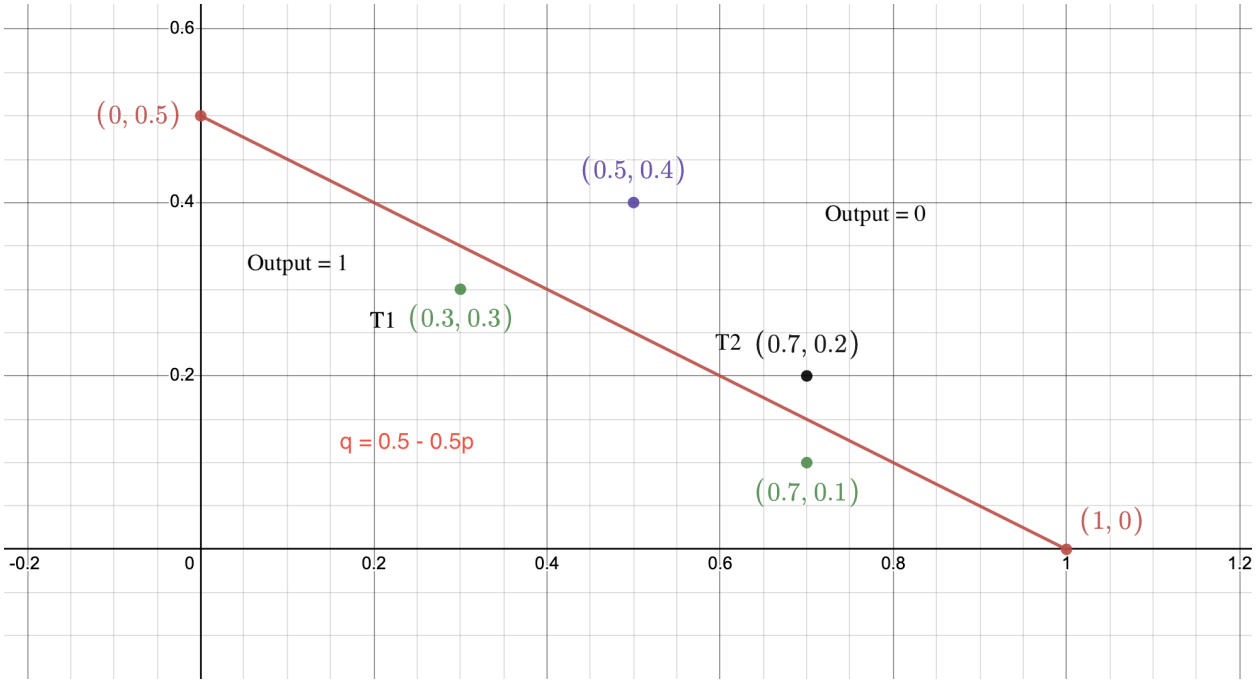
Problem 1



Since we have the line that separates the data we can use the equation to find out w1, w2, and the threshold -

q = 0.5 - 0.5*p
0.5*p + q = 0.5
w1x + w2x2 - θ >= threshold

We get **w1** = 0.5, **w2** = 1, **threshold** = 0.5

Activation function f(x) = 0, x >= 0.5
1, x < 0.5

Running test T1 -

T1 = (0.3, 0,3)
Expected output = 1
=> f (w1x1 + w2x2)
=> f (0.5*0.3 + 1*0.3)
=> f(0.45)
=> 1, equal to the expected output

Running test T2 -

T2 = (0.7, 0,2)
Expected output = 0
=> f (w1x1 + w2x2)

=> f (0.5*0.7 + 1*0.2)
=> f(0.55)
=> 0, equal to the expected output

Hence the perceptron classifies the input points correctly to the defined classes in the problem using weights w1 = 0.5, w2 = 1, and threshold=0.5 and given f(x) activation function.

