CSC411: Assignment 2

Due on Sunday, February 19, 2017, 11pm

Minh Nguyen (1000468059)

September 13, 2017

Instruction for reproducing the results:
Step 1 Please include all of the following files in a directory.

- *digits.py*: codes for part 1, 2, 3, 4, 5.
- *get_data_and_crop.py*: download images for part 7, 8, 9.
- *get_data_and_crop_part10.py*: download images for part 10.
- *auto_pick.py*: automatically pick non-overllaping training, test and validation sets.
- *mnist_all.mat*: sample hand writing.
- *faces.py*: codes for part 7, 8 and 9.
- *facescrub_actors.txt*: image url for actors.
- *facescrub_actresses.txt*: image url for actresses
- *deepfaces.py*: codes for part 10.
- *bvlc_alexnet.npy*: sample weights for Alexnet.

Step 2: Downloading the images

- run *get_data_and_crop.py* to download images of the actors specified in *act*. The script stores the cropped and uncropped images in the folders *cropped/* and *uncropped/*, respectively.
- run *get_data_and_crop_part10.py* to download images of the actors specified in *act*. The cropped and uncropped images are stored in the folders *cropped_p10/* and *uncropped_p10/*, respectively. Different from *get_data_and_crop.py*, the cropped images of *get_data_and_crop_part10.py* are not in gray scale, and have size 227×227 instead of 32×32.
- For either file, after downloading all of the images for male actors in *facescrub_actors.txt*, please make small modifcation to download the images for female actors in *facescrub_actresses.txt*.

Step 3: Running the program

- Each part in the assignment can be tested independently. You can reproduce the results by running the codes in *digits.py* for part 1 2 3 4 5, *faces.py* for parts 7 8 9 and *deepfaces.py* for part 10.
- There are plenty of comments in the files, which would help you to test and understand the codes.

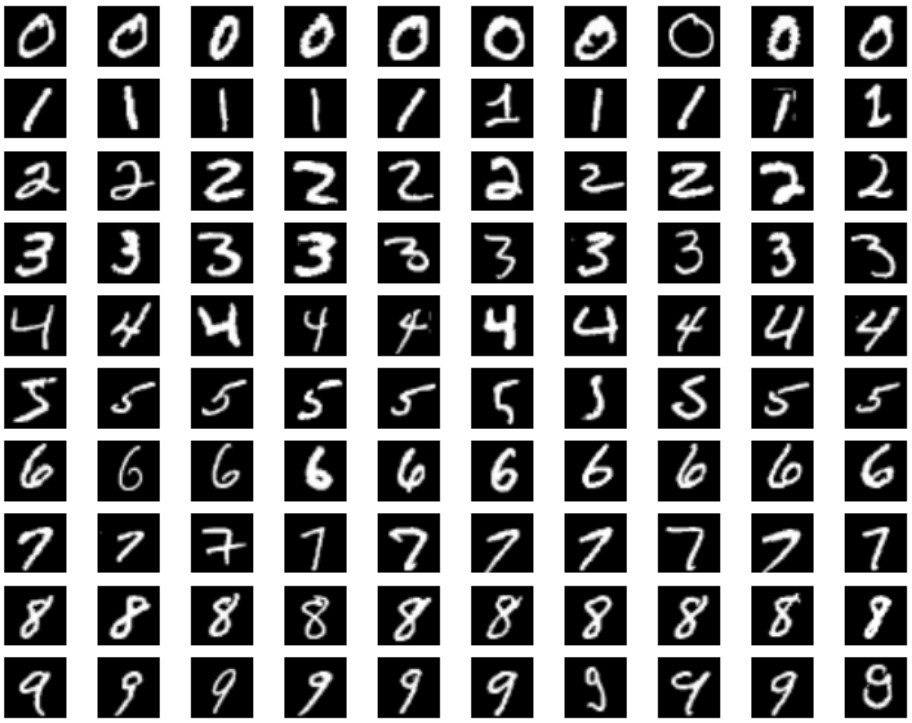
Note on LaTeX report: Along with *deepnn.tex* and *deepnn.pdf*, I submitted a zip file called *full_report_folder.zip*. It stores all of the images which can be used by *deepnn.tex* to regenerate *deepnn.pdf*.

Part 1

Dataset description:
There are in total 70000 images for the numbers from 0 to 9 in the dataset. Out of 70000 images, there are 60000 images for training and 10000 images for testing. The table below shows the number for images for each number, you may reproduce the results by running the function *part1()* in *digits.py*.

Number	No. of images in training set	No. of images in testing set
0	5923	980
1	6742	1135
2	5958	1032
3	6131	1010
4	5842	982
5	5421	892
6	5918	958
7	6265	1028
8	5851	974
9	5949	1009

Figure 1: 100 randomly picked 28×28 images in gray scale, 10 images for each number. The first eight columns are images from training sets, the last two columns are images from the validation sets. The handwritten numbers in the images are all recognizable.



Part 2

```
def softmax(y):
    '''Return the output of the softmax function for the matrix of output y. y
    is an NxM matrix where N is the number of pixels, and M is the number of samples'''
    return exp(y)/tile(sum(exp(y),0), (len(y),1))

def output(W, x):
    # Add 1 on top of x, this is for the bias b.
    x = vstack((ones((1, x.shape[1])), x))
    o = dot(W.T, x)
    p = softmax(o)
    return o, p
```

Note: I added the bias b directly into the matrix calculation. The professor stated that it is acceptable @332 Piazza.

Part 3

Part A:
We have the cost function as follows:

$$C = -\sum_i (y_i log(p_i)) \tag{1}$$

In which:

$$p_i = \frac{e^{o_i}}{\sum_k e^{o_k}} \tag{2}$$

Let's compute $\frac{dp_i}{do_j}$ using the quotation rule.

