**TASK 1:**

import numpy as np

import matplotlib.pyplot as plt

class NeuralNetwork():

  def \_\_init\_\_(self, input\_size, hidden\_size, output\_size):

    super(NeuralNetwork, self).\_\_init\_\_()

    self.weights\_input\_to\_hidden = np.random.random((input\_size, hidden\_size))

    self.weights\_hidden\_to\_output = np.random.random((hidden\_size, output\_size))

  # For Task 2, change the sigmoid function to tan-h and reLU here

  ## TASK 2 CODE STARTS HERE

  def sigmoid(self, x, deriv = False):

    if deriv:

      return x \* (1 - x)

    return 1 / (1 + np.exp(-x))

  ## TASK 2 CODE ENDS HERE

  def train(self, train\_x, train\_y, num\_epochs):

    loss\_dict = {}

    for epoch in range(num\_epochs):

        # Forward prop

        self.l0 = train\_x

        self.l1 = self.sigmoid(np.dot(self.l0, self.weights\_input\_to\_hidden))

        l2 = self.sigmoid(np.dot(self.l1, self.weights\_hidden\_to\_output))

        # Backprop

        # Finding final and hidden layer losses

        loss = train\_y - l2

        if epoch % 10 == 0:

          print('Epoch {}/{} \tLoss:{}'.format(epoch+1, num\_epochs, np.mean(np.abs(loss))))

          #plt.plot(epoch+1,np.mean(loss))

        l2\_delta = loss \* self.sigmoid(l2, deriv = True)

        l1\_error = l2\_delta.dot(self.weights\_hidden\_to\_output.T)

        l1\_delta = l1\_error \* self.sigmoid(self.l1, deriv = True)

        # Optimizing weights

        self.weights\_hidden\_to\_output += self.l1.T.dot(l2\_delta)

        self.weights\_input\_to\_hidden += self.l0.T.dot(l1\_delta)

        # Store loss in a dictionary

        loss\_dict[epoch] = np.abs(np.mean(loss))

    return loss\_dict

  def test(self, test\_x):

    self.l0 = test\_x

    self.l1 = self.sigmoid(np.dot(self.l0, self.weights\_input\_to\_hidden))

    output = self.sigmoid(np.dot(self.l1, self.weights\_hidden\_to\_output))

    if output < 0.5:

      return 0

    return output

    # Training Set

arr\_x = np.array([[0,0,0],

                [1,1,1],

                [1,0,0],

                [0,0,1],

                [1,1,0],

                [1,0,1]])

arr\_y = np.array([[0],

                 [1],

                 [1],

                 [1],

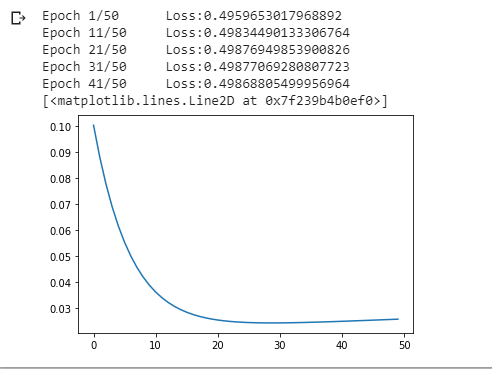
                 [0],

                 [0]])

nn = NeuralNetwork(input\_size=3, hidden\_size = 1, output\_size = 1)

loss = nn.train(train\_x = arr\_x, train\_y = arr\_y, num\_epochs = 50)

plt.plot(list(loss.keys()),list(loss.values()))

OUTPUT OF TASK 1:

TASK 2:

import numpy as np

import matplotlib.pyplot as plt

class NeuralNetwork():

  def \_\_init\_\_(self, input\_size, hidden\_size, output\_size):

    super(NeuralNetwork, self).\_\_init\_\_()

    self.weights\_input\_to\_hidden = np.random.random((input\_size, hidden\_size))

    self.weights\_hidden\_to\_output = np.random.random((hidden\_size, output\_size))