Ans-2) $\{(p_i, q_i)\}_{i=1}^{D}$

$q = 0.5 - 0.5p$

	$i=1$	$i=2$
p	0	1
q	0.5	0

—— (1)

To calculate gradient descent

$$w \leftarrow w - \eta \frac{\partial E}{\partial w}$$

Given → $E = \sum_{i=1}^{D} (q_i - p_i^w)^2$

For $i=1, i=2$

$$E = \sum_{i=1}^{2} (q_i - p_i^w)^2$$

$$E = (q_1 - p_1^w)^2 + (q_2 - p_2^w)^2$$

Substitute value of q_1 & p_1 from (1)

$$E = (0.5 - 0^w)^2 + (0 - 1^w)^2$$

$$= (0.5)^2 + (0^w)^2 - 2(0.5)(0^w) + (0)^2 + (1^w)^2 - 2(0)(1^w)$$

$$= (0.5)^2 + 0 - 0 + 0 + 1^{2w} - 0$$

Now take derivative w.r.t. w

$$\frac{\partial \mathcal{E}}{\partial w} = \frac{\partial}{\partial w}(0.5)^2 + \frac{\partial}{\partial w}(1)^{2w}$$

$$= 0 + 1^{2w} \ln(1)$$

As $\ln(1) = 0$

$$\frac{\partial \mathcal{E}}{\partial w} = 0 + 0$$

$$\frac{\partial \mathcal{E}}{\partial w} = 0$$

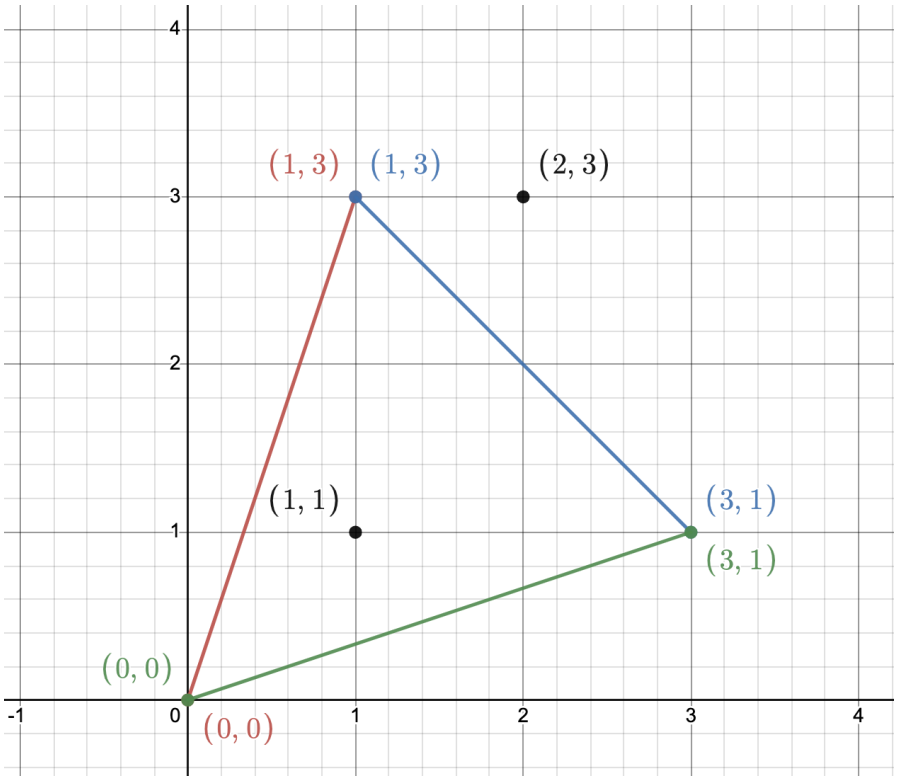
$$w \leftarrow w - \eta(0)$$

$$w \leftarrow w$$

Since the gradient is 0, there is no update to the weights w.
The weights w remain unchanged.

Thus the weights w will not change regardless of learning rate η or the number of iterations

Problem 3



We have 3 lines 0
L1 -> y = 3x
L2 -> y = -x + 4
L3 -> y = $^{1}/_{3}$x

w1x1 + w2x2 - θ = 0 eq (1)

We can get the value of weights by comparing eq (1) with the line equations -

Eq 1 => -3x + y = 0 implies w13 = -3, w23 = 1, θ = 0
Eq 2 => x + y - 4 = 0 implies w14 = 1, w24 = 1, θ = 4
Eq 3 => $^{-1}/_{3}$x + y = 0 implies w15 = -$^{1}/_{3}$, w25 = 1, θ = 0

Threshold function being used in the network -
f(x) = 1, for x>=0
f(x) = -1, for x < 0

L1, L2, and L3 act as linear separators. We define a class inside the triangle as 1 and outside as 0.

