CS 559 Hwk2

September 27, 2017

0.1 Q.2

```
In [2]: import os
        import struct
        import numpy as np
        import matplotlib.pyplot as plt
        import matplotlib as mpl
```

0.1.1 (c)

Each image is 28 × 28, so that we will have a neural network 28 × 28 = 784 nodes in the input layer, and 10 nodes in the output layer. We will ignore the biases. We wish to find 784 × 10 = 7840 weights such that the network outputs [1 0 0 ... 0]T if the input image corresponds to a 0, [0 1 0 ... 0]T if the input image corresponds to a 1, and so on.

```
In [4]: # save the original binary MNIST data files in 0-255
        def read(dataset = "training", path = "."):
            if dataset is "training":
                fname_img = os.path.join(path, 'train-images-idx3-ubyte')
                fname_lbl = os.path.join(path, 'train-labels-idx1-ubyte')
            elif dataset is "testing":
                fname_img = os.path.join(path, 't10k-images-idx3-ubyte')
                fname_lbl = os.path.join(path, 't10k-labels-idx1-ubyte')
            else:
                raise ValueError( "It needs to be between 'testing' and 'training'")

            with open(fname_lbl, 'rb') as flbl:
                magic, num = struct.unpack(">II", flbl.read(8))
                lbl = np.fromfile(flbl, dtype=np.int8)
                # print(len(lbl))
            with open(fname_img, 'rb') as fimg:
                magic, num, rows, cols = struct.unpack(">IIII", fimg.read(16))
                image = np.fromfile(fimg, dtype=np.uint8)
                image = image.reshape(len(lbl), rows, cols)

            get_image = lambda idx: (lbl[idx], image[idx])

            for i in range(len(lbl)):
```

```
            yield get_image(i)

    training_data = list(read(dataset = "training",path = r'C:\Users\Han\Desktop\Box Sync\CS
    testing_data = list(read(dataset = "testing",path = r'C:\Users\Han\Desktop\Box Sync\CS 5
    training_label = np.zeros((len(training_data),1))
    training_desiredout =np.zeros((len(training_data),10))
    training_image = np.zeros((len(training_data),28*28))
    testing_label = np.zeros((len(testing_data),1))
    testing_desiredout = np.zeros((len(testing_data),10))
    testing_image = np.zeros((len(testing_data),28*28))

    # split the training and testing data to labels and images
    for i in range(len(training_data)):
        temp = training_data[i]
        training_label[i] = temp[0]
        training_desiredout[i,temp[0]] = 1
        training_image[i,] = temp[1].reshape(1,28*28)
    #training_label = training_label.reshape((1,60000))
    for i in range(len(testing_data)):
        temp = testing_data[i]
        testing_label[i] = temp[0]
        testing_desiredout[i,temp[0]] = 1
        testing_image[i,] = temp[1].reshape(1,28*28)
    #testing_label = testing_label.reshape((1,10000))
```

0.1.2 (d)(e)(f)

Run Steps (d) and (e) for n = 50, $\eta = 1$, and some very small ϵ ($\epsilon = 0$ should also work). You should observe that step (d) terminates with 0 errors eventually. So, we have 0% error according to our training samples. Plot the epoch number vs. the number of misclassification errors (including epoch 0). Now, run Step (e) and record the percentage of misclassified test samples (over all 10000 test samples).

