Lab2

November 1, 2018

1 ECE 285 Assignment #2

2 Python, Numpy and Matplotlib

3 Mehrnaz Motamed

4 A53245061

---

5 1. Getting started

```
In [1]: import numpy as np
        from matplotlib import pyplot
```

6 2. Read MNIST Data

```
In [2]: import MNISTtools
```

```
In [3]: help(MNISTtools.load)
        help(MNISTtools.show)
```

```
Help on function load in module MNISTtools:

load(dataset='training', path='.')
    Import either the training or testing MNIST data set.
    It returns a pair with the first element being the collection of
    images stacked in columns and the second element being a vector
    of corresponding labels from 0 to 9.

    Example:
        x, lbl = load(dataset = "training", path = "/datasets/MNIST")

Help on function show in module MNISTtools:

show(image)
```

```
Render a given MNIST image provided as a column vector.

Example:
    x, lbl = load(dataset = "training", path = "/datasets/MNIST")
    show(x[:, 0])
```

```
In [4]: MNISTtools.load
```

```
Out[4]: <function MNISTtools.load>
```

```
In [5]: xtrain, ltrain = MNISTtools.load(dataset = "training", path = "/datasets/MNIST")
```

```
In [6]: xtrain.shape
```

```
Out[6]: (784, 60000)
```

```
In [7]: ltrain.shape
```

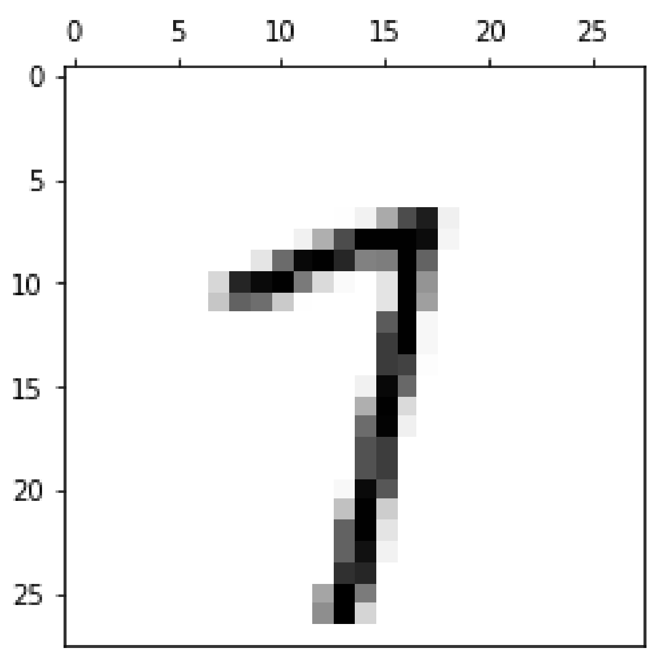
```
Out[7]: (60000,)
```

6.0.1 1. What are the shapes of both variables? What is the size of the training dataset? What is the feature dimension?

The shape of xtain is (784, 60000) while the shape of ltrain is (60000,). The training data set has the size of 60000 samples with the feature dimension of 784.

6.0.2 2. Display the image of index 42 and check that its content corresponds to its label.

```
In [8]: MNISTtools.show(xtrain[:, 42])
```



```
In [9]: ltrain[42,]
```

```
Out[9]: 7
```

6.0.3 3. What is the range of xtrain (minimum and maximum values)? What is the type of xtrain?

```
In [10]: type(xtrain)
```

```
Out[10]: numpy.ndarray
```

```
In [11]: np.amax(np.amax(xtrain,axis=0),axis=0)
```

```
Out[11]: 255
```

```
In [12]: np.amax(np.amin(xtrain,axis=0),axis=0)
```

```
Out[12]: 0
```

```
In [13]: print(np.amax(np.amax(xtrain,axis=0),axis=0)-np.amax(np.amin(xtrain,axis=0),axis=0))
```

```
255
```

xtrain is a 784 by 60000 array that shows numbers from 0 to 9. Each image in xtrain (each pixel) has a value between 0 to 255. So the range of xtrain is 255