For line L1, points above the line in the direction of the +ve y-axis return a positive result if put in Eq 1 on the left-hand side and negative for points below the line in the direction of -ve y-axis. The class we are interested in for line L1 is below the line.
So if we put **w36 = -1**, then w36*(output of node 3) = 1 (for points below L1 in the plane).

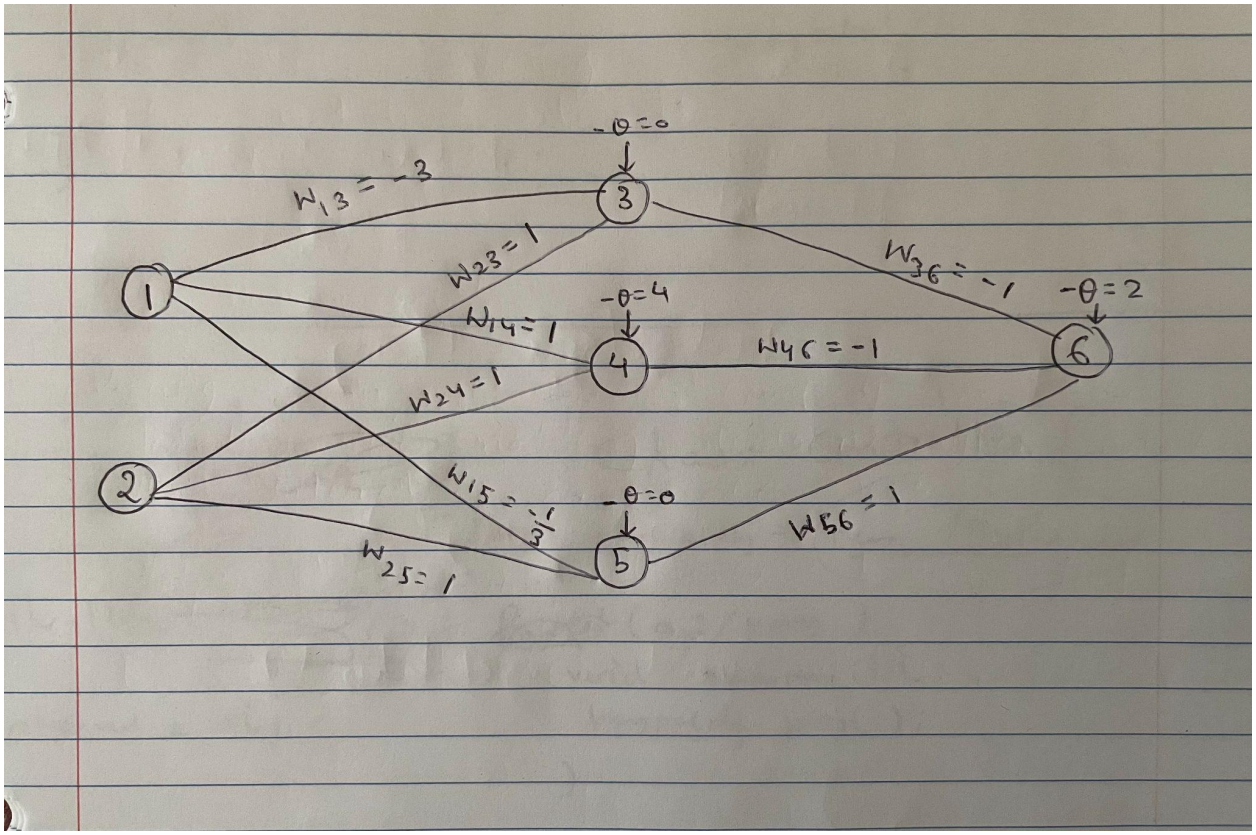
For line L2, we apply the above logic again since we are interested in points below the line L2 so we put **w46 = -1**, then w46*(output of node 4) =1 (for points below L2 in the plane).

For line L3, we apply the reverse. Because this time we are interested in the points lying above the line as that is the class that contributes to the triangle. Hence **w56 = 1**, then w56*(output of node 5) = 1 (for points above L3 in the plane).

Finally for node6, if weighted sum will be 3 when the input point lies inside the line. So we can use **θ6 = 2** so that we only output 1 if all three nodes fired under the defined conditions.

