Problem 2

(a) (c) (d)

```
from mnist_tools import *
import numpy as np
import matplotlib.pyplot as plt
import time
from scipy.special import logsumexp

"""
Sigmoid function that takes a numpy array of any shape.
"""
def f(t) :
    return 1/(1+np.exp(-t))

"""
Forecast function which given the learned parameter vectors w and
the data x produces the forecasts.
"""
def h(w,x) :
    return f(np.dot(x,w))>0.5

"""
Computes the loss function L.
Parameters:
w: numpy array of length m containing the parameter vector
X: numpy array of shape (n,m) containing n data samples as rows (each row is a data
y: numpy array of length n containing the labels (0 or 1)
Returns:
A single float, the loss evaluated on the given arguments.
"""
def L(w,X,y):
    n = len(y)
    sum = 0
    for i in range(n):
        g_i = np.dot(w, X[i])
        term = y[i] * np.logaddexp(0, - g_i) + (1 - y[i]) * np.logaddexp(0, g_
        sum = sum + term / n
    return sum

"""
Tests the L function
"""
def test_L() :
    np.random.seed(1000)
    v = np.array([1000])
    w = np.random.randn(10)
    X = np.random.randn(20,10)
    y = np.random.randint(0,2,20)
```

```
        L1 = L(v,v,np.array([0]))
        L2 = L(v,v,np.array([1]))
        L3 = L(w,X,y)
        assert np.abs(L1-1000000) < 1e-9
        assert np.abs(L2) < 1e-9
        assert np.abs(L3-1.08007365415) < 1e-9

"""
Computes the gradient of the loss function.
Parameters:
w: numpy array of length m containing the parameter vector
X: numpy array of shape (n,m) containing n data samples as rows (each row is a data
y: numpy array of length n containing the labels (0 or 1)
Returns:
A numpy vector of length m containing the gradient of the
loss evaluated on the given arguments.
"""
def dL(w,X,y) :
        n = len(y)
        sum = 0
        for i in range(n):
                g_i = np.dot(w, X[i])
                term = X[i] * (f(g_i) - y[i])
                sum = sum + term / n
        return sum

"""
Tests the dL function
"""
def test_dL() :
        np.random.seed(1000)
        v = np.array([1000])
        w = np.random.randn(3)
        X = np.random.randn(200,3)
        y = np.random.randint(0,2,200)
        dL1 = dL(v,v,np.array([0]))
        dL2 = dL(v,v,np.array([1]))
        dL3 = dL(w,X,y)
        assert np.abs(dL1-1000) < 1e-9
        assert np.abs(dL2) < 1e-9
        assert np.linalg.norm(dL3-np.array([-0.12669153,-0.00341384,0.02274541])) < 1e-6

"""
Runs (batch) gradient descent with a backtracking line search to minimize L.
While typically this would include conditions/tolerances for how to stop the
algorithm, here we only required a simplified implementation that has a given fixed
number of steps.
Parameters:
w0: numpy array of length m containing the initial value of w
X: numpy array of shape (n,m) containing the n data samples as rows
y: numpy array of length n containing the labels (0 or 1)
num_steps: number of gradient descent steps to run
alpha: Armijo constant used to make sure the L function sufficiently decreases on each
iteration
beta: backtracking line search constant that determines how much to shrink the step
```

```
size parameter by each time
Returns: the tuple w,ws where
w: numpy array of length m containing the final value of w
ws: a python list of num_steps numpy arrays of length m containing the w-values compu
at each iteration
"""



def gradient_descent(w0,X,y,num_steps=200,alpha=0.01,beta=0.5) :
        w_s = []
        w = w0
        for i in range(num_steps):
                t = 1
                while (L(w, X, y) - L(w - t * dL(w, X, y), X, y) - alpha * t * n
                        t = t * beta
                print("t = ", t)
                w = w - t * dL(w, X, y)
                w_s.append(w)
        return w, w_s



"""
Standarizes the training and test data using the training data to compute
the mean and standard deviation.
"""
def standardize(train,test) :
        m = np.mean(train,axis=0)
        std = np.std(train,axis=0)
        std[np.abs(std)<1e-9] = 1
        return (train-m)/std,(test-m)/std,m,std

"""
Runs the optimization and creates the plots
"""
def run(name,fun,train_x,train_y,test_x,test_y,mean,std) :
        t = time.time()
        g_w,g_ws = fun(np.zeros(train_x.shape[1]),train_x,train_y)
        print('%s Time = %fs'%(name,time.time()-t))
        print('%s Training Loss = %f'%(name,L(g_w,train_x,train_y)))
        test_err = np.sum(np.abs(h(g_w,test_x)-test_y))*1.0/test_x.shape[0]
        print('%s Test Error = %f'%(name,test_err))
        ls = [L(w,train_x,train_y) for w in g_ws]
        tls = [L(w,test_x,test_y) for w in g_ws]
        terr = [np.sum(np.abs(h(w,test_x)-test_y))/test_x.shape[0] for w in g_ws]
        plt.plot(ls)
        plt.title('%s Training Loss'%name)
        plt.savefig('%s_Train_Loss.pdf'%name,bbox_inches='tight')
        plt.close()
        plt.title('%s Test Loss'%name)
        plt.plot(tls)
        plt.savefig('%s_Test_Loss.pdf'%name,bbox_inches='tight')
        plt.close()
        plt.plot(terr)
```

```
        plt.title('%s Test Error'%name)
        plt.savefig('%s_Test_Error.pdf'%name,bbox_inches='tight')
        plt.close()

def main() :
        test_L()
        test_dL()

        train = load_train_data("mnist_all.mat")
        test = load_test_data("mnist_all.mat")
        print('Using %d training examples and %d test examples'%(train.shape[0],test.shape[0
        #We will determine if the image is a '5' or not
        train[:,-1] = train[:,-1]==5
        test[:,-1] = test[:,-1]==5
        train_x,train_y = train[:,:-1],train[:,-1]
        test_x,test_y = test[:,:-1],test[:,-1]
        train_x,test_x,mean,std = standardize(train_x,test_x)

        run('GD',gradient_descent,train_x,train_y,test_x,test_y,mean,std)


if __name__ == "__main__" :
        main()
```