Assignment 1

October 16, 2019

SHAKEEL AHAMED MANSOOR SHAIKNA - A53315574

Read MNIST data

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

Question 1

[2]:
```
# Loading the training datasets

xtrain, ltrain = MNISTtools.load(dataset="training", path=None)

print("Shape of xtrain is ", np.shape(xtrain))
print("Shape of ltrain is ", np.shape(ltrain))
print("Size of training dataset is ", xtrain.shape[1])
print("Feature dimension is ", xtrain.shape[0])
```

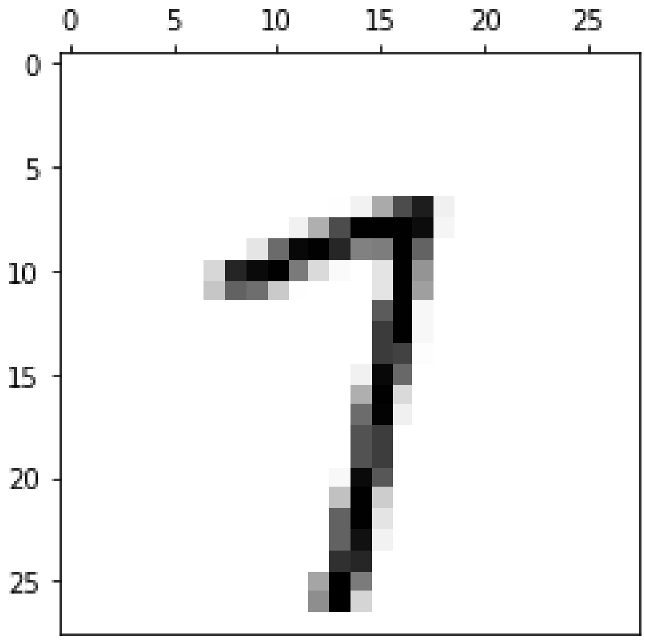
```
Shape of xtrain is  (784, 60000)
Shape of ltrain is  (60000,)
Size of training dataset is  60000
Feature dimension is  784
```

Question 2

[3]:
```
# Displaying an image from the training dataset

print("Image of index 42:")
MNISTtools.show(xtrain[:, 42])
print("Label of the above image is ", ltrain[42])
```

```
Image of index 42:
```




```
Label of the above image is  7
```

Question 3

[4]:
```
# Finding the range of xtrain

xtrain_max = np.max(xtrain)
xtrain_min = np.min(xtrain)
print("Range of xtrain is [", xtrain_min, ", ", xtrain_max, "]")
print("Type of xtrain is ",type(xtrain))

xtrain = xtrain.astype(np.float32)  # Converting array type from int to float
```

```
Range of xtrain is [ 0 ,  255 ]
Type of xtrain is  <class 'numpy.ndarray'>
```

Question 4

[5]:
```
# Normalizing the datasets

def normalize_MNIST_images(x):
    x = -1 + (2*x/255)
    return x

# Checking the normalize_MNIST_images() function

xtrain = normalize_MNIST_images(xtrain)
```

```
xtrain_min = np.min(xtrain)
xtrain_max = np.max(xtrain)
print("Range of normalized xtrain is [", xtrain_min, ", ", xtrain_max, "]")
```

```
Range of normalized xtrain is [ -1.0 ,  1.0 ]
```

Question 5

[6]:
```
# Creating one-hot codes for the labels

def label2onehot(lbl):
    d = np.zeros((lbl.max() + 1, lbl.size))
    for i in range(lbl.max()):
        d[lbl, np.arange(lbl.size)] = 1
    return d

# Checking the label2onehot() function

dtrain = label2onehot(ltrain)
print("Shape of dtrain is ", np.shape(dtrain))
print("One hot code for index 42 is ", dtrain[:,42])
print("Label for index 42 is ", ltrain[42])
```

```
Shape of dtrain is  (10, 60000)
One hot code for index 42 is  [0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
Label for index 42 is  7
```

Question 6

[7]:
```
# Creating labels from the one-hot codes

def onehot2label(d):
    lbl = d.argmax(axis=0)
    return lbl

# Checking the onehot2label() function

print("Comparing ltrain == onehot2label(dtrain)")
print(all(onehot2label(dtrain)==ltrain))
```

```
Comparing ltrain == onehot2label(dtrain)
True
```