If any one of the nodes did not fire
=> the weighted sum will always be less than 3
=> and subtraction by θ6 = 2 will always result in a value < 1
=> Point lies outside the triangle

Hence following is the weight and bias configuration in the network -



Point (1,1) - lies inside the triangle -

Node3 -

=> f(w13*x + w23*y)
=> f((-3*1) + (1*1))
=> f(-2)
=> -1

Node4 -

=> f(w14*x + w24*y - θ)
=> f((1*1) + (1*1) - 4)
=> f(-2)
=> -1

Node5 -

=> f(w15*x + w25*y)
=> f((-⅓*1) + (1*1))
=> f(⅔)
=> 1

Node6

=> f(Node3*w36 + Node4*w46 + Node5*w56 - θ)
=> f((-1*-1) + (-1*-1) + (1*1) - 2)
=> f(1)
=> 1 (equal to the expected output since the point lies inside the triangle)

Point (2,3) - lies inside outside the triangle -

Node3 -

=> f(w13*x + w23*y)
=> f((-3*2) + (1*3))
=> f(-3)
=> -1

Node4 -

=> f(w14*x + w24*y - θ)
=> f((1*2) + (1*3) - 4)
=> f(1)
=> 1

Node5 -

=> f(w15*x + w25*y)
=> f((-⅓*2) + (1*3))
=> f(⅓)
=> 1

Node6

=> f(Node3*w36 + Node4*w46 + Node5*w56 - θ)
=> f((-1*-1) + (1*-1) + (1*1) - 2)
=> f(-1)
=> -1 (equal to the expected output since the point lies outside the triangle)

Problem 4-

Widrow Hoff learning rule -

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

We need to prove - $\Delta W^{(k+1)} = (1 - \eta) \Delta W^{(k)}$

Given to us - $x^{(k)}$ presented at k-th iteration is equal to the one presented on the (k+1)th iteration.

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

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

(1) - (2)

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

$$- W^{(k)} + \eta \left(t^{(k)} - W^{(k)} x^{(k)} \right) \frac{x^{(k)}}{\|x^{k}\|^2}$$

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

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

} since input vector is same hence the output (expected) will be same.

Replacing $t^{(k+1)}$ with $t^{(k)}$ and $x^{(k+1)}$ with $x^{(k)}$, and rearranging

$$W^{k+2} - W^{k+1} = W^{k+1} - W^k + \eta \frac{x^k}{\|x^k\|^2} \left[t^k - W^{k+1} x^k - t^k + W^k x^k \right]$$

on simplifying

$$W^{k+2} - W^{k+1} = W^{k+1} - W^k + \eta \frac{x^k}{\|x^k\|^2} \left[W^k x^k - W^{k+1} x^k \right]$$

taking $(W^{k+1} - W^k)$ common on the right hand side

$$W^{k+2} - W^{k+1} = W^{k+1} - W^k - \eta \frac{x^k}{\|x^k\|^2} \left[(W^{k+1} - W^k) x^k \right]$$

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

$\frac{x^k \cdot x^k}{\|x^k\|^2}$ simplifies to 1

we can write the above equation as →

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

Novikoff on Perceptron

The paper discusses a theorem related to the training process of a perceptron. It states that for a given type of perceptron and a set of incoming signals divided into two classes, there exists a satisfactory assignment of output weights that will result in correct classifications for each class. The process of recursively readjusting the weights using error correction will eventually converge on this satisfactory assignment, as long as it exists.

1. To prove this it discusses a theorem related to the training process of a perceptron. The key assumption in this theorem is that there exists a vector y (equation 1)

$(w_i, y) > \theta > 0$ for i = 1, ..., N. —---------equation 1

a)The sequence w represents the training sequence.
b)The vector y represents the satisfactory assignment of associator outputs,
which we assume to exist, such that the dot product between each vector in the training sequence, denoted as W1, ..., WN, and y is greater than zero. This assumption is crucial for the theorem to hold.

2. The training process involves a recursive adjustment of weight vectors using error correction. The goal is to converge on a satisfactory assignment of weight vectors that will result in correct classifications for each class. The correction procedure is guided by the condition

$(V_{n-1}, w) <= \theta$ —----------(equation 2),

which ensures that the corrections made at each step do not exceed a certain limit represented by the constant θ.

