PW_classifier_softmax_stud

October 10, 2019

1 Group 24

- Sebastian Häni haeniseb@students.zhaw.ch
- Raffael Affolter affolraf@students.zhaw.ch
- Benjamin Mäder maedeben@students.zhaw.ch

1.1 MNIST Data

Full classification of MNIST data.
The original MNIST dataset is used.
The following notation is used: m: Number of samples n: Number of features
Here the features refer to the pixel values of the images.

1.1.1 Data Folder

The data can be loaded by using suitable functionality in sklearn which will use a dedicated folder on your local disk for caching. Specify the folder to be used.

[1]:
```
### START YOUR CODE ###
data_home = "/Users/taahase8/deeplearning_data"
### END YOUR CODE ###
```

1.1.2 Data Preparation

Some preparatory steps to be applied before training: * Loading the data * Some plots * Extracting two digits and restricting the classification task to that so that the dataset is well balanced. * Splitting the dataset into train and test * Normalizing the intensities to the range [-1,1]

Plotting Utility

[2]:
```
import numpy as np
import matplotlib.pyplot as plt

def plot_img(img, label, shape):
    """
    Plot the x array by reshaping it into a square array of given shape
    and print the label.

    Parameters:
```

```
    img -- array with the intensities to be plotted of shape
 ↪(shape[0]*shape[1])
    label -- label
    shape -- 2d tuple with the dimensions of the image to be plotted.
    """
    plt.imshow(np.reshape(img, shape), cmap=plt.cm.gray)
    plt.title("Label %i"%label)


def plot_digits(x,y,selection,shape, cols=5):
    """
    Plots the digits in a mosaic with given number of columns.

    Arguments:
    x -- array of images of size (n,m)
    y -- array of labels of size (1,m)
    selection -- list of selection of samples to be plotted
    shape -- shape of the images (a 2d tuple)
    selected_digits -- tuple with the two selected digits (the first associated
 ↪with label 1, the second with label 0)
    """
    if len(selection)==0:
        print("No images in the selection!")
        return
    cols = min(cols, len(selection))
    rows = len(selection)/cols+1
    plt.figure(figsize=(20,4*rows))
    for index, (image, label) in enumerate(zip(x.T[selection,:], y.T[selection,:
 ↪])):
        plt.subplot(rows, cols, index+1)
        plt.imshow(np.reshape(image, shape), cmap=plt.cm.gray)
        plt.title('Sample %i\n Label %i\n' % (selection[index],label), fontsize
 ↪= 12)
    plt.tight_layout()
```

Load Data

[3]:
```
import numpy as np
from sklearn.datasets import fetch_openml

def load_mnist(data_home):
    """
    Loads the mnist dataset, prints the shape of the dataset and
    returns the array with the images, the array with associated labels
    and the shape of the images.
    Parameters:
    data_home -- Absolute path to the DATA_HOME
```

```
    Returns:
    x -- array with images of shape (784,m) where m is the number of images
    y -- array with associated labels with shape (1,m) where m is the number of␣
↪images
    shape -- (28,28)
    """
    mnist = fetch_openml(name='mnist_784', version=1, cache=True,␣
↪data_home=data_home)
    x, y = mnist['data'].T, np.array(mnist['target'], dtype='int').T
    m = x.shape[1]
    y = y.reshape(1,m)
    print("Loaded MNIST original:")
    print("Image Data Shape" , x.shape)
    print("Label Data Shape", y.shape)
    return x,y,(28,28)
```

1.2 Split Data and bring it in the correct shape

Split the data into training set and test set. We use the scikit-learn function 'train_test_split' and use 20% of the samples as test data.

Furthermore, we bring the input data (x) into the shape (n,m) where n is the number of input features and m the number of samples.

[4]:
```
from sklearn.model_selection import train_test_split

def prepare_train_test(x, y, test_size=0.20):
    """
    Split the dataset consisting of an array of images (shape (m,n)) and an␣
↪array of labels (shape (n,))
    into train and test set.

    Parameters:
    x -- Array of images of shape (n,m) where m is the number of samples
    y -- Array of labels of shape (m,) where m is the number of samples
    test_size -- fraction of samples to reserve as test sample

    Returns:
    x_train -- list of images of shape (n,m1) used for training
    y_train -- list of labels of shape (1,m1) used for training
    x_test -- list of images of shape (n,m2) used for testing
    y_test -- list of labels of shape (1,m2) used for testing
    """
    # split
    # We use the functionality of sklearn which assumes that the samples are␣
↪enumerated with the first index
```

```
    x_train, x_test, y_train, y_test = train_test_split(x.T, y.T, test_size=0.
↪20, random_state=1)

    # reshape - transpose back the output obtained from the␣
↪train_test_split-function
    x_train = x_train.T
    x_test = x_test.T
    m_train = x_train.shape[1]
    m_test = x_test.shape[1]
    y_train=y_train.reshape(1,m_train)
    y_test=y_test.reshape(1,m_test)

    print("Shape training set: ", x_train.shape, y_train.shape)
    print("Shape test set:     ", x_test.shape, y_test.shape)

    return x_train, x_test, y_train, y_test
```

Data Normalisation Normalize the data - apply min/max normalization.

