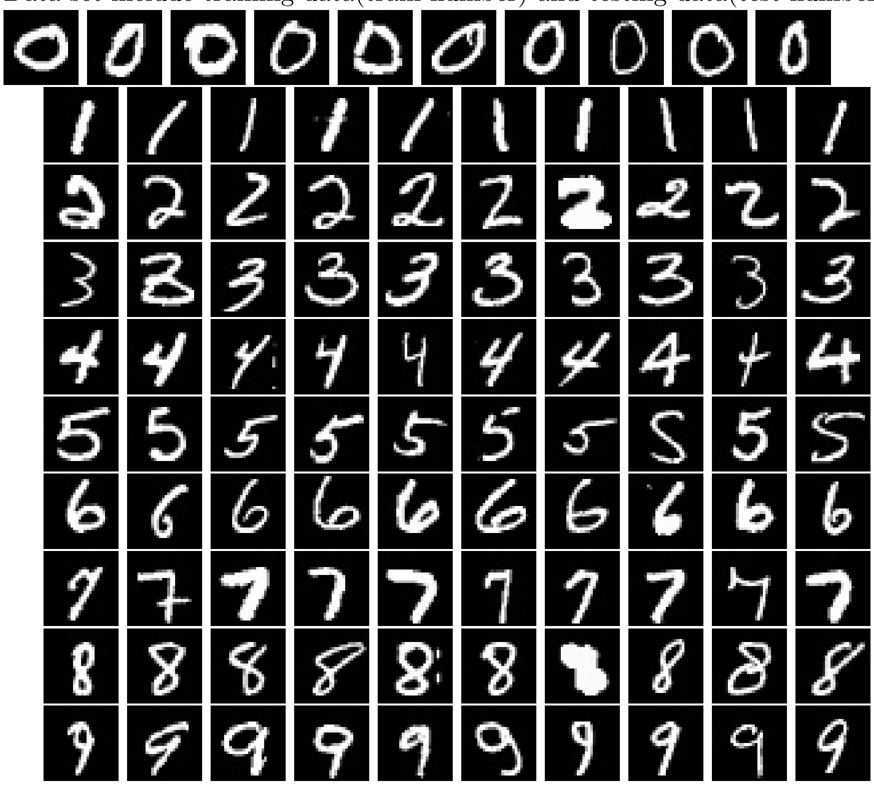
Deep Neural Networks for Handwritten Digit and Face Recognition

March 2017

1 Part1

Data set include training data(train number) and testing data(test number) from digit 0 to digit 9.



2 Part2

Implementation as follow:

```
#Return the dot product between x(images) and w(weight) and then add b(bias).
def net_w(x, w):
    b = ones((1000, 10))
    return (dot(x, w) + b).T

#script to demo:
output = []
w = 0.0 * np.ones((28*28,))
w[:] = np.random.rand()
y = [i for i in range(10)]
for i in range(10):
    x_temp = M["train"+str(i)][:100,:]/255.0
    o_i = net_w(x_temp, w)
```

```python
    output.append(o_i) # 100 img per digit # m = 900
output = np.array(output) # [10x100]
prob = softmax(output)
```

3 Part3

3(a)

Cost funtion: $-\sum_L y_L \log P_L$ # L is number of labels

$$\frac{\partial C}{\partial w_{ij}} = -\sum_L \frac{\partial C}{\partial P_L}\cdot\frac{\partial P_L}{\partial w_{ij}}$$ # chain rule.

i, j are fix number. w_{ij}: weight from j to i.

$$= -\sum_L \frac{\partial C}{\partial P_L}\left(\sum_m \frac{\partial P_L}{\partial O_m}\cdot\frac{\partial O_m}{\partial w_{ij}}\right)$$ # ~~mth~~ m^{th} component of Output vector.

① However, w_{ij} is indep of O_m, for $m \neq j$. # $m \neq j$ means these nodes are not
connected ⇒ m, j are not relevant.

$$\longrightarrow \frac{\partial P_L}{\partial w_{ij}} = \left(\frac{\partial P_L}{\partial O_j}\cdot\frac{\partial O_j}{\partial w_{ij}}\right)$$ # removed sum.