Activation functions
Question 7

[8]:
```
# Defining activation function for the output layer

def softmax(a):
```

```
    M = a.max(axis=0)
    num = np.exp(a-M)
    den = np.exp(a-M)
    y = num/(den.sum(axis=0))
    return y
```

Question 8

We need to show that

$\frac{\partial g(a)_i}{\partial a_i} = g(a)_i(1 - g(a)_i)$

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

$= \frac{e^{a_i}}{\sum_{j=1}^{10} e^{a_j}} - \frac{(e^{a_i})^2}{(\sum_{j=1}^{10} e^{a_j})^2}$
$= g(a)_i - (g(a)_i)^2$
$= g(a)_i(1 - g(a)_i) = RHS$

Question 9

We need to show that

$\frac{\partial g(a)_i}{\partial a_j} = -g(a)_i g(a)_j \qquad for j \neq i$

$LHS = e^{a_i}.\frac{-1}{(\sum_{j=1}^{10} e^{a_j})^2}.e^{a_i}$
$= -\frac{e^{a_i}}{\sum_{j=1}^{10} e^{a_j}}.\frac{e^{a_j}}{\sum_{j=1}^{10} e^{a_j}}$
$= -g(a)_i g(a)_j = RHS$

Question 10

We need to show that

$\delta = (\frac{\partial g(a)}{\partial a})^T \times e = g(a) \otimes e - < g(a), e > g(a)$

Jacobian of $\frac{\partial g(a)}{\partial a}$ is

$$\begin{bmatrix} \frac{\partial g(a)_1}{\partial a_1} & \frac{\partial g(a)_1}{\partial a_2} & \cdots & \frac{\partial g(a)_1}{\partial a_{10}} \\ \frac{\partial g(a)_2}{\partial a_1} & \frac{\partial g(a)_2}{\partial a_2} & \cdots & \frac{\partial g(a)_2}{\partial a_{10}} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial g(a)_{10}}{\partial a_1} & \frac{\partial g(a)_{10}}{\partial a_2} & \cdots & \frac{\partial g(a)_{10}}{\partial a_{10}} \end{bmatrix}$$

Using properties from the above questions 8 and 9, Jacobian of the softmax function is

$$\begin{bmatrix} g(a)_1.(1 - g(a)_1) & -g(a)_1.g(a)_2 & \cdots & -g(a)_1.g(a)_{10} \\ -g(a)_2.g(a)_1 & g(a)_2.(1 - g(a)_2) & \cdots & -g(a)_2.g(a)_{10} \\ \vdots & \vdots & \vdots & \vdots \\ -g(a)_{10}.g(a)_1 & -g(a)_{10}.g(a)_2 & \cdots & g(a)_{10}.(1 - g(a)_{10}) \end{bmatrix}$$

We see that $\frac{\partial g(a)_i}{\partial a_j} = \frac{\partial g(a)_j}{\partial a_i} = -g(a)_i.g(a)_j$

Hence, the Jacobian matrix is symmetrical

$LHS = (\frac{\partial g(a)}{\partial a})^T \times e = \frac{\partial g(a)}{\partial a} \times e$ As matrix is symmetrical

$$= \begin{bmatrix} g(a)_1.(1-g(a)_1) & -g(a)_1.g(a)_2 & \cdots & -g(a)_1.g(a)_{10} \\ -g(a)_2.g(a)_1 & g(a)_2.(1-g(a)_2) & \cdots & -g(a)_2.g(a)_{10} \\ \vdots & \vdots & \vdots & \vdots \\ -g(a)_{10}.g(a)_1 & -g(a)_{10}.g(a)_2 & \cdots & g(a)_{10}.(1-g(a)_{10}) \end{bmatrix} \begin{pmatrix} e_1 \\ e_2 \\ \vdots \\ e_{10} \end{pmatrix}$$

$$= \begin{bmatrix} g(a)_1.(1-g(a)_1).e_1 - g(a)_1.g(a)_2.e_2 - \cdots - g(a)_1.g(a)_{10}.e_{10} \\ -g(a)_2.g(a)_1.e_1 + g(a)_2.(1-g(a)_2).e_2 - \cdots - g(a)_2.g(a)_{10}.e_{10} \\ \vdots \\ -g(a)_{10}.g(a)_1.e_1 - g(a)_{10}.g(a)_2.e_2 - \cdots + g(a)_{10}.(1-g(a)_{10}).e_{10} \end{bmatrix}$$

$$= \begin{bmatrix} g(a)_1.e_1 - (\sum_{i=1}^{10} g(a)_i.e_i).g(a)_1 \\ g(a)_2.e_2 - (\sum_{i=1}^{10} g(a)_i.e_i).g(a)_2 \\ \vdots \\ g(a)_{10}.e_{10} - (\sum_{i=1}^{10} g(a)_i.e_i).g(a)_{10} \end{bmatrix}$$

$$= g(a) \otimes e - < g(a), e > g(a) = RHS$$

[23]:

```
# Defining directional derivative of softmax()

def softmaxp(a, e):
    y = softmax(a)
    d = y*e - (y*e).sum(axis=0)*(y)
    return d
```

Question 11

[10]:

```
# Checking softmax() and softmaxp() functions

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 = (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')
```

```
4.962551548193228e-07 should be smaller than 1e-6
```

Question 12

[11]:

```
# Defining activation function for the hidden layers

def relu(a):
    a = np.maximum(a, 0)
    return a

def relup(a,e):
    a = np.maximum(a, 0)
    a[a>0] = 1
```

```
    return a*e

# Checking relu() and relup() functions

eps = 1e-6 # finite difference step
a = np.random.randn(10, 200) # random inputs
e = np.random.randn(10, 200) # random directions
diff = relup(a, e)
diff_approx = (relu(a + eps*e) - relu(a)) / eps
rel_error = np.abs(diff - diff_approx).mean() / np.abs(diff_approx).mean()
print(rel_error,'should be smaller than 1e-6')
```

```
3.761694632947271e-11 should be smaller than 1e-6
```

Backpropagation
Question 13

[12]:
```
# He  and  Xavier  initializations

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

Question 14

[13]:
```
# Evaluates the prediction of initial network

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) # We use relu for hidden layers
    a2 = W2.dot(h1) + b2
    y = softmax(a2) # We use softmax for output layer

    return y

yinit = forwardprop_shallow(xtrain, netinit)
```

Question 15

[14]:
```
# Computes the average cross-entropy loss

def eval_loss(y, d):
    err = -d*np.log(y)
    err = np.sum(err)/(err.shape[0]*err.shape[1])
    return err

print("Loss of initial prediction is", eval_loss(yinit, dtrain), '(should be␣
 ↪around .26)')
```

```
Loss of initial prediction is 0.26045800015639636 (should be around .26)
```

Question 16

[15]:
```
# Calculates the percentage of misclassified samples

def eval_perfs(y, lbl):
    y=onehot2label(y)
    t=sum(np.equal(y,lbl))
    per=((lbl.size-t)/lbl.size)*100
    return per

print("Performance of initial prediction is", eval_perfs(yinit, ltrain))
```

```
Performance of initial prediction is 89.82
```

For the initial prediction, we take the weights and bias to be random. There will be a 10% probability that our prediction is correct. So, we get the percentage of misclassified images around 90%.

Question 17

[16]:
```
# Updating the weights and bias of the network

def update_shallow(x, d, net, gamma=.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]

    gamma = gamma / x.shape[1] # normalized by the training dataset size

    # Forward Propagation
    a1 = W1.dot(x) + b1
```

```
    h1 = relu(a1)
    a2 = W2.dot(h1) + b2
    y = softmax(a2)

    # Calculating delta
    del_e = -d/y
    delta2 = softmaxp(a2, del_e)
    delta1 = relup(a1, W2.T.dot(delta2))

    # Update weights and bias
    W2 = W2 - gamma * delta2.dot(h1.T)
    W1 = W1 - gamma * delta1.dot(x.T)
    b2 = b2 - gamma * delta2.sum(axis = 1, keepdims = True)
    b1 = b1 - gamma * delta1.sum(axis = 1, keepdims = True)


    return W1, b1, W2, b2
```

Show that $(\nabla_y E)_i = -\frac{d_i}{y_i}$
$E_i = -d_i.log(y_i)$

Taking partial derivative of E_i w.r.t y_i, we get

$(\nabla_y E)_i = -d_i.\frac{1}{y_i} = -\frac{d_i}{y_i}$

Question 18

[20]:
```
# Updating the network T number of times

def backprop_shallow(x, d, net, T, gamma=.05):
    lbl = onehot2label(d)
    for t in range(T):
        net=update_shallow(x, d, net)
        y = forwardprop_shallow(x, net)
        print("Iteration ",t+1)
        print("Loss", eval_loss(y,d))
        print("Performance", eval_perfs(y,lbl))
    return net

nettrain = backprop_shallow(xtrain, dtrain, netinit, 20)
```

```
Iteration  1
Loss 0.23063985669708123
Performance 85.64166666666667
Iteration  2
Loss 0.21483824378585523
Performance 77.29166666666667
Iteration  3
Loss 0.20530700175831376
```

```
Performance 68.16499999999999
Iteration  4
Loss 0.1972938131814259
Performance 61.901666666666664
Iteration  5
Loss 0.18991323025420803
Performance 57.01833333333334
Iteration  6
Loss 0.1830356274324692
Performance 53.39
Iteration  7
Loss 0.1766077785249424
Performance 50.24999999999999
Iteration  8
Loss 0.170565355852825
Performance 47.77
Iteration  9
Loss 0.16484062616327846
Performance 44.97833333333333
Iteration  10
Loss 0.15941337740630612
Performance 43.126666666666665
Iteration  11
Loss 0.1542611072927551
Performance 40.928333333333335
Iteration  12
Loss 0.14937329769049068
Performance 39.64333333333334
Iteration  13
Loss 0.14473668937978654
Performance 37.54833333333333
Iteration  14
Loss 0.1403567759621174
Performance 36.76166666666666
Iteration  15
Loss 0.1362395173787731
Performance 34.696666666666665
Iteration  16
Loss 0.13242189930066336
Performance 34.58
Iteration  17
Loss 0.12897402865276697
Performance 32.71
Iteration  18
Loss 0.1260192729493239
Performance 33.781666666666666
Iteration  19
Loss 0.12395624812825284
```

```
Performance 32.821666666666665
Iteration  20
Loss 0.12283342968766965
Performance 36.038333333333334
```

We can see that the training errors decreased with fluctuations. At the end of 20 iterations, the training error was found out to be 36.04%

Question 19

[21]:
```python
# Loading the testing datasets

xtest, ltest = MNISTtools.load(dataset="testing", path=None)
xtest = normalize_MNIST_images(xtest)
dtest = label2onehot(ltest)
print("Size of testing sets", xtest.shape, " and ", ltest.shape)

# Testing the trained network with testing datasets

y = forwardprop_shallow(xtest, nettrain)
print("Testing case")
print("Loss", eval_loss(y, dtest))
print("Performance", eval_perfs(y, ltest))
```

```
Size of testing sets (784, 10000)  and  (10000,)
Testing case
Loss 0.1682837735643639
Performance 48.88
```

Question 20

[22]:
```python
# Defining backpropagation using mini-batch

def backprop_minibatch_shallow(x, d, net, T, B=100, gamma=.05):
    N = x.shape[1]
    NB = int((N+B-1)/B)
    lbl = onehot2label(d)
    for t in range(T):
        shuffled_indices = np.random.permutation(range(N))
        for l in range(NB):
            minibatch_indices = shuffled_indices[B*l:min(B*(l+1), N)]
            net = update_shallow(x[:, minibatch_indices], d[:,␣
↪minibatch_indices], net)
        y = forwardprop_shallow(x, net)
        print("Epoch", t+1)
        print("Loss", eval_loss(y,d))
        print("Performance", eval_perfs(y,lbl))
    return net
```

```
netminibatch = backprop_minibatch_shallow(xtrain, dtrain, netinit, 5, B=100)
```

```
Epoch 1
Loss 0.03144932108574833
Performance 9.345
Epoch 2
Loss 0.024013696017750375
Performance 6.94
Epoch 3
Loss 0.018645576183527572
Performance 5.496666666666666
Epoch 4
Loss 0.016375259974174886
Performance 4.6883333333333335
Epoch 5
Loss 0.01418760218948036
Performance 4.115
```

For the training of minibatch network, after 5 epochs the training error was found out to be 4.12%

Question 21

[25]:
```
# Testing the trained minibatch network with testing datasets

y = forwardprop_shallow(xtest, netminibatch)
print("Testing case: Minibatch network")
print("Loss", eval_loss(y, dtest))
print("Performance", eval_perfs(y, ltest))
```

```
Testing case: Minibatch network
Loss 0.042238856948625905
Performance 12.44
```

The minibatch trained network had lower testing error than that of the previously trained network. Also, the training of the minibatch network was faster and had less error.