6.0.4 4. Create a function that takes a collection of images (such as xtrain) and return a modified version in the range [1, 1] of type float64. Update xtrain accordingly.

```
In [14]: from __future__ import division
         def normalize_MNIST_images(x):
             x = x.astype(np.float64)
             min_x = min(x.flatten())
             range_x = max(x.flatten()) - min_x
             x = 2*(x - min_x) / range_x - 1
             return x
```

```
In [15]: xtrain = normalize_MNIST_images(xtrain)
```

6.0.5 5. Using integer array indexing, complete the following function

```
In [16]: def label2onehot(lbl):
             d = np.zeros((lbl.max() + 1, lbl.size))
             d[lbl, np.arange(0, lbl.size)] = 1
             return d
```

In [17]:
```
dtrain = label2onehot(ltrain)
```

In [18]:
```
dtrain[:,42]
```

Out[18]:
```
array([0., 0., 0., 0., 0., 0., 0., 1., 0., 0.])
```

In [19]:
```
ltrain[42]
```

Out[19]:
```
7
```

Both dtrain[:,42] and ltrain[42] correspond to 7.

In [20]:
```
dtrain.shape
```

Out[20]:
```
(10, 60000)
```

6.0.6 6. Complete the following function such that *ltrain* == *onehot2label(dtrain)*.

In [21]:
```
def onehot2label(d):
    lbl = d.argmax(axis=0)
    return lbl
```

In [22]:
```
onehot2label(dtrain)[42]
```

Out[22]:
```
7
```

In [23]:
```
onehot2label(dtrain).shape
```

Out[23]:
```
(60000,)
```

7 3. Activation functions

7.0.1 7. Create a function that returns an array whose columns are the 60000 predictions y from an array whose columns are the 60000 vectors a.

In [24]:
```
from __future__ import division
def softmax(a):
    M = a.max(axis=0)
    difference = a - a.max(axis=0)
    y = np.exp(difference) / np.exp(difference).sum(axis=0)
    return y
```

7.0.2 8. Show that $\frac{g(a)_i}{a_i} = g(a)_i(1g(a)_i)$.

$g(a_i) = \frac{e^{a_i - M}}{\sum_{j=1}^{10} e^{a_j - M}}$ $\frac{\partial g(a_i)}{\partial a_i} = \frac{\frac{\partial e^{a_i - M}}{\partial a_i} \sum_{j=1}^{10} e^{a_j - M} - \frac{\partial \sum_{j=1}^{10} e^{a_j - M}}{\partial a_i} e^{a_i - M}}{(\sum_{j=1}^{10} e^{a_j - M})^2} = \frac{e^{a_i - M} \sum_{j=1}^{10} e^{a_j - M} - (e^{a_i - M})^2}{(\sum_{j=1}^{10} e^{a_j - M})^2} = \frac{e^{a_i - M}}{\sum_{j=1}^{10} e^{a_j - M}} . \frac{\sum_{j=1}^{10} e^{a_j - M} - e^{a_i - M}}{\sum_{j=1}^{10} e^{a_j - M}} = \frac{e^{a_i - M}}{\sum_{j=1}^{10} e^{a_j - M}} . (1 - \frac{e^{a_i - M}}{\sum_{j=1}^{10} e^{a_j - M}}) = g(a_i).(1 - g(a_i))$

7.0.3 9. Show that $\frac{\partial g(a)_i}{\partial a_j} = g(a)_i g(a)_j \quad for\, j \neq i$

$for\, i \neq j: \quad \frac{\partial g(a_i)}{\partial a_j} = \frac{\frac{\partial e^{a_i - M}}{\partial a_j}\sum_{j=1}^{10} e^{a_j - M} - \frac{\partial \sum_{j=1}^{10} e^{a_j - M}}{\partial a_j} e^{a_i - M}}{(\sum_{j=1}^{10} e^{a_j - M})^2} = \frac{0 - e^{a_i - M} e^{a_j - M}}{(\sum_{j=1}^{10} e^{a_j - M})^2} = -\frac{e^{a_i - M}}{(\sum_{j=1}^{10} e^{a_j - M})^2} \cdot \frac{e^{a_j - M}}{(\sum_{j=1}^{10} e^{a_j - M})^2}$ $=- g(a)_ig(a)_j $