Case 1: When $i \neq j$

$$\begin{aligned} \frac{dp_i}{do_j} &= -\frac{e^{o_i} \times e^{o_j}}{(\sum_k e^{o_k})^2} \\ &= -p_i p_j \end{aligned} \tag{3}$$

Case 2: When $i = j$

$$\begin{aligned} \frac{dp_i}{do_j} &= \frac{e^{o_i}(\sum_k e^{o_k}) - e^{o_i} e^{o_i}}{(\sum_k e^{o_k})^2} \\ &= \frac{e^{o_i}((\sum_k e^{o_k}) - e^{o_i}))}{(\sum_k e^{o_k})(\sum_k e^{o_k})} \\ &= \frac{e^{o_i}}{\sum_k e^{o_k}} \times (1 - \frac{e^{o_i}}{\sum_k e^{o_k}}) \\ &= p_i - (1 - p_i) \\ &= p_j - (1 - p_j) \end{aligned} \tag{4}$$

We want $\frac{dC}{do_j} = \frac{dC}{dp_i} \times \frac{dp_i}{do_j}$.

$$\begin{aligned}
\frac{dC}{do_j} &= -\sum_i (y_i \frac{d}{do_j} log(p_i)) \\
&= -\sum_i (y_i \frac{d}{p_i} log(p_i) \times \frac{dp_i}{do_j}) \\
&= -\sum_i (\frac{y_i}{p_i} \times \frac{dp_i}{do_j}) \\
&= -(\frac{y_j}{p_j} p_j(1-p_j) + \sum_{i \neq j} (\frac{y_i}{p_i} \times \frac{dp_i}{do_j})) \\
&= -(y_j(1-p_j) + \sum_{i \neq j} (\frac{y_i}{p_i} \times \frac{dp_i}{do_j})) \\
&= -(y_j - y_j p_j + \sum_{i \neq j} (\frac{y_i}{p_i} \times -(p_i p_j))) \\
&= -(y_j - y_j p_j - \sum_{i \neq j} (y_i p_j)) \\
&= -(y_j - \sum_i (y_i p_j)) \\
&= -(y_j - p_j \sum_i (y_i)) \\
&= -(y_j - p_j \times 1) \qquad \textit{Only one 1 in } y. \\
&= p_j - y_j
\end{aligned} \tag{5}$$

We have $do_j = \sum_i w_{i,j} x_i + b_j$, then $\frac{do_j}{dw_{i,j}} = x_i$. Again, we apply the chain rule to get the final deriative:

$$\begin{aligned}
\frac{dC}{dW_{i,j}} &= \frac{dC}{do_j} \times \frac{do_j}{dw_{i,j}} \\
&= x_i(p_j - o_j)
\end{aligned} \tag{6}$$

That is for a single sample. Let m be the size of the training set, then, the gradient with respect to $W_{i,j}$ is:

$$\sum_k^m (x_i^{(k)}(p_j^{(k)} - o_j^{(k)})) \tag{7}$$

You can run the code in *digits.py*, to verify this result agaist the vectorized gradient function.

Part B

```
def compute_p_part3(W,x):
    o = dot(W.T, x)
    p = softmax(o)
    return o, p

# Vectorized gradient function.
def vectorized_gradient_part3(x, y, W):
    x = vstack((ones((1, x.shape[1])), x))
    return dot(x, (compute_p_part3(W, x)[1] - y).T)
```

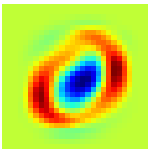
Note: This is almost identical to the vectorized gradient in assignment 1.

Part 4

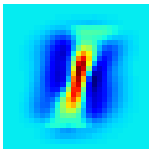
Step size $\alpha = 10^{-11}$
Initial weight $w_{initial} = 0$

Like in linear regression, I need a step size α that is neither too big or too small. α should not be too small, because it would take too many iterations for gradient descent to reach the local minimum. Else if α is too big, the algorithm could miss the local optimal point, and diverge to infinity, causing numerical overflow. $\alpha = 10^{-11}$ was the result of a trial and error process. This alpha level allowed me to construct a smooth learning curve and produced nice visualizations of the *weights*, which I could not get with the lower values. I also need an intital $weight_{initial}$, which is sufficiantly small to run gradient descent. To keep it simple, I set the $weight_{initial}$ to be a vector with only zeros.

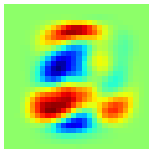
Visualization of the *weights*



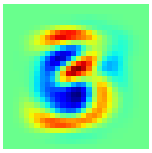
(a) 0



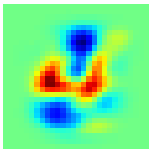
(b) 1



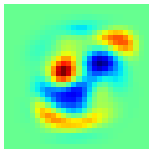
(c) 2



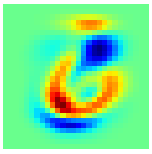
(d) 3



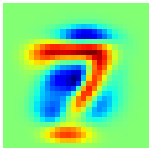
(e) 4



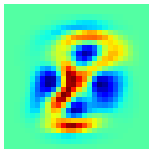
(f) 5



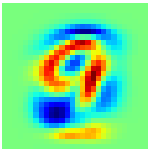
(g) 6



(h) 7



(i) 8



(j) 9

Figure 2

Learning curve:

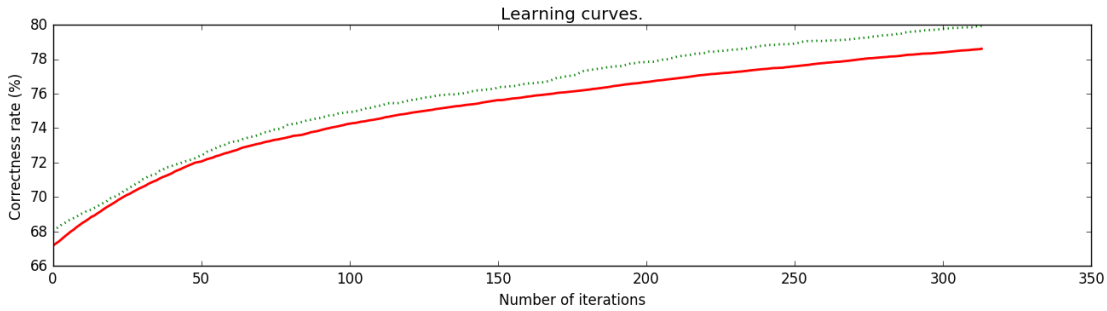


Figure 3: Learning curve: red line represents for the performance on the taining set. Green dotted line represents the performance on the testing set.