②

$$\frac{\partial O_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}}\left(\sum_n W_{nj} + b_j\right)$$ # b_j is bias from node j.
n range is (0, 784).
28×28.

$$= \sum_n \frac{\partial}{\partial w_{ij}} W_{nj}\cdot X_n$$ # b_j ~~is not~~ can consider as constants.

W_{ij} is indep of $W_{nj}\cdot X_n$, for $n \neq i$. # $n \neq i$ ⇒ there is no
connection from ~~i to j~~
node n or node
to node j

$$= \frac{\partial O_j}{\partial w_{ij}} = X_i.$$ # removed sum sign.

$$\frac{\partial C}{\partial w_{ij}} = -\sum_L \frac{\partial C}{\partial P_L}\cdot\frac{\partial P_L}{\partial O_j}\cdot\frac{\partial O_j}{w_{ij}}$$

$$= -\sum_L\left(\frac{\partial C}{\partial P_L}\right)\cdot\frac{\partial P_L}{\partial O_j}\cdot X_i$$

$$= X_i\cdot\frac{\partial C}{\partial O_j}$$

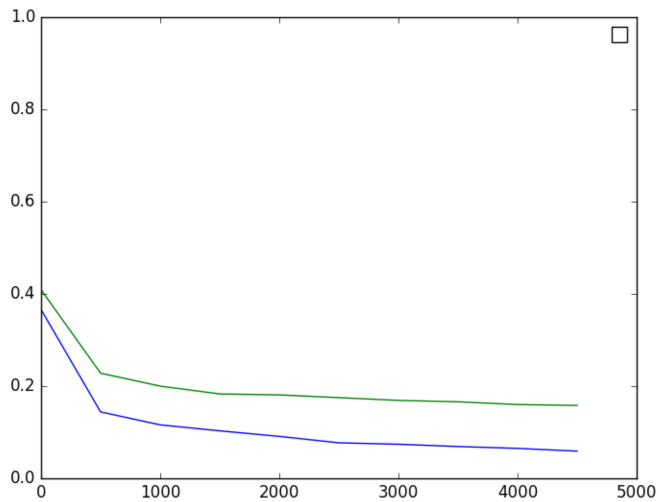
$$= X_i(P_j - y_j)$$

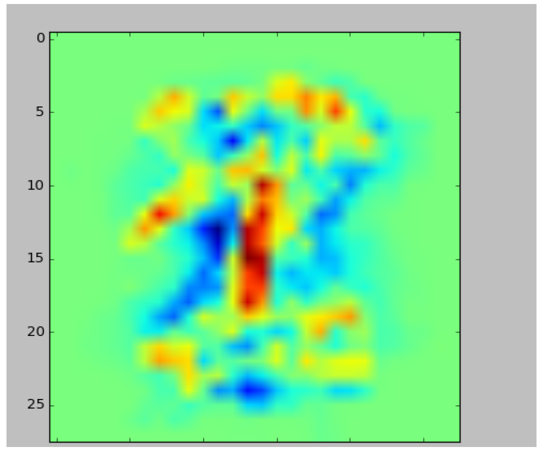
3(b)
Implementation includes cost(x,y,w), dcost(x,y,w), grad$_{d}escent(f, df, x, y, init_{t}, alpha) and finiteDiff(x, y, theta)$.

```
def cost(x, y, w):
    return (-1)*sum(log(softmax(net_w(x, w))))

def dcost(x, y, w):
    return dot((softmax(net_w(x, w)) - y.T),x).T

def finiteDiff(x, y, theta):
    EPS = 1e-10
    grad = zeros((theta.shape)) #construct a n*k matrix
    for i in range(theta.shape[0]):
        for j in range(theta.shape[1]):
            J0 = cost(x, y, theta) # J(p,y)
            theta[i][j] += EPS
            diff = cost(x, y, theta) - J0
            grad[i][j] = float(diff)/EPS
    return grad

#Script to demo:
x = [] #sizeoft * 784
y = [] #10 * sizeoft
w = [] #shape (784,10) .
w_temp = np.ones((28*28,))
w_temp[:] = np.random.rand()
for i in range(10): # every label
    x_temp = M["train"+str(i)][:25,:]/255.0
    y_temp = zeros((25,10))
    y_temp[:,i] = 1
    if i == 0:
        x = x_temp
        y = y_temp
        w = w_temp
    else:
        x = vstack((x,x_temp))
        y = vstack((y,y_temp))
        w = vstack((w,w_temp))
w = w.T
alpha = 1e-5
```

4 Part4

Implementation includes performance(x, y , w, sizeofdata) and batch(datasize).
4(a)
Graph of learning curves as follow: error rate will decrease when iteration increase. In terms of training set, error rate will close to 0 when iteration is extremely big. For testing set, error rate will keep decreasing but will not close to 0. Blue line is error rate in training set and Green line is error rate in testing set.





