ECE 657 Assignment 1
Group 47

- Chikodili Odinakachukwu
- Shreyas Bangalore Suresh
- Yashvardhan Kukreja

Question 1 :

PROBLEM 1:

Consider the scenario where data samples D have two inputs, p and q, and the output of the threshold activation satisfies the following figure (Figure 2) for the given input pair (p, q). Your task is to demonstrate a perception by mentioning the proper weights and threshold. Show the necessary mathematical calculations.

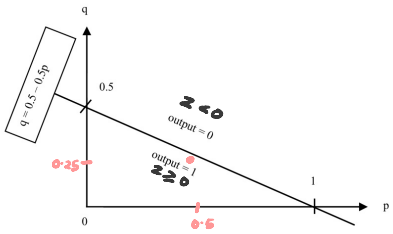
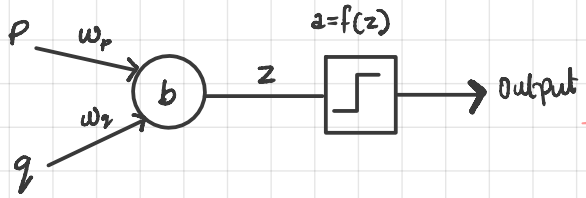


Figure 2: Output of the threshold function for the given inputs.

Inputs = p , q

Threshold = b



Predichon output representation of a linear classifier

$$f(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$

$W_p \cdot p + W_q \cdot q + b = z$ ... eq ①

From problem diagram

Line equation

$0.5 - 0.5p - q = z$ ... eq ②

Equating ① and ② as

$$z = (W_i^T \cdot X_i) + b$$

$-0.5p - q + 0.5 = W_p p + W_q q + b$

$\therefore W_p = -0.5 \quad W_q = -1 \quad b = 0.5$

$\therefore$ Weights = -0.5 & -1

Threshold = 0.5

Testing an input pair of

$= [0.5, 0.25]$

[0.5, 0.25] falls below the line, therefore

$z \geq 0$

$\vec{W_i} = [0.5, -1]$

$z = [W_i^T \times \text{Input}] + b$

$$z = [0.5 \quad 0.25] \cdot \begin{bmatrix} -0.5 \\ -1 \end{bmatrix} + 0.5$$

$z = 0.5(-0.5) + (-1)0.25 + 0.5$

$z = -0.5 + 0.5$

$z = 0$

Which correctly classifies the Point [0.5, 0.25] below the line

PROBLEM 2:

For the given data samples D, there is an input-output mapping of $\{(p_i\,,\,q_i\,)\}_{i=1}^{D}$ that fits the function $q = f(p)$. Calculate the gradient descent update method $\omega \leftarrow \omega - \eta\frac{d\xi}{d\omega}$ for the given sum-of-squared-error value $\xi = \sum_{i=1}^{D}(q_i - p_i^{\omega})^2$ and weights ω (can be figured out employing gradient descent repeatedly).

Q-2

As mentioned in the question, the gradient descent update method is performed by :-

$$\omega \leftarrow \omega - \eta\frac{\partial \epsilon}{\partial \omega} \quad ; \quad \text{— (1)}$$

where, $\epsilon = \sum_{i=1}^{D}(q_i - p_i^{\omega})^2$

p_i represents the input of the i-th data sample

q_i represents the output of the i-th data sample $= f(p_i)$

Also, p_i^{ω} represents the dot-product between the input vector p_i and weights ω.

Therefore,

$$\frac{\partial \epsilon}{\partial \omega} = \frac{\partial}{\partial \omega}\left(\sum_{i=1}^{D}(q_i - p_i^{\omega})^2\right)$$

$$\frac{\partial \epsilon}{\partial \omega} = 2\sum_{i=1}^{D}\left[(q_i - p_i^{\omega}) \cdot \frac{\partial}{\partial \omega}(q_i - p_i^{\omega})\right]$$

$$\frac{\partial \epsilon}{\partial \omega} = 2\sum_{i=1}^{D}(q_i - p_i^{\omega})(0 - p_i)$$

$$\frac{\partial \epsilon}{\partial \omega} = 2\sum_{i=1}^{D}(q_i - p_i^{\omega})(-p_i)$$

OR

$$\boxed{\frac{\partial \epsilon}{\partial \omega} = -2\sum_{i=1}^{D} p_i \cdot (q_i - p_i^{\omega})} \quad \text{— (2)}$$

Comparing (1) and (2), we get the final gradient descent update method as;

$$\omega \leftarrow \omega - \eta \left[-2 \sum_{i=1}^{p} h_i \cdot (q_i - h_i^{\omega}) \right]$$

OR

$$\boxed{\omega \leftarrow \omega + 2\eta \sum_{i=1}^{p} h_i \cdot (q_i - h_i^{\omega})} \quad -(3)$$