Part 5

Generate a training sample of size 200, that has normal distribution $N(\bar{x} = 0, \sigma^2 = 0.1)$

```
def part5():
    m = 200
    n = 2 # number of feature + 1
    k = 2 # number of possible labels

    # Training data
    np.random.seed(1)
    mu, sigma = 0, 0.1 # mean and standard deviation.
    s = np.random.normal(mu, sigma, m)
    s = s.reshape((1,m))

    print("------- Training data with no outlier -------")
    print(s)
```

Because of the small variance, the data points are all around 0. Below is the set of 200 data points that gets printed. Note that the first element in this array will be replaced by an outlier.

```
[[ 0.16243454 -0.06117564 -0.05281718 -0.10729686  0.08654076 -0.23015387
   0.17448118 -0.07612069  0.03190391 -0.02493704  0.14621079 -0.20601407
  -0.03224172 -0.03840544  0.11337694 -0.10998913 -0.01724282 -0.08778584
   0.00422137  0.05828152 -0.11006192  0.11447237  0.09015907  0.05024943
   0.09008559 -0.06837279 -0.01228902 -0.09357694 -0.02678881  0.05303555
  -0.06916608 -0.03967535 -0.06871727 -0.08452056 -0.06712461 -0.00126646
  -0.11173103  0.02344157  0.16598022  0.07420442 -0.01918356 -0.0887629
  -0.07471583  0.16924546  0.00508078 -0.06369956  0.01909155  0.21002551
   0.0120159   0.06172031  0.03001703 -0.03522498 -0.11425182 -0.03493427
  -0.02088942  0.05866232  0.08389834  0.09311021  0.02855873  0.08851412
  -0.07543979  0.12528682  0.05129298 -0.02980928  0.04885181 -0.00755717
   0.11316294  0.15198168  0.21855754 -0.13964963 -0.14441138 -0.05044659
   0.01600371  0.08761689  0.03156349 -0.20222012 -0.0306204   0.08279746
   0.02300947  0.07620112 -0.02223281 -0.02007581  0.01865614  0.04100516
   0.01982997  0.01190086 -0.06706623  0.03775638  0.01218213  0.11294839
```

```
 0.11989179  0.01851564 -0.0375285  -0.06387304  0.04234944  0.00773401
-0.03438537  0.00435969 -0.06200008  0.0698032  -0.04471286  0.12245077
 0.04034916  0.05935785 -0.10949118  0.01693824  0.07405565 -0.09537006
-0.02662185  0.00326145 -0.13731173  0.03151594  0.08461606 -0.08595159
 0.0350546  -0.13122834 -0.00386955 -0.16157724  0.11214177  0.04089005
-0.0024617  -0.07751616  0.12737559  0.19671017 -0.18579819  0.1236164
 0.16276508  0.03380117 -0.1199268   0.08633453 -0.01809203 -0.06039206
-0.12300581  0.05505375  0.07928069 -0.06235307  0.05205763 -0.11443414
 0.0801861   0.00465673 -0.01865698 -0.01017459  0.08688862  0.07504116
 0.05294653  0.01377012  0.00778211  0.06183803  0.02324946  0.06825514
-0.03101168 -0.24348378  0.10388246  0.21869796  0.04413644 -0.01001552
-0.01364447 -0.01190542  0.00174094 -0.11220187 -0.05170945 -0.09970268
 0.02487992 -0.02966412  0.04952113 -0.01747032  0.09863352  0.02135339
 0.21906997 -0.18963609 -0.06469167  0.09014869  0.25283257 -0.02486348
 0.0043669  -0.02263142  0.13314571 -0.02873079  0.06800698 -0.03198016
-0.12725588  0.03135477  0.05031848  0.12932259 -0.0110447  -0.06173621
 0.05627611  0.02407371  0.02806651 -0.00731127  0.11603386  0.03694927
 0.19046587  0.11110567  0.06590498 -0.16274383  0.06023193  0.04202822
 0.08109517  0.10444421]]
```

I do the classification on these data, all data points less than 0 are labelled 0 and other data points greater than 0 are labelled 1. The fun thing is I will change the first element in the array to be a very large number, that will definitely make big impact on the cost function as well as the gradient produced by the gradient descent algorithm.

```
# classification
y = zeros((k, m))
for i in range(m):
    if s[0][i] > 0:
        y[1][i]= 1;
    elif s[0][i] <= 0:
        y[0][i] = 1;

# Let put an outlier into s.
s[0][0] = 50000;
```

The very first element is now set to 50000, a value that is far away from the sample mean 0. I will run both linear and logistic regression on the data, and print the performance.

```
init_theta = np.zeros((n, k)) # each sample has only 1 feature + 1 for the bias.
alpha = 0.000000000001

# Linear gradient descent.
theta = grad_descent_linear_part5(s, y, init_theta, alpha)
# Logistic graident descent.
weight = grad_descent_logistic_part5(s, y, init_theta, alpha)

# Linear regression result on training data.
linear_result = (dot(theta[1:].T, s) + theta[0].reshape(k,1))

# Logistic regression on training data
logistic_result = (dot(weight[1:].T, s) + weight[0].reshape(k,1))

correct_count_linear = 0;
corect_count_logistic = 0;
```

```
    y = y.T
    linear_result = linear_result.T
    logistic_result = logistic_result.T

    for i in range(m):
        if (argmax(y[i]) == argmax(linear_result[i])):
            correct_count_linear = correct_count_linear + 1;
        if (argmax(y[i]) == argmax(logistic_result[i])):
            corect_count_logistic = corect_count_logistic + 1;

    print("------- Performance report on training data -------")
    print("Linear regression on training set with the outlier.",
     float(correct_count_linear)/float(len(y)) * 100)
    print("Logistic to regression on training set with the outlier.",
     float(corect_count_logistic)/float(len(y)) * 100)
```