3. The theorem claims that the sequence of weight vectors, denoted as vn, is convergent. This means that for some index m, we have $v_m = v_{m+1} = v_{m+2} = ... \approx v$.
In other words, the weight vectors reach a satisfactory state where further adjustments are unnecessary.

4. To simplify the analysis, the proof focuses only on the indices n where the next term Vn+1 is different from the current term Vn.

$V_n = V_{n+1} + w_{in}$

It assumes that the current term Vn can be obtained by adding the correction vector w to the next term Vn+1, represented by the equation $V_n = V_{n+1} + w$.

5. The proof argues that the variable n, which counts the number of corrections up to the nth step, can only take values from a finite set of integers.
This means that equations (1) and (2) cannot hold simultaneously for all values of n.

6. It further suggests that either aligning the vectors with y requires larger corrections (violating equation 2) or limiting the corrections prevents the vectors from aligning with y (violating equation 1).

7.(Continuous case), The paper also introduces a continuous version of the theorem, which deals with smooth curves described by vector functions. It states that a similar conclusion holds in the continuous case: there exists a finite interval where certain conditions are compatible, representing the convergence of the curve.
It explains this by stating that in the continuous case, when vectors change smoothly over time, the requirement for infinitely many vectors to satisfy (1) is stronger than just requiring that the dot product between a vector and y is greater than zero for a specific time interval.
It mentions that the left-hand side of the condition (1) can be positive for each value of t, but it doesn't have to be bounded away from zero. This means that the dot product between a vector and y can be positive but can approach zero without reaching a specific limit.

8. (Geometric Interpretation) The proof then goes on to mention the geometric interpretation of equation 1 in terms of proper cones.
Equation 1 states that there exists a vector y such that the dot product between each vector in the training sequence and y is greater than zero.
This condition is precisely equivalent to the fact that the polyhedral cone C, generated by W1,..., and WN, is a proper cone. A proper cone never contains both a vector and its negation, except for the zero vector. The existence of a positive vector y satisfying equation 1 is neither stronger nor weaker than the existence of a zero vector y satisfying it.

In summary, the theorem addresses the convergence of weight vectors in a perceptron algorithm. It relies on the assumption of a vector y that satisfies certain dot product conditions (equation 1). The algorithm iteratively adjusts the weight vectors using error correction, guided by a limit specified by equation 2. The text explores the compatibility of these conditions, the finite nature of corrections, and the geometric interpretation in terms of proper cones.