  # For Task 2, change the tanh function to tan-h and reLU here

  ## TASK 2 CODE STARTS HERE

  # TanH Function

  def tanh(self, x, deriv = False):

    f = np.tanh(x)

    if deriv:

      return 1 - f\*\*2

    return f

  ## TASK 2 CODE ENDS HERE

  def train(self, train\_x, train\_y, num\_epochs):

    loss\_dict = {}

    for epoch in range(num\_epochs):

        # Forward prop

        self.l0 = train\_x

        self.l1 = self.tanh(np.dot(self.l0, self.weights\_input\_to\_hidden))

        l2 = self.tanh(np.dot(self.l1, self.weights\_hidden\_to\_output))

        # Backprop

        # Finding final and hidden layer losses

        loss = train\_y - l2

        if epoch % 1000 == 0:

          print('Epoch {}/{} \tLoss:{}'.format(epoch+1, num\_epochs, np.mean(np.abs(loss))))

          #plt.plot(epoch+1,np.mean(loss))

        l2\_delta = loss \* self.tanh(l2, deriv = True)

        l1\_error = l2\_delta.dot(self.weights\_hidden\_to\_output.T)

        l1\_delta = l1\_error \* self.tanh(self.l1, deriv = True)

        # Optimizing weights

        self.weights\_hidden\_to\_output += self.l1.T.dot(l2\_delta)

        self.weights\_input\_to\_hidden += self.l0.T.dot(l1\_delta)

        # Store loss in a dictionary

        loss\_dict[epoch] = np.abs(np.mean(loss))

    return loss\_dict

  def test(self, test\_x):

    self.l0 = test\_x

    self.l1 = self.tanh(np.dot(self.l0, self.weights\_input\_to\_hidden))

    output = self.tanh(np.dot(self.l1, self.weights\_hidden\_to\_output))

    if output < 0.5:

      return 0

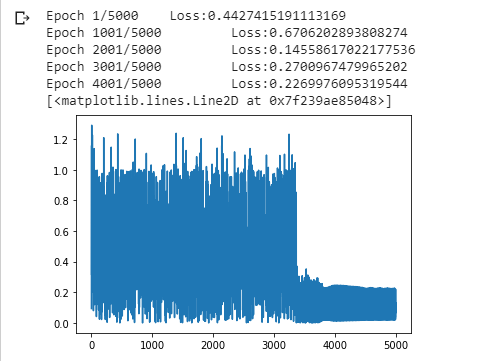
    return output

nn = NeuralNetwork(input\_size=3, hidden\_size = 5, output\_size = 1)

loss = nn.train(train\_x = arr\_x, train\_y = arr\_y, num\_epochs = 5000)

plt.plot(list(loss.keys()),list(loss.values()))

OUTPUT OF TASK 2A:



import numpy as np

import matplotlib.pyplot as plt

class NeuralNetwork():

  def \_\_init\_\_(self, input\_size, hidden\_size, output\_size):

    super(NeuralNetwork, self).\_\_init\_\_()

    self.weights\_input\_to\_hidden = np.random.random((input\_size, hidden\_size))

    self.weights\_hidden\_to\_output = np.random.random((hidden\_size, output\_size))

  ## TASK 2 CODE STARTS HERE

  # Rectified Linear Unit (ReLU) Function

  def relu(self, x, deriv = False):

    if deriv:

      if x.any() > 0:

        return 1

      else:

        return 0

    else:

      return x \* (x > 0)

  ## TASK 2 CODE ENDS HERE

  def train(self, train\_x, train\_y, num\_epochs):

    loss\_dict = {}

    for epoch in range(num\_epochs):

        # Forward prop

        self.l0 = train\_x

        self.l1 = self.relu(np.dot(self.l0, self.weights\_input\_to\_hidden))

        l2 = self.relu(np.dot(self.l1, self.weights\_hidden\_to\_output))

        # Backprop

        # Finding final and hidden layer losses

        loss = train\_y - l2

        if epoch % 1000 == 0:

          print('Epoch {}/{} \tLoss:{}'.format(epoch+1, num\_epochs, np.mean(np.abs(loss))))

          #plt.plot(epoch+1,np.mean(loss))

        l2\_delta = loss \* self.relu(l2, deriv = True)

        l1\_error = l2\_delta.dot(self.weights\_hidden\_to\_output.T)

        l1\_delta = l1\_error \* self.relu(self.l1, deriv = True)