[5]:
```
import numpy as np

def normalize(x_train,x_test):
    """
    Applies min/max-normalizes - min and max values computed from the training␣
↪set.
    Common min and max values for all features are used.

    Parameters:
    x_train -- Array of training samples of shape (n,m1) where n,m1 are the␣
↪number of features and samples, respectively.
    x_test -- Array of test samples of shape (n,m2) where n,m2 are the number␣
↪of features and samples, respectively.

    Returns:
    The arrays with the normalized train and test samples.
    """
    ### START YOUR CODE ###
    min_train = np.min(x_train)
    span = np.max(x_train) - min_train
    x_train = (x_train - min_train) / span
    x_test = (x_test - min_train) / span
    ### END YOUR CODE ###
    return x_train, x_test
```

1.2.1 Softmax

[6]:
```
def predict(W, b, X):
    '''
    Compute the per class probabilities for all the m samples by using a␣
 ↪softmax layer with parameters (W, b).

    Arguments:
    W -- weights, a numpy array with shape (ny, nx) (with ny=10 for MNIST).
    b -- biases, a numpy array with shape (ny,1)
    X -- input data of size (nx,m)

    Returns:
    A -- a numpy array of shape (ny,m) with the prediction probabilities for␣
 ↪the digits.
    '''
    ### START YOUR CODE ###
    zl = np.dot(W, X) + b
    return np.exp(zl)/sum(np.exp(zl))
    ### END YOUR CODE ###
```

TEST Softmax

[7]:
```
W = np.array([[1,-1],[0,1],[-1,1]]).reshape(3,2)
b = np.array([0,0,0]).reshape(3,1)
X = np.array([2, 3]).reshape(2,1)
A = predict(W,b,X)
Aexp = np.array([0.01587624,0.86681333,0.11731043]).reshape(A.shape)
np.testing.assert_array_almost_equal(A,Aexp,decimal=8)
np.testing.assert_array_almost_equal(np.sum(A, axis=0), 1.0, decimal=8)

X = np.array([[2,-1,1,-1],[1,1,1,1]]).reshape(2,4)
A = predict(W,b,X)
Aexp = np.array([[0.46831053, 0.01321289, 0.21194156, 0.01321289],
 [0.46831053, 0.26538793, 0.57611688, 0.26538793],
 [0.06337894, 0.72139918, 0.21194156, 0.72139918]]
)
np.testing.assert_array_almost_equal(A,Aexp,decimal=8)
np.testing.assert_array_almost_equal(np.sum(A, axis=0), np.
 ↪ones(4,dtype='float'), decimal=8)
```

1.2.2 Cost Function (Cross Entropy)

[8]:
```
def cost(Ypred, Y):
    """
    Computes the cross entropy cost function for given predicted values and␣
 ↪labels.
```

```
    Parameters:
    Ypred -- prediction from softmax, a numpy array of shape (ny,m)
    Y -- ground truth labels - a numpy array with shape (1,m) containing digits␣
↪0,1,...,9.

    Returns:
    Cross Entropy Cost - a scalar value
    """
    ### START YOUR CODE ###
    m, n = Ypred.shape
    return -(1 / n) * np.sum(np.log(Ypred[Y, range(n)]))
    ### END YOUR CODE ###
```

TEST Cross Entropy Cost

[9]:
```
Y = np.array([1])
Ypred = np.array([0.04742587,0.95257413]).reshape(2,1)
J = cost(Ypred,Y)
Jexp = 0.04858735
np.testing.assert_almost_equal(J,Jexp,decimal=8)

Y = np.array([1,1,1,0])
Ypred = np.array([[1.79862100e-02, 6.69285092e-03, 4.74258732e-02, 9.
↪99088949e-01],
                  [9.82013790e-01, 9.93307149e-01, 9.52574127e-01, 9.
↪11051194e-04]])
Jexp = 0.01859102
J = cost(Ypred,Y)
np.testing.assert_almost_equal(J,Jexp,decimal=8)
```

1.2.3 Update Rules for the Parameters

Different update rules associated with the different cost functions.

[10]:
```
def onehot(y,n):
    """
    Constructs a one-hot-vector from a given array of labels (shape (1,m),␣
↪containing numbers 0,1,...,n-1)
    and the number of classes n.
    The resulting array has shape (n,m) and in row j and column i a '1' if the␣
↪i-th sample has label 'j'.

    Parameters:
    y -- labels, numpy array of shape (1,m)
    n -- number of classes
```

```
    Returns:
    On-hot-encoded vector of shape (n,m)
    """

    ### START YOUR CODE ###
    _, m = y.shape
    result = np.zeros((n, m))
    result[y[0, :], range(m)] = 1
    ### START YOUR CODE ###
    return result
```

[11]:
```
## Test ##
Y = np.array([1,3,0]).reshape(1,3)
onehot_comp = onehot(Y,4)
onehot_exp = np.array([[0,0,1],[1,0,0],[0,0,0],[0,1,0]]).reshape(4,3)
np.testing.assert_almost_equal(onehot_exp,onehot_comp,decimal=8)
```

[12]:
```
def gradient(X, Y, A):
    """
    Computes the update of the weights and bias - by using the cross entropy␣
 ↪cost.

    Arguments:
    X -- input data of size (nx,m)
    Y -- output labels - a numpy array with shape (1,m).
    A -- predicted scores - a numpy array with shape (ny,m)

    Returns:
    gradJ -- dictionary with the gradient w.r.t. W (key "dW" with shape␣
 ↪(ny,nx))
             and w.r.t. b (key "db" with shape (ny,1))
    """
    ### START YOUR CODE ###
    m, n = A.shape
    mask = onehot(Y, m)
    dW = -(1 / n) * np.dot(mask - A, X.T)
    db = -(1 / n) * np.sum(mask - A, axis=1).reshape(m, 1)
    return {"dW": dW, "db": db}
    ### END YOUR CODE ###
```

Test the Calculation of the Gradient

[13]:
```
W = np.array([[1,-1],[0,1],[-1,1]]).reshape(3,2)
b = np.array([0,0,0]).reshape(3,1)
X = np.array([[2,-1,1,-1],[1,1,1,1]]).reshape(2,4)
A = predict(W,b,X)

Y = np.array([1,1,1,1]).reshape(1,4)
```

```
gradJ = gradient(X,Y,A)
dW = gradJ['dW']
db = gradJ['db']
dWexp = np.array([[ 0.28053421,0.17666947],
                  [-0.00450948,-0.60619918],
                  [-0.27602473,0.42952972]]).reshape(3,2)
dbexp = np.array([0.17666947,-0.60619918,0.42952972]).reshape(3,1)
np.testing.assert_array_almost_equal(dW,dWexp,decimal=8)
np.testing.assert_array_almost_equal(db,dbexp, decimal=8)
```

1.2.4 Metrics for measuring the performance of the algorithm

[14]:
```
def error_rate(Ypred, Y):
    """
    Compute the error rate defined as the fraction of misclassified samples.

    Arguments:
    Ypred -- Predicted label, a numpy array of size (1,m)
    Y -- ground truth labels, a numpy array with shape (1,m)

    Returns:
    error_rate -- an array of shape (1,m)
    """
    Ypredargmax = np.argmax(Ypred, axis=0)
    return np.sum(Y != Ypredargmax) / Y.size
```

1.2.5 Initialize and Optimize (Learn)

Initialize Parameters First we provide a utility method to generate properly intialized parameters.