Problem6

June 20, 2023

[31]:
```
import numpy as np
import pandas as pd
from IPython.display import display, Markdown

x = pd.read_csv('/Users/ankushdubey/Desktop/train_data.csv', header=None)
y = pd.read_csv("/Users/ankushdubey/Desktop/train_labels.csv", header=None)
data = np.append(x,y,axis=1)
print("Dataset dimension: ", data.shape)

np.random.shuffle(data)
r,c = data.shape
split = 0.2
train_set = data[int(r*split):] # 80% of the whole dataset
test_set = data[:int(r*split)] # 20% of the whole dataset

train_x = train_set[:, :-4]
train_y = train_set[:, -4:]

test_x = test_set[:, :-4]
test_y = test_set[:, -4:]

training_accuracies = []
loss_variation = []

print("Training Input Data Shape: ", train_x.shape)
print("Training Expected Data Shape: ", train_y.shape)
print("Test Input Data Shape: ",test_x.shape)
print("Test Expected Data Shape: ",test_y.shape)

class MLP:
    def __init__(self, input_dim, hidden_dim, output_dim):
        '''
            Initializating network architecture
            input_dim = 784
            Hidden_dim = 332
            Output dim = 4
        '''
```

```
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.output_dim = output_dim

        '''
            Weights between input and hidden layer
            Dimensions (784 X 64)
        '''
        self.weights1 = 0.013*np.random.randn(self.input_dim, self.hidden_dim)
        self.bias1 = np.zeros((1, self.hidden_dim))

        '''
            Weights between hidden and output layer
            Dimensions (64 X 4)
        '''
        self.weights2 = 0.037*np.random.randn(self.hidden_dim, self.output_dim)
        self.bias2 = np.zeros((1, self.output_dim))

    def forward(self, X):
        '''
            Calculate the weighted sum output of the hidden layer
            Dimensions (19804 X 784) . (784 X 64) = (19804 X 64)
        '''
        self.hidden_layer = np.dot(X, self.weights1) + self.bias1

        '''Activate the weighted sum output from the hidden layer nodes'''
        self.hidden_activation = self.sigmoid(self.hidden_layer)

        '''
            Activate output from the hiddedn layer feeded to the output layer
            Dimensions (19804 X 64) . (64 X 4) = (19804 X 4)
        '''
        self.output_layer = np.dot(self.hidden_activation, self.weights2) +␣
↪self.bias2

        '''Activated result from output layer'''
        self.output_activation = self.softmax(self.output_layer)

        return self.output_activation

    def backward(self, X, y, learning_rate):
        m = X.shape[0]

        '''
            Calculating error gradients
        '''
        '''Partial derivative of error function w.r.t network output'''
```

```
        output_error = (y - self.output_activation)
        hidden_error = np.dot(output_error, self.weights2.T) * self.
↪sigmoid_derivative(self.hidden_layer)

        '''
            output error X d(output_layer)/d(output_layer_weights)
            partial derivative equivalent to output_from_previous_layer
            (in this case self.hidden_activation.T)
        '''
        weights2_gradient = np.dot(self.hidden_activation.T, output_error)/m
        bias2_gradient = np.sum(output_error)/m

        '''
            hidden_error X d(hidden_layer)/d(hidden_layer_weights)
            partial derivative equivalent to output_from_previous_layer
            (in this case X.T)
        '''
        weights1_gradient = np.dot(X.T, hidden_error) / m
        bias1_gradient = np.sum(hidden_error) / m

        '''Update weights and biases for output layer'''
        self.weights2 += learning_rate * weights2_gradient
        self.bias2 -= learning_rate * bias2_gradient
        '''Update weights and biases for hidden layer'''
        self.weights1 += learning_rate * weights1_gradient
        self.bias1 -= learning_rate * bias1_gradient

    def train(self, X, y, epochs, learning_rate):
        for epoch in range(epochs):
            '''Training step'''
            output = self.forward(X)
            '''Backwards propogation'''
            self.backward(X, y, learning_rate)

            '''Network metrics after every epoch'''
            training_predictions = np.argmax(output, axis=1, keepdims=True)
            one_hot_encoded_predictions = np.zeros((training_predictions.
↪shape[0], 4))

            for i, value in enumerate(training_predictions):
                one_hot_encoded_predictions[i, value] = 1

            count_diff = np.sum(np.any(one_hot_encoded_predictions != y,␣
↪axis=1))
            training_accuracies.append((1.0 - count_diff/X.shape[0])*100)

            loss = self.cross_entropy_loss(y, output)
```

```
            loss_variation.append(loss)
            if epoch % 20 == 0:
                print(f"{epoch}th epoch Loss = {loss}")
                print(f"{epoch}th epoch Accuracy = {(1.0 - count_diff/X.
↪shape[0])*100}%")


    def predict(self, X):
        output = self.forward(X)
        predictions = np.argmax(output, axis=1)

        one_hot_encoded_predictions = np.zeros((len(predictions), 4))

        for i, value in enumerate(predictions):
            one_hot_encoded_predictions[i, value] = 1

        return one_hot_encoded_predictions

    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def sigmoid_derivative(self, x):
        return self.sigmoid(x) * (1 - self.sigmoid(x))

    def softmax(self, x):
        exp_scores = np.exp(x - np.max(x, axis=1, keepdims=True))
        return exp_scores / np.sum(exp_scores, axis=1, keepdims=True)

    def cross_entropy_loss(self, y_true, y_pred):
        epsilon = 1e-10
        y_pred = np.clip(y_pred, epsilon, 1. - epsilon)
        loss = -np.sum(y_true * np.log(y_pred)) / y_true.shape[0]
        return loss
```

```
Dataset dimension:  (24754, 788)
Training Input Data Shape:  (19804, 784)
Training Expected Data Shape:  (19804, 4)
Test Input Data Shape:  (4950, 784)
Test Expected Data Shape:  (4950, 4)
```