Below is the performance on the training data reported by the program.

```
------- Performance report on training data -------
('Linear regression on training set with the outlier.', 63.5)
('Logistic regression on training set with the outlier.', 96.5)
```

The correctness rate on the training set using logistic regression is 96.5%, whereas the corretness rate for linear regression is only 63.5%. The difference is due to the cost function $C = \sum_i (\sum_j (\theta^T x^{(i)} - y^{(i)})_j^2)$ of linear regression is heavily impacted by the outlier value 50000. If you make the outlier value larger, the performance of linear regression will get even worse.

I create a test set with the same sample size, same sample distribution, but with no outlier.

```
    np.random.seed(3)

    test_data = np.random.normal(mu, sigma, m)
    test_data = test_data.reshape((1, m))

    print("------- Test data with no outlier. -------")
    print(test_data)

    # Classification of the test data.
    y_test = zeros((k, m))
    for i in range(m):
        if test_data[0][i] > 0:
            y_test[1][i]= 1;
        elif test_data[0][i] <= 0:
            y_test[0][i] = 1;

    linear_result_test = (dot(theta[1:].T, test_data) + theta[0].reshape(k,1))
    logistic_result_test = (dot(weight[1:].T, test_data) + weight[0].reshape(k,1))

    y_test = y_test.T
    linear_result_test = linear_result_test.T
    logistic_result_test = logistic_result_test.T

    correct_count_linear = 0;
    corect_count_logistic = 0;
    for i in range(m):
```

```
        if (argmax(y_test[i]) == argmax(linear_result_test[i])):
            correct_count_linear = correct_count_linear + 1;
        if (argmax(y_test[i]) == argmax(logistic_result_test[i])):
            corect_count_logistic = corect_count_logistic + 1;

    print("------- Performance report on training data -------")
    print("Linear regression on test set",
    float(correct_count_linear)/float(len(y)) * 100)
    print("Logistic to regression on test set",
    float(corect_count_logistic)/float(len(y)) * 100)
```

Below is the performance report on the testing data. As expected, the performance using logistic regression is still better. In conclusion, if you have a dataset with outliers that are much larger than other values, you better use logistic regression, as it is less impacted by the magnitude of the data.

```
------- Performance report on training data -------
('Linear regression on test set', 56.49999999999999)
('Logistic to regression on test set', 96.0)
```

Part 6

Assuming that we have a fully connected $n \times n = 4 \times 4$ neural network with the following layout:
Layer 4: $l_{4,1}$ $l_{4,2}$ $l_{4,3}$ $l_{4,4}$
Layer 3: $l_{3,1}$ $l_{3,2}$ $l_{3,3}$ $l_{3,4}$
Layer 2: $l_{2,1}$ $l_{2,2}$ $l_{2,3}$ $l_{2,4}$
Layer 1: $l_{1,1}$ $l_{1,2}$ $l_{1,3}$ $l_{1,4}$
Each neuron in the network is a linear combination of all neurons in the layer right below it.
Let compute the deriative of the cost with respect every weight that connect layer 1 to layer 2. That is computing $\frac{dC}{dW_{1,i,j}}$ for every possible i and j. $W_{1,i,j}$ is a component of the weight matrix that connects the neuron i in layer 1 to the neuron j in layer 2.

1. Weights connect $l_{1,1}$ in layer 1 to all nodes in layer2.

$$\begin{aligned}\frac{dC}{dW_{1,1,1}} &= \frac{dC}{dl_{2,1}} \times \frac{dl_{2,1}}{dW_{1,1,1}} \\ \frac{dC}{dl_{2,1}} &= \sum_{k}^{n} \frac{dC}{dl_{3,k}} \times \frac{dl_{3,k}}{dl_{2,1}} \\ \frac{dC}{dl_{3,k}} &= \sum_{i}^{n} \frac{dC}{dl_{4,i}} \times \frac{dl_{4,i}}{dl_{3,k}}\end{aligned} \tag{1}$$

$$\begin{aligned}\frac{dC}{dW_{1,1,2}} &= \frac{dC}{dl_{2,2}} \times \frac{dl_{2,2}}{dW_{1,1,2}} \\ \frac{dC}{dl_{2,2}} &= \sum_{k}^{n} \frac{dC}{dl_{3,k}} \times \frac{dl_{3,k}}{dl_{2,2}} \\ \frac{dC}{dl_{3,k}} &= \sum_{i}^{n} \frac{dC}{dl_{4,i}} \times \frac{dl_{4,i}}{dl_{3,k}}\end{aligned} \tag{2}$$

Note that we already computed $\{\frac{dC}{dl_{3,k}} | 0 \leq i \leq n\}$ in (1), so we can reuse them in (2). Let assume that $\frac{dC}{dl_{3,k}}$ costs $\theta(n)$ due to the n iterations of the summation $\sum_{i}^{n} \frac{dC}{dl_{4,i}} \times \frac{dl_{4,i}}{dl_{3,k}}$. Then by reusing the derivatives $\{\frac{dC}{dl_{3,k}} | 0 \leq i \leq n\}$ already computed in (1), we save $\theta(n \times n) = \theta(n^2)$ units of time in the calculation of $\frac{dC}{dW_{1,1,2}}$.