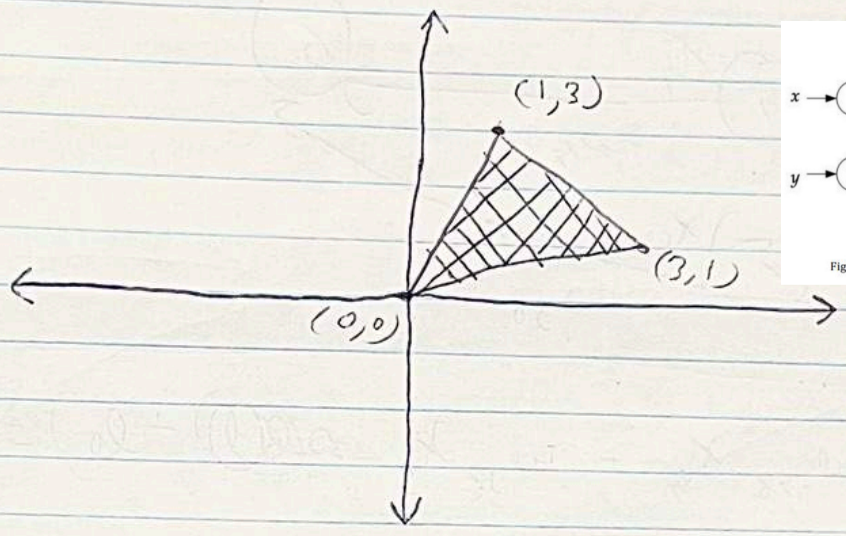
* Now, all we have to do is :-
  - Randomly initialize the weights ω.
  - Choose a learning rate η depending on how slow or accurate we want the gradient descent to perform.
  - Keep on computing and recomputing weights via (3) until they reach convergence i.e. stop changing much.
  - Finally, we would be left with weights ω which would lead to a minimized error represented by $\sum_{i=1}^{p} (q_i - h_i^{\omega})^2$

PROBLEM 3:

The three-layer network in Fig. 1 divides the plane with three lines forming a triangle. Calculate the weights that will give a triangle with its vertices at (x,y) coordinates (0,0), (1,3), and (3,1). Remember that a single perceptron can act as a linear separator. Using this idea, create the triangle with the vertices given. Assign a class to be inside the triangle and the other one to be outside the triangle and try to separate them using 3 perceptrons as shown in the figure below.

Q-3

In the question, we have a triangle formed by join three vertices (hence, lines) (0, 0) (1, 3) (3, 1)



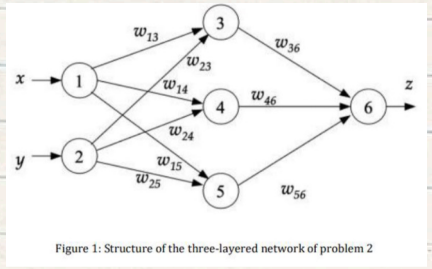
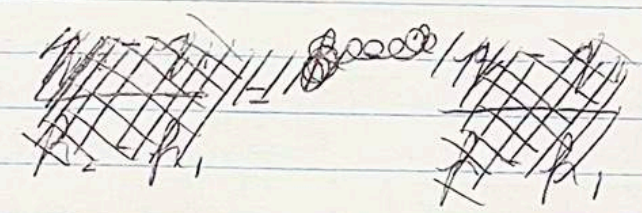


Figure 1: Structure of the three-layered network of problem 2

Let's look at this triangle as a combination of three lines

- Line 1 :- $(0, 0) \longleftrightarrow (1, 3)$
- Line 2 :- $(1, 3) \longleftrightarrow (3, 1)$
- Line 3 :- $(0, 0) \longleftrightarrow (3, 1)$

As per coordinate geometry, we can find the equation of any line between the points (p_1, q_1) and (p_2, q_2) with the formula,