[15]:
```
def initialize_params(nx, ny, random=False):
    """
    This function provides initialized parameters: a weights matrix and a bias␣
 ↪vector.

    Argument:
    nx -- number of input features
    ny -- number of output dimensions (number of different labels)
    random -- if set to True standard normal distributed values are set for the␣
 ↪weights; otherwise zeros are used.

    Returns:
    w -- initialized weights matrix of shape (ny,nx)
    b -- initialized bias vector of shape (ny,1) - always initialized with␣
 ↪zeros
```

```
    """
    if random:
        w = np.random.randn(*(ny,nx)) / np.sqrt(nx)
    else:
        w = np.zeros((ny,nx))

    b = 0.0

    return w, b
```

Metrics Class For not littering the optimization loop with code to keep track of the learning results over the epochs we defined a suitable metrics object that keeps all the data (cost function, classification error vs epochs). It also provides utility methods for updating, printing values or plotting the learning curves.

It is defined as python class the metrics object then needs to be instantiated from. It means that some small knowledge about object-oriented programming is needed here.

[16]:
```
class Metrics():
    """
    Allows to collect statistics (such as classification error or cost) that␣
↪are of interest over the course of training
    and for creating learning curves that are a useful tool for analyzing the␣
↪quality of the learning.
    """

    def __init__(self, cost, smooth=False):
        """
        Constructor for a metrics object.
        Initializes all the statistics to track in form of python lists.

        Parameters:
        cost -- cost function to use (a python function)
        smooth -- if set to true updates learning curve after each training␣
↪step and also provides learning curves
        smoothed over the epoch
        """
        self.epochs = []
        self.smooth = smooth
        self.train_costs_last = []
        self.test_costs_last = []
        self.train_errors_last = []
        self.test_errors_last = []
        self.stepsize_w_last = []
        self.stepsize_b_last = []
        if self.smooth:
            self.train_costs_smoothed = []
            self.test_costs_smoothed = []
```

```
            self.train_errors_smoothed = []
            self.test_errors_smoothed = []
            self.stepsize_w_smoothed = []
            self.stepsize_b_smoothed = []

        self.cost_function = cost
        self.init_epoch()


    def init_epoch(self):
        self.train_costs_epoch = []
        self.test_costs_epoch = []
        self.train_errors_epoch = []
        self.test_errors_epoch = []
        self.stepsize_w_epoch = []
        self.stepsize_b_epoch = []


    def update_epoch(self, epoch):
        """
        Computes the average of the metrics over the epoch and adds the result␣
↪to the per epoch history

        Parameters:
        epoch -- the epoch to add to the per epoch cache
        """
        self.epochs.append(epoch)
        if self.smooth:
            self.train_costs_smoothed.append(np.mean(self.train_costs_epoch))
            self.test_costs_smoothed.append(np.mean(self.test_costs_epoch))
            self.train_errors_smoothed.append(np.mean(self.train_errors_epoch))
            self.test_errors_smoothed.append(np.mean(self.test_errors_epoch))
            self.stepsize_w_smoothed.append(np.mean(self.stepsize_w_epoch))
            self.stepsize_b_smoothed.append(np.mean(self.stepsize_b_epoch))

        self.train_costs_last.append(self.train_costs_epoch[-1])
        self.test_costs_last.append(self.test_costs_epoch[-1])
        self.train_errors_last.append(self.train_errors_epoch[-1])
        self.test_errors_last.append(self.test_errors_epoch[-1])
        self.stepsize_w_last.append(self.stepsize_w_epoch[-1])
        self.stepsize_b_last.append(self.stepsize_b_epoch[-1])

        self.init_epoch()


    def update_iteration(self, ypred_train, y_train, ypred_test, y_test, dw,␣
↪db):
```

```
        """
        Allows to update the statistics to be tracked for a new epoch.
        The cost is computed by using the function object passed to the␣
↪constructor.

        Parameters:
        epoch -- Epoch
        ypred_train -- predicted values on the training samples, a numpy array␣
↪of shape (1,m1)
        y_train -- ground truth labels associated with the training samples, a␣
↪numpy array of shape (1,m1)
        ypred_test -- predicted values on the test samples, a numpy array of␣
↪shape (1,m2)
        y_test -- ground truth labels associated with the test samples, a numpy␣
↪array of shape (1,m2)
        dw -- some lenght measure for the gradient w.r.t. the weights, a numpy␣
↪array of shape (1,n)
        db -- gradient w.r.t. the bias, a scalar
        """
        Jtrain = self.cost_function(ypred_train, y_train)
        Jtest = self.cost_function(ypred_test, y_test)
        train_error = error_rate(ypred_train, y_train)
        test_error = error_rate(ypred_test, y_test)

        self.train_costs_epoch.append(Jtrain)
        self.test_costs_epoch.append(Jtest)
        self.train_errors_epoch.append(train_error)
        self.test_errors_epoch.append(test_error)
        self.stepsize_w_epoch.append(dw)
        self.stepsize_b_epoch.append(db)


    def print_latest_errors(self):
        print ("Train/test error after epoch %i: %f, %f" %(self.epochs[-1],␣
↪self.train_errors_last[-1], self.test_errors_last[-1]))

    def print_latest_costs(self):
        print ("Train/test cost after epoch %i: %f, %f" %(self.epochs[-1], self.
↪train_costs_last[-1], self.test_costs_last[-1]))

    def plot_cost_curves(self, ymin=None, ymax=None, smooth=True):
        plt.semilogy(self.epochs, self.train_costs_last, "b-", label="train")
        plt.semilogy(self.epochs, self.test_costs_last, "r-", label="test")
        if self.smooth and smooth:
            plt.semilogy(self.epochs, self.train_costs_smoothed, "b--",␣
↪label="train_smoothed")
```

```
            plt.semilogy(self.epochs, self.test_costs_smoothed, "r--",␣
↪label="test_smoothed")
        plt.ylabel('Cost')
        plt.xlabel('Epochs')
        xmax = self.epochs[-1]
        if not ymin:
            ymin = min(max(1e-5,np.min(self.train_costs_last)),max(1e-5,np.
↪min(self.test_costs_last))) * 0.8
        if not ymax:
            ymax = max(np.max(self.train_costs_last),np.max(self.
↪test_costs_last)) * 1.2
        plt.axis([0,xmax,ymin,ymax])
        plt.legend()
        plt.show()

    def plot_error_curves(self, ymin=None, ymax=None, smooth=True):
        plt.semilogy(self.epochs, self.train_errors_last, "b", label="train")
        plt.semilogy(self.epochs, self.test_errors_last, "r", label="test")
        if self.smooth and smooth:
            plt.semilogy(self.epochs, self.train_errors_smoothed, "b--",␣
↪label="train_smoothed")
            plt.semilogy(self.epochs, self.test_errors_smoothed, "r--",␣
↪label="test_smoothed")
        plt.ylabel('Errors')
        plt.xlabel('Epochs')
        xmax = self.epochs[-1]
        if not ymin:
            ymin = min(max(1e-5,np.min(self.train_errors_last)),max(1e-5,np.
↪min(self.test_errors_last))) * 0.8
        if not ymax:
            ymax = max(np.max(self.train_errors_last),np.max(self.
↪test_errors_last)) * 1.2
        plt.axis([0,xmax,ymin,ymax])
        plt.legend()
        plt.show()

    def plot_stepsize_curves(self, ymin=None, ymax=None, smooth=True):
        plt.semilogy(self.epochs, self.stepsize_w_last, label="dw")
        plt.semilogy(self.epochs, self.stepsize_b_last, label="db")
        if self.smooth and smooth:
            plt.semilogy(self.epochs, self.stepsize_w_smoothed, label="dw--")
            plt.semilogy(self.epochs, self.stepsize_b_smoothed, label="db--")
        plt.ylabel('Step Sizes (dw,db)')
        plt.xlabel('Epochs')
        xmax = self.epochs[-1]
        if not ymin:
```

```
            ymin = min(max(1e-5,np.min(self.stepsize_w_last)),max(1e-5,np.
↪min(self.stepsize_b_last))) * 0.8
        if not ymax:
            ymax = max(np.max(self.stepsize_w_last),np.max(self.
↪stepsize_b_last)) * 1.2
        plt.axis([0,xmax,ymin,ymax])
        plt.legend()
        plt.show()
```