7.0.4 10.a. From the previous question, deduce that the Jacobian of softmax is symmetric

$J(g) = \frac{\partial g(a)}{\partial a}$ $J(g)_{ij} = \frac{\partial g(a)_i}{\partial a_j} = -g(a)_i . g(a)_j = -g(a)_j . g(a_i) = \frac{\partial g(a)_j}{\partial a_i} = J(g)_{ji}$ *Therefore, the Jacobian of softmax is symmetric.*

7.0.5 b. From $= \frac{\partial g(a)}{\partial a}^T e$ Deduce that $= g(a) e g(a), e g(a)$ where is the element-wise product.

$\delta = \frac{\partial g(a)^T}{\partial a} e$ $\delta i : i^{th}\, row\, of\, \delta : \delta i = g(a)_i (1 - g(a)_i) e_i - \sum_{j \neq i} g(a)_i g(a)_j e_j = g(a)_i e_i - g(a)_i g(a)_i e_i - \sum_{j \neq i} g(a)_i g(a)_j e_j = g(a)_i e_i - g(a)_i (\sum_j g(a)_j e_j) = g(a)_i e_i - g(a)_i < g(a), e >$ *Therefore,* $= g(a) e g(a), e g(a)$

7.0.6 c. Based on this formula, write a function that gets a and e and generates δ

```
In [25]: def softmaxp(a, e):
             g_a = softmax(a)
             gg = g_a * e
             delta = gg - gg.sum(axis=0) * g_a
             return delta
```

7.0.7 11. Complete the following script to check your function softmaxp as follows.

7.0.8 $\delta = lim_{\epsilon \to 0} \frac{g(a + \epsilon e) - g(a)}{\epsilon}$

```
In [26]: eps = 1e-6                           # finite difference step
         a = np.random.randn(10, 200)         # random inputs
         e = np.random.randn(10, 200)         # random directions
         diff = softmaxp(a, e)
         diff_approx = np.true_divide((softmax(a+eps*e)-softmax(a)),eps)
         rel_error = np.abs(diff - diff_approx).mean() / np.abs(diff_approx).mean()
         print(rel_error, 'should be smaller than 1e-6')
```

```
(4.874945246311934e-07, 'should be smaller than 1e-6')
```

7.0.9 12. For the hidden layers, we will be using $ReLU(a)_i = max(a_i, 0)$. Write two functions:

```
In [27]: def relu(a):
             return a * (a > 0)
         relu(np.array([1,2,-3]))

Out[27]: array([1, 2, 0])
```

```
In [28]: def relup(a, e):
             return e*(a > 0) / np.linalg.norm(e, ord=2)
```

8 4. Backpropagation

8.0.1 13. Use the following function to create/initialize your shallow network as follows.

```
In [29]: def init_shallow(Ni, Nh, No):
             b1 = np.random.randn(Nh, 1) / np.sqrt((Ni+1.)/2.)
             W1 = np.random.randn(Nh, Ni) / np.sqrt((Ni+1.)/2.)
             b2 = np.random.randn(No, 1) / np.sqrt((Nh+1.))
             W2 = np.random.randn(No, Nh) / np.sqrt((Nh+1.))
             return W1, b1, W2, b2
```

```
In [30]: Ni = xtrain.shape[0]
         Nh = 64
         No = dtrain.shape[0]
         netinit = init_shallow(Ni, Nh, No)
```

8.0.2 14. Complete the function forwardprop shallow to evaluate the prediction of our initial network:

```
In [32]: def forwardprop_shallow(x, net):
             W1 = net[0]
             b1 = net[1]
             W2 = net[2]
             b2 = net[3]
             a1 = W1.dot(x) + b1
             h1 = relu(a1)
             a2 = W2.dot(h1) + b2
             y = softmax(a2)
             #print("W2 old is", W2[:,0:2])
             return y
```

8.0.3 15. Complete the function eval loss:

```
In [33]: def eval_loss(y, d):
             loss = (-1)* np.average(np.log(y) * d)
             return loss
         print(eval_loss(yinit, dtrain), 'should be around .26')
```

```
(0.28533054802674385, 'should be around .26')
```

