Project Report on WSNN

By group: 17

LIBRARIES USED: To implement this question, we have used numpy, and random.

1.) NUMPY: Numpy is a python library used when we want to work with arrays.

2.) **RANDOM:** It is an inbuilt module of python that is used to generate random numbers.

PART 1

1. Building our Neural Network

1a) **Preprocess**: Use this module to preprocess the data and divide into train and test.

```
# For making compatibility we do one hot encoding
def oneHotEncoding(y):
    if y == 1:
        return [1, 0, 0]
    elif y == 2:
        return [0, 1, 0]
    else :
        return [0, 0, 1]
```

+ Code + Markdown

```
# print(y)
def split_data(X, y, split_ratio = 0.8):

    indexes = [i for i in range(len(X))]
    train_size = len(indexes) * split_ratio
    train_index = np.random.choice(indexes, size=int(train_size), replace=False)

    X_train = []
    y_train = []
    X_test = []
    y_test = []

    for i in range(len(X)):

        if i in train_index:
            X_train.append(X[i])
            y_train.append(oneHotEncoding(y[i]))

        else:
            X_test.append(X[i])
            y_test.append(oneHotEncoding(y[i]))

    return X_train, X_test, y_train, y_test
```

1b) **Data loader**: Use this module to load all datasets and create mini-batches, with each minibatch having 32 training examples.

Open the data set

```
# Load the data set
def load_dataset():

    # open the data set
    file = open("seeds_dataset.txt")
    lines = file.readlines()
    data_set = []

    for line in lines:
        x = []
        line = line.strip().split()
        for elem in line:
            x.append(float(elem))
        data_set.append(x)

    return data_set

data_set = load_dataset()
```

1c) **Weight initializer**: This module should initialize all weights randomly between -1 and 1.

```
# Function that initializes weights between -1 and 1
def Initialize_Weights(self, size1, size2):
    return np.random.uniform(low=-1, high=1, size=(size1, size2))
```

1d) **Forward pass**: Define the forward() function where you do a forward pass of the neural network.

```python
def forward(self, X):

    # For ANN Specification 1
    if self.no_of_hidden_layer == 1:

        # Hidden layer
        self.z1 = np.dot(X, self.weight_I_H1) + self.bias_1
        self.a1 = self.sigmoid(self.z1)

        # Output layer
        self.z2 = np.dot(self.a1, self.weights_H1_O) + self.bias_2
        self.a2 = self.softmax(self.z2)

        return self.a2

    else:

        # Hidden layer 1
        self.z1 = np.dot(X, self.weight_I_H1) + self.bias_1
        self.a1 = self.relu(self.z1)

        # Hidden layer 2
        self.z2 = np.dot(self.a1, self.weights_H1_H2) + self.bias_2
        self.a2 = self.relu(self.z2)

        # output layer
        self.z3 = np.dot(self.a2, self.weights_H2_O) + self.bias_3
        self.a3 = self.softmax(self.z3)

        return self.a3
```

1e) **Backpropagation**: Define a backward() function where you compute the loss and do a backward pass (backpropagation) of the neural network and update all weights.

```python
def backward(self, X, y, y_hat, learning_rate = 0.01):

    # For ANN Specification 1
    if self.no_of_hidden_layer == 1:
        # Compute gradients

        m = y.shape[0]
        delta2 = y_hat - y
        dW2 = np.dot(self.a1.T, delta2) / m
        db2 = np.sum(delta2, axis=0, keepdims=True) / m

        delta1 = np.dot(delta2, self.weights_H1_O.T) * self.a1 * (1 - self.a1)
        dW1 = np.dot(X. (parameter) learning_rate: float
        db1 = np.sum(de

        # Now update the weights
        self.weights_H1_O -= learning_rate * dW2
        self.bias_2 -= learning_rate * db2
        self.weight_I_H1 -= learning_rate * dW1
        self.bias_1 -= learning_rate * db1
```

```python
    else:

        # compute graients

        m = y.shape[0]
        delta3 = y_hat - y
        dW3 = np.dot(self.a2.T, delta3) / m
        db3 = np.sum(delta3, axis=0, keepdims=True) / m

        delta2 = (np.dot(delta3, self.weights_H2_O.T) * (self.a2 > 0)) / m
        dW2 = np.dot(self.a1.T, delta2)
        db2 = np.sum(delta2, axis=0, keepdims=True)

        delta1 = (np.dot(delta2, self.weights_H1_H2.T) * (self.a1 > 0)) / m
        dW1 = np.dot(X.T, delta1)
        db1 = np.sum(delta1, axis=0, keepdims=True)

        # Now update the weights
        self.weight_I_H1 -= learning_rate * dW1
        self.bias_1 -= learning_rate * db1
        self.weights_H1_H2 -= learning_rate * dW2
        self.bias_2 -= learning_rate * db2
        self.weights_H2_O -= learning_rate * dW3
        self.bias_3 -= learning_rate * db3
```