[17]:
```
class MiniBatches():

    def __init__(self, x, y, batchsize):
        self.x = x
        self.y = y
        m = x.shape[1]
        if not batchsize:
            self.batchsize = m
        else:
            self.batchsize = batchsize
        self.n = x.shape[0]
        self.mb = int(m/batchsize)
        self.indices = np.arange(m)
        np.random.shuffle(self.indices)
        self.ib = 0

    def number_of_batches(self):
        return self.mb

    def next(self):
        it = self.indices[self.ib*batchsize:(self.ib+1)*batchsize]
        xbatch = self.x[:,it].reshape(self.n,batchsize)
        ybatch = self.y[:,it].reshape(1,batchsize)
        self.ib += 1
        return xbatch, ybatch
```

Optimisation

[18]:
```
def optimize(W, b, x_train, y_train, x_test, y_test, nepochs, alpha,␣
↪batchsize=32, debug=True):
    """
    This function optimizes W and b by running (mini-batch) gradient descent.␣
↪It starts with the given
    weights as initial values and then iteratively updates the parameters for␣
↪nepochs number of times.
    It returns the trained parameters as dictionary (keys "W" and "b") and␣
↪various quantities
```

```
    collected during learning in form of a Metrics object. You don't need to␣
↪provide smoothingăwithin
    the epoch so that training will be somewhat faster.

    Arguments:
    W -- weights, a numpy array of size (ny,nx)
    b -- biases, a numpy array with shape (ny,1) (with ny=10 for MNIST).
    x_train -- input data for training of shape (nx,m1)
    y_train -- ground-truth labels - a numpy array with shape (1,m1)
    x_test -- input data for training of shape (nx,m2)
    y_test -- ground-truth labels - a numpy array with shape (1,m2)
    nepochs -- number of iterations of the optimization loop
    alpha -- learning rate of the gradient descent update rule
    batchsize -- batch size, defaults to 32
    debug -- if true prints training and test error values after each epoch.␣
↪Defaults to True.

    Returns:
    params -- dictionary containing the (final) weights w and bias b
    metrics -- contain the information about the learning curves
    """
    metrics = Metrics(cost = cost, smooth=False)

    m = x_train.shape[1] # number of samples
    nx = x_train.shape[0] # number of input features
    mb = int(m/batchsize) # number of mini-batches
    print("Optimisation with batchsize %i and %i number of batches per epoch.
↪"%(batchsize,mb))

    # compute and set the initial values for the metrics curves
    ypred_train = predict(W,b,x_train)
    ypred_test = predict(W,b,x_test)
    metrics.update_iteration(ypred_train, y_train, ypred_test, y_test, 0, 0)
    metrics.update_epoch(0)

    # Loop over the epochs
    for i in range(nepochs):

        # prepare shuffled mini-batches for this epoch
        batches = MiniBatches(x_train, y_train, batchsize)

        ### START YOUR CODE ###
        for j in range(batches.number_of_batches()):
            xbatch, ybatch = batches.next()
            ypred_train = predict(W, b, xbatch)
            gradJ = gradient(xbatch, ybatch, ypred_train)
            W = W - (alpha * gradJ['dW'])
```

```
            b = b - (alpha * gradJ['db'])

        ypred_train = predict(W, b, x_train)
        ypred_test = predict(W, b, x_test)

        dw_norm = np.linalg.norm(gradJ['dW'])
        db_norm = np.linalg.norm(gradJ['db'])

        metrics.update_iteration(ypred_train, y_train, ypred_test, y_test,␣
↪dw_norm, db_norm)
        metrics.update_epoch(i)
        ### END YOUR CODE ###

        if debug:
            metrics.print_latest_errors()

    metrics.print_latest_costs()
    metrics.print_latest_errors()

    return {"W": W, "b": b}, metrics
```