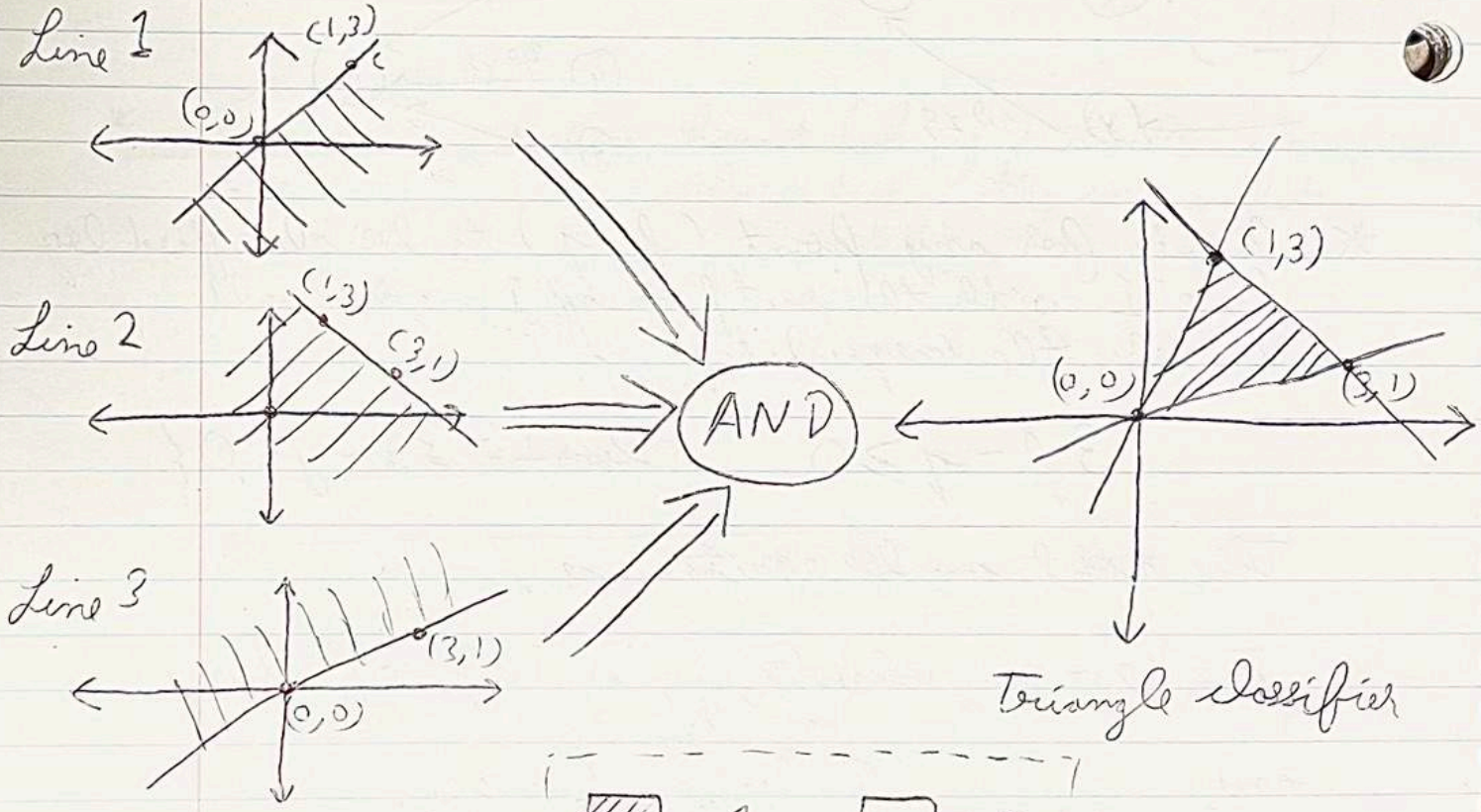


$$\frac{q_2 - q_1}{p_2 - p_1} = \frac{y - q_1}{x - p_1}$$

On the basis of this, we can find the equations of:-

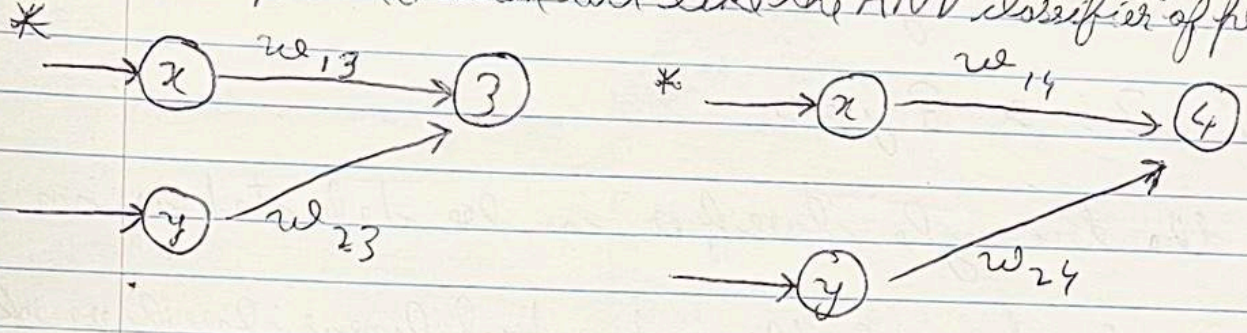
- Line 1:- $3x - y = 0$
- Line 2:- $x + y - 4 = 0$
- Line 3:- $x - 3y = 0$

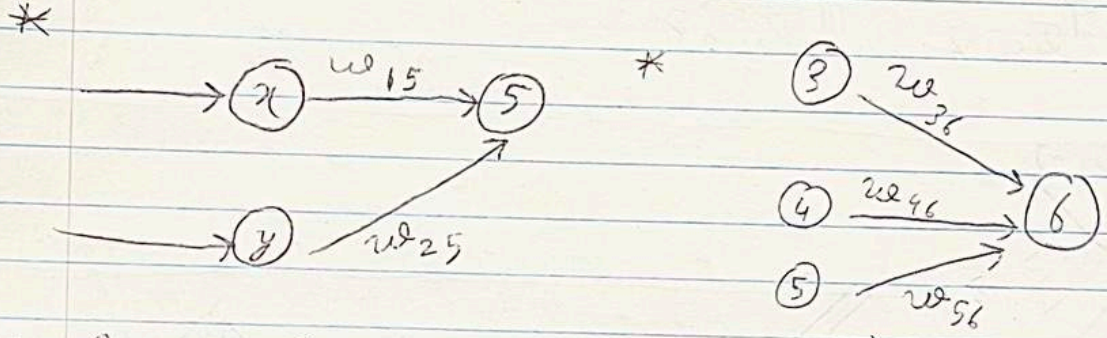
Now, the triangle classifier can be depicted as an AND operation of the individual linear classifiers of line 1, line 2 and line 3.