We are also able to reuse $\{\frac{dC}{dl_{3,k}} | 0 \leq i \leq n\}$ in the calculations of $\frac{dC}{dW_{1,1,3}}$ and $\frac{dC}{dW_{1,1,4}}$.

$$\begin{aligned}\frac{dC}{dW_{1,1,3}} &= \frac{dC}{dl_{2,3}} \times \frac{dl_{2,3}}{dW_{1,1,3}} \\ \frac{dC}{dl_{2,3}} &= \sum_{k}^{n} \frac{dC}{dl_{3,k}} \times \frac{dl_{3,k}}{dl_{2,3}} \\ \frac{dC}{dl_{3,k}} &= \sum_{i}^{n} \frac{dC}{dl_{4,i}} \times \frac{dl_{4,i}}{dl_{3,k}}\end{aligned} \tag{3}$$

$$\begin{aligned}\frac{dC}{dW_{1,1,4}} &= \frac{dC}{dl_{2,4}} \times \frac{dl_{2,4}}{dW_{1,1,4}} \\ \frac{dC}{dl_{2,4}} &= \sum_{k}^{n} \frac{dC}{dl_{3,k}} \times \frac{dl_{3,k}}{dl_{2,4}} \\ \frac{dC}{dl_{3,k}} &= \sum_{i}^{n} \frac{dC}{dl_{4,i}} \times \frac{dl_{4,i}}{dl_{3,k}}\end{aligned} \tag{4}$$

At each of the computations of $\frac{dC}{dW_{1,1,2}}$, $\frac{dC}{dW_{1,1,3}}$, $\frac{dC}{dW_{1,1,4}}$ we saved $\theta(n^2)$ units of time, so overall we saved $3 \times \theta(n^2) = (n-1) \times \theta(n^2) = \theta(n^3)$ units of time.

2. Weights connect $l_{1,2}$ in layer 1 to all nodes in layer2.

$$\frac{dC}{dW_{1,2,1}} = \frac{dC}{dl_{2,1}} \times \frac{dl_{2,1}}{dW_{1,2,1}} \tag{5}$$

$$\frac{dC}{dW_{1,2,2}} = \frac{dC}{dl_{2,2}} \times \frac{dl_{2,2}}{dW_{1,2,2}} \tag{6}$$

$$\frac{dC}{dW_{1,2,3}} = \frac{dC}{dl_{2,3}} \times \frac{dl_{2,3}}{dW_{1,2,3}} \tag{7}$$

$$\frac{dC}{dW_{1,2,4}} = \frac{dC}{dl_{2,4}} \times \frac{dl_{2,4}}{dW_{1,2,4}} \tag{8}$$

$\frac{dC}{dl_{2,1}}$, $\frac{dC}{dl_{2,2}}$, $\frac{dC}{dl_{2,3}}$ and $\frac{dC}{dl_{2,4}}$ were computed in the first part, so we can basically reuse the answer and save another $\theta(n^3)$.

3. Weights connect $l_{1,3}$ in layer 1 to all nodes in layer2.

$$\frac{dC}{dW_{1,3,1}} = \frac{dC}{dl_{2,1}} \times \frac{dl_{2,1}}{dW_{1,3,1}} \tag{9}$$

$$\frac{dC}{dW_{1,3,2}} = \frac{dC}{dl_{2,2}} \times \frac{dl_{2,2}}{dW_{1,3,2}} \tag{10}$$

$$\frac{dC}{dW_{1,3,3}} = \frac{dC}{dl_{2,3}} \times \frac{dl_{2,3}}{dW_{1,3,3}} \tag{11}$$

$$\frac{dC}{dW_{1,3,4}} = \frac{dC}{dl_{2,4}} \times \frac{dl_{2,4}}{dW_{1,3,4}} \tag{12}$$

We continue reuse $\frac{dC}{dl_{2,1}}$, $\frac{dC}{dl_{2,2}}$, $\frac{dC}{dl_{2,3}}$ and $\frac{dC}{dl_{2,4}}$ to save $\theta(n^3)$ units of time.

4. Weights connect $l_{1,4}$ in layer 1 to all nodes in layer2.

$$\frac{dC}{dW_{1,4,1}} = \frac{dC}{dl_{2,1}} \times \frac{dl_{2,1}}{dW_{1,4,1}} \tag{13}$$

$$\frac{dC}{dW_{1,4,2}} = \frac{dC}{dl_{2,2}} \times \frac{dl_{2,2}}{dW_{1,4,2}} \tag{14}$$

$$\frac{dC}{dW_{1,4,3}} = \frac{dC}{dl_{2,3}} \times \frac{dl_{2,3}}{dW_{1,4,3}} \tag{15}$$

$$\frac{dC}{dW_{1,4,4}} = \frac{dC}{dl_{2,4}} \times \frac{dl_{2,4}}{dW_{1,4,4}} \tag{16}$$

Again, we are able reuse $\frac{dC}{dl_{2,1}}$, $\frac{dC}{dl_{2,2}}$, $\frac{dC}{dl_{2,3}}$ and $\frac{dC}{dl_{2,4}}$ from the first stage, so $\theta(n^3)$ more units of time can be saved.

If we sum the results, we get $TotalTime_{save} = \theta(n^3) + \theta(n^3) + \theta(n^3) + \theta(n^3) = 4\theta(n^3) = n\theta(n^3) = \theta(n^4) = \theta(n^n)$ units of time.

In general, if we have an $n \times n$ neural network, and we use back-propagation that "remember" the previous computed values, we will save $\theta(n^n)$ units of time.

Note 1: one of the key ideas in my explaination is that you can think of the summation $\sum$ as the for loop in the program, and the runtime of the summation is the number of iterations it makes.
Note 2: there are several interpretations of this question. I used the interpretation that has been confirmed by the professor at Piazza468. Below is the screenshot of the professor reply [1st answer].