1.2.6 Run the Training for Specific Setting

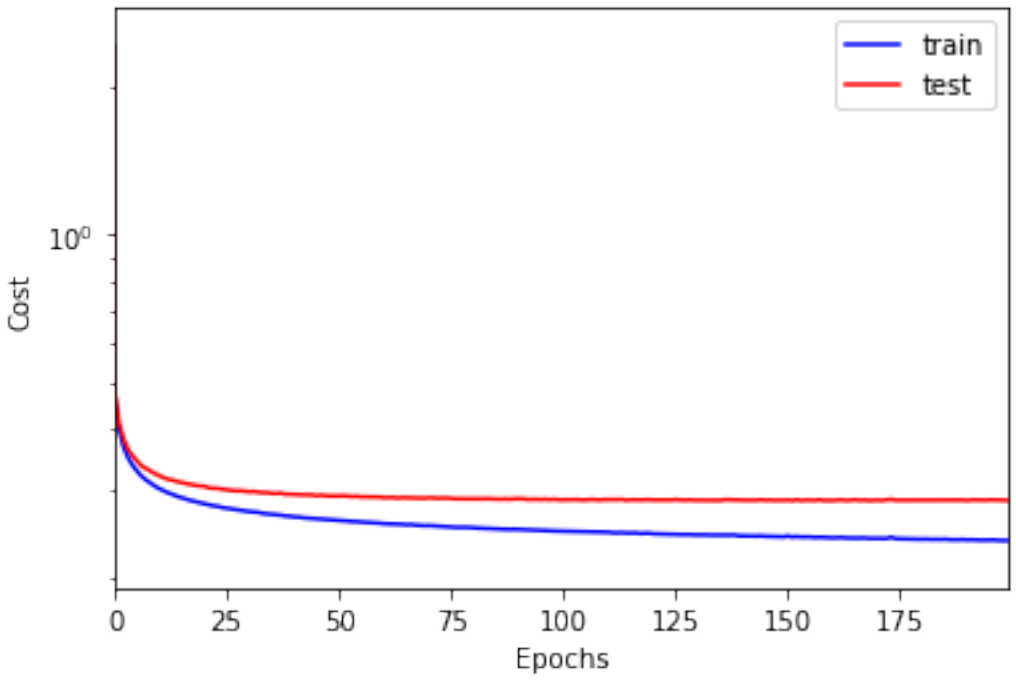
[19]:
```
# preparing the data
x,y, shape = load_mnist(data_home)
x_train1, x_test1, y_train, y_test = prepare_train_test(x, y, test_size=0.20)
x_train,x_test = normalize(x_train1,x_test1)
```

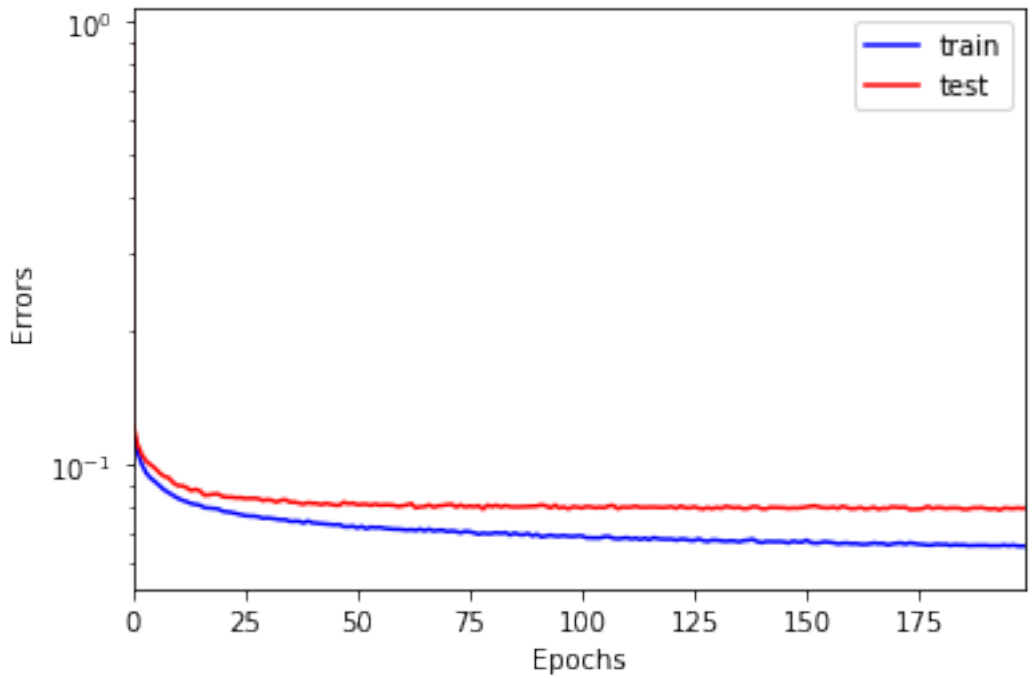
```
Loaded MNIST original:
Image Data Shape (784, 70000)
Label Data Shape (1, 70000)
Shape training set:  (784, 56000) (1, 56000)
Shape test set:      (784, 14000) (1, 14000)
```