Therefore, with step function as the activation function:-

- Perceptron 3 can act like the linear classifier of line 1
- Perceptron 4 can act like the linear classifier of line 2.
- Perceptron 5 can act like the linear classifier of line 3.
- Perceptron 6 can act like the AND classifier of perceptron 3, 4, 5.





* Now, for any point (p, q) to be classified by line 1 with the output of 1, it should satisfy the inequality;

$$3p - q \geq 0 \quad \{\text{line 1: } 3x - y = 0\}$$

This model can be written as;

$Z = 3x - y$; where (x, y) are inputs features

and;

$$\text{output} = \begin{cases} 1 & ; \text{if } Z \geq 0 \\ 0 & ; \text{if } Z < 0 \end{cases}$$

for such linear classifiers;

$$\text{model}(z) = w_1 x_1 + w_2 x_2 + \ldots + w_n x_n + b$$

where; $(w_1, w_2 \ldots w_n)$ are the weights and
$(x_1, x_2 \ldots x_n)$ are the input features
b is the bias / threshold

So, for line 1; model $Z = 3x - y + 0$

Therefore, its weights are $(3, -1)$ and bias is 0

(1)

* Similarly, for a point (p, q) to be classified by line 2, it should satisfy the inequality

$$p + q - 4 \leq 0 \qquad \{\text{Line 2} : x + y - 4 = 0\}$$

OR

$$-p - q + 4 \geq 0$$

This makes the model for this classifier;

$Z = -x + y + 4$, where (x, y) are input features.

$$\text{output} = \begin{cases} 1, & \text{if } Z \geq 0 \\ 0, & \text{if } Z < 0 \end{cases}$$

Therefore:
this model's weights are $(-1, -1)$ with bias $= 4$

→ (2)

* Similarly; for any point (p, q) to be classified by line 3 with the output of 1; it should satisfy the inequality:

$$p - 3q \leq 0$$

OR $\quad \{ \text{Line 3} : -x - 3y = 0 \}$

$$-p + 3q \geq 0$$

This makes the model for ~~this~~ this classifier

$Z = -x + 3y$; where (x, y) are input features.

$$\text{output} = \begin{cases} 1 \text{ ; if } Z \geq 0 \\ 0 \text{ ; if } Z < 0 \end{cases}$$

Therefore:
this model's weights are $(-1, 3)$ with bias $= 0$

→ (3)

* Let's say, the output of perceptron 3 is X_1
the output of perceptron 4 is X_2
the output of perceptron 5 is X_3

and, the output of perceptron 6 is X

Then,

X_1	X_2	X_3	X
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Because, for a point to occur inside the triangle (perceptron 6 output = 1), it should be successfully classified by perceptron 3, 4, 5 with output = 1.

Therefore, we can define the model of perceptron 6 as,

$$Z = W_1 X_1 + W_2 X_2 + W_3 X_3 + b$$

where, (W_1, W_2, W_3) are the weights and b is the bias

$$\text{output} = \begin{cases} 1, & \text{if } Z \geq 0 \\ 0, & \text{if } Z < 0 \end{cases}$$

upon solving the above equation with respect to the above table;

> we get this model's weights to be (1, 1, 1) with bias = -3

→ (4)

* on the basis of (1), (2), (3) and (4) we get;

Perceptrons	(3)	(4)	(5)	(6)
Weights	$w_{13} = 3$ $w_{23} = -1$	$w_{14} = -1$ $w_{24} = -1$	$w_{15} = -1$ $w_{25} = 3$	$w_{36} = 1$ $w_{46} = 1$ $w_{56} = 1$
Biases	0	4	0	-3
Activation function	Step	Step	Step	Step

The normalized Widrow Hoff learning rule also known as the normalized least mean square learning rule is expressed as:

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

Where $\Delta w^{(k)} = w^{(k+1)} - w^{(k)}$ represents the weight vector change from iteration (k) to iteration (k+1), $||x^{(k)}||$ is the Euclidean norm of the input vector $x^{(k)}$ at iteration (k), $t^{(k)}$ is the target at iteration (k) and η is a positive number ranging from 0 to 1: $0 < \eta < 1$.