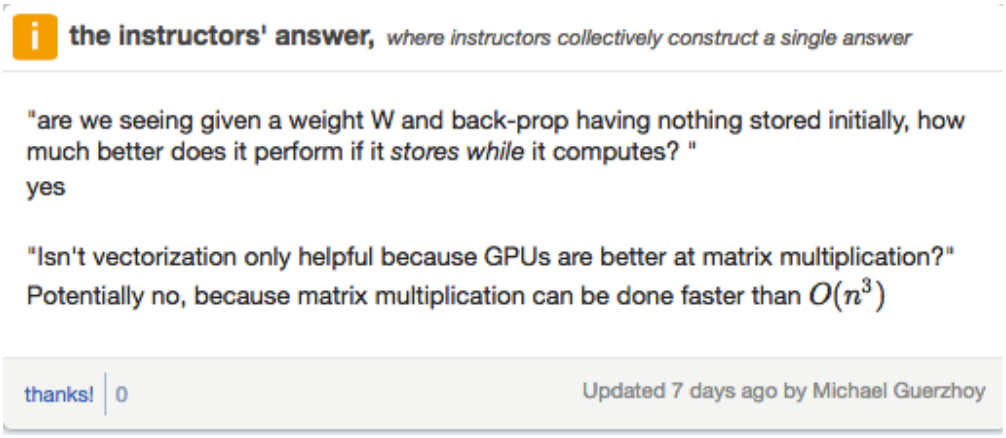


Figure 4: Professor reply

Part 7

Preprocessed the inputs: I used the scipts *get_data_and_crop.py* to download the images of the 6 actors from the 2 tables of image URLs inside *facescrub_actors.txt* and *facescrub_actresses.txt*. For every image, the script computed its sha256 checksum and compared with the sha256 value given in the tables. Images failed the sha256 checksum were removed. *get_data_and_crop.py* put the downloaded images in the directory *uncropped* and the cropped images in the directory *cropped*. The cropped images are all grayscale of size $32x32$ pixels. All of the corrupted, bad images should had been removed vy sha256 checksum, but just to be sure I went over the images one more times, and removed few images that i thought were not good for classification purpose. Below are examples of images that I removed.

(a) Original images

(b) Cropped images

Figure 5: Image with incorrect bounding box

(a) Original images

(b) Cropped images

Figure 6: Image shows only one side of the face

(a) Original images

(b) Cropped images

Figure 7: Images show only one side of the face

Picking training, validation and test set:
I used the function *def pickrandom()* in the script *auto_pick.py* to randomly pick 60 images for training, 30 images for validating and 30 images for testing. The three sets did not have any images in common. The script put the training images into *part7_data/train_data*, validating images into *part7_data/valid_data* and all testing images into *part7_data/test_data*.

Architecture of the network:
It is a single-hidden-layer fully connected neuron network. There are 30 neurons in the hidden layer. The number of neuron in the input layer is corresponding to the number of pixels in an image which is $32 * 32 = 1024$ neurons. There are 6 possible outputs corresponding to the 6 actors, so the output layer has 6 neurons.

Weight initialization:
To run gradient descent, I need to initialize the weights for the edges that connect the input layer to the hidden layer as well as the edges that connect the hidden layer to the output layer. The weights were randomly generated usning normal distribution centralized at 0 with a small standard deviation of 0.01. Below are the codes used for initializing the weights.

```
W0 = tf.Variable(tf.random_normal([1024, nhid], stddev=0.01))
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))
```

Activation function: I used the *tanh* activation function for the network. I also tried with *Relu* and *Sigmoid*, but none gave better performance than *tanh* after 500 iterations of gradient descent. Below is the code illustration of how the activation can be applied.

```
layer1 = tf.nn.tanh(tf.matmul(x, W0)+b0)
# layer1 = tf.nn.sigmoid(tf.matmul(x, W0)+b0)
# layer1 = tf.nn.relu(tf.matmul(x, W0)+b0)
layer2 = tf.matmul(layer1, W1)+b1
```

References: https://www.tensorflow.org/api_guides/python/nn

Learning curves with 30 hidden units

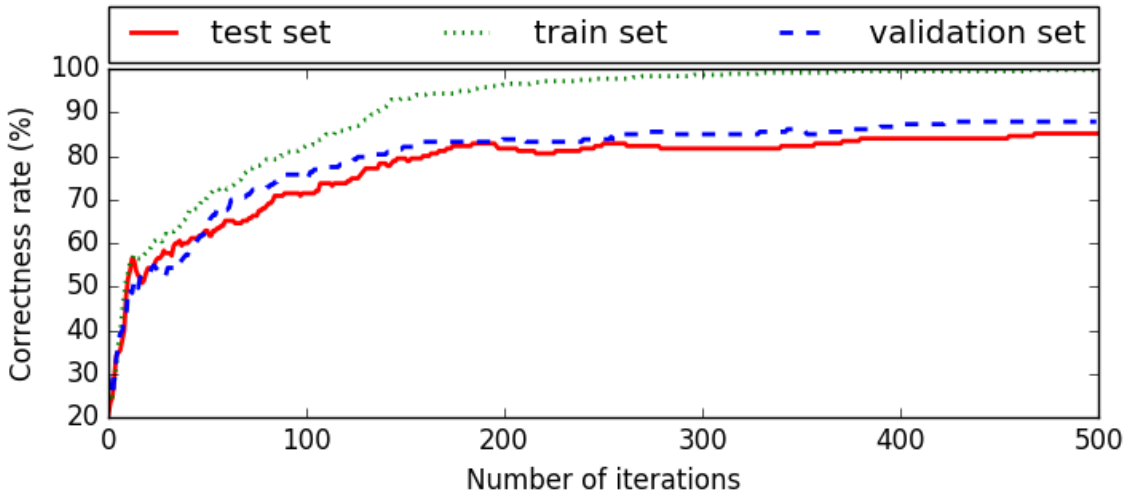


Figure 8: The performance on the training set come very closed to 100% after 500 iterations of gradient descent. The correctness rates on the validation and testing set are peaked at 87%, this is the highest value that I could get after trying to vary the number of hidden units as well as using different activation functions. Any number of hidden units greater than 30 would not give better performance on the validation and testing set than this configuration.