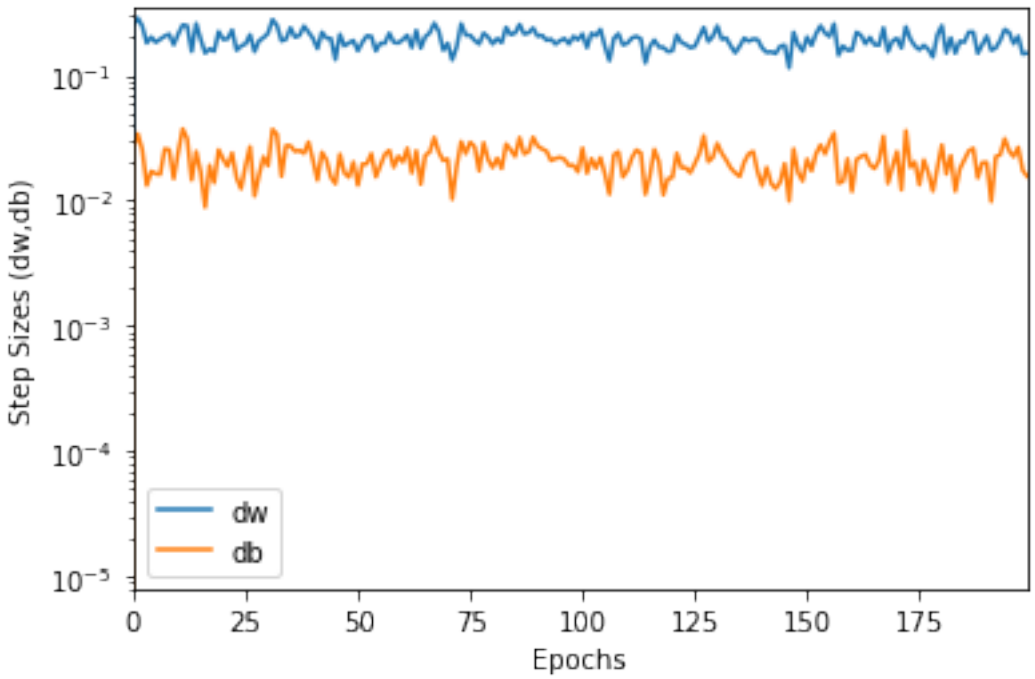
[54]:
```
### START YOUR CODE ### -- try different settings, ...


def train(hyper_params):
    learning_rate = hyper_params['learning_rate']
    nepochs = hyper_params['nepochs']
    batchsize = hyper_params['batchsize']
    print('alpha:', learning_rate, 'nepochs:', nepochs, 'batchsize:', batchsize)
    W,b = initialize_params(28*28, 10, random=True)
    params, metrics = optimize(W, b, x_train, y_train, x_test, y_test,
                               nepochs=nepochs,
                               alpha=learning_rate,
                               batchsize=batchsize,
```

```
                                debug=False)
    metrics.plot_cost_curves()
    metrics.plot_error_curves()
    metrics.plot_stepsize_curves()
    print('\n\n')


# tuning parameters to let them run. careful, execution time is super long the␣
 ↪more params you have!
hyper_params = {
    'learning_rate': [0.1],
    'nepochs': [200],
    'batchsize': [256]
}

for learning_rate in hyper_params['learning_rate']:
    for nepochs in hyper_params['nepochs']:
        for batchsize in hyper_params['batchsize']:
            train({'learning_rate': learning_rate, 'nepochs': nepochs,␣
 ↪'batchsize': batchsize})
### END YOUR CODE ###
```

```
alpha: 0.1 nepochs: 200 batchsize: 256
Optimisation with batchsize 256 and 218 number of batches per epoch.
Train/test cost after epoch 199: 0.237729, 0.286977
Train/test error after epoch 199: 0.065214, 0.079429
```



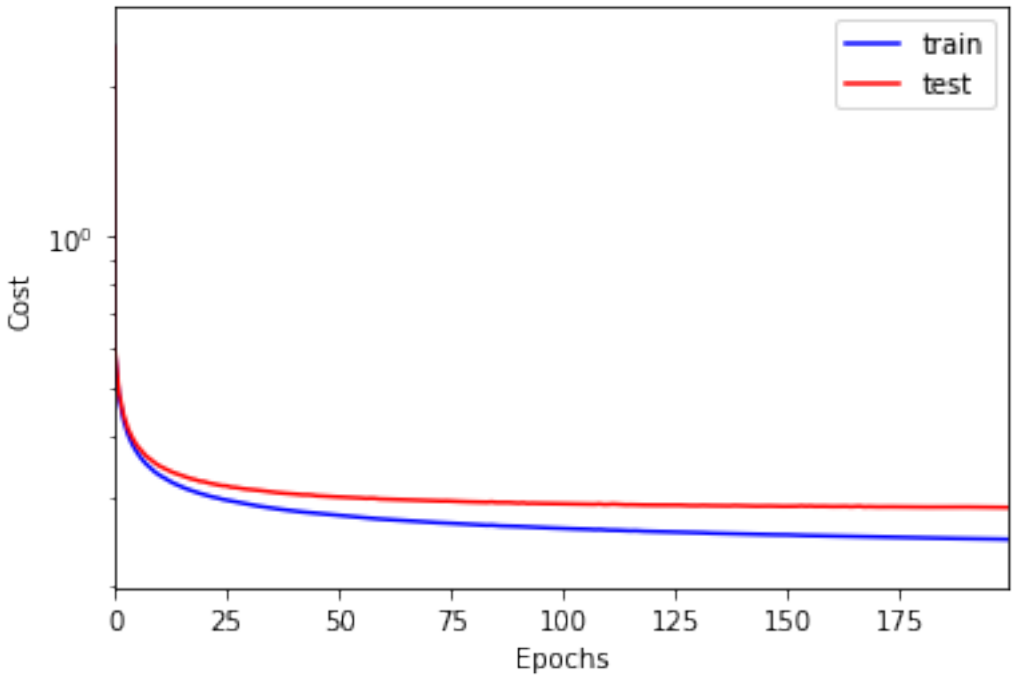

train
test
Errors
Epochs
10^{0}
10^{-1}
0
25
50
75
100
125
150
175


dw
db
Step Sizes (dw,db)
Epochs
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
0
25
50
75
100
125
150
175