4(b)

4(c) Instead of using all training data together, split training data into several sets and use it several times.By using batch to graph certain number of training data:

```
def performance(x, y , w, size_of_data):
    hitnum = 0.0
    for i in range(size_of_data):
        guess = np.where(softmax(net_w(x,w)).T[i] == amax(softmax(net_w(x,w)).T[i]))[0][0]
        result = np.where(y[i] == amax(y[i]))[0][0]
        if guess == result:
            hitnum += 1.0
    return hitnum / float(size_of_data)

def batch(datasize):# 5 - 50
    new_x = [] #sizeoft * 784
    new_y = [] #10 * sizeoft
    for i in range(10):
        x_temp = M["train"+str(i)][:datasize*random.randint(0,50),:]/255.0
        y_temp = zeros((datasize,10))
        y_temp[:,i] = 1
        if i == 0:
            x = x_temp
            y = y_temp
        else:
            x = vstack((x,x_temp))
            y = vstack((y,y_temp))
    return new_x,new_y
```

5 Part5

Pick 90 images for each number, and other 10 bad images for each number. For example, if the number actually is 9 but treat it as 4. If it is 1, treat as 6. These 100 bad images will act as noise in the training data and it will effect weight that generate by gradient descent.

In the beginning, the accuracy of linear regression(86.8%) is higher than logistic regression(91.8%) on training data. However, the accuracy of logistic regression(81.4%) is higher than linear regression(71.4%).

The reasons of that is because in linear regression, each image will have same effect on theta,however, in logistic regression, each image may or may not have same effect on theta. e.g linear regression is a line.one random point would shift the line a lot. But for logistic regression, one random noise will not have big effect on total weight.

```
#script to demo
```

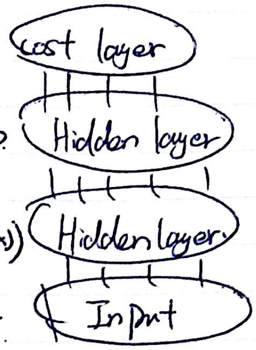
```
#Part 4 has testing data
noise_x = [] #sizeoft * 784
noise_y = [] #10 * sizeoft
noise_w = [] #shape (784,10) .
w_temp = np.ones((28*28,))
w_temp[:] = np.random.rand()
for i in range(10):
    x_temp = M["train"+str(i)][:90,:]/255.0
    y_temp = zeros((90,10))
    y_temp[:,i] = 1
    if i == 0:
        noise_x = x_temp
        noise_y = y_temp
        noise_w = w_temp
    else:
        noise_x = vstack((noise_x,x_temp))
        noise_y = vstack((noise_y,y_temp))
        noise_w = vstack((noise_w,w_temp))
    x_temp = M["train"+str(i)][90:100,:]/255.0
    y_temp = zeros((10,10))
    y_temp[:,i-5] = 1
    noise_x = vstack((noise_x,x_temp))
    noise_y = vstack((noise_y,y_temp))
noise_w = noise_w.T

w_logistireg = grad_descent(cost, dcost, noise_x, noise_y, noise_w, alpha)
w_lineareg = grad_descent(f, df, noise_x, noise_y, noise_w, alpha)
print("Using bad tranining data set to training")
print("logisti Regression")
print performance(noise_x,noise_y,w_logistireg, 1000)
print("linear Regression")
print performance(noise_x,noise_y,w_lineareg, 1000)

print("Testing data")
print("logistic Regression ")
print performance(testx,testy,w_logistireg, 1000)
print("linear Regression")
print performance(testx,testy,w_lineareg, 1000)
```

6 Part6

BackProbgation Approach:

Backprob save the result from previous step.

therefore, we don't need to compute result from last step.