1f) **Training**: Implement a simple mini batch SGD loop and train your neural network, using forward and backward passes.

```
def train(self, X_train, y_train, X_test, y_test, epochs = 200, batch_size = 32):

    # Calulate the number of batches
    num_batch = X_train.shape[0] // batch_size
    train_acc = []
    test_acc = []
    for i in range(epochs):
        # Shuffle training data
        perm = np.random.permutation(X_train.shape[0])
        X_train = X_train[perm]
        y_train = y_train[perm]

        # Train on mini-batches
        for j in range(num_batch):

            start = j * batch_size
            end = (j + 1) * batch_size

            batch_X = X_train[ (variable) start: Any
            batch_y = y_train[start : end]

            # print(batch_X, batch_y)
            # Forward pass
            y_hat = self.forward(batch_X)
            # print(y_hat)

            # Backward pass
            self.backward(batch_X, batch_y, y_hat)

        # Compute the accuracy on every 10 epochs
```


```
            # print(batch_X, batch_y)
            # Forward pass
            y_hat = self.forward(batch_X)
            # print(y_hat)

            # Backward pass
            self.backward(batch_X, batch_y, y_hat)

        # Compute the accuracy on every 10 epochs
```


```
        if i % 10 == 0:
            acc_train = self.predict(X_train, y_train)
            acc_test = self.predict(X_test, y_test)

            print(f"Epoch : {i} -> Training Accuracy : {acc_train}, Testing Accuracy : {acc_test}")
            train_acc.append(acc_train)
            test_acc.append(acc_test)
    return train_acc, test_acc
```

1g) **Predict**: To test the learned model weights to predict the classes of the test set.

```
# Function to calculate the accuracy
def accuracy(self, y_pred, y):
    acc = np.mean(y_pred == np.argmax(y, axis=1))
    return acc

# Function that predict the output
def predict(self, X, y):
    y_hat = self.forward(X)
    y_pred = np.argmax(y_hat, axis=1)
    acc = self.accuracy(y_pred, y)
    return acc
```

Here is the complete code for ANN

(class) NeuralNetwork

```python
class NeuralNetwork:

    # Constructor
    def __init__(self,
                 no_of_hidden_layer = 0,          # number of hidden layer
                 input_size = 0,                  # input size of neural network
                 output_size = 0,                 # output size of neural network
                 hidden1_size = 0,                # hidden layer 1 size of neural network
                 hidden2_size = 0                 # hidden layer 2 size of neural network
                 ) :

        # Initialize the variables
        self.no_of_hidden_layer = no_of_hidden_layer
        self.input_size = input_size
        self.output_size = output_size
        self.hidden1_size = hidden1_size
        self.hidden2_size = hidden2_size

        # Initialize weights for ANN Specification 1
        if self.no_of_hidden_layer == 1:
            # Initialize weight and bias between input and hidden layer
            self.weight_I_H1 = self.Initialize_Weights(self.input_size, self.hidden1_size)
            self.bias_1 = np.zeros((1, self.hidden1_size))

            # Initialize weight and bias between hidden layer and output layer
            self.weights_H1_O = self.Initialize_Weights(self.hidden1_size, self.output_size)
            self.bias_2 = np.zeros((1, self.output_size))
```

```python
        else:
            # Initialize weight and bias between input and hidden layer 1
            self.weight_I_H1 = self.Initialize_Weights(self.input_size, self.hidden1_size)
            self.bias_1 = np.zeros((1, self.hidden1_size))

            # Initialize weight and bias between hidden layer 1 and hidden layer 2
            self.weights_H1_H2 = self.Initialize_Weights(self.hidden1_size, self.hidden2_size)
            self.bias_2 = np.zeros((1, self.hidden2_size))

            # Initialize weight and bias between hidden layer 2 and output layer
            self.weights_H2_O = self.Initialize_Weights(self.hidden2_size, self.output_size)
            self.bias_3 = np.zeros((1, self.output_size))

    def Info(self):
        print(f"number of hidden layer : {self.no_of_hidden_layer}")
        print(f"input size of neural network : {self.input_size}")
        print(f"output size of neural network : {self.output_size}")
        print(f"hidden layer 1 size of neural network : {self.hidden1_size}")
        print(f"hidden layer 2 size of neural network : {self.hidden2_size}")

    # Function that initializes weights between -1 and 1
    def Initialize_Weights(self, size1, size2):
        return np.random.uniform(low=-1, high=1, size=(size1, size2))


    # Sigmoid activation function
    def sigmoid(self, y):
        return 1 / (1 + np.exp(-y))
```