Show that if the same input vector $x^{(k)}$ is presented at iteration $(k + 1)$, then the weight vector decreases by factor $(1 - \eta)$ going from iteration (k) to iteration (k+1). That is: $\Delta w^{(k+1)} = (1 - \eta)\Delta w^{(k)}$

Q-4

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

For the next iteration $(k+1)$; the input used is same which is x^k. which means the target for that input is t^k as well

Therefore;

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

Dividing (2) by (1)

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

$$\frac{\Delta w^{k+1}}{\Delta w^k} = \frac{t^k - w^{k+1}x^k}{t^k - w^k x^k} \quad \text{— (3)}$$

now; if we go back to (1)

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

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

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

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

or, $(t^k - w^{k+1} x^k) = (t^k - w^k x^k) - \frac{\eta x^k x^k}{\|x^k\|^2} [t^k - w^k x^k]$

Now; $\frac{x^k \cdot x^k}{\|x^k\|^2} = 1$

So, $(t^k - w^{k+1} x^k) = (t^k - w^k x^k) - \eta (t^k - w^k x^k)$

OR

$(t^k - w^{k+1} x^k) = (t^k - w^k x^k)(1 - \eta)$ — (4)

Now, comparing (3) and (4)

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

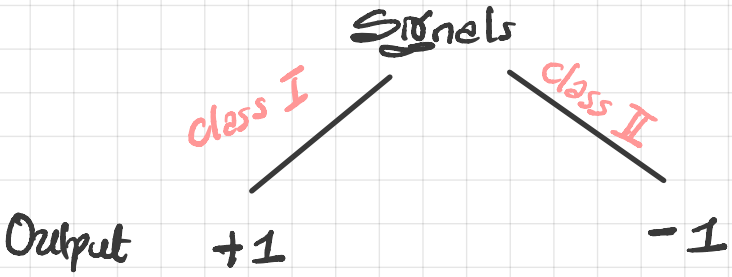
$$\boxed{\frac{\Delta w^{k+1}}{\Delta w^k} = 1 - \eta \; ; \; OR, \; \Delta w^{k+1} = (1-\eta) \Delta w^k}$$

PROBLEM 5:

Download and read the technical paper titled *On Convergence Proofs for Perceptrons* (link) by Albert Novikoff which provides a proof for the convergence of perceptrons in a finite number of steps. After understanding the paper, restate the proof and explain each step in your way (be clear and concise).

A single threshold acting on a set of inputs.

Type of perceptron = α-type Perceptron



The proof states that regardless of the initial weights of the input to a system, if class I can be linearly separable from class II i.e if there are weights that give an output response of +1 for class I and -1 for class II, only a finite number of error correction is needed by the perceptron before the values of these weights are found. And this is given on the premise that such weights that divides the two classes actually exists.

PROOF

→ Assumption: the data set is linearly separable

Continue on Next Page.

θ = Threshold } Value that must be exceeded for the response of the perceptron to be correct.

$\vec{w} = w_1 \dots w_N$ [weight vector]

There exists a $\underline{v}$ such that

$(w_i, \underline{v}) > \theta > 0$ where $i = 1, \dots N$ ... eq (1)

Infinite sequence $\{w_{i_n}\}$

$w_{i_1}, w_{i_2}, w_{i_3} \dots (1 \leq i_k \leq N$ for every $k)$

i.e each w_i vector occurs infinitely often

for $w_1 \dots w_N$

} This is the training sequence of the perceptron

∴ Construct a sequence of recursive vectors $\{v_n\}$

$v_0, v_1, \dots v_N$

where v_0 is arbitary

and $$v_n = \begin{cases} v_{n-1} & \text{if } (v_{n-1}, w_{i_n}) > \theta \\ v_{n-1} + w_{i_n} & \text{if } (v_{n-1}, w_{i_n}) \leq \theta \end{cases}$$

... eq (2)

} This sequence construction describes the error correction procedure of the perceptron

The proof of the theorem is that the recursive vector sequence $\{v_N\}$ is convergent which means that there exists an index m where

$$v_m = v_{m+1} = v_{m+2} = \dots = \tilde{v}$$

A vector v such that

$$(w_i, v) > \theta$$

would give weight values that successfully separates the i-th data provided $v_{n+1} \neq v_n$

PROBLEM 6:

In this problem, you will implement the backpropagation algorithm and train your first multi-layer perceptron to distinguid between 4 classes. You should implement everything on your own **without** using existing libraries like Tensorflow, Keras, or PyTorch.
You are provided with two files "train data.csv" and "train labels.csv" The dataset contains 24754 samples, each with 784 features divided into 4 classes (0,1,2,3). You should divide this into training, and validation sets (a validation set is used to make sure your network did not overfit). You will then provide your model which will be tested with an unseen test set.
Use one input layer, one hidden layer, and one output layer in your implementation. The labels are one-hot encoded. For example, class 0 has a label of [1, 0, 0, 0] and class 2 has a label of [0,0,1,0]. Make sure you use the appropriate activation function in the output layer. You are free to use any number of nodes in the hidden layer. You need to provide one single function that allows us to use the network to predict the test set. This function should output the labels one-hot encoded in a numpy array.
Your work will be graded based on the the successful implementation of the backpropagation algorithm, the multi-layer perceptron, and the test set accuracy. You may want to thoroughly comment your implementation to allow us to easily understand it.

