical data
    @staticmethod
    def softmax_func(x):
        exps = np.exp(x - np.max(x, axis=1, keepdims=True))
        return exps / np.sum(exps, axis=1, keepdims=True)

    @staticmethod
    def one_hot_enc(arr):
        argmax_labels = np.argmax(arr, axis=1)
        encoded_arr = pd.get_dummies(argmax_labels)
        return encoded_arr.values


    #loss function to determine the error
    @staticmethod
    def cross_ent(original_label, predict_label):
        m = original_label.shape[0]  # Number of training examples
        cost = - np.sum(np.multiply(original_label, np.log(predict_label))) /
        cost = np.squeeze(cost)
        return cost

    #updating the weight after performing back propagation and getting gradie
    def update_weights(self, weights, costs, train_data):
        updated_weights = []
        self.features = train_data.shape[1]
        self.samples = len(train_data)

        for weight, cost in zip(weights, costs):
            #checking whether the dimension match and perfoming the updation.
            # w = w - n*dw
            if cost.shape == (self.features, self.hidden_nodes):
                updated_weight = weight - self.learning_rate * cost
            elif cost.shape == (self.hidden_nodes,):
```

```
                updated_weight = weight - self.learning_rate * cost
            elif cost.shape == (self.samples, self.hidden_nodes):
                updated_weight = weight - self.learning_rate * cost.sum(axis=
            elif cost.shape == (self.samples, self.num_classes):
                updated_weight = weight - self.learning_rate * cost.sum(axis=
            else:
                updated_weight = weight
            #updated weights
            updated_weights.append(updated_weight)

        return updated_weights


    #intializing random values for weights and bias
    def initialize_weights(self, train_data):
        np.random.seed(0)
        self.features = train_data.shape[1]
        self.W1 = np.random.randn(self.features, self.hidden_nodes)
        self.b1 = np.random.randn(self.hidden_nodes)
        self.W2 = np.random.randn(self.hidden_nodes, self.num_classes)
        self.b2 = np.random.randn(self.num_classes)
    #performing forward propagation
    def forward_prop(self, inp_data):
        #z1=W1*x+b1
        Z1 = np.dot(inp_data, self.W1) + self.b1
        #A1 = relu(Z1)
        A1 = self.relu_func(Z1)
        #z2=W2*x+b2
        Z2 = np.dot(A1, self.W2) + self.b2
        #A2 = relu(Z2)
        A2 = self.softmax_func(Z2)
        return A2, A1, Z1
    #Backpropagation
    def backward_prop(self, X_train, y_train, net_hidden, act_hidden, weight_
        m = X_train.shape[0]  # Number of training examples
        #finding gradient descent values
        dZ2 = act_output - y_train
        dW2 = (1 / m) * np.dot(act_hidden.T, dZ2)
        db2 = dZ2
        dZ1 = np.multiply(np.dot(dZ2, weight_output.T), self.relu_drv(net_hid
        dW1 = np.dot(X_train.T, self.relu_drv(net_hidden) * dZ1)
        db1 = (dZ1 * self.relu_drv(net_hidden))
        return dW2, db2, dW1, db1


    #accuracy
    def accuracy(self, y_true, y_pred):
        if len(y_true) != len(y_pred):
            print('Size does not match.')
            return 0

        num_samples = len(y_true)
        num_correct = sum(np.all(true_label == pred_label) for true_label, pr
        accuracy = num_correct / num_samples
        #the number of correct predictions divided by the total number of pre
        return accuracy
```

```
    def fit(self, train_data, train_labels, val_data, val_labels):
        #initializing the weights and bias
        self.initialize_weights(train_data)

        #looping untill we the better accuracy
        for epoch in range(1, self.num_epochs + 1):
            #performing forward propagation and getting ouptut values of acti
            act_output, act_hidden, net_hidden = self.forward_prop(train_data
            #performing the backward propagation and getting the gradient val
            wto_grad, bo_grad, wth_grad, bh_grad = self.backward_prop(train_d
                                                   self.W2, act_output)
            #updating the weight using gradient descent
            self.W1, self.b1, self.W2, self.b2 = self.update_weights([self.W1
                                                 [wth_grad, bh_grad, 
                                                 train_data)


            #calculating loss values
            Loss = self.cross_ent(train_labels, act_output)

            #predicting the labels for train and val data
            self.y_pred, _, _ = self.forward_prop(val_data)
            # One hot encoding the prediction
            self.y_pred_enc = self.one_hot_enc(self.y_pred)
            # Calculating the accuracy
            val_acc = self.accuracy(val_labels, self.y_pred_enc)
            train_acc = self.accuracy(train_labels, self.one_hot_enc(act_outp

            if epoch % 10 == 0:
                print('epoch =', epoch, 'Loss function value:', Loss, 'Traini