Markdown | Run All | Clear All Outputs | Outline ...

```python
        else:
            # Initialize weight and bias between input and hidden layer 1
            self.weight_I_H1 = self.Initialize_Weights(self.input_size, self.hidden1_size)
            self.bias_1 = np.zeros((1, self.hidden1_size))

            # Initialize weight and bias between hidden layer 1 and hidden layer 2
            self.weights_H1_H2 = self.Initialize_Weights(self.hidden1_size, self.hidden2_size)
            self.bias_2 = np.zeros((1, self.hidden2_size))

            # Initialize weight and bias between hidden layer 2 and output layer
            self.weights_H2_O = self.Initialize_Weights(self.hidden2_size, self.output_size)
            self.bias_3 = np.zeros((1, self.output_size))

    def Info(self):
        print(f"number of hidden layer : {self.no_of_hidden_layer}")
        print(f"input size of neural network : {self.input_size}")
        print(f"output size of neural network : {self.output_size}")
        print(f"hidden layer 1 size of neural network : {self.hidden1_size}")
        print(f"hidden layer 2 size of neural network : {self.hidden2_size}")

    # Function that initializes weights between -1 and 1
    def Initialize_Weights(self, size1, size2):
        return np.random.uniform(low=-1, high=1, size=(size1, size2))


    # Sigmoid activation function
    def sigmoid(self, y):
        return 1 / (1 + np.exp(-y))

    # Soft max activation function
```

```python
    else:

        # compute graients

        m = y.shape[0]
        delta3 = y_hat - y
        dW3 = np.dot(self.a2.T, delta3) / m
        db3 = np.sum(delta3, axis=0, keepdims=True) / m

        delta2 = (np.dot(delta3, self.weights_H2_O.T) * (self.a2 > 0)) / m
        dW2 = np.dot(self.a1.T, delta2)
        db2 = np.sum(delta2, axis=0, keepdims=True)

        delta1 = (np.dot(delta2, self.weights_H1_H2.T) * (self.a1 > 0)) / m
        dW1 = np.dot(X.T, delta1)
        db1 = np.sum(delta1, axis=0, keepdims=True)

        # Now update the weights
        self.weight_I_H1 -= learning_rate * dW1
        self.bias_1 -= learning_rate * db1
        self.weights_H1_H2 -= learning_rate * dW2
        self.bias_2 -= learning_rate * db2
        self.weights_H2_O -= learning_rate * dW3
        self.bias_3 -= learning_rate * db3


def train(self, X_train, y_train, X_test, y_test, epochs = 200, batch_size = 32):

    # Calulate the number of batches
    num_batch = X_train.shape[0] // batch_size
```

```python
                print(f"Epoch : {i} -> Training Accuracy : {acc_train}, Testing Accuracy : {acc_test}")
                train_acc.append(acc_train)
                test_acc.append(acc_test)
        return train_acc, test_acc


    # Function to calculate the accuracy
    def accuracy(self, y_pred, y):
        acc = np.mean(y_pred == np.argmax(y, axis=1))
        return acc

    def predict(self, X, y):
        y_hat = self.forward(X)
        y_pred = np.argmax(y_hat, axis=1)
        acc = self.accuracy(y_pred, y)
        return acc
```

[60]

PART 2

ANN Specification 1

● No of hidden layers: 1

● No. of neurons in hidden layer: 32

● Activation function in the hidden layer: Sigmoid

● 3 neurons in the output layer.

```
clf = NeuralNetwork(no_of_hidden_layer=1, input_size=7, output_size= 3, hidden1_size=32)
clf.Info()
✓ 0.0s

number of hidden layer : 1
input size of neural network : 7
output size of neural network : 3
hidden layer 1 size of neural network : 32
hidden layer 2 size of neural network : 0
```

● Activation function in the output layer: Softmax
Activation Function in hidden layer: Sigmoid

```
# Sigmoid activation function
def sigmoid(self, y):
    return 1 / (1 + np.exp(-y))

# Soft max activation function
def softmax(self, y):
    exp_y = np.exp(y)
    return exp_y / np.sum(exp_y, axis=1, keepdims=True)
```

● Optimisation algorithm: Mini Batch Stochastic Gradient Descent (SGD)

● Loss function: categorical cross-entropy loss

```
# Cross Entropy loss function
def cross_entropy_loss(self, y_true, y_pred):
    num_samples = y_true.shape[0]
    loss = -np.sum(y_true * np.log(y_pred)) / num_samples
    # print(loss)
    return loss
```

● Learning rate: 0.01

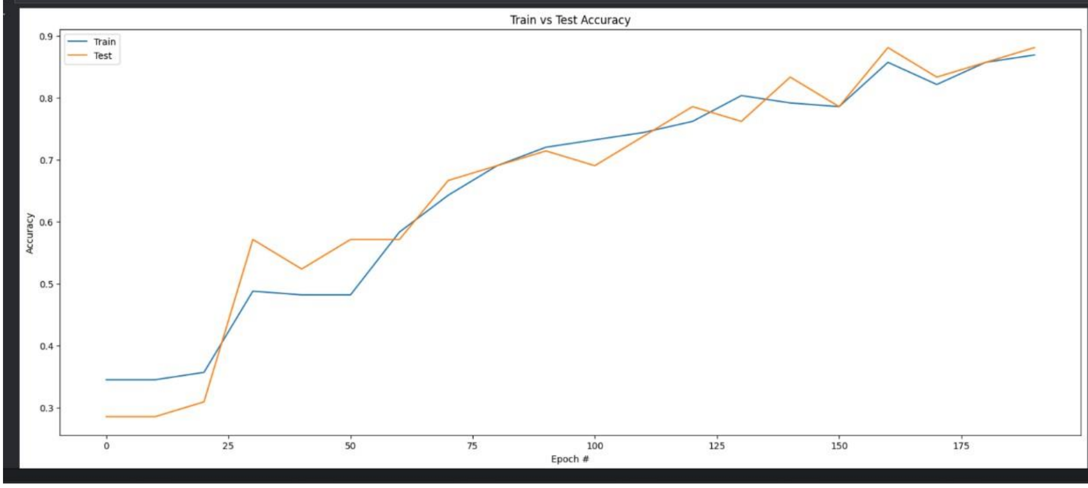
- No. of epochs = 200

```
train_acc, test_acc = clf.train(X_train, y_train, X_test, y_test)
```

```
Epoch : 0 -> Training Accuracy : 0.34523809523809523, Testing Accuracy : 0.2857142857142857
Epoch : 10 -> Training Accuracy : 0.34523809523809523, Testing Accuracy : 0.2857142857142857
Epoch : 20 -> Training Accuracy : 0.35714285714285715, Testing Accuracy : 0.30952380952380953
Epoch : 30 -> Training Accuracy : 0.4880952380952381, Testing Accuracy : 0.5714285714285714
Epoch : 40 -> Training Accuracy : 0.48214285714285715, Testing Accuracy : 0.5238095238095238
Epoch : 50 -> Training Accuracy : 0.48214285714285715, Testing Accuracy : 0.5714285714285714
Epoch : 60 -> Training Accuracy : 0.5833333333333334, Testing Accuracy : 0.5714285714285714
Epoch : 70 -> Training Accuracy : 0.6428571428571429, Testing Accuracy : 0.6666666666666666
Epoch : 80 -> Training Accuracy : 0.6904761904761905, Testing Accuracy : 0.6904761904761905
Epoch : 90 -> Training Accuracy : 0.7202380952380952, Testing Accuracy : 0.7142857142857143
Epoch : 100 -> Training Accuracy : 0.7321428571428571, Testing Accuracy : 0.6904761904761905
Epoch : 110 -> Training Accuracy : 0.7440476190476191, Testing Accuracy : 0.7380952380952381
Epoch : 120 -> Training Accuracy : 0.7619047619047619, Testing Accuracy : 0.7857142857142857
Epoch : 130 -> Training Accuracy : 0.8035714285714286, Testing Accuracy : 0.7619047619047619
Epoch : 140 -> Training Accuracy : 0.7916666666666666, Testing Accuracy : 0.8333333333333334
Epoch : 150 -> Training Accuracy : 0.7857142857142857, Testing Accuracy : 0.7857142857142857
Epoch : 160 -> Training Accuracy : 0.8571428571428571, Testing Accuracy : 0.8809523809523809
Epoch : 170 -> Training Accuracy : 0.8214285714285714, Testing Accuracy : 0.8333333333333334
Epoch : 180 -> Training Accuracy : 0.8571428571428571, Testing Accuracy : 0.8571428571428571
Epoch : 190 -> Training Accuracy : 0.8690476190476191, Testing Accuracy : 0.8809523809523809
```

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"]=(20,8)
plt.plot(range(0, 200, 10), train_acc)
plt.plot(range(0, 200, 10),test_acc)
```

```
plt.plot(range(0, 200, 10), train_acc)
plt.plot(range(0, 200, 10),test_acc)
plt.xlabel('Epoch #')
plt.ylabel('Accuracy')
plt.legend(['Train','Test'])
plt.title('Train vs Test Accuracy')
plt.show()
```




```
print("For ANN Specification 1")
print(f"Final Training accuracy: {clf.predict(X_train, y_train)}")
print(f"Final Testing accuracy: {clf.predict(X_test, y_test)}")
```

0.0s

```
For ANN Specification 1
Final Training accuracy: 0.9523809523809523
Final Testing accuracy: 0.9047619047619048
```

PART 3

3. ANN Specification 2

- No of hidden layers: 2
- No. of neurons in the 1st hidden layer: 64
- No. of neurons in the 2nd hidden layer: 32
- 3 neurons in the output layer.

```
clf = NeuralNetwork(no_of_hidden_layer=2, input_size=7, output_size=3, hidden1_size=64, hidden2_size=32)
clf.Info()
```

```
number of hidden layer : 2
input size of neural network : 7
output size of neural network : 3
hidden layer 1 size of neural network : 64
hidden layer 2 size of neural network : 32
```

- Activation function in both the hidden layers: ReLU

```
# Relu activation function
def relu(self, y):
    return np.maximum(0, y)

def relu_der(self, y):
    y[y <= 0] = 0
    y[y > 0] = 1

    return y
```

- Activation function in the output layer: Softmax

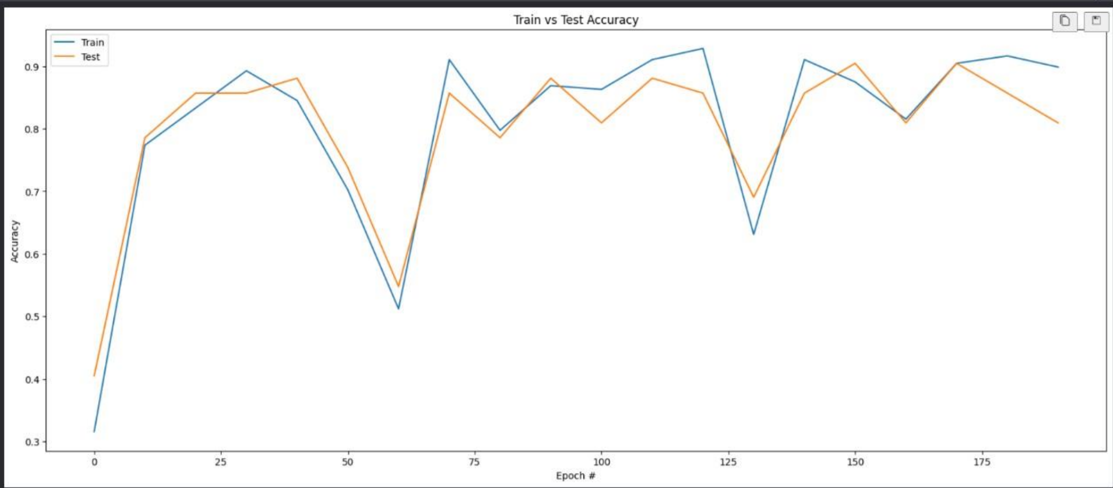
```
# Soft max activation function
def softmax(self, y):
    exp_y = np.exp(y)
    return exp_y / np.sum(exp_y, axis=1, keepdims=True)
```

- Optimisation algorithm: Mini Batch Stochastic Gradient Descent (SGD)
- Loss function: categorical cross-entropy loss
- Learning rate: 0.01
- No. of epochs = 200

```python
train_acc, test_acc = clf.train(X_train, y_train, X_test, y_test)
```

```
Epoch : 0 -> Training Accuracy : 0.31547619047619047, Testing Accuracy : 0.40476190476190477
Epoch : 10 -> Training Accuracy : 0.7738095238095238, Testing Accuracy : 0.7857142857142857
Epoch : 20 -> Training Accuracy : 0.8333333333333334, Testing Accuracy : 0.8571428571428571
Epoch : 30 -> Training Accuracy : 0.8928571428571429, Testing Accuracy : 0.8571428571428571
Epoch : 40 -> Training Accuracy : 0.8452380952380952, Testing Accuracy : 0.8809523809523809
Epoch : 50 -> Training Accuracy : 0.7023809523809523, Testing Accuracy : 0.7380952380952381
Epoch : 60 -> Training Accuracy : 0.5119047619047619, Testing Accuracy : 0.5476190476190477
Epoch : 70 -> Training Accuracy : 0.9107142857142857, Testing Accuracy : 0.8571428571428571
Epoch : 80 -> Training Accuracy : 0.7976190476190477, Testing Accuracy : 0.7857142857142857
Epoch : 90 -> Training Accuracy : 0.8690476190476191, Testing Accuracy : 0.8809523809523809
Epoch : 100 -> Training Accuracy : 0.8630952380952381, Testing Accuracy : 0.8095238095238095
Epoch : 110 -> Training Accuracy : 0.9107142857142857, Testing Accuracy : 0.8809523809523809
Epoch : 120 -> Training Accuracy : 0.9285714285714286, Testing Accuracy : 0.8571428571428571
Epoch : 130 -> Training Accuracy : 0.6309523809523809, Testing Accuracy : 0.6904761904761905
Epoch : 140 -> Training Accuracy : 0.9107142857142857, Testing Accuracy : 0.8571428571428571
Epoch : 150 -> Training Accuracy : 0.875, Testing Accuracy : 0.9047619047619048
Epoch : 160 -> Training Accuracy : 0.8154761904761905, Testing Accuracy : 0.8095238095238095
Epoch : 170 -> Training Accuracy : 0.9047619047619048, Testing Accuracy : 0.9047619047619048
Epoch : 180 -> Training Accuracy : 0.9166666666666666, Testing Accuracy : 0.8571428571428571
Epoch : 190 -> Training Accuracy : 0.8988095238095238, Testing Accuracy : 0.8095238095238095
```

```python
plt.title('Train vs Test Accuracy')
plt.show()
```




```python
print("For ANN Specification 2")
print(f"Final Training accuracy: {clf.predict(X_train, y_train)}")
print(f"Final Testing accuracy: {clf.predict(X_test, y_test)}")
```

```
For ANN Specification 2
Final Training accuracy: 0.9345238095238095
Final Testing accuracy: 0.9047619047619048
```

PART 4

4. Implementation with scikit learn

4a) Use the MLP implementation of scikit learn.

4b) Use the specifications from Part 2 and Part 3, and use the same training and test data.

STEPS:

1 We created a with MLPClassifier with 1 hidden layer and 32 neurons

2 Then, we trained the model and computed the accuracy of ANN specification 1 and 2, which was 85.71 and 66.67, respectively.