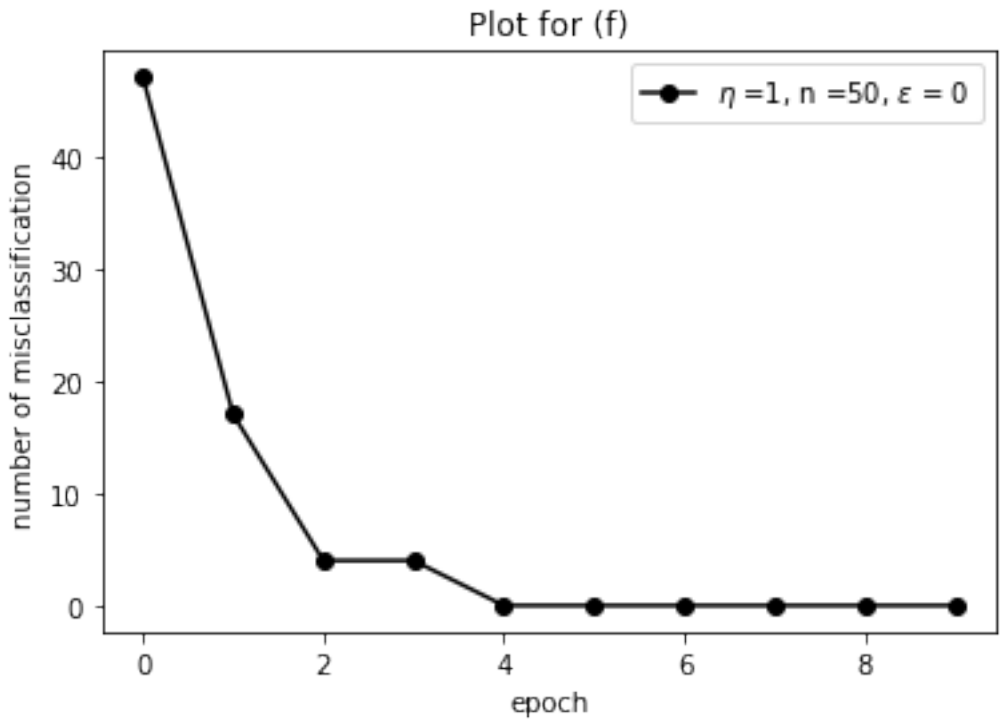
In [5]:
```
### Do (d) under (f)'s request

    ## initialization
    np.random.seed(1)
    # use first n samples from training data to train the NN
    n = 50
    # learning rate
    eta = 1
    # convergence threshold
    epsi = 0
    # epoch number
    epoch = 0
    m = 9
    # initialize errors
    errors = np.zeros((m+1,1))
```

```
# initialize real outputs given the current w
y = np.zeros((n,1))
# initialize w0
w = np.random.rand(10,28*28)
# initialize condi
conti = True

# realize 3)

while conti ==True:
    if epoch >m:
        print('Not converged yet, need more epoches.')
        print('But the results are saved.')
        break
    else:
        for i in range(n): # 3.1.1) this loop is where we count the misclassification er
            v = w.dot(training_image[i,])  #compute the induced local field
            y[i,:] = np.argmax(v)  # the output of image i by argmax() instead of using
            diff = y[i,:] - training_label[i]
            if diff != 0:
                errors[epoch,:] += 1
        epoch += 1
        for i in range(n): # 3.1.3) (this loop is where we update the weights)
            v = w.dot(training_image[i,])
            w += eta*(training_desiredout[i].reshape(10,1)-np.heaviside(v.reshape(10,1),
        conti = errors[(epoch-1),:]/n > epsi
        conti = conti.astype(bool)
## Plot the epoch number vs. the number of misclassification errors (including epoch 0).
x = np.arange(0,m+1,1)
plt.plot(x,errors,'k-o',label = '$\eta$ =1, n =50, $\epsilon$ = 0')
plt.xlabel('epoch')
plt.title('Plot for (f)')
plt.ylabel('number of misclassification')
plt.legend()
plt.show()
```



In [6]:
```
## Now, run Step (e) and record the percentage of misclassified test samples under (f).
# Given W obtained from the multicategory PTA.
W = w
#  Initialize errors = 0.
error_test = 0
# loop on all testing samples
for i in range(10000):
    # Calculate the induced local fields with the current test sample and weights:
    vprime = W.dot(testing_image[i,])
    # Find the largest component of v0 = [v0', v1', ...v9']^T^
    predic_out = np.argmax(vprime)
    #  If the prediceted output is different to the testing label, error +=1
    diff = predic_out - testing_label[i]
    if diff != 0:
        error_test += 1
error_test/10000
```

Out[6]: 0.4559

Q: What is the error rate when n=50?.</span.>

A: Error rate is 45.59%.

Q: Explain the discrepancy between the percentages of errors obtained through the training and test samples.</span.>

A: Here we see the NN was tuned to have a 0 error on those 50 training samples, however, it was clearly an overfitting for these 10,000 testing samples. Thus, the error rates on two batches of samples are different.

0.1.3 (d)(e)(g)

Run Steps (d) and (e) for n = 1000, $\eta = 1$, and some very small ϵ ($\epsilon = 0$ should also work). Again, you should observe that step (d) terminates with 0 errors eventually. Repeat the same tasks as in Step (f).

