```
import numpy as np
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split

class NeuralNet:

    #constructor, we will hardcode this to a 1 hidden layer network, for
     simplicity
    #the problem we will grade on is differentiating 0 and 1s
    #Some things/structure may need to be changed. What needs to stay
     consistant is us being able to call
    #forward with 2 arguments: a data point and a label. Strange architecture,
     but should be good for learning
    def __init__(self, input_size=784, hidden_size=100, output_size=1):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

        self.weight1 =
         np.random.rand(input_size*hidden_size)/np.sqrt(input_size)
        self.weight1 = self.weight1.reshape(self.input_size, self.hidden_size)
        self.bias1 = np.random.rand(hidden_size).reshape(self.hidden_size, 1)
        self.weight2 =
         np.random.rand(hidden_size*output_size)/np.sqrt(hidden_size)
        self.weight2 = self.weight2.reshape(self.hidden_size, self.output_size)
        self.bias2 = np.random.rand(output_size).reshape(self.output_size, 1)
        self.loss_total = 0

    #Potentially helpful, np.dot(a, b), also @ is the matrix product in numpy
     (a @ b)

    def loss(self, y0, y1):
        #return np.sum(np.square(y1 - int(y0)), axis=0)
        return np.sum(np.abs(y1-int(y0)), axis=0)

    def loss_grad(self, y0, y1):
        #return np.sum(2*(y1-int(y0)), axis=0)
        if y1 > int(y0):
            return 1
        elif y1 == int(y0):
            return 0
        return -1
        #not completely accurate since at 0, the L1 function has no derivative

    def relu(self, x):
        return np.maximum(0, x)

    def relu_grad(self, x):
        y = np.copy(x)
        y[y < 0] = 0
        y[y > 0] = 1
```

```
    return y

def sigmoid(self, x):
    return 1/(1 + np.exp(-x))

def sigmoid_grad(self, x):
    return self.sigmoid(x) * (1 - self.sigmoid(x))

#forward function, you may assume x is correct input size
#have the activation from the input to hidden layer be relu, and from
 hidden to output be sigmoid
#have your forward function call backprop: we won't be doing batch
 training, so for EVERY SINGLE input,
#we will update our weights. This is not always (maybe not even here)
 possible or practical, why?
#Also, normally forward doesn't take in labels. Since we'll have forward
 call backprop, it'll take in labels
def forward_pass(self, x):
    x = x.reshape(self.input_size, 1)
    self.z1 = (np.dot(self.weight1.T, x) +
     self.bias1).reshape(self.hidden_size, 1)
    self.a1 = self.relu(self.z1).reshape(self.hidden_size, 1)
    self.z2 = (np.dot(self.weight2.T, self.a1) +
     self.bias2).reshape(self.output_size, 1)
    self.a2 = self.sigmoid(self.z2).reshape(self.output_size, 1)

def forward(self, x, label, lr=.1):
    self.forward_pass(x)
    self.loss_total += self.loss(label, self.a2[0])
    self.backprop(x, label, lr)

#implement backprop, might help to have a helper function update weights
#Recommend you check out the youtube channel 3Blue1Brown and their video
 on backprop
def backprop(self, x, label, lr):
    x = x.reshape(self.input_size, 1)
    dyh_a2 = self.loss_grad(label, self.a2[0])
    dyh_z2 = dyh_a2 * self.sigmoid_grad(self.z2)
    dyh_a1 = np.dot(self.weight2, dyh_z2)
    dyh_w2 = np.dot(dyh_z2, self.a1.T).T
    dyh_b2 = dyh_z2

    dyh_z1 = dyh_a1 * self.relu_grad(self.z1)
    dyh_a0 = np.dot(self.weight1, dyh_z1)
    dyh_w1 = np.dot(dyh_z1, x.T).T
    dyh_b1 = dyh_z1

    self.update_weights(dyh_w1, dyh_b1, dyh_w2, dyh_b2, lr)

def update_weights(self, dw1, db1, dw2, db2, lr):
    self.weight1 = self.weight1 - lr*dw1
```

```
        self.bias1 = self.bias1 - lr*db1
        self.weight2 = self.weight2 - lr*dw2
        self.bias2 = self.bias2 - lr*db2

    def accuracy(self, X_test, y_test):
        predictions = []
        for i in range(len(X_test)):
            self.forward_pass(X_test[i])
            prediction = self.a2[0]
            predictions.append(prediction[0])
        predictions = [round(prediction) for prediction in predictions]
        total = len(X_test)*1.0
        correct = 0.0
        for i in range(len(X_test)):
                if int(y_test[i]) == predictions[i]:
                    correct += 1
        return 100 * correct/total

    def train(self, num_epochs, X_train, y_train, X_test, y_test, lr=.1):
        for epoch in range(num_epochs):
            print("Starting Epoch {}".format(epoch))
            for i in range(len(X_train)):
                self.forward(X_train[i], y_train[i], lr)
            print("Train Accuracy: {:.2f}, Total Loss:
             {:.2f}".format(self.accuracy(X_train, y_train), self.loss_total))
            self.loss_total = 0
        print("Final Test Accuracy: {}".format(network.accuracy(X_test,
         y_test)))

if __name__ == "__main__":
    network = NeuralNet()
    X, y = fetch_openml('mnist_784', version=1, return_X_y=True,
     as_frame=False)
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2)
    ###### have to remove labels model is unable to guess due to sigmoid and
     output_size of 1 ######
    map(int, X_train)
    map(int, X_test)
    mask_train = [i for i in range(len(X_train)) if int(y_train[i]) > 1 ]
    mask_test = [i for i in range(len(X_test)) if int(y_test[i]) > 1]
    X_train = np.delete(X_train, mask_train, axis=0)
    y_train = np.delete(y_train, mask_train, axis=0)
    X_test = np.delete(X_test, mask_test, axis=0)
    y_test = np.delete(y_test, mask_test, axis=0)
    #not necessary but it prevents having to tweak learning rate a lot more
    X_train, X_test = X_train/255, X_test/255

     ########################################################################
     ######################
    network.train(10, X_train, y_train, X_test, y_test, .01)
```