        # Optimizing weights

        self.weights\_hidden\_to\_output += self.l1.T.dot(l2\_delta)

        self.weights\_input\_to\_hidden += self.l0.T.dot(l1\_delta)

        # Store loss in a dictionary

        loss\_dict[epoch] = np.abs(np.mean(loss))

    return loss\_dict

  def test(self, test\_x):

    self.l0 = test\_x

    self.l1 = self.relu(np.dot(self.l0, self.weights\_input\_to\_hidden))

    output = self.relu(np.dot(self.l1, self.weights\_hidden\_to\_output))

    if output < 0.5:

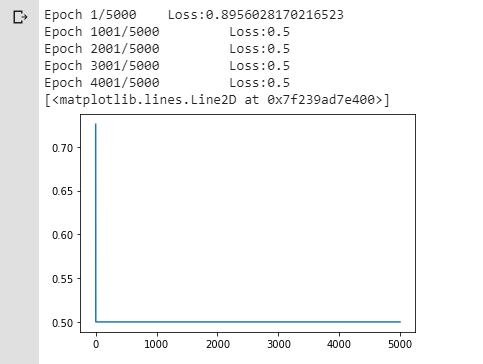
      return 0

    return output

nn = NeuralNetwork(input\_size=3, hidden\_size = 5, output\_size = 1)

loss = nn.train(train\_x = arr\_x, train\_y = arr\_y, num\_epochs = 5000)

plt.plot(list(loss.keys()),list(loss.values()))

OUTPUT OF TASK2B:

TASK3:

import numpy as np

import matplotlib.pyplot as plt

class NeuralNetwork():

  def \_\_init\_\_(self, input\_size, hidden\_size, output\_size):

    super(NeuralNetwork, self).\_\_init\_\_()

    self.weights\_input\_to\_hidden = np.random.random((input\_size, hidden\_size))

    self.weights\_hidden\_to\_output = np.random.random((hidden\_size, output\_size))

  # For Task 2, change the sigmoid function to tan-h and reLU here

  ## TASK 2 CODE STARTS HERE

  def sigmoid(self, x, deriv = False):

    if deriv:

      return x \* (1 - x)

    return 1 / (1 + np.exp(-x))

  ## TASK 2 CODE ENDS HERE

  def train(self, train\_x, train\_y, num\_epochs):

    loss\_dict = {}

    for epoch in range(num\_epochs):

        # Forward prop

        self.l0 = train\_x

        self.l1 = self.sigmoid(np.dot(self.l0, self.weights\_input\_to\_hidden))

        l2 = self.sigmoid(np.dot(self.l1, self.weights\_hidden\_to\_output))

        # Backprop

        # Finding final and hidden layer losses

        loss = train\_y - l2

        if epoch % 10 == 0:

          print('Epoch {}/{} \tLoss:{}'.format(epoch+1, num\_epochs, np.mean(np.abs(loss))))

          #plt.plot(epoch+1,np.mean(loss))

        l2\_delta = loss \* self.sigmoid(l2, deriv = True)

        l1\_error = l2\_delta.dot(self.weights\_hidden\_to\_output.T)

        l1\_delta = l1\_error \* self.sigmoid(self.l1, deriv = True)

        # Optimizing weights

        self.weights\_hidden\_to\_output += self.l1.T.dot(l2\_delta)

        self.weights\_input\_to\_hidden += self.l0.T.dot(l1\_delta)

        # Store loss in a dictionary

        loss\_dict[epoch] = np.abs(np.mean(loss))

    return loss\_dict

  def test(self, test\_x):

    self.l0 = test\_x

    self.l1 = self.sigmoid(np.dot(self.l0, self.weights\_input\_to\_hidden))

    output = self.sigmoid(np.dot(self.l1, self.weights\_hidden\_to\_output))

    if output < 0.5:

      return 0

    return output

    # Training Set

arr\_x = np.array([[0,0,0,0],

                [1,1,1,1],

                [1,0,0,0],

                [0,0,1,0],

                [1,1,0,0],

                [1,0,1,1],

                [0,0,1,1],

                [1,0,0,1]])

arr\_y = np.array([[1],

                 [1],

                 [1],

                 [0],

                 [0],

                 [1],

                [1],

                [0]])

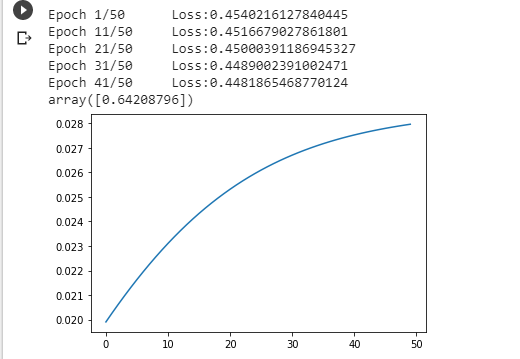
nn\_3 = NeuralNetwork(input\_size=4, hidden\_size = 1, output\_size = 1)

loss = nn.train(train\_x = arr\_x, train\_y = arr\_y, num\_epochs = 50)

plt.plot(list(loss.keys()),list(loss.values()))

nn\_3.test(np.array([0,1,0,0]))

OUTPUT FOR TASK 3A:



TASK 3B:

import numpy as np

import matplotlib.pyplot as plt

class NeuralNetwork():

  def \_\_init\_\_(self, input\_size, hidden\_size, output\_size):

    super(NeuralNetwork, self).\_\_init\_\_()

    self.weights\_input\_to\_hidden = np.random.random((input\_size, hidden\_size))

    self.weights\_hidden\_to\_output = np.random.random((hidden\_size, output\_size))

  # For Task 2, change the sigmoid function to tan-h and reLU here

  ## TASK 2 CODE STARTS HERE

  def sigmoid(self, x, deriv = False):

    if deriv:

      return x \* (1 - x)

    return 1 / (1 + np.exp(-x))

  ## TASK 2 CODE ENDS HERE

  def train(self, train\_x, train\_y, num\_epochs):

    loss\_dict = {}

    for epoch in range(num\_epochs):

        # Forward prop

        self.l0 = train\_x

        self.l1 = self.sigmoid(np.dot(self.l0, self.weights\_input\_to\_hidden))

        l2 = self.sigmoid(np.dot(self.l1, self.weights\_hidden\_to\_output))

        # Backprop

        # Finding final and hidden layer losses

        loss = train\_y - l2

        if epoch % 10 == 0:

          print('Epoch {}/{} \tLoss:{}'.format(epoch+1, num\_epochs, np.mean(np.abs(loss))))

          #plt.plot(epoch+1,np.mean(loss))

        l2\_delta = loss \* self.sigmoid(l2, deriv = True)

        l1\_error = l2\_delta.dot(self.weights\_hidden\_to\_output.T)

        l1\_delta = l1\_error \* self.sigmoid(self.l1, deriv = True)

        # Optimizing weights

        self.weights\_hidden\_to\_output += self.l1.T.dot(l2\_delta)

        self.weights\_input\_to\_hidden += self.l0.T.dot(l1\_delta)

        # Store loss in a dictionary

        loss\_dict[epoch] = np.abs(np.mean(loss))

    return loss\_dict

  def test(self, test\_x):

    self.l0 = test\_x

    self.l1 = self.sigmoid(np.dot(self.l0, self.weights\_input\_to\_hidden))

    output = self.sigmoid(np.dot(self.l1, self.weights\_hidden\_to\_output))

    if output < 0.5:

      return 0

    return output

    # Training Set

arr\_x = np.array([[0,0,0,0],

                [1,1,1,1],

                [1,0,0,0],

                [0,0,1,0],

                [1,1,0,0],

                [1,0,1,1],

                [0,0,1,1],

                [1,1,0,1]])

arr\_y = np.array([[1],

                 [1],

                 [1],

                 [1],

                 [0],

                 [0],

                [0],

                [0]])

nn\_3 = NeuralNetwork(input\_size=4, hidden\_size = 5, output\_size = 1)

loss = nn.train(train\_x = arr\_x, train\_y = arr\_y, num\_epochs = 5000)

plt.plot(list(loss.keys()),list(loss.values()))

nn\_3.test(np.array([0,1,0,0]))

OUTPUT OF TASK 3B:

