```
import matplotlib
import matplotlib.pyplot as plt
import mnist
import numpy as np

# Gradient descent optimization
# The learning rate is specified by eta
class GDOptimizer(object):
    def __init__(self, eta):
        self.eta = eta

    def initialize(self, layers):
        pass

    # This function performs one gradient descent step
    # layers is a list of dense layers in the network
    # g is a list of gradients going into each layer before the nonlinear
     activation
    # a is a list of outputs from previous layer
    def update(self, layers, g, a):
        m = a[0].shape[1]
        for layer, curGrad, curA in zip(layers, g, a):
            # TODO: PART E
             #################################################################
            dw = self.eta / m * curGrad.dot(curA.T)
            db = self.eta * np.mean(curGrad)
            layer.updateWeights(dw)
            layer.updateBias(db)
            # PART E
             #################################################################
             ######


# Cost function used to compute prediction errors
class QuadraticCost(object):

    # Compute the squared error between the prediction yp and the
     observation y
    # This method should compute the cost per element such that the output
     is the
    # same shape as y and yp
    @staticmethod
    def fx(y,yp):
        # TODO: PART B
         ####################################################################
        return (np.square(y-yp)) / 2
        # PART B
         ####################################################################
         ######

    # Derivative of the cost function with respect to yp
    @staticmethod
    def dx(y,yp):
        # TODO: PART B
         ####################################################################
```

```python
        return yp - y
        # PART B
         ##################################################################
         ######

# Sigmoid function fully implemented as an example
class SigmoidActivation(object):
    @staticmethod
    def fx(z):
        # PART C Example
         ##################################################################
        return 1 / (1 + np.exp(-z))
        # PART C
         ##################################################################
         ######

    @staticmethod
    def dx(z):
        # PART C Example
         ##################################################################
        return SigmoidActivation.fx(z) * (1 - SigmoidActivation.fx(z))
        # PART C
         ##################################################################
         ######


# Hyperbolic tangent function
class TanhActivation(object):

    # Compute tanh for each element in the input z
    @staticmethod
    def fx(z):
        # TODO: PART C
         ##################################################################
        return (np.exp(z) - np.exp((-1)*z)) / (np.exp(z) + np.exp((-1)*z))
        # PART C
         ##################################################################
         ######

    # Compute the derivative of the tanh function with respect to z
    @staticmethod
    def dx(z):
        # TODO: PART C
         ##################################################################
        return 1 - np.square(np.tanh(z))
        # PART C
         ##################################################################
         ######


# Rectified linear unit
class ReLUActivation(object):
    @staticmethod
    def fx(z):
```

```
        # TODO: PART C
         #####################################################################
        return z * (z > 0)
        # PART C
         #####################################################################
         ######

    @staticmethod
    def dx(z):
        # TODO: PART C
         #####################################################################
        return 1.0 * (z > 0)
        # PART C
         #####################################################################
         ######


# Linear activation
class LinearActivation(object):
    @staticmethod
    def fx(z):
        # TODO: PART C
         #####################################################################
        return z
        # PART C
         #####################################################################
         ######

    @staticmethod
    def dx(z):
        # TODO: PART C
         #####################################################################
        return np.ones_like(z)
        # PART C
         #####################################################################
         ######


# This class represents a single hidden or output layer in the neural
 network
class DenseLayer(object):

    # numNodes: number of hidden units in the layer
    # activation: the activation function to use in this layer
    def __init__(self, numNodes, activation):
        self.numNodes = numNodes
        self.activation = activation

    def getNumNodes(self):
        return self.numNodes

    # Initialize the weight matrix of this layer based on the size of the
     matrix W
    def initialize(self, fanIn, scale=1.0):
        s = scale * np.sqrt(6.0 / (self.numNodes + fanIn))
```

```
        self.W = np.random.normal(0, s, (self.numNodes,fanIn))
        self.b = np.random.uniform(-1, 1, (self.numNodes,1))

    # Apply the activation function of the layer on the input z
    def a(self, z):
        return self.activation.fx(z)

    # Compute the linear part of the layer
    # The input a is an n x k matrix where n is the number of samples
    # and k is the dimension of the previous layer (or the input to the
     network)
    def z(self, a):
        return self.W.dot(a) + self.b # Note, this is implemented where we
         assume a is k x n

    # Compute the derivative of the layer's activation function with
     respect to z
    # where z is the output of the above function.
    # This derivative does not contain the derivative of the matrix
     multiplication
    # in the layer.  That part is computed below in the model class.
    def dx(self, z):
        return self.activation.dx(z)

    # Update the weights of the layer by adding dW to the weights
    def updateWeights(self, dW):
        self.W = self.W - dW

    # Update the bias of the layer by adding db to the bias
    def updateBias(self, db):
        self.b = self.b - db


# This class handles stacking layers together to form the completed neural
 network
class Model(object):

    # inputSize: the dimension of the inputs that go into the network
    def __init__(self, inputSize):
        self.layers = []
        self.inputSize = inputSize

    # Add a layer to the end of the network
    def addLayer(self, layer):
        self.layers.append(layer)

    # Get the output size of the layer at the given index
    def getLayerSize(self, index):
        if index >= len(self.layers):
            return self.layers[-1].getNumNodes()
        elif index < 0:
            return self.inputSize
        else:
            return self.layers[index].getNumNodes()
```

```
# Initialize the weights of all of the layers in the network and set
 the cost
# function to use for optimization
def initialize(self, cost, initializeLayers=True):
    self.cost = cost
    if initializeLayers:
        for i in range(0,len(self.layers)):
            if i == len(self.layers) - 1:
                self.layers[i].initialize(self.getLayerSize(i-1))
            else:
                self.layers[i].initialize(self.getLayerSize(i-1))

# Compute the output of the network given some input a
# The matrix a has shape n x k where n is the number of samples and
# k is the dimension
# This function returns
# yp - the output of the network
# a - a list of inputs for each layer of the newtork where
#     a[i] is the input to layer i
#     (note this does not include the network output!)
# z - a list of values for each layer after evaluating layer.z(a) but
#     before evaluating the nonlinear function for the layer
def evaluate(self, x):
    curA = x.T
    a = [curA]
    z = []
    for layer in self.layers:
        z.append(layer.z(curA))
        curA = layer.a(z[-1])
        a.append(curA)
    yp = a.pop()
    return yp, a, z

# Compute the output of the network given some input a
# The matrix a has shape n x k where n is the number of samples and
# k is the dimension
def predict(self, a):
    a, _, _ = self.evaluate(a)
    return a.T

# Computes the gradients at each layer. y is the true labels, yp is the
# predicted labels, and z is a list of the intermediate values in each
# layer. Returns the gradients and the forward pass outputs (per layer).
#
# In particular, we return a list of dMSE/dz_i. The reasoning behind
 this is that
# in the update function for the optimizer, we do not give it the z
 values
# we compute from evaluating the network.
def compute_grad(self, x, y):
    # Feed forward, computing outputs of each layer and
    # intermediate outputs before the non-linearities
    yp, a, z = self.evaluate(x)

    # d is inialized here to be (dMSE / dyp)
```

```
        d = self.cost.dx(y.T, yp)
        grad = []

        # Backpropogate the error
        for layer, curZ in zip(reversed(self.layers), reversed(z)):
            # TODO: PART D
             ###############################################################
             ####
            # grad[i] should correspond with the gradient of the output of
             layer i before the
            # activation is applied (dMSE / dz_i); be sure values are
             stored in the correct
            # ordering!
            grad_i = np.multiply(d,layer.dx(curZ))
            grad = [grad_i] + grad
            d = layer.W.T.dot(grad_i)
            # PART D
             ###############################################################
             ##########
        return grad, a

    # Train the network given the inputs x and the corresponding
     observations y
    # The network should be trained for numEpochs iterations using the
     supplied
    # optimizer
    def train(self, x, y, numEpochs, optimizer):

        # Initialize some stuff
        n = x.shape[0]
        x = x.copy()
        y = y.copy()
        hist = []
        optimizer.initialize(self.layers)

        # Run for the specified number of epochs
        for epoch in range(0,numEpochs):

            # Compute the gradients
            grad, a = self.compute_grad(x, y)

            # Update the network weights
            optimizer.update(self.layers, grad, a)

            # Compute the error at the end of the epoch
            yh = self.predict(x)
            C = self.cost.fx(y, yh)
            C = np.mean(C)
            hist.append(C)
        return hist

    def trainBatch(self, x, y, batchSize, numEpochs, optimizer):
        # Copy the data so that we don't affect the original one when
         shuffling
        x = x.copy()
```

```
        y = y.copy()
        hist = []
        n = x.shape[0]

        for epoch in np.arange(0,numEpochs):

            # Shuffle the data
            r = np.arange(0,x.shape[0])
            x = x[r,:]
            y = y[r,:]
            e = []

            # Split the data in chunks and run SGD
            for i in range(0,n,batchSize):
                end = min(i+batchSize,n)
                batchX = x[i:end,:]
                batchY = y[i:end,:]
                e += self.train(batchX, batchY, 1, optimizer)
            hist.append(np.mean(e))

        return hist


if __name__ == '__main__':
    N0 = 3
    N1 = 9

    # Switch these statements to True to run the code for the corresponding
     parts
    # Part E
    SGD = False
    # Part F
    DIFF_SIZES = True

    # Generate the training set
    np.random.seed(9001)
    y_train = mnist.train_labels()
    y_test = mnist.test_labels()
    X_train = (mnist.train_images()/255.0)
    X_test = (mnist.test_images()/255.0)
    train_idxs = np.logical_or(y_train == N0, y_train == N1)
    test_idxs = np.logical_or(y_test == N0, y_test == N1)
    y_train = y_train[train_idxs].astype('int')
    y_test = y_test[test_idxs].astype('int')
    X_train = X_train[train_idxs]
    X_test = X_test[test_idxs]
    y_train = (y_train == N0).astype('int')
    y_test = (y_test == N0).astype('int')
    y_train *= 2
    y_test *= 2
    y_train -= 1
    y_test -= 1
    X_train = X_train.reshape(X_train.shape[0], -1)
    X_test = X_test.reshape(X_test.shape[0], -1)
    y_test = y_test[:, np.newaxis]
```

```
y = y_train[:, np.newaxis]
x = X_train
xLin=np.linspace(-np.pi,np.pi,250).reshape((-1,1))



yHats = {}

activations = dict(ReLU=ReLUActivation,
                   tanh=TanhActivation,
                   linear=LinearActivation)
lr = dict(ReLU=0.02,tanh=0.02,linear=0.005)
names = ['ReLU','linear','tanh']

#### PART E ####
if SGD:
    print('\n----------------------------------------\n')
    print('Using SGD')
    for key in names[0:3]:
        for batchSize in [10, 50, 100, 200]:
            for epoch in [10, 20, 40]:
                activation = activations[key]
                model = Model(x.shape[1])
                model.addLayer(DenseLayer(4,activation()))
                model.addLayer(DenseLayer(1,LinearActivation()))
                model.initialize(QuadraticCost())
                hist = model.trainBatch(x, y, batchSize, epoch,
                 GDOptimizer(eta=lr[key]))
                y_hat_train = model.predict(x)
                y_pred_train = np.sign(y_hat_train)
                y_hat_test = model.predict(X_test)
                y_pred_test = np.sign(y_hat_test)
                error_train = np.mean(np.square(y_hat_train - y))/2
                error_test = np.mean(np.square(y_hat_test - y_test))/2
                print(key)
                print('Batch size: ', batchSize ,'Epoch: ', epoch,
                 'Train Error: ', error_train, 'Test Error: ',
                 error_test)


#### PART F ####
# Train with different sized networks
if DIFF_SIZES:
    print('\n----------------------------------------\n')
    print('Training with various sized network')
    names = ['ReLU', 'tanh']
    widths = [2, 4, 8, 16, 32]
    errors = {}
    for key in names:
        error = []
        for width in widths:
            activation = activations[key]
            model = Model(x.shape[1])
            model.addLayer(DenseLayer(width,activation()))
            model.addLayer(DenseLayer(1,LinearActivation()))
```

```
model.initialize(QuadraticCost())
epochs = 256
hist =
 model.trainBatch(x,y,x.shape[0],epochs,GDOptimizer(eta=lr
 [key]))

y_hat_train = model.predict(x)
y_hat_test = model.predict(X_test)

error_train = np.mean(np.square(y_hat_train - y))/2
error_test = np.mean(np.square(y_hat_test - y_test))/2

print(key+' neural nework width('+str(width)+') train MSE:
 '+str(error_train))
print(key+' neural network width('+str(width)+') test MSE:
 '+str(error_test))
```