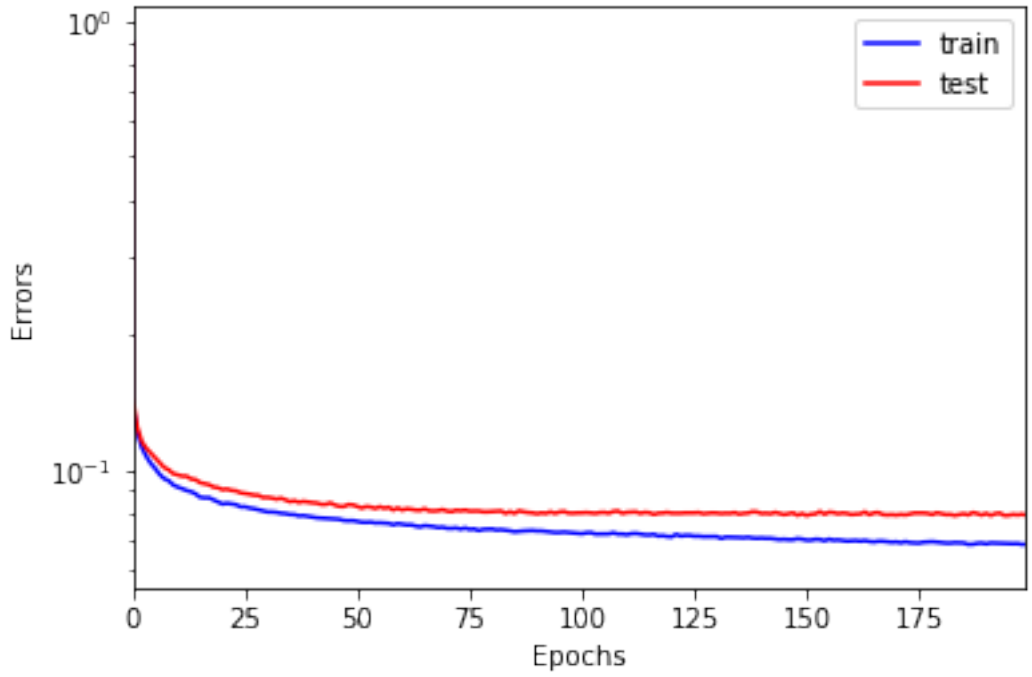
1.2.7 Plot Learning Curves

Cost Error Rate Learning Speed (Lenght of Parameter Change)

[55]:
```
metrics.plot_cost_curves()
```

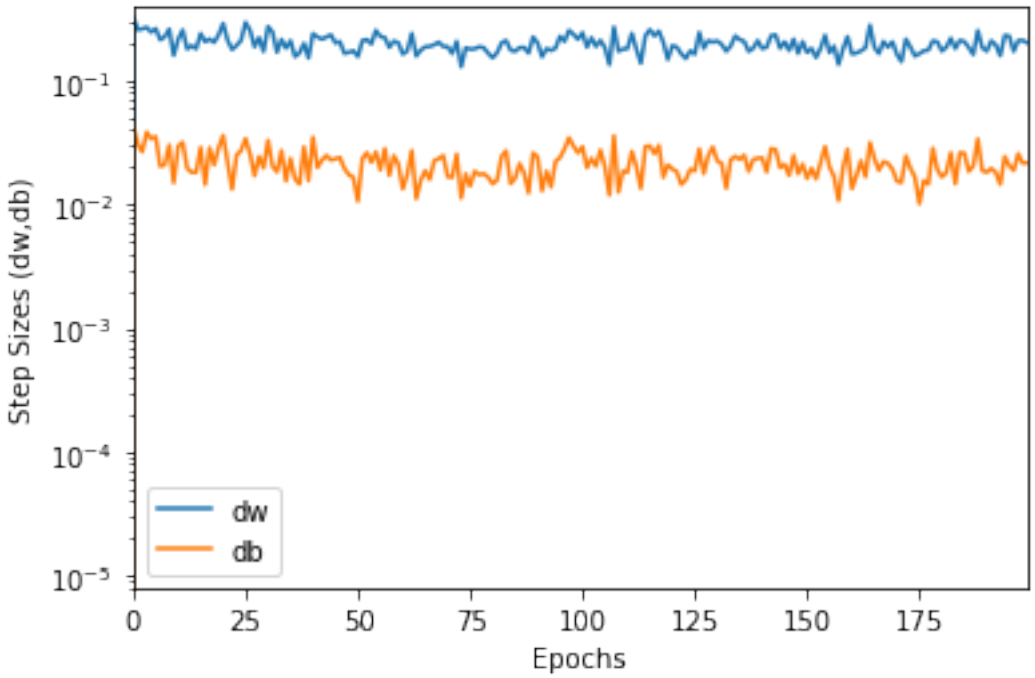


[56]:
```
metrics.plot_error_curves()
metrics.print_latest_errors()
```




```
Train/test error after epoch 199: 0.068304, 0.079214
```

[57]:
```
metrics.plot_stepsize_curves()
```



1.2.8 Describe your Findings for Exercise 1b

By running the training with different settings for learning rate, number of epochs, batch size explore which combination is best suited to obtain good test performance. Keep an eye on random estimates for the error rates due to random parameter initialisation and randomly shuffled mini-batches.

Specify your choice of these hyper-parameters and justify why you consider your choice best suited.

YOUR FINDINGS ...