[32]:
```
import matplotlib.pyplot as plt

def plot_graphs_accuracy(length, step):
    start_value = 1
    my_list = [start_value + i * step for i in range(length)]
    epochs = my_list
    accuracies = training_accuracies
```

```
    fig, ax = plt.subplots()
    ax.plot(epochs, accuracies)
    ax.set_xlabel('Epochs')
    ax.set_ylabel('Accuracy')
    ax.set_title('Accuracy Over Epochs during training')


def plot_graphs_loss(length, step):
    start_value = 1
    my_list = [start_value + i * step for i in range(length)]
    epochs = my_list
    fig, ax = plt.subplots()
    ax.plot(epochs, loss_variation)
    ax.set_xlabel('Epochs')
    ax.set_ylabel('Loss')
    ax.set_title('Loss Over Epochs during training')


def plot_graph(length, step):
    plot_graphs_accuracy(length, step)
    plot_graphs_loss(length, step)
    plt.tight_layout()
    plt.show()
```

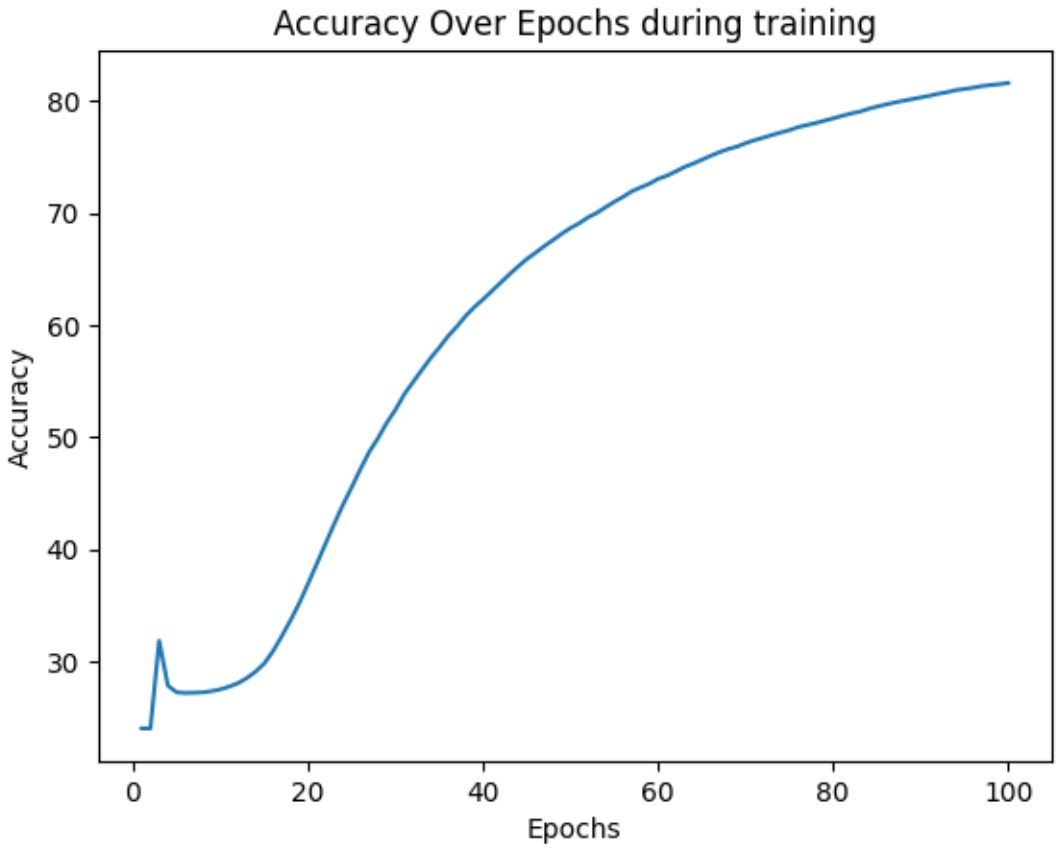
[33]:
```
'''Create MLP object'''
number_of_input_nodes = 784
number_of_hidden_layer_nodes = 720
number_of_output_nodes = 4
NN = MLP(number_of_input_nodes, number_of_hidden_layer_nodes,␣
 ↪number_of_output_nodes)
number_of_epochs = 100
learning_rate = 0.01
'''Start training'''
NN.train(train_x, train_y, number_of_epochs, learning_rate)
```