Making imports of the required libraries which would help us implement the MLP Network from scratch in an elegant manner

```
import csv #for importing CSV
import random #for generating random number
import math #basic math module
import numpy as np #for a faster matrix, array mathematical operation
from tqdm.auto import tqdm #for visualization of progress using a bar graph
```

Dataset Class

This class represents a dataset. It would be constructed by providing:

- **data_path**: This path points to the csv containing the training_data
- **labels_path**: This path points to the csv containing the labels_data

This class has multiple methods to make operating with the Dataset, it is representing at the moment, easier:

```
dataset = Dataset("/foo/data.csv", "/foo/labels.csv")
```

- `dataset.load()`: This method takes the data_path and labels_path attributes of `dataset` object, scans through their corresponding CSVs and loads the entire data in the `dataset._data` and `dataset._labels` attributes.
- `dataset.get_data_dimensions()`: This method returns the dimensions of the training_data loaded to this dataset.
- `dataset.get_labels_dimensions()`: This method returns the dimensions of the training_labels loaded to this dataset.
- `dataset.load_data()` and `dataset.load_labels()`: These methods are called by `dataset.load()` under the hood to individually load the training_data and training_labels respectively in the `dataset` object.
- `dataset.normalize_data()`: This method goes through the loaded training_data `dataset._data` and normalizes every data point against the highest value of the data points present in the training_data.
- `data.k_fold_split()`: This method goes through the loaded training_data and training_labels and generates k splits of dataset which, when fed to any neural network during its training and validation phase, can lead to a more fair outcome and analysis of its accuracies.

```
# Class to create objects with different data set
class Dataset:
  def __init__(self, data_path, labels_path=""):
    self.data_path = data_path
    self.labels_path = labels_path
```

```
  self._data_loaded, self._labels_loaded = False, False

# Method to load the data and label point
def load(self, force=False):
  self.load_data(force)
  self.load_labels(force)

# Method to get data matrix dimensions
def get_data_dimensions(self):
  return len(self._data), len(self._data[0])

# Method to get label matrix dimensions
def get_labels_dimensions(self):
  return len(self._labels), len(self._labels[0])

# Method to return the parsed data
def get_parsed_data(self):
  return self._data

# Method to load the data
def load_data(self, force=False):
  if (not force) and self._data_loaded:
    return

  data = []
  with open(self.data_path, 'r') as file:
    reader = csv.reader(file)
    print("Loading the data...")
    for row in tqdm(reader):
        data.append([float(val) for val in row])
  self._data_loaded = True
  self._data = data

# Method to load the label points
def load_labels(self, force=False):
  if (not force) and self._labels_loaded:
    return

  labels = []
  with open(self.labels_path, 'r') as file:
      reader = csv.reader(file)
      print("Loading the labels...")
      for row in tqdm(reader):
          label = [int(round(float(val))) for val in row]
          labels.append(label)
  self._labels_loaded = True
  self._labels = np.array(labels)
```

```
  # Method to normalize data to bring input features to a similar
scale
  def normalize_data(self, force_load=False):
    self.load(force=force_load)

    max_val = max([max(row) for row in self._data])
    normalized_data = [[val / max_val for val in row] for row in
self._data]
    self.data = normalized_data

  # Method to split the data into k folds for k cross validation
  def k_fold_split(self, k, force_load=False):
    self.load(force=force_load)

    combined_data = list(zip(self._data, self._labels))
    random.shuffle(combined_data)
    data, labels = zip(*combined_data)
    fold_size = len(data) // k
    folds = []
    for i in range(k):
        start = i * fold_size
        end = start + fold_size
        val_data = data[start:end]
        val_labels = labels[start:end]
        train_data = data[:start] + data[end:]
        train_labels = labels[:start] + labels[end:]
        folds.append((train_data, train_labels, val_data, val_labels))
    return folds
```

NeuralNetwork Class

This class represents a NeuralNetwork. It would be constructed by providing:

- **input_layer_size**: This parameter is used to set the number of input features
- **hidden_layer_size**: This parameter is used to set the number of hidden layer perceptrons.
- **output_layer_size**: This parameter is used to set the number of output classes
- **activation_func**: This parameter is used to set the type of input activation function

This class has multiple methods to make operating with the NeuralNetwork, it is representing at the moment, easier:

```
neural_network = NeuralNetwork(input_layer_size, hidden_size,
output_layer_size, activation_func=activation_func)
```

- `neural_network.initialize_weights()`: This method is used to initialize weights `W1,W2` and bias `b1,b2` with random values between -1 to 1.
- `neural_network._forward_propagate()`: This method performs forward propagation operations using a given input.

- `neural_network.backward_propagate()`: This method performs backward propagation operations and updates the weights `W1,W2` and bias `b1,b2` based on the `output_error, hidden_error` and `learning_rate`.
- `neural_network.train()` : This method is used to train k-1 fold data and validate with 1 fold for k times.
- `neural_network._set_training_accurary()` and `neural_network.get_training_accurary()`: This method set and get last training accuracy.
- `neural_network.predict()`: This method is used to predict output of one fold data.
- `neural_network._calculate_accuracy()`: This method is used to calculate accuracy of the predicted labels.
- `neural_network.sigmoid()`, `neural_network.relu()`, `neural_network.tanh()` and `neural_network._softmax()`: This method helps in using different activation function based on `activation_func` parameter for the perceptrons in input and output layers.

```
class NetworkUntrainedError(Exception):
    def __init__(self, msg):
        self.msg = msg

# Class to create NeuralNetwork objects
class NeuralNetwork:
  def __init__(self, input_layer_size, hidden_layer_size,
output_layer_size, activation_func):
    self.input_size = input_layer_size
    self.hidden_size = hidden_layer_size
    self.output_size = output_layer_size
    self.activation_func = activation_func

    self._trained_before, self._last_training_accuracy = False, 0.0

    self.initialize_weights()

  # Method to initialize random weights and bias between -1 to 1 range
  def initialize_weights(self):
    self.W1 = np.random.uniform(-1, 1, (self.input_size,
self.hidden_size))
    self.b1 = np.random.uniform(-1, 1, self.hidden_size)
    self.W2 = np.random.uniform(-1, 1, (self.hidden_size,
self.output_size))
    self.b2 = np.random.uniform(-1, 1, self.output_size)

  # Method to perform forward propagation operations
```

```
  def _forward_propagate(self, input_X):
    hidden_layer_input = np.dot(input_X, self.W1) + self.b1

    hidden_layer_output = self.activation_func(hidden_layer_input)

    output_layer_input = np.dot(hidden_layer_output, self.W2) +
self.b2
    output_layer_output = NeuralNetwork._softmax(output_layer_input)

    return hidden_layer_output, output_layer_output

  # Method to perform backward propagation operations
  def _backward_propagate(self, input_X, target_Y,
hidden_layer_output, output_layer_output, learning_rate):
    output_error = output_layer_output - target_Y
    hidden_error = np.dot(output_error, self.W2.T) *
self.activation_func(hidden_layer_output)

    dW2 = np.dot(hidden_layer_output.T, output_error)
    db2 = np.sum(output_error, axis=0)

    dW1 = np.dot(input_X.T, hidden_error)
    db1 = np.sum(hidden_error, axis=0)

    # weight adjustment
    self.W2 -= learning_rate * dW2
    self.b2 -= learning_rate * db2
    self.W1 -= learning_rate * dW1
    self.b1 -= learning_rate * db1

  # Method to train k-1 fold data and validate with 1 fold for k times
  def train(self, dataset, learning_rate, num_epochs, k):
    if self._trained_before:
      print("Retraining the neural network with the provided
configuration...")

    self._trained_before = True

    folds = dataset.k_fold_split(k)

    accuracies = []
    print(f"\nTraining through the {len(folds)} folds with
{num_epochs} epochs each...")
    for i, fold in tqdm(enumerate(folds), total=len(folds)):
        self.initialize_weights()
        train_data_fold, train_labels_fold, val_data_fold,
val_labels_fold = fold

        for epoch in range(num_epochs):
```

```
            for input_X, target_Y in zip(train_data_fold,
train_labels_fold):
                input_X = np.array([input_X])
                target_Y = np.array([target_Y])

                # forward propagation
                hidden_layer_output, output_layer_output =
self._forward_propagate(input_X)

                # backwards propagation
                self._backward_propagate(input_X, target_Y,
hidden_layer_output, output_layer_output, learning_rate)

        val_predictions = self.predict(val_data_fold)
        val_accuracy =
NeuralNetwork._calculate_accuracy(val_predictions, val_labels_fold)
        print(f"Fold-{i+1}'s validation set's accuracy =
{val_accuracy}")
        accuracies.append(val_accuracy)

    avg_training_accuracy = sum(accuracies) / k
    self._set_training_accurary(avg_training_accuracy)

  # Method to set last training accuracy
  def _set_training_accurary(self, accuracy):
    self._last_training_accuracy = accuracy

  # Method to get last training accuracy
  def get_last_training_accuracy(self):
    if not self._trained_before:
      raise(NetworkUntrainedError("training accuracy not found because
the network wasn't ever trained"))
    return self._last_training_accuracy

  # Method to predict output of one fold data
  def predict(self, test_data):
    if not self._trained_before:
      raise(NetworkUntrainedError("Attempt to predict found to happen
before getting the neural network trained"))

    predictions = []
    for X in test_data:
        X = np.array([X])
        _, output_layer_output = self._forward_propagate(X)
        predicted_label = np.argmax(output_layer_output)
        one_hot_label = [1 if i == predicted_label else 0 for i in
range(4)]
        predictions.append(np.array(one_hot_label))
    return predictions
```

```
  # Method to calculate accuracy of predicted output
  @staticmethod
  def _calculate_accuracy(predictions, labels):
    num_correct = 0
    num_total = len(labels)
    for pred, true_label in zip(predictions, labels):
        if np.array_equal(pred, true_label):
            num_correct += 1
    accuracy = num_correct / num_total
    return accuracy

  # Static Method for sigmoid activation function for input layer
  @staticmethod
  def sigmoid(x):
    return 1 / (1 + np.exp(-x))

  # Static Method for relu activation function for input layer
  @staticmethod
  def relu(x):
    return np.maximum(0, x)

  # Static Method for tanh activation function for input layer
  @staticmethod
  def tanh(x):
    return np.tanh(x)

  # Static Method for softmax activation function for output layer(for
multi-class classification)
  @staticmethod
  def _softmax(x):
    exps = np.exp(x - np.max(x, axis=1, keepdims=True))
    return exps / np.sum(exps, axis=1, keepdims=True)
```

Executing the above setup against the input data provided in the assignment

```
# Reading Data set
dataset = Dataset(data_path='/content/train_data.csv',
labels_path='/content/train_labels.csv')
dataset.normalize_data()

_, input_layer_size = dataset.get_data_dimensions()
_, output_layer_size = dataset.get_labels_dimensions()

# Set hyperparameters
hidden_size = 64
learning_rate = 0.01
num_epochs = 1
k = 5
```

```
def find_accuracy_with_activation(activation_func):
  neural_network = NeuralNetwork(input_layer_size, hidden_size,
output_layer_size, activation_func=activation_func)
  neural_network.train(dataset, learning_rate, num_epochs, k)

  accuracy = neural_network.get_last_training_accuracy()
  return neural_network, accuracy

# Finding accuracy with different activation function
print("---------- With RELU Activation ----------")
relu_net, relu_accuracy =
find_accuracy_with_activation(NeuralNetwork.relu)
print("\nAverage Validation Accuracy:", relu_accuracy)
trained_neural_network = relu_net
trained_net_accuracy = relu_accuracy

print("\n\n---------- With SIGMOID Activation ----------")
sigmoid_net, sigmoid_accuracy =
find_accuracy_with_activation(NeuralNetwork.sigmoid)
print("\nAverage Validation Accuracy:", sigmoid_accuracy)
if sigmoid_accuracy > trained_net_accuracy:
  trained_neural_network = sigmoid_net
  trained_net_accuracy = sigmoid_accuracy

print("\n\n---------- With TANH Activation ----------")
tanh_net, tanh_accuracy =
find_accuracy_with_activation(NeuralNetwork.tanh)
print("\nAverage Validation Accuracy:", tanh_accuracy)
if tanh_accuracy > trained_net_accuracy:
  trained_neural_network = tanh_net
  trained_net_accuracy = tanh_accuracy

print(f"\n\nUltimately, trained model chosen with
{trained_net_accuracy*100}% accuracy!")
```

```
Loading the data...

{"model_id":"d71d41c9abd6448e8825945b1d2af4b5","version_major":2,"vers
ion_minor":0}

Loading the labels...

{"model_id":"895ff0cb19694441a2e7071e580abbff","version_major":2,"vers
ion_minor":0}

---------- With RELU Activation ----------

Training through the 5 folds with 1 epochs each...

{"model_id":"deb9ef66ef1f4556afcd2a320307eba3","version_major":2,"vers
ion_minor":0}
```

```
Fold-1's validation set's accuracy = 0.9666666666666667
Fold-2's validation set's accuracy = 0.9626262626262626
Fold-3's validation set's accuracy = 0.9682828282828283
Fold-4's validation set's accuracy = 0.9723232323232324
Fold-5's validation set's accuracy = 0.9662626262626263

Average Validation Accuracy: 0.9672323232323233


---------- With SIGMOID Activation ----------

Training through the 5 folds with 1 epochs each...

{"model_id":"fe0bedb5c07c4d4bb82fe3d435973efd","version_major":2,"vers
ion_minor":0}

Fold-1's validation set's accuracy = 0.9656565656565657
Fold-2's validation set's accuracy = 0.9646464646464646
Fold-3's validation set's accuracy = 0.9670707070707071
Fold-4's validation set's accuracy = 0.9705050505050505
Fold-5's validation set's accuracy = 0.965050505050505

Average Validation Accuracy: 0.9665858585858584


---------- With TANH Activation ----------

Training through the 5 folds with 1 epochs each...

{"model_id":"743c4908dac148e98a42313da32549c8","version_major":2,"vers
ion_minor":0}

Fold-1's validation set's accuracy = 0.9454545454545454
Fold-2's validation set's accuracy = 0.9531313131313132
Fold-3's validation set's accuracy = 0.9402020202020202
Fold-4's validation set's accuracy = 0.9482828282828283
Fold-5's validation set's accuracy = 0.9501010101010101

Average Validation Accuracy: 0.9474343434343435


Ultimately, trained model chosen with 96.72323232323234% accuracy!
```

Steps to run a test_data against the above trained model

- Run all the above cells to train the model.
- The trained model would be stored in the variable called `neural_network`
- Say, the path to your test_data is `/path/content/test_data.csv` Run the following code to get predictions against your test_data

```
testdata = Dataset(data_path="/path/content/test_data.csv")
testdata.load_data()

predictions = trained_neural_network.predict(testdata)
print(predictions)
```