Assume activation function is ReLU.

$H = \text{ReLU}(\max(0, x+\Upsilon))$ with $\Upsilon \sim N(0, \sigma(x))$

Case1. $n = 0$ $\Rightarrow$ No hidden layer.

$f(n) = \frac{d\,cost}{d\,H_I} = f(0)$ H_I is weight from Input to Cost.

define $\frac{d\,cost}{d\,H_I} = f(0)$. n is # of layer. (Hidden)

Case2. $n > 0$. $\Rightarrow$ at least 1 hidden layer.

$$\frac{d\,cost}{d\,H_n} = \text{weight}_{n-1} \cdot \text{ReLU}(\max(0, x+\Upsilon))$$

weightn-1 can observe from network.

$\text{ReLU}(\max(0, x+\Upsilon))$ with $\Upsilon \sim N(0, \sigma(x))$

A: $\text{ReLU}(\max(0, x+\Upsilon))$ takes $O(k)$ to compute. for layer H^{n-1}

Since there is k units in each Hidden layer. and it's fully connected

$$\frac{d\,cost}{d\,H_n} = O(k^2).$$

Since n can be any interger, so that ~~the~~ complexity need to consider each layer unit hit the Input layer. Assume we have n hidden layers.

$$n \cdot O(k^2) \quad \# \, n \text{ - each layer's complexity}$$
$$= O(nk^2)$$

Non-BackProbgation Approach:

There is no step such saving previous step result. Every time need to re-clculate previous layer result.

Case1: $n = 0$. will be same $\frac{d\,cost}{d\,H_I} = f(0)$

Case2: $n > 1$ $\Rightarrow \sum_{1}^{k} \frac{d\,cost}{d\,H_{n-1}} \cdot \frac{d\,H_{n-1}}{d\,H_i}$ or $\sum_{1}^{k} \left(\frac{d\,cost}{d\,H_{n-2}} \cdot \frac{d\,H_{n-2}}{d\,H_{n-1}} \cdot \frac{d\,H_{n-1}}{d\,H_i} \right)$

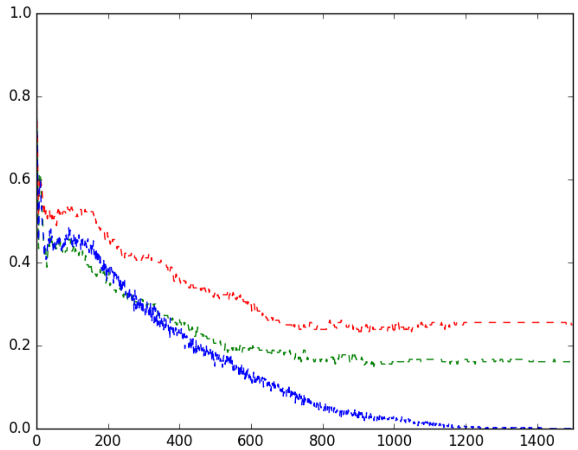
each items take $O(k)^2$. # from step A.

Since need to reconsider previous step, Insted of $O(nk^2)$, the complexity is $O(k^2)^n = O(k^{2n})$

$$O(k^{2n}) > O(nk^2) \Rightarrow \text{Backprob is faster}$$

7 Part7

The learning curve for the test, training, and validation sets has been shown below. The x-axis is the number of iterations and y-axis is error rate. The learning curve is shown below. Testing set has 83.8889% accuracy, validation set has 75% accuracy, training set has 100% accuracy which means penalty(error) is 0.



The final performance classification on the test set is about 78% of accuracy. The text description of our system is as followed. In particular, the input is preprocess by the following steps. Firstly, all the images has been re-downloaded, cropped and resized used the same function from A01 (included) except the following lines. Particularly, the follwoing codes are used to remove non-faces from our imgae dataset, utilized the SHA-256 hashes to remove bad images.

```
import hashlib

m = hashlib.sha256()
m.update(open("uncropped/"+filename, 'rb').read())
if m.hexdigest() != line.split()[6]:
    os.remove("uncropped/"+filename)
    print('not match sha256, has removed')
else:
    print(filename)
    i += 1
```

After preprocessing all the images for each actors to size of 32x32, head-centered,they are ready to be read into flattened numpy array, with size [1024, 1]. There are two main functions used to read every images for actors and categorized them into a dictionary with keys as followed: $['test_drescher','train_hader','train_carell',$ $'train_chenoweth','test_carell','valid_chenoweth','valid_drescher','valid_carell','valid_ferrera','test_ferrera',$ $'test_hader','test_chenoweth','valid_hader','train_baldwin','test_baldwin','train_ferrera','train_drescher',$ $'valid_baldwin']$. As names suggested, the images are grouped into the training, test, valid sets of each actor by the function formM. The input of formM comes from the output of function readSet which reads all the images as 1D numpy array.

The specifications of the the neural networks are as followed. The activation functions of input layer is tanh, which takes the linear combination of x, W0 and the bias units b0. The one hidden unit use the softmax as the activation functions generate outputs as probability. The code snippet is as followed:

```
W0 = tf.Variable(tf.random_normal([1024, nhid], stddev=0.1))
b0 = tf.Variable(tf.random_normal([nhid], stddev=0.01))
W1 = tf.Variable(tf.random_normal([nhid, 6], stddev=0.01))
b1 = tf.Variable(tf.random_normal([6], stddev=0.01))
layer1 = tf.nn.tanh(tf.matmul(x, W0)+b0)
layer2 = tf.matmul(layer1, W1)+b1
```

```
y = tf.nn.softmax(layer2)
y_ = tf.placeholder(tf.float32, [None, 6])
```

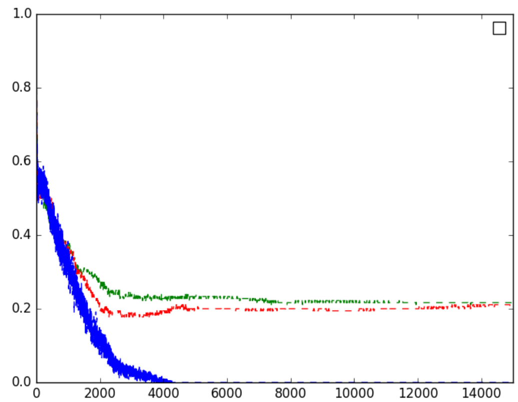
The cost function is shown

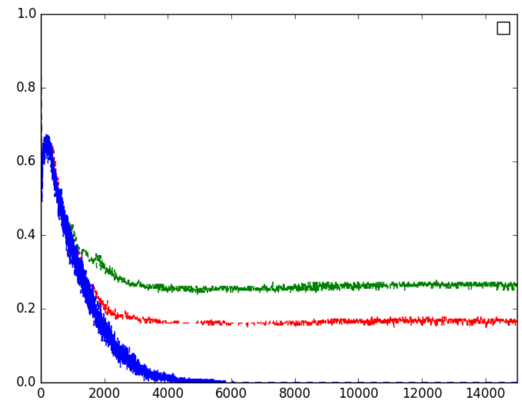
```
lam = 0.0
decay_penalty =lam*tf.reduce_sum(tf.square(W0))+lam*tf.reduce_sum(tf.square(W1))
reg_NLL = -tf.reduce_sum(y_*tf.log(y))+decay_penalty
train_step = tf.train.AdamOptimizer(0.0005).minimize(reg_NLL)
```

8 Part8

A scenario where using regularization is necessary in the context of face classification, would be when we experience overfitting. Specifically, a small sized training set, a limited number of features, and over-repeated iterations can all be possible reasons causing overfitting. From the learning curve, overfitting can be observed from the learning curve as well, specifically, when the error rate of the training set stays on zero and that of the test set starts to go up. There should be a clearly global minimum has been presented before, as seen in figure.

we have 40 pictures for each actor.15000 iterations.