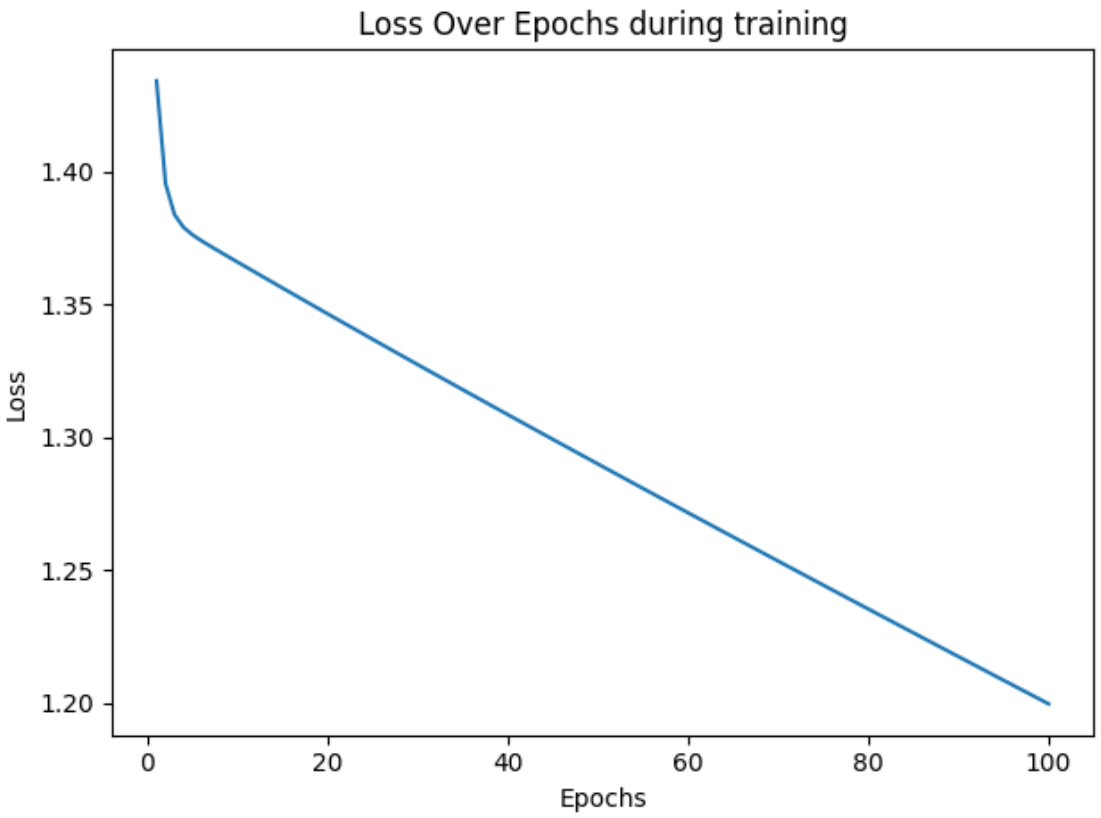
```
0th epoch Loss = 1.4340360474879301
0th epoch Accuracy = 23.9850535245405%
20th epoch Loss = 1.344462391791473
20th epoch Accuracy = 38.65885679660674%
40th epoch Loss = 1.3066033039832612
40th epoch Accuracy = 63.01757220763482%
60th epoch Loss = 1.269739630130581
60th epoch Accuracy = 73.31347202585337%
80th epoch Loss = 1.233586868140048
80th epoch Accuracy = 78.65077762068269%
```

[34]:
```
'''Running on validation set'''
actual = NN.predict(test_x)
count_diff = np.sum(np.any(actual != test_y, axis=1))
accuracy_percentage = (1.0 - count_diff/test_y.shape[0])*100
test_acc = f"{accuracy_percentage}%"
output_text = f"Network Accuracy on test/validation set =␣
 ↪{accuracy_percentage}%"
print(output_text)
```

```
Network Accuracy on test/validation set = 82.1010101010101%
```

[35]:
```
plot_graph(number_of_epochs,1)
```





[36]:
```
'''
    To test with any data call the predict method using the NN object
    Only assumption here is that the test input does not have output labels␣
 ↪attached i.e.
    Input test set dimensions could be n X 784
'''

NN.predict(train_x[0])
```

[36]:
```
array([[0., 0., 1., 0.]])
```

[ ]:

[ ]:

[ ]: