```
#Single Layer Perceptron
class Perceptron:

  #1. Define an initial constructor
  def __init__ (self, learning_rate = 0.1, n_iters = 1000):
    self.weights = None
    self.bias = None
    self.iterations = n_iters
    self.learning_rate = learning_rate

  #sigmoid function
  def sigmoid(self, x):
    return (1.0 / (1.0 + np.exp(-x)))   #Reference: https://www.digitalocean.com/community/tutorials/sigmoid-activation-function-python

  #Classification based on comparison between activation result and bias
  def classifier(self, activation):
    threshold = self.bias
    if activation <= threshold:
      return 0
    else:
      return 1

  #Train the single neural node using the fit method to make it ready for use ahead
  def fit(self, X, Y):
    #Making an array of ones for each column in the dataframe
    self.weights = np.random.rand(X.shape[1])

    #Initial assignment of bias to calculate further
    self.bias = 0

    #Iterating for 'n_iters' time to adjust weight and bias each time we fit the data
    for i in range(self.iterations):
      for row, label in zip(X, Y):
        result = self.sigmoid(np.dot(self.weights, row) + self.bias)
        y_pred = self.classifier(result)

        #To minimise error, we adjust the weights and the bias according to the learning rate
        if label == 1 and y_pred == 0:
          self.weights = self.weights + self.learning_rate * row
          self.bias = self.bias - self.learning_rate * 1
        elif label == 0 and y_pred == 1:
          self.weights = self.weights - self.learning_rate * row
          self.bias = self.bias + self.learning_rate * 1

    return self

  #Defining the predict method to pass new data and calculating labels for them using the existing model
  def predict(self, new_data):
    #List to hold the predictions given for 'n' rows of data
    predictions = []
    self.pred_proba = []

    #Predicting label for each row and appending it to the list
    for row in new_data:
      result = self.sigmoid(np.dot(self.weights, row) + self.bias)
      self.pred_proba.append(result)
      predictions.append(self.classifier(result))

    #Returning an Numpy Array of predictions
    return np.array(predictions)

---------------------------------------
#Multi-Layer Perceptron
```

```python
class MultiLayerPerceptron():

    def __init__(self, num_inputs=3, hidden_layers=[3, 3], num_outputs=2):
        self.num_inputs = num_inputs
        self.hidden_layers = hidden_layers
        self.num_outputs = num_outputs

        # create a generic representation of the layers
        layers = [num_inputs] + hidden_layers + [num_outputs]

        # create random connection weights for the layers
        weights = []
        for i in range(len(layers) - 1):
            w = np.random.rand(layers[i], layers[i + 1])
            weights.append(w)
        self.weights = weights

        # save derivatives per layer
        derivatives = []
        for i in range(len(layers) - 1):
            d = np.zeros((layers[i], layers[i + 1]))
            derivatives.append(d)
        self.derivatives = derivatives

        # save activations per layer
        activations = []
        for i in range(len(layers)):
            a = np.zeros(layers[i])
            activations.append(a)
        self.activations = activations


    def forward_propagate(self, inputs):

        # the input layer activation is just the input itself
        activations = inputs

        # save the activations for backpropogation
        self.activations[0] = activations

        # iterate through the network layers
        for i in range(len(self.weights)):
            # calculate matrix multiplication between previous activation and weight matrix
            net_inputs = np.dot(activations, self.weights[i])

            # apply sigmoid activation function
            activations = self.sigmoid(net_inputs)

            # save the activations for backpropogation
            self.activations[i + 1] = activations

        # return output layer activation
        return activations

    def back_propagate(self, error):

        # iterate backwards through the network layers
        for i in reversed(range(len(self.derivatives))):

            # get activation for previous layer
            activations = self.activations[i+1]

            # apply sigmoid derivative function
            delta = error * self.sigmoid_derivative(activations)

```

```
            # reshape delta as to have it as a 2d array
            delta_re = delta.reshape(delta.shape[0], -1).T

            # get activations for current layer
            current_activations = self.activations[i]

            # reshape activations as to have them as a 2d column matrix
            current_activations = current_activations.reshape(current_activations.shape[0],-1)

            # save derivative after applying matrix multiplication
            self.derivatives[i] = np.dot(current_activations, delta_re)

            # backpropogate the next error
            error = np.dot(delta, self.weights[i].T)

    #Gradient Descent, used for calculating weights by multiplying derivatives and
    def gradient_descent(self, learningRate=1):

        # update the weights by stepping down the gradient
        for i in range(len(self.weights)):
            weights = self.weights[i]
            derivatives = self.derivatives[i]
            weights += derivatives * learningRate

    def classifier(self, x):
        if x <= 0.5:
            return 0
        else:
            return 1

    #sigmoid function
    def sigmoid(self, x):
        return (1.0 / (1.0 + np.exp(-x)))   #Reference: https://www.digitalocean.com/community/tutorials/sigmoid-activation-function-python


    def sigmoid_derivative(self, x):
        return x * (1.0 - x)


    def mse(self, target, output):
        return np.average((target - output) ** 2)


    def fit(self, inputs, targets, epochs, learning_rate):


        targets=targets.to_numpy()

        self.error_rate = []
        for i in range(epochs):
            sum_errors = 0

            # iterate through all the training data
            for j, input in enumerate(inputs):
                target = targets[j]

                # activate the network!
                output = self.forward_propagate(input)

                error = target - output

                self.back_propagate(error)

```

```python
                # now perform gradient descent on the derivatives
                # (this will update the weights
                self.gradient_descent(learning_rate)

                # keep track of the MSE for reporting later
                sum_errors += self.mse(target, output)

            # Epoch complete, report the training error
            self.error_rate.append(sum_errors / len(inputs))

        return self

    def predict(self,new_data):
        self.result = []
        self.pred_proba = []
        temp = self.forward_propagate(new_data)


        for i in temp:
            for point in i:
                self.pred_proba.append(point)
                self.result.append(self.classifier(point))

        return np.array(self.result)
```