ECE 657 Assignment 1 – Group 80

Rumna Samanta – 20883387
rsamanta@uwaterloo.ca

Juhi Vasudev Bachani – 20979706
jvbachani@uwaterloo.ca

Frank Huang Huang – 20433010
g7huang@uwaterloo.ca

June 08, 2022

1 Problem 1

Restating the proof "On Convergence proof for Perceptrons"

This theorem provides a concise proof of a theorem concerning perceptrons which was earlier proved by Rosenblatt and his collaborators. This theorem appealed to model called α -perceptron, which is a structure that consist of a single threshold element acting on a weighted set of inputs.

Statement of the theorem: The theorem considers that the structure is having a α -perceptron and a set of incoming signals that can be seperated by two disjoint classes. There exists a "satisfactory" assignment of output weights for the associator units which allows the perceptrons to produce an output of +1 which the signal belongs to class I otherwise produce an output of -1 (For class II). The theorem then states that no matter what initial weight the perceptron starts with, the process recursively adjusts the weights by "error correction" method and will terminate after a finite number of steps. It is assumed that the satisfactory assignment exists. Alternatively it can be said that if a data set is linearly separable then the perceptron would find a separating hyperplane after a finite number of updates to the weights.

Let $w_1, w_2 \ldots\ldots w_N$ be the set of vectors in a Euclidean space of fixed finite dimension, and there exists a vector y so that,

$$(w_i, y) > \theta > 0 \qquad i = 1, \dots, N \qquad (1)$$

Recursively, a sequence of vectors $v_0, v_1, \dots, v_n$ can be constructed: v_0 is chosen randomly

$$v_n = \begin{cases} v_{n-1} & \text{if} (w_{i_n}, x_{n-1}) > \theta \\ v_{n-1} + w_{i_n} & \text{if} (w_{i_n}, v_{n-1}) \leq \theta \end{cases} \qquad (2)$$

Remarks:

- The sequence $\{w_{i_n}\}$ represents the ”training sequence”
- The rule for defining $\{v_n\}$ describes the error-correction procedure
- The positive number θ is a threshold which must be exceeded for the response of the perceptron to be correct
- a vector v is such that $(w_i, v) > \theta$ is an assignment of the associator outputs that successfully classifies the ith signal
- The theorem states that the sequence v_n is convergent, which means that after m updates we have

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

- It is evident from equation(2) that as n varies, the sequence $\{v_n\}$ changes, only by adding another set of $w_1 \dots \dots w_N$. For this reason ”convergence” implies ”convergence in a finite number of steps”.

Proof of Theorem:

It is evident that w_{i_n} is inessential, thus we can omit all terms of w_{i_n} from the training sequence for which $v_n = v_{n-1}$. Since we have dropped all other inputs, the training sequence will have a smaller number of inputs and the correction will happen at each step of the sequence. Now, adjusting the notion of theorem, we may re-write the equation(1) as follows,

$$v_n = v_{n-1} + \omega_{i_n} \text{ and } (\omega_{i_n}, v_{n-1}) \leq \theta \quad \text{for each } n \qquad (3)$$

Means whenever the output is lesser than or equal to the threshold (θ) we would adjust the vector.

It is observed that n is the number of corrections made up to the n-th step. After the change of the notion of the theorem, it can be said that n can range only through a finite set of integers. Thus, condition (1) and (2) cannot simultaneously hold true for all n=1,2,3,....

- First, we can show that inequality (1) alone implies

$$\| v_n \|^2 > C n^2 \qquad (4)$$

Where C is a positive constant and n is sufficiently large

- Since $v_n = v_0 + w_{i1}...... + w_{in}$, inequality (1) implies that (v_n,y) satisfies $(v_n,y) > (v_0,y) + n\theta$. Thus, Using the Cauchy-Schwartz inequality we can write,

$$\| v_n \|^2 \geq \frac{(v_n, y)^2}{\| y \|^2} > \frac{[v_0, y + n\theta]^2}{\| y \|^2} = \frac{\theta^2}{\| y \|^2} \left[n + \frac{(v_0, y)}{\theta} \right]^2$$

- If $(v_0,y) > 0$ and we choose $C=\theta^2/\|y\|^2$ then inequality(4) is satisfied for all n.
- If $(v_0,y) < 0$ and we choose $C=(1/4)(\theta^2/\|y\|^2)$ then inequality(4) is satisfied for all $n > -2[(v_0,y)/\theta]$
- On the other hand, we show that inequalities (3) alone imply the inequality

$$\| v_n \|^2 \leq \| v_0 \|^2 + (2\theta + M)\,n \qquad (5)$$

where

$$M = \max_{i=1,\ldots,N} \| \omega_i \|^2$$

- Using inequality (3), the integer-argument function $\|v_k\|^2$ satisfies for each k the difference inequality

$$\| v_k \|^2 - \| v_{k-1} \|^2 = 2\left(v_{k-1}, \omega_{i_k}\right) + \| \omega_{i_k} \|^2 \leq 2\theta + M$$

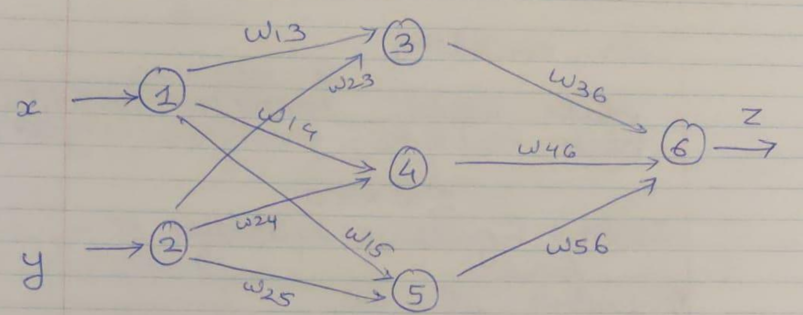
Adding the inequalities for k= 1, 2....n, we obtain inequality (5).

Clearly, inequalities (4) and (5) are incompatible for n sufficiently large.

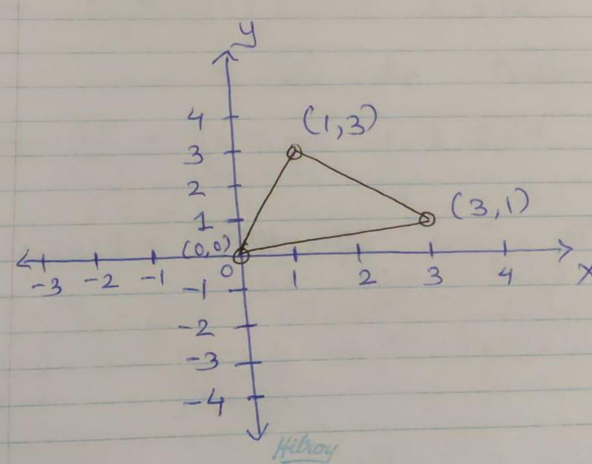
2 Problem 2

1

2) Three - layer Network divides the plane with three lines forming a triangle.



Triangle with vertices at (x, y) co-ordinates (0, 0), (1, 3) and (3, 1).



2

Given in question,

Single perceptron can act as a linear seperator.

Lets consider,

Class inside triangle = positive (+ve)
class outside triangle = Negative (-ve)

$$\sum w x = 0$$

From Figure, equation

Equation at ③,

$$w_{13} x + w_{23} y = 0 \quad \text{— (1)}$$

Equation at ④

$$w_{14} x + w_{24} y = 0 \quad \text{— (2)}$$

Equation at ⑤

$$w_{15} x + w_{25} y = 0 \quad \text{— (3)}$$

- To find equation between (0,0) and (1,3)

$$\text{slope} = m = \frac{y_2 - y_1}{x_2 - x_1} = \frac{3-0}{1-0} = 3$$

$$y = mx + b$$

$y = 3x + c$ (substituting (0,0) point in equation to find c)

$0 = 3(0) + c$

$c = 0$

$y = mx + c$
$y = 3x + 0$
$\boxed{3x - y = 0}$ —— (4)

- To find equation between (1,3) and (3,1)

$m = \frac{1-3}{3-1} = \frac{-2}{2} = -1$

$y = mx + c$ (substituting initial value $x=1$ & $y=3$)

$3 = -1(1) + c$
$3 = -1 + c$
$c = 4$

$y = mx + c$
$y = -x + 4$
$\boxed{x + y - 4 = 0}$ —— (5)

- To find equation between (3,1) and (0,0)

$m = \frac{0-1}{0-3} = \frac{1}{3}$

$y = mx + b$

$y = \frac{1}{3}x + b$ (Substituting $x=0$ and $y=0$)

$0 = 0 + b$

$b = 0$

$y = \frac{1}{3}x + 0$

$3y = x$

$\boxed{x - 3y = 0}$ —— (6)

Comparing equation (1) and (4)

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

$3x - y = 0$

$\boxed{\begin{array}{l} w_{13} = 3 \\ w_{23} = -1 \end{array}}$

Comparing equation (2) and (5)

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

$x + y - 4 = 0$

$\boxed{\begin{array}{l} w_{14} = 1 \\ w_{24} = 1 \\ \text{bias} = -4 \end{array}}$

Comparing equation 3 and 6

$$w_{15}\,x + w_{25}\,y = 0$$
$$x - 3y = 0$$

$$\boxed{\begin{aligned} w_{15} &= 1 \\ w_{25} &= -3 \end{aligned}}$$

Lets assume a point inside triangle (2, 1). substituting it in equation ~~3, 4 and~~ 4, 5 and 6 to know the criteria

eqn 4	eqn 5	eqn 6
$3x - y$	$x + y - 4$	$x - 3y$
$3(2) - 1$	$2 + 1 - 4$	$2 - 3(1)$
$6 - 1$	$-1 < 0$	$-1 < 0$
$5 > 0$		
So $3x - y > 0$	$x + y - 4 < 0$	$x - 3y < 0$

- so point to be inside triangle weights should be ~~Sam~~ same for > 0 (+ve) and negative of old weight for < 0.

$w_{13} = 3$	$w_{14} = -1$	$w_{15} = -1$
$w_{23} = -1$	$w_{24} = -1$	$w_{25} = 3$
	bias $= 4$	

6

To find output, we will use sign activation function

$$y = \begin{cases} +1 & x \geq 0 \\ -1 & x < 0 \end{cases}$$

In this truthtable, we will use AND gate to get output

3	4	5	6 (O/P)
-1	-1	-1	-1
1	1	-1	-1
1	-1	1	-1
-1	-1	1	-1
1	-1	-1	-1
-1	1	-1	-1
1	1	1	1

To have point inside triangle, our output should be +ve. In our truth table when values at 3, 4, 5 is 1, 1, 1 our output is +ve.

So, $w_{36} = 1$
$w_{46} = 1$
$w_{56} = 1$

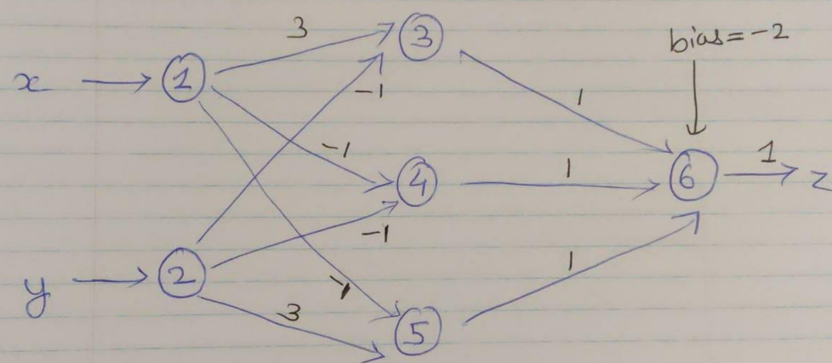
To have output z to be 1

$$w_{36} + w_{46} + w_{56} + bias = 1$$
$$1 + 1 + 1 + bias = 1$$

$\boxed{bias = -2}$

So final Network will have following weights,

$w_{13} = 3$
$w_{23} = -1$
$w_{14} = -1$
$w_{24} = -1$
$w_{15} = -1$
$w_{25} = 3$
$w_{36} = 1$
$w_{46} = 1$
$w_{56} = 1$



3 Problem 3

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

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

weight vector change from iteration (k) to $(k+1)$

$\|x^{(k)}\|$ is the Euclidean norm of the input vector $x^{(k)}$ at iteration (k)

$t^{(k)}$ = target at iteration (k)

η = positive number ranging from 0 to 1

$0 < \eta < 1$

show

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

if, same input vector $x^{(k)}$ is present at iteration $(k+1)$

$\Rightarrow$ At iteration k,

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

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

At iteration $k+1$,

$$\Delta w^{(k+1)} = \eta \, (t^{(k+1)} - w^{(k+1)} x^{(k+1)}) \frac{x^{(k+1)}}{\|x^{(k+1)}\|^2} \quad -(3)$$

As given in question,

Same input vector $x^{(k)}$ is present at iteration $(k+1)$.

Input at iteration $k = x^{(k)}$

$$x^{(k)} = x^{(k+1)} \quad - ④$$

We assume that target is also same so

$$t^{(k)} = t^{(k+1)} \quad - ⑤$$

Substituting ④ & ⑤ in eq^n ③

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

$$\Delta w^{(k+1)} = \eta \left(t^{(k)} - w^{(k+1)} x^{(k)} \right) \frac{x^{(k)}}{\|x^{(k)}\|^2} \quad - ⑥$$

Substituting value of $w^{(k+1)}$ from eq^n ② in eq^n ⑥

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

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

$$\Delta w^{(k+1)} = \eta \left[t^{(k)} - w^{(k)} x^{(k)} - \Delta w^{(k)} x^{(k)}\right] \frac{x^{(k)}}{\|x^{(k)}\|^2} \quad -(7)$$

Substituting value of $\Delta w^{(k)}$ from eq^n (1) into eq^n (7)

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

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

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

$$\Delta w^{(k+1)} = (1-\eta)\, \eta \left[t^{(k)} - w^{(k)} x^{(k)} \right] \frac{x^{(k)}}{\|x^{(k)}\|^2} \quad -\ (8)$$

From eq^n (1)

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

Substituting value of

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

in eq^n (8)

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

4 Problem 4

```python
import pickle
import numpy as np

STUDENT_NAME = 'Rumna Samanta, Juhi Vasudev Bachani, Frank Huang Huang'
STUDENT_ID = '20883387, 20979706, 20433010'

# fixing seed for reproducibility
np.random.seed(0)


def sigmoid(z):
    """
    Non-linear activation function that converts a node output to a value between 0 and 1
    :param z: value
    :return:
    """
    return 1. / (1. + np.exp(-z))


def softmax(x):
    """
    Normalize model outputs and turn them into probabilities
    :param x:
    :return:
    """
    e = np.exp(x)
    return e / np.sum(e, axis=1, keepdims=True)


def accuracy(y_true, y_pred):
    """
```

test_mlp.py .\ | test_mlp.py Final | acc_calc.py | test.py

Final > test_mlp.py > ...

```python
    return e / np.sum(e, axis=1, keepdims=True)


def accuracy(y_true, y_pred):
    """
    Report prediction accuracy

    :param y_true: ground truth labels 2d array
    :param y_pred: predicted labels 2d array
    :return:
    """
    if not (len(y_true) == len(y_pred)):
        print('Size of predicted and true labels not equal.')
        return 0.0

    corr = 0
    for i in range(0,len(y_true)):
        corr += 1 if (y_true[i] == y_pred[i]).all() else 0

    return corr/len(y_true)


class Layer:
    """
    Base MLP layer
    """
    def __init__(self, dim):
        self.dim = dim  # number of neurons
        self.outputs = None  # w^t * b
        self.next_layer = None  # points to the next layer object
        self.prev_layer = None  # points to the previous layer object
```

test_mlp.py .\ | test_mlp.py Final | acc_calc.py | test.py

Final > test_mlp.py > ...

```python
class Layer:
    """
    Base MLP layer
    """
    def __init__(self, dim):
        self.dim = dim  # number of neurons
        self.outputs = None  # w^t * b
        self.next_layer = None  # points to the next layer object
        self.prev_layer = None  # points to the previous layer object
        self.weights = None  # weights between previous layer and next layer
        self.bias = None  # bias of neurons
        self.gradient = None  #

        self.initialized = False

    def __call__(self, next_layer):
        """
        Enables bi-directional traversal between layers
        :param next_layer: instance of subclass of Layer class
        :return:
        """
        self.next_layer = next_layer
        next_layer.prev_layer = self

    def init(self):
        """
        Initialize weights and biases. This should be called before fit()
        :return:
        """
        assert not self.initialized, "This layer has been initialized"
```

```python
    def __call__(self, next_layer):
        """
        Enables bi-directional traversal between layers
        :param next_layer: instance of subclass of Layer class
        :return:
        """
        self.next_layer = next_layer
        next_layer.prev_layer = self

    def init(self):
        """
        Initialize weights and biases. This should be called before fit()
        :return:
        """
        assert not self.initialized, "This layer has been initialized"
        # use given weights if present, otherwise generate weights based on
        self.weights = self._gen_weights()
        self.bias = self._gen_bias()
        self.initialized = True

    def __str__(self):
        return f"{self.__class__.__name__}(dim={self.dim}, weights={self.weights}, " \
               f"gradient={self.gradient}, initialized={self.initialized}) outputs={self.outputs}"

    def _gen_weights(self):
        """
        The interface for initializing weights and bias of the neurons

        :return:
        """
```

```python
    def __str__(self):
        return f"{self.__class__.__name__}(dim={self.dim}, weights={self.weights}, " \
               f"gradient={self.gradient}, initialized={self.initialized}) outputs={self.outputs}"

    def _gen_weights(self):
        """
        The interface for initializing weights and bias of the neurons

        :return:
        """
        raise NotImplemented()

    def _gen_bias(self):
        """
        The interface for initializing biases of the neurons

        :return:
        """
        raise NotImplemented()

    def forward_prop(self):
        """
        Propagate the signals to the next layer
        :return:
        """
        if self.next_layer is not None:
            self.next_layer.outputs = sigmoid(np.dot(self.outputs, self.next_layer.weights) + self.next_layer.bias)
            self.next_layer.forward_prop()
```

test_mlp.py .\ test_mlp.py Final acc_calc.py test.py

Final > test_mlp.py > ...

```python

    def forward_prop(self):
        """
        Propagate the signals to the next layer
        :return:
        """
        if self.next_layer is not None:
            self.next_layer.outputs = sigmoid(np.dot(self.outputs, self.next_layer.weights) + self.next_layer.bias)
            self.next_layer.forward_prop()

    def update_gradient(self):
        raise NotImplemented()

    def back_prop(self, lr):
        if self.prev_layer is not None:
            self.update_gradient()
            w_delta = np.dot(self.prev_layer.outputs.T, self.gradient)
            self.weights = self.weights - lr * w_delta
            self.prev_layer.back_prop(lr)


class InputLayer(Layer):
    def __init__(self, dim):
        super().__init__(dim)

    def apply_input(self, input_matrix):
        # assert input_vector.shape == (self.dim,), "invalid input dim"
        self.outputs = input_matrix

```

test_mlp.py .\ test_mlp.py Final acc_calc.py test.py

Final > test_mlp.py > ...

```python

class InputLayer(Layer):
    def __init__(self, dim):
        super().__init__(dim)

    def apply_input(self, input_matrix):
        # assert input_vector.shape == (self.dim,), "invalid input dim"
        self.outputs = input_matrix

    def _gen_weights(self):
        return None

    def _gen_bias(self):
        return None


class HiddenLayer(Layer):
    def __init__(self, dim):
        super().__init__(dim)

    def _gen_weights(self):
        return np.random.randn(self.prev_layer.dim, self.dim)  # 784 x 5

    def _gen_bias(self):
        return np.zeros(self.dim)  # a vector with 5 elements

    def update_gradient(self):
        if self.next_layer is not None and self.weights is not None:
            #
            self.gradient = np.dot(self.next_layer.gradient, self.next_layer.weights.T) * self.outputs * (1 - self.outputs)
```

```python
class HiddenLayer(Layer):
    def __init__(self, dim):
        super().__init__(dim)

    def _gen_weights(self):
        return np.random.randn(self.prev_layer.dim, self.dim)  # 784 x 5

    def _gen_bias(self):
        return np.zeros(self.dim)  # a vector with 5 elements

    def update_gradient(self):
        if self.next_layer is not None and self.weights is not None:
            #
            self.gradient = np.dot(self.next_layer.gradient, self.next_layer.weights.T) * self.outputs * (1 - self.outputs)


class OutputLayer(Layer):
    def __init__(self, dim):
        super().__init__(dim)
        self.targets = None

    def set_target(self, train_targets):
        self.targets = train_targets

    def update_gradient(self):
        self.gradient = (self.outputs - self.targets)

    def _gen_weights(self):
        return np.random.rand(self.prev_layer.dim, self.dim)  # 5 x 4

```

test_mlp.py .\ | test_mlp.py Final | acc_calc.py | test.py

Final > test_mlp.py > ...

```python
class OutputLayer(Layer):
    def __init__(self, dim):
        super().__init__(dim)
        self.targets = None

    def set_target(self, train_targets):
        self.targets = train_targets

    def update_gradient(self):
        self.gradient = (self.outputs - self.targets)

    def _gen_weights(self):
        return np.random.rand(self.prev_layer.dim, self.dim)  # 5 x 4

    def _gen_bias(self):
        return np.zeros(self.dim)  # a vector with 4 elements


class NNModel:
    def __init__(self):
        # define layers and nodes
        self.input_layer = InputLayer(784)  # input layer that can take 784 features
        self.h1_layer = HiddenLayer(5)  # only hidden layer with neurons
        self.output_layer = OutputLayer(4)  # output layer that spits out 4 outputs

        # link layers
        self.input_layer(self.h1_layer)
        self.h1_layer(self.output_layer)

        # initialize weights and biases of the entire network
```

test_mlp.py .\ | test_mlp.py Final | acc_calc.py | test.py

Final > test_mlp.py > ...

```python

class NNModel:
    def __init__(self):
        # define layers and nodes
        self.input_layer = InputLayer(784)  # input layer that can take 784 features
        self.h1_layer = HiddenLayer(5)  # only hidden layer with neurons
        self.output_layer = OutputLayer(4)  # output layer that spits out 4 outputs

        # link layers
        self.input_layer(self.h1_layer)
        self.h1_layer(self.output_layer)

        # initialize weights and biases of the entire network
        self.input_layer.init()
        self.h1_layer.init()
        self.output_layer.init()

    def print(self):
        print("=========================")
        print(self.input_layer)
        print(self.h1_layer)
        print(self.output_layer)
        print("=========================")

    def fit(self, training_data, training_labels, lr, epochs, eps=1e-10):
        """
        Fit neural networks with a hidden layer

        :param eps: float point precision tolerance
        :param training_data: 2d np array
        :param training_labels: 2d np array
```

test_mlp.py .\ | test_mlp.py Final | acc_calc.py | test.py

Final > test_mlp.py > ...

```python
    def print(self):
        print("=========================")
        print(self.input_layer)
        print(self.h1_layer)
        print(self.output_layer)
        print("=========================")

    def fit(self, training_data, training_labels, lr, epochs, eps=1e-10):
        """
        Fit neural networks with a hidden layer

        :param eps: float point precision tolerance
        :param training_data: 2d np array
        :param training_labels: 2d np array
        :param lr: learning rate
        :param epochs: total number of epoches
        :return:
        """
        for i in range(epochs):
            # train with bpl
            self.input_layer.apply_input(training_data)
            self.output_layer.set_target(training_labels)

            self.input_layer.forward_prop()
            self.output_layer.back_prop(lr)

            # Since we have multi-class labels, apply softmax to get probabilities
            probs = softmax(self.output_layer.outputs)

```

test_mlp.py .\ | test_mlp.py Final | acc_calc.py | test.py

Final > test_mlp.py > ...

```python
        :return:
        """
        for i in range(epochs):
            # train with bpl
            self.input_layer.apply_input(training_data)
            self.output_layer.set_target(training_labels)

            self.input_layer.forward_prop()
            self.output_layer.back_prop(lr)

            # Since we have multi-class labels, apply softmax to get probabilities
            probs = softmax(self.output_layer.outputs)

            # calculate loss at end of each epoch
            r, c = training_labels.shape
            predictions = np.clip(training_labels, eps, 1 - eps)
            loss = (- 1 / r) * (np.sum(probs * np.log(predictions)))
            print("Current epoch", i, "loss", loss)

    def predict(self, test_data):
        h1_out = np.dot(test_data, self.h1_layer.weights) + self.h1_layer.bias
        out = np.dot(h1_out, self.output_layer.weights) + self.output_layer.bias
        probs = softmax(out)

        y_class_preds = []

        for y_pred_prob in probs:
            y_class_pred = [0] * self.output_layer.dim
            y_class_pred[np.argmax(y_pred_prob)] = 1
            y_class_preds.append(y_class_pred)
```

test_mlp.py .\ | test_mlp.py Final | acc_calc.py | test.py

Final > test_mlp.py > ...

```python
    def predict(self, test_data):
        h1_out = np.dot(test_data, self.h1_layer.weights) + self.h1_layer.bias
        out = np.dot(h1_out, self.output_layer.weights) + self.output_layer.bias
        probs = softmax(out)

        y_class_preds = []

        for y_pred_prob in probs:
            y_class_pred = [0] * self.output_layer.dim
            y_class_pred[np.argmax(y_pred_prob)] = 1
            y_class_preds.append(y_class_pred)

        return np.array(y_class_preds)

    @classmethod
    def load(cls, path):
        """
        Loading this model
        :param path:
        :return:
        """
        with open(path, "rb") as o:
            return pickle.load(o)

    def dump(self, path):
        """
        Dumping this model
        :param path:
        :return:
        """
        with open(path, "wb") as o:
```

test_mlp.py .\ | test_mlp.py Final | acc_calc.py | test.py

Final > test_mlp.py > ...

```python

    @classmethod
    def load(cls, path):
        """
        Loading this model
        :param path:
        :return:
        """
        with open(path, "rb") as o:
            return pickle.load(o)

    def dump(self, path):
        """
        Dumping this model
        :param path:
        :return:
        """
        with open(path, "wb") as o:
            pickle.dump(self, o)


def test_mlp(data_file):
    test_x = np.loadtxt(data_file, delimiter=',')
    nn = NNModel.load("nn.pkl")

    return nn.predict(test_x)

```

test_mlp.py .\ | test_mlp.py Final | acc_calc.py | test.py

Final > test_mlp.py > ...

```python

def test_mlp(data_file):
    test_x = np.loadtxt(data_file, delimiter=',')
    nn = NNModel.load("nn.pkl")

    return nn.predict(test_x)


if __name__ == '__main__':
    nn = NNModel()
    raw_x = np.loadtxt("./train_data.csv", delimiter=',')
    raw_y = np.loadtxt("./train_labels.csv", delimiter=',')

    # 7 to 3 split where 70% of data is for training and 30% is for testing
    # we chose this split because training data must be the majority otherwise model may underfit
    train_split = int(raw_x.shape[0] * 0.7)
    test_split = raw_x.shape[0] - train_split

    train_x = raw_x[: train_split, :]
    test_x = raw_x[test_split:, :]

    train_y = raw_y[:train_split, :]
    test_y = raw_y[test_split:, :]

    nn.fit(train_x, train_y, lr=0.5, epochs=1000)

    nn.dump("nn.pkl")
    nn = NNModel.load("nn.pkl")

    train_y_pred = nn.predict(train_x)
    test_y_pred = nn.predict(test_x)
```

```python
    return nn.predict(test_x)


if __name__ == '__main__':
    nn = NNModel()
    raw_x = np.loadtxt("./train_data.csv", delimiter=',')
    raw_y = np.loadtxt("./train_labels.csv", delimiter=',')

    # 7 to 3 split where 70% of data is for training and 30% is for testing
    # we chose this split because training data must be the majority otherwise model may underfit
    train_split = int(raw_x.shape[0] * 0.7)
    test_split = raw_x.shape[0] - train_split

    train_x = raw_x[: train_split, :]
    test_x = raw_x[test_split:, :]

    train_y = raw_y[:train_split, :]
    test_y = raw_y[test_split:, :]

    nn.fit(train_x, train_y, lr=0.5, epochs=1000)

    nn.dump("nn.pkl")
    nn = NNModel.load("nn.pkl")

    train_y_pred = nn.predict(train_x)
    test_y_pred = nn.predict(test_x)

    print("Training Accuracy", accuracy(y_true=test_y, y_pred=train_y_pred))
    print("Test Accuracy", accuracy(y_true=test_y, y_pred=test_y_pred))
```

Accuracy reported:

```
  return e / np.sum(e, axis=1, keepdims=True)
Training Accuracy 0.2697524095342529
Test Accuracy 0.2715415247879033
```

Model Explanation:

Our team chose 7 to 3 to be the test-train split ratio because we believed that majority of data should be allocated for training, otherwise the network would have less information to extract and can cause underfitting. The network we built has three layers. The first layer has 784 nodes which matches total number of features in the train set. The second layer has 5 nodes and we picked five because it gives us the best accuracy during experiments. The third layer has 4 nodes and each node maps to a class in the labels. For layer two, we chose Sigmoid as our activation function, for output layer we chose softmax which can convert non-normalised outputs to probabilities of classes. The training accuracy is about 26.97% and test accuracy is about 27.15% which are slightly better than random guesses (25%).