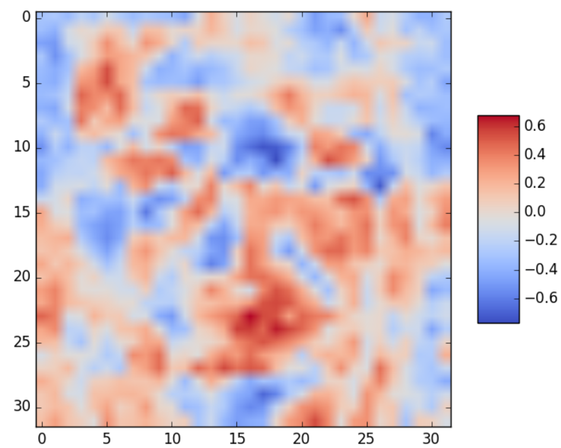


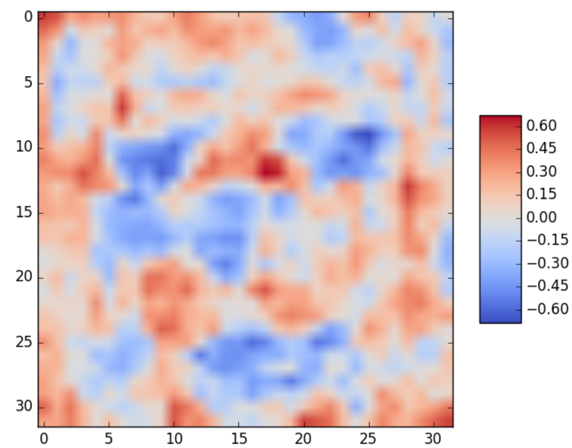
left:before apply regularization right:after regularization

The best regularization parameter is selected as 0.003. This leads to an enhancement of 6 percent roughly, from original error rate of 2.2 percent to 1.6 percent after the weight penalty. The degree of improvement is substantial, given that the size of our test size, N, is 180. The expected should be $(\frac{75}{\sqrt{N}})\% = 5.59\%$.

9 Part9

The two of the actors selected are .The visualization the weights of the hidden units that are useful for classifying input photos as those particular actors has been shown below.





The first image is female actor, Fran Drescher and the second one is male actor, Bill Hader. The selected hidden units are chosen because their inputs, weights from previous layer, have the biggest value across all the weights. This means that the weights are close to their connected inputs the most. In another words these weights are supposed to look like a face the most. The code snippet has included.

```
#Code for displaying a feature from the weight matrix mW
fig = figure(1)
ax = fig.gca()

#get the index at which the output is the largest
ind_a0 = argmax(sess.run(W1)[:,0])#Fran Drescher
ind_a4 = argmax(sess.run(W1)[:,4])#Bill Hader

#heatmap = ax.imshow(sess.run(W0)[:,ind_a0].reshape((32,32)), cmap = cm.coolwarm)
heatmap = ax.imshow(sess.run(W0)[:,ind_a4].reshape((32,32)), cmap = cm.coolwarm)
fig.colorbar(heatmap, shrink = 0.5, aspect=5)
show()
```

As shown, we first got the index of the biggest value in the W1 ([650x6]) for each actor ([650x1]). That is, the index would be the selected hidden unit that we intend to display, as explained by the reasons above.

10 Part10

Extract the values of the activations of AlexNet on the face images has been achieved by the following. By modifying the original alexnet codes provided, we changed the input to a dictionary of numpy array images, with the dimension of [227,227,3]. We substitue into the images using the codes as followed.

```
def partXread(sset):
    '''return dim(x) = m * n
    '''
    x = ones(0)
    for a in sset:
        # read each img in trainingSet into 2d array
        a = './part10/'+a
        im1 = (imread(a)[:,:,:3]).astype(float32)
        im1 = im1 - mean(im1)
        im1[:, :, 0], im1[:, :, 2] = im1[:, :, 2], im1[:, :, 0]
        # append x1 up to x1024
        if (len(x) == 0):
            x = temp
```

```
        else:
            # vstack them
            x = vstack((x, temp))
    return x
```

After that, the output from conv4 has the dimension of [, 13, 13, 384]. We flattened these activations into [, 13*13*384] and feed into the same network as accomplished in part7. In other words, we used the activations from alexNet's conv4 layers as our features to our fully connected neural networks. The specifications are as followed. The specifications of the the neural networks are as followed. The activation functions of input layer is tanh, which takes the linear combination of x, W0 and the bias units b0. The one hidden unit use the softmax as the activation functions generate outputs as probability.