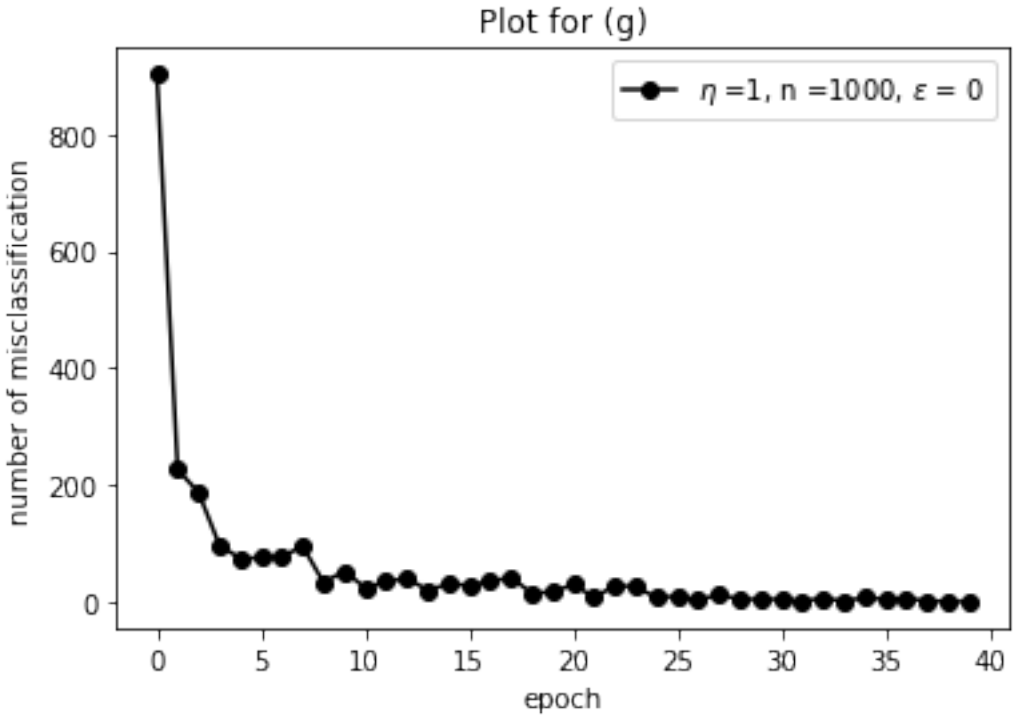
In [7]:
```
### Do (d) under (g)'s request

## initialization
np.random.seed(1)
# use first n samples from training data to train the NN
n = 1000
# learning rate
eta = 1
# convergence threshold
epsi = 0
# epoch number
epoch = 0
m = 39
# initialize errors
errors = np.zeros((m+1,1))
# initialize real outputs given the current w
y = np.zeros((n,1))
# initialize w0
w = np.random.rand(10,28*28)
# initialize condi
conti = True

# realize 3)
while conti ==True:
    if epoch >=m:
        print('Not converged yet, need more epoches.')
        print('But the results are saved.')
        break
    else:
        for i in range(n): # 3.1.1) this loop is where we count the misclassification er
            v = w.dot(training_image[i,])  #compute the induced local field
            y[i,:] = np.argmax(v)  # the output of image i by argmax() instead of using
            diff = y[i,:] - training_label[i]
            if diff != 0:
                errors[epoch,:] += 1
```

```
            epoch += 1
            for i in range(n): # 3.1.3) (this loop is where we update the weights)
                v = w.dot(training_image[i,])
                w += eta*(training_desiredout[i].reshape(10,1)-np.heaviside(v.reshape(10,1),
            conti = errors[(epoch-1),:]/n > epsi
            conti = conti.astype(bool)

    ## Plot the epoch number vs. the number of misclassification errors (including epoch 0).
    x = np.arange(0,epoch+1,1)
    plt.plot(x,errors,'k-o',label = '$\eta$ =1, n =1000, $\epsilon$ = 0')
    plt.xlabel('epoch')
    plt.ylabel('number of misclassification')
    plt.title('Plot for (g)')
    plt.legend()
    plt.show()
```



In [8]:
```
## Now, run Step (e) and record the percentage of misclassified test samples under (g).
# Given W obtained from the multicategory PTA.
W = w
#  Initialize errors = 0.
error_test = 0
# loop on all testing samples
for i in range(10000):
    # Calculate the induced local fields with the current test sample and weights:
```

```
        vprime = W.dot(testing_image[i,])
        # Find the largest component of v0 = [v0', v1', ...v9']^T^
        predic_out = np.argmax(vprime)
        #  If the prediceted output is different to the testing label, error +=1
        diff = predic_out - testing_label[i]
        if diff != 0:
            error_test += 1
    error_test/10000
```

```
Out[8]: 0.1772
```

Q: What is the error rate when n=1000?.</span.>

A: Error rate is 17.72%.

Q: Compare what you obtain here with what you have obtained in Step (f).

A: Here we see the NN was tuned to have a 0 error on those 1,000 training samples, which was still an overfitting for these 10,000 testing samples. Thus, the error rates on training and testing are different. However, with the increased training sample size in this case, it will represent more patterns of thses 10,000 testing samples. Therefore, the level of overfitting would be less and the error rate becomes significantly lower.

0.1.4 (h)

Run Step (d) for n = 60000 and $\epsilon = 0$. Make note of (i.e., plot) the errors as the number of epochs grow large, and note that the algorithm may not converge.

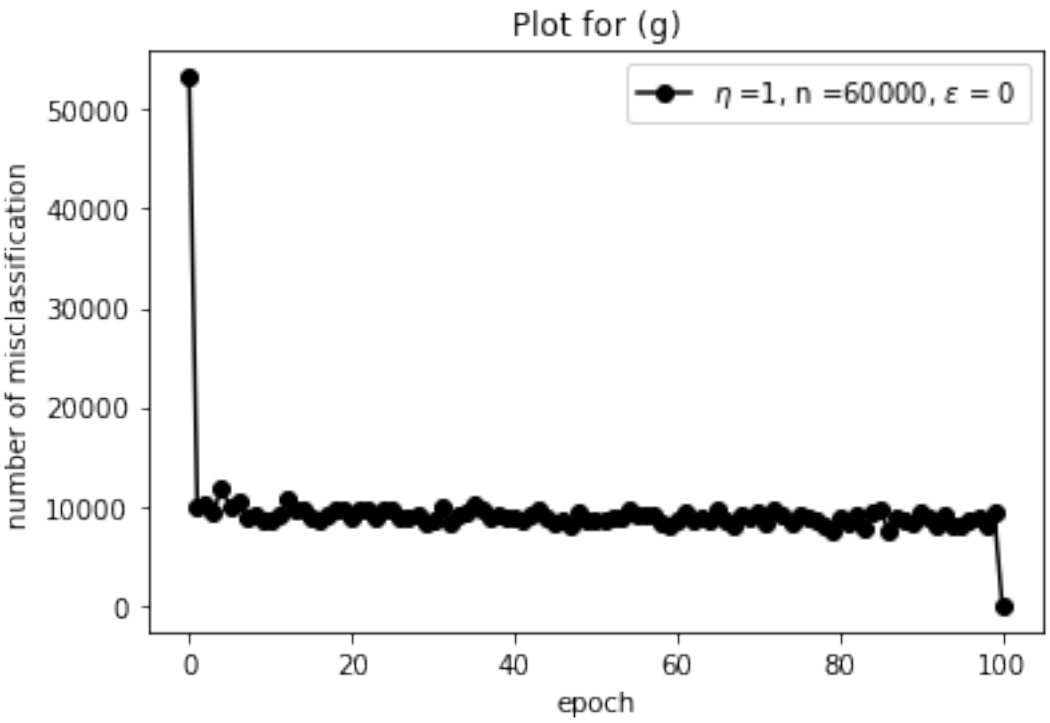
```
In [9]: ### Do (d) under (h)'s request

        ## initialization
        np.random.seed(1)
        # use first n samples from training data to train the NN
        n = 60000
        # learning rate
        eta = 1
        # convergence threshold
        epsi = 0
        # epoch number
        epoch = 0
        m = 100
        # initialize errors
        errors = np.zeros((m+1,1))
        # initialize real outputs given the current w
        y = np.zeros((n,1))
        # initialize w0
        w = np.random.rand(10,28*28)
```

```
        # initialize condi
        conti = True

        # realize 3)
        while conti ==True:
            if epoch >=m:
                print('Not converged yet, need more epoches.')
                print('But the results are saved.')
                break
            else:
                for i in range(n): # 3.1.1) this loop is where we count the misclassification er
                    v = w.dot(training_image[i,])  #compute the induced local field
                    y[i,:] = np.argmax(v)  # the output of image i by argmax() instead of using
                    diff = y[i,:] - training_label[i]
                    if diff != 0:
                        errors[epoch,:] += 1
                epoch += 1
                for i in range(n): # 3.1.3) (this loop is where we update the weights)
                    v = w.dot(training_image[i,])
                    w += eta*(training_desiredout[i].reshape(10,1)-np.heaviside(v.reshape(10,1),
                conti = errors[(epoch-1),:]/n > epsi
                conti = conti.astype(bool)
        ## Plot the epoch number vs. the number of misclassification errors (including epoch 0).
        x = np.arange(0,epoch+1,1)
        plt.plot(x,errors,'k-o',label = '$\eta$ =1, n =60000, $\epsilon$ = 0')
        plt.xlabel('epoch')
        plt.ylabel('number of misclassification')
        plt.title('Plot for (g)')
        plt.legend()
        plt.show()
```

```
Not converged yet, need more epoches.
But the results are saved.
```




```
In [10]: ## Now, run Step (e) and record the percentage of misclassified test samples under (g).
         # Given W obtained from the multicategory PTA.
         W = w
         #  Initialize errors = 0.
         error_test = 0
         # loop on all testing samples
         for i in range(10000):
             # Calculate the induced local fields with the current test sample and weights:
             vprime = W.dot(testing_image[i,])
             # Find the largest component of v0 = [v0', v1', ...v9']^T^
             predic_out = np.argmax(vprime)
             #  If the prediceted output is different to the testing label, error +=1
             diff = predic_out - testing_label[i]
             if diff != 0:
                 error_test += 1
         error_test/10000
```

```
Out[10]: 0.1608
```

Q: What is the error rate when n=60000?.</span.>

A: Error rate is 16.08%.

Q: Comment on the results.

A: At first, the selection of 0 as the ϵ decides the tuned NN are predestined to be overfitted over the training data, regardless of the fact that the entire training sets are used in the training. Nevertheless, as the reason described in question (g) already, the inclusion of all training samples mitigates the influences of the overfitting to its minimal possible extension. Besides, it should be noted that after 100 epochs, the convergence was still not achieved and it seemed like the convergence was likely to be impossible. Therefore, the error rate on the testing samples was brought down to a new low level.

0.1.5 (i)

Using your observations in the previous step, pick some appropriate value for ϵ (such that your algorithm in (d) will eventually terminate). Repeat the following two subitems three times with different initial weights and comment on the results:
Run Step (d) for n = 60000, some η of your choice and the ϵ you picked.
Run Step (e) to with the W you obtained in the previous step.

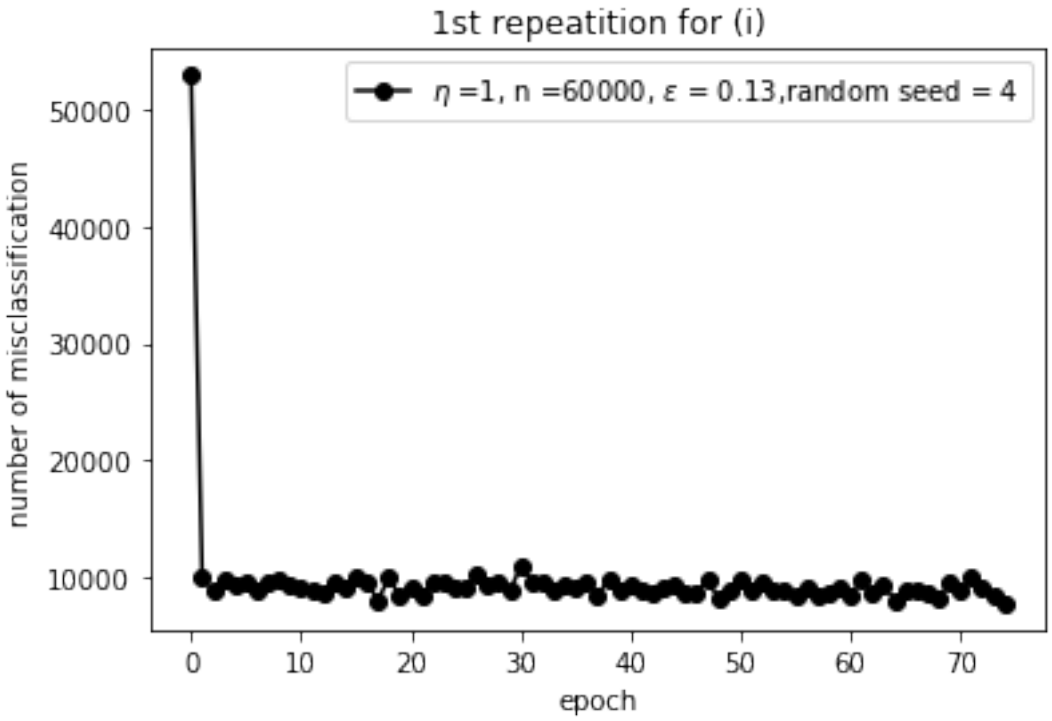
In [13]:
```
### 1st intialized w
### Do (d) under (i)'s request

## initialization
np.random.seed(4)
# use first n samples from training data to train the NN
n = 60000
# learning rate
eta = 1
# convergence threshold
epsi = 0.13
# epoch number
epoch = 0
m = 100
# initialize errors
errors = np.zeros((m,1))
# initialize real outputs given the current w
y = np.zeros((n,1))
# initialize w0
w = np.random.rand(10,28*28)
# initialize condi
conti = True

# realize 3)
while conti ==True:
    if epoch >=m:
        print('Not converged yet, need more epoches.')
        print('But the results are saved.')
        break
    else:
```

```
            for i in range(n): # 3.1.1) this loop is where we count the misclassification e
                v = w.dot(training_image[i,])  #compute the induced local field
                y[i,:] = np.argmax(v)  # the output of image i by argmax() instead of using
                diff = y[i,:] - training_label[i]
                if diff != 0:
                    errors[epoch,:] += 1
            epoch += 1
            for i in range(n): # 3.1.3) (this loop is where we update the weights)
                v = w.dot(training_image[i,])
                w += eta*(training_desiredout[i].reshape(10,1)-np.heaviside(v.reshape(10,1)
            conti = errors[(epoch-1),:]/n > epsi
            conti = conti.astype(bool)

            ## Plot the epoch number vs. the number of misclassification errors (including
x = np.arange(0,m,1)
plt.plot(x[0:epoch],errors[0:epoch],'k-o',label = '$\eta$ =1, n =60000, $\epsilon$ = 0.
plt.xlabel('epoch')
plt.ylabel('number of misclassification')
plt.title('1st repeatition for (i)')
plt.legend()
plt.show()
```




```
In [14]: ## Now, run Step (e) and record the percentage of misclassified test samples under (g).
         # Given W obtained from the multicategory PTA.
```

```
W = w
#  Initialize errors = 0.
error_test = 0
# loop on all testing samples
for i in range(10000):
    # Calculate the induced local fields with the current test sample and weights:
    vprime = W.dot(testing_image[i,])
    # Find the largest component of v0 = [v0', v1', ...v9']^T^
    predic_out = np.argmax(vprime)
    #  If the prediceted output is different to the testing label, error +=1
    diff = predic_out - testing_label[i]
    if diff != 0:
        error_test += 1
error_test/10000
```