8.0.4 16. Complete the function eval_perfs that given your predictions y and the desired labels lbl, computes the percentage of misclassified samples. Interpret the result.

```
In [34]: def eval_perfs(y, lbl):
             y_choice = 1*(y==np.amax(y,axis=0))
```

```
        error = 50 * np.average(np.absolute(y_choice-label2onehot(lbl)))
        return error
    print(eval_perfs(yinit, ltrain))
    # print ltrain[42,]
    # print(label2onehot(ltrain)[:,42])
    # y_choice = 1*(yinit==np.amax(yinit,axis=0))
    # print(np.absolute(y_choice-label2onehot(ltrain))[:,42])
    # print(eval_perfs(yinit, ltrain))
```

```
9.1605
```

9% error. It is different from eval_loss because the error has been quantized.

8.0.5 17. Complete the following function update shallow

```
In [35]: def update_shallow(x, d, net, gamma=0.05):
             W1 = net[0]
             b1 = net[1]
             W2 = net[2]
             b2 = net[3]
             Ni = W1.shape[1]
             Nh = W1.shape[0]
             No = W2.shape[0]
             #print Ni
             #gamma = gamma * 50
             #gamma = gamma / x.shape[1] # normalized by the training dataset size
             a1 = W1.dot(x) + b1
             #print np.sum(a1,axis=0)
             h1 = relu(a1)
             a2 = W2.dot(h1) + b2
             y = softmax(a2)
             e = eval_loss(y, d)
             #print e
             delta2 = softmaxp(a2, e)
         #     print W2.T.dot(delta2).shape
         #     print a1.shape
             #print np.sum(delta2,axis=0)
             delta1 = relup(a1,W2.T.dot(delta2))
             #print np.sum(delta1,axis=0)
             W2 = W2 - gamma * delta2.dot(h1.T)
             #print("W2 new is", W2[:,0:2])
             W1 = W1 - gamma * delta1.dot(x.T)
             b2 = b2 - gamma * delta2.sum(axis=1).reshape(10, 1)
             b1 = b1 - gamma * delta1.sum(axis=1).reshape(64, 1)
         #     print (np.sum(b1-net[1]))
         #     print (np.sum(b2-net[3]))
         #     print (np.sum(W1-net[0]))
```

```
#     print (np.sum(W2-net[2]))
#     print (np.sum(delta1,axis=0))
#     print gamma
    return W1, b1, W2, b2
```

```
In [36]: net_array = update_shallow(xtrain, dtrain, netinit, gamma=0.05)
```

8.0.6 Show that $(\nabla_y E)_i = -d_i/y_i$

From the previous section we have: $E = -\sum_{i=1}^{10} d_i log y_i$ *Taking the derivative of E with respect to* y_i *will result in the following:* $\partial E/\partial y_i = -d_i/y_i$ Thus, $(\nabla_y E)_i = -d_i/y_i$

8.0.7 18. Using update shallow, complete the function backprop shallow.

```
In [37]: def backprop_shallow(x, d, net, T, gamma=0.05):
             lbl = onehot2label(d)
             for t in range(0, T):
                     y = forwardprop_shallow(x, net)
                     #print("y is", y[:,0:5])
                     #print(eval_loss(y, d))
                     #print(eval_perfs(y, lbl))
                     #gamma = gamma / x.shape[1]
                     net = update_shallow(x, d, net, gamma=0.05)
                     gamma = gamma * x.shape[1]
                     y = forwardprop_shallow(x, net)
                     print(eval_loss(y, d))
                     print(eval_perfs(y, lbl))
             return net
```

```
In [38]: net1 = backprop_shallow(xtrain, dtrain, netinit, 10)
```

```
0.37476193470059477
9.434
1.499068983138514
9.007
0.3326158604731642
9.129333333333333
0.31798476186778346
8.812000000000001
0.3314409310297138
9.121166666666666
0.42628100998452967
8.799166666666666
0.3997652801302264
8.567333333333334
0.3220764579555972
9.0305
0.6472344840254411
9.007
```

```
0.5476347172173387
9.007
```