```
# Create a MLPClassifier with 1 hidden layer of 32 neurons
model1 = MLPClassifier(hidden_layer_sizes=(32,), activation='logistic', solver='sgd', learning_rate_init=0.01, max_iter=200)
```

```
# Train the model
model1.fit(X_train, y_train)
```

```
c:\Users\Yaman Sahu\AppData\Local\Programs\Python\Python311\Lib\site-packages\sklearn\neural_network\_multilayer_perceptron.py:684:
  warnings.warn(
```

```
MLPClassifier
MLPClassifier(activation='logistic', hidden_layer_sizes=(32,),
              learning_rate_init=0.01, solver='sgd')
```

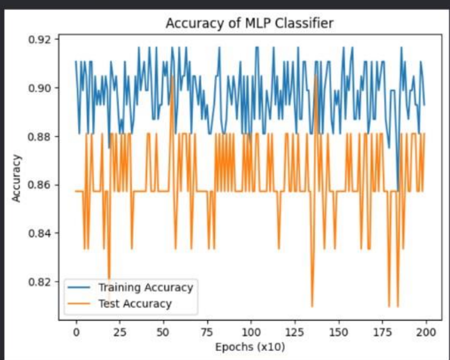
```
# Predict on test set
y_pred1 = model1.predict(X_test)
```

```
# Compute accuracy
accuracy1 = accuracy_score(y_test, y_pred1)
print("Accuracy of ANN Specification 1: ", accuracy1*100)
```

```
Accuracy of ANN Specification 1:  85.71428571428571
```

```
c:\Users\Yaman Sahu\AppData\Local\Programs\Python\Python311\Lib\site-packages\sklearn\neural_network\_multilayer_perceptron.py:684:
  warnings.warn(

Epoch: 200
Training Accuracy: 0.8928571428571429
Test Accuracy: 0.8809523809523809
```




```
Final Training Accuracy: 0.8928571428571429
Final Test Accuracy: 0.8809523809523809
```

```python
# ANN Specification 2
# Create a MLPClassifier with 2 hidden layers of 64 and 32 neurons respectively
model2 = MLPClassifier(hidden_layer_sizes=(64, 32), activation='logistic', solver='sgd', learning_rate_init=0.01, max_iter=200)
```

```python
# Train the model
model2.fit(X_train, y_train)
```

```
c:\Users\Yaman Sahu\AppData\Local\Programs\Python\Python311\Lib\site-packages\sklearn\neural_network\_multilayer_perceptron.py:684:
  warnings.warn(
```

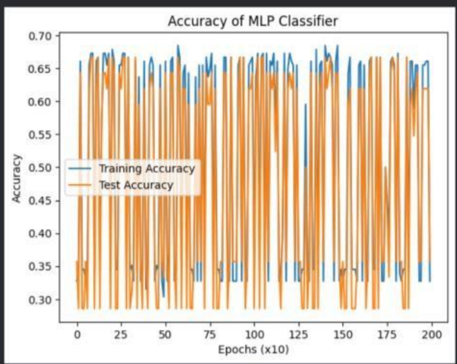
```
MLPClassifier
MLPClassifier(activation='logistic', hidden_layer_sizes=(64, 32),
              learning_rate_init=0.01, solver='sgd')
```

```python
# Predict on test set
y_pred2 = model2.predict(X_test)

# Compute accuracy
accuracy2 = accuracy_score(y_test, y_pred2)
print("Accuracy of ANN Specification 2: ", accuracy2*100)
```

```
Accuracy of ANN Specification 2:  66.66666666666666
```

```
Epoch: 200
Training Accuracy: 0.3273809523809524
Test Accuracy: 0.35714285714285715
```




```
Final Training Accuracy: 0.3273809523809524
Final Test Accuracy: 0.35714285714285715
```