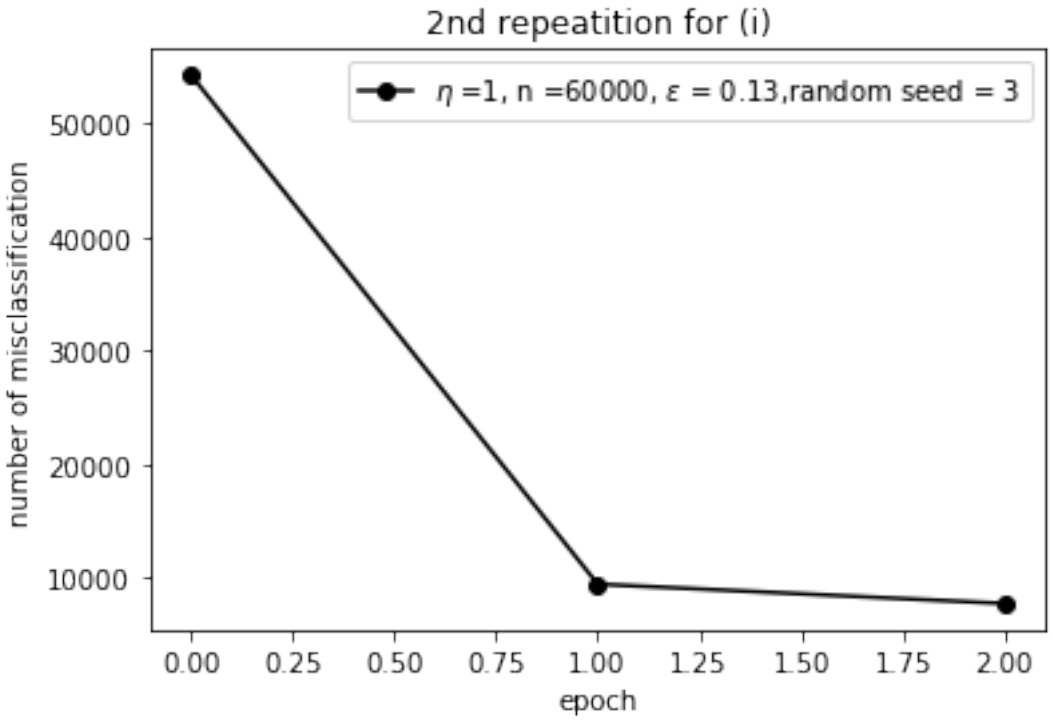
Out[14]: 0.1425

```
In [15]: ### 2nd intialized w
         ### Do (d) under (i)'s request
         ## initialization
         np.random.seed(3)
         # use first n samples from training data to train the NN
         n = 60000
         # learning rate
         eta = 1
         # convergence threshold
         epsi = 0.13
         # epoch number
         epoch = 0
         m = 100
         # initialize errors
         errors = np.zeros((m,1))
         # initialize real outputs given the current w
         y = np.zeros((n,1))
         # initialize w0
         w = np.random.rand(10,28*28)
         # initialize condi
         conti = True

         # realize 3)
         while conti ==True:
             if epoch >=m:
                 print('Not converged yet, need more epoches.')
                 print('But the results are saved.')
                 break
             else:
                 for i in range(n): # 3.1.1) this loop is where we count the misclassification e
                     v = w.dot(training_image[i,])  #compute the induced local field
```

```
            y[i,:] = np.argmax(v)  # the output of image i by argmax() instead of using
            diff = y[i,:] - training_label[i]
            if diff != 0:
                errors[epoch,:] += 1
        epoch += 1
        for i in range(n): # 3.1.3) (this loop is where we update the weights)
            v = w.dot(training_image[i,])
            w += eta*(training_desiredout[i].reshape(10,1)-np.heaviside(v.reshape(10,1)
        conti = errors[(epoch-1),:]/n > epsi
        conti = conti.astype(bool)

        ## Plot the epoch number vs. the number of misclassification errors (including
x = np.arange(0,m,1)
plt.plot(x[0:epoch],errors[0:epoch],'k-o',label = '$\eta$ =1, n =60000, $\epsilon$ = 0.
plt.xlabel('epoch')
plt.ylabel('number of misclassification')
plt.title('2nd repeatition for (i)')
plt.legend()
plt.show()
```