#spliting the data into train and Validation
def training_data(df, encoded_labels, train_ratio, val_ratio):
    np.random.seed(42)

    indices = np.arange(len(df))
    np.random.shuffle(indices)

    shuffled_df = df.iloc[indices]
    shuffled_labels = encoded_labels.iloc[indices]

    num_examples = len(df)
    num_train = int(train_ratio * num_examples)
    num_val = int(val_ratio * num_examples)

    train_data = shuffled_df[:num_train]
    val_data = shuffled_df[num_train:num_train + num_val]
    train_labels = shuffled_labels[:num_train]
    val_labels = shuffled_labels[num_train:num_train + num_val]

    return train_data, train_labels, val_data, val_labels

# Usage example:
np.random.seed(0)
```

```
values = []
for i in range(784):
    values.append(i)
df = pd.read_csv('C:/Users/saiko/OneDrive/Desktop/657/Assign-1/train_data.csv
label = pd.read_csv('C:/Users/saiko/OneDrive/Desktop/657/Assign-1/train_label
                    names=[i for i in range(0, 4)])
encoded_labels = pd.get_dummies(label)
train_data, train_labels, val_data, val_labels = training_data(df, encoded_la
train_labels = train_labels.to_numpy()
val_labels = val_labels.to_numpy()

hiddenlyr_nodes = 35
num_classes = 4
learning_rate = 0.0001
num_epoch = 200
#calling the model

nn = NeuralNetwork(hiddenlyr_nodes, num_classes, learning_rate, num_epoch)
nn.fit(train_data, train_labels, val_data, val_labels)
```

```
C:\Users\saiko\AppData\Local\Temp\ipykernel_35112\2451931313.py:104: Depreca
tionWarning: elementwise comparison failed; this will raise an error in the
future.
  num_correct = sum(np.all(true_label == pred_label) for true_label, pred_la
bel in zip(y_true, y_pred))
```

```
epoch = 10 Loss function value: 0.7731534068008766 Training Accuracy: 0.8425
491087209008 Validation Accuracy: 0.8381818181818181
epoch = 20 Loss function value: 0.5267145567140168 Training Accuracy: 0.8736
050093420189 Validation Accuracy: 0.8660606060606061
epoch = 30 Loss function value: 0.43510915631697317 Training Accuracy: 0.890
6731303337878 Validation Accuracy: 0.8840404040404041
epoch = 40 Loss function value: 0.38282893444267585 Training Accuracy: 0.900
4191284148866 Validation Accuracy: 0.8923232323232323
epoch = 50 Loss function value: 0.34627712769381036 Training Accuracy: 0.911
225571883048 Validation Accuracy: 0.8985858585858586
epoch = 60 Loss function value: 0.3184621987069905 Training Accuracy: 0.9189
011765894056 Validation Accuracy: 0.9066666666666666
epoch = 70 Loss function value: 0.2988521216466954 Training Accuracy: 0.9229
914659395041 Validation Accuracy: 0.9109090909090909
epoch = 80 Loss function value: 0.2843451272495522 Training Accuracy: 0.9261
223047013079 Validation Accuracy: 0.915959595959596
epoch = 90 Loss function value: 0.2720774114838826 Training Accuracy: 0.9288
491642680402 Validation Accuracy: 0.9175757575757576
epoch = 100 Loss function value: 0.2616527332559589 Training Accuracy: 0.930
3640862495581 Validation Accuracy: 0.9204040404040404
epoch = 110 Loss function value: 0.2523413138948342 Training Accuracy: 0.931
980003029844 Validation Accuracy: 0.9234343434343434
epoch = 120 Loss function value: 0.24307109190065562 Training Accuracy: 0.93
43028834015048 Validation Accuracy: 0.9252525252525252
epoch = 130 Loss function value: 0.23464230324907612 Training Accuracy: 0.93
61207897793263 Validation Accuracy: 0.9276767676767677
epoch = 140 Loss function value: 0.2272896915890379 Training Accuracy: 0.937
3327273645408 Validation Accuracy: 0.9290909090909091
epoch = 150 Loss function value: 0.22002737831469646 Training Accuracy: 0.93
92011311417462 Validation Accuracy: 0.9317171717171717
epoch = 160 Loss function value: 0.21311108820756633 Training Accuracy: 0.94
12210271171034 Validation Accuracy: 0.9345454545454546
epoch = 170 Loss function value: 0.20706272654639196 Training Accuracy: 0.94
233196990355 Validation Accuracy: 0.934949494949495
epoch = 180 Loss function value: 0.20153796354238518 Training Accuracy: 0.94
31904256930768 Validation Accuracy: 0.9365656565656566
epoch = 190 Loss function value: 0.19673409464103342 Training Accuracy: 0.94
39983840832197 Validation Accuracy: 0.937979797979798
epoch = 200 Loss function value: 0.19255756366551224 Training Accuracy: 0.94
49073372721305 Validation Accuracy: 0.9383838383838384
```

Test data should be given here

In [64]:
```
#test data prediction
test_data = pd.read_csv("test data location goes here")
def predict_labels(test_data, neural_network):
    A2, _, _ = neural_network.forward_prop(test_data)
    labels = np.argmax(A2, axis=1)  # Get the index of the maximum activation
    encoded_labels = np.eye(neural_network.num_classes)[labels]  # One-hot en
    return encoded_labels
```

In [71]:
```
# Assuming you have loaded the test dataset into 'test_data'
predicted_labels = predict_labels(test_data, nn)
predicted_labels = pd.DataFrame(predicted_labels)
predicted_labels
```

Out[71]:

	0	1	2	3
0	0.0	1.0	0.0	0.0
1	1.0	0.0	0.0	0.0
2	0.0	0.0	1.0	0.0
3	1.0	0.0	0.0	0.0
4	0.0	0.0	1.0	0.0
...	...	...	...	...
4945	0.0	0.0	1.0	0.0
4946	1.0	0.0	0.0	0.0
4947	0.0	0.0	1.0	0.0
4948	0.0	0.0	0.0	1.0
4949	0.0	0.0	0.0	1.0

4950 rows × 4 columns

In [ ]:
```
```