Note: I recognized that there was a bit of overfitting since the performance on the traing set was 10 % higher than the validation and test set, but there were no other configurations that could give me better testing and validating set performance than this one.

Learning curves with only 5 hidden units

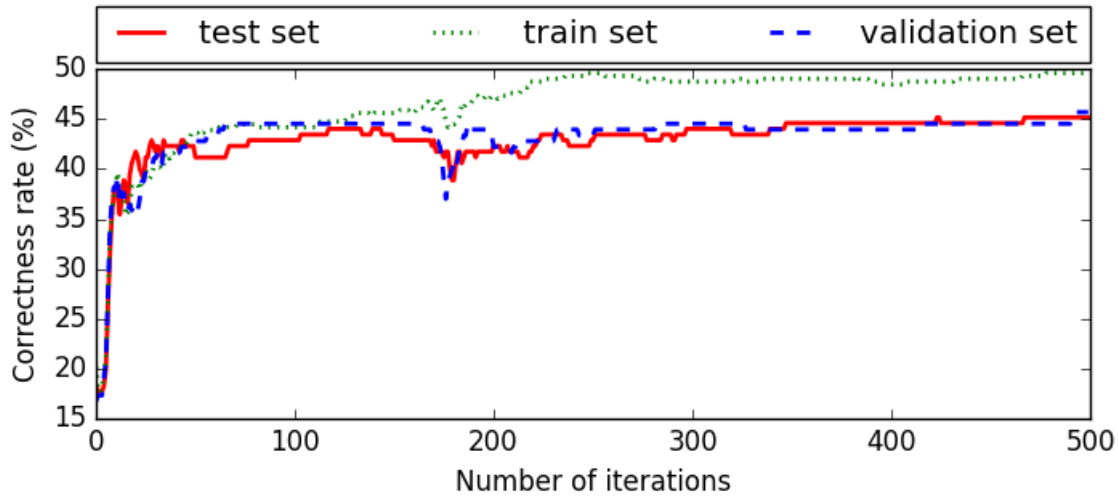


Figure 9: After 500 iterations, a neural network with only 5 hidden units only gave the correct rates no better than 50 %. This is to illustrate that the number of hidden units in a network could influence the performance. In this case, the low number of hidden units caused underfitting.

Part 8

Codes: The codes for this part are in *faces.py* and *get_data_and_crop.py*.
Prepare Data: Like in part 7, please use the script *get_data_and_crop.py* to download and crop the faces of the 6 actors.
Overfitting Scenario: To create an overfitting scenario for which I can demonstrate the use of weight decay, I do the followings:

- I have a small training set that consists only 48 images, 8 for each of the actors. The size of validation and test set are both $40 \times 6 = 280$, that is much larger than the training set.
- I use a single hidden layer network to do the classification. The hidden layer has 2000 units, which is almost twice the number of features $32 \times 32 = 1024$ (i.e. the pixels) in an image.
- Recalling from class that small training test size and large number of hidden units are common reasons for overfitting.

Strategy for testing:
The function *part8_with_no_regularization()* is used compute the performance on the training, validation and test set without the use of weight decay. The function *part8_with_regularization()* is used to compute the performance with the same sets, but with the use weight decay.
Regularization codes:

```
lam = 0.8
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
```

With the λ for weight decay set to 0.8, the performances on both validation and test set are improved. I chose the lamda based on how much improvement was made on the validation set with the use of weight decay.
Comparision results: Below are the results obtained from different runs of *part8_with_no_regularization()* and *part8_with_regularization()*. Even though the improvement is only 2 to 6 %, it is significant because we have a fairly large validation and testing sets.

```
## ========= RUN 1 ========= ##
-----With no regularization------
('Performance on the test set: ', 0.80000001)
('Performance on the train set: ', 1.0)
('Performance on the validation set:', 0.82080925)

-----With-regularization------
('Performance on the test set: ', 0.81142855)
('Performance on the train set: ', 0.99430197)
('Performance on the validation set:', 0.84393066)

## ========= RUN 2 ========= ##
-----With no regularization------
('Performance on the test set: ', 0.81714284)
('Performance on the train set: ', 1.0)
('Performance on the validation set:', 0.79768789)

-----With-regularization------
('Performance on the test set: ', 0.81714284)
('Performance on the train set: ', 0.99715102)
```

```
    ('Performance on the validation set:', 0.82080925)

    ## ========= RUN 3 ========= ##
    -----With no regularization------
    ('Performance on the test set: ', 0.81142855)
    ('Performance on the train set: ', 0.99715102)
    ('Performance on the validation set:', 0.79768789)

    -----With-regularization------
    ('Performance on the test set: ', 0.81714284)
    ('Performance on the train set: ', 0.99430197)
    ('Performance on the validation set:', 0.83236992)
```

Part 9

Codes: The codes for this part are in *faces.py*
Selecting hidden units: From part 7, we have a single hidden layer network with W0 and W1. W0 is a weight matrix vector of size 1024×40, in which 1024 is the number of pixels of an image and 40 is the number of hidden units. W0 connects the input layers to the hidden layer. W1 is a weight matrix of size 40×6, in which 40 is the number of hiddens units and 6 is the number of possible labels.

Step 1: I first find the 6 sensitive neurons, by looking at the 40×6 W1 weight matrix. The sensitive neurons are those that have high outgoing weight components.

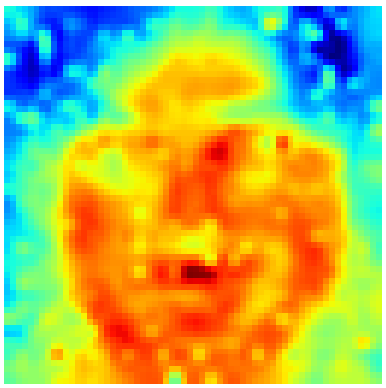
```
#  40 * 6
W1 = sess.run(W1, feed_dict={x: xs, y_: ys})
# 6 * 40
W1 = W1.T
# Step 1: Find 6 hiddens unit that are most sensitive.
unit = zeros(6)
unit[0] = argmax(W1[0])
unit[1] = argmax(W1[1])
unit[2] = argmax(W1[2])
unit[3] = argmax(W1[3])
unit[4] = argmax(W1[4])
unit[5] = argmax(W1[5])
```

Step 2: I then extract the incoming weights to the sensitive neurons from W0, and dislay only these weights.

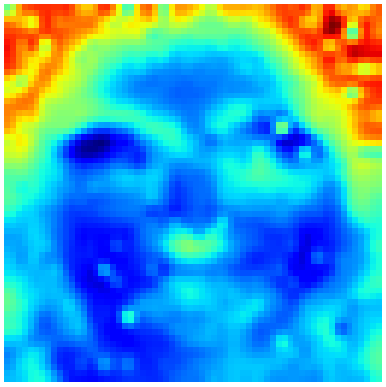
```
# 1024 * nhid = 1024 * 40
W0 = sess.run(W0, feed_dict={x: xs, y_: ys})
# 40 * 1024
W0 = W0.T
# Step 2: Extract 6 weights come into the 6 sensitive neurons.
img0 = imresize((W0[unit[0]]).reshape(32, 32), (64, 64))
img1 = imresize((W0[unit[1]]).reshape(32, 32), (64, 64))
img2 = imresize((W0[unit[2]]).reshape(32, 32), (64, 64))
img3 = imresize((W0[unit[3]]).reshape(32, 32), (64, 64))
img4 = imresize((W0[unit[4]]).reshape(32, 32), (64, 64))
img5 = imresize((W0[unit[5]]).reshape(32, 32), (64, 64))

 # Save images to local folder.
imsave("part9_img0.png", img0)
imsave("part9_img1.png", img1)
imsave("part9_img2.png", img2)
imsave("part9_img3.png", img3)
imsave("part9_img4.png", img4)
imsave("part9_img5.png", img5)
```

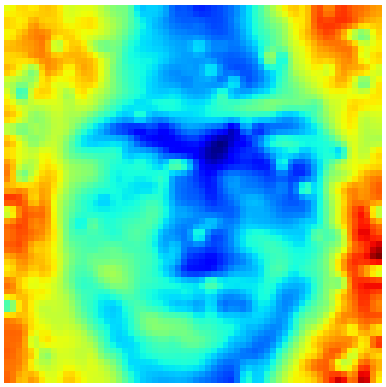
This strategy was confirmed by the Professor during the office hours as well as on Piazza @472.
For better resolution, I resize the images from 32×32 to 64×64 before saving the image to local directory. Out of the 6 images, I choose the best 3 to include in this report. Below are the images:



(a)



(b)



(c)

Figure 10: To make the images look smoother, I also used weight decay with $\lambda = 10$.

Part 10

Codes: The codes for this part is in *get_data_and_crop_part10.py* and *deepfaces.py*.

Getting the data:
The script *get_data_and_crop_part10* was used to download the images for this part. The cropped face images for this part have size 227×227, not 32×32 like in part 7 or A1.

Extract activations of AlexNet:
To extract the values of activations, I input the 227×227 cropped face image into AlexNet, and grap the $13 \times 13 \times 384$ matrix returned at *conv4*. The matrix can be thought as a set of features of a person image, and can be used for classification.

Doing the classification:
I build a single hidden layer network that is completely separated from the AlexNet. It takes in the extracted activations from *conv4* of the cropped face images to do the training.

Learning curves:

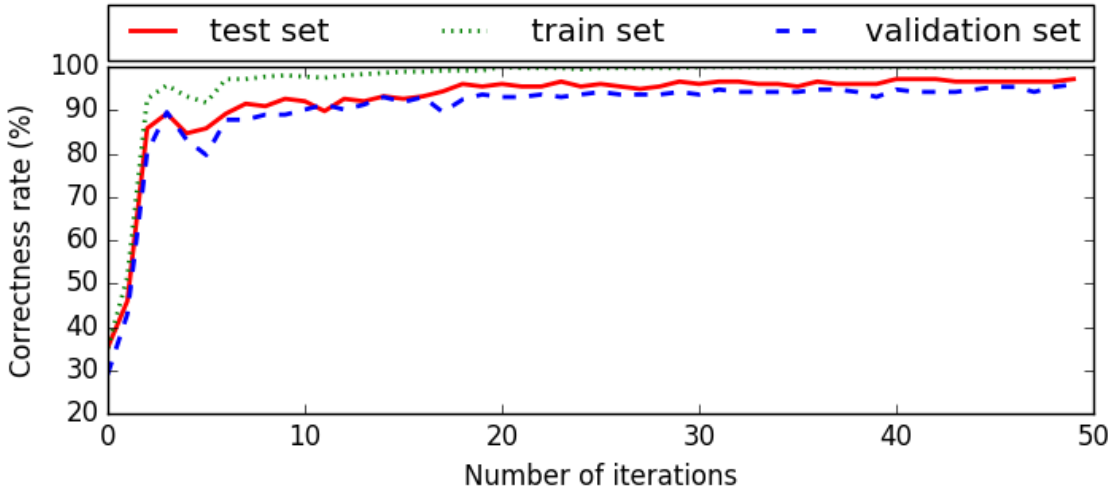


Figure 11: After just 50 iterations, the correctness rates on the training, validation and testing sets are all greater than 95%.

This is a big improvement on the performance when comparing to what we got in part 7. In part 7, we did 500 iterations, but if we only counted the first 50 iterations, none of the 3 sets has higher performance than 70%. In conclusion, studying from features of the images is better than studying the images directly.