In [16]:
```
## Now, run Step (e) and record the percentage of misclassified test samples under (g).
# Given W obtained from the multicategory PTA.
W = w
#  Initialize errors = 0.
```

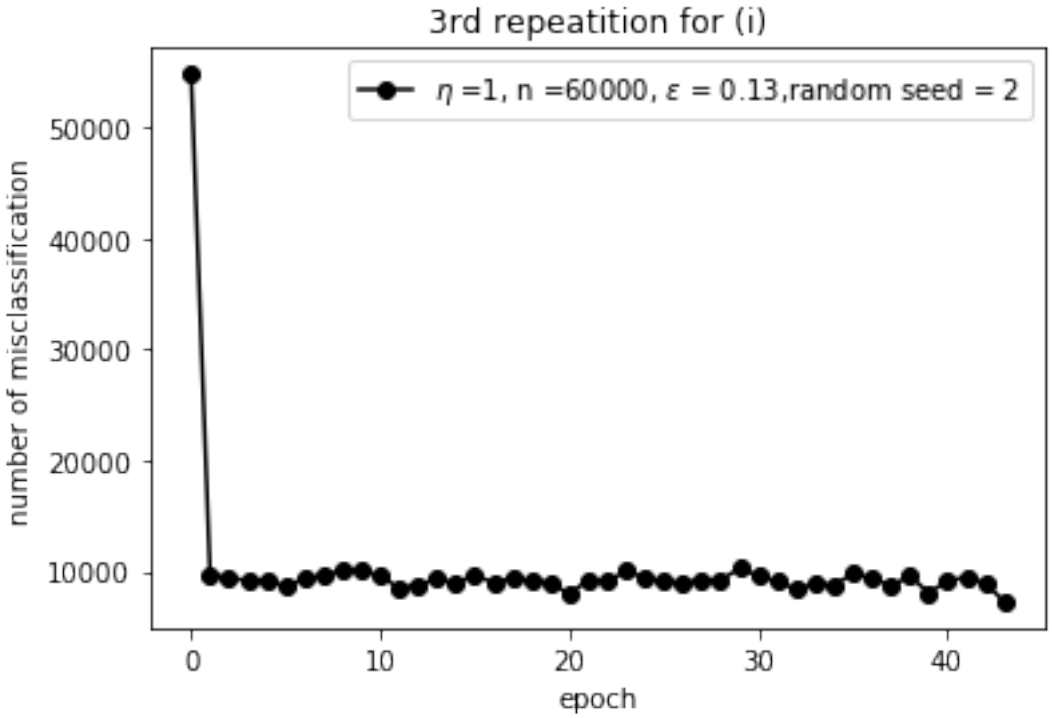
```
error_test = 0
# loop on all testing samples
for i in range(10000):
    # Calculate the induced local fields with the current test sample and weights:
    vprime = W.dot(testing_image[i,])
    # Find the largest component of v0 = [v0', v1', ...v9']^T^
    predic_out = np.argmax(vprime)
    #  If the prediceted output is different to the testing label, error +=1
    diff = predic_out - testing_label[i]
    if diff != 0:
        error_test += 1
error_test/10000
```

Out[16]: 0.1587

```
In [17]: ### 3rd intialized w
         ### Do (d) under (i)'s request
         ## initialization
         np.random.seed(2)
         # use first n samples from training data to train the NN
         n = 60000
         # learning rate
         eta = 1
         # convergence threshold
         epsi = 0.13
         # epoch number
         epoch = 0
         m = 100
         # initialize errors
         errors = np.zeros((m,1))
         # initialize real outputs given the current w
         y = np.zeros((n,1))
         # initialize w0
         w = np.random.rand(10,28*28)
         # initialize condi
         conti = True
         # realize 3)
         while conti ==True:
             if epoch >=m:
                 print('Not converged yet, need more epoches.')
                 print('But the results are saved.')
                 break
             else:
                 for i in range(n): # 3.1.1) this loop is where we count the misclassification e
                     v = w.dot(training_image[i,])  #compute the induced local field
                     y[i,:] = np.argmax(v)  # the output of image i by argmax() instead of using
                     diff = y[i,:] - training_label[i]
                     if diff != 0:
```

```
                errors[epoch,:] += 1
        epoch += 1
        for i in range(n): # 3.1.3) (this loop is where we update the weights)
            v = w.dot(training_image[i,])
            w += eta*(training_desiredout[i].reshape(10,1)-np.heaviside(v.reshape(10,1)
        conti = errors[(epoch-1),:]/n > epsi
        conti = conti.astype(bool)

        ## Plot the epoch number vs. the number of misclassification errors (including
x = np.arange(0,m,1)
plt.plot(x[0:epoch],errors[0:epoch],'k-o',label = '$\eta$ =1, n =60000, $\epsilon$ = 0.
plt.xlabel('epoch')
plt.ylabel('number of misclassification')
plt.title('3rd repeatition for (i)')
plt.legend()
plt.show()
```



In [18]:
```
## Now, run Step (e) and record the percentage of misclassified test samples under (g).
# Given W obtained from the multicategory PTA.
W = w
#  Initialize errors = 0.
error_test = 0
# loop on all testing samples
for i in range(10000):
```

```
            # Calculate the induced local fields with the current test sample and weights:
            vprime = W.dot(testing_image[i,])
            # Find the largest component of v0 = [v0', v1', ...v9']^T^
            predic_out = np.argmax(vprime)
            #  If the prediceted output is different to the testing label, error +=1
            diff = predic_out - testing_label[i]
            if diff != 0:
                error_test += 1
        error_test/10000
```

```
Out[18]: 0.1587
```

Q: Comment on the results.

A: By setting three different random seeds, we initialized the W matrix differently. Yet, with the same learning rate, only by relaxing the convergence threshold from 0 to 0.13, we discovered three record low error rates on the testing samples as 0.1425, 0.1587 and 0.1587 compared to the 0.1608 when $\epsilon = 0$. Meanwhile, it should be noted that when $\epsilon = 0$ the training won't converge whatsoever, but when $\epsilon = 0.13$ was set, the convergence happened at the epoch 75, 2 and 44, which was another plus advantage. Hence, they both collaborated our previous argument on the overfitting on the training sample would compromise the tuned NN's performance on the testing samples. Therefore, the overfitting would be suppressed by setting an appropriate convergence threshold rather than 0 and the inclusion of a larger training sample size.

1.

From the graph. we call the region, left and right.

from the 3 boundaries confining the left region, we get. $\begin{cases} NOT(y \geq 0) \ \langle d \rangle \\ NOT(-2-x \geq 0) \ \langle e \rangle \\ -x+y+1 \geq 0 \ \langle f \rangle \end{cases}$

- - - - - - right - - $\begin{cases} x-y+1 \geq 0 \ \langle a \rangle \\ x+y-1 \geq 0 \ \langle b \rangle \\ 1-x \geq 0 \ \langle c \rangle \end{cases}$

So. 1st layer. Compute $\langle a \rangle$ to $\langle f \rangle$

2nd layer combine. $\langle a \rangle \langle b \rangle \langle c \rangle$ to form. right, $\langle d \rangle \langle e \rangle \langle f \rangle$ to form left. by AND operation

3rd layer combine. left & right by OR operation.