- `learning_rate`: Smaller learning rates output better performance but are slower. At learning rates of about 0.5 it starts to wiggle a bit and 10 it's absolutely unusable. We would choose `0.1`.
- `nepochs`: The longer we train, the more we overfit. We would stop training when train- and test-error start to deviate too much. This has shown to be at around `200`.
- `batchsize`: The lower we choose the batch size, the lower is the train cost. But the deviation of train and test error is larger when having a small batch size. So we would choose `256` where it is relatively stable but still low.

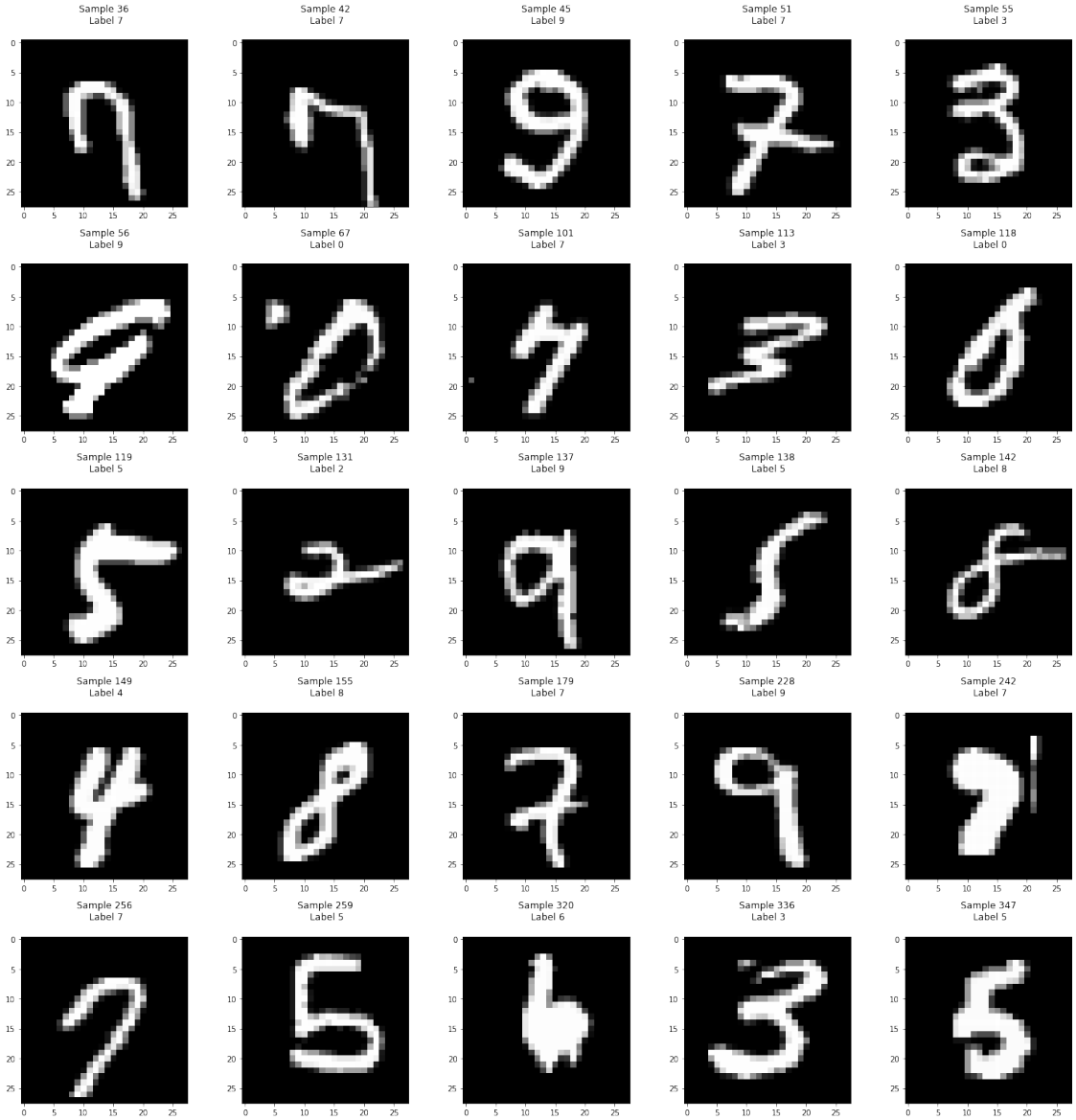
According to literature, when we increase batch size by `k` we should increase the learning rate by either $\sqrt{batchsize}$ [1] or kN [2].

- [1] Krizhevsky. One weird trick for parallelizing convolutional neural networks: https://arxiv.org/abs/1404.5997
- [2] P.Goyal et al.: Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour https://arxiv.org/abs/1706.02677

Plot the Misclassified Images

[58]:
```
y_pred = predict(params['W'], params['b'], x_test)
yhat = np.argmax(y_pred, axis=0)
indices = np.where(yhat != y_test)[1]
print(len(indices))

plot_digits(x_test, y_test, indices[0:25], shape)
print(y_test[:,indices[0:25]])
print(yhat[indices[0:25]])
```

```
1109
[[7 7 9 7 3 9 0 7 3 0 5 2 9 5 8 4 8 7 9 7 7 5 6 3 5]]
[9 4 8 2 5 5 3 9 2 3 8 4 7 1 1 9 3 9 3 8 9 6 9 5 6]
```



1.2.9 Plot the Trained Weights as Image

[59]:
```
weights = params['W']
biases = params['b']
cols = 5
rows = 2
plt.figure(figsize=(20,4*rows))
for i in range(10):
    plt.subplot(rows, cols, i+1)
    plt.imshow(np.reshape(weights[i], (28,28)), cmap=plt.cm.gray)
    plt.title('Digit %i'%i, fontsize = 12)
```

```
plt.figure(figsize=(20,4))
plt.plot(range(10), [biases[i] for i in range(10)], '+')
```

[59]: `[<matplotlib.lines.Line2D at 